CITS 3001- Project

Elliot Khoo – 10421053 Charles Owens - 22244175

Introduction

This project will evaluate the use of and improvement of an AI algorithm in the game of three chess. The focus will be to replicate some of the techniques used in the two state of the art engines AlphaGo, and Stockfish, whilst limited by commodity hardware and training time.

Three chess can be characterised as a perfect information (all the pieces are visible, at all times), free for all, zero-sum strategy game. Each player starts with the same material (chess pieces) and cannot regenerate pieces. Three chess adds extra difficulty to chess by introducing a third player. This introduces some elements of game theory, such as cooperation and betrayal (Miller, 1985). However, the game theory aspects are not examined in this project, as achieving good gameplay is more important.

Literature Review

The authors review the literature for both machine-learning based chess engines and traditional chess engines, focusing on the state of the art for both - namely Stockfish and the AlphaGo family, AlphaGo, AlphaZero, and AlphaStar.

Stockfish

Stockfish is considered a traditional chess engine that uses an alpha-beta search strategy. It derives its strong performance by having significant refinements in performance and heuristics.

Performance is improved by using a very efficient bitboard data structure to the board. The use of a bitboard allows for rapid calculation of moves. This concept is difficult to replicate in three chess due to the nature of three chess. A transposition table is used to prevent duplication (Ray 2014).

The stockfish algorithm itself is an advanced alpha-beta search algorithm that uses a complex heuristic, opening book, closing book, and pruning techniques. For this project, the focus will be on the heuristics used in Stockfish.

Heuristics (Ray 2014)

- Pawn Structure
  - Penalize doubled, backward, and blocked pawn
  - Encourage pawn advancement
  - Encourage control of the centre
- Piece Placement
  - Knights to occupy the centre
  - Queens and rooks to defend each other
  - 7th rank attacks for rooks.
- Passed Pawns
  - Enormous incentives for pawns near promotion.
- King Safety
  - Encourage the King to stay to the corner
  - Try to retain an effective pawn shield
  - Try to stop enemy pieces from getting near to the King

AlphaGo, AlphaZero, and AlphaStar

AlphaGo and its successors represent the best board and strategy game playing algorithms to date. In 2016 AlphaGo beat Lee Sedol, one of the best Go players of all time, and is also stronger than Stockfish. Its successor AlphaZero was not trained on any human matches and was trained entirely through self-play. AlphaZero is stronger than AlphaGo in Go, and can play Chess and Shogi. It is stronger than any human player in all three games. AlphaStar attempts to play the real-time strategy game Starcraft, which has significantly more complexity and time constraints than a board game.

AlphaGo and its successor AlphaGo Zero use a deep Policy/Value network in a Monte Carlo tree search algorithm (Silver, 2017). In AlphaGo, the Policy and Value network are separate, but in AlphaGo Zero, the Policy and Value networks are combined, and another network is used to guide the Monte Carlo Tree Search (MCTS). It uses the policy network to narrow down the search to high probability moves and the value network to improve the rollout policy (Silver, 2017). These networks are all trained by self-play.

The goal of learning techniques is to generate stable and consistent improvement in performance. Learning through self-play can result in agents that become too specialised; an example is an agent learning to play rock-paper-scissors, in which the agent may at first prefer to play rock, then on the next round of self-play paper, and then on the next round scissors resulting in a continuous loop (Salloum, 2019). AlphaStar solves this problem through multi-agent reinforcement learning (Vinyals, 2019), it maintains a variety of ‘exploiter’ agents which continually try and exploit certain strategies in order to ensure that AlphaStar does not become too specialised and forget how to defend against past strategies. An example of an exploiter agent in chess might be an agent that tries to run down the clock and makes only defensive moves.

Although AlphaGo and its successors use heavy rollouts – rollouts guided by their policy network. Heavy rollouts have not conclusively proved to be beneficial (James, 2016), who suggests that smoothness is the key characteristic in determining usefulness. Chess does not have a smooth heuristic, as many parts are binary, such as a piece being taken. Other heuristics might be smoother, such as possible movement.

Method

Rationale for Agent selection

We will look at both alpha-beta and MCTS algorithms that represent the techniques used in Stockfish and AlphaGo, however, make adaptations for resource and time constraints.

MCTS, in combination with the numerous other techniques AlphaGo represent the state of the art. However, with resource and time constraints, MCTS might not be the optimal strategy. Nor would it be possible to replicate the deep policy and value network that AlphaGo has, nor do we have the compute resources to train such networks. Thus alpha-beta may outperform MCTS in our resource-constrained case. One way around this would be to use similar heuristics to alpha-beta in the MCTS algorithm, which would be computationally cheaper and easier to tune.

Alpha-beta is unlikely to be able to search to a significant depth without a large number of performance tweaks. We will attempt to replicate the bitboard data structure for three chess.

Other techniques, such as Q-Learning are also likely to face resource constraints greater than alpha-beta or MCTS. Q-Learning must enumerate both state and actions for the chessboard, which would be impractically large. Whilst other techniques such as reinforcement learning with regret are unnecessary with our simple heuristic.

Validation technique

We will validate the success of each agent through self-play, that is, each agent will play the other agents using the tournament setting provided in the three chess framework. This kind of reinforcement learning is used by AlphaGo to improve its play.

There are several metrics we could use, we could attempt to minimise losses, maximise wins to losses or maximise wins to games played. It is important to look at wins, losses and draws, and the ratios between the three might suggest the next improvement step. For example, a change that results in the same win-games ratio, but increased losses-games ratio would suggest the agent is playing too aggressively and leaving the King unguarded. We focus on the ratio: (won-lost)/played but discuss the other metrics when necessary.

Heuristics implemented

We implemented the following heuristics; these were shared between the alpha-beta and MCTS algorithms. These heuristics were chosen as they are likely to encourage safer play, as an overly offensive player is likely to be penalised in three chess compared to traditional chess.

- Material: Number of piece-points for a player.
- Min material disparity: Minimum points difference between player and opponent.
- Material threatened: Points of material threatened by either opponent.
- King threatened: If King is threatened by either opponent.
- King safety: Encourage King to move to corners of the board.
- Early game cut-off: Allow King to be used in the late game.

Bitboard data structure

One of the key performance improvements in Stockfish is to use bitboards to represent the current state, this aids with deeper searching and hence improved performance. In order to use this paradigm for three chess, we introduce the concept of home colour – that is the side of the board the bitboard is taken from. We then observe that outside of the centre ‘star’ squares and transitions into it, all other squares behave as if a normal chess board. We then hard-code the extra moves associated with ‘star’ squares and transitions between different ‘home’ boards. With how Three Chess is set up, without the bitboard, to get a list of possible moves would require iterating over every piece as an object data type and evaluating their steps, reps, and then finally moves. Constantly making evaluations on these objects requires intense computational power and increases the overall move time. Pre-calculating all of the possible moves using the bitboard and including adequate conditionals to remain within a legal move allows for much less computational power to be required. The only computations required are basic arithmetic for movement and system calls to the pieces to evaluate their moves correctly. This drop-in computational power results in a ~90% reduction in move time depending on the board state. In the figure below, it shows the associated numerical value of the bitboard to each position with the home colour being green.

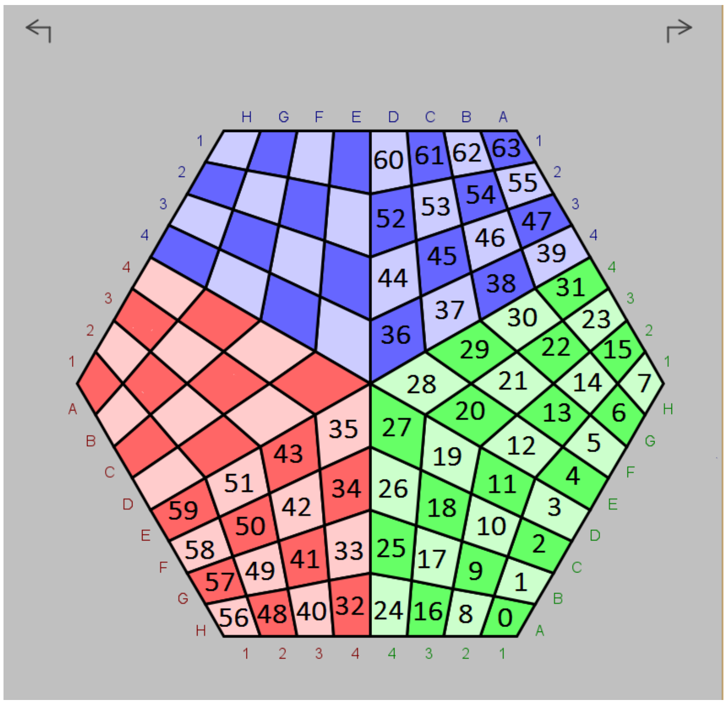

Figure 3: Numerical representation of green's bitboard

Importance of search depth in MCTS

We evaluate using different search depths in MCTS, the search depth in Alpha-beta is time-limited, and not possible to increase it as it increases exponentially. With 50 rollouts taking approximately 1s it exhibits poor play, even against random opponents. In the figure below, it leaves an exposed knight untaken for several moves. This shows that 50 rollouts are far short of the number required to converge or even approximate an optimal solution.

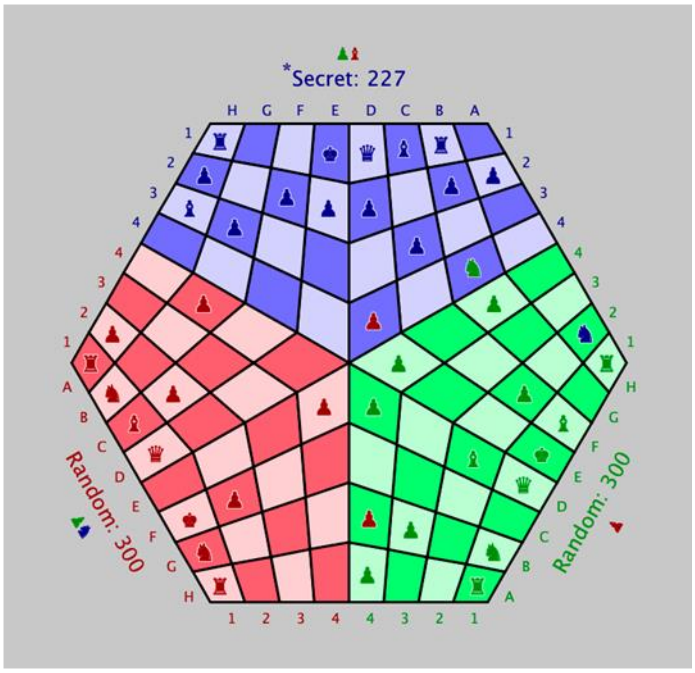

Figure 1: Agent leaves greens exposed knight untaken for several turns

We then modify MCTS to have a maximum search depth of between 10 moves, at which point a heuristic score is returned. This allows us to increase the number of rollouts to 200, which significantly improves play.

Number Rollouts	Search Depth	Wins	Losses	Win Rate
400	8	22	8	73%
400	10	26	15	63%
100	10	25	22	53%
200	10	23	22	51%
50	100	4	33	11%

Figure 2: Comparison of Rollouts vs Depth and Win Rate

Conceptually, this implies that it is better to do a broad search rather than a deep search in chess as moves down the tree are unlikely to happen and shows why the policy network that guides the rollout down likely paths is so important to AlphaGo.

Importance of exploiter agents

In our second test, we looked at the players material count (piece points) vs the difference in material points to its opponents. We played a tournament with 100 rounds with each algorithm randomly selected to play, multiple instances of the same agent could be in the game.

Material	Material difference	Won	Lost	Played	Score
0	100	98	0	100	0.98
50	50	2	51	100	0.49
100	0	0	49	100	-0.49

Here we see that Material difference was more important, however, when we went back to test it against the original, we found that performance had decreased when played against several different kinds of algorithms.

Material	Material difference	Won	Lost	Played	Score
50	50	19	13	42	0.14
0	100	15	16	51	-0.01

We therefore, ran all future tests to include an Alpha-beta (with score heuristic), a pure MCTS agent, and a random agent. This is similar to the exploiter agents used when training AlphaStar.

Using a heuristic instead of score

The heuristics discussed above were implemented, and initial values based chosen based on assumed importance:

1. Material difference
2. Material
3. King Threatened
4. King Safety

We attempted to tune the heuristic, and whilst we achieved better performance than the score heuristic, performance was still below that of alpha-beta. The results of the tuning are included in the Appendix.

Altering UCT

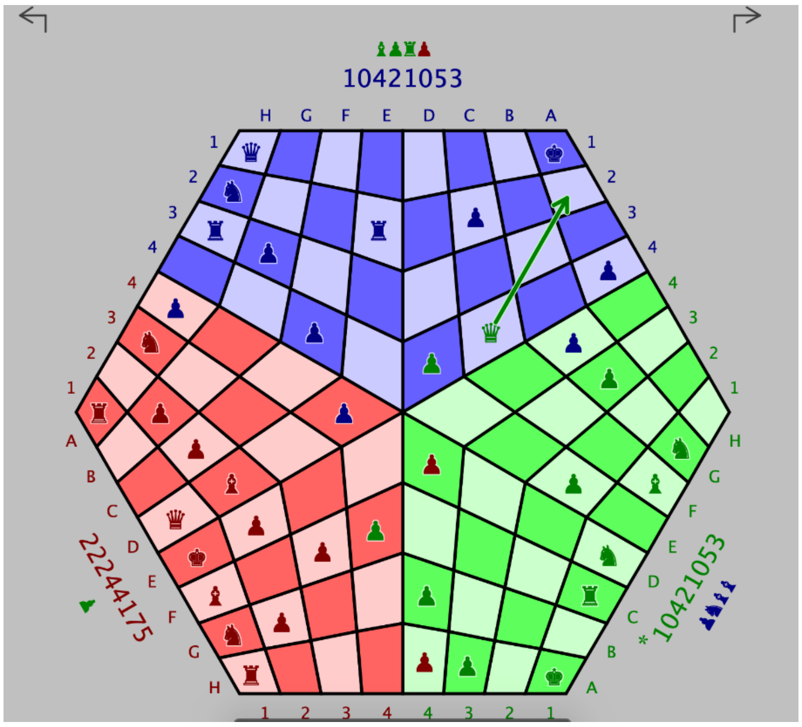

Figure 3: Green sees an opportunity for checkmate, but it is a trap

From the results of the tuning, we see that MCTS Wins at nearly the same rate as alpha-beta, but experiences significantly more losses. Alpha-beta is much better at playing to a draw. This is demonstrated in Figure 3, where the MCTS algorithm assess it can checkmate, but does not see that blue can simply take its queen.

We conclude that UCT is not suitable by itself, in line with James (2016), and that before converging to minimax UCT does not adequately take into account intermediate states. The UCT parameter was modified to include an adjustment for King threatened, material threatened and movement potential.

We only apply this at the first ply under the root node, that is when choosing the move to make, allowing the agent to evaluate immediate threats. This is designed to mimic the strong defensive behaviour of alpha-beta. We finally get a MCTS agent that plays competitively with alpha-beta, but runs in a faster time.

Agent	Win	Loss	Draw	Score
MCTS+UCT mod	35	3	63	0.50
Alpha-beta	41	7	95	0.36
MCTS+Heuristic	41	22	83	0.22
Random	32	161	271	-0.47

Analysis of performance

Time and space complexity

MCTS

Our MCTS algorithm is modified over the standard in that the rollout is depth limited. MCTS stores the board state for every node, and this will depend on how many new nodes are visited every turn. The result of the rollout itself is discarded, and only the score is kept, so this does not add to the space complexity. The time taken to select a node should be much smaller than the time taken to do a rollout, as this is only navigating a tree and selecting the best UCT. Since we have a high exploration parameter and the branching factor in chess is high, we can assume the number of nodes expanded is at worst, and close to the number of rollouts.

Time complexity: O(numRollouts*searchDepth)

Space complexity: O(numRollouts)

Alpha-beta

Min-max algorithm searches each node completely and so its complexity is $O(b^d)$ where b is branching factor and d is depth. If alpha-beta is implemented perfectly, and node selection is perfect then its $O(b^{d/2})$, however, if nodes are selected at random time complexity is $O((b \log(b))^d$ (Russell, 1995). Our algorithm does not have any node selection optimisation and would be closer to this.

Space complexity is O(d) as at each depth only the best board state is remembered at any point in time, and then the alpha-beta algorithm is called recursively.

Conclusion

This project aimed to replicate some of the current techniques used by Stockfish and AlphaGo to the game of three chess, in a time and resource-constrained environment. We implemented alpha-beta search and MCTS, showed the importance of exploiter agents when training to prevent over optimisation. We tested a variety of heuristics, but found that alpha-beta still outperformed until we modified the UCT to take into account immediate threats. We also implemented a version of bitboards for alpha-beta that provided a 90% speedup and implemented a first-valid-move procedure in MCTS that also provided a similar speedup.

References

Steven, J., Rosman, B., Konidaris, G. (2016). An Investigation into the Effectiveness of Heavy Rollouts in UCT, Duke University, https://users.cs.duke.edu/~gdk/pubs/mcts_heavy_ws.pdf

Miller, N. (1985). Nice Strategies Finish First: A Review of The Evolution of Cooperation [Review of Nice Strategies Finish First: A Review of The Evolution of Cooperation]. 4(1), 86–91. Cambridge University Press. https://doi.org/10.1017/S0730938400020852

Ray, C. How Stockfish works (2014). How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine. http://rin.io/chess-engine/#:~:text=Stockfish%20begins%20by%20developing%20a,by%20using%20an%20evaluation%20function.

Russell, S., Norvig, P. (1995). Artificial intelligence A Modern Approach. Prentice Hall.

Sallour, Z. (2019) Introduction to Regret in Reinforcement Learning. https://towardsdatascience.com/introduction-to-regret-in-reinforcement-learning-f5b4a28953cd

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature (London), 550(7676), 354–359. https://doi.org/10.1038/nature24270

Vinyals, O., Babuschkin, I., Czarnecki, W., Mathieu, M., Dudzik, A., Chung, J., Choi, D., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J., Jaderberg, M., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature (London), 575(7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z

Appendix – Heuristics Tuning

We trialled a large number of bots with different heuristic settings. Tuning was conducted using the MTCS agent, Alpha-beta agent, 2 MCTS with limited depth using the built-in score heuristic and several random agents were kept as exploiter agents. All the heuristics were run on a 400 rollout agent, to compensate for the heuristic processing time. "MCTS-800-6" represents 800 rollouts with a depth of 6.

Agent	Mat	Mat Diff	King threat	King Safety	Early Game	Win Value	Win	Loss	Played	Score
Alpha-B							45	3	69	0.61
MCTS-800-6							36	12	72	0.33
MCTS	100	50	-	5	30	100000	26	10	52	0.31
MCTS	50	100	-	5	30	100000	30	14	70	0.23
MCTS	100	100	-	5	30	100000	31	16	74	0.20
MCTS-400-6							21	25	65	-0.06
Random							11	120	198	-0.55

Table 1: Material vs Material disparity

Agent	Mat	Mat Diff	King threat	King Safety	Early Game	Win Value	Win	Loss	Played	Score
Alpha-B							40	5	71	0.49
MCTS	100	50	-	5	30	1000000	37	15	65	0.34
MCTS-800-6							34	12	72	0.31
MCTS	100	50	-	5	30	100000	32	16	75	0.21
MCTS	100	50	-	5	30	10000	28	14	67	0.21
MCTS-400-6							24	14	59	0.17
Random							5	124	191	-0.62

Table 2: Win value

Agent	Mat	Mat Diff	King threat	King Safety	Early Game	Loss Value	Win	Loss	Played	Win %
Alpha-B							32	3	62	0.47
MCTS-800-6							31	8	55	0.42
MCTS	100	50	-	5	45	-1E7	37	11	70	0.37
MCTS	100	50	-	5	15	-1E7	29	13	63	0.25
MCTS	100	50	-	5	30	-1E7	30	14	67	0.24
MCTS-400-6							30	18	70	0.17
Random							11	133	213	-0.57

Table 3: Early game cut-offs

Agent	**Mat**	**Mat Diff**	**King threat**	**King Safety**	**Early Game**	**Win Value**	**Win**	**Loss**	**Played**	**Win %**
Alpha-B							35	8	67	0.40
MCTS	100	50	-1000	5	45	-1E7	38	9	73	0.40
MCTS	100	50	-1E5	5	45	-1E7	36	14	67	0.33
MCTS-800-6							34	13	72	0.29
MCTS	100	50	0	5	45	-1E7	29	16	71	0.18
MCTS-400-6							15	18	58	-0.05
Random							13	125	197	-0.57

Table 4: King threatened by opposing player

Agent	**Mat**	**Mat Diff**	**King threat**	**King Safety**	**Early Game**	**Win Value**	**Win**	**Loss**	**Played**	**Win %**
Alpha-B							49	7	85	0.49
MCTS	100	50	-1000	0	45	-1E7	33	10	66	0.35
MCTS-400-6							30	13	60	0.28
MCTS-800-6							24	12	65	0.18
MCTS	100	50	-1000	10	45	-1E7	30	19	72	0.15
MCTS	100	50	-1000	15	45	-1E7	22	20	63	0.03
Random							12	119	189	-0.57

Table 5: King safety - distance to the corner of the board