Report

Team Name: Charlie_Gone_Wrong
Team Members: Ashley Knipler - 999159
Girish Madnani - 934130

Question 1: Describe the approach your game-playing program uses for deciding on which actions to take throughout the game. Comment on your choice of the search algorithm, and on any modifications, you have made, and why. Explain your evaluation function and its features, including their strategic motivations.

Answer 1:
The main approach decided for our game-playing programming was minimax. This was done as it is a two-player game. In order to improve the efficiency of minimax, we used $\alpha - \beta$ pruning.
The heuristic function was made up of five considerations:

First Stage – To check if a player has won; the game is over.
Second Stage – Add to the node score depending on how many more opponents coins are destroyed by a boom than ours.
Third Stage – A distance heuristic which adds scores to nodes that move our coins towards the enemy.
Fourth Stage – Through multiple trials of the programme we found out that the centre tiles of the board made the player more flexible and have more control of the game. Hence we gave a score booster if there was at least one coin in the centre of the board.
Fifth Stage – The fifth heuristic was dreamt up after realising the limitations of our software, as it could only run with a depth of two at the start of the game. As such, scenarios were found where if a coin moved to a location adjacent to two or more opposing tiles, there was a large chance of the next move being a strategic boom, depending on those black coins positions relative to each other. This is essentially looking an extra move ahead, albeit in a limited capacity as it only checks if the next move was a boom and it does not simulate the opponents move. However, the logic is sound that given one of the situations in Figure 1., the opponent will not be able to escape without losing at least one piece.

	B	
B		B
	W	

	B	
	B	B
	W	

Figure 1. The scenarios where although Black can move, White is guaranteed a successful boom (White has just moved).

Hence this is essentially a trapping function.

When our AI has more coins and it comes to a situation where there will be an equal trade when it explodes it will take that option as it would win the game.

When performing our first action we always got a movement from the far-left tile of the board if we are white but in order to get most of the starting move, we hardcoded to move the starting middle block to stack up. This gives us more flexibility in the movement and gets the coin closer to the middle of the board.

We also created a function that decided the depth. The function observed the number of coins left on the board to judge an approximate branching factor then, using a log calculation, a depth was given with a depth range from 2-4. Our number for 'max nodes' used in this calculation was 115000.

Question 2: If you have applied machine learning, discuss how it fits into your program's overall game-playing approach, and discuss the learning methodology you followed for training and why you followed this methodology.

Answer 2:
We did not use any machine learning approach however, we used a smart heuristic in order to make our AI perform at a high capacity.

Question 3: Comment on the overall effectiveness of your game-playing program. If you have created multiple games- playing programs using different techniques, compare their relative effectiveness, and explain how you choose which program to submit for performance assessment.

Answer 3:
In order to make our AI effective we used an $\alpha - \beta$ pruning approach. Along with this, we made a test dumb AI allow us to compare different iterations of our heuristics function, noting different levels of effectiveness. In terms of making it smarter, we made use of the heuristic function to create a more effective and a more targeted AI.

Issues we noted with our AI include:

- In some end scenarios, becoming stuck in a loop with the opposing AI, resulting in a draw even though our player was winning, often by a lot.
- Due to a lack of depth at the start, our AI could not see far ahead. Opponents AI's would be able to get into strong positions adjacent to our starting clumps of coins, and we would lose four or more coins at the start of the game.

We were able to partially resolve the first problem by implementing a simple pattern recognition function, which after seeing the same moves repeated, gives a strong negative score to those moves to promote different behaviour. This seems to work sometimes but in other scenarios, the opposing AI just continues to dodge our AI until eventually a draw is called.

We discussed implementing another heuristic which would promote the use of stacks, as this would allow us to theoretically chase down the opponent, however, we ran out of time in implementing this and found it difficult to rationalise an approach after this that would give our AI reason to 'fire tokens' at locations that would trap the opponent. Perhaps this approach could be implemented by telling our AI to spread out our pieces, placing the most amount of squares in danger as possible; however, once again, we ran out of time to implement this. This last function would have only been activated past a certain point in the game - when there were only around 12 coins left on the board. The reason for this is to enhance the speed of our program early on with so many pieces and because this strategy is more useful in the late game. The stacking heuristic can be useful always.

We tested our program against our previous modified program in which it did not have variable depth and the fifth stage heuristic. It was faster as it had a depth of 2 but the AI was not smart to tackle the opponents offensive moves. Our program is slower as it has a bigger depth and could tackle certain problems and could not be handled before making it more smarter. Despite being a smart intelligent AI, it would always lose if it was black and the time would increase exponentially.

We included the variable depth but it made our programme too slow for the 60 seconds time limit hence we choose our programme without the variable depth, making it a bit a not as smart as the variable depth version of our program as it could not see more possible moves ahead.

Question 4: Include a discussion of any other important creative or technical aspects of your work, such as: algorithmic optimisations, specialised data structures, any other significant efficiency optimisations, alternative or enhanced algorithms beyond those discussed in class, other significant ideas you have incorporated from your independent research, and any supporting work you have completed to assist in the process of developing an Expendibots-playing program.

Answer 4 :

We first went through the lectures in order to understand the concept of minimax function to create a two-player game and to improve its overall time duration we also learnt how to implement $\alpha - \beta$ pruning.

We decided to do some research online in order to find a good and effective minimax function which could help us achieve our goal of creating expendibots for two players. Along with this $\alpha - \beta$ pruning was also found online in order to further increase our game's efficiency.

The closest game to expendibots was checkers and hence we took some ideas online for the implementation and heuristics.

Originally we implemented the data structure as a class with coin data being saved in a dictionary with the values being lists. We realised later that our loop up times could be improved by implementing a dictionary within a dictionary. This allowed for less complex control and faster lookup times as opposed to implementing a list of lists.

The coin data was stored in a class that acted as a node in our alpha-beta pruning tree, along with other useful data such as the number of coins left in-game, which colour we were playing as, turn count and more.

Our heuristic makes our coin perform the best defensive move when the situation gets tied up where the opponent coin has cornered us to a position where we lose a lot of coins compared to opponents coins our program figures out the best defensive move in order to minimize our loss and get the best of every action we perform defensively.

At the start of the game, we made our programme to be more aggressive and it hence gets more opportunities to get in the best offensive positions early making us have the advantage until the end of the game. We also turn off the aggression after a certain amount of turns, when we don't turn it off it stops performing defensive moves and focuses more on booming of opponent's tiles making it reckless and putting us at a disadvantage.

When the program performs the same action 2 times, a repetition checker function is activated which stops the coin from performing the same action again. This helped improve the speed of our algorithm, as the repetition is checked at the start of the heuristic function, letting that node skip the rest of the function.

Examining a document written by Carl Felstiner and Professor David Guichard in 2019 (a proper copy of this document could not be found, so we are unable to properly cite it)[1], on the implementation of $\alpha - \beta$ pruning in different games gave valuable insight into potential improvements we could make to our function. One such potential improvement would be to randomise the search order in the tree, as this increases the possibility of healthy pruning. We realised however, that smarter than doing a random order is by ordering our generated moves based on boom statements first (booms are only generated if our coin is next to an opponents coin) and then the first move generated is a move towards the opponent's side of the board. This is also known as *action ordering*. In the late game this may not be optimal, but in the starting case with lots of coins it has a much higher optimality.

[1] Document can be found at this link:
https://www.whitman.edu/documents/Academics/Mathematics/2019/Felstiner-Guichard.pdf