

CITS3001 Project Report

An AI agent for ThreeChess

William Au (22257005)

A Literature Review of Possible Techniques (20%)

ThreeChess is a three-player variant of the well-known two-player strategy board game Chess. The game is played on a hexagonal shaped board containing 96 squares of alternating colours. Each player has 16 pieces: 1 King, 2 Rooks, 2 Knights, and 8 Pawns. The game is played in Free for All mode, where each player plays for themselves. The goal of the game is to take either of the other two kings. The *first* who captures a King is the winner. Unlike standard chess, check (when the king is in danger) is no longer considered and checkmate (when the king is in check but cannot escape from capture) no longer terminates the game – a King needs to be captured. Points are allocated as follows – the player who captures a King receives 1 point, the player who has their King captured receives -1 (loses a point) and the remaining player receives 0 points.

There are many different techniques which can be used to approach developing an agent to play ThreeChess. Since Chess as well as ThreeChess are zero-sum games as well as games with perfect information, algorithms which are used for ThreeChess can also be applied to other zero-sum games such as tennis and poker.

The most common approach to implementing AI algorithms for Chess is the Min-max search (also known as the Minimax algorithm). The min-max search algorithm is a recursive algorithm which provides an optimal move for a player assuming the opponent also plays optimally. The two players can also be named MAX and MIN, respectively. The algorithm simulates the moves of each player and since both players of the game are opponents to each other, MAX selects the maximised value while MIN selects the minimised value. The algorithm performs a depth-first search for exploration of the game tree and backtracks using recursion.

Minimax implementations use recursive depth-first searches to traverse a game tree and find optimal moves (Rivest 1995). While the Minimax search algorithm can be used to output a perfectly played game, by returning moves which result in either a win or a draw but requires too much computing power due to the sheer number of legal positions in chess (let alone ThreeChess). Furthermore, Minimax is designed for games played by only two players,

The Monte Carlo Tree Search is an alternative algorithm which employs a heuristic to solve a game tree, hence its use in board game agents. The MCTS was originally introduced for computer Go but also used in other games such as chess, shogi and even turn-based strategy video games (Champanand 2014). The algorithm expands leaf nodes and simulates moves until a resulting score is returned, which is then updated to the nodes up a tree to find the optimal move. The more iterations of the algorithm, the more reliable the estimate becomes.

Many AI's have already been created to beat human players at chess among a range of other games, the most recent example being AlphaZero, created by DeepMind. AlphaZero implements a version of the Monte Carlo tree search and maximises efficiency through expert policies and value approximation (Silver, et al. 2018). Furthermore, the algorithm produces better policies and functions by playing against itself with an accelerated Monte Carlo tree search and represents the information with deep neural networks.

Description of Rationale of Selected Techniques (20%)

The selected technique used to produce the agent was the Monte Carlo Tree Search. The algorithm uses an Upper Confidence Bound (UCT) to find promising branches (also known as the selection phase). The formula plays promising branches more often than note but also ensures that no state will be a victim of starvation.

The formula is as follows (baeldung 2020):







$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln t}{n_i}}$$

where:

- w_i = score of the node (based on wins after the i -th move)
- n_i = number of visits/simulations (after the i -th move)
- c = exploration parameter (usually $\sqrt{2}$)
- t = the total no. of simulations for the parent node.

When UCT can no longer be applied to find a successor node, the game tree is expanded by appending all possible states from the leaf node. This is followed by a simulation phase where a randomised game is played out through a “copy” of the game until the resulting state of the game. The algorithm evaluates the state to find out which player has one and assigns a score to it. The tree is then traversed upwards to the root (also known as backpropagation), incrementing the visit score to each visited node and updating the win score for each node if the player for that position wins. These steps, selection, expansion, simulation, and back propagation are repeated for some fixed number of iterations or amount of time to return the optimal move. The higher the number of iterations, the higher the accuracy of the algorithm.

As the Monte Carlo Tree Search generally employs a heuristic to assist in decision process, various heuristics were considered. One promising heuristic assigned each piece to their corresponding values:

-  Pawn = 1
-  Knight = 3
-  Bishop = 3
-  Rook = 5
-  Queen = 9
-  King = 40 (more than all pieces combined) or ∞

The proposed heuristic would assess the pieces in terms of their value and play according to taking/losing pieces of that value. The added third player would prove to be an issue since trading a piece with one player would result in vulnerability to the other. Centre control was also considered, since more control of the centre of the board would result in greater influence over the board, greater mobility of pieces, the restriction of mobility of enemy pieces as well as own pieces moving closer to the enemy King. These heuristics were not properly implemented in the final agent but were considered when developing the algorithm.

Description of validation test and metrics (15%)

The agent was tested through multiple trial runs with other various agents. As the random agent was already provided as part of the source code, the MCTS agent could be easily tested against the random agent. The agent was run against the random agents a total of 50 times with the following results:

<i>Agent</i>	<i>Win rate</i>	<i>Loss rate</i>	<i>Draw rate</i>	<i>Extra Time(ms)</i>
<i>MonteCarlo</i>	<i>1.00</i>	<i>0.00</i>	<i>0.00</i>	<i>291606</i>
<i>Random</i>	<i>0.00</i>	<i>0.54</i>	<i>0.46</i>	<i>300000</i>
<i>Random</i>	<i>0.00</i>	<i>0.46</i>	<i>0.54</i>	<i>300000</i>

The same test was also performed with the Monte Carlo Tree Search algorithm against two “copy-cat” agents which would essentially copy the last move played or play a random move if invalid. Similarly to the results of the random agent, the Monte Carlo algorithm won consistently but the loss ultimately depended on who the MonteCarlo algorithm decided to attack:

<i>Agent</i>	<i>Win rate</i>	<i>Loss rate</i>	<i>Draw rate</i>	<i>Extra Time(ms)</i>
<i>MonteCarlo</i>	<i>1.00</i>	<i>0.00</i>	<i>0.00</i>	<i>285623</i>
<i>CopyCat</i>	<i>0.00</i>	<i>0.64</i>	<i>0.36</i>	<i>300000</i>
<i>CopyCat</i>	<i>0.00</i>	<i>0.36</i>	<i>0.64</i>	<i>300000</i>

The MonteCarlo agents were tested against each other to see the movement of the algorithm better. The results were not very useful in terms of win rate but one interesting observation to note was that the algorithm heavily favoured the use of the knight and the bishop.

Agent	Win rate	Loss rate	Draw rate	Extra Time(ms)	Win By Bishop	Win By Knight	Win By Knight or Bishop
MonteCarlo 1	0.26	0.28	0.46	294307	0.615	0.385	1.00
MonteCarlo 2	0.36	0.52	0.12	294340	0.389	0.611	1.00
MonteCarlo 3	0.38	0.20	0.42	294942	0.632	0.368	1.00

The results from the testing were useful to get an idea of how certain algorithms performed towards one another, but due to the small sample size, the results were not as accurate as would be expected in a proper test with adequate resources.

Analysis of Agent Performance (15%)

The agent can be divided into four separate “phases”: Selection (UCT checking), Expansion, Simulation and Update (Back-Propagation). We can represent the **branching factor** with b , the **depth** of the tree with d , the number of **pieces** on the board is p , the number of **steps** each piece can make is s and the number of **iterations** as i . The analysis can be show in a table as follows:

Phase	Time Complexity	Space Complexity
Selection	$O(bd)$	$O(bd)$
Expansion	$O(ps)$	$O(ps)$
Simulation	$O(i)$	$O(1)$
Update	$O(d)$	$O(d)$

References

- baeldung. 2020. *Monte Carlo Tree Search for Tic-Tac-Toe Game*. 3 October. Accessed October 25, 2020. <https://www.baeldung.com/java-monte-carlo-tree-search>.
- Champanand, Alex J. 2014. *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI*. 12 August. Accessed 10 26, 2020. <https://web.archive.org/web/20170313041719/http://aigamedev.com/open/coverage/mcts-rome-ii/>.
- Rivest, Ronal L. 1995. *Game Tree Searching by Min/Max Approximation*. PhD Thesis, Cambridge: MIT Laboratory for Computer Science.
- Silver, David, Thomas Huber, Julian Schrittwieser, and Demis Hassabis. 2018. *AlphaZero: Shedding new light on chess, shogi, and Go*. 06 December. Accessed October 26, 2020. <https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go>.