CS 4701 Final Report

Philip Ayoub (pja66)

Fall 2021

1 Project Title

You cannot beat me in Connect4! I have the power of Mini-max Search boosted by Alpha/Beta Pruning!

2 Links

Github: https://github.com/pja66/CS-4701-Project-Connect-4.git

Video: https://drive.google.com/file/d/12FPl4tguwOIXCPGoBV8bAoeZofVhFMx0/view?usp=sharing

3 Project Overview

3.1 Introduction

This paper documents our CS 4701 Project. Our program harnesses the power of Artificial Intelligence to create a CPU that plays *Connect4*. Using this program, I was able to gain great insight into not only the power of Mini-Max Search but also how output variance is affected by input parameters. I hope you enjoy our work.

3.2 System Design

I used Java (SE 13) accompanied with GitHub source control to implement this project. Our overall program has 4 main aspects: Game Architecture, Mini-Max search w/ AB pruning, Heuristic Search Algorithms, and User Interface.

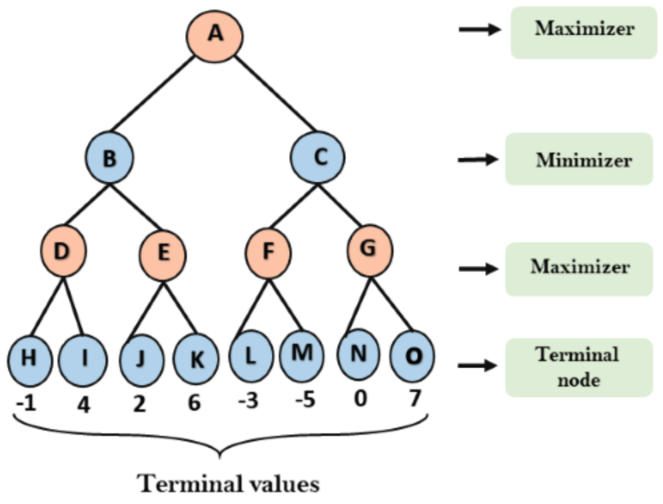
3.3 Game Architecture

Connect4 is a two-player, complete knowledge game consisting of a vertical 7x6 board where players alternate stacking their specific game pieces in columns. Since the board is vertical, the pieces always drop down to the lowest empty row in that column. If the column already has 6 pieces in it, that column is considered full and there cannot be any more pieces stacked on that column. The object of the game is to have 4 pieces in a row without the opponent's game piece interrupting your pattern. The 4 in a row pattern must be vertical, horizontal, or diagonal.

3.4 Mini-Max with Alpha-Beta Pruning

3.4.1 Overview

The Mini-Max algorithm is a tree-based search algorithm that leverages a computers ability to simulate multiple "board states" to determine the "best" playing strategy based on the given inputs. Each move simulated falls into 2 categories: A minimization or a maximization. If it is the players turn that initiated the algorithm then Mini-Max wants to pick the move that will result in the highest possible score. This represents the person wanting to pick the best possible move. However, if in the simulation it is the opponents turn, then the algorithm picks the move minimizing the score to simulate the opponent wanting to make a move that is the worse for there adversary. This is where the algorithm gets its name.



There are 2 main node states that result, terminal leaf nodes and non-terminal leaf nodes. Terminal leaf nodes mean that the game has ended before the search-depth limit has been reached. Thus, the node represents either a win, a lose, or a tie. In our implementation a win is weighted as 100, a lose as -100, and a tie as 0. These weights are independent of the heuristic algorithm being used. If the algorithms search contains non-terminal leaf nodes, the algorithm then uses a heuristic algorithm to weight the current state of the on going game. This is when choosing a "good" heuristic algorithm comes into play since it is very hard to simulate all games to completion as discussed later. These leaf values are then propagated back up the tree in accordance to the rules defined above. What is returned is a final score that represents the best move that can be made.

3.4.2 Alpha-Beta Pruning

A/B Pruning is an optimization add on to the tree search. It allows me to eliminate search branches that would have no impact on the outcome of the algorithm.

3.4.3 Parameters

Mini-Max needs 4 input parameters:

- **Board:** This is required to run the Mini-max Algorithm
- **Turn:** Knowing the initial turn helps the algorithm keep track of whether it is minimizing or maximizing particular moves.
- **Search depth:** The search depth of the algorithm relates to how "smart" the CPU is. The larger the depth, the more moves it can simulate and thus the better the predictions become.
- **Heuristic Algorithm:** The heuristic is key to implementing our knowledge of Connect4 into the program. These algorithms have a direct impact on the effectiveness of the search.

3.4.4 Complexities

Space Complexity: $O(bm)$

Time Complexity: $O(b^m)$

The upper bound for possible game states is 7^{42} which is equivalent to $3*10^{35}$. Thus, this is why I cannot simulate every move to completion. This would involve much more computing power and time then I have available.

3.5 Heuristic Algorithms

Heuristic algorithms are a key component behind Mini-Max search. I have developed 4 main algorithms to be used in our implementation.

3.5.1 Naive Move

The *Naive Move* heuristic provides a random score for a non-terminal leaf node. In essence, this algorithm does not favor any particular strategy, and assigns a random score for the board state in question.

3.5.2 Maximum Connected Pieces

The *Maximum Connected Pieces* heuristic, generates a score based on how many connected pieces are on the board. If there are more connected pieces on the board, the algorithm weights that particular board state higher.

3.5.3 Value Center of Board

The *Value Center of Board* heuristic generates a score for the board based off of a matrix. This matrix weights the center of the board more favorably than the edges. This causes the algorithm to place moves that are more center oriented.

$$\begin{vmatrix}
3 & 4 & 5 & 7 & 5 & 4 & 3 \\
4 & 6 & 8 & 10 & 8 & 6 & 4 \\
5 & 8 & 11 & 13 & 11 & 8 & 5 \\
5 & 8 & 11 & 13 & 11 & 8 & 5 \\
4 & 6 & 8 & 10 & 8 & 6 & 4 \\
3 & 4 & 5 & 7 & 5 & 4 & 3
\end{vmatrix}$$

3.5.4 Value Corners of Board

The *value corners of the board* heuristic generates a score for the board based off of a matrix. This matrix weights the edges of the board more favorably than the center. This causes the algorithm to place moves that are oriented to the sides of the board.

13	11	8	3	8	11	13
10	8	6	4	6	8	10
7	5	4	1	4	5	7
7	5	4	1	4	5	7
10	8	6	4	6	8	10
13	11	8	3	8	11	13

3.6 User Interface

The project provides 3 main ways to interact with the algorithm:

- **Human Vs. CPU:** A person can play the CPU with specific input parameters.
- **CPU Vs. CPU:** A person can watch the computer play itself with specific input parameters.
- **Large Scale CPU vs. CPU:** A person can conduct research using multiple simulations(ie. 500) on CPU vs CPU games with specific input parameters.

3.7 Goals

I had 4 main questions I aimed to answer over the course of this project?

1. How important is it to go first when playing Connect4?
2. Was our intuition of how to optimally play *Connect4*, shown in our heuristic algorithms, correct?
3. How does win rate increase with search depth?
4. What is more important to developing a good CPU, heuristic strength or search depth?

4 Code Base

4.1 Game.java

4.1.1 Overview

The game class implements the physical board and all aspects associated with playing the game. It makes sure to start and end the game at the proper times. It also ensures that moves are recorded properly and are inputted by the correct player, since consecutive moves by the same player creates an invalid game. Finally, it provides aspects used for user interface such as ”pretty printing” the game board.

4.1.2 Key Methods

GameOver

The input for this function is void. There are 3 possible outputs from the gameOver method:

- 0, when the game is still going on
- 1, when the game ends as a draw
- 2, when the game is won

The first output is 0. This is the result when the function checks the board and there are still remaining empty spots and neither player has a 4 in a row. The second output is 1. This output is the result when the function checks the board and there are no remaining empty spots, but neither player has a 4 in a row. The final output is 2. This output is the result when the function checks the board and one of the players has a 4 in a row, regardless of how many remaining empty spots there are.

ChangeTurn

Both the input and output of this function are void. All that this function does is change the turn to the other player when it is called. It does this by checking the current player's color (either yellow or red), and changes the turn to the other player's color.

PlaceMove

The input for this method is a number indexed at 0. The output for this method is the row index of the input if the input is valid. If the input is not valid, it returns -1. It does this by checking whether the input is a column within the predefined Connect 4 board which is 6 columns. If it is a valid column, it loops through the spaces in the column to make sure that the column is not full. If the column is not full, it adds the color of the current player to the top of the column. Otherwise, it will return -1.

RemoveMove

The input for this method is a number indexed at 0. The output for this method is the row index of the input if the input is valid. If the input is not valid, it returns -1. It does this by looping through each row of the specified column, starting from the top of the column and moving towards the bottom. When it finds an occupied row, it checks that the color in that row matches the color to be removed. If it does, it sets the space to 0 (empty) and returns the row. If it does not find a spot that fulfils those constraints, it returns -1.

ResetBoard

Both the input and output of this function are void. All that this function does is remove all of the pieces from the board to result in an empty board. It does this by creating a new 2D array with 6 rows and 7 columns and setting the current board to be the new board.

ShowBoard

The input for this method is void, and the output for this method is the Terminal output. This method prints the board to the terminal. It does this by looping through each space on the board. Each space in the board is surrounded by a " — ", to clearly denote the space to the user. If the space is full, the program checks the color of the player that filled the space. If the player color is yellow, the lines are filled with a yellow circle. Otherwise, if the player color is red, the lines are filled with a red circle. If the space is empty, the space is filled with empty space. Once the board is printed to the screen, the function loops through the columns one more time. Below each column on the board, the index of the column is printed, so that the user can see which column it wants to place its piece in without having to waste time counting columns.

4.1.3 Testing

First, I test that the game can sense when the board is empty or full. In order to test this, I create a new game, which is always empty, and make sure that the game is not over. Then I fill the board with identical pieces, and make sure that the game is over. Next, I test that the game can sense the 3 types of wins: horizontal win, vertical win, and diagonal win. I test the horizontal and vertical wins by creating their corresponding 4 in a row boards. Then I check that the game returns 2, which is the output for a game that has been won. For the diagonal win, I increment both the row and column each time I go through the loop and add the same colored piece to the board to create a diagonal 4 in a row. For this, I also check that the game returns 2.

Next, I test that the game handles turn changes correctly. To test this, I simply test odd and even sequences of moves, and make sure that the resulting turn is of the correct color.

Finally, I test that the game is able to remove moves correctly. If I try to remove a move that does not exist, I check that the function returns -1. Otherwise, I make sure that when I remove a move, the resulting board has an empty space where the move used to be and that the turn is changed back to the correct color (either red or yellow).

4.2 HeuristicAlgos.java

4.2.1 Key Methods

NaiveMove

This heuristic algorithm takes the Game as an input and returns a random score for the board layout. The function obtains this random score simply by using the built-in random function in Java.

MaxConnected

This heuristic algorithm takes the Game as an input and returns a score for the board layout based on how many connected pieces are on the board. It does this using a simple depth-first search algorithm, adding contiguous sets of connected pieces to a counter. A board layout with more connected pieces will have a higher score.

ValueCenterofBoard

This heuristic algorithm takes the Game as an input and returns a score for the board layout based on a matrix of values that are weighted towards the center of the board. This matrix, shown earlier in section 2.4, was created by researchers Xiyu Kang, Yiqi Wang, and Yanrui Hu in their research paper published in the Journal of Intelligent Learning Systems and Applications titled Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game. The values of the matrix increase as you move towards the center of the board, and are lower at the edges of the board. As a result, the score output is greater when there are more pieces in the center of the board.

ValueCornersofBoard

This heuristic algorithm takes the Game as an input and returns a score for the board layout based on a matrix of values that are weighted towards the edges of the board. This matrix, shown earlier in section 2.4, is adapted from the matrix used in *valueCenterofBoard.* The values are switched to favor the edges of the board, and they decrease as you move towards the center of the board. As a result, the score output is greater when there are more pieces on the edges of the board.

4.2.2 Testing

For the naive method, I simply test that the output score is within the given constraints, which means that the integer output must be less than 50.

For the *maxConnected* method, I test that the score output is equivalent to the maximum number of connected pieces. More specifically, I test 5 separate situations: no connected pieces (empty game), 3 connected pieces, 2 connected pieces with a break and then 1 more piece, a 2x2 square of connected pieces, and 4 diagonally connected pieces. The outputs of these tests should be as follows: 0, 3, 2, 4, and 4.

The *valueCenterofBoard* and *valueCornersOfBoard* methods are tested in the same way. I simply test that the score output is equal to the corresponding sum of the scores that are described in the matrix. Each piece corresponds to a score based on its position in the matrix, and if the sum of those pieces is equal to the sum of their spaces in the matrix, the methods are working correctly.

4.3 Minimax.java

4.3.1 Key Methods

BestMove

The inputs for this method are the Game, the depth of search, and the heuristic algorithm of your choice. The output of this method is the column to place the next move. If there is a tie between the columns to choose, the algorithm will choose randomly. This move goes through each of the columns, running the Minimax algorithm on each of the 7 possible moves and chooses the column with the best score. If no legal moves are returned, the algorithm will choose one of the columns at random.

MiniMaxSearch

The inputs for this method are the Game, the depth of search, the current game turn, the heuristic algorithm of your choice, the alpha integer for alpha-beta pruning, and the beta integer for alpha-beta pruning. This algorithm implements the Mini-Max algorithm with alpha-beta pruning, as described in section 2.3. However, if the input depth leaves non-terminal leaf nodes, the algorithm returns a score for these nodes based on the heuristic function decided by the user.

4.4 UserPlayGame.java

4.4.1 Implementation

This feature is implemented entirely in the main() method. First, it asks the user in the terminal whether they would like to play Connect4 or read the directions. When the user decides to play the game, they are given the option to play Human vs CPU or watch CPU vs CPU. For both options, the user gets to choose a search depth between 1 and 8, as well as a Heuristic algorithm which is represented by an integer between 0 and 3. The game then allows the user or CPU to input moves during their turn, until the board is full or one of them wins. The CPU moves are decided by the *BestMove* method in Minimax.java, the search depth, and the heuristic algorithm.

4.5 ResearchPlayGame.java

4.5.1 Key Methods

PlayGame

This method's inputs are a heuristic algorithm for CPU1, a heuristic algorithm for CPU2, the search depth for CPU1, and the search depth for CPU 2. The implementation of this function is the same as the CPU vs CPU implementation in UserPlayGame.java described in section 3.4.1. The output is 0 if the game is a tie, 1 if the CPU1 wins, and 2 if the CPU2 wins.

Main

The method asks the user for the search depth for CPU1, the search depth for CPU2, the heuristic algorithm for CPU1, and the heuristic algorithm for CPU2. It also asks for the number of simulations that the user would like to run. After that, the method runs the specified number of simulations, keeping track of the wins, losses, and ties between the 2 CPUs. The output is the number of CPU1 wins, the number of CPU2 wins, and the number of ties.

5 Research

Now allow me to discuss the findings of the questions posed at the start of our project. To understand our research properly you must know that all of our tables/findings are from the point of view of CPU1 unless otherwise stated. In addition, provided below are the mappings between the names of the heuristic algorithms and an ID that will represent them in the proceeding tables.

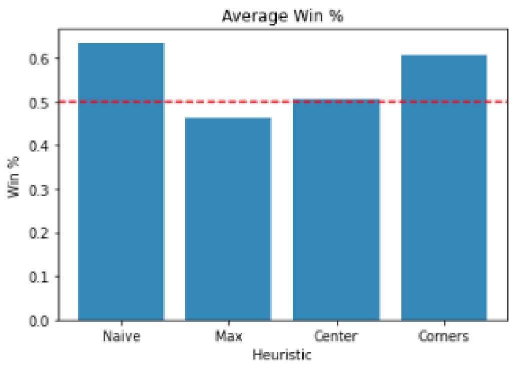
Heuristic Name	**ID**
Naive Move	0
Maximum Connected Pieces	1
Value Center of Board	2
Value Corners of Board	3

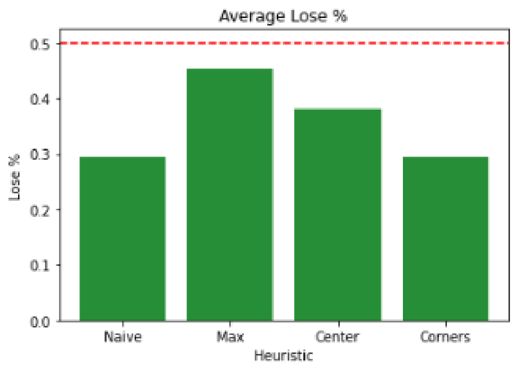
5.1 How Important Is It To Go First When Playing Connect4?

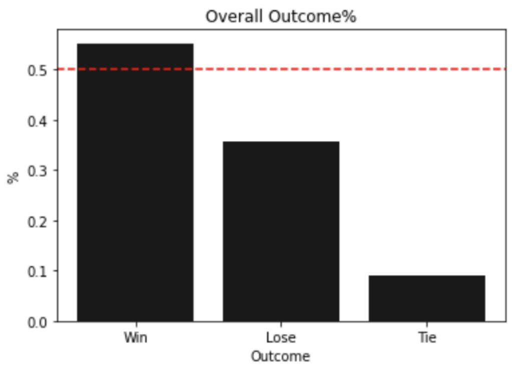
I investigated this question by conducting simulation analysis on the outcome of multiple games played by 2 identical CPUs (Same heuristic and search depth). Each permutation was played **1000** times as to occur steady-state values.

Table 1: **Move First**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
0	5	0	5	635	295	70
1	5	1	5	462	453	85
2	5	2	5	506	382	112
3	5	3	5	607	295	98







As you can see from the following results, placing the first move incurs a substantial advantage. For 3 out of the 4 heuristic algorithms it results in a 2 to 1 win to lose ratio. This is a major advantage.

5.2 Was Our Intuition Of How To Optimally Play Connect4 Correct?

Our knowledge of *Connect4* was implemented into the code base through our design choices for the various heuristic algorithms. As the programmers I have an idea of which algorithms are superior but it is better to allow simulation to tell us. Thus, I will simulate different heuristics against each other but keep the search depth the same. For solidarity sake, I will do each permutation twice and average the results. (So each heuristic will have a chance to go first). Again, each row is simulated **1000** times as to occur steady-state values.

Table 2: **Naive Algorithm**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
0	5	1	5	635	295	70
1	5	0	5	315	589	96
0	5	2	5	658	269	73
2	5	0	5	334	567	99
0	5	3	5	502	425	73
3	5	0	5	445	468	87

Table 3: **Max Connected Algorithm**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
1	5	0	5	315	589	96
0	5	1	5	635	295	70
1	5	2	5	488	408	104
2	5	1	5	465	425	110
1	5	3	5	356	518	126
3	5	1	5	512	359	129

Table 4: **Value the Center Algorithm**

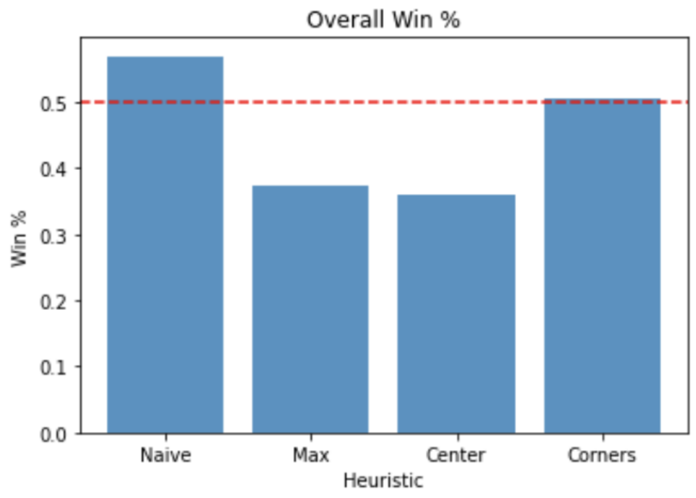
CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
2	5	0	5	334	567	99
0	5	2	5	658	269	73
2	5	1	5	465	425	110
1	5	2	5	488	408	104
2	5	3	5	405	497	98
3	5	2	5	642	273	85

Table 5: **Value the Edges Algorithm**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
3	5	0	5	445	468	87
0	5	3	5	502	425	73
3	5	1	5	512	359	129
1	5	3	5	356	518	126
3	5	2	5	642	273	85
2	5	3	5	405	497	98

Table 6: **Overall Comparison Results**

Heur.	Wins	**Win %**	Loses	**Lose %**	Ties	**Ties %**
Naive	3419	**56.9**	2083	**34.7**	498	**8.3**
Max	2238	**37.3**	3127	**52.1**	635	**10.6**
Center	2154	**35.9**	3277	**54.6**	569	**9.4**
Corners	3039	**50.6**	2363	**39.3**	598	**9.9**



From our data, you can see there is a wide range in the success of our models as well as a number of surprises. The two algorithms, *Value Center* and *Max Connected* that I thought would do the best, actually did the worse. In addition, the fact that our random heuristic did the best was very surprising. This provides evidence that the search depth is a much greater defining factor of Mini-max search but this will be discussed later. The **final rankings** of our algorithms are:

1. Naive Algorithm
2. Value the Edges Algorithm
3. Max Connected Algorithm
4. Value the Center Algorithm

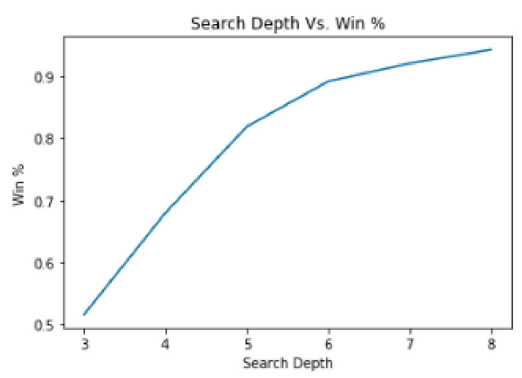
This greatly surprised me as I thought that *Max Connected* would be the best by far. Overall, I would say our initial intuition on how to optimal play *Connect4* was completely wrong.

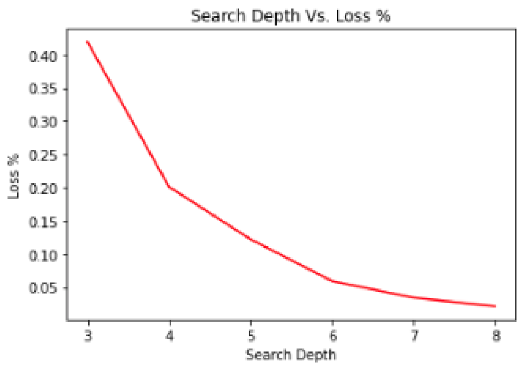
5.3 How Does Win Rate Increase with Search Depth?

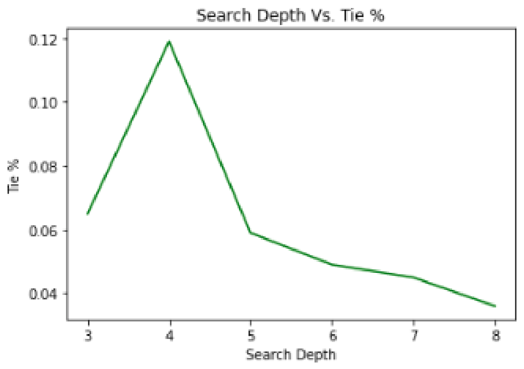
It is easy to assume that as the search depth of a model increases that the win rate should increase as well. However, is this increase linear? polynomial? exponential? This is what I plan to investigate below. Again, each row is simulated **1000** times.

Table 7: **Win Rate Vs. Search Depth**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
0	3	0	3	516	419	65
0	4	0	3	680	201	119
0	5	0	3	819	122	59
0	6	0	3	892	59	49
0	7	0	3	921	34	45
0	8	0	3	943	21	36







Base off the above graphs, you can see that increasing search depth will increase your win percentage linearly but it is not immune to **diminishing returns**.

5.4 What Is The More Important Input Parameter: Heuristic Strength or Search Depth?

For our implementation of Mini-max search, there are 2 primary parameters: Search Depth and Heuristic Algorithm. When developing algorithms it is often important to decided which parameter has a larger correlation to the success of the algorithm. Based on the questions answered above and the experiments conducted below I hoped to answer to provide an answer.

I first thought it would be helpful to develop a control value as a lower bound to base our judgement off of for future results. Thus I tested CPU's with identical heuristics but different search depths. Each permutation was played **1000** times as to occur steady-state values.

Table 8: **Base Line Statistic**

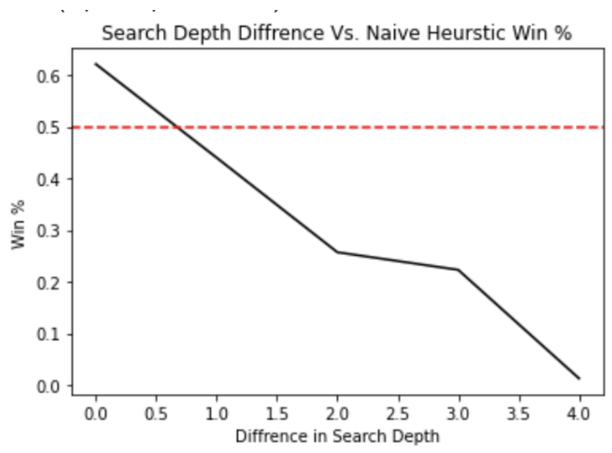
CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
0	1	0	5	10	990	0
1	1	1	5	53	947	0
2	1	2	5	25	975	3
3	1	3	5	33	962	5

As expected the CPU with the larger search depth preformed much better than its counterpart, even with the disadvantage of going second.

Now that I have intuition of our lower bound, and knowledge of which heuristics are superior (From sect 5.2) I can now finally investigate which parameter is stronger. I will pin a strong heuristic, *Naive*, with a low search depth against a weak heuristic, *Value the Center*, with a high search depth. I will try a couple different spreads of search depth to get a more comprehensive view. Again, each row will be simulated **1000** times.

Table 9: **Heuristic Vs Search Depth**

CPU 1 Heur.	**CPU 1 Depth**	**CPU 2 Heur.**	**CPU 2 Depth**	**Wins**	**Loses**	**Ties**
0	1	2	5	22	978	0
2	5	0	1	992	5	3
0	2	2	5	275	636	89
2	5	0	2	745	183	72
0	3	2	5	303	606	91
2	5	0	3	709	179	112
0	4	2	5	468	392	140
2	5	0	4	419	433	148
0	5	2	5	657	267	76
2	5	0	5	327	559	114



As you can clearly see from our data, a large search depth is more important than a strong heuristic algorithm. As the gap closes between search depths, the superior heuristic starts to take over but this is only when the difference is very small as shown by the 50% win ratio marker in the above graph.

6 Conclusion

While heuristics are clearly important, the search depth of the Mini-max search algorithm impacts the success rate of our program far more than any single heuristic function did. The search depth does have diminishing returns as it is increased, however, and I found that a search depth greater than 8 significantly bogs down the computer's resources, making the game much less playable. While I could have predicted that search depth was incredibly important, I did not predict the difference in success between the different heuristic algorithms. The Naive algorithm and the Value the Edges algorithms were the only algorithms to win the majority of the time, while the algorithms I predicted to be most successful, the Value the Center and Max Connected algorithms, had win percentages of 37.3% and 35.9% respectively. As a result, our research has helped me conclude that the ability to think ahead of your opponent is a far better predictor of success in *Connect4* than the strategy that you use.

7 References

"Artificial Intelligence: Mini-Max Algorithm - Javatpoint." *www.javat point.com,* https://www.javatpoint.com/mini-max-algorithm-in-ai.

Garcia Diez, Silvia, et al. "RMINIMAX: An Optimally Randomized MINIMAX Algorithm."

IEEE Transactions on Cybernetics, vol. 43, no. 1, 2013, pp. 385–393., https://doi.org/10.1109/tsmcb.2012.2207951.

Kang, Xiyu, et al. "Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game." *Journal of Intelligent Learning Systems and Applications,* vol. 11, no. 02, 2019, pp. 15–31., https://doi.org/10.4236/jilsa.2019.112002.

Kerr, Wesley. "Connect Four and Effectiveness of Decision Trees." *University of Wyoming,* Department of Computer Science, http://www.u.arizona.edu/ wkerr/pubs/tree.pdf.

Megalooikonomou, Vasilis. "CIS603 S03." *CIS603 Spring 03: Lecture 7,* https://cis.temple.edu / vasilis/Courses/CIS603/Lectures/l7.html.

Nasa, Rijul, et al. "Alpha-Beta Pruning in Mini-Max Algorithm –An Optimized Approach for a Connect-4 Game ." *International Research Journal of Engineering and Technology (IRJET),* vol. 05, no. 04, Apr. 2018, pp. 1637–1641.

Shock, Michele. "AI for a Connect 4 Game." *University of Michigan – Dearborn,* Electrical and Computer Engineering Department, 6 Dec. 2007, http://www- personal.engin.umd.umich.edu/ shaout/connect4.pdf.