Edwin Zhu 911277
COMP30024 Artificial Intelligence Kailiang Zhu 1157777
Anastasiya Bogoslovskaya 1377798

Project Report: AI Agent Playing Infexion

1. Approach :

For this project we have chosen to implement an enhanced Minimax algorithm, with alpha-beta pruning to help decrease the branching factor and create a more efficient algorithm. This particular algorithm was chosen as opposed to the alternative we explored; Monte Carlo tree search (MCTS), mainly due to the optimality of Minimax. We believed that the branching factor would be quite high and thus we decided to create a good heuristic to make it work efficiently.

Our first approach to get the algorithm working and exploring states was the total power of the current player minus the total power of the opponent with the aim of creating a larger power difference. The idea is to make a move that puts you in a better position with more pieces and possibilities to spread and capture. This proved to be good against random agents but didn't take into consideration high power pieces and pieces that are placed next to each other.
We found these setups to be valuable in the game so to enhance the evaluation function we wanted something to capture the board state better.

Evaluation function:

The evaluation function we used considers the 'coverage' of pieces on the board for each colour, taking into account the power of the pieces on the board and which player just made a move.

Coverage gives an indication of what squares on the board are controlled by which player, giving us a capture exchange outcome through the potential spreads and reach of each piece on the board. A piece covered more times by Player A than Player B suggests Player A can overpower the opponent in securing that piece through a capture sequence/exchange.

We evaluate a board state by providing a score for that state using the following implementation:

1. **Generate a constant dictionary** where the keys are all combinations of (position, power) pairs for:
    - Every position on the 7x7 board, and
    - Every power from 1 to 6.

    The output of this constant dictionary is what nodes are covered, and how many times it is covered.
2. How many times each **colour covers each position** on the 7x7 board can then be quickly calculated by looping through every piece on the board and using the constant dictionary.

3. Then, for each piece (colour, power) on the board, the **power of that node is 'won'** by whoever attacks or defends it the most, where the defender is 'colour' and attackers are enemies of 'colour'. In the case of a tie in number of attackers and defenders, the power of the node goes in favour to the defender, as the defender will come out on top at the end of the capture exchange sequence.
4. The difference of the **sums of these scores** over all pieces on the board for each colour is the unadjusted evaluation.
5. To account for whose move it is, the final **evaluation is adjusted**. The player who just moved (opposite to the player currently to move on the current position), will suffer a penalty to their score by subtracting the highest power node on the board that the player who just moved attacks more than the player currently to move defends. This adjustment is based on the idea that the player currently moving can react to this threat, and potentially not lose their highest value node.
6. In the case of a **tie in evaluation**, a secondary evaluation is used to rank alternatives. The secondary function evaluates by the overall differences in coverage across the board only for where there is a piece. This aims to favour positions where the player's nodes connect with other nodes more, providing overall better potential in terms of attack and defence.

Some examples of nodes, the coverage of that node by each colour, and who 'wins' the node are shown in the table below.

In the table, if we assume those are the only 4 pieces on the board, the unadjusted evaluation is clearly in blue's favour. It is 4 - 3 + 2 + 1 = 4 in blue's favour.

(Colour, power)	# Enemy Attackers	# Player Defenders	Winner for eval.
(RED, 4)	3 (blue coverage)	2 (red coverage)	4 in BLUE's favour
(RED, 3)	3 (blue coverage)	3 (red coverage)	3 in RED's favour
(BLUE, 2)	2 (red coverage)	3 (blue coverage)	2 in BLUE's favour
(BLUE, 1)	4 (red coverage)	4 (blue coverage)	1 in BLUE's favour

An additional consideration is used to give an evaluation of a position. If no captures can be made at a current position, the difference in powers is used as an evaluation as this is considered a relatively stable evaluation. This is also used to return an evaluation quickly if a stable position is found during the minimax search.

Coverage figures generated for the constant dictionary

For a particular piece (position and power) and in all 6 directions, we consider how many times it will cover other cells. We consider 3 nodes ahead in each direction, regardless of the piece's power. This is because the 3 nodes in the opposite direction will be covered by moving in the opposite direction instead. E.g. 4 nodes ahead in a direction for a piece with power greater than 3 is simply 2 nodes backwards in the opposite direction.

Here are the values a node is covered based on its distance in a particular direction from the original node, as well as the original node's position and power.

Power \ Nodes ahead	1 node ahead	2 node ahead	3 node ahead
Power 1	1	0	0
Power 2	2	2	0
Power 3	2	3	2
Power 4	2	3	3
Power 5	2	3	3
Power 6	2	3	3

Why do higher power nodes have different coverage of nodes in its spread path?

Higher power nodes can cover certain nodes multiple times as they can not only attack a node via direct spreading but the spreading also generates nodes next to the attacked node, which can then spread back onto the attacked node, resulting in covering a particular position more times than just once. Our coverage values assume an empty board for simplicity, where it doesn't take into account if the player has other nodes in the path of the spread that can be used to spread in different combinations for a greater effect.

Getting successors during the minimax search:

For our agent to make a move in its allocated time we choose our successors based on a fixed depth and breadth. It wasn't computationally feasible to explore all of the options at a certain depth even with alpha beta pruning. This high branching factor led us to rank the successors and only explore ones with high potentiality moving forward in our successors tree.

In choosing what child to consider as a successor, we first get all the possible successors of the board state, by performing all possible spreads in all six directions for each piece of the player's colour and any spawns available. A greedy component is first added as a potential successor. This simply includes the child position with the best evaluation in terms of power, and aims to check for obvious capture sequences. We then consider the children with the top evaluations for the player, as they are considered to indicate a better position on the board. Note that the children considered are also based on the secondary evaluation function in cases where there is a tie in the primary evaluation. Spawns at a position that is not covered enough by the player compared to the opponent is not considered in order to save time, as it suggests the player does not have enough defenders for that spawned piece.

The total number of children considered at each branching step is the BREADTH constant we set in the program due to time constraints.

2. Performance evaluation :

This game agent was tested against simple hard coded edge cases, random agents and itself (with different BREADTH and DEPTH values for the minimax search) to see what would lead to the best performance. The evaluation function works well enough to beat random agent as well as the hard coded examples. It was found that if the agent played against the exact same agent, it did not end in a draw as expected but rather depended on what the Breadth and Depth values were.

Through testing it was found that the best values for BREADTH and DEPTH were depth = 4, breadth = 6 and depth = 3, breadth = 11. Of course the higher the breadth and depth the more nodes we explore the better chances we have of finding a better move so we didn't go above depth 5 as breadth got very small due to the time limit. Our agent depth = 4, breadth = 6 beat depth = 5, breadth = 4 which were the highest breadths in our time so then the testing was between 3 and 4.

As the depth got larger we did not see a better performance by our agent as it was trying to anticipate too many moves from the opponent who wasn't playing optimally. So from the tests we thought depth 4 was sufficient and maximising breadth we got 6.

Overall, through the tests we have conducted as well as the greedy approaches on gradescope, the agent has been optimal in being the winner.

A drawback of our minimax algorithm is that it does not consider all combinations of captures available in a position, due to limited depth and a non-perfect evaluation function when choosing children. This may mean the agent:

- does not go into a capture sequence when it is beneficial for the agent,
- goes into a capture sequence where it loses out at the end of the sequence, or
- does not parry the threat of a detrimental capture by the opponent.

Another small drawback of our minimax algorithm is we assume the other player is also smart so we don't take risks and make moves that could easily capture and win faster against a completely random agent.

3. Other aspects

Specialised Data Structure:

We created a board class to keep track of the internal state of the game for the agent along with other functionality like, spread , spawn, counting pieces and getters.

A Minimax class was added to do the calculations for minimax and for ease of call in the main agent function.

Coverage.py was created as a utility for the minimax class to calculate the scores of a specific state.

Choosing a depth and breadth:

One optimization we added was making BREADTH and DEPTH adjustable. In a minimax implementation, the choice of an odd or even depth depends on the specific implementation and the nature of the game being played. We wanted to maximise the deepest search possible without timing out and not exploring enough breadth. This deviates from the vanilla minimax algorithm and allows us to test and adjust our minimax tree to explore promising states according to the coverage evaluation.

In the end we opted for an even depth as this gives the opponent the last move and may better account for the opponent's ability to counter-spread during capture sequences.

4. Supporting work

To assist in evaluating if the agent was good we tested it against a spectrum of other agents.

A random algorithm agent was made that made valid but random moves. This was to test if our agent was able to beat something with no strategy and see what moves it performed to beat the agent.

A greedy agent was created that followed a simple set of instructions. It would generate a dictionary of valid moves and play thoughts valid moves on a temporary board to evaluate the score of the state (calculated by total power of a colour of that state - the total enemy power + any piece that was not next to the enemy - any piece next to the enemy) and pick the highest value. We used this test to see if our algorithm was able to strategize and attack and defend against an agent with no foresight and took pieces when possible.

We then played against our first minimax agent whose evaluation function was based on the highest power on the board and the new strategy won every time.

Tests were also hard coded, for example spawning pieces in a specific order to test how greedy our own algorithm was.

Finally we created agent2 which was the same as our agent but had different depth and breadth values in order for us to see if certain breadth and depth was better than another. We then vigorously tested different depths and breadths to see how they were affected when playing itself, how long moves took and who had the best strategy. This showed us the influence of breadth and depth for our agent.