William Cunningham, David Stephenson, Marty Vo

CS 4100: Artificial Intelligence

Final Project Paper

25 June 2021

**Monte Carlo Tree Search Chess AI**

**Introduction:**

For our final project this semester, our team was interested in working with the game of chess. The popularity of chess has certainly seen a resurgence in recent months, so we were excited to see what different sorts of techniques we could apply to make an AI that can contend with humans. After looking through the semester’s material, we decided it would be interesting to apply the Monte Carlo Tree Search (MCTS) algorithm to model the game.

Creating a successful chess algorithm is a difficult task, as games are computationally expensive to simulate since the branching factor of chess is so high. In fact, in chess there are on average 31 unique moves to choose from for most board states. As a result, we tried to focus our development on making our simulations as fast and informative as possible. While we had initially hoped to create our own chess engine to simulate and visualize the board, we quickly realized this was an overly ambitious goal. Instead, we opted to use the python-chess module to handle the board state, validation of legal moves, and endgame behavior. We were eager to focus on the MCTS algorithm so utilizing this module ended up being a really good idea.

Focusing on MCTS, we were excited about the prospect of implementing multiple different rollout strategies to see where we could improve our AI’s win rate. The three strategies we implemented were fully random, attacking, and defending rollouts. Whereas the random rollout will always select a completely random move, the attacking and defending rollouts are more carefully designed heuristics. The attacking rollout first filters its list of moves for any moves that actually capture an enemy piece. Then, it will commit to the move that captures the enemy piece of maximum value. Note that we defined piece values based on common heuristics used by chess professionals. Conversely, the defending rollout will first filter its list of moves for moves that will take a friendly piece out of immediate danger. If there are multiple pieces currently being threatened by the opponent, this rollout strategy will defend the piece of the highest value by moving it to a safe square. One interesting property of our rollout strategies is that the defending rollout consistently caused longer games than either the random or attacking rollout. Intuitively, this makes sense because the defensive rollout will do everything it can to save its own pieces, thus drawing out the game. On the other hand, the attacking rollout strategy will always greedily take opponent pieces when possible, making those games much shorter to simulate. After our first meeting with our TA, Nathan, we came up with the idea of directly comparing our different rollout strategies by simulating games against each other.

**Experimental Results:**

In order to develop a model with better performance, we conducted several different experiments. After initially testing the first version of our AI, we quickly realized that our model would need to at least simulate tens of thousands of games to learn anything of value for playing chess. To increase the learning of our AI, we tested several different simulation values for our rollout strategies. As we increased the number of simulations, our AI performed better. However, it was clear that too many simulations would take an unreasonable amount of time. We needed to keep our original goal in mind, which was to develop an AI that could play against another human. Following this experiment, we tested making our rollout execution more efficient so that more games could be played in a shorter amount of time. To make our rollouts more efficient, we started by cutting off our simulations after a predetermined number of moves. We designed a heuristic to declare a winner in those cases where the game went longer than 80 moves. This heuristic was based on material and would award a win, loss, or tie to whichever side had more valuable pieces left. This was a great optimization for us as we noticed certain simulations were taking 2000+ moves just to eventually end in a tie. So limiting each simulation to a maximum of 80 moves allowed us to run an order of magnitude more simulations allowing our model to see thousands of more example games.

Another thing we experimented with was finding a good *c* value for the UCB1 equation. Our initial implementation with a *c* value of 0.1 faltered as the algorithm chose to repeatedly exploit known values in the tree rather than explore new ones. As a result our simulations were primarily focused on specific paths down the tree, rather than spreading out towards the other branches. After doing some research and setting our *c* value to the square root of 2, we noticed that our algorithm was more equitably exploring the whole game tree.

For our final experiment, we wanted to determine which of our rollout strategies would perform the best so that we could offer the player a challenging game of chess. We decided to focus on our attacking and defending rollouts, and had these two rollouts play against each other. For testing, we had the option of the two AIs playing more games against each but with less simulations and less node depth, or less games but with more simulations and more node depth. We opted to focus on less games as we believed that the deeper node depth and more simulations would provide a more accurate representation of the games played in chess. We had these two AIs play against each other for a total of 24 games, which took a total of 5 hours and 10 minutes. Each AI played half of the games as the opposing color. From our results, we determined that the defending rollout was the better strategy, and opted to use it in our games against human players. The final results from the two AIs playing against each other were 9 wins for the attacking rollout, and 15 wins for the defensive rollout.

**Retrospective:**

We learned a lot as a team from working together these past few weeks. Firstly, we all learned the ins and outs of Monte Carlo Tree Search, the python-chess module, and designing heuristics. We also all gained an understanding of chess notation which was a fun byproduct of working on this project. Next, we also learned great lessons about how to work as a team and produce high-quality software. While implementing, we actually employed a lot of pair programming which was nice to help quickly spot bugs and to spread out the knowledge about what specific functions are responsible for.

Another lesson we learned was the importance of efficiency in rollout execution. Realizing how many thousands of simulated games it takes for an AI to figure out which of its current moves is best was a big takeaway from our project. It helped us understand how coming up with ways to shorten the execution time of the rollouts is critical in creating a successful AI that can make good judgements about the game board in a reasonable amount of time.

If we were to continue this project, optimizing the efficiency of rollout execution would still be the top priority. One option for future optimization could be using dynamic programming to cache state values thus limiting the need for recomputing known values. Another option might be hashing or compressing the MCTS Node’s game state field. Right now, we are storing Node.state as a chess.Board object from the python-chess library. If we were to hash or compress this board into a String representation, it is possible we could increase the number of simulations as each individual Node is much lighter weight. While there is always room for improvement, we are happy with our team's effort to implement and optimize MCTS as much as possible during the brief final project window.

**Appendix:**

All of our project’s source code is contained in the jupyter notebook we submitted. To run the code that sets up the MCTS algorithm, run the first three code cells (one for imports, one setting up the gameplay, and one setting up a node data structure). If you want to run locally, you will want to ensure you have the python-chess module installed. You can find the module [here](https://python-chess.readthedocs.io/en/latest/). At the bottom of our notebook, you’ll see the various experiments that our group ran. The fourth code block is just an initial test, where MCTS is run from a starting board. If you would like to simulate games between the two rollout strategies, you can run the two cells that initialize and then call the ai\_matchup function, inputting the number of games as the third input, and the number of simulations per move as the fourth input. Please note that with the currently preset inputs of 24 games and 500 simulations per move, the cell takes over 5 hours to run. So if you would like to experiment playing the AIs against each other, you should adjust the number of simulations and/or games to be much smaller. You are also free to play an interactive chess game against our AI, by running one of the last two cells with the play\_game method. You can adjust the third input to the play\_game function to change the number of simulations run each turn- having more simulations means the AI thinks longer, but generally makes smarter moves. You will be able to see the wins and losses of each node as the AI ‘thinks’ about its moves.