Literature Review

Three Chess is an adaptation of conventional chess to include a third player. It can be usefully some characteristics provided by Game Theory: it is a multi-player, sequential, non-cooperative, non-stochastic, zero-sum finite game (Wikipedia, n.d.). Given these traits, two algorithms will be considered for intelligent agents to use in a Three Chess tournament: Maxn and Monte Carlo Tree Search.

Minimax and Maxn

Minimax is a broadly used decision theory that has been applied to two player games (Wikipedia, 2020). Upon this basis, minimax has more recently been adapted for use in multiplayer games with the Max^n algorithm (Luckhardt & Irani, 1986). Minimax as an algorithm utilises a tree of game states with each node recursively spawning multiple child nodes representing future potential game states based on the single move of a player whose turn it is. Each depth of the tree represents alternating players turns, and these are expanded down until a specified limit is reached. At this point a single utility value of the game state for the player is evaluated, and this is passed back up the tree. As multiple children pass up their utilities, the parent selects the maximum value if the depth represents the agents move, or the minimum value if the depth represents the opponents view. The result is the agent can "see" several moves ahead to determine better rewards that may lie concealed in the future. This algorithm can be further optimised by use of alpha beta pruning, which may be able to determine some children can be safely ignored before they are visited. The Max^n algorithm (herein referred to as Maxn) is an adaptation of the Minimax algorithm to multiplayer games presented by Luckhardt and Irani (ibid). It achieves this by replacing the single utility value with a tuple of values for each player. In addition, instead of alternate nodes that select minimum and maximum values of their children, each node represents a turn of the next player and the tuple is selected which represents the maximum utility value for that player. Pruning for multiplayer variations of minimax such as Maxn has been found to not be effective (Korf, 1991).

Monte Carlo Tree Search

The MCTS represents a different approach for an AI agent to make decisions in a multiplayer game such as ThreeChess (Coulom, 2006). The MCTS utilises a game tree like Maxn, but instead of generating all possible child nodes to a specified depth, it uses a policy (in the most simple case, random) to expand some of the possible paths down the tree to a specified depth, then a utility is backpropagated up the tree. When this utility reaches a node that has been visited before, the utilities are combined and averaged out based on the number of visits. The result is that the immediate children of the root node represent a list of potential next moves and their utilities that are representative of a full expansion of the tree, but without the huge computational cost of a full expansion. Furthermore, the MCTS algorithm has added advantages in that it allows for a valid move to be provided at any

point of termination of the search, and a backtracked utility only needs to be considered when reaching the list of immediate next moves, not at every intermediate node (Lanzi, 2014).

Description and rationale of selected techniques

For the purposes of this report, one Maxn and one MCTS agent have been selected for consideration. This section will discuss the choices made in implementing the algorithms in java, and then two significant choices that impact their operation: how utility values are determined and how time limitations are managed.

Implementation choices

Both Maxn and MCTS Agents require knowledge of potential next moves for operation, and so the implementation choice was made to represent this information in a HashMap data structure. This was chosen primarily for its speed benefits and suitability for the required purpose. The key of this *nextMoves* HashMap is an Integer representation of the start and end positions, as a wrapper class is required for the key and the conversion to an Integer is fast. The value stored with each key is an array of position objects that can be used in performing a move in the game.

The Maxn agent uses this *nextMoves* HashMap throughout its operation. A foreach loop is used to iterate through all potential moves and recursively calls a method that performs the same operation until the specified depth limit is reached and an int array of utilities is backtracked up.

For the MCTS Agent, a *nextMovesNode* HashMap is used. The key used is also an Integer, but the value of the HashMap is a custom node object which holds Position objects for the move as well as additional statistics: variables for games played and games won. However, unlike the operation of Maxn, for MCTS the *nextMovesNode* Hashmap only creates an entry for the range of "first level" moves for the current player from the current board state. This is because the source code can easily be used to rapidly simulate further moves by other players until the end of a game, and the stored history included in such a board object allows direct updating of the node statistics back at the "first level". The result is a more efficient returning of game outcomes.

Description of validation tests and metrics

Testing

Testing was performed using a modified version of the tournament method provided in the source code, and results were printed to log files for collation. The changes made were to institute a rotating roster of starting positions for each player, and no Agent playing themselves in order to prevent resulting statistics being skewed. Tournaments of 30 games length were conducted using a selection of three bots. For each tournament, the bots were

composed of MCTSAgent, MaxNAgent and RandomAgent selected in a random order the possibility of bots playing themselves in one round. For different rounds of testing different metrics were tested.

Metrics

Utilities

Due to the differing operation of the Agents, they assess the utility of a potential next moves in different ways. The utility variable for the MCTS agent is based on the average outcome of multiple simulated playthroughs of games. These are combined by determining the number of games won divided by the number of games played. Consequently, the distinction between scoring zero or minus one is removed, with the intent that the agent will select moves that lead to more assured wins, rather than considering a zero score as less bad than a loss.

The utility variable for the Maxn board state is different because it is restricted to foresight only a few moves ahead, and not to finished game simulations like MCTS. Consequently, this value judgement must be based on board states. A calculation based on assessing the relative location of pieces on the board was rejected for the intensity of processing it would require. Instead the decision was made for a simpler measure of utility by adding the values of pieces taken by the Maxn agent, minus the value of Maxn's pieces lost to other agents. This results in maximizing gains and minimizing losses of pieces based on their values.

In addition, when experimenting with tournaments involving a range of simpler agents, one of the simpler agents stood out as very well performing: the "Grudge" Agent. This Agent utilized a different utility measure to all the others: when calculating utility, additions were only considered when they were scored against the player whose turn was next. The result was an agent that while seeking to minimize its own losses to both players, only focused on taking pieces consistently from one. Consequently, the first research question is if Maxn and MCTS determine utility by only considering gains against one other player instead of two, does this make them perform better?

Time Limitations

Due to the time limit of 20 seconds cumulative processing time per agent per game, the agents require a way of managing this limitation. To this end, the source code provides a useful function for returning time remaining. For both agents, the number of pieces on the board has a positive correlation with the branching factor how much processing needs to be done to return a result, so the early game will be more computationally demanding than the late game. Consequently, a general time policy of gradually reducing available processing time was chosen for MCTS.

For the MCTS agent performing repeated simulations of games played to their end, a rate is used allocate a percentage of the remaining time to processing for this move. An alternate

time policy based on estimating the game length through variables such as number of past moves or pieces left on the board were rejected ass less providing a less reliable strategy. The agent's remaining time was also chosen because it also has flexibility to be applicable independent of specific processor speed and game time limit constraints imposed. This leads to the second research question: what is the optimal time ratio policy for MCTS in ThreeChess?

Maxn on the other hand cannot be so responsive and requires a full depth level of the game tree to be processed then the results back propagated before it can return a move. So, while MCTS can consider its options in terms of a range of thousands or millions of milliseconds of processing time, Maxn only has depth range in the single digits available to it for variable options. Consequently, the decision was made not to test Max according to a ratio, but rather according to a set depth maintained for the entire game. Thus, the third research question is what is the optimal depth of search for a Maxn algorithm applied to a ThreeChess game?

(Results continue on the following page)

Analysis of Agent Performance

Utility Tests

Baseline

Rank	Agent	Won	Lost	Played	Avg
1	MaxN	23	2	30	0.70
2	MCTS	5	13	30	-0.26
3	Random	2	15	30	-0.43

MaxnGrudge

Rank	Agent	Won	Lost	Played	Avg
1	MaxNG	26	2	30	0.80
2	MCTS	3	12	30	-0.30
3	Random	1	16	30	-0.50

MCTSGrudge

Rank	Agent	Won	Lost	Played	Avg
1	MaxN	27	0	30	0.9
2	MCTSG	2	14	30	-0.4
3	Random	1	16	-0.5	-0.5

Research Question:

If an agent determines utility in a ThreeChess game by only considering gains against one other player instead of two, does this make them perform better?

Discussion:
Results testing for this question were mixed. For Maxn, a grudge utility did improve its win rate up from 0.7 to 0.8. However, for MCTS use of a grudge utility caused a worsening of performance from -0.26 down to -0.4. This could be a consequence of MaxN's consistently better performance: in the games where MCTS shifted its focus away from MaxN, this may have allowed Maxn's baseline advantage to dominate. Overall, considering rankings were consistent throughout testing it can be concluded that algorithm choice had a stronger effect on win rate than the use of a utility focused against defeating one or both opponents.

MCTS ratio tests

MCTS01					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN	26	0	30	0.87
2	MCTS01	2	15	30	-0.43
3	Random	2	15	30	-0.43

MCTS02					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN	24	0	30	0.80
2	MCTS02	5	15	30	-0.33
3	Random	1	15	30	-0.47

MCTS03					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN	23	3	30	0.67
2	MCTS03	4	13	30	-0.30
3	Random	3	14	30	-0.37

MCTS04					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN	25	3	30	0.73
2	MCTS04	4	11	30	-0.23
3	Random	1	16	30	-0.50

MCTS05					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN	27	0	30	0.90
2	Random	1	13	30	-0.40
3	MCTS05	2	17	30	-0.50

Research Question:

What is the optimal time ratio policy for MCTS in ThreeChess?

Discussion:

In altering the time ratio policy of MCTS, a trend emerged of an increasing win rate from 10% which peaked at an optimal rate of 40%, then a sharp decline into the worst results at 50%. The increasing win rate is attributable to a relative better use of available resources, before competitive advantage is lost. Interestingly MCTS was not able outperform Maxn at any of the tested rates. This is likely due to the 20 second limit, as informal testing with greater time allowances yielded a win rate more competitive with Maxn.

MaxN depth tests

MaxN 01					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN01	26	0	30	0.867
2	MCTS	4	14	30	-0.333
3	Random	0	16	30	-0.533

Maxn 02					
Rank	Agent	Won	Lost	Played	Avg
1	Max02	25	1	30	0.800
2	MCTS	3	13	30	-0.333
3	Random	2	16	30	-0.467

Maxn 03					
Rank	Agent	Won	Lost	Played	Avg
1	Maxn03	25	0	30	0.833
2	MCTS	4	11	30	-0.233
3	Random	1	19	30	-0.600

Maxn 04					
Rank	Agent	Won	Lost	Played	Avg
1	Maxn04	25	1	30	0.800
2	MCTS	4	12	30	-0.267
3	Random	1	17	30	-0.533

Maxn 05					
Rank	Agent	Won	Lost	Played	Avg
1	MaxN05	24	1	30	0.767
2	MCTS	5	14	30	-0.300
3	Random	1	15	30	-0.467

Research Question:

What is the optimal depth of search for a Maxn algorithm applied to a ThreeChess game?

Discussion:

Testing showed a peak in Maxn performance at only one level of depth analysis. This is could be due to the declining advantage in “thinking ahead” about other players moves, especially in a game with a huge branching factor like ThreeChess. After the first level, win rates fluctuated with a sharp drop at depth 5. MaxN was able to maintain its dominant ranking over all other agents for every depth tested.

Conclusion

In response to the test results, a Maxn Agent was selected for competition in the tournament. A search depth of 3 was chosen, representing a round of moves. Despite this being technically the second best performing depth, this was only due to a one game loss to a random agent, so the difference is reasonably negligible. Also a depth of 1 would effectively be functioning as an inefficient greedy algorithm. A grudge utility will not be used, as it may increase vulnerability to attack by the non-targeted player, and has been shown to provide no competitive advantage over algorithm choice.

References

Coulom, R., (2006). "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In Computers and Games: CG 2006, edited by H. Jaap van den Herik et al., 72–83. Berlin, Germany: Springer.

Game Theory (n.d.) . In Wikipedia. Retrieved October 16, 2020, from https://en.wikipedia.org/wiki/Game_theory

Korf, R.E., (1991) "Multi-player alpha-beta pruning" (Research Note), Artificial intelligence48, 99-111.

Lanzi (2014) "Monte Carlo Tree Search algorithms applied to the card game Scopone"

Luckhardt, C.A, and Irani, K.B (1986) "An algorithmic solution of N-person games", Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86), p.158-162, AAAI Press.

Minimax (n.d). In Wikipedia. Retrieved October 16, 2020 from https://en.wikipedia.org/wiki/Minimax