Connect 4 Final Report

Sam Coleman, Caitlin Coffey, and Kate Mackowiak

Question

Our goal was to implement a Connect Four AI using a MiniMax algorithm and alpha-beta pruning. We decided on using a MiniMax algorithm after additionally investigating two other algorithms, Q-Learning and Monte Carlo Tree Search. This allowed us to gain broad exposure to new algorithms, before diving deep into an algorithm to implement.

Motivation

All of our team members were interested in learning more about some of the most common AI algorithms used to solve games. We were excited to learn about new algorithms! We decided to do an implementation-based project because we were all excited to create a Connect 4 AI, and felt we would learn a lot in the process.

Initial Research

MiniMax: This algorithm builds a tree of potential outcomes for the game to a certain depth, and calculates a heuristic value for each terminal node of the tree. For each level of the tree, it alternates maximizing and minimizing the heuristic value of the terminal node to simulate competing players making moves. At the top level of the tree, it selects the move which has the highest heuristic to be the most optimal play.

Monte-Carlo Tree Search: The Monte Carlo Tree Search will build a tree with *n* nodes with each node annotated with the win and visit counts. One can think of it as "planning ahead" to reach some goal (in this case, to make the best move). Monte Carlo builds a statistics tree that guides the AI to look at only the most interesting nodes in the game tree. Values of nodes in the tree are determined by running simulations. After enough simulations, Monte Carlo looks a lot like the game tree in MiniMax; however, Monte Carlo doesn't have to go through as many nodes to find an optimal play.

Q-Learning: This is a type of model-free reinforcement learning. The goal is to maximize the reward of an agent by taking a series of actions in response to a dynamic environment. It updates its value function based on a form of a Bellman Equation to determine the Q value. This Q value is an estimation of how good it is to take action A at the state S. These values are stored in a Q table, and the agent will choose an action, perform an action, measure the reward, and evaluate and update the Q table. Over time, it will learn to make moves to maximize its rewards.

Ultimately, we decided to implement the MiniMax algorithm since it seemed like the least complex, viable solution that we could truly delve into and fully understand.

Method

AI.py holds our MiniMax source code.

MiniMax Function

Our Mini-Max function, `minimax`, assigns the AI as the 'maximizer' and the human player as the 'minimizer'. The goal for the maximizer is to choose the decisions with the highest assigned-value as possible, which are usually the decisions that will lead it to winning. The minimizer's goal is to counteract the maximizer by selecting the move with the lowest value possible, also known as the move that will make it most likely to win.

Our function uses a game tree to simulate a sequence of moves in the game, which we develop using a recursive algorithm. Here is a snippet of our code that highlights how the AI (the maximizer) selects its move from the game tree:

```
for move in possible_moves:
    new_board = copy.deepcopy(board)
    new_board = make_move(new_board, move, player)
    # Finding the heuristic of the move tree
    # This is where we recurse!
    result = ((minimax(copy.deepcopy(new_board), PLAYER, depth-1, alpha, beta)[0], move))
    results.append(result)
    [...]
try:
    max_heur = max([heur[0] for heur in results])
    max_results = [key for key in results if key[0] == max_heur]
    return random.choice(max_results)
except:
    return 0, move # (when results is empty, max will return an error)
```

In MiniMax, it is also determined if either player, AI or human, has won. This is intentionally done outside of the heuristic, since checking if a player has won should happen at every level in the game tree (and if someone has won, the game tree should stop recursing down that branch). The current depth is also a factor in this calculation since the AI should prioritize blocking the human player from immediately winning more than blocking the human player from winning in 3 moves. If the depth was not included in the calculation, we would treat a move that allows the AI to win in three moves as something just as important as the AI winning in 1 move. This calculation is achieved in the code snippet below:

```
#check if either player has won
if calc_adjacent(board, AI, 4) > 0:
    #SUPER GOOD--Win
    return 1000000000 * (depth+1), -1
if calc_adjacent(board, PLAYER, 4) > 0:
    #SUPER BAD--Lose
    return -100000000000 * (depth+1), -1
```

Our depth is 4, so we are simulating a sequence of 4 moves for each turn. Further analysis of varying depths is explained in the Results and Interpretation section.

Alpha-Beta Pruning

To speed up our implementation, we incorporated alpha-beta pruning to evaluate fewer branches. Alpha-beta pruning is the process of eliminating possible moves from consideration in a decision tree that are not optimal because they are not to the benefit of the player whose current turn it is. It will have the same results as a typical MiniMax tree, but evaluate fewer branches in the process since they are known to be non-optimal. This significantly decreases the time complexity and allows us to evaluate to a further depth. Alpha is the best value the maximizer can guarantee at the current level or above (and is initialized to negative infinity), and beta is the best value the minimizer can guarantee at that level or above.

While playing as the maximizer, below is the alpha-beta pruning implementation.

```
#alpha beta pruning
alpha = max(max(results)[0], alpha)
if alpha > beta:
  break
```

Our Heuristic

This is implemented in `calc_heuristic`. For our heuristic, we focused on counting the number of streaks (3 or 2) and the position of the piece relative to the rest of the board. A streak in our scenario is either multiple pieces with less than 1 space next to each other (either horizontally, vertically, or diagonally), or multiple pieces where there is only one space separating them, but that space is not occupied.

For example, a streak of 3 pieces of the AI player has a weight of 1000 since it is extremely valuable. Below is our heuristic calculation:

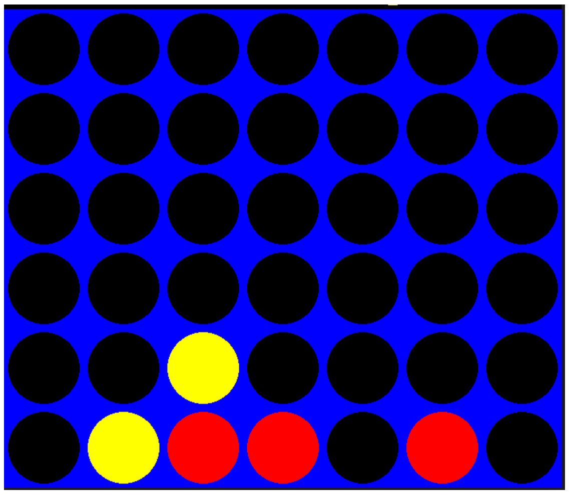
```
rating = (our_middle_perc * 1050 + our_threes*(1000) + our_twos*(20) + enemy_threes*(-1100) + enemy_twos*(-10))
```

We place a higher weight on creating a streak of 2's rather than blocking an opponent's streak of 2's because we felt like the latter was less important. Similarly, we place a slightly larger weight on blocking an opponent's streak of three instead of creating a streak of three because we felt like blocking these streaks was more important in the long run.

Note that we do not take into account double counting when calculating our heuristic. This means that a streak of 3 has two 2 streaks also. We determined our heuristic weights to take this into account, and it does not impact our gameplay.

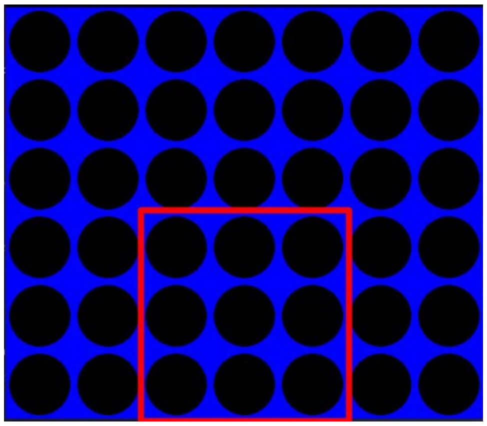
Heuristic Helper Functions

To calculate the heuristic, we have a function, `calc_adjacent`, to calculate the number of streaks for a given player of a given length. This calculates horizontal, vertical, and diagonal streaks. An important improvement to this was the ability to calculate a "disjoint set of 3" in a row (see image below for an example):


Disjoint 3 in a row (in red)

Before, our algorithm was unable to catch disjoint streaks. Now, our algorithm treats disjoint threes as if they were a connected 3 in a row, and appropriately updates their assigned heuristic value.

After some testing, we also decided to weigh the center spots more heavily since they have a higher potential of forming a 4 in a row. After discussing many ways we could weigh the center more, we decided to calculate the percentage of our (the AI) tokens that fill the middle 3 columns from the bottom to third from the bottom row (see image below). After all of these spaces are filled, this part of the heuristic will be the same for all subsequent moves and will not affect our gameplay.


Center spots

Unit-Testing

While unit testing for our whole MiniMax algorithm is difficult due to the need to calculate out the potential game tree, we unit tested each of our helper functions to ensure that our algorithm is functional.

Results and Interpretation

Our metrics of interest were the win-rate of our algorithm as well as our runtime.

Win Rate Vs. Humans - Initial Heuristic

After outsourcing our gameplay to the Olin community via mailing lists, we had a total of 120 games played. These games were played with our initial, non-optimized heuristic. For our non-optimized heuristic, we did not weigh the center spots more, weigh the AI streaks and human streaks differently, and did not take into account the current depth when seeing if either player has won. Furthermore, our MiniMax algorithm did not recognize "disjoint threes", a scenario where a player could still win in one turn.

Our final results were 79 AI wins, which equates to 66%. We lost 38 games, which is 32%, and 3 games, or 2.5%, were a tie. From this, we could see our algorithm played well, winning significantly more games than it lost.

The comments play-testers provided gave more insight into our algorithm and gameplay experience. Some comments about the AI being very hard include: "ai won every time :(," "AI seemed to make some informed decisions (it wasn't only trying to avoid human winning)," and "I thought the AI did really well. I made a dumb error, and it immediately capitalized on it. When I won, it was because I saw the AI couldn't make any other moves but to stack on the same column." Some comments about improvements we could make include: "There were two instances where the AI had to place just one token to win, but it didn't see it. Although I eventually lost, the AI could have won much earlier," "AI sometimes could have lasted longer, instead it place piece that allowed me to win on next move," and "If the AI has no good options, it does really stupid things." Finally, this comment sums up a lot of the feelings players had from both the form and our conversations: "LOOK I WON!!!! LOOK I WON!!! (Also why is it so good. I was so annoyed)."

Overall, we were happy with our results and feedback. We took what we learned from this experience to improve our heuristic function in our next iteration.

Win Rate Vs. Humans - Improved Heuristic

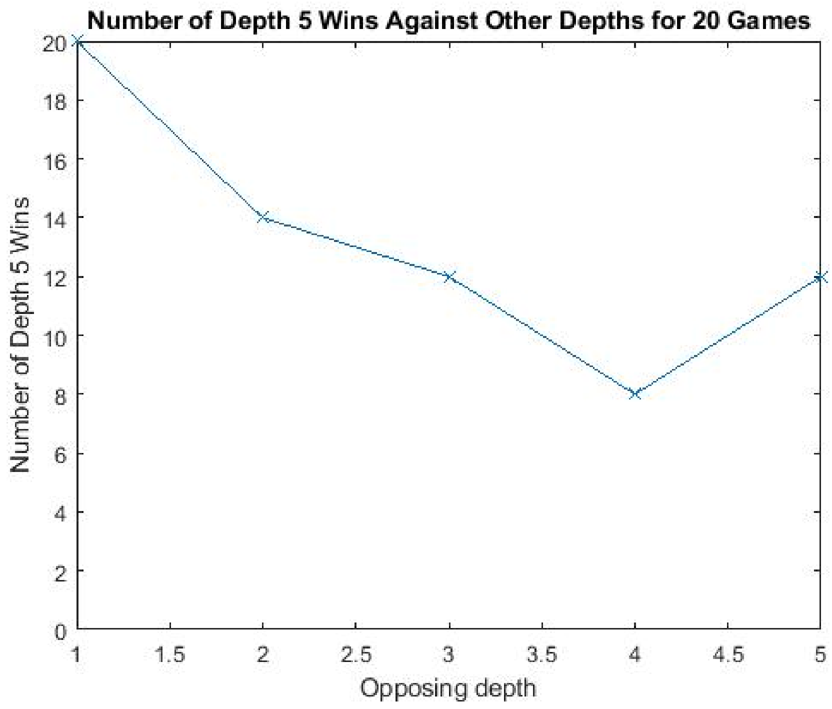
After improving our heuristic to utilize all the strategies described in our methodology, we increased our AI vs. Human win rate to 75% for 32 games. We feel that this is a solid improvement from our prior heuristic function. While connect four is a solved game and ideally our AI would win every time, we are quite happy with our heuristic performance having not looked at the solved strategy. Additionally, since our AI always plays second, the AI has a slight disadvantage. Our performance could potentially be better if we played first or randomized who went first.

Validation Based on Win Rate

Validating our AI by calculating our heuristic values for potential tree branches and ensuring that those heuristics made sense would have been a particularly intense endeavor that we were not equipped to take on. We instead decided to validate our AI by using our win rate testing to ensure that our AI was making smart decisions. We used a comparison win rate of selecting a random piece, and that strategy had a win rate of 0%. Since our AI had a win rate of 75%, we can reasonably assume that it is functional and significantly better than random.

Win Rate Per Depth

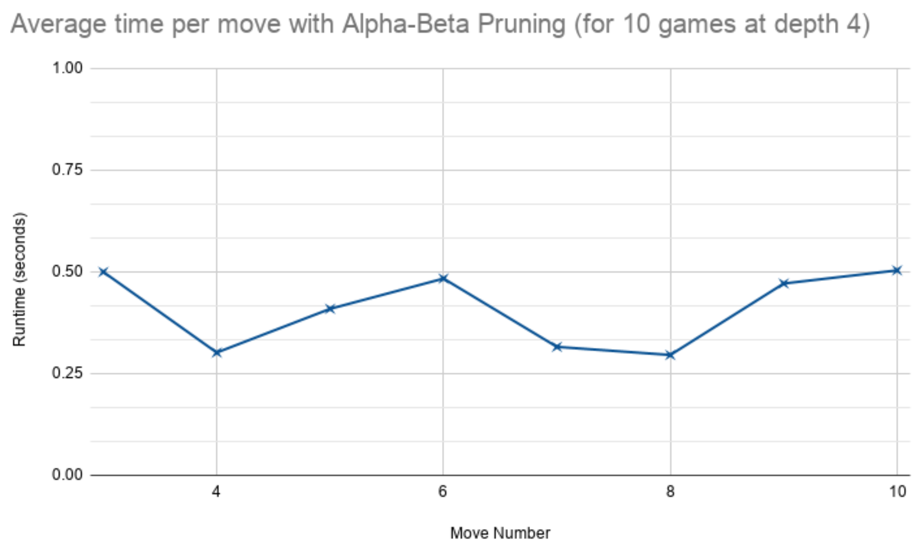
Another method we used to analyze the win rate of our algorithm was by comparing how different tree depths played against each other. If our algorithm worked well, higher depths should win against lower depths. The following graph shows the average win rate of depth five against depths 1-6. The graph follows a roughly linear trend as it decreases, and proves that higher depths are more effective since depth 5 won 100% of times against depth 1, and won ~50% of the time against itself (when both depths were 5).



Depth of 5 Vs. Other depths win-rate

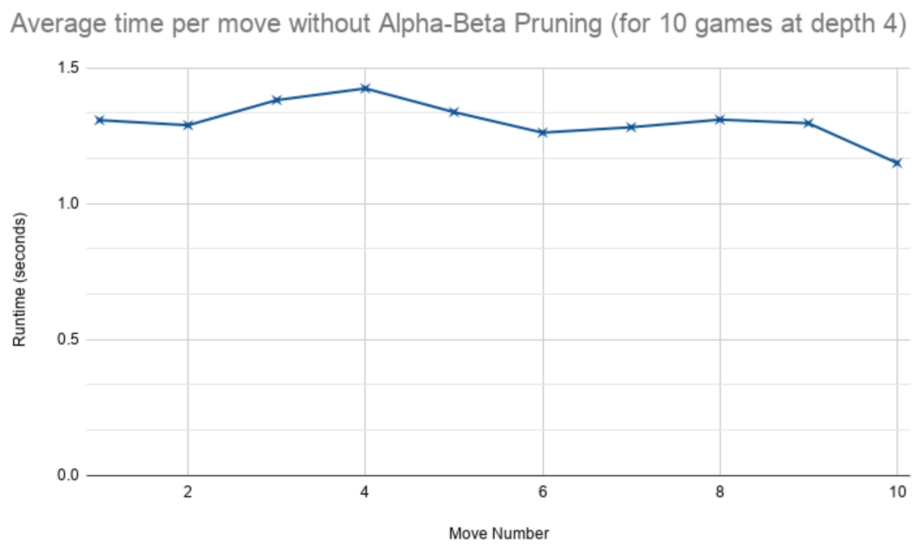
Runtime Per Move

We found that the runtime of our algorithm remained relatively stable over time with alpha-beta pruning. When compared to the runtime graph without alpha-beta pruning (below), it is evident that alpha-beta pruning substantially speeds up our algorithm. This makes sense because we can prune any branches that will not lead us to an optimal solution, allowing us to avoid searching through the entire game tree.



Average time per move with Alpha-Beta pruning

Here is a graph of the average runtime without alpha-beta pruning:

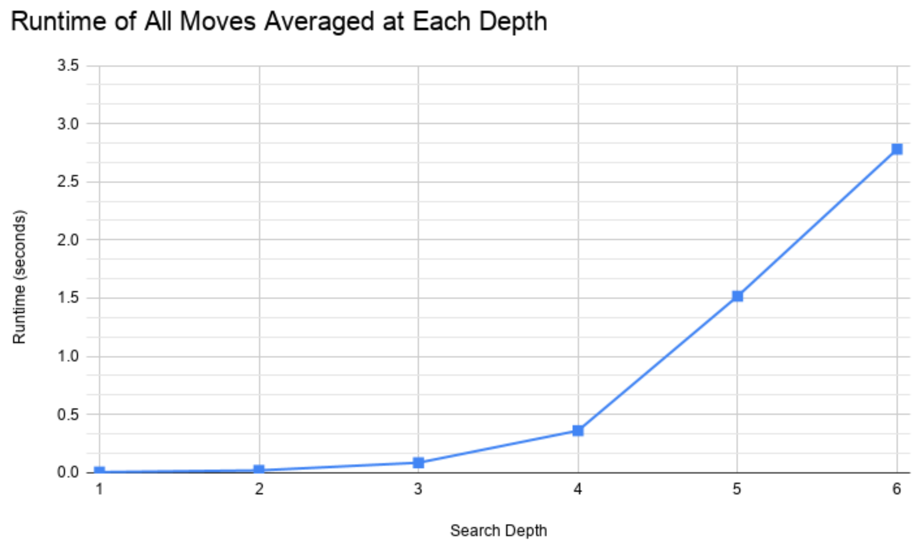


Average time per move without Alpha-Beta pruning

The results from our data show that alpha-beta pruning helps us shave off about .75 seconds on average from each move. This is incredibly useful, especially in the middle and end stages of the game.

Runtime Per Depth

We also looked at the average runtime for a move at different depths, since while higher depths result in a greater win rate, they also result in a longer calculation time. The following graph shows the average time per move at each depth.



Runtime of all moves averaged at each depth

Theoretical Runtime

The overall runtime of building the game tree is O(column_count^depth). In this case, depth is the depth of our calculated game tree, and column count is the number of potential moves per turn. Our heuristic function has the runtime of O(row_count * column_count), drawing from the fact that we iterate through the rows and columns of the board to calculate the number of adjacent pieces.

Optimization Based on Data

Generally, it seems that as the depth is increased, the win rate is also increased. There were exceptions to this trend; however, we feel this is likely due to the low sample size. The average runtime increases approximately exponentially as the depth increases, which is important to note as the game could quickly become too slow if the depth is increased. By optimizing these two axes (depth and runtime) we ended up with an optimal depth of four, to ensure a quick, smart AI.

Next Steps

Our next steps would be to further improve our heuristic algorithm. As of now, our algorithm occasionally misses an obvious move, which indicates that the heuristic assigned to the move should be improved. The heuristic also currently doesn't take into account the fact that a three in a row on an edge doesn't have the potential to become a four in a row, so that could be improved in future iterations as well. We would also like to test different heuristics more rigorously, as opposed to having humans play against it, and have two MiniMax heuristics "battle it out" for a large number of games to edit and refine our heuristic.

Link to Implementation and GitHub

https://github.com/sam-coleman/Connect_4

Works Cited

Base Connect 4 Pygame Implementation:

https://github.com/KeithGalli/Connect4-Python/blob/master/connect4.py

MiniMax Research and Understanding:

Overview of MiniMax: https://sites.google.com/view/lazyy/home?authuser=0#h.mjmrtodxv0f7

Connect four MiniMax: https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f

Alpha-beta basics: https://en.wikipedia.org/wiki/Alpha–beta_pruning

For Q learning Initial Research:

https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

For Monte Carlo Initial Research:

https://mcts.netlify.app/

https://www.youtube.com/watch?v=Fbs4lnGLS8M

https://pranav-agarwal-2109.medium.com/game-ai-learning-to-play-connect-4-using-monte-carlo-tree-search-f083d7da451e