Chess AI

Tyler Gier and Christopher DiNome

Introduction:

Our project is a simple chess program that allows the user to play games with a variety of AI. The final program supports games involving any combination of players from the following types: HUMAN, RANDOM, MINIMAX, MCTS, and MCTS_PB (progressive bias). Based on the feedback we received from milestone 2, we added MCTS_PB to use progressive bias when selecting nodes for exploration, evaluating the board using the same heuristic as our Minimax algorithm.

We did omit some parts of chess, in accordance with the feedback we received on our initial proposal. Features omitted include:

- The 50 move without advancing a pawn or piece capture stalemate
- The Three-fold Repetition stalemate
- Castling
- En Passant
- Opening books

Topics:

- Minimax with Alpha-Beta Pruning on a new game (1 point)
- Implementation of Monte Carlo Tree Search (3 points)

Team:

We did most of the work for this project together in person.This means that we pair programmed for the majority of the assignment. As a result, we did not allocate out tasks to team members as far as which team member was responsible for which part of the project, because we were always on the same page. In a few cases, we would both talk through adding an additional feature, such as adding yellow squares on the board to denote where the last piece moved to/from, and Tyler typically did these tasks. We are happy with how we structured this project, and working in a group made talking through design decisions and implementing MCTS much easier. This pairing worked out very well for this project.

Instructions:

Our project is written in Processing, so running it is as simple as unzipping the file and running the processing sketch.

1) To setup a game, open the “chess” file in processing. This is the main program.
2) You can change which player type each player is by changing the whitePlayer and blackPlayer values at the top of the file. Each can be one of the following values: HUMAN, RANDOM, MINIMAX, MCTS, or MCTS_PB.
3) For HUMAN players:

    a) Click on a piece you want to move. To change the selected piece, simply click a different piece you control.
    b) Click where you want to move the piece to.
        i) Any legal moves for that piece are highlighted on the board. If a piece has no legal moves, then no spaces will be highlighted.
    c) When moving a pawn for promotion, click on the piece type in the bottom right corner that you want to promote it into BEFORE you make the move with the pawn.
4) For RANDOM, MINIMAX, MCTS, or MCTS_PB, the game will use the chosen AI to play for that player. No player input is required, simply watch the AI make moves.
5) The previous move on the board will be highlighted in yellow making moves easier to track.
6) Some values can be tweaked to experiment with our AI players in the chess file.
    a) whitePlayer - a PlayerType, determines the white player
    b) blackPlayer - a PlayerType, determines the black player
    c) MINIMAX_DEPTH - How deep the MINIMAX AI search will look
    d) MCTS_CACHE - whether or not to save MCTS training data between moves, a failed experiment
    e) MAX_MCTS_DEPTH - how many moves out MCTS will differentiate between when exploring moves
    f) MAX_PLAYOUT_MOVES - how many moves MCTS will playout from a board before deciding the game is boring
    g) TIME_PER_MCTS_EXPLORATION - how long MCTS has to train before deciding on a move in milliseconds.
7) To load a custom board from the SampleBoards class, change the method call in the setup() function in “chess” to the board you would like to show. A picture of each sample board is included in the “SampleBoards” folder. We have marked the line with a comment which is easily searchable by the term “MARK”.

Systems:

1) Random - This AI makes random moves. This was used primarily to test our initial code structure when building the game, but we’ve left it in as an “easy” opponent to play against.
2) Minimax - Our minimax AI uses alpha-beta pruning. This is the same algorithm from the class assignment but refitted to use our conceptual model of a chess board. Ultimately it performs as you’d expect. Its playstyle is relatively defensive and conservative, and it plays to protect its pieces while capturing its opponent’s pieces, often setting up attacks and threats from a long range. This is the most fun of the AI to play against and watch as you can see its “thought process” develop, and what moves it may be setting up.
3) MCTS - This is the standard MCTS algorithm. We followed the 4 stages (selection, exploration, random playout, and back propagation) to build a simple MCTS move selector. In order to make the AI playable and work around our lack of stalemate conditions, we had to make a few modifications. First, the AI is only allowed to playout

games for 10 seconds. We wanted the AI to play quickly enough that you wouldn't get bored and due to the nature of MCTS we are able to stop it whenever. Secondly, we automatically stop playouts that have gone for 70 moves as they are "uninteresting". This value was chosen as the average chess game is somewhere around 47 moves. We wanted the AI to be more playable, however in the process we limit the AI's ability to do more training and gather more data to make a more informed decision. We discuss this choice in more depth in "Failures and Setbacks."

4) MCTS_PB - This is our implementation of MCTS with an extra progressive bias added to the selection evaluation function. We attempted to add this to our algorithm and opted to make it its own AI type. This version of MCTS functions generally better as it gives a bias when exploring to the capture of pieces.

Failures and Setbacks:

Overall we don't feel we experienced any major setbacks, though we did implement some features that we could consider failures:

1) The biggest rabbit hole was attempting to reuse our training data for MCTS. The idea behind this was that, for any given run of MCTS, we could simply move the root of our tree to the selected child. This would then give us access to all of the previously used training data for any future moves from that child. This broke down in 2 parts. The first was that, when playing against a non-MCTS opponent, the opponent was very likely to pick a move that wasn't explored from much, and thus very minimal data was saved. Secondly, when playing against another MCTS AI, the cached data stored simulations that were interesting to player A but may have been uninteresting to player B. However, player B saw the high number of simulations from A's play throughs and used those for exploration due to MCTS's implementation, causing player B to make moves that were good for A. Ultimately, this was turned off. You can turn it on and see for yourself by setting the MCTS_CACHE constant to true.
2) We spent a bit too much time trying to get the MCTS selection heuristic to pick specific moves. It was very frustrating to see it make poor choices, but ultimately that is a result of the poor fit the algorithm has for chess and point 3, below. We spent a lot of time trying to tune the algorithm to perform like a "good" player, even though it's ultimately up to random playouts. We did end up testing our MCTS_PB AI on "check-in-2" sample boards and other examples to verify that the progressive bias held up when moves were very close to a finished game. Ultimately though, in the larger game with more moves, finding a win on random playouts was difficult when the win was multiple moves out.
3) MCTS works best when it has plenty of time to train or more resources. The very successful MCTS algorithms with AlphaGo have a larger training data set due to time, processing power and the inclusion of a neural net. In our application, we wanted the AI to be playable without having to wait a significant chunk of time. Thus the 10 seconds of training was chosen. Ultimately, this is a major hindrance on the system and a drawback of trying to use MCTS in both a single system and single playthrough game, as it didn't really have a lot of time to explore and starts from scratch every game. We think this is the largest failing of our MCTS AI.

Successes:

Minimax, even at a depth of 5, can hold its own against a semi-competent player. We are not very good at chess, but Chris's twin brother Cole who regularly plays Chess on Chess.com traded games with the Minimax AI. Additionally, Minimax's ability to solve check puzzles makes us happy. Minimax is especially good at solving "check in 2" and some "check in 3" puzzles. We have included these boards in the "SampleBoards" file.

We are proud of MCTS_PB for being able to close out a game and make a capture and check. We also dreaded testing MCTS vs MCTS_PB as we were afraid that both AI together could cause the game to go on forever. But we never got stuck in our testing as the AI were able to close out the games with each other.

Overall, we're just happy that it works and that we learned a lot about MCTS and where its faults are. It was pretty easy to read all of the papers throughout this semester and assume that MCTS was this amazing perfect AI algorithm due to its prevalence. However, we learned a lot about its drawbacks and where it does excel from this project. We assumed going into this that Minimax was probably going to be better, and having these suspicions confirmed was validating.

Evaluation:

We ran a number of simulation games with the final AI players, pitting each AI against each of the other AI, tracking stats such as the winner, the total value of pieces of the board at the end, and how long the game took (number of moves). For the sake of time and due to some stalemate rules not being implemented, a game is also considered a draw in data collection if it exceeds 150 moves (this never occurred during data collection). Our data can be found here:

https://docs.google.com/spreadsheets/d/1hQlyGYE6yqpaXKjHPxrGNn5khWPyut-xEqqaUXhSRKY/edit?usp=sharing

AI	Opponent	Total Wins	Total Games	Win Rate	Average # moves	Average Margin of Victory (pts)
Minimax	MCTS	10	10	100.00%	42.30	23.30
MCTS	Minimax	0	10	0.00%	0.00	0.00
Minimax	MCTS_PB	10	10	100.00%	35.10	19.00
MCTS_PB	Minimax	0	10	0.00%	0.00	0.00
MCTS	MCTS_PB	1	10	10.00%	49.00	7.00
MCTS_PB	MCTS	9	10	90.00%	50.67	14.11

Overall MCTS performed poorly for chess. Despite our efforts to provide a bias towards moves with captures (with MCTS_PB), random playouts often ran into dead ends or lost games meaning the node that looked promising ended up seeming like a bad play. MCTS's reliance on random playouts in our model of chess seemed finicky and unreliable versus Minimax's consistent evaluations. This can be very clearly seen in the data, as Minimax defeated both versions of MCTS in every game played. It can also be seen that MCTS_PB put up a better fight

against Minimax - Minimax held a lower average margin of victory over MCTS_PB than MCTS (19 versus 23.3).

Lessons Learned:

1) MCTS is not nearly as complicated as we originally thought. It's a very versatile algorithm and has a lot of strengths with being able to function without a heuristic. When we sat down to actually implement it, we had a "working" version in an afternoon after talking through the steps a lot. It still required a good amount of tweaking.
2) MCTS is really bad when a game doesn't always converge to an end state. When building our MCTS it hit our "you've played way too many moves, it's a draw" condition very frequently.
3) You can lead MCTS to good moves with progressive bias, but you can't make it choose that move. MCTS was very frustrating to watch for chess, as we'd often see it make objectively bad moves at the time or ignore really good moves. This was why we built MCTS_PB, but even that didn't fix the issue entirely.
4) Opening move books are REALLY vital to great chess AIs. Our program includes no notion of an opening move book, and when the AI players go to move and only see boards that - based on our evaluation function - look the same (no captures are made on the first move, for example), then they simply make random moves.