

Project Introduction

Connect 4 is a popular adversarial game where pieces are dropped into one of 7 columns. The first player to get 4-in-a-row wins the game. In this project, 2 AI agents using different algorithms were developed and used for playing. The first agent used the Minimax algorithm with alpha-beta pruning. The second agent used the Monte-Carlo tree search algorithm with a random rollout policy. The goal of the project was to compare the effectiveness of the two algorithms by allowing these agents to battle each other as well as a human player. The way the effectiveness was evaluated will be explained more concretely later.

Minimax Agent

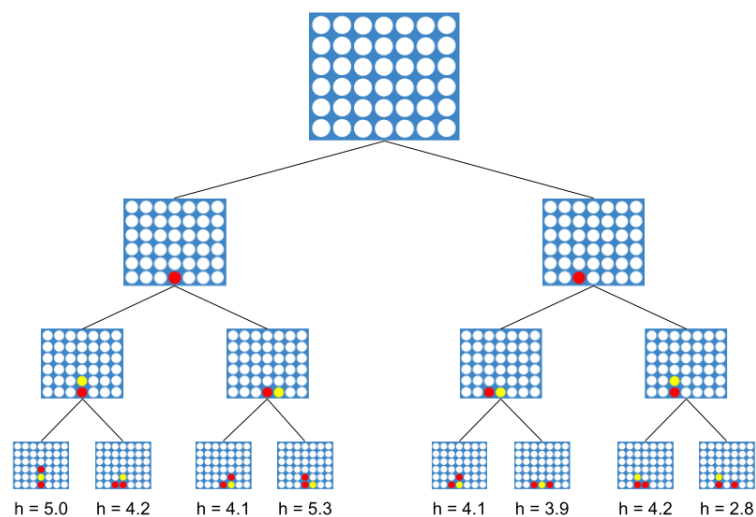
The Minimax algorithm is a recursive algorithm that works well for adversarial games, specifically simple games with a small branching factor and a reasonably small number of moves (to the game's end). It works by creating a tree of all the ways the game can be played out from the current state and assuming that both players play optimally (always choose the best move). At each node, the player looks at its children and chooses the move that brings it to the best child. It returns its move to the parent (which is the opponent), which then does the same: chooses the move that's the best for itself. At the root of the tree, the move that produces the best child is the move that Minimax returns.

The problem is that Connect 4 has a branching factor of 7 and the games can be quite long; the plain Minimax algorithm doesn't work as the search tree is too big to search through. One solution is to stop the search at some depth and apply a heuristic, a function that estimates the favorability of a game state. This allows only a small subset of the tree to be searched. The figure below (Figure 1) shows an example of such a search; the search stops at depth 3 before estimating using the heuristic.

Figure 1

A Connect-4 game tree at the beginning state (red to move) that is cut short (cut at depth 3). Once a depth of 3 is reached, the agent applied a heuristic on the board to estimate how favorable that position is.

Note that there are only two branches from each node. In a real Connect-4 game, there would be 7 branches. Also note that the heuristic values in this figure were made up.

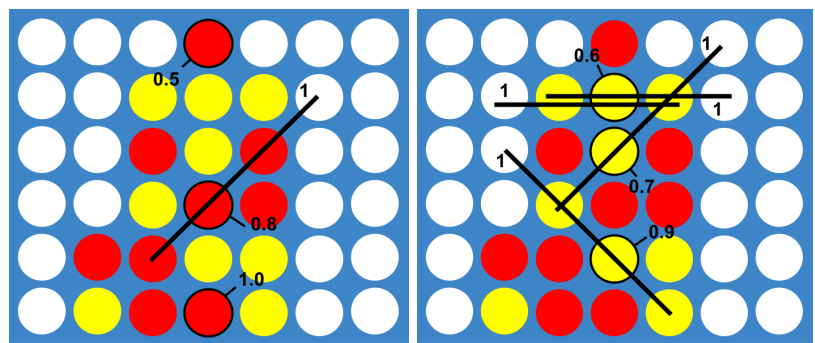


Heuristic

Three slightly different heuristic functions were implemented (in the final experiment, only the best one was used). The heuristic functions counted the number of 4-in-a-rows that the player was one piece away from making, giving 1 point for each occurrence. The second and third heuristics also counted the number of pieces the player had in the middle column, awarding extra points for the pieces for lower pieces. The second heuristic awarded 1, 0.9, 0.8, 0.7, 0.6, 0.5 points for each piece in the middle column; the third heuristic did not award 0.6 or 0.5 for the top two middle pieces. The final output of the heuristic is the agent's points minus the opponent's points. The best heuristic was the second one (rewarding all center pieces), and below is an example of it being applied on a board.

Figure 2

Player red has one occurrence where they need just one more piece to win. They also have a few pieces in the middle column, receiving a score of 3.3 (left). Player yellow has 4 near win occurrences, and a few pieces in the middle as well, receiving a score of 6.1 (right). The heuristic returns 3.3 - 6.1, or -2.8, to player red.

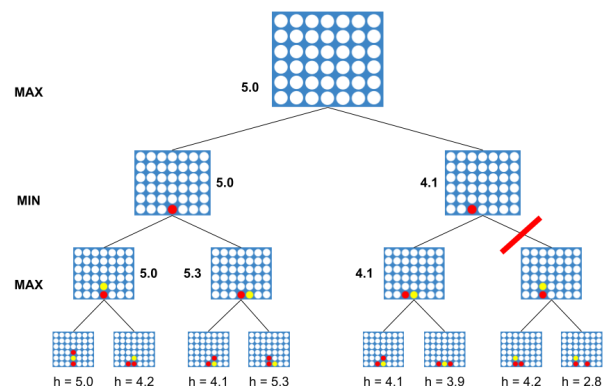


Alpha-Beta Pruning

With the heuristic, the Minimax agent takes a board, searches the entire tree to a certain depth, and returns the move it evaluates to be the best (the "output move"). The application of alpha-beta pruning allows parts of this tree to be pruned, specifically the parts that can't change the output move. In alpha-beta pruning, if a child node detects that it already has information that will repel the parent node from choosing a move from this search tree, it will return without wasting the time to search the remainder of the subtree. Figure 3 applies this pruning method to the tree from figure 1.

Figure 3

After searching the left subtree (no pruning was possible), the search continues to the right subtree. After searching the right subtree's left subtree, the right subtree's right subtree is pruned. This is because the parent will choose the maximum between 5 and whatever the right subtree returns, but the right subtree is already guaranteed to return at most 4.1 since it's going to choose the minimum between 4.1 and whatever its right subtree returns. The right subtree is safe to return 4.1 since the parent is going to choose 5 anyway.

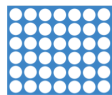


Monte-Carlo Tree Search Agent

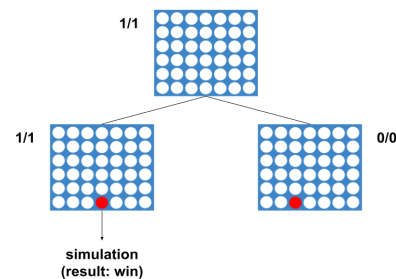
The second agent uses the Monte-Carlo Tree Search algorithm. The main idea behind MCTS is to play many theoretical games from the current board state and choose the move that shows the best win rate. Each game, MCTS expands its search tree by one node (creates a leaf node) and runs a simulation from that leaf node. The tree is built in the directions of the best moves; these are the games that are more likely to happen. After a specified number of games and simulations, the agent chooses the move that resulted in the greatest win rate during the search.

Figure 4

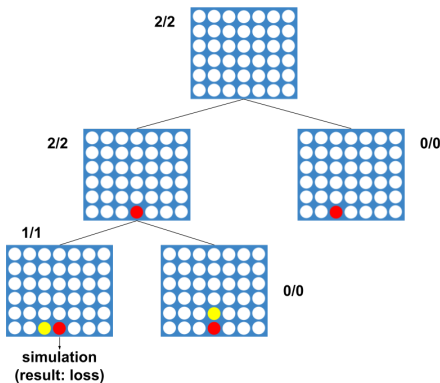
Note: not all children are shown.



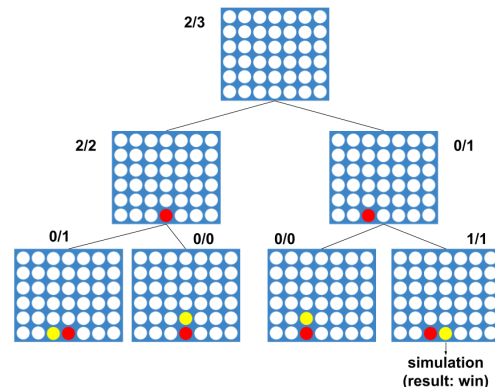
Initially, the tree consists of just the current state. No games or simulations have been run yet.



In the first game, the search instantly encounters a leaf (the root). So, it expands the leaf and creates its children. It then runs a simulation from one of the children. In this case, the simulation resulted in a win for red, so red records the win and game played (notice 1/1).



The search now plays a second game. At the root, it has a decision to make: to go left or go right. Going left is a good idea because it has a good win rate. But the right may have good games that haven't been discovered yet. The search chooses left (more on choosing later), expands the leaf, and plays a simulation on one of the children. The child records a loss. This result is propagated up the tree. The parent records this as a win (since the parent is a red player node).



The search plays a third game. Although going to the left at the root looks like a good and likely move, the search decides to "explore" and go to the right. It encounters a leaf, so it creates its children and runs a simulation on one of them. The child records a win, and this result is propagated up. The parent sees this as a loss (since it's a red player) and updates it as such.

Figure 4 above shows an example of how the first three games and simulations of a MCTS could turn out when considering a move for the empty board. After many iterations of games with simulations, the win rate of each node gets closer to its real win rate. After a reasonable number of simulations, the MCTS agent chooses its official game move to be the one that leads to the child that has the best win ratio.

Exploration vs Exploitation

As already mentioned in figure 4, the algorithm needs to decide which child to traverse to at each node. It can choose the child that produces the most wins since the assumption is that both players will make good moves. The problem with choosing such a child every time is that other children may be better candidates but they have not had the chance to run enough games to show that. One great solution is to favor exploiting (choosing the move that gave good results in the past) but allow exploration to happen as well. For this project, this decision was made using a well-known Monte-Carlo exploration vs exploitation selection function. Each child receives an exploration favorability value based on the following function:

$$child\ favorability = \frac{w_i}{s_i} + c \sqrt{\frac{\ln s_{total}}{s_i}}$$

In this function, w_i is the score (from the parent's perspective) after choosing move (or child) i . For this project, players were awarded 1 point for each win and half a point for each tie (and 0 for loss). Next, s_i is the total number of games played through child i . The exploration parameter, c , was set to the commonly used value $\sqrt{2}$. This value is used to encourage or discourage exploration. Finally, s_{total} is the total number of games played through the current node. There are two main things this equation favors: one, children that produce good win rates (exploitation, left term), and two, children that have had few games played through them compared to the total number of games played (exploration, right term).

Simulation Rollout Policy

One thing not yet discussed is how the simulations are played. In order to run a simulation, both players need a rule to follow. For this project, the simulations were run using a random rollout policy. In other words, the players chose random moves (out of the ones available) until the game ended. Random rollout policies simulate many games very quickly, allowing the search tree to expand to many nodes in little time.

Testing the Agents

To compare the agents, they needed to play in a fair environment. Most notably, they needed to have the same amount of time to think and to start the same number of times. The problem is that the agents don't take time as a rule like us humans do. Instead, as hinted earlier, the Minimax agent accepts a search depth argument, and the MCTS agent takes a simulation count argument.

Luckily, the MCTS agent's simulations take very little time. This fact makes it easy to control the MCTS agent to run for a desired amount of time.

An empty board has the biggest search tree, so the Minimax agent was timed to make a move on the empty board. It was timed for depths 1 through 7; the results can be seen in figure 5. The challenge was to make the MCTS agent run some number of simulations in the same amount of time. To get an idea of how many simulations the MCTS agent could run per second, the agent was commanded to run 50000 simulations for an empty board. This took 49.65 seconds. So, the agent could run about 1007 simulations per second. To make the run times between the agents similar, the MCTS agent ran about 1007 simulations for each second the Minimax agent took. Figure 5 is a table showing the number of simulations the MCTS agent ran in order to get a time similar to that of the Minimax agent.

Figure 5

Minimax Depth	1	2	3	4	5	6	7
Minimax Time (sec)	0.0042	0.0330	0.2001	0.7010	3.5643	33.7586	106.1434
Monte-Carlo Simulations	4	25	185	660	3425	33500	109000
Monte-Carlo Time (sec)	0.0047	0.0331	0.2033	0.7107	3.5345	33.4860	109.2360

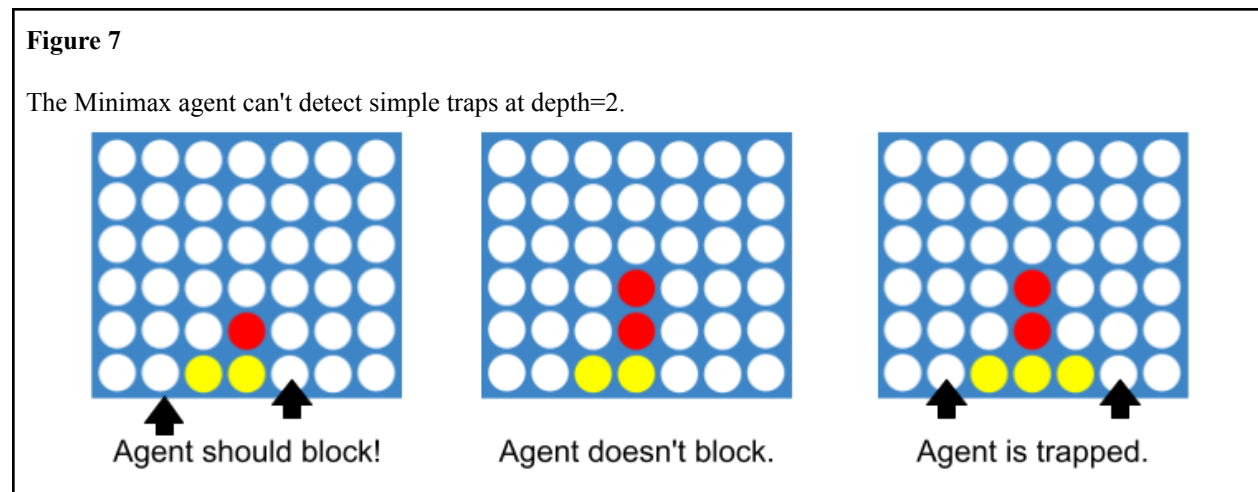
Playing a Human

The first comparison was an indirect one. The two agents played against an average human player, and the results are shown in figure 6 below. The agents and the human started the same number of times.

Figure 6

Minimax Depth	Minimax Record against Human (W-L-T)	MCTS Simulations	MCTS Record against Human (W-L-T)
1	5 - 4 - 1	4	0 - 10 - 0
2	2 - 7 - 1	25	0 - 10 - 0
3	5 - 5 - 0	185	1 - 9 - 0
4	9 - 0 - 1	660	4 - 6 - 0
5	Not played	3425	8 - 2 - 0

The first thing that stands out is how much better the Minimax agent performed against the human. Even with a search depth of 1, the Minimax agent performed well. At depth=2, the human noticed that the agent could not detect upcoming traps. One such trap is a simple 3-in-a-row that needs to be blocked from both sides. When the human would have 2-in-a-row on the bottom row, the agent wouldn't care since it could not see the loss in its shallow tree (see figure 7 below).



Even though the human began to catch on to the agent's patterns by depth=3, he was not able to see every possible game played out after 3 moves like the agent did. The human's strategy that works on many other people, to set up traps within the next few moves, did not work as the agent was able to sniff them out. The human learned he was better off playing it safe and playing to the end, where the agent had a harder time setting unexpected traps. Finally, by depth=4, the human was dumbfounded. The agent was setting up traps everywhere, and the human was just left in the dust. The human decided to not play at depth=5.

The Monte-Carlo Tree Search agent had more weaknesses against the human. At simulations=4, the agent appeared to move randomly; the agent showed a similar behavior at simulation=25. At simulations=185, the agent usually avoided immediate losses but still could not see simple traps. By simulations=3425, the agent was able to set up traps, especially in the endgame.

In the agent vs human comparison, it's safe to say that the Minimax agent performed better than the MCTS agent. At every time group, the Minimax agent recorded more wins and fewer losses than the MCTS agent.

Playing Each Other

The final part was letting the agents play one another head-to-head. Both of the agents started the same number of times. Their results are recorded in figure 8.

Figure 8

Minimax Depth	MC Simulations per move	Minimax Wins	MCTS Wins	Ties	Rate (Minimax to MC score)
1	4	204	186	10	0.52 - 0.48
2	25	35	5	0	0.875 - 0.125
3	185	8	12	0	0.40 - 0.60
4	660	1	19	0	0.05 - 0.95
5	3425	2	8	0	0.20 - 0.80
6	33500	0	4	0	0.00 - 1.00
7	109000	0	2	0	0.00 - 1.00

We can see that the Minimax agent beat the MCTS agent when the agents were given little time to make a decision. This result makes sense because the Minimax agent detects immediate winning and immediate losing moves and applies useful heuristic. These strategies allow the agent to get a decent understanding of the moves in little time. MCTS, on the other hand, relies on many simulations to make an accurate move, so it does not perform well when it receives little time.

With more time, the MCTS agent pulled away and demolished the Minimax agent.

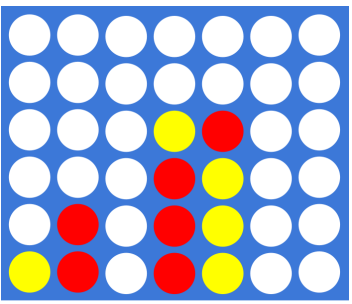
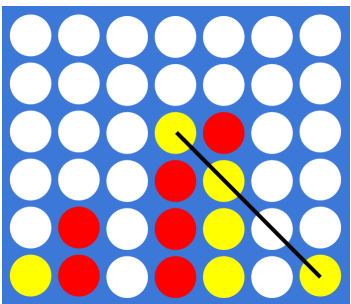
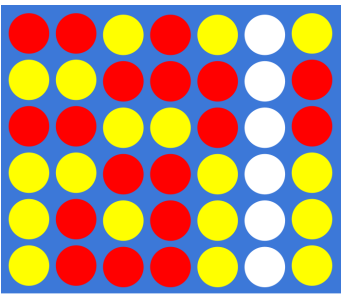
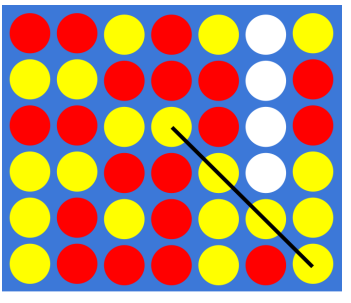
Overview

It may seem surprising that the MTCS agent dominated the Minimax agent since the Minimax agent performed so much better than the MCTS agent when playing against the human. But the way humans think is different. Humans are more focused on short-term play. Their hope relies on setting up traps a few moves in advance (they can't see 20 moves ahead). Relying on short-term play is likely the reason the human struggled to compete with the Minimax agent. The Minimax agent detected all of the human's potential traps and countered them. In the human's favor, the Minimax agent similarly focused on setting quick traps. Unfortunately for the human, he couldn't detect every coming trap and fell victim into many of them.

The human performed better against the MCTS agent because some of the traps he tried worked. The human learned that he needed to win before the endgame, even if it meant making some risky moves to try traps that the Minimax agent wouldn't. If the endgame came, the MCTS agent was the favored winner since it already simulated many games to the end and timed its wins and losses (see figure 9). If the Minimax agent was aware of this and would play more aggressively in the beginning, it may have scored better than it did against the MCTS agent.

Figure 9

A common problem the human experienced when playing against the MCTS agent. The agent was capable of finding moves in the beginning that helped it win in the endgame.

1 The MCTS agent (yellow) is to move. With red controlling the middle, it looks like red is the clear favorite.		2 The agent places in the corner to form a near 4-in-a-row. The MCTS likely played out a game and realized red will be the one to place in the empty column, giving yellow the 4-in-a-row.	
3 All the columns have been filled, and red is forced to place in the empty column.		4 The agent wins.	

Concluding Statements

The goal of the project, to compare the agent, was done. The results suggested that both agents have their strengths and weaknesses. One agent performs very well against human players, and the other one plays well against the first agent.

For the sake of concluding a winner, the MCTS agent was the favorite. It outperformed the Minimax agent in the most important battle: head-to-head. And although it didn't play strongly against the human when it was given little time, it began to look promising with slightly more time. With some optimization, the agent would be able to expand a lot more nodes and run more simulations in less time.

There is also a potential to use a non-random simple rollout policy on the MCTS agent's simulations. One idea is to always take immediate wins and block immediate losses. This would keep the rollout policy quick (although a lot slower) and allow the MCTS agent to not expand many nodes in a path where there is an immediate winning move (same for losing). Will sacrificing so much precious time for the non-random rollout policy be worth it? Or is the random rollout policy with more simulations the better way to go?

One thing is certain, though: humans are better at creating and implementing algorithms for many adversarial games than they are at playing those same games.