Project Part B Report

1.0 - Agent Design/ Strategy:

Our program seeks to find the best moves in its position by utilising the adversarial game playing algorithm: Minimax with Alpha-Beta pruning. We chose this method as our aim of finding the best move is a classic adversarial search problem where both players compete in a perfect information, static environment. Each player seeks to maximise their own utility function while minimising that of their opponent, so we utilised Minimax to effectively explore all feasible action sequences to a certain depth. To increase the depth that we searched to, we implemented Alpha-Beta pruning which helped avoid exploring branches that we knew lead to suboptimal positions, this was necessary as the number of states grows exponentially with depth, so we sought to eliminate as much expansion as possible to save space and time. This implementation was based on the AIMA textbook; we adapted it to suit our game's conditions.

To further encourage pruning and a more efficient solution, we sorted each layer of children (moves) in our decision tree by a "move heuristic" that measures the desirability of a given move based on its local effect (doesn't consider the whole board state). This allowed for expansion in order of desirability for all moves available for min and max, helping prune as many branches from this sub branch as possible as we had likely seen one of the most desirable positions at the start of our pruning.

Evaluating each position was done by our evaluation function, which heuristically evaluated positions based on the location of player and enemy pieces. The main component of which was our difference in vertical distances sum, this component consisted of the sum of vertical distances for each enemy piece to their respective end of the board (goal state) minus the same sum of vertical distances for the player frogs. This provides a simple heuristic that ensures we know which player has the advantage when it comes to proximity to their goal, the heuristic is hence positive when our player is closer to its goal and negative otherwise.

Another component we added was a penalty to prevent frogs being left behind as a player moves other frogs forward, this ensured we were more likely to reach positions with more control of the board and avoid our frogs being cut off on the back rows with no beneficial moves to play. Hence we would encourage less grow action usage which are moves that don't make forward progress, this also prevents growth in the back rows which may help the enemy find more paths to their goal state.

Lastly we had a heuristic boost for frogs close to the end of the board to encourage positions where we are close to winning and frogs don't get stranded too far apart (vertically).

Deciding on weights was difficult as we lacked the time to use machine learning techniques, we first set our weights all as 1 and tested against a slightly worse program (simplified eval function) in order to see how many moves we would win by. We then adjusted the weights manually (+-1 each time) in order to beat the other program by the least number of moves.

When our program was at the stage where it could see forced wins or losses, our heuristic returned infinity or negative infinity respectively. For won positions, we then adapted our search function to keep track of the depth in which each terminal state was found and propagate it back up the tree. This allowed us to pick the solution at the shallowest depth, so we could win as fast as possible and avoid cycles. While for lost positions we implemented a fallback measure which returned the move that maximised a forward progress heuristic (greedy best move), so in the case our opponent makes a mistake and blunders their forced win we still have a chance

2.0 Algorithmic Enhancements & Constraints:

Although we were unable to fully reimplement the board class in order to save resources due to time constraints, some of the features we implemented still drastically reduced computation time, allowing us to search to a deeper depth. We found our greatest issue to be time as we often exceeded the time limit at higher depths, while for memory usage, we used recursion and propagation in our search tree and hence didn't need to store each node as its own object. Our solution would always time out before we even reached a memory usage 1/100th of the limit.

Our best computation time reduction was through pruning; these two figures show the total number of nodes our algorithm expanded (min and max calls) during a typical game against a similarly skilled opponent, tracked by turns of just our agent. So 25 turns here would be 50 in game turns (25 by each player). The figures are Minimax with Alpha-Beta Pruning vs Simple Minimax.

Figure 2.1 With Alpha-Beta Pruning (depth 4):

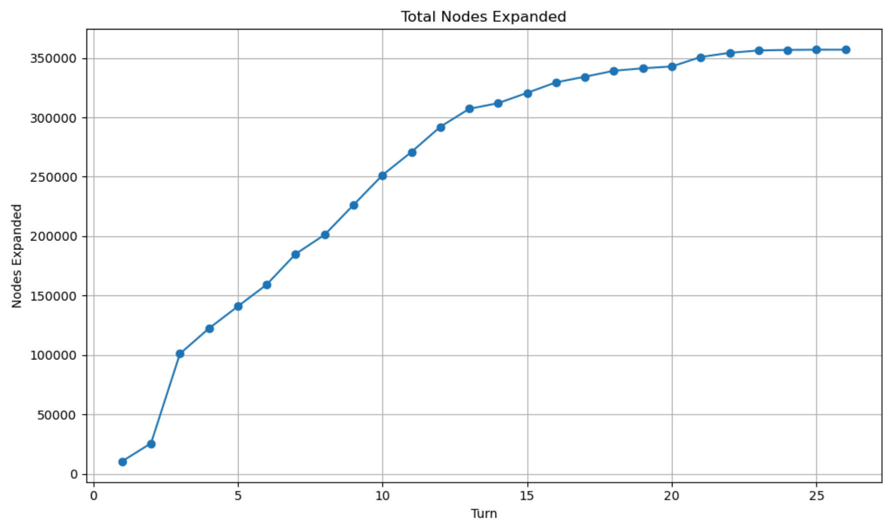

Figure 2.2 Minimax Without Alpha-Beta Pruning (Depth 3): (Y axis scale is 1 = 1 million)

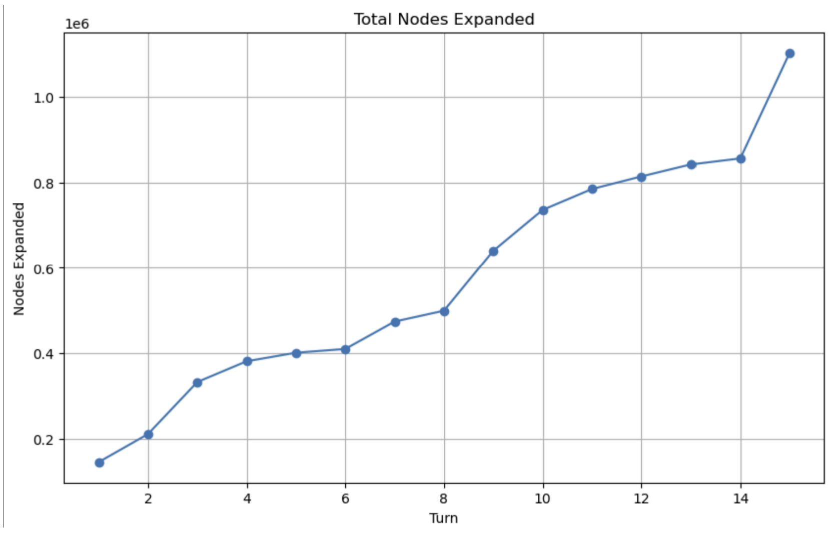

Figure 2.2 was forced to end early due to exceeding time constraints and we can see why. At a lower depth than figure 1, Simple Minimax was already expanding around 4x as many nodes (300k vs 1.1 million) per turn compared to Alpha-Beta pruning by turn 15. Having to evaluate 1 million nodes per turn is obviously very inefficient for the program and our Alpha-Beta cutoffs greatly increased the performance of our agent as we could look to higher depths and see move orders that we previously couldn't.

Lastly, another thing to note was the fairly linear increase in total nodes expanded that both models had at the start of each game which plateaued as the game reached its closing stages, this is to be expected, as it reflects the size of the branching factor (available moves). During our middle game we had the most complex positions with the largest amount of possible moves, we therefore required the most computation, while during the start and particularly endgame we had fewer move possibilities and less nodes needed to be expanded.

Next time to improve our model, we could exploit this property and allocate more time and resources to middle game positions and less to the end game (for example: by setting a time limit based on the move number) in order to increase search depth and improve our model's performance. Although depth still exponentially affects nodes expanded, we could decrease the effect that this has on our constraints by choosing when to use our resources effectively.

3.0 Performance Evaluation:

To evaluate our agent's performance, we ran tests simulating matches of our agent at depth 4 against three different agents. A random agent that plays a random move from all possible moves on the board; a greedy agent that seeks to advance as far as possible from the current board position; and another agent that prioritizes positions with the most jumping opportunities. The latter has greater depth in the algorithm and doesn't just analyze the current state of the board.

The results were as follows:

3.1 Random:

We used this agent as a base, an agent without any strategic thinking that allows us to evaluate whether our strategy makes sense. We simulated 10 matches in total, 5 with our agent as blue and 5 with our agent as red. The results were as follows:

As BLUE: victories at turns 82, 72, 80, 50, and 62

As RED: victories at turns 63, 63, 57, 53, and 55

Our agent had a 100% win rate in all 10 matches. What stands out most about these results is that, on average, when our agent played as red, it won much faster, suggesting that our agent generally performs better playing as red. However, the fastest win was as a blue player, noting that if the perfect situation presents itself, it can also win efficiently a blue.

3.2 Greedy Agent

This agent represents a very simple player who doesn't think ahead and only plays the move that gives them the greatest advantage at the moment. Only two games were played against this agent because each time we played it, the result was the same (the same game is played).

The results were as follows:

As RED: victory at turn 57

As BLUE: victory at turn 68

Our agent won both games, demonstrating its superiority against more naive strategies without forward planning. This time, winning much faster as red than as blue, demonstrating an advantage or better performance when playing first against this type of agent.

3.3 Jump-Opportunity Agent

This agent maximizes the number of opportunities it has to jump and minimizes the opponent's. It searches to a depth of 3, so it is able to plan and not only look at the current board position, the agent also has a vertical distance-to-goal heuristic so that it advances and doesn't just search for jumps.

What we want to achieve by testing our agent against this agent is to evaluate our strategy against an agent that already plans ahead and uses medium-term positional thinking. Only two games were played against this agent because each time we ran it, the result was the same (the same game is played).

The results were as follows:

As RED: victory at turn 63

As BLUE: victory at turn 62

Our agent won both games, demonstrating its good planning ability and correctly identifying bad and good moves for itself and also for the opponent. By beating an agent who already sees three moves ahead and has a strategy that we also believe is sound, we can say that our strategy is quite successful. Here, it showed equal performance playing as both red and blue. This tells us that against agents who already have a clear, more sophisticated strategy, our agent performs equally well in both roles.