Artificial Intelligence: Final Project

Everett Harding, Kenedi Heather, Dean Schifilliti, Dan Seaman

Methodology

For our project, we wanted to implement a concept we had learned about in class but that hadn't been touched upon by our group assignments. We landed on solving simple games, since we covered minimax but never worked with the code ourselves. The obvious choice for a game to solve was checkers, but checkers takes a little bit of time to play and we're millennials, we got bored too quickly. We therefore landed on Connect Four. Connect Four is a popular game with simple rules. It is played on a board with 7 rows and 6 columns, for a maximum of 42 moves per game. The rules are intuitive - connect four of your pieces horizontally, vertically, or diagonally. Do so before your opponent and you win. This game has been fully solved by Artificial Intelligence algorithms.

Rather than write a Connect Four game from scratch, we found a public github implementation from user erikackermann: https://github.com/erikackermann. We built our algorithms on top of the framework that came from that starting code, so we could focus on gameplay logic rather than spending too much time trying to figure out how to display the board (we're not IMGD majors, we're horrible with graphics). Our project examined three different search algorithms applied them to the game; basic minimax, minimax with alpha-beta pruning, and a greedy search with a significantly more complex heuristic.

The main attraction of the framework is its ability to have two search algorithms play each other easily. Additionally, it allows the algorithm to be selected at run time (or even substituted for a human player, if we so desired). Part of the starter code was a basic minimax algorithm which we edited to fit with our testing scheme and to improve its heuristic. Building on that basic minimax, we implemented the alpha-beta pruning algorithm, improving the heuristic and wrapping both in an iterative deepening function to allow us to compare their performance relative to time. Finally we designed a greedy algorithm with a more sophisticated heuristic.

Algorithms

MiniMax

The minimax algorithm we tested was a basic minimax implementation written to expand nodes to a specified depth. Because the nodes at that depth are (usually) not terminal nodes, the terminal values are replaced by heuristic values of the nodes. The heuristic function used in the minimax algorithm is simple:

*#fours * 1000000 + #threes * 100 + #twos*

Where the streaks (fours, threes, twos) are evaluated from the perspective of the current player (either min or max).

AlphaBeta

The alphabeta algorithm we tested extended the minimax algorithm to include alpha-beta pruning on the nodes. To compare this to the minimax algorithm, both were wrapped with an iterative deepening function and limited by time. Due to the pruning, the alphabeta algorithm was often able to search at least one level deeper into the search space than the minimax algorithm. Our heuristic for this algorithm was:

*#max_fours * 100000 + #max_threes * 10 + #max_twos - #min_threes * 100*

Where each streak was calculated with respect to the maximum player or minimum player as indicated.

Greedy

The implementation of a greedy search began when we wanted to develop a more complex heuristic. This desire led us to look into additional situations and certain scenarios that would directly lead into desirable or undesirable situations. The primary scoring system started with a weighted blank board as can be seen below.

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

The purpose of these values was to weight the squares with more possibilities of making a four in a row combination with that cell included. In other words, the value of a given square is the number of possible sets of four that include that square. Once moves are on the board these values begin to have modifiers. There is a set modification to these numbers based on the square's surrounding pieces. Because the only move that can be made next turn are the bottom open squares of each column, those were the only squares evaluated. To modify these numbers we experimented with weights of different occurrences. We developed a list of scenarios to look out for, as follows:

- Creating 2 in a row
- Creating 3 in a row
- Creating 4 in a row
- Blocking opponent's 2 in a row
- Blocking opponent's 3 in a row

- Blocking opponent's 4 in a row

The heuristic would examine if going in a certain space would create any of these scenarios. We experimented with the relative order in which these would be weighted. The chart below displays two options we decided between.

	Offensive	**Defensive**
Priority 1	Make 4	Make 4
2	Block 4	Block 4
3	Make 3	Block 3
4	Block 3	Make 3
5	Make 2	Block 2
6	Block 2	Make 2

The two plans restructure the priority of the choices the system makes, The top two are consistent between both because no matter what placing four in a row (and winning) is the ideal case. After assessing the benefits and disadvantages of the two play styles, we decided on the offensive plan because it causes the opponent to respond to its moves more often, and opens up more possibilities of making three and then four in a row.

Due to the compounding nature of the modifiers, a useful attribute of the heuristic was that it would more heavily weight a move if, for example, it could block a three in a row and make a two in a row. This allowed those moves to be prioritized and establish moves that would be better for the future, despite not actually looking any moves ahead.

After implementing this heuristic we discovered the gap scenario: if there is an opponent piece on the left and the right of an empty square, if the opponent goes in the middle of those it creates a "three in a row". The heuristic originally would not have recognized that was a three in a row scenario, and that situation also applied to "two in a rows" that were a gap away from linking to a "four in a row". By adding recognition of the gap scenarios we significantly improved the capability of playing the game.

Testing

We compared these algorithms by scripting them to run against each other 10 times. We recognize that using a number higher that 10 could have helped with our results - after all, a larger sample size is always better. However, because the minimax and alpha-beta algorithms were time-delimited, we wanted to test multiple decision times. Once we allowed each to take up to 10 seconds for per move

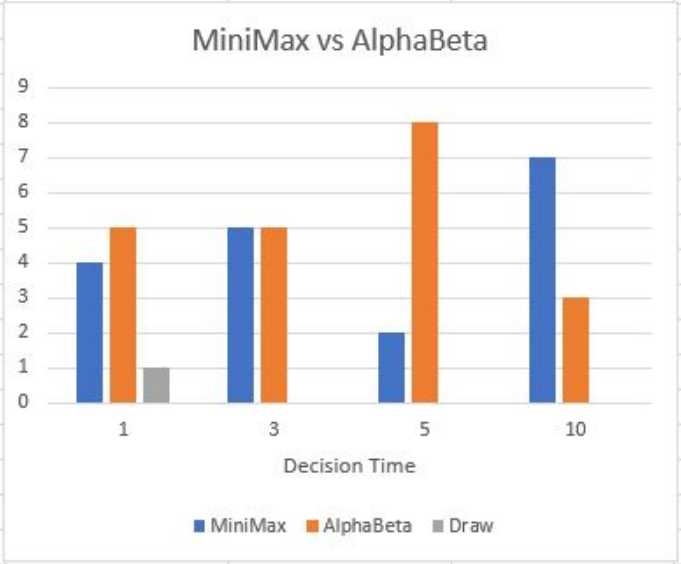
decision, the series of 10 games began to take longer than an hour. Our tester at that point started to fall asleep at his desk, and we felt it best to not extend the testing time any more.

Results

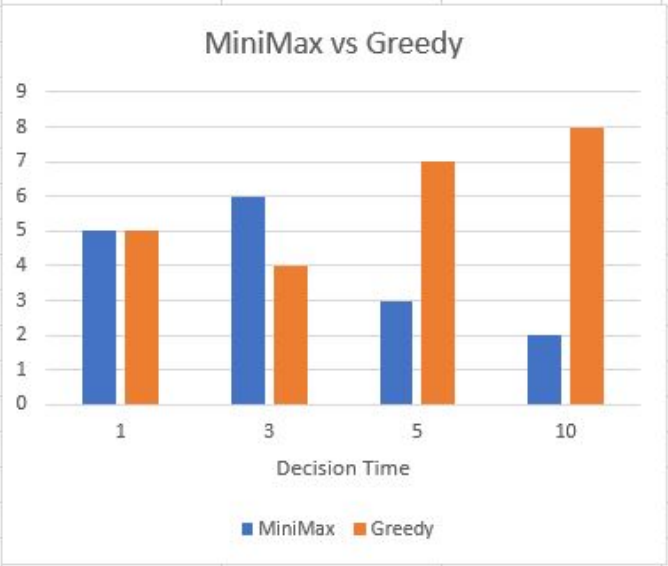
You can watch the algorithms play each other to the soothing sounds of the Halo soundtrack here: https://www.youtube.com/watch?v=MkxqH38ZTZk . We personally recommend playing it on higher speed.

The results for playing each of the algorithms against each other are as follows. We allowed MiniMax and AlphaBeta 1 second, 3 seconds, 5 seconds, and 10 seconds to make their decisions. Greedy, true to its name, was always nice a quick.

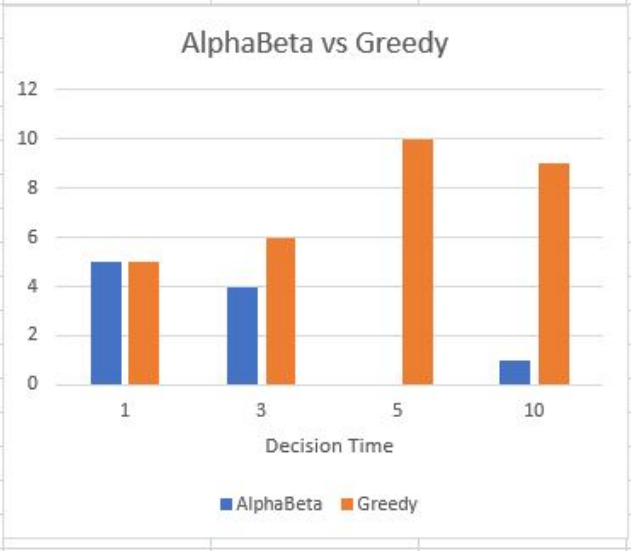
Time	MiniMax	AlphaBeta	Draw
1	4	5	1
3	5	5	0
5	2	8	0
10	7	3	0



Time	MiniMax	Greedy	Draw
1	5	5	0
3	6	4	0
5	3	7	0
10	2	8	0



Time	AlphaBeta	Greedy	Draw
1	5	5	0
3	4	6	0
5	0	10	0
10	1	9	0



While the decision time is short, the algorithms all perform very similarly - this makes sense logically. If minimax and alpha-beta searches don't have time to search the move tree very deeply, they essentially become greedy algorithms.

As time increased, we ran into a problem regarding the simpler heuristic. It doesn't significantly weight early advantageous moves. In other words, while the greedy algorithm would jump at an opportunity to create three in a row as soon as possible, minimax or alpha-beta might make a different move because of a perceived advantage 5-6 moves down the line. This resulted in the greedy algorithm outperforming the other two with longer decision times, five or more seconds usually. We had two thoughts on how we could change this with more time to work on this assignment. The first would be editing the simpler heuristic to weight earlier moves more, or moves at a lower depth in the search tree. The second would be to actually implement our more complex heuristic as the heuristic used by all three algorithms, but regrettably we simply didn't have enough time to integrate it into the framework to that degree.

The next thing to look at is how Minimax and AlphaBeta performed against each other, since they used the same heuristic. At 1 second and 3 second decision times, they're about even - AlphaBeta doesn't have enough time to prune enough nodes to make it more efficient. 5 second decision times appeared to be the sweet spot for AlphaBeta. It could beat Minimax about 80% of the time, and consistently search the tree to about 6 or 7 depth levels while Minimax would only reach 5. At 10 second decision times, we started to hit a normalization point. At that time period, Minimax could consistently search 6 levels throughout the early and mid game while AlphaBeta usually hit 6 as well, sometimes 7. At that point, it was hard to say if MiniMax was performing better that AlphaBeta because the lack of early-move weighting was a factor again, or if it just got lucky for a few of our ten games. Unfortunately, without the time to run more tests at around two hours per 10-game series at that decision time, we can't know for sure.

As a final note, playing against the algorithms ourselves was a lot of fun. Minimax and AlphaBeta could often play very well, but still not well enough to beat any of us unless we were being particularly unobservant. Interestingly enough, Greedy was often a more difficult opponent, and was able to box us into a loss every once in awhile.

Future Work

Given more time to work on this project, both better algorithms and better heuristics could be implemented. We weren't able to evaluate certain strategies that make sense to humans, for example:

- Creating traps for the opponent, where forcing them to block one winning move provides the player with another.
- Creating situations where two winning moves arrive simultaneously

Additionally, other algorithms are potentially better for this game. For example, NegaMax is documented as being a very good choice for playing the game to an almost guaranteed win. Alternatively, training a neural network to play the game could be very interesting, especially comparing what would happen if we trained it against the algorithms or against ourselves, actual human opponents.