Assignment 2 COMP30024: Artificial Intelligence Semester 1 2020

Group Name: DeepMagic

Student 1 Name: Chan Jie Ho
Student 1 Number: 961948

Student 2 Name: Shivam Agarwal
Student 2 Number: 951424

Going through the specification, we deduced that this game is deterministic with perfect information, and with that, we already knew that the best way to approach this game is by using the minimax algorithm. However, before jumping straight into that, we decided to go at it step-by-step, building our program to first do the bare minimum before we started to make it smarter. Our approach was mainly based on the lecture notes (in which we could immediately tell that this was a problem best approached by the minimax algorithm) and a mix of our own research material (links at the bottom) that we followed when it came to making our program smarter.

When creating the program we had to first think about what kind of data structures our player would need to use when storing the board and its pieces. We ultimately decided on an 8x8 matrix of tiles (of the CellObject class) as well as two dictionaries of the coordinates of the pieces and the number of pieces on that coordinate like Assignment 1. Our reasons for this is because an 8x8 matrix would make it a lot easier to check a tile and its contents out – whether it's empty, has a friendly or opponent piece, number of pieces, etc. – through indexing. The 8x8 matrix would be made up of the CellObject class which sets the coordinate of the tile, as well as the colour and number of pieces on that tile.

We initially thought of only using this 8x8 matrix throughout the implementation of our game, but we quickly realised that when wanting to find our pieces or the opponents when it came to generating all the valid moves or our evaluation functions, we would need to constantly iterate through the whole 8x8 matrix to find the pieces. This would mean a cost of 56 searches each time we want to search the board. It would also mean having to pass the large 8x8 matrix around a lot, which is unnecessary as some functions only require the coordinates of the player's piece or that of the opponent's. By storing a dictionary with the key of the coordinates of the player/opponent pieces and a value of the number of pieces on that coordinate, we managed to simplify our code.

After coming up with the two dictionaries, we thought that it would make our 8x8 matrix redundant, but it proved to still be of value when it came to applying the move action, where we would need to keep track of the colour of any tiles that we affect. For example, when moving a tile, we need to check if the new tile that we are moving to has any pieces on it and if it is of the same colour as our piece, where we would then determine if we are stacking or just moving it.

Our program can be developed as a mix of both reflex models and of the adversarial search models. The reason why we decided to have a mix of the two rather than having a full adversarial search algorithm based approach is because of the ridiculous size of the tree at the beginning stages of the game. Despite the minimax algorithm being optimal as well as being complete since there are only a finite number of actions that can be created, this "finite number"

is very big in the beginning; in our first round, the white player itself already has 50 possible actions, and that's already exclusive of all the invalid moves cause by some of the pieces being at the edge of the board. This number, however, is actually towards the lower end of the range of the possible total number of valid actions as we have yet to incorporate the stacking feature into our game. Once stacks are brought into the picture, we get 1 boom and up to $4n^2$ moves per stack, where n is the number of pieces in the stack. To help visualise this number, if we had 4 stacks of 3 pieces, we would have 111 possible moves. Even with a depth of just 2, with a time complexity of $O(b^m)$ and a space complexity of $O(bm)$, it is just unfeasible to use a minimax algorithm to its fullest extent at the beginning while the branching factor is so large.

That said, there are two ways to approach this problem: either reduce the branching factor, or the max depth that we look ahead in the tree. By playing the game ourselves, we quickly realised that an easy way to reduce the branching factor is by getting rid of all the boom branches at the beginning of the game. This is because there is no good outcome that comes out from booming our pieces early in the game as the opponent's pieces are far off at the other side of the board and would take at least 5 turns before a reasonably valuable opportunity to boom appears. By removing the boom actions from the list of possible actions (which is also the branching factor), we managed to bring our possible moves down to 38. Alpha beta pruning was considered at this point but we quickly realised that it was redundant and useless with a minimax algorithm with a max depth of 1.

However, this is still a relatively large number when an exponential is brought in for the time complexity. Hence we decided to reduce the max depth that our minimax algorithm would traverse to a max depth of 1. This essentially means that we are using a greedy approach where we would choose the most immediately rewarding action available each turn, without considering the opponent's responses, where "most rewarding" is defined in our evaluation function (explained later). This then changes our model to be that of a reflex model.

In our first version of the game, we had our program randomly decide on the next best move from the list of possible moves (still excluding the booms). As expected, it failed spectacularly, so we knew we needed to think up features that would go into our evaluation function. It was, however, a decent model to test our earlier models against. Our baseline model was then created to maximise the number of our pieces remaining on the board. It did only slightly better and got better after we turned it into an adversarial baseline by maximising the number of our pieces remaining minus the opponent's pieces, which would signify us booming their pieces off.

We then evolved into making our evaluation favour specific states by picking hand-weighted features to consider other factors such as:

- Number of player pieces
- Number of opponent pieces
- Number of player stacks
- Number of opponent stacks
- Max height of player stacks
- Max height of enemy
- Number of player clusters
- Number of opponent clusters
- Average size of player clusters
- Average size of opponent clusters

- Clustering score (How many of their pieces are connected to ours vs how many of our pieces are connected with theirs)

We based these features based on our own knowledge and strategies of the game. We tried emphasizing stacking as part of our strategy as we realised that stacking provides our player the mobility it needs to attack from a far distance. This is where the number of player/opponent stacks and max height of the player/opponent stack comes into play. We also realised that clustering (grouping of our pieces) was something that we wanted to avoid. This is broken up between the number of player/opponent clusters, the average size of the clusters, and a clustering score defined as a way to compare the number of the opponent's pieces that were clustered together that are connected to our pieces against the number of our pieces connected to the opponent's pieces. We tried to minimise clustering so that we can minimise the risk that we would take if we were to have a large cluster as it would be very easy for them to attack and get rid of all our pieces in one move in a cluster, as well as maximising the number of clusters on their end so as to emphasize attacking their larger clusters while sacrificing as few of our pieces as possible.

The main problem next was to decide on what weights to run these on. We considered a few methods when thinking of how to get the weights such as gradient descent learning, TD leaf learning, and machine learning methods. The problem about gradient descent learning for expendibots is delayed reinforcements where the reward of a move in a certain direction to boom something further out is not immediate as the game requires a multi-step approach. TDleaf learning would have been best applied in this situation, but our research on it was deemed unfruitful and we decided to stick with methods that we were more comfortable with.

While we did not use any particular machine learning algorithms in our work, we did manually run it and manually hand picked the weights based on how much of an effect it had on our actual game play. After running it manually, we came to the conclusion that the most prominent feature was still the initial adversarial baseline features where it just took the difference in the number of pieces on the board.

That was how we modelled our reflex model, but having said all that, we knew that leaving it with a max depth of 1 throughout the whole game would be a big waste of the minimax algorithm, so continuing with our variable cut-off depth approach, when there are less pieces on the board, we tried increasing this depth to 2-3. With this depth, we were then able to start making smarter moves by looking further ahead without forcing too big a tree onto our algorithm.

We took the help of various sources including the ones mentioned below. These articles gave us valuable information in terms of data structures as well as how to choose factors for our evaluation function.

https://towardsdatascience.com/pathwayz-ai-3c338fb7b33
https://stackoverflow.com/questions/15697479/beating-a-minimax-opponent