ThreeChess: AI Agent

Jason Ho, 22360143

Literature Review on Suitable Techniques

ThreeChess is a game for 3 players. Three-man chess is actually a chess variant for three players. The game is played on a hexagonal board comprising 96 quadrilateral cells. Each player controls a standard army of chess pieces. The player would need to work out which piece he should move to ensure his victory or survival. Pieces move the same as they do in normal chess, with some unique features like a bishop moving along a long diagonal will change square colors when crossing the center of the board. More of the unique features can be read on the official threechess website. The first player to take an opponent's king wins the game. Standard strategies are still valid like castling, a pawn's initial two-step option, *en passant*, and promotion.

Each state the player is in is evaluated. The evaluation is based on the value of each piece left on the board and the value of each piece he captured. Therefore, it is easy to see that the greedy algorithm based agent can be implemented. They are agents that make decisions in a distributed and asynchronous fashion utilizing only information from the current state in a game. In each turn, the agent would choose the move that may seem to be the best in the current state of the game. However, it is important to know that moving the piece based on the score of the state after moving may not be the best move in the long run. For example, moving the knight of the current player to take an opponent's pawn will yield the best score of the player's state at that current point of time. However, in 2 to 3 turns, that knight would be taken by the opponent's pawn. This is an unfavorable trade as a knight piece is worth much more than a pawn, thus putting the player in a losing situation.

We can extend on this idea by making sure that the agent does not only consider making the best move in its current state but also look into the future to make the general optimal move.

Firstly, is the paranoid algorithm. In this approach, our AI takes a paranoid assumption that 2 of the other chess players are working together to defeat him. They do it by minimizing our AI's evaluation value. In each of their turns, they will each choose a move with the lowest score for our AI. This paranoid assumption also reduces this 3 player chess game to a 2 player game which is very similar to the conventional minimax algorithm. Our AI will be the max player who

would be focused on making chess moves that maximize its score and the other 2 players will be the min player who would be focused on making moves that minimize our AI's score. As seen from the photo below, player one (us) will be programmed to take the highest score whereas player 2 & 3 will be programmed to take the lowest score that minimizes the score of our player. Player one would make the move with a utility value (our score) of 3 and it assumes that Player 2 would make the move with a utility value of 3 instead of 6. This is a form of looking ahead and getting the general optimal move.

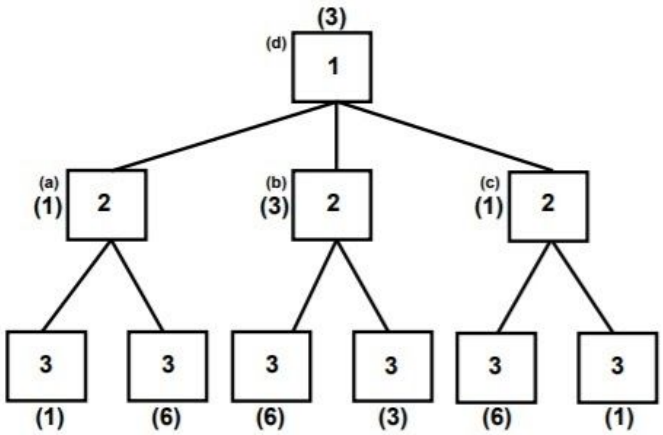
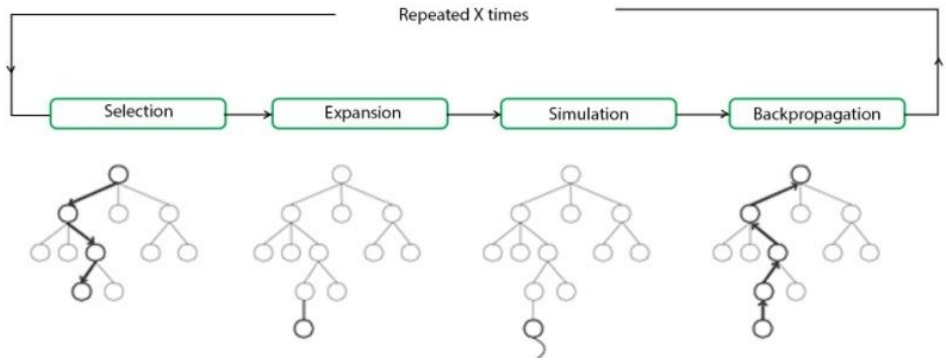


Fig. 2. 3-player Paranoid game tree

(Photo from Zuckerman & Felner (2011))

The Monte Carlo tree search (MCTS) algorithm is another appropriate algorithm to be implemented. It is a computerised mathematical approach for making generally optimal decisions in many decision problems. For example is DeepMind's AlphaGo which was a success after defeating the best human Go players in the world. AlphaGo uses the MCTS and is partnered with artificial neural networks (NN) for maximum performance and efficiency.

MCTS repeatedly performs these 4 steps in order which are selection, expansion, simulation and back-propagation. For the selection step, the algorithm selects the node to begin the search. For the expansion step, if the chosen node is not a node that ends the game, the algorithm will expand that node by adding the child nodes, each representing different board states after making a legal chess move. Next, is the simulation step where the agent will get a state of the board and then make random moves until the game is over where it will return a score to measure if the move was good. Lastly, is the back-propagation step. After our algorithm returns a score or value for that node, we need to update the rest of the tree. This would go through previously visited nodes (the parents) up to the root node. After that, the algorithm starts from the selection step and the whole process starts again.



(Photo from geeksforgeeks)

Selected Technique and Rationale

The structure of the MCTS based agent can be easily applied in the context of ThreeChess due to the idea that at any state, actions can be performed generating many different other states. Also, MCTS is effective for games with high branching factors like Chess as it does not squander computations on all possible branches. The MCTS based agent was the initially selected technique but was not submitted to the tournament because of time and performance issues which we would see later.

Additionally, the paranoid based agent can also be easily applied in ThreeChess by just simplifying the 2 opponents into one opponent against my agent. However, this technique was not submitted to the tournament because while it had satisfactory performance, it took a long time to process the best move to move. The main weakness of this algorithm is the assumption that the opponents' behavior is fixed throughout the game.

As I am not a good programmer, with a useless group partner and limited time, I resorted to just employing a simple agent which was the greedy agent. At each turn, the greedy agent makes a move which maximizes the utility value in its current state. As this technique only considers the AI's current best moves and disregards all surrounding information, the algorithm may not necessarily make the most global optimal decisions towards winning. Therefore, other agents based on more complex algorithms like MCTS and can learn should produce better results than the greedy based agent in theory. This is because all of those algorithms focus on making the most globally optimal decision towards winning.

The aim of my study is to test out if an AI should focus on the present (maximizing the score in the present) or focus on the future (making a general optimal move).

Implementation

Greedy: Firstly, I get a list of all the possible moves for my AI in a 2D array. The first column would be the initial position of the piece and the 2nd column would be the final position of the

piece. Then I try out each of the moves and get its player score using the utility function. It is calculated using:

playerScore = Sum of ((value of each piecetype left on board) * (no. of each piecetype on board)) + Sum of ((value of each piecetype captured) * (no. of each piecetype captured))

It then returns the move that increases the playerScore the most. This encourages aggressive playing to win the tournament. The data structure used is an array.

Paranoid: Like greedy, I used the same function to generate a list of possible moves in a 2D array. Similarly to the conventional minimax algorithm, I assigned the MaxPlayer to be me who would be focused on maximizing my score and the other 2 opponents to be the MinPlayer who would be focused on minimizing my score. The evaluation heuristic I used was:

```
int score = 2 *
board.score(myColour)-board.score(Colour.values()[(myColour.ordinal() + 1)
% 3]) -board.score(Colour.values()[(myColour.ordinal() + 2) % 3]);
```

I multiplied my score by 2 to ensure that I would get a positive result and not negative. The depth I used was 3 because I found that it would give me the most balanced result in terms of time and win performance. Obviously, having more depth would increase the win rate of the agent but would increase the time taken. If the current turn is me, I would try out all the legal moves from the list and do recursion to see the best move the next opponent would make and so on. At the end if I reached a depth of 3 or terminal state gameover, it would return a score which would backpropagate up through choosing whether it is the maximum or minimum. I would then choose the maximum value out of all this (at the root) whereas the opponents would do the opposite. Pruning was done to decrease time taken searching the tree. No complicated data structures were used, but just arrays.

MCTS:

As stated in the current literature, MCTS repeatedly goes through these 4 steps: Selection, expansion, simulation and backpropagation. In consideration of time, the MCTS agent was set to do this for less than 5 seconds. I also created an inner node class to create the MCTS tree and store important values like number of times visited (see code).

Selection: In this process, the agent traverses the tree from the root node using this formula called Upper Confidence Bound (UCB) formula. When traversing this tree during this process, the child node that returns the greatest value from the UCB equation will be one to be selected. Once a child node is found which is also a leaf node, the agent will go to the expansion step.

UCB Formula: Average value + Math.sqrt(2.0 * Math.log(no. Of parentnode visits)/(double) node.getVisits()). This would return infinity/max int number if the node was visited 0 times as dividing by 0 = infinity.

Expansion: "Expanding" the node means trying each move from the legal move list to get to a new board state. This will be the child nodes.

Simulation: For this process, I would take in the Node and convert it into a board state. Then simulation was performed by randomly choosing a move from the legal move list until game over. In consideration of time, I limited it to only choosing 10 random moves before returning a score/evaluation.

Backpropagation: Once the score/value of the node is evaluated, I have to update the nodes up the tree till the root.

Validation & Analysis

Validation of each agent's performance was done by playing 100 games against random-based agents. This was done to eliminate any bias to the agent's position. Validation was measured using the win rates and average total time taken for each completed game. Ideally, I would like to have a high percentage win rate (>=80%) and low average time taken to complete one game (<=20 seconds). Lastly, was a tournament between all of my implemented agents. Each agent was given a reasonable amount of total time to make all of its moves which was fixed at 600 seconds.

MCTS

Agent	Won	Lost	Played	Winrate(%)	Average time taken for each round (s) /(1 d.p)
MCTSAgent	60	20	100	60%	42.3
Random	20	30	100	20%	1.2
Random	20	50	100	20%	1.3

Greedy

Agent	Won	Lost	Played	Winrate(%)	Average time taken for each round (s) /(1 d.p)
GreedyAgent	100	0	100	100	2.5
Random	0	43	100	0	1.0
Random	0	57	100	0	1.0

Paranoid

Agent	Won	Lost	Played	Winrate(%)	Average time taken for each round (s) /(1 d.p)
ParanoidAgent	100	0	100	100%	185.2
Random	0	40	100	0	1.5
Random	0	60	100	0	1.2

Real Inter-Tournament

Agent	Won	Lost	Played	Winrate(%)	Average time taken for each round (s) /(1 d.p)
GreedyAgent	65	0	100	65	1.6
ParanoidAgent	35	10	100	35	78.4
MCTSAgent	0	90	100	0	35.7

From my results, I observed that greedy-based agents have a much higher win rate than MCTS-based and minimax-based agents. Also, greedy-based agents have a much lower average time taken to complete each tournament, meaning average time taken to make each move was low as well. Additionally, it did not lose even once to paranoid or MCTS based agents. The worst result it got was a draw. One big reason for this result was that the greedy-based agent made aggressive moves to maximize its score.

The paranoid-based agent made defensive and cautious moves. As it was assuming that the 2 players were working together to defeat him, it would make more defensive or passive moves resulting in lower aggressive moves to lead to him winning. On the flip side, this means that it would lose less as seen in the inter-tournament (10 games). Average time taken was longer (185.2s & 78.4s) because the agent had to look (depth) moves ahead to decide on the optimal move. Average time taken and time complexity to make a move could be greatly improved by reverting the board to its previous state by using a hashmap or hashset instead of the .clone method. Additionally, decreasing the depth of the minimax would decrease the time taken but it would mean that the agent would look ahead less meaning it might make risky moves. The difficulty was finding the right depth for the agent to have a balance performance and average time taken to make an optimal move. This needs more research.

The MCTS-based agent made weighted aggressive and defensive moves. As the simulation was time-consuming, I ran it to only make 10 consecutive random legal moves which likely decrease the likelihood that the agent makes a global optimal move. Additionally, I had to limit the 4 steps loop to 5 seconds Looking at its win rate against random opponents (60%) and other agents (0%), I think that my simulation part of the MCTS can be improved by using a proper weighted simulation instead of doing random rollouts. Additionally I think that my evaluation function in the simulation is not good. Furthermore, I think that instead of returning a score, I should have returned the win ratio (no. Of wins/no. Of games). From my research, it seems that MCTS-based agents using that would perform better.

My hypothesis was that the agents who used the global optimal move (MCTS & Minimax) would perform better than agents who just used the current optimal move (Greedy). However the results above contradict my hypothesis. As greedy was my best performing agent out of the 3 that I have implemented, I chose to submit it for the tournament. However, in theory, MCTS and paranoid should be beating the greedy agent because greedy based agents only consider making the best move for that current turn. As I am not a good programmer, there were definitely other problems in my implementations of paranoid and MCTS. However, following my suggestions which I stated above would definitely improve those agents and increase the chance of them to defeat the greedy-based agent.

Word Count: 2415

References:

Di PALMA, S. T. E. F. A. N. O. (2014). Monte Carlo Tree Search algorithms applied to the card game Scopone.

GeeksforGeeks. (2019). *ML | Monte Carlo Tree Search (MCTS)*. Retrieved from: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Wikipedia. (2020). *Three-man chess*. Retrieved from: https://en.wikipedia.org/wiki/Three-man_chess.

Sturtevant, N. (2002, July). A comparison of algorithms for multi-player games. In *International Conference on Computers and Games* (pp. 108-122). Springer, Berlin, Heidelberg.

Zuckerman, I., & Felner, A. (2011). The MP-Mix algorithm: Dynamic search strategy selection in multiplayer adversarial search. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(4), 316-331. Retrieved from: http://www.bgu.ac.il/~felner/2011/Journal_mixed.pdf