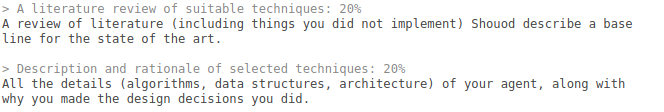
You are required to submit a research report (1,500-2,000 words), and Java source code for one or two agents (**pairs must submit two agents**). The report should include:

* A literature review of suitable techniques: 20%
* Description and rationale of selected techniques: 20%
* Description of validation tests and metrics: 15%
* Analysis of agent performance: 15%



Text

Description automatically generated

# Literature Review

Three Chess is a variant of conventional chess that includes a third player and altered game-board. It is a multi-player, sequential, non-cooperative, non-stochastic (deterministic), zero-sum finite game (Wikipedia). The winner is the player who captures a king, and the loser is the player who has his king captured. Traditional chess engines focus on evaluating as many moves as possible ahead, using a minimax algorithm and pruning unpromising states. There are many ways to evaluate positions in chess, with traditional chess engines for two-player chess including heuristics such as material advantage, piece development, king safety, pawn structure, and threats on the board (chess.com). Many neural-net based chess AIs also perform well, utilizing a Monte-Carlo tree search. The neural-net based ‘Leela Chess Zero’ is the current strongest two-player engine in the world as of May 2019, and is completely self-taught. An advantage of self-tought agents is that they have no reliance on the expertise of their creators at the game.

The addition of a third player not only increases the size of the game tree, but also adds a political element into human games, and some interesting new strategies for AI, such as turtling, sandbagging, and king-making (Pulsipher, 2007). This completely changes how we can use heuristics to evaluate game states. The inclusion of a point-based scoring system means that it can no longer be relied upon that the enemy agents are trying to win first place. They may be trying to avoid last place. Turtling, for example, is playing defensively and letting your opponents weaken each other before attacking. A ‘turtle’ agent might look something like a paranoid algorithm, acting as if both opponents have formed a coalition against the player. A paranoid approach is best in games where ganging-up is easy, and when a brute-force search would be otherwise required (Sturtevant, 2002).

If not using a paranoid strategy, which simplifies both opponents into one, we can instead use a maxn algorithm, a variation of the minimax algorithm, viable for games of any number of players. For a maxn game tree, each node will have a utility for each player, and the move at each node will be the one that maximizes that players utility. ‘Shallow pruning’ is possible in a maxn tree, requiring a lower bound on each player’s score, and an upper bound on the sum of all player’s scores (Sturtevant, 2002).

The optimal weights for the different heuristics in the evaluation function for ThreeChess would be difficult to estimate. This could be solved with a genetic tournament algorithm. The evaluation function is parameterized and agents with different evaluations play against each other, with the functions being mutated or preserved based on their performance (Autones et al., 2004 & wikipedia). A significant cost to using this method is the processing power required for the simulations, thus the time may be better spent on other parts of the agent.

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 well established algorithm for two player games (, which has more recently been adapted for use in multiplayer games with the Maxn algorithm. Minimax 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 Maxn algorithm is an adaptation of the Minimax algorithm to multiplayer games presented by Lockhard and Irani (reference). It acheives 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.

## Monte Carlo Tree Search

The MCTS represents a different approach to processing a multplayer game tree and was first proposed by Remi Coulom etal in 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 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. Some further benefits of the MCTS algorithm are 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). MCTS is the base line for state of the art game playing AI agents.

# 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 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, an additional *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

The overall performance of agents was tested by using a custom tournament altered from that in the source code. Testing tournaments were organized to rotate the ordering of players, so advantages conferred by turn order are cancelled out. Also players were prevented from playing themselves so each player in a tournament plays an equal number of games, and results are not skewed by a player defeating themselves. The custom tournament also allowed for one selected agent to appear in every game, with a group or other randomly chosen agents with different strategies. File input and output was used extensively so a significant number of tournaments could be played, their results stored for later extraction and processing.

## 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 playthroughs of games. These are combined by processing the scores returned by the source code: the number of games won divided by the number of games played. In this way, 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.

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 MCTS, Maxn, and a range of simpler agents, one of the simpler agents stood out as very well performing: 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 how much processing needs to be done, 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 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. For the Maxn agent assessing every move of a tree with a huge branching factor, a declining depth is utilized. An important distinction between how this time policy affects these agents is that MCTS can simply be instructed to run for a number of milliseconds and it can return a valid move. This leads to the second research question: what is the optimal time policy for MCTS?

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 of search of around one to four levels of depth. If this is set too high, all the allocated time could be entirely used up on determining the first move. If it is set too low and considering the board state only one move ahead, then it becomes a less efficient implementation of a simpler algorithm. Maxn also an additional problem: what to do when an optimal move cannot be detected. This will inevitably happen in the early game before there is any engagement between pieces. Consequently, useful time could be wasted on considering a large number of inconsequential moves at this stage. This leads to a second research question: Does Maxn perform better with a delayed start, and if so how much?

# Analysis of Agent Performance

Results of testing