Palash A. [pa334], Melody Na [ln244]

CS 4701

A Perfect Solver for Connect-4

Game Introduction

Our project is modelled after the game Connect 4. This game is played by two players on an upright board which has six rows and seven columns of empty holes. Each player starts off with an equal number of 21 pieces. Each player will take turns dropping in one checker piece at a time, and the board grows vertically. In order to win this game, all a player has to do is connect four of their colored checker pieces in a row, similar to tic tac toe. The win can be done horizontally, vertically, or diagonally.

Goals of the Project

The goal of our project was to model, design, and test different artificial intelligence agents for the game Connect 4. We wanted to build a perfect solver that would be able to produce the best move each time, rather than focusing on making an agent that would return a move at a manageable rate and would then be playable against a human player. Since this game is technically a zero-sum game, we decided to build our base agent using a minimax algorithm. This heuristic allows the AI player to check for optimal moves in advance by playing out all possible decision trees and selecting the most optimal move. After implementing the basic agent, we decided to implement an advanced agent which would also utilize alpha beta pruning to make the agent more efficient. The differences in our two agents lie in efficiency rather than correctness, since they all utilize the same minimax heuristic, but with different types of strategies to help shorten the search process.

Game Design

Our game is played on the command line, where the board is printed out after each move by either player. We maintained a state class for the game, so that each move creates a new state or alters the existing state. The state maintains useful information such as the width of the board, the height of the board, and the present board with entries filled in. We decided to use the traditional six row by seven column board. When prompted, the player enters the column that they wish to place their piece in. Depending on whether the move is legal or not, the player may be prompted to re-enter a piece. If the move is legal, then the game state is printed out again in board format in the command line, where the next player can now make its next move.

The board is implemented as an array of arrays, with marks such as 'x' and 'o' indicating where each player has made a move. If the entry is 0, then that position is empty and has not been played yet. The majority of our game functions are implemented in the board (though they are called on from the state class), including checking whether a move is legal, updating the board by adding a position, checking for whether the game has ended and whether a player has won or a draw has occurred, getting all legal positions, and more.

Approach

Many AI implementations for games such as Connect 4 explore the game tree up to a certain depth and use heuristic score functions to evaluate these non-final positions. We chose to build an agent that does not rely on heuristic scores, and would instead give exact scores for final positions. To start off, we had to decide how we wanted to score the positions. We explored many variations including scoring the number of empty positions around a move, likelihood of achieving a win, and more. We decided to score positions based on how many more moves are required to be made in order for the player to win, since it would be ideal to have an agent that wins as soon as possible. Hence, if the agent finds a positive score of 1 for a specific position, it means that the current player can win with his last stone.

Next, we decided on an algorithm called the minimax algorithm. Minimax is a backtracking algorithm that is used in decision making and game theory that helps find the optimal move for a player, assuming that your opponent will always play optimally. With this algorithm, one player is the minimizer and the other acts as the maximizer. The maximizer will essentially try to play the optimal move that enables it to get the highest score possible, while the minimizer tries to do the opposite and gets the lowest score possible. With our game, the agent which we called ‘negamax’ is the maximizer and assumes that its opponent will play with the same strategy as the minimizer. This enables the agent to iterate through the game tree and find the optimal move. This way, the agent actually scores all possible positions and chooses the one that would lead to the quickest win, which is the definition of our scoring function.

The next step in our process was to incorporate optimization with alpha beta pruning. With alpha beta pruning, the agent does not need to fully explore all possible nodes in the game tree to decide on the most optimal move. Instead, it can search at a much faster rate by cutting off branches in the game tree which do not need to be explored. This happens because the agent keeps track of the current best move available, so if it determines that certain branches cannot yield a better score, it will not explore these. In our case, this was helpful because if for example the agent discovered a move that would lead to a possible path to a win with less than 5 moves, then the agent would have no need to explore any other path that would require more than 5 moves. This helped make our agent become more efficient and process decisions at a faster rate than previously with only minimax.

Testing Protocol

In order to perform testing on our artificial intelligence robots, we utilized a variety of test cases to play against each of the robots. We automated the tests by developing a program that would output valid initial states and allow the robot to play the game from there. Positions are inputted based on column number, so for example if we want the player to play the first column, then the third on his next round, and the fifth on his next round, the test would simply be: ‘135’. The test set includes both simple and hard positions, where some positions would be easier to solve when the game is closer to the end. The game depth and number of remaining moves before the end of the game are also mixed up so some games have more moves made already versus some have less. We chose to stick with a testing protocol like this since our agents are *perfect* solvers, which means the execution time for a full board may be too long for full sets of diagnostic tests.

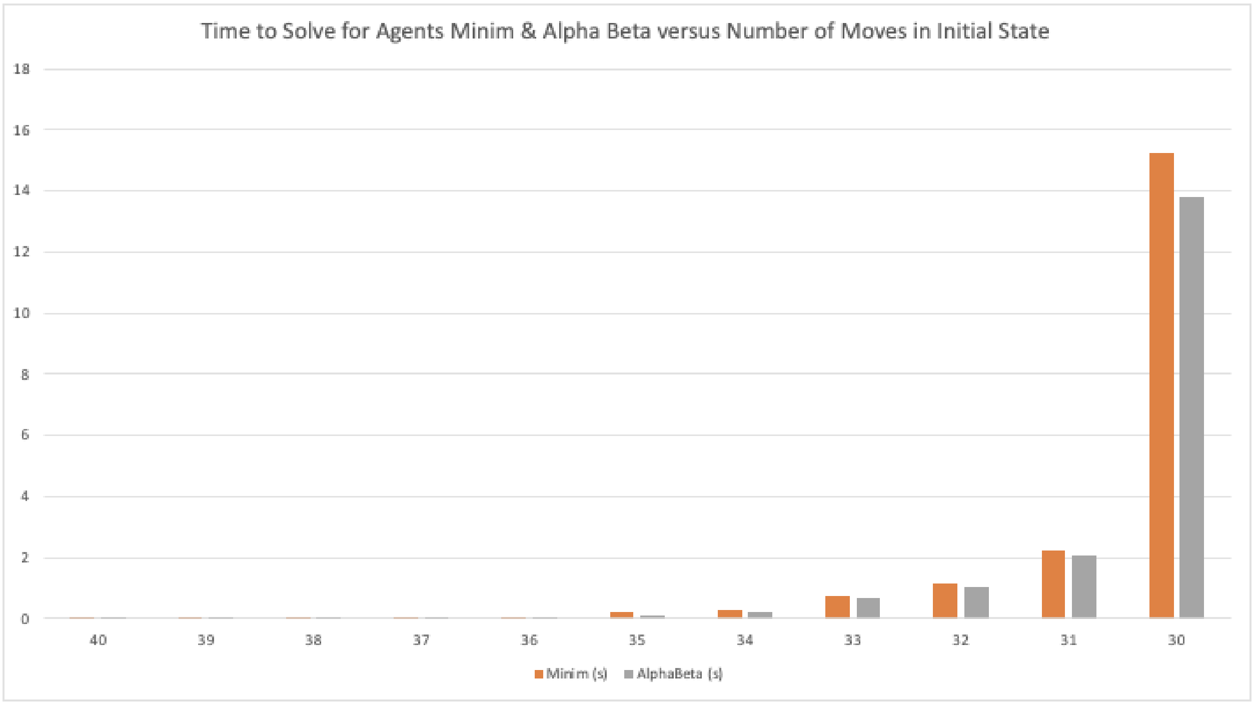
In total, we had 10 different classes of tests, ranging from 30 to 40 initial moves in the game state. The tests also vary from easy, medium to hard, as well as varying whether the artificial intelligence agent begins or ends the game. We wanted to compare the two agents and see how they performed against the tests by examining a variety of traits including accuracy, number of explored positions, and execution time. Our expectation was that all of the robots would have similar accuracies, but the advanced robot would have a lower execution time and higher rate of nodes explored in a given amount of time than the simple robot. This is because the solver we developed solves the Connect 4 problem perfectly rather than relying on heuristics, so we expected that the different robots would have similar accuracies since they use the same minimax algorithm.

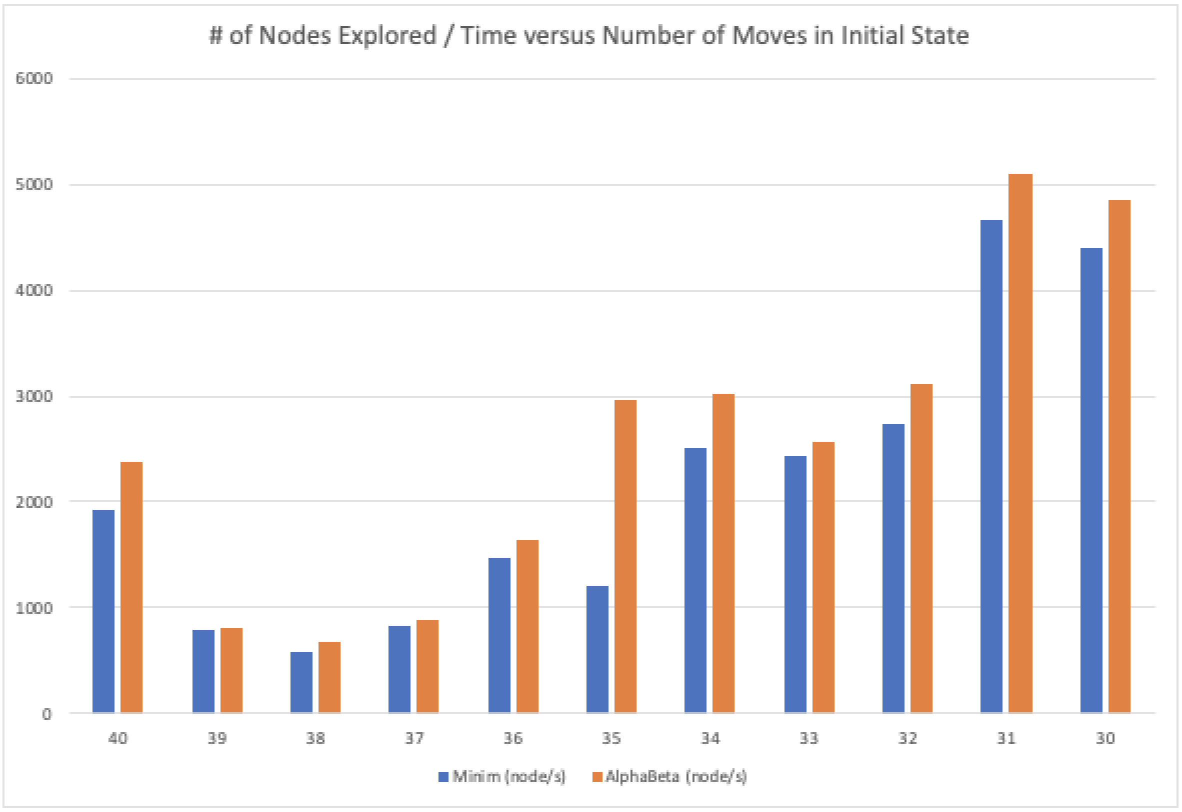
We also expected that the average time to solve would increase when there are less moves in the initial state, since the agents would have a much larger and expansive game tree to explore. We also expected that the average rate of nodes explored in a given period would increase when there are less moves in the initial state, since there are more nodes available to the agents to explore. The detailed breakdowns of the testing and the evaluations of the various robots are included below.

Testing Results

Please note that for the below results, we took the average for all the test cases for each category. For example, in state 40, the time to solve data-point reported for each agent is the average of the time to solve for that specific agent in all of the test cases that contained 40 initial moves prior to the game starting.

State	Minim (s)	AlphaBeta (s)	Minim (node/s)	AlphaBeta (node/s)
40	0.011950075	0.0096949	1924.674113	2372.381355
39	0.030551219	0.029426966	785.5660358	815.5784732
38	0.04438954	0.03850214	585.7235736	675.2871399
37	0.060735039	0.056429085	823.2480101	886.067885
36	0.0821908	0.07346526	1472.184235	1647.03698
35	0.205433775	0.0836232	1202.333939	2953.72576
34	0.258075356	0.214338467	2510.894535	3023.255742
33	0.75268706	0.71250588	2427.303586	2564.189365
32	1.17360224	1.02589552	2726.647829	3119.22602
31	2.266278025	2.0747436	4658.29871	5087.857603
30	15.2413429	13.8332996	4395.413215	4842.80699


Time to Solve for Agents Minim & Alpha Beta versus Number of Moves in Initial State
18
16
14
12
10
8
6
4
2
0
40
39
38
37
36
35
34
33
32
31
30
Minim (s)
AlphaBeta (s)


# of Nodes Explored / Time versus Number of Moves in Initial State
6000
5000
4000
3000
2000
1000
0
40
39
38
37
36
35
34
33
32
31
30
Minim (node/s)
AlphaBeta (node/s)

Testing Evaluation

Based on the results from our tests, we observed that the average time to solve increases significantly when the number of moves in the initial state decreases. For example, for the state with 40 initial moves made, the solving time for both agents was less than 0.012 seconds. For the state with only 30 initial moves made, the solving time increased dramatically to over 13 seconds for both agents. This makes sense because when there are less moves in the initial state, the agents would have to explore much larger game trees whose sizes increase exponentially when the number of moves decreases. This would also explain the shape of the curve for the average time to solve for the agents with respect to the number of moves in the initial state. The existence of a branching factor for the game tree demonstrates that the number of nodes to explore increases at an exponential rate when the number of initial moves decreases, hence the time to explore these nodes would also increase at an exponential rate. This observation matches what we expected prior to conducting the testing.

We also expected that the average time to solve would be lower for the advanced agent in almost all of the cases, which was also true. As observed in the graphs above, there is almost always a noticeable decrease in the average time to solve for the advanced agent. This makes sense because the advanced agent utilizes an alpha beta pruning technique to help prune off branches that do not need to be explored, which in turn can decrease the number of nodes that are evaluated by the minimax algorithm in the search tree. With less nodes to explore, it would make sense that the advanced agent can on average solve the game with much less time than the simple agent.

Something else that we noted that we did expect was that the average rate of nodes explored is generally higher when the number of moves in the initial state decreases. This made sense to us because when the number of moves in the initial state decreases, then there are more possible nodes to explore along each branch of the game tree. With more possible nodes to explore along each branch, it would actually be possible for the agents to explore nodes at a faster rate than when there are more moves in the initial state since the agent could explore more nodes at a faster rate when going down a specific branch, and it could more efficiently prune or make decisions about more nodes at a faster pace.

Another observation we made was that the average rate of nodes explored is better for the advanced agent than the simple agent in almost all of the cases. This makes sense because the advanced agent utilizes alpha beta pruning, which enables it to make decisions about more nodes and cut off specific branches, or exploring specific branches, more efficiently than the simple agent, which only utilizes the minimax algorithm. For example, if the advanced agent is looking down a specific branch that has over 20 nodes, it might be able to prune all 20 nodes at once if it realizes that the maximum along that branch cannot beat the current best score. Then, it would be exploring all 20 nodes at a much faster rate than if the simple agent actually went down the branch and calculated a score for each node. This matches our hypothesis as well.

Overall, our hypothesis was that the advanced agent would perform better than the simple agent in terms of efficiency. We expected that the advanced agent would have a much lower average time to solve than the simple agent because the advanced agent uses alpha beta pruning. Through pruning, the advanced agent can prune useless branches in our game tree, and can hence help optimize the minimax algorithm. The effect of the pruning is that the decision trees, which can be quite large for our game, can decrease in complexity since the agent does not have to look at useless nodes. This makes the advanced agent a much quicker and better solver than the simple agent. In addition, the advanced agent also has a much higher average rate of nodes explored than the simple agent, since it can prune off many useless nodes all at once without the need to actually play them out and derive scores. This would enable it to explore much more nodes at a much faster rate than the simple agent.

Conclusion

In this project, we developed two agents that can perfectly solve the Connect 4 game. Both agents utilize a minimax algorithm, where the agent decides on its next best move by evaluating the scores of the individual positions. For our project, we decided to score positions based on the number of moves left at the end till the win occurs, so a score of 1 would mean that the agent wins with its last piece. The agents assume that their opponents are also playing optimally as well through minimizing the score for the agent. The simple agent uses just the minimax algorithm, while the advanced agent also uses a technique called alpha beta pruning. Through alpha beta pruning, the agent can prune off useless branches that it does not need to explore by maintaining a current best score, and comparing the maximum along the branches with this best score. We expected that the advanced agent would outperform the simple agent in terms of efficiency. Since both agents use the same minimax algorithm, we expected that they would not differ in accuracy. Our testing results demonstrated that the advanced agent did perform better in terms of average time to solve and average rate of nodes explored. The results also indicated that the average time to solve was generally higher when there were less moves in the initial state, which makes sense since the game tree is exponentially larger when there are less moves made initially. We also noted that the average rate of nodes explored was higher when there were less moves in the initial state, which we thought made sense since there are more possible nodes to explore in states that have less initial moves. For the future, we would want to add elements of randomness to see if the accuracy of the robots are affected by this. We would also want to add more algorithms such as iterative deepening to help make the agents faster so that they are actually playable against a human player.

Additional Resources

YouTube Video: https://youtu.be/w3nzmuwLw54

GitHub Repository: https://github.coecis.cornell.edu/ln244/connect4

We had to create a new GitHub repository because I own the old repository and I am currently back home in Hong Kong and cannot log in to my GitHub account since it asks for a verification code sent to my US phone number that I cannot access. We have taken all the files and copied it over to the repository above.