Three-Chess

An investigation into automatic agents

Introduction

The purpose of this report is to investigate and implement researched methods for game-playing algorithms that can effectively play the three-player, perfect-information game three-chess. The difficulty of implementing simple algorithms such as minimax is due to the fact that a player (agent) can act co-operatively or non-cooperatively towards any other player, a question aptly addressed by such fields as decision theory and game theory.

Literature Review

A very basic algorithms for game-playing is minimax, which searches a game-tree until a fixed depth is reached or a terminal state is reached and works backwards assuming the other player makes optimal short term decisions. This technique does not work well when more than 2 players are present, as it assumes both opponents are targeting it. For minimax to be a viable option for multi-player games, alteration need to be made.

One such alteration is the max^n algorithm, which generalises minimax to include *n* number of players. It does so by representing all n players' scores as an n-tuple at each node. However, in (Sturtevant, 2002), the shortcomings of such an algorithm are explored. The paper characterises max^n as inefficient as for games such as Chinese-checkers, little to no pruning occurs. A lack of pruning also occurs in games with a high branching factor and no immediate rewards, such as chess. The paper offers the paranoid algorithm as an alternative, similar to the max^n algorithm except it assumes that the remaining n-1 players form a coalition against the player at which point the algorithm proceeds as a regular minimax. This algorithm can reach deeper into the search space. Sturtevant concludes that both algorithms offer advantages over each-other but is highly dependant on the game being played.

In the paper (Zuckerman & Felner, 2011), an algorithm dubbed "MP-MIX" was created as a mix of both max^n and paranoid, in which the agent switches between optimising and defending its score depending on the game-state. The researchers chose to play aggressively, by utilising the max^n algorithm, when the agent lead by a pre-defined margin and to play defensively, by utilising the paranoid algorithm, when losing or winning by a small margin. Their results for the perfect-information, deterministic multi-player game *Quoridor* showed that it was significantly stronger when played against 3 other paranoid and max^n agents.

Another approach to game-playing is reinforcement learning. A widely-used algorithm is the Monte Carlo Tree Search (MCTS), that randomly simulates games until a terminal state, and uses this information to update the value of nodes representing game states. In the paper (Liu & Tsuruoka, 2016), the extension of this algorithm to incorporate a priority search based on the confidence interval of various decisions is discussed. According to the paper, the Upper-Confidence Bound (UCB) tree search is the most common of the MCTS family of algorithms. Lui discusses the shortcomings of such an algorithm and suggest an improvement by altering the level of exploration through an "exploration regulating factor", for a given decision, that is dependent on the amount of simulated plays of both the current state and that decision.

Approach was taken in (Real & Blair, 2016) in which a temporal difference method of learning was modified and applied to multi-player chess. Temporal difference learning is similar to Monte Carlo methods but offers the advantage of not having to reach a terminal state to update its value functions. It does so by iteratively estimating the current estimate based on previously learned estimates. Real and Blair applied "TreeStrap" algorithm, which is an improvement over TD-root that uses temporal difference learning with minimax as a heuristic, with the inclusion of weights for each of the pieces, their positions, and weather they are attacking or defending. The inclusion of weights proved useful, but needed computation time to achieve optimal weight values.

Selected Technique

This report will investigate the Monte Carlo Tree Search (MCTS) method of game playing. From the literature, the most widely used variant of this algorithm included an Upper Confidence Bound (UCB), which takes a statistical approach to searching the tree nodes, by assigning a priority P_n to each node as it is discovered and updated.

$$P_n = \frac{w_i}{v_i} + \sqrt{2\frac{ln(N)}{v_i}}$$

Where w_i is the total score after the i^{th} simulation, v_i is the number of simulations after the i^{th} simulation, N is the total visits of the parent node. This priority formula, also called the tree policy, ensures every node is given a chance to be explored. For example, a node with a low win rate and low number of visits will have a higher variance so will be favoured. When it is determined that most simulations from that node result in failure, the variance will be lower and the node will be ignored in the future.

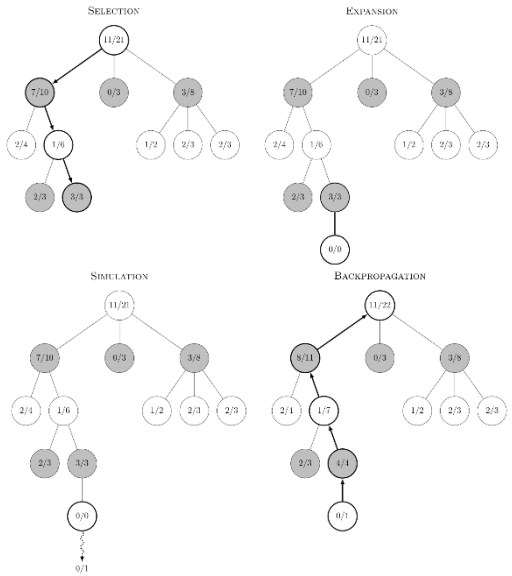

Fig 1: Cycle of MCTS

First the algorithm selects the best child node based on the Tree Policy, in our case the UBCT, and recursively repeats until an unexpanded node is reached. It then expands this node, if it can, into a randomly selected child. In this report, the expansion step will be changed to expand all nodes that capture a piece, and if no such moves exist, the remaining nodes will be expanded. The

algorithm will then select a child node, in this case the best immediate move, and simulate a game. In its pure form, the MCTS runs completely random simulations from each game-state, but this can be improved with a rollout policy. The result is then backpropagated up the tree, and all nodes record their wins from them and all their children, as well as how many times they and their children are visited. This whole cycle is repeated n number of times, denoted $\mathbf{R_n}$ and the child of the root node with the most visits is selected. See Fig 1.

This report will test the effectiveness of two rollout policies to the game of three-chess and rate the performance of such an algorithm, as testing a mixed policy. Multiple children are expanded per expansion step and the selection phase for newly expanded nodes will prioritise immediately profitable moves.

Random Tree Policy

The first rollout policy this report will test is a completely random playout. In theory this playout should take less time and data to simulate, so more simulations can be made. This will be used as a baseline test to test other policies. The playout will follow one rule: make a random valid move every step.

Greedy Tree Policy

The second rollout policy will be a greedy Tree Policy. The standard values for each chess piece will be used. The rules for this roll-out will be

- Make the most profitable move, according to the pieces captured
- If you cannot capture a piece, make a random move

This policy should theoretically take longer to complete the same number of cycles.

Mixed Policy

A supplementary policy that will be tested is the greedy tree policy with a p% chance a random move will be played. This is to encourage the algorithm to expect sub-optimal play from its adversaries, and to address the concern that the Greedy Tree Policy is playing against the same opponent that it is simulating so is better able to predict the outcome to a game against those opponents compared to human or other AI's. Note this is not a perfect solution.

Tests and Metrics

First, we will unrestricted the time limit, to effectively test the bounds of the two agents. The score for a rollout was defined as being 500 for a win, -500 for a loss, 0 for a draw, plus the value of any piece belong to or captured by that team. This was done to prioritise wins and losses, but also include the strength of wins/magnitude of a defeat. The first 3 moves were also picked randomly, as at the root node depth a simulation is not a reasonable approximation for the strength of a move early game. This was done additionally to save time.

A basic agent was created for testing purposes, henceforth referred to as the *Greedy Algorithm*. The greedy algorithm simulates all possible moves and evaluates the one in which its score is maximised from that move. If its score cannot be increased, then it will choose a random valid move. This algorithm was a good approximation for an amateur player, who acts as a maximiser on a Mini-Max game, however the limitation of testing against this player is that strategic thinking will not be tested, as the moves of such an opponent are largely deterministic and greedy. Against random opponents in 50 games, the greedy algorithm had a win-loss-draw total of 44-1-5.

We will test the time efficiency and win-loss-draw percentage of the algorithms, per n number of rollouts, denoted R_n. We will also test the win-loss-draw percentage of the mixed policy vs R_n.

Time Comparison

As previously outlined, the random algorithm is expected to be faster per move. The results show the average move time over a 10-game period[1].

R_n	Average Time (ms)	
	$\text{Rollout}_{\text{Random}}$	$\text{Rollout}_{\text{Greedy}}$
R_{50}	157	695
R_{100}	255	1436
R_{250}	653	3822
R_{500}	1333	6189
R_{1000}	3061	11033

Table 1: Average turn time per rollout number, for the two agents

Win-rate Comparison

The win rate of the two agents against two Greedy Agents was tested. Below are the results of each algorithm for a 100-game simulation[2]. The results are shown as the percentage of games of each result.

R_n	$\text{Rollout}_{\text{Random}}$			$\text{Rollout}_{\text{Greedy}}$		
	W	L	D	W	L	D
R_{50}	0.32	0.4	0.28	0.7	0.2	0.1
R_{100}	0.41	0.32	0.27	0.90	0.02	0.08
R_{250}	0.5	0.14	0.36	0.92	0.04	0.04
R_{500}	0.5	0.1	0.4	0.92	0	0.08

Table 2: Win-Loss-Draw percentages for the two agents, for a n 100 game simulation

[1] Note that the first 3 moves are random (they only expand one node), so these results should be expected to be slightly higher than if all moves had the same R_n. The relationships between the rollout policies is still clear.

[2] 100 games was chosen as a compromise between time and accuracy of results. With more computing power or time more accurate results could be derived

Effect of Mixing Policies

Below shows the result of these three rollouts playing 50 games against each-other. The mixed policy was defined as having an 15% chance of performing a random move. Scoring +1 for a win, -1 for a draw and 0 for a draw and dividing by the games played the final score was calculated[3].

R_n	$Rollout_{Random}$		$Rollout_{Greedy}$		$Rollout_{Mixed}$	
	W-L-D	Score	W-L-D	Score	W-L-D	Score
R_{50}	4-28-18	**-0.28**	23-8-19	**0.08**	23-14-14	**0.18**
R_{100}	7-29-14	**-0.14**	22-12-16	**0.2**	21-9-20	**0.24**
R_{250}	3-29-18	**-0.4**	21-10-19	**0.24**	26-11-13	**0.26**

Table 3: Win-Loss-Draw percentages for three agents playing against each-other, for a n 100 game simulation

Analysis of Performance

Quantitative Analysis

The results are as expected by the theory. **Table 1** showed a weakly linear relationship between the R_n and the average time taken for a move. The random agent outperformed the greedy agent in this respect, and this shouldn't be surprising as with every step of the random rollout O(1) should be expected whereas for the greedy policy O(m) is expected, where m is the number of legal moves for a given step, as every move needs to be indexed before a result is drawn.

Similarly, the result of the effectiveness, measured by win/loss/draw ratio, was as the theory expected. **Table 2** showed that for low R_n, the random rollout policy was ineffective, only winning 1/3 of the time, however as the R_n increased it won a larger percentage of the games to a limit of 50% win rate. The losses decreased from this point. The greedy agent started with a high win rate of 70% at low R_n, and similarly increased rapidly to ≈91% for R_n greater than 100. The losses and draws however remained relatively equal after this point.

Comparing both agents to the mixed agent shows that mixed agent outperformed both agents. **Table 3** showed that with R_{50}, the mixed agent beat the Greedy rollout agent only by incurring less losses. As R_n increased, the wins stayed level and the mixed agent lost less games. For larger R_n the results show that mix rollout improved more than the greedy rollout. While not tested, this mixed agent should theoretically have a lower time per move, as 15% of the moves generated would be random.

Qualitative Analysis

While the MCTS method with UCBT proved successful statistically, it would be less so against a human player. From observing the agent play, it occasionally misses easy opportunities to win, which may be an error in the programming of the agent. The agent shows no long-term strategy besides moving a piece a few moves to then capture the king. This is especially apparent early game where the branching factor is higher, and the end state nodes have not yet been reached. Many wins would not occur had the opponent moved out of check.

The agent showed very little aggression control and often lost valuable pieces early in the game for seemingly no reason. Other times it would make pointless moves and pad the time until it got a draw. It is up to the creator to decide how much to punish losses.

Final Remarks

While the random agent was the fastest, the greedy-rollout algorithm proved to be vastly dominate in terms of game success. The speed of the random rollout can be balanced with the success of the greedy rollout by mixing them. The strength of the mixed agent, coupled with the theory that it is faster, shows that it is superior, at least against greedy agents.

Potential improvements to this algorithm are to dynamically reward rollout simulations differently based on the agent's current score relative to its opposition, introducing an aggression/passivity feature. Additionally, the percentage chance of a random move being made in a rollout could similarly be dynamically changed to reduce bias when there are more pieces such as in early game.

The tree could be reused. Instead of a new root being created every move, the root node could be reassigned if it has already been searched, thus improving memory usage, performance and correctness (Powley, Cowling, & Whitehouse, 2014).

[3] More values of R_n would ideally be tested, but due to time constraints was omitted. The data as is sufficient to draw conclusions from.

References

Liu, Y.-C., & Tsuruoka, Y. (2016). Modification of improved upper confidence bounds for regulating exploration in Monte-Carlo tree search. *Theoretical Computer Science, 644*, 92-105. doi:https://doi.org/10.1016/j.tcs.2016.06.034

Powley, E. J., Cowling, P. I., & Whitehouse, D. (2014). Information capture and reuse strategies in Monte Carlo Tree Search, with applications to games of hidden information. *Artificial Intelligence, 217*, 92-116. doi:https://doi.org/10.1016/j.artint.2014.08.002

Real, D., & Blair, A. (2016, 24-29 July 2016). *Learning a multi-player chess game with TreeStrap.* Paper presented at the 2016 IEEE Congress on Evolutionary Computation (CEC).

Sturtevant, N. (2002). *A comparison of algorithms for multi-player games.* Paper presented at the International Conference on Computers and Games.

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. doi:10.1109/TCIAIG.2011.2166266

Fig 1 – *"Step of Monte Carlo Tree Search."* From
https://en.wikipedia.org/wiki/Monte_Carlo_tree_search#/media/File:MCTS-steps.svg
Permission for use granted here:
https://creativecommons.org/licenses/by-sa/4.0/deed.en
Image was split into two to fit the page