Report Project B

Tien Foong Leong 1025208
Stephen Iskandar 1024303

Supporting work :

- Sebastian Lague: Chess AI https://www.youtube.com/watch?v=U4ogK0MIzqk&t=1024s
- Sebastian Lague: Minimax https://www.youtube.com/watch?v=l-hh51ncgDI&t=575s
- Static Exchange Evaluation (SEE): https://www.chessprogramming.org/Static_Exchange_Evaluation
- Move Ordering: https://www.chessprogramming.org/Move_Ordering

Introduction

For this project B, we were tasked with implementing an AI that can play the game RoPaSci 360. We have decided to use the minimax algorithm with alpha beta pruning and an evaluation function which evaluates each state of the board for the main approach to the project. In the later part of the project, we also added move ordering to further accelerate the speed of the algorithm. We also added a static exchange evaluation (SEE) to further sharpen our evaluation function. Ideas were mostly inspired by Sebastian Lague and concepts from chessprogramming.com.

Main Approach/ Techniques

Conceptualising The Game

When deciding on features to take from the game and its given rule-set, we first thought of features like: Win, Lose, Neutral States, Features to associate with a player, Potential Branching Factor, Algorithmic Performance and Balance, Edge Cases, “What is the best move”, Significant features to use as a point of evaluation.

Ideas and Concepts

Static Utility AI

A utility based AI would be the best as with this game, moves are dictated by their worth and desirability (e.g. will i defeat any tokens by executing this move, will this move land me in hot water in the future etc.), along with an element of uncertainty due to there being another opponent whose thought process is obfuscated behind the referee. Evaluations too can only be accurate when information is fed to us, so this is more of a static reflex AI with some environment simulation added to it, along with a set list of desired, undesired and possible edge case states.

Player Information

We have decided to represent our players with a struct with the following information: player side, number of throws available, furthest throw depth, tokens defeated and number of tokens on board. How the information will be processed and evaluated will be under in the “Evaluation” section of this report.

Branching Factor

RoPaSci is a highly variable game, with 64 hexes, technically on an empty board as the number of throws increase the number of possible permutations will increase too, although it's not as complex as chess, the high search complexity stems from mostly: Throws as you can throw almost anything up to a certain range depending on how much you’ve thrown, Moves depending on the permutation on the board

(swing/slide).
This also brings in issues like move constraints that we want to avoid or have to handle, like if we are surrounded, is there an ally token we can use to swing out of it, or if we are backed into a corner, can we defeat the enemy that's hounding on our token? Another issue would be search depth, as this is one factor that can be influenced not only by the nature of the environment, but also how we process the environment (e.g. how many features we are evaluating).

Handling Input/Output
Due to the simultaneous nature of the game, we have to include safe-guards in our player when updating it. Things such as both players moving onto the same hex, both players doing a criss-cross-exchange (e.g. two tokens on 0,0 and 0,1 respectively, and they both move towards each other at the same time), and as mentioned in the spec sheets, the ideas of coexistence of same typed-pieces on the same hex, and invalidating having three tokens of differing types on any given hex. This can be treated as an edge case, but still must be dealt with due to the unpredictability of the other player as well.What we output and how we will output will be mentioned in the “Evaluation” section of the report.

What is the Best Move?
The quality of the move is usually associated by: How much is that piece worth, The yield/reward the move gives on execution, Assessment of key-features of the environment (board), Move-availability (swing/slide/throw), Open-information from the player (nThrows, tokens on board etc). For example, we can say that if no throws exist, swings are the best move as they provide more mobility, however conversely if we are in a deficit in terms of token composition (e.g. no paper to defeat a threatening rock), a throw would be the best token to use to bring balance in game state. That is why we use an evaluation function that considers the following key-features.

Evaluation
We decided to implement a similar evaluation function that is normally used for chess evaluation. The evaluation can be broken down into 8 parts below.

Token Difference
Token difference will be based on each player’s number of tokens on the board and the token types. We will then assign a value to each of these tokens, and then the value of these tokens will be summed up for each player, which will be then subtracted between them. For example, a neutral and even game state would have an evaluation value of zero.This would be a base guideline for whether we are vulnerable or strong against our opponent.

Number of Throws Available
This part of the evaluation will be calculated by obtaining the difference between the number of throws available for each player. This part of the evaluation is to prevent the evaluation function from encouraging the AI to blindly throw and get a high token difference value. The difference in number of throws between the two players will be multiplied by an arbitrary value and added to the evaluation value. Therefore, two players with the same number of throws will have an evaluation value of zero. Such trends will continue in the next parts of the evaluation.

Depth of Throw
Similarly to the previous part of the evaluation, the depth of throw of each player will be used to evaluate the state of the game, since a player with a deeper throw value will have more options and hence, have the upper hand. The calculation for the evaluation value will be the difference in the depth of throw between both players and then multiplied by an arbitrary value.

We can also possibly execute surprising moves like using our last throw all across the board to defeat a token that has been bothering us for a while, provided that the opponent doesn't move it of course.

Token Clustering
We decided to implement a way for the token cluster to be evaluated, since tokens that are grouped together might be less vulnerable to aggressive moves from the opponent, and also it allows the tokens to make more variety of moves through swings, or sneak by enemies by going through their own tokens via coexistence. Each cluster component is also important as a cluster with only one type of tokens would not give a high evaluation score. We also compare the components of each cluster with their nearest enemy cluster as it gives a value if a cluster is vulnerable. E.g. in one grouping we have on our side, two rock tokens, and on their side, one scissors token. The evaluation value for this cluster will be positive and favour us as rock defeats scissors, and there is a low chance of us dying unless the enemy throws a paper into the mix.

Token Mobility
The board state and the arrangements of each token on the board. This part of the evaluation is very closely related to the previous part token clustering. The number of moves that each token can make will be accounted for here, assigning a higher evaluation value for more options that a token has. This means that a token that is in the middle of the board is valued higher compared to a token that is at the corner/edge of the board as it is restricted in terms of its movement. Similarly, a token that is adjacent to an allied token is valued higher compared to a lone token as it has more mobility and more moves it can make through the swing mechanic.

Token Vulnerability
Token Vulnerability takes into account how likely a token will be defeated by an opponent token. It takes into account the hex surrounding the token and analyses if an opponent token can defeat the token within 1 or 2 moves. It also takes into account if an ally token is nearby (reinforcement counter) so that it can retreat by swinging into a further hex or an ally token can recapture the opponent token that defeated it. This concept is similar to that of a ‘hanging’ piece in chess. This might also decrease the chances of our AI making very questionable moves due to its inability to search through more iterations of the game because of the high branching factor, in other words the “horizon effect”. A way to analogise it is that you’re just seeing the tip of the iceberg (possible move) and assuming that the whole thing (blind execution) when in reality there's a larger mass (more information to consider) below it.

Board Development/Board Authenticity
Board Development refers to the progress of the board after we make the move. Some moves have a high impact on the board, while some moves do not. We want to differentiate these moves by comparing the current board state that we have to the previous board state. We will award more points to a board that has more changes, such as making a ‘swing’ move as a ‘swing’ move is inherently more valuable than that of a ‘slide’ move since the token can cover a greater distance of hexes. On the other hand, we want to avoid moves that result in a similar board state between the current board state and the previous board state, such as backtracking.

Move Desirability
Move desirability takes into account if the move is an inherently good move, it will punish and subtract evaluation value if the move causes the token to move to the edge, to cause it to defeat its allied token, or to cause it to get defeated by an opponent token. This method is used as a safeguard to ensure that the other evaluation factors do not cause such moves to be highly evaluated if there is one.

Minimax and Alpha-Beta pruning
We decided to implement minimax with alpha-beta pruning to choose the best move that our agent can make. We implemented a similar version of the algorithm with the one taught to us in the lecture with several modifications.

Simulate Move and Un-simulate Move
Simulate Move will perform the chosen move during the minimax algorithm and then the evaluation function will evaluate that particular game state and return the evaluation value. Un-simulate Move will be used to undo the chosen move and return the game state to the previous game state before the move is made, including returning the defeated tokens to its original hex. Simulate Move and Un-simulate Move goes hand in hand during the traversal of the minimax tree to find the evaluation value of each game state, with the result of the move we want to evaluate being generated in between these two functions.This allows us to safely consider permutations of the board without altering the original board.

Ordered Moves based on Evaluation
For each depth of the minimax, we generate all the possible moves that particular game state could make. To improve on this we decided to sort all these possible moves by generating these moves first and assigning the evaluation value to each of all these possible moves. We then sort the moves based on a decreasing order of evaluation so that the minimax algorithm will first take in the best moves based on the evaluation first. Since an unordered list of possible moves does not have any guarantee or pattern on the location of good moves, an ordered moves will improve the time taken for the alpha-beta pruning since the algorithm would not need to traverse and evaluate until the last few moves in the list of possible moves generated.

Pre-pruning of Ordered Moves
To further speed up the efficiency, we will also cull the number of moves to evaluate by a certain number depending on how many possible moves are generated.. With our new selection of generated moves all seemingly good as they are ordered best-first, regardless of if the alpha-beta pruning prunes a move that is deemed to be very good for us, our alternatives would not be that bad for us.
This also can decrease the branching factor of MINIMAX as there are less possible options to consider at each depth. For example, if the length of the list of ordered moves is larger than 200, we prune it to an eighth of the list of ordered moves, hence, only using the top one eighth of the list of the ordered moves for the minimax function, this would decrease the search time for our best move.

Removing one Token Type to Consider
Within our generate_possible_throws algorithm, we know that for any given row, there are 3*n [n = row length] possibilities for a possible throw assuming the row is empty. Carrying that forward within multiple search branches, we can encounter a decrease in performance via a high search time. On top of the pre-pruning of ordered moves mentioned previously, we have also decided to prune one token type. We decide which token to not consider by first looking at our weaknesses, comparing the composition of the enemy tokens to ours, (e.g. enemy rock - our scissors), if it so happens that the enemy has no rocks, the value is negative, and we can conclude that it might be better to consider throwing other tokens like paper or rock as the enemy rocks aren't a threat yet. This then decreases our possible throw size per row to be 2*n which scaling up will result in a shorter performance due to less throws to consider.

Restricted Time Limit
To ensure that algorithm completes within a reasonable time, we decided to implement a restricted time limit into the minimax function. Utilising time.time(), we take the time difference from when minimax is first called and compare it to another comparison within the algorithm, limiting our search time to at most 1

second, to which if reached, will immediately evaluate the best move and return the alpha/beta value associated with it depending on which player side we are simulating for in MINIMAX. The impacts this has might lead to the best move not being returned, but considering that we have pre-evaluated, pre-sorted and pre-culled the moves in advance, this shouldn't be a problem.

Additional Techniques Used (move-ordering.py)

Static Exchange Evaluation

Static Exchange Evaluation (SEE) is a term in chess programming where it examines the consequences of a series of exchanges after a given move. A positive static exchange evaluation indicates a winning move. SEE is useful in move ordering, futility pruning. In this project, SEE is used to reduce “bad” captures, and to reduce moves that seem good on the surface, but it is not as good in actuality. This can also help to improve on lower search depths by acting as a mini-extension to minimax. We implemented SEE by combining the following cases below.

Win Outcomes/ Captures Outcomes

We use this to only generate moves that only result in a capture or defeating enemy token, using whatever method that we have. At this point, we do not care about the consequences as we will further evaluate the move to see if it is a ‘good’ capture or a ‘bad’ capture. A basic alpha-beta pruning is executed here, as the number of possible wins wont equal to the number of generated moves in the late-game, depth would not be a concern. A similar evaluation function is used here to evaluate each win case, with the assumed best one returned along with its evaluation value.

Forward Moving

In chess, there is a concept of developing the board and a concept such as ‘tempo’. Similarly, in this project, we want to implement an idea as such by encouraging the agent to make moves that develop the board and moving the token forward. We do not want to waste any moves and hence, ‘tempo’ by moving back the token, when we can make a more threatening move for the opponent. Similarly, we will use this idea to encourage good ‘captures’ which defeat an opponent token, and at the same time, develop the board and threaten other opponent tokens.

More Specific Outcomes

In this part, we will explore more niche cases and outcomes, such as defeating tokens without using up a throw and defeating token at the expense of sacrificing our own token.
Firstly, the throw is considered to be a very strong move as it can highly affect the game situation (as shown in our evaluation function), due to its high variability in move execution. However as we are limited to only 9 of them, (8 technically since our first move is always a throw), throws should only be used when needed as dictated by our evaluation function.

Although, defeating tokens is a desirable move on the surface level of our evaluation function. However, there are many other factors that come into play when using our throw moves, we do not want to create an imbalance of token composition in a way that it gives rise to an invincible token for the opponent and hence, loses us the game right there and then.
Our SEE algorithm takes this into account and slightly pushes non-throw winning moves up, with throws being a possibility that can come through our alpha-beta pruning. Also, defeating a token at the expense of sacrificing our own token, such a concept is not rare in chess. There are many inherently great moves that involve sacrificing queen and major pieces or to set up traps for the opponent. Similarly, we want to implement a similar idea in this project by analysing the consequences if our captures will give us more advantage if we take such sacrifice. On the other hand, we also want to avoid falling into such traps that the opponent has created and hence, we have implemented this method for that.

False Hope

As discussed before under token vulnerability, the “horizon effect” is a big underlying issue for games of high complexity like go or chess, where it is impossible to look into more simulations without sacrificing search time due to the large branching factor. In our case, since we are doing a shallow search, it is easy to mis-evaluate a winning move that can be a trap. (e.g. rock that can defeat scissors, but there is a paper adjacent to the scissors that can pounce on our rock once the scissors is defeated).
Our false hope part of the SEE takes the surroundings of a defeatable/co-exist token into consideration, assigning a high value if it is seemingly safe to execute that move and low if it is a trap.

Testing

Firstly, we developed our evaluation function and tested it against specific board states on basic moves like: Basic Sliding, Moving away from potential defeat, Overturning that danger by producing a winning throw, Choosing between winning swings/slides. Next, we developed our minimax and combined it with the evaluation function and then, similarly, we passed on the board state that we use for testing into the minimax and compared the move decided by the minimax and see if it is logical. We kept repeating this process and through supervised trial and error, slowly tweaked how each evaluation component in the main evaluation and SEE to suit our preferences.

Performance

Evaluation over Search Dept

We have decided to use more of our computational time on evaluation over search depth as although arguably with a larger search depth more moves can be evaluated, simulated and realised, due to the complexity of the game, we placed more emphasis on getting better quality static evaluations on each simulated permutation. With a complex and thorough evaluation function over key-features, we believe this can combat shallow depth runs of minimax as there is succinct information that is processed and evaluated well, hopefully bringing us closer to the true evaluation rather than letting minimax run unguided for a long time. Besides this our move-ordering and possible move quantity culling will also aid us with performance, with best-moves more likely to occur and the bulk of the processing occurring when ordering those moves. Therefore in the early runs, depth will be at 4, but will decrease to about 2, but since we are thoroughly evaluating our input environment well on each simulation, this shouldn't be a problem. With more time to enhance the algorithm, evaluation functions, of course a better agent can be made.

Alternatives

Use of Machine Learning

Initially we wanted to use machine-learning tools like tensor-flow or sckit-learn to further narrow down and evaluate the quality of the evaluation function, by incrementing/decrementing certain values to the overall evaluation at those certain key-features, however due to time constraints this was not feasible, even if it could improve the performance of our agent.

Monte-Carlo Search Tree

Another option to machine learning, where we let the agent play against itself over multiple simulations, this time with the moves randomised and the best-moves stored. The quality of the move is dictated by the ratio of number of wins to the number of children simulated after that move is executed. With the final behaviour of the agent reflected by what it sees as a good move depending on a move-bank in storage. This could improve our AI as more varied outcomes are encountered, evaluated and given enough time over multiple runs, can achieve performance.