COMP30024 Artificial Intelligence
Project Part B Report

Student names: Liangdongfang Xu (1174154)
Jinrun Ji (1394227)

Describe Your Approach

Search Algorithm Chosen

By analyzing the insights of the Infexion game, we figured out that each state of the game is observable (perfect information) and it is a zero-sum game, which means it is impossible to cooperate with the opponent but you need to defect it. Therefore, the Minimax algorithm has been used to find the optimal move for each state. Considering the time and memory constraints, the α-β pruning algorithm is implemented as well.

Modifications to Existing algorithm

Since Minimax algorithm provides a way to find the optimal move for each state, it is chosen reasonably as an initial attempt for solve this chess problem. Providing with an suitable evaluation function, score value for each possible action is calculated and saved into a list called children. By implementing this algorithm as recursive functions with a fixed DEPTH parameter controlling the max depth of the recurring process, it is proved to minimize the opponent's benefit and maximize the player's gain simultaneously. However, when the depth increases, the computation time increases tremendously. By searching for possible solutions for similar chess games on the internet, we found that mostly α-β pruning is used to reduce the function searching time. Initially, the α will be set to negative infinity and β will be set to infinity. While the searching processes, αwill be updated to a larger score than α for the max player while β will be updated to a smaller score than β for the mini player. In another word, αrepresents the player's optimized score and β represents the opponent's optimized score, while the player prioritizes the missions to maximized its own gain and the opponent prioritize the missions to minimize the player's gain.

Features of Evaluation function and Strategic Motivation

At first, we came up with three ideas of the variables that could be used in the evaluation function. By playing the game multiple times, our team tries to figure out when a player can gain advantages. We found that if a player's pieces have a larger *total power* than the opponent's pieces have, a player is most likely to win. We found that if a player has *more pieces* than the opponent, a player is more likely to win. We found that if a player has pieces more *centralized* rather than dispersed around the board, a player is somehow possible to gain advantages. However, in the end, we only use the power difference in the evaluation function and the reasons will be unfolded in the following contexts.

The first variable is the *total power difference* between two players. Apparently, if a player has a larger *total power*, it has more potentials to win. In order to gain a larger power, the player might choose to spawn more or playing more aggressively to capture the opponent's pieces.

The second variable is the *total piece difference* between two players. Considering the winning stage is that the player captures all the pieces from the opponent, it will be intuitive to include the *piece difference* variable. However, this variable seems a bit overlapped with the *total power difference*. After testing the performance of the evaluation with both variables *total power difference* and *total piece difference*, we found that the agent is more willing to spawn but not playing as aggressively as before. The difference of powers, to some extends, represents the difference of the number of pieces. In the other words, the information that the variable *total piece difference* could bring is partially lied underneath the information that the variable *total power difference* could bring. This is caused by the agent gains more potentials to increase the piece difference, which means adding such a variable is not an improvement to the evaluation function. Therefore, it is reasonable to either choose to use only the variable power difference or to use parameters to balance two variables. However, it is impossible to manually deduce a parameter for the pieces difference variable, which will be solved by using Machine Learning Methodology.

The third variable is the *intensity* (centralization). At the beginning stages, if the agent spawns randomly on board, it is more likely that the opponent attacks our pieces and lose huge benefit. Therefore, it is reasonable to play a little bit conservatively by spawning more centralized. By designating a benchmark position (3,3), which is our first move, and setting up a function to calculate the Euclidean Distance between our average pieces' positions and the benchmark position, we will penalize the evaluation score if this distance is large. However, there are two drawbacks if we want to implement this variable. The first one is that calculate such a distance requires a huge amount of time and space, as a nested loop is used in the calculation process. The second drawback is that considering the winning stage of the game, what mostly happens is that pieces are not centralized around the first-spawned position (3,3) as our agent needs to attack the opponent's pieces. Therefore, a dynamic parameter needs to be implemented, which will be further discussed under the Machine Learning Methodology section.

To sum up, the most efficient and crucial variable is the *total power difference*. Therefore, this would be the only one variable used in the evaluation function.

Machine Learning and Methodology

Considering the three variables mentioned above, total power difference, total piece difference and the intensity, to balance them, parameters need to be wisely calculated. In this case, a Linear Regression model will be the best choice to train these parameters and the label of this training is how likely our agent can win, which is a percentage between 0 and 1. The training attributes will be the *total power difference*, *total piece difference* and the *intensity*. By using methodology like LASSO Regression, we can minimize the absolute sum of the coefficients to gain a best estimate of the linear regression parameters. Last but not least, we add these parameters into our evaluation function to see if there are any improvements.

Performance Evaluation

Comparison Between Minimax and Minimax α-β pruning

We implemented both minimax algorithm with and without the α-β pruning. Using α-β pruning cannot help to win the game, but can generally help to decrease the searching time cost. Generally, selecting a good child from the children list can remove a substantial number of branches of the tree to save time and space. However, there are some edge cases such that our algorithm still needs to go through almost all the branches as few of branches can be removed by α-β pruning due to a poor selection of child. To show this precisely, the time complexity of minimax algorithm is $O(b^m)$ respectively and, in term of minimax with α-β pruning algorithm, it is $O(b^{m/2})$. The worst case time complexity of using pruning is the same as minimax algorithm, $O(b^m)$, which means generally α-β pruning shortens the searching time a lot.

Comparison Between Minimax α-β pruning and Random Selection with Greedy

In order to test the efficiency of our agent implemented with minimax algorithm, we created agent using different approaches. We created another strategy called Random Selection with Greedy. It plays aggressively if there are opponent's pieces lie within the spread range (power) of its pieces. If it cannot make any attack opponent's pieces, it will arbitrarily choose a position on board to spawn a piece. This agent is considered to be a little bit more cleaver than one just arbitrarily making actions. According to the **Table 1**, the minimax with α-β pruning agent, playing as RED to make the first move, wins 85% out of the 20 games playing against Random with Greedy agent, and wins 90% when playing as BLUE (losing a tempo). This comparison illustrates that Minimax α-β pruning can deal with most of the situation and then it is a better strategy than Random Selection with Greedy.

Comparison Between Minimax α-β pruning and Random Selection with MTC

The Monte-Carlo tree search algorithm utilize a similar strategy with the minimax algorithm, instead of using an evaluation function, MTC plays against itself, by using random selections to estimate a win rate for each potential action. It will choose the action with the highest probability of winning to be the next step. Due to time constraints, it is only possible for MTC to play against itself by using random selections, but not any other algorithms. This tremendously reduces accuracy of selection, as it can be seen as a Random Selection algorithm with depth. In **Table 1**, we can see that MTC agent plays better than the Random with Greedy agent but not as powerful as the Minimax with α-β pruning agent.

Therefore, by comparing across three agents implemented with different algorithms, we conclude that Minimax with α-β pruning is the most effective and efficient choice.

Blue / Red	MTC	Minimax with α-β	Random with Greedy
MTC	undefined	**6**:14	**14**:3:3(draw)
Minimax with α-β	**17**:2:1(draw)	undefined	**17**:3
Random with Greedy	**4**:15:1(draw)	**2**:18	undefined

Table 1

Other Aspects

Algorithmic Optimization

Since the second part of the project involves both spawning and spreading, it is crucial to consider when and where a spawn and spread should take place. In order to find out an optimal strategic algorithm, it is important that we try multiple games to inspect the whole gaming process. At first, we need to come up with a first move, if the player is on the offensive. Apparently, we can choose any spot on the board, since there are no edges on the board, which makes every spot on the board having the same effect to be the first move. Considering the situation that opponent spawns a piece, it will be a bad choice to spawn next the opponent's piece, and it will also be a bad choice to spread our piece which will land next to opponent's piece, unless doing so is a trap. In order to make such a trap, our algorithm need to think opponent's potential moves and make alluring spreads.

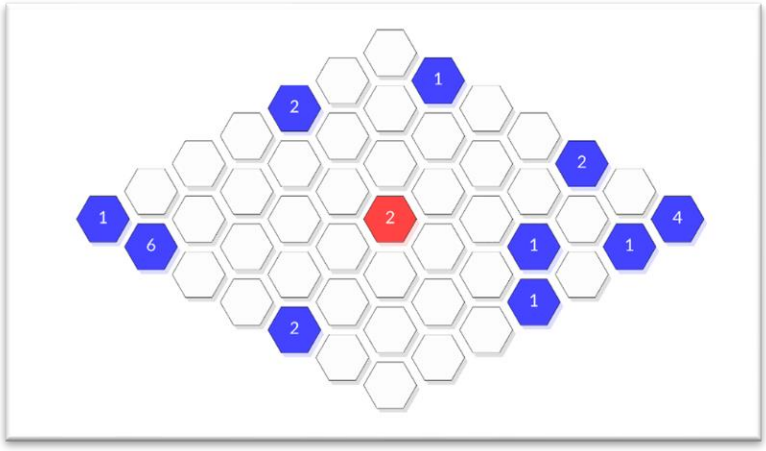

Figure 1

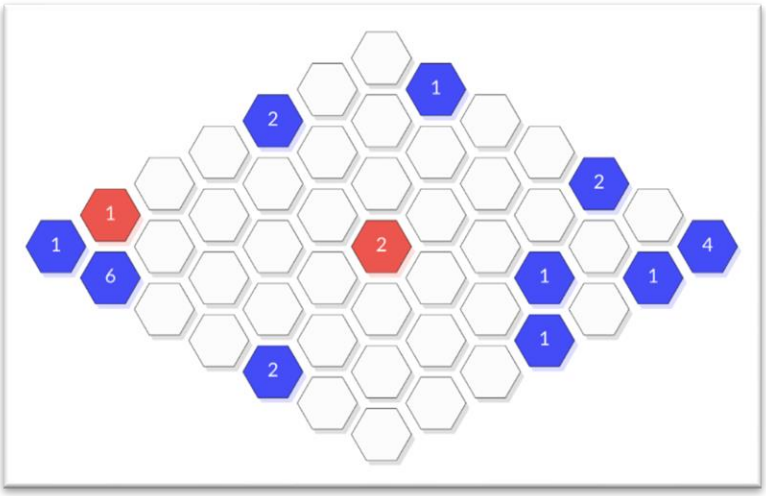

Figure 2

Since our evaluation function is only based on the power difference, as long as the player is falling into a very great disadvantageous position (**Figure 1**), our algorithm will select a spot that is closest to (0,0), which is a constraint of the algorithm. (**Figure 2**) This spawn action not only makes no improvement to player's situation, but also freely giving opponent the chance to increase its power. As this situation does not always happen and it will take a lot of time to search for a "best" point that may makes the player's situation get better, if we choose to add some useful variables, e.g., intensity of points, into our evaluation function. Therefore, we decide not to implement such a strategy to overcome this issue.

Specialized Data Structures

In order to observe the game board in a better way, rather than observing the board state in the cmd output, we created a dictionary to represent the board. The keys of the dictionary are the coordinates or position of every single spot, and the values of the dictionary are tuples, consisting the color of the current piece and the power of this piece. If there is no piece in one position, initially it will be set to (None, 0). Once there is an action made by an agent, the action would be captured and updated into this dictionary.

Another important data structure is that, in the minimax function, a list named as children will be created, consisting of all potential actions that could be take. There are only two kinds of actions, spawn and spread. To make a spawn action, we need to go through all spots on the board to find spots that are not occupied by any player. Then a tuple consists "spawn" and the position of this spot will be appended to the children list. To make a spread action, we need to go through all spots on the board to find spots that is occupied by a specific player. Then a tuple consists "spread", position of this spot and the potential direction will be memorized by the children list. Creating such a list, it reveals all possible actions that could be taken in this situation, which makes the algorithm able to find a "best" action along with the help of the evaluation function.

Significant efficiency optimizations

To reduce the time when computing more depth in minimax, we have to use α-β pruning. As discussed earlier, the first step does not make any difference to the whole game, so we hardcoded the first step to be spawning at (3,3).

Supporting Work

Creating other agents

We created two agents, one implementing random with greedy and the other one implementing Monte Carlo tree search. By allowing them playing with each other, we try to observe which algorithm plays the best.

Besides, when the depth of our minimax function increases, the computing time increases exponentially. In order to run within 30 seconds, it is impossible to run depth larger than 3, which is max-min-max. By searching for possible solutions online, we find that the game trees can be reused to shorten algorithm searching time.

Lasso Regression Methodology

When creating variables of the evaluation function, we found that we need to use parameters to modify how much each variable contribute to the final score. Therefore, a Linear Regression Model is developed to process this task. By searching for possible methodologies, we find Lasso Regression Methodology is suitable for this task, which stands for least absolute shrinkage and selection operator. It minimizes the sum of squares of parameters.

Reference

c++ - Reusable AI Game Tree. (n.d.). Code Review Stack Exchange. Retrieved May 4, 2023, from https://codereview.stackexchange.com/questions/190937/reusable-ai-game-tree

Jason Brownlee. (2016, March 24). Linear Regression for Machine Learning. Machine Learning Mastery. Retrieved May 7, 2023, from https://machinelearningmastery.com/linear-regression-for-machine-learning