Heuristic Algorithms and Ultra-weak Solutions to Size n Games of Linear Ataxx

Abstract

Solving a game played on a 7×7 board called Ataxx has recently come to niché interest in game theory due to its high complexity [1]. In this paper, we propose a unique abstraction of the game to a board of size $n \times 1$ which we call a size n game of linear Ataxx, or simply linear Ataxx if the game size is unspecified. Aside from the manipulation of board dimensions the games of Ataxx and linear Ataxx are played identically. In this paper we first explain how the game of linear Ataxx is played and introduce the nature of solving games. We then attempt to create computational methods which tend towards moves that are beneficial to the player in the game, though find, and explain the issues with performing this using typical methods. Finally, we generally solve the game ultra-weakly for any size n. It is our hope that these solutions can be furthered to produce a full ultra-weak solution to the game of Ataxx.

Contents

1 Introduction

In 1988 two game developers Dave Crummack and Craig Galley set about to create a game that logistically worked better on a computer, rather than a board, out of this they created the game Ataxx. Quickly, however, Ataxx fell into obscurity, though recently, a revival of the game came from a Chess AI forum discussing how they could solve it [2]. The forum found great difficulty in this, eventually spawning a variety of papers on the subject of Ataxx, though making no great progress in solving the game. To understand this game better however, we pose a linear reduction of the game. The hope is that this linearization will prove easier to solve, and thus yield some generalizable results to the larger game.

2 Linear Ataxx Instructions

Linear Ataxx is a competitive two player game which can be played on any board of size $n \times 1$, where $n \geq 2$. To set-up the game blue and red counters are placed on opposing ends of the board, as seen in Figure 1 on a board of size 7×1. Conventionally, the blue counter takes the left position of the board.

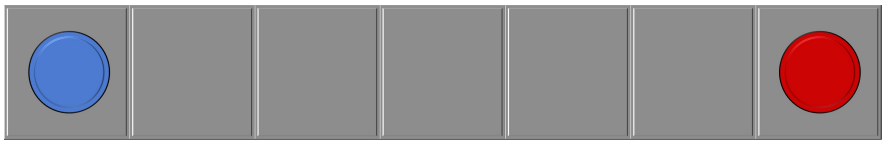


Figure 1: 7-space linear Ataxx: Starting Position

2.1 Gameplay

The game is played sequentially with the blue player moving first. In each turn a player can move one of their pieces by either jumping, or duplicating. If a player elects to jump one of their pieces, they simply select the piece they want to perform the action and move said piece two spaces forwards or two spaces backwards, hopping over any pieces in its path. An example of this is demonstrated by blue in Figure 2.

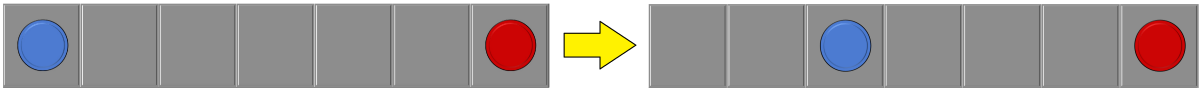


Figure 2: 7 space linear Ataxx: Blue Jumping Right

If a player elects to duplicate they select the piece they want to duplicate and create a new piece either directly in front or directly behind of the piece, as shown in Figure 3.

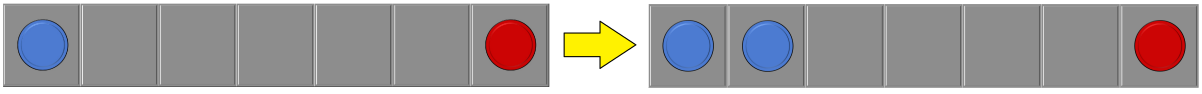


Figure 3: 7 space linear Ataxx: Blue Duplicating Right

After any move is played any enemy pieces next to the newly placed piece is converted to the movers colour. Examples of this conversion can be seen in Figure 4 and 5.

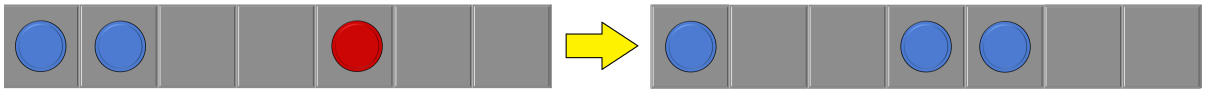


Figure 4: 7 space linear Ataxx game: Blue Jumping Right

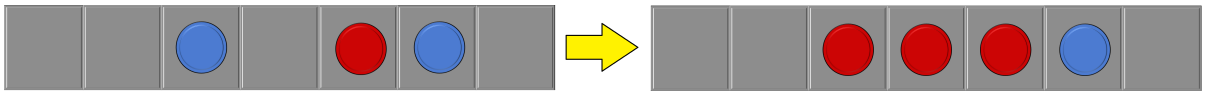


Figure 5: 7 space linear Ataxx game: Red Duplicating Left

During the game if there are moves available to a player, they must take one of them, that is to say, there is no passing in this game. If a player on their turn cannot play any move, all remaining spaces that are unoccupied are given to the opponent. It should be made clear however that this filling of unoccupied spaces does not convert any of the opponent's pieces that are adjacent to unoccupied spaces.

2.2 End Conditions

The game is won in two ways. The first method of victory is called an extermination victory. For a player to win by extermination, they must fully convert all the opponents' pieces to their own colour. The second, and more common method, is called a space-count victory. If the game reaches a state where all spaces on the board are filled, we count the number of tiles each player controls and the player with the most spaces wins. This does mean however that on boards that are of even size, we have the possibility that both players control an equal partition of the board, and in this case the game is declared a draw.

There is another often unmentioned game ending called deadlock. Deadlock occurs when both players continuously repeat the same set of moves in such a way that the same states of the game cycle indefinitely and no forward progress is made. This possibility for never ending games exists in many board games, with the primary example being Chess. Typically, this is addressed using the threefold repetition rule. This rule states that if the game board returns to the exact same position three times the game is ended and called a draw. In Ataxx however we use a variant of this rule. Here, when a game enters the exact same position for a third cycle the player who first initiated the repetition is considered to have played unbecomingly, an outcome considered even worse than a loss. For the player who did not start the cycle the game is just considered a draw. This is to stop players from abusing the repetition rule.

3 Applicable Theory

From reading the rules of linear Ataxx it is clear the game is a strictly competitive, sequential, two player game of finite size, n. We also see that it is a game of perfect information, as both players have all the information at all times. This means that linear Ataxx has a clear application of Zermelo's Theorem [3].

Theorem 3.1 (Zermelo's Theorem)**.** *Any sequential two-player game of finite size that has perfect information, and is strictly competitive, must end by either:*

1. *One of the players being able to force a winning strategy on the other,*

2. Or, both players being able to guarantee a draw.

It is a common question in game theory to discover what these winning and drawing strategies are for games which fall under Zermelo's Theorem. This question is often called solving the game, and itself under three categories, ultra-weak solving, weak solving, and strong solving [3] [4].

Definition 3.2 (Ultra-weakly Solved)**.** *From the initial position if both players play optimally the final state has been determined.*

Definition 3.3 (Weakly Solved)**.** *From the initial position, a strategy has been determined which obtains the best set of moves for both players under reasonable resources.*

Definition 3.4 (Strongly Solved)**.** *For all legal positions, a strategy has been determined which obtains the best possible play, for both players under reasonable resources.*

The reader may be wondering the what reasonable resources means in the context of weakly and strongly solving games. If the reader is not familiar with computational complexity classes [5], it is fine to take this to mean that the strategies could be determined if we used all the computers in the world for the next 10,000 years to solve it. For the reader who is more familiar with complexity classes, this is in reference to what computational complexity class the algorithm for determining the strategy runs in. Typically, one may assume Cobham's Thesis [6] for computational ease, however personally the author finds Cobham's Thesis an inferior definition to the analogous $\mathcal{BPP}$, or $\mathcal{BQP}$ definitions of computational ease [5]. There is also a strong argument to be made for using threshold definitions like those made by Diffie and Hellman, however generally these have their more philosophical issues [7].

The reader may ask now, *"Why is strongly or weakly solving a game not commonly done if we can use such a huge computational allowance?"*. The reason is that most of the time these games must be solved in a brute force approach. This means searching through every possible way the game is played to determine the best strategy. Doing this is an incredibly daunting task in many games, and to describe why we must introduce the notion of a game tree [3].

Definition 3.5 (Game Tree)**.** *A game tree is a tree where the nodes denote possible states of the game and edges extruding from each state are the possible moves from that state.*

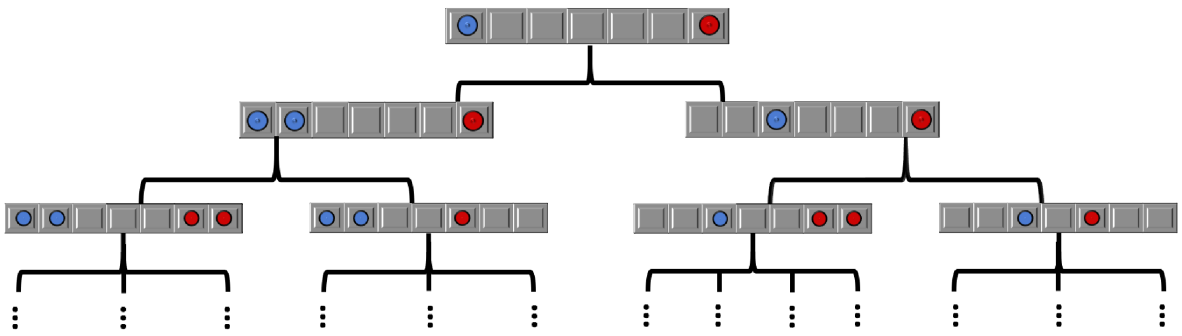


Figure 6: Game Tree of a size 7 game of linear Ataxx

The game tree is essentially all positions of the game in every possible order, and hence every single possible game. This makes it the abstract object that we must brute force search to determine strategies for strong and weak solutions. These game trees can be massive in many games, making

finding the strategies an incredibly difficult task even if were are using all the computers in the world for 10,000 years. To describe how massive these trees are we need to define measurements for it, one of which is the called the Game Complexity [3] [4].

Definition 3.6 (Game Complexity)**.** *The complexity of a game is the number of distinct games which can be played. This number is equivalent to the number of termination nodes in the game tree. A common estimate of a game's complexity is:*

$$\text{GTC} \approx b^d \tag{1}$$

Where b denotes the average number of edges from each game state, and d denotes the average length of the game.

With simplification, as a rule of thumb when a game has a large game complexity, it is difficult to strongly or weakly solve. To give an idea of what we mean by large complexity, the most complex game that we have weakly or strongly solved via brute force is Checkers [3]. Solving Checkers took 200 computers over 18 years to compute, cutting through the game's complexity of $\approx 10^{31}$ [8]. Chess is a game with a famously large game complexity approximated by the father of information theory, Claude Shannon, at $\approx 10^{120}$. Manipulating these values we can find that if we tried to solve Chess at the same rate at which we solved Checkers, it would take 3.5×10^{80} computers, approximately one for every atom in the observable universe [9], approximately a trillion years to compute a solution. Ataxx has an estimated game complexity of $\approx 10^{198}$ [1]. To find the strategy at the rate of Checkers would take the 3.5×10^{80} computers working for about 10^{100} years to solve it, just in time to celebrate by watching the last of the black holes evaporate [10]. Clearly, computing these are unreasonable, however, there is hope.

In these calculations we have assume two things, the first is the computers used to solve Checkers are as fast as the computers we'd use now. Though, while it is true that our current computers are more powerful, this is negligible as even an absurdly more efficient computer would merely be a drop in the ocean of computations required. One may propose the use of quantum however, even an absurdly powerful quantum computer running Grovner's search would only produce a quadratic speed up, on top of the absurd classical computer, which still is not enough [11].

The second and much more plausible form of hope comes from the assumption that game complexity directly determines the difficulty of strongly or weakly solving a game. As we said this is simply a rule of thumb, not some strict fact. To explain why, imagine that we have a two-player sequential game which is played on a board consisting of a Rayo's number of spaces, the largest named number. In this game if the starting player chooses any space, we say they instantly win. This is a boring game indeed and unfair to the second player, however, we have just constructed a game which falls under Zermello's Theorem and possesses a game complexity of larger size then any game ever made, yet it is clearly strongly solved by choosing any move.

Now let us imagine another game. This game is another sequential two player game, but is played on a board of two spaces. Here first player must choose the first space. The second player however has options. They can either immediately win by choosing the other space or move the game to a game of Chess. To weakly solve this game, we do not have to solve Chess, as clearly the second player has an unstoppable winning strategy by choosing to win in their first move.

When we are brute forcing a method, these splits where we do not have to calculate very complicated sections often occur, however they are not always so clear. To show this, imagine for example in the second game instead of winning instantly by selecting the other space, choosing this move switches us to a variant of Chess. In this variant, the second player gets eight queens

instead of pawns. Choosing this variation of chess is clearly advantageous to the second player, but how do we know this.

We can deduce this from the use of heuristics [3]. This quantifies the qualities of the game, allowing us to deduce which states are advantageous to us and which are not. We can use these heuristics on any sub-section of the game tree putting ourselves in the best position that can achieve within a certain number of moves, which we call the depth. In essence, we are breaking the game into sub-games of a specific size and playing perfectly on these games. This greatly reduces the total number of games that we must play, as we don't have to calculate on all branches [3]. This however only point us towards positions that are favorable to us, it won't guarantee a perfect solution. In the subsequent sections we develop a heuristic algorithm which we use to try and point us towards strong solutions strategies of linear Ataxx, however first we must make a method to test this algorithm.

4 Baseline

To test how well our algorithms perform we create a basic player called, Bogo. The name of the algorithm is in reference to the inefficient sorting algorithm called Bogosort, which attempts to sort items by randomizing the whole set. Our player of Bogo performs similarly when playing linear Ataxx as at any point in a game it chooses its moves completely at random. Bogo, will act as our baseline player for our algorithms as when it runs in large batches against different algorithms it will represent all different types of games, and what we want to see is how well our later algorithms react to any moves not just the optimal ones.

We begin by having Bogo play itself 1000 times for each game of linear Ataxx of size 3 to 14, shown in Table 1. From doing this we can see that in smaller games there is a clear bias towards certain players, that averages out at larger games.

	3	4	5	6	7	8	9	10	11	12	13	14
Blue Wins	1000	515	255	482	613	444	528	435	519	441	512	424
Red Wins	0	485	745	513	387	410	472	435	481	433	488	407
Draw	0	0	0	5	0	146	0	130	0	126	0	169

Table 1: Bogo vs Bogo Player on Different Sizes of linear Ataxx

5 Computational Method A

The first method that we explore for playing linear Ataxx is one where we choose moves that maximize the sum, of the difference between the number of pieces possessed by the player and the opponent, for each possible subsequent game that can come from that choice. That is to say, if we have n possible immediate actions, indexed b_i, we decide which action is the best by,

$$Max\{f(b_1), f(b_2), f(b_3), \ldots, f(b_n)\} \text{ such that } f(b_i) = \sum_{j=0}^{|b_i|}(p_{i,j} - q_{i,j}). \tag{2}$$

Where $p_{i,j}$ is the player piece count for the j^{th} sub-branch of choice i, and $q_{i,j}$ is the opponents piece count for the j^{th} sub-branch of choice i. The idea behind this heuristic is that the end goal

of the game is to end the game in a position where you have more pieces than your opponent, thus by choosing actions that maximize the difference between each others pieces count we will hopefully result favorable endings.

The major issue with the algorithm however is that for even on small depth searches this computation can become very expensive to compute. Though, we can speed this up immensely by before applying this large computation looking at the subset of the game tree to see if we can force ourselves a victory, or if we should avoid nodes that would lead to an immediate loss. This is a simplification on the idea behind Alpha-Beta pruning. The implementation of this massively speeds up the computation times and is the idea applied in method B.

6 Computational Method B

Method B, as touched on in the previous section, takes the subset of the game tree that is within a certain amount of moves and checks if there is a method to force a win for itself through backwards induction. If there is no way to directly force a win, it checks if there is a method for the opponent to force a win any of the possible moves and avoids choosing them if they exist. Finally, with these checks passed the algorithm proceeds as Algorithm A had. When we comparing this Algorithm against Bogo for different depth counts we find that there is a definite improvement for both the blue player, Table 2, and the red player, Table 3.

		3	4	5	6	7	8	9	10	11	12	13	14
Depth of 1	Blue Wins	1000	1000	500	1000	1000	892	834	865	759	671	662	553
	Red Wins	0	0	500	0	0	77	166	100	241	245	338	319
	Draw	0	0	0	0	0	31	0	35	0	84	0	128
Depth of 3	Blue Wins	1000	1000	436	1000	1000	861	1000	775	779	700	807	642
	Red Wins	0	0	564	0	0	0	0	62	221	167	193	226
	Draw	0	0	0	0	0	139	0	163	0	133	0	132

Table 2: Blue as Algorithm B, Red as Bogo

		3	4	5	6	7	8	9	10	11	12	13	14
Depth of 1	Blue Wins	1000	473	0	517	360	155	101	71	108	71	48	31
	Red Wins	0	527	1000	483	640	845	899	912	892	893	952	906
	Draw	0	0	0	0	0	0	0	17	0	36	0	63
Depth of 3	Blue Wins	1000	514	0	546	380	0	304	120	144	66	108	56
	Red Wins	0	486	1000	456	620	853	696	852	856	891	892	868
	Draw	0	0	0	0	0	145	0	28	0	43	0	76

Table 3: Red as Algorithm B, Blue as Bogo

The most important note we gather is that method B's performance compared to the baseline in all cases is either better or at least on par with Bogo[1]. This means that in many cases the algorithm is generally making positive moves. Though, we also notice that this algorithm is not perfect and as we tend to larger games the algorithm begins to not play the game as well as it was, this is most noticeable in Table 2.

[1] Allowing for variance in both methods.

The reason for this is that as the game increases in size, the depth required to observe interaction between players increases. This means the algorithm will only perform duplication in some direction until it can interact with the opponent. In some cases, it can still win regardless of this, however this may be a bad strategy and over time is adversely affecting the quality of play given by the algorithm. This could be improved by implementing a variable depth allowing it to see the node, however as we increase the depth on larger boards the time taken in computations quickly explode making this an inefficient strategy. What may be a stronger implementation is to have a drop out in the algorithm saying if no opposition is to be encountered within any of the branches observed, to perform a jump towards them instead of a duplication. This, however, may be met with its own issues and would require further analysis to determine the optimal implementation.

7 An Ultra-weak Solution

We now diverge to show an ultra-weak solution to any size game of linear Ataxx. We claim that if both players play optimally there will be a draw if the board is size $6k+2$ for integer k. The second to move player will win on all boards size $6k+5$ for integer k, and the player who moves first will win on all other board sizes.

Proof. To prove this, we develop the following notation. First, we develop the sets of $Player = \{Blue, Red\}$ and $Outcome = \{BlueWin, RedWin, Draw\}$. With this we create the following function:

$$\mathcal{PP}(a, n) = b, \quad \text{such that } a \in Player, \text{ and } b \in Outcome \tag{3}$$

This function denotes that the outcome of a perfectly played game of size n, where the inputted element of $Player$, a, begins the game. It is trivial to see that from symmetry we have the following properties:

$$\{\mathcal{PP}(Blue, n) = BlueWin\} \Leftrightarrow \{\mathcal{PP}(Red, n) = RedWin\} \tag{4}$$

$$\{\mathcal{PP}(Blue, n) = RedWin\} \Leftrightarrow \{\mathcal{PP}(Red, n) = BlueWin\} \tag{5}$$

$$\{\mathcal{PP}(Blue, n) = Draw\} \Leftrightarrow \{\mathcal{PP}(Red, n) = Draw\} \tag{6}$$

With this we now propose the following,

Proposition 7.1.
If $\mathcal{PP}(Blue, n) = RedWin$, then both $\mathcal{PP}(Blue, n+1) = BlueWin$ and $\mathcal{PP}(Blue, n+2) = BlueWin$.

To prove this proposition we use the fact that on the game of size $n+1$ where blue starts, they simply duplicate their starting piece, moving the game to the form where blue has an extra piece and there is a sub-game of size n where red must begin. From (5) we have that blue will win this game thus $\mathcal{PP}(Blue, n+1) = BlueWin$. We can invoke a similar strategy for the game were blue starts on $n+2$, but with a jump, again forcing the blue win.

We now propose the following,

Proposition 7.2.
If $\mathcal{PP}(Blue, n) = Draw$, then both $\mathcal{PP}(Blue, n+1) = BlueWin$ and $\mathcal{PP}(Blue, n+2) = BlueWin$.

To prove this we use the fact that at the start of the game of $\mathcal{PP}(Blue, n+1)$, blue can duplicate giving themselves an extra piece and moving the game to the sub-game of $\mathcal{PP}(Red, n)$. We know that from (6) this will end as a draw, however as draws can only happen from ending with an equal number of pieces and blue now has an extra piece this implies $\mathcal{PP}(Blue, n+1) = BlueWin$. We can invoke a similar strategy with the jump on the $\mathcal{PP}(Blue, n+2)$, as once we collect these two squares at the end this implies again by similar logic that $\mathcal{PP}(Blue, n+2) = BlueWin$.

This brings us to the final proposition,

Proposition 7.3.
If the board size, n, is odd, $\mathcal{PP}(Blue, n-1) = BlueWin$, $\mathcal{PP}(Blue, n-2) = BlueWin$, and $\mathcal{PP}(Blue, n-3k) \neq BlueWin$ for any integer k such that $0 < 3k < n-2$, then $\mathcal{PP}(Blue, n) = RedWin$.

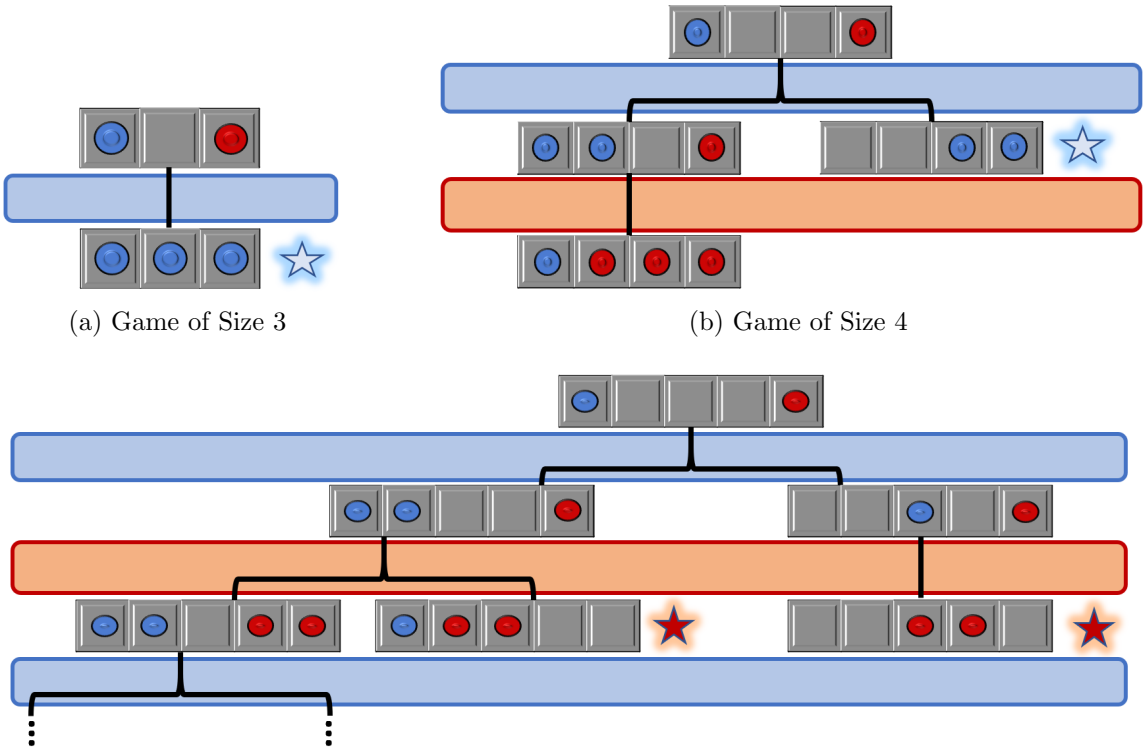
Further, if n is even, $\mathcal{PP}(Blue, n-1) = BlueWin$, $\mathcal{PP}(Blue, n-2) = BlueWin$, and $\mathcal{PP}(Blue, n-3k) \neq BlueWin$ for any integer k such that $0 < 3k < n-2$, then $\mathcal{PP}(Blue, n) = Draw$.

To prove this is true let us assume it is false. Since blue begins let us suppose their starting move is a duplication. Here they get an extra piece but find themselves in the sub-game of $\mathcal{PP}(Red, n-1)$ which from (4) and our assumptions is a $RedWin$. Since at most a win is a difference of one piece, which blue may have matched in their duplication, the best the blue may have achieved is a draw, but never a win and only on odd sized boards otherwise they still will have lost. This implies that blue must begin with a jump.

When blue begins with a jumps they gain two spaces behind them and move the game to a $\mathcal{PP}(Red, n-2)$. Here it will be most players initial thought for red to duplicate and move to one where blue has two spaces behind, red has an extra piece, and we have the sub-game of $\mathcal{PP}(B, n-3)$, which we know red wins. This, however, is a losing move as by jumping blue has allowed themselves to effectively pass their turn by duplicating backwards, moving the game to one where blue has an extra space behind still and $\mathcal{PP}(Red, n-3)$ is the sub-game, a winning state for blue. Instead for red to play optimally they will match blue by jumping. This will move the game to a state of $\mathcal{PP}(Blue, n-5)$, where red has as many backwards duplication passing moves as blue. From now on any move time that Blue attempts to switch the board Red will switch it back, and as blue must act first they can always be mirrored by red forcing blue to continue jumping or duplicate. This will move eventually to the point where the opponents pieces are five spaces apart, and blue, eventually will have to move first. When consulting the game tree, Figure 7c, for the game of size 5 where Blue starts, we see blue will lose the sub-game all scenarios. It is crucial to note however that if they duplicate to the sub game of size 5, they can also lose the sub-game by jumping and force a draw with their remaining piece. This however only works on an even sized board as if it is odd the players cannot draw and as red will have won by one piece, thus proving the original proposition.

When we generate the first 3 game trees, Figure 7, for linear Ataxx, we can see that $\mathcal{PP}(Blue, 3) = BlueWin$, $\mathcal{PP}(Blue, 4) = BlueWin$, and $\mathcal{PP}(Blue, 5) = RedWin$. From this, and the proposi-

tions we have developed, we now can generate the ultra-weak solution to all games given in Table 4.



(a) Game of Size 3

(b) Game of Size 4

(c) Game of Size 5

Figure 7: Linear Ataxx Game Trees of size 3,4, and 5.

n	3	4	5	6	7	8	9	10	11	12	13	14	...
Outcome	Blue Win	Blue Win	Red Win	Blue Win	Blue Win	Draw	Blue Win	Blue Win	Red Win	Blue Win	Blue Win	Draw	...

Table 4: The Ultra-weak solutions to various game sizes of linear Ataxx, assuming Blue starts

□

8 Conclusion

In this paper we attempted to create computational algorithms which tend towards the strongest move that can be made on a given depth, and thus hopefully tending towards the true perfect solutions. Unfortunately, we found these algorithms in larger sized games, require a depth which causes them to exceed their usefulness. Afterwards however we presented the ultra-weak solution to the game of general linear Ataxx. We believe with more work this can be expanded to create a weak solution to all games of linear Ataxx. This provides a promising direction in finding either a weak or ultra-weak solution to the game of Ataxx using the linear patterns, as a break down for the full game.

References

[1] Ribeiro, Leonardo F. R., and Daniel R. Figueiredo. *"Performance of Monte Carlo Tree Search Algorithms When Playing the Game Ataxx."* Anais Do XV Encontro Nacional De Inteligência Artificial e Computacional (ENIAC 2018), 2018, doi:10.5753/eniac.2018.4423.

[2] *"TalkChess.com."* Ataxx - TalkChess.com, www.talkchess.com/forum3/viewtopic.php?t=72089.

[3] Allis, Victor. *"Searching for Solutions in Games and Artificial Intelligence."* Proefschrift Rijksuniversiteit Limburg, Maastricht, 1994.

[4] Herik, H.jaap Van Den, et al. *"Games Solved: Now and in the Future."* Artificial Intelligence, vol. 134, no. 1-2, 2002, pp. 277–311., doi:10.1016/s0004-3702(01)00152-7.

[5] Stephen A. Cook. *"An overview of computational complexity"*. Commun. ACM 26, 6 (June 1983), 400–408. DOI:https://doi.org/10.1145/358141.358144

[6] A. Cobham. *"The intrinsic computational difficulty of functions"*. I.B.M. Research Center, Yorktown Heights, N.Y., USA.

[7] W. Diffie and M. Hellman. *New directions in cryptography.* IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644-654, November 1976.

[8] Schaeffer, Jonathan, et al. *"Project Chinook."* Webdocs, University of Alberta, 2007, webdocs.cs.ualberta.ca/ chinook/project/.

[9] Villanueva, John Carl. *"How Many Atoms Are There in the Universe?"* Universe Today, 22 Apr. 2018, www.universetoday.com/36302/atoms-in-the-universe/.

[10] Cain, Fraser. *"The End of Everything."* Universe Today, 26 Dec. 2015, www.universetoday.com/11430/the-end-of-everything/.

[11] Tarrataca, Luís, and Andreas Wichert. *"Tree Search and Quantum Computation."* Quantum Information Processing, vol. 10, no. 4, 2010, pp. 475–500., doi:10.1007/s11128-010-0212-z.

Appendices

A LinearAtaax Backend Code

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 23 16:25:46 2020

@author: jonny
"""
import random as rnd, matplotlib.pyplot as plt
import numpy as np
import copy

class Board:
    def __init__(self,board_size=5):
        self.board_size = board_size
        if self.board_size > 2:
            self.board = [0 for _ in range(self.board_size + 4)]
            self.board[2] = 1
            self.board[-3] = 2
            self.board[0] = '|'
            self.board[1] = '|'
            self.board[-1] = '|'
            self.board[-2] = '|'
        else:
            raise ValueError("Board must be of integer size larger than 2")
        self.turn_count = 0
        self.current_player = 1
        self.done = False
        self.human_render = True
        self.addl_output = False

    def reset(self):
        self.__init__(self.board_size)

    def convert_adjacent_pieces(self,index,player_id):
        # print('b',self)
        for i in [-1,1]:
            # print(index+i,index,self[index+i])
            if self[index+i] not in [0,player_id]:  #adjacent space has the ...
                opposing player's piece
                 # print('c',player_id)
                 self[index+i] = player_id
```

```
1
2       def move(self,board_idx,move_type):
3           err = None
4           valid_piece = (self[board_idx] == self.current_player)
5           if (0 ≤ (board_idx + move_type) ≤ (len(self))) and valid_piece:
6               player_id = self[board_idx]
7               if self[board_idx + move_type] == 0:
8                   self[board_idx + move_type] = player_id
9                   self.convert_adjacent_pieces(board_idx+move_type,player_id)  ...
                        # convert adjacent piece to current players piece
10                  if abs(move_type) == 2:
11                      self[board_idx] = 0
12                  self.turn_count += 1
13              else:
14                  err = "error in move"
15                  if self.human_render:
16
17                      print("Error in Move")
18                  return self.board[2:-2], 0, self.done, err
19          else:
20              err = "illegal move"
21              if self.human_render:
22
23                  print("Illegal Move! Try Again")
24              return self.board[2:-2], 0, self.done, err
25          reward = self.end_state_eval(human=0)
26          state = self.board[2:-2]
27          self.current_player = self.turn_count % 2 + 1
28          return state, reward, self.done, err
29
30
31      def end_state_eval(self, human=1, output=1):
32          next_player = (self.turn_count % 2) + 1
33          checking_player = ((self.turn_count + 1) % 2) + 1
34          indexes_to_check = [idx for idx, val in enumerate(self) if val == ...
                next_player]
35          if not indexes_to_check:  # the next player has no remaining pieces
36              if self.human_render:
37                  print("Extermination was achieved by Player " + ...
                        str(checking_player))
38                  print("""  ----------------------------------------
39  |                                       |
40  |           Game Outcome:               |
41  |                                       |
42  |           board:                      |
43  |           {}
44  |---------------------------------------|""".format(self))
45              reward = 1
46              self.done = True
47              return reward
```

```
1
2         else:
3             for i in indexes_to_check:
4                 if 0 in self[(i - 2):(i + 3)]:
5                     if self.human_render and self.addl_output:
6                         print("Valid Move is available for player " + ...
                              str(next_player))
7                     reward = 0
8                     return reward
9
10            indexes_to_check = [idx for idx, val in enumerate(self) if val ...
                 == 0]
11            for i in indexes_to_check:
12                self[i] = checking_player
13            if self.human_render:
14                #print(self)
15                print("Player " + str(checking_player) + " Won by Space ...
                     Count\nThe final score was " + str(
16                    self.count(1)) + " v " + str(self.count(2)))
17                print("""  ----------------------------------------
18  |                                       |
19  |           Game outcome:               |
20  |                                       |
21  |           board:                      |
22  |           {}
23  |---------------------------------------|
24            """.format(self))
25            if self.count(checking_player) > self.count(next_player):
26                reward = 1
27            elif self.count(next_player) > self.count(checking_player):
28                reward = -1
29            else:  # tie
30                reward = 0
31            self.done = True
32            return reward
33
34    def count(self, val):
35        return self.board.count(val)
36
37    def get_state(self):
38        return self.board[2:-2]
39
40    def show_valid_moves(self):
41        moves = []
42        indexes_to_check = [idx for idx, val in enumerate(self) if val == ...
              self.current_player]
43        if not indexes_to_check:
44            return None
45        else:
46            for board_idx in indexes_to_check:
47                for move_type in [-2,-1,1,2]:
48                    if (0 <= (board_idx + move_type) <= (len(self))) and ...
                         self[board_idx + move_type] == 0:
49                        moves.append((board_idx,move_type))
50        return moves
```

```
1
2       def __len__(self):
3           return self.board_size
4
5       def __getitem__(self, item):
6           if isinstance(item, slice):
7               return self.board[item.start+2:item.stop+2]
8           return self.board[item+2]
9
10      def __setitem__(self, key, value):
11          if self.board[key+2] != '|':
12              self.board[key+2] = value
13
14      def __iter__(self):
15          return iter(self.board[2:-2])
16
17      def __str__(self):
18          return str(self.get_state())
19
20  #  ============================================================================
21
22
23  class Agent:
24      def __init__(self,player_no,game,algo=0,depth=10,debug=False):
25          if player_no not in [1,2]:
26              raise ValueError("player_no must be 1 or 2")
27          self.player_no = player_no
28          self.game = game
29          # self.game.human_render = False
30          if algo==1:
31              self.algorithm_choice = self.mostly_type_one
32          elif algo ==2:
33              self.algorithm_choice = self.minimax
34          elif algo==3:
35              self.algorithm_choice = self.minimax_improved
36          elif algo==4:
37              self.algorithm_choice = self.random_improved
38          else:
39              self.algorithm_choice = self.random_algo
40          self.debug = debug
41          self.depth = depth
```

```
    def play(self):
        old_state = self.game.get_state()
        if self.player_no != self.game.current_player or self.game.done:
            return
        else:
            print("""  ---------------------------------------
 |                                       |
 |           Player {}'s turn:            |
 |                                       |
 |           board currently:            |
 |           {}
 |---------------------------------------|
            """.format(self.player_no,self.game))
            x,y = self.algorithm_choice(self.game)
#            print("""
#             -------
#             move selected for player {}:
#              {} {}
#
#    {}
#        |
#        |
#        \\/""".format(self.player_no,x,y,self.game))

            state, _, _, err = self.game.move(x,y)

            if state != old_state:
                pass
                #print("         ",self.game)

    def random_algo(self,game):
        if self.debug:
            print("choices are",game.show_valid_moves())
        x,y = rnd.choice(game.show_valid_moves())
        return x,y

    def mostly_type_one(self,game):
        valid_places = [key for key, value in enumerate(self.game) if value ...
            == self.player_no]
        move_types = [-2, -1, 1, 2]
        move_prob = [.05,.45,.45,.05]
        x, y = rnd.choice(valid_places), ...
            int(np.random.choice(move_types,1,p=move_prob))
        return x, y
```

```
    # def minimax_old(self,game,layers=1):
    #     actions = game.show_valid_moves()
    #     valid_x_y = []
    #     print("vals to search: {}".format(actions))
    #     for x,y in actions:
    #         print("testing",x,y)
    #         test_board = copy.deepcopy(game)
    #         test_board.human_render = False
    #         curr_player = test_board.current_player
    #         state,reward,done, err = test_board.move(x,y)
    #         if reward > 0:
    #             print("this move wins for player {}!".format(curr_player),x,y)
    #             if curr_player == self.game.current_player:
    #                 return x,y
    #             else:
    #                 print("other player wins with this")
    #         elif reward < 0:
    #             print("this move wins for player {}!".format(curr_player), ...
        x, y)
    #             if curr_player != self.game.current_player:
    #                 return x,y
    #             else:
    #                 print("other player wins with this")
    #         elif reward == 0:
    #             if err is not None:
    #                 print(err)
    #             else:
    #                 valid_x_y.append((x,y))
    #
    #
    #     x, y = rnd.choice(valid_x_y)
    #     return x,y
    def random_improved(self,game):
        actions = game.show_valid_moves()
        non_terminating_moves = []
        for x, y in actions:
            test_board = copy.deepcopy(game)
            test_board.addl_output, test_board.human_render = False, False
            _, reward, _, err = test_board.move(x, y)
            if reward > 0:
                if self.debug:
                    print("""{}, {} is an immediate win""".format(x, y))
                return x, y
            if reward == 0:
                non_terminating_moves.append((x,y))
        if len(non_terminating_moves)==0:
            x, y = rnd.choice(game.show_valid_moves())
        else:
            x, y = rnd.choice(non_terminating_moves)
        return x,y
```

```
    def minimax_values(self,game,depth,layers=1):
        spacing = "       " * (depth + 1)
#         print("""{0}true game turn:{1}, depth of tree: {2}, testing game ...
    turn: {3}
# {0}real game current board: {4}
# {0}testing imagined board: {5}
#          ...
    """.format(spacing,self.game.turn_count,depth,game.turn_count,self.game,game))
        if depth > self.depth:
            if self.debug:
                print("too deep")
            return 0
        actions = game.show_valid_moves()
        if actions is None:
            return 0
        if self.debug:
            print("{}moves to find values for: {}".format(spacing, actions))
        value=0
        for x,y in actions:

            test_board = copy.deepcopy(game)
            test_board.addl_output = False
            test_board.human_render = False
            curr_player = test_board.current_player
            state,reward,done, err = test_board.move(x,y)
            if self.debug:
                print("""
{}testing move {},{}: {}-->{}""".format(spacing,x, y, game,test_board))
            if reward == 0:
                if err is not None:
                    if self.debug:
                        print(err)
                else:
                    if self.debug:
                        print("{}not a terminating node, begin ...
                            recursion:".format(spacing))
                    value += self.minimax_values(test_board,depth+1)
            elif curr_player == self.game.current_player:
                if self.debug:
                    print("{}---a terminating node, value ...
                        {}".format(spacing,reward))
                value += reward
            else:
                if self.debug:
                    print("{}---a terminating node, value ...
                        {}".format(spacing,-reward))
                value -= reward
        if self.debug:
            print("""
{}total value: {}
        """.format(spacing,value))
        return value
```

```
    def minimax(self,game):
        actions = game.show_valid_moves()
        if actions is None:
            #print("""no actions2""")
            return None

        #print("""moves to test: {}____""".format(actions))
        best_move = actions[0]
        best_value = -1000
        for x,y in actions:

            test_board = copy.deepcopy(game)
            test_board.addl_output = False
            test_board.human_render = False
            _,reward,_,err = test_board.move(x,y)
            if self.debug:
                print("""
testing move {},{}: {} --> {}
""".format(x,y,game,test_board))
            value = 0
            if reward == 0:
                if err is not None:
                    print(err)
                else:
                    if self.debug:
                        print("not a terminating node, begin recursion:")
                    value = self.minimax_values(test_board,0)
            else:
                value = reward
            #print("final value of {},{}:{}".format(x,y,value))
            if value>best_value:
                best_value = value
                best_move = x,y
            #print("_____")
        return best_move

    def minimax_improved(self, game):
        actions = game.show_valid_moves()
        if actions is None:
            print("""


            no actions2

            """)
            return None
        #print("""
#moves to test: {}
#____""".format(actions))
```

```
1          #print("first check if any are immediate wins/forced loss:")
2          best_move = actions[0]
3          best_value = -1000
4          non_terminating_nodes=[]
5          for x, y in actions:
6              test_board = copy.deepcopy(game)
7              test_board.addl_output, test_board.human_render = False, False
8              _, reward, _, err = test_board.move(x, y)
9              if reward > 0:
10                 # print("""{}, {} is an immediate win, by doing this we have ...
                       removed
11 #the need to search {} potentially recursive ...
       moves""".format(x,y,len(actions)-actions.index((x,y))))
12                 return x,y
13             if reward == 0:
14 #                print("""{},{} is a non-terminating node""".format(x, y))
15                 non_terminating_nodes.append((x,y))
16         if len(non_terminating_nodes) == 0:  # no winning or non-terminating ...
               moves, so only remaining must be force lose
17 #            print('this is a force lose')
18             return actions[0]  # return anything
19 #         print("\nBecause there is no immediate win, we must check ...
       recursively down each non-terminating move:")
20         for x, y in non_terminating_nodes:
21             test_board = copy.deepcopy(game)
22             test_board.addl_output, test_board.human_render = False, False
23             _, reward, _, err = test_board.move(x, y)
24 #             print("""
25 #testing move {},{}: {} --> {}
26 #""".format(x, y, game, test_board))
27             value = -self.minimax_values_two(test_board, 0)
28             if value == 1:
29 #                print("\nMove {},{} will eventually force a win for player ...
       {}".format(x,y,game.current_player))
30                 if ...
                       non_terminating_nodes.index((x,y))<len(non_terminating_nodes):
31                     pass
32                     # print("no need to check the other moves as this ...
                           already forces a win")
33                 return x,y
34             if value > best_value:
35                 best_value = value
36                 best_move = x, y
37     #          print("_____")
38      #     print("None of the possible choices have a positive value, so just ...
           pick {} with a value of {}".format(best_move,best_value))
39         return best_move
```

```
    def minimax_values_two(self, game, depth, layers=1):
        spacing = "       " * (depth + 1)
        actions = game.show_valid_moves()
        non_terminating_nodes = []
        if actions is None:
            print("""


                  no actions1

                  """)
            return 0
        if self.debug:
            print("{}Moves to find values for: {}".format(spacing, actions))
            print("{}First check if any are immediate wins/forced ...
                loss:".format(spacing))
        for x, y in actions:
            test_board = copy.deepcopy(game)
            test_board.addl_output, test_board.human_render = False, False
            _, reward, _, err = test_board.move(x, y)
            if reward > 0:
                if self.debug:
                    print("""{}Move {}, {} is an immediate win for player ...
                        {}""".format(spacing,x,y,game.current_player))
                return 1
            if reward == 0:
                if self.debug:
                    print("""{}{},{} is a non-terminating ...
                        node""".format(spacing,x,y))
                non_terminating_nodes.append((x, y))

        if len(non_terminating_nodes) == 0:  # no winning or non-terminating ...
            moves, so only remaining must be force lose
            if self.debug:
                print('{}All of the available moves create a loss, so this ...
                    is a force lose for player ...
                    {}'.format(spacing,game.current_player))
            return -1  # return anything

        if depth > self.depth:
            if self.debug:
                print("Too deep")
                print(game,game.current_player,game.count(game.current_player),game.count(
            return ...
                (game.count(game.current_player)-game.count(test_board.current_player))/len
        if self.debug:
            print("\n{}Because there is no immediate win, we must check ...
                recursively down each non-terminating move:".format(spacing))
        zero_values = []
        for x, y in non_terminating_nodes:

            test_board = copy.deepcopy(game)
            test_board.addl_output,test_board.human_render = False, False
```

```
1
2               _, _, _, _ = test_board.move(x, y)
3               if self.debug:
4                   print("""
5 {}testing move {},{}: {}-->{}\n""".format(spacing, x, y, game, test_board))
6               value = -self.minimax_values_two(test_board, depth + 1)
7               if value == 1:
8                   if self.debug:
9                       print("{}I think this move should result in a force win ...
                            for player {}".format(spacing, game.current_player))
10                  return value
11              if value == 0:
12                  if self.debug:
13                      print("{}This returns a value of 0 (likely because of ...
                            recursion depth".format(spacing))
14                  zero_values.append(value)
15              if value == -1:
16                  if self.debug:
17                      print("{}This results in a negative value for player {} ...
                            so it wouldn't pick {},{} unless ...
                            forced".format(spacing,game.current_player,x,y))
18          if len(zero_values) == 0:
19              if self.debug:
20                  print("{} All the non-terminating nodes ({}) have a negative ...
                        value, so player {} would be forced to pick from one of ...
                        them".format(spacing,non_terminating_nodes,game.current_player))
21              return -1
22          #print("{}there are some zero value nodes somehow".format(spacing))
23          return ...
                (game.count(game.current_player)-game.count(test_board.current_player))/len(gam
24
25
26      '''def minimax_values_type_two(self, game, depth, layers=1):
27          spacing = "       " * (depth + 1)
28          #         print("""{0}true game turn:{1}, depth of tree: {2}, ...
                    testing game turn: {3}
29          # {0}real game current board: {4}
30          # {0}testing imagined board: {5}
31          #         ...
                """.format(spacing,self.game.turn_count,depth,game.turn_count,self.game,game))
32          is_opposing_player = game.current_player != self.game.current_player
33          print('opp', is_opposing_player)
34
35          if depth > 10:
36
37              estimated_value = ...
                    self.game.count(self.game.current_player)/len(self.game)
38              print("too deep, estimated value",estimated_value)
39              return 0
```

```
        actions = game.show_valid_moves()
        if actions is None:
            print("""


            no actions1????????

            """)
            return 0
        print("{}moves to find values for: {}".format(spacing, actions))
        value = 0
        non_terminating_nodes = []
        for x, y in actions:

            test_board = copy.deepcopy(game)
            test_board.addl_output, test_board.human_render = False, False

            _, reward, _, err = test_board.move(x, y)
            # if test_board.boardself.state_memory
            # self.state_memory[test_board.board]
            print("""
        testing move {},{}: {} --> {}
        """.format(x, y, game, test_board))
            if is_opposing_player:
                if reward > 0:  # a winning move for the opposing player
                    return -1000  # don't need to search others, already a ...
                        low value move
                if reward < 0:
                    value -= reward
                else:
                    non_terminating_nodes.append([x,y])
            else:
                if reward > 0:
                    value += reward
                if reward < 0:
                    pass

        for x, y in actions:

            test_board = copy.deepcopy(game)
            test_board.addl_output,test_board.human_render = False, False
            curr_player = test_board.current_player
            state, reward, done, err = test_board.move(x, y)
```

```
{}testing move {},{}: {}-->{}""".format(spacing, x, y, game, test_board))
            if reward == 0:
                if err is not None:
                    print(err)
                else:
                    print("{}not a terminating node, begin ...
                        recursion:".format(spacing))
                    value += self.minimax_values_type_two(test_board, depth ...
                        + 1)
            elif curr_player == self.game.current_player:
                print("{}---a terminating node, value {}".format(spacing, ...
                    reward))
                value += reward
            else:
                print("{}---a terminating node, value {}".format(spacing, ...
                    -reward))
                value -= reward
        print("""
{}total value: {}
        """.format(spacing, value))
        return value

    def minimax_part_two_type_two(self, game):
        actions = game.show_valid_moves()
        if actions is None:
            print("""


            no actions2

            """)
            return None
        print("""
moves to test: {}
____""".format(actions))
        non_terminating_nodes = []
        for x, y in actions:

            test_board = copy.deepcopy(game)
            test_board.addl_output, test_board.human_render = False, False

            _, reward, _, err = test_board.move(x, y)
            print("""
testing move {},{}: {} --> {}
""".format(x, y, game, test_board))
```

```
            if reward > 0:  # a winning move
                return x, y
            if reward == 0:
                if err is not None:
                    print(err)
                else:
                    non_terminating_nodes.append([x,y])
                    print("not a terminating node, begin recursion:")
                    # value = self.minimax_values_type_two(test_board, 0)
            print("_____")
        if len(non_terminating_nodes) == 0:
            return actions[0]
        else:
            print("no immediate winning moves, check non_terminating moves")
            best_value, best_move = -1000, non_terminating_nodes[0]
            for x,y in non_terminating_nodes:
                test_board = copy.deepcopy(game)
                test_board.addl_output, test_board.human_render = False, False
                test_board.move(x, y)
                value = self.minimax_values_type_two(test_board, 0)
                if value > best_value:
                    best_value = value
                    best_move = x,y
            return best_move'''



    # Returns optimal value for current player
    # (Initially called for root and maximizer)
    def minimax_alpha_beta(self, depth, nodeIndex, maximizingPlayer,
                values, alpha, beta):

        # Terminating condition. i.e
        # leaf node is reached
        if depth == 3:
            return values[nodeIndex]

        if maximizingPlayer:

            best = -100

            # Recur for left and right children
            for i in range(0, 2):

                val = minimax_alpha_beta(depth + 1, nodeIndex * 2 + i,
                                False, values, alpha, beta)
                best = max(best, val)
                alpha = max(alpha, best)

                # Alpha Beta Pruning
                if beta <= alpha:
                    break

            return best
```

```
        else:
            best = 100

            # Recur for left and
            # right children
            for i in range(0, 2):

                val = minimax_alpha_beta(depth + 1, nodeIndex * 2 + i,
                             True, values, alpha, beta)
                best = min(best, val)
                beta = min(beta, best)

                # Alpha Beta Pruning
                if beta ≤ alpha:
                    break

            return best



# board_game = Board(8)
#
# print('a')
# player_one = Agent(1,board_game,algo=0)
#
# player_two = Agent(2, board_game,algo=1)
# print('b')
# q = 0
# while not board_game.done:
#     q += 1
#     player_one.play()
#     player_two.play()
#     if q ≥ 3000:
#         print('too many steps')
#         break


# Board, Count = MakeBoard(6)
# print(Board[2:-2])
# Board, Count = Move(Board, 0, 2, Count)
# EndStateChecker(Board,Count)
# print(Board[2:-2])
# Board, Count = Move(Board, 5, -1, Count)
# EndStateChecker(Board,Count)
# print(Board[2:-2])
# Board, Count = Move(Board, 2, 1, Count)
# print(Board[2:-2])
# EndStateChecker(Board,Count)
# print(Board[2:-2])
```

B LinearAtaax Frontend Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 29 14:34:12 2020

@author: jonny
"""
from FinalLinearAtaxx import Board, Agent


Blueboard=[]
Redboard=[]
drawboard=[]
for j in range(3,9):
    win=0
    draw=0
    loss=0
    for i in range(1):
        print("""  ----------------------------------------
 |                                        |
 |                                        |
 |                                        |
 |       NEW GAME START SIZE:             |
 |                    {}                  |
 |----------------------------------------|
            """.format(j))
        board_game = Board(j)
        player_one = Agent(1,board_game,algo=3,debug=False,depth=3)
        player_two = Agent(2,board_game,algo=3,debug=False,depth=3)
        q = 0
        while not board_game.done:
            q += 1
            player_one.play()
            if board_game.done:
                break
            player_two.play()
            if q ≥ 3000:
                print('too many steps')
                break
        b=board_game.count(1)
        c=board_game.count(2)
        if b>c:
            win+=1
        if b<c:
            loss+=1
        if b==c:
            draw+=1
    Blueboard.append(win)
    Redboard.append(loss)
    drawboard.append(draw)
```