1 Minimax Approach

Algorithm

The team decided to develop a game playing agent that utilises the adversarial search algorithm Minimax. Minimax was selected due to the Freckers being a deterministic, perfect information, zero sum game, and the determinism and strong strategic play of the algorithm in structured games.

Alpha-Beta Pruning

To optimise the Minimax algorithm, alpha-beta pruning was used to prevent the evaluation of nodes that would have no impact on the decisions made by Minimax.

Further Optimisation

Alpha-beta pruning is improved when better moves are explored earlier, as they are the most likely to move alpha up or bring beta down to the pruning threshold. As such, we needed a way to estimate which moves are generally better and explore those first, without spending extra computational power to do so. Based on our heuristic, moves that travel a further distance to the other end of the board tend to be better. The cheapest way to optimise this is to first expand jump-move nodes, then forward-moving nodes, then finally sideways-moving nodes. This wasn't done in the Minimax algorithm, rather the `Board.getMoves()` method we developed returned a list of moves that was ordered in that way.

Static Evaluation Function

Minimax needs a way to compare leaf nodes. This is simple when the leaf node is a terminal node, the game is over, and there is a clear method to determine the winner. However, for non-terminal game states, we needed to develop a heuristic that could accurately represent the advantage a player has, where a higher advantage corresponds to a higher likelihood of winning the game. This allows the agent to make decisions based on board states without having to search through the entire game tree without a limited depth, which could be up to 150 nodes deep, and isn't feasible within time constraints.

The heuristic function used for the leaf nodes of the Minimax game uses a static evaluation of the board state purely on the positions of all frogs. The static evaluation is made up of a 4-tuple that progressively tie-break based on the importance of each variable. In first place is the point difference, calculated by how many more frogs the player has on the winning row than their opponent. Next is a distance difference, calculated by how much closer all the player's frogs are to the home row than the opponent. The total distance doesn't take into account lily pads, but does take into account horizontal distance that would need to be to the nearest win row cell that isn't occupied by a player's frog, but recognises that diagonal moves exist, so it is not Manhattan distance. At this point, the static evaluation function cannot differentiate between cases where both players make an equal amount of distance, such as neither of them moving forward, or both of them moving forward an equal number of spaces. So to tie break once again, the third element of the tuple represents the distance travelled or the progress made. This breaks out the evaluation from a zero-sum scenario, but allows us to consider moves where the player is closer to winning, even if the advantage is the same.

The final element of the tuple was a solution to a problem we came across, which is more prominent with a lower depth. Consider a scenario where Blue has all its frogs on the winning row except 1, and that frog only has one lily pad in front of it, with no clear path forward. The moves that Blue can take are either:

1. move forward, where then its only option is to grow,
2. grow now, and then next turn have the option to move forward.

If the agent considers this scenario with a depth of 2 or 3, essentially only looking at two if its own moves, the best case scenario for both looks the same, the last frog has moved one space forward, since the heuristic doesn't take into account lily pads. This meant that agent could potentially waste a move or get stuck in an infinite loop of move-wasting. As such, the last element of the tuple is another variable length tuple that represent the history of the progress made by the frogs, and so if two scenarios look the same, the one that made progress earlier will be favoured.

Time Allocation and Iterative Deepening

Time Allocation Goal

Minimax isn't an algorithm that is bound by time, rather it's bound by the depth of your search and the time taken depends on the depth, branching factor, and of course the computational power of the machine the agent is running on.

Given that there's a time limit across the entire game of 180 seconds per player, we need to effectively allocate time per round for the agent to make decisions. There are two parts to this, how much do we allocate for each round, and given that we've allocated a set amount of time, how do we ensure that the agent remains within that allocated time given that the time taken cannot be predicted.

Complexity Analysis to Effectively Allocate Time

Early on in the game, the branching factor is relatively lower; there are fewer interactions between frogs until they meet in the centre and open up various opportunities for successive jump moves. When all the frogs are concentrated in the middle, that's where the branching factor is the highest, and the same depth of calculations will take a longer time to complete. Towards the end, when frogs are at or near their winning rows, the branching factor is reduced a lot more, so the time required for the same depth is lower.

Each player has 75 turns to play, but the intention of the agent is to reduce the number of rounds required to win, where in a matched game, it shouldn't take more than 70 rounds to complete a game, with leniency ideally up to 100 rounds. Not only do we need a time allocation function that roughly follows a negative quadratic, but one that is also positively skewed such that more time is spent on earlier rounds.

Mathematical Calculation of an Appropriate Function

The general cubic was determined to be:

$$f(x) = ax(x - 75)^2 \tag{1}$$

Where the value of a is to be calculated such that:

$$\int_0^{75} f'(x) \approx 170 \tag{2}$$

The reason we pick 170 instead of 180 is to ensure that we have a safety net in case some calculations take longer than expected. Despite planning to correct that along the way, the leeway is still needed for more gradual correction, and especially to not risk timing out. And so a was determined to be ≈ 0.000065

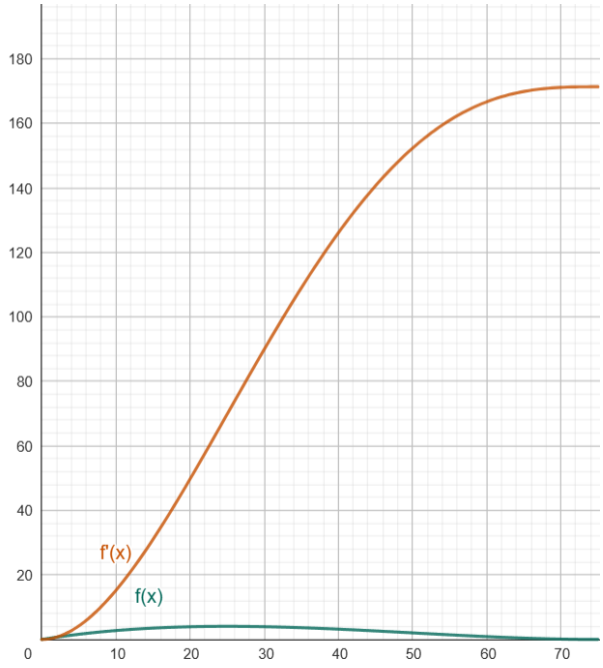

Figure 1: Time Allocation Function, $f(x)$ and it's antiderivative, $f'(x)$

This strategy ensures that in the:

- Early Game: Quick evaluations prevent over-investment of resources when there is little advantage to be gained.
- Mid Game: Greater time is used when strategic decisions are most impactful
- End Game: Moderate precision with shallow depths, as fewer choices often make exhaustive search unnecessary

It also allocates half of the 180 seconds by round 60, and $^5/_6$ of the time by round 100.

Iterative Deepening to Effectively Use Time

To complement the cubic time allocation strategy above, the agent employs an iterative deepening search strategy. Rather than committing to a fixed search depth at the outset, the agent incrementally increases the depth of its Minimax search, starting from the shallowest level and deepening if there is allocated time available.

The agent will always complete the depth that it is searching, but instead estimates how long searching that depth will take compared to the taken time, `timeTaken`, of the depth before it. Generally, if there is still `depth * timeTaken` seconds left within the overall allocated time, the agent would be permitted to search another depth. This estimate balances out the branching factor with the depth searched. In case there is no time to perform a search, a random move is picked.

By combining iterative deepening with a time allocation function, the agent can guarantee responsiveness while still benefiting from deep strategic planning in complex situations. The cooperation between time allocation and search depth ensures that the agent remains both efficient and effective across all stages of the Freckers game.

Amplitude Rescaling

Given a round number, r, we initially allocate a time of $f(r)$ for the agent to play its turn. However, because of early termination or potentially going over time, $f'(r)$ does not always represent the actual time used by the agent so far. So in reality, the time allocated then becomes:

$$g(x) = \frac{f(x)f'(x-1)}{\text{actualTimeUsed}} \tag{3}$$

$$g(x) = \frac{0.000065f'(x-1)}{\text{actualTimeUsed}}x(x-75)^2 \tag{4}$$

This allows the total time used to oscillate around the function, and with testing, it has showed that it generally keeps the actual time used very close to $f'(x)$.

Approach Limitations

- Evaluation Accuracy: The heuristic is limited in what it can assess, and these limited evaluations may lead to suboptimal decisions.
- Hardware Limitation: May miss better moves due to not being able to search a sufficient depth.
- No Learning: The agent lacks pattern recognition and any deeper learning since all decisions are based off of a heuristic.

2 MCTS Approach

The team developed an agent that utilised a Monte Carlo Tree Search algorithm to find the best move to play. However, thorough testing the team found that the agent performed significantly worse than the above Minimax agent implementation.

Simulation

The accuracy and effectiveness of MCTS is largely dependent on the simulation phase. In this case, random playouts are simulated to estimate the value of a move. The non-determinism associated with selecting random moves during playouts often lead to the evaluation of suboptimal moves with minimal strategic advantage. Although, replacing the selection of random moves with moves based on a heuristic may result in the agent prioritising investigating more promising moves, the agent may overvalue moves that align with the heuristic even if other unexplored moves are stronger. Furthermore, the time and memory constraints placed upon MCTS prevent a large enough volume of simulations to be completed, accumulating statistically insignificant results for the agent to evaluate and choose an optimal action. If the agent was to be given more resources, the accuracy of the MCTS agent would likely improve. A large number of simulations results in a more accurate approximation of the true value of a move. However, such modifications do not align with the given time and resource constraints.

3 Performance Evaluation

To objectively assess an agent's effectiveness, a series of tests were conducted using performance metrics. The metrics enable quantitative comparison of the decision making capabilities of each agent. These were done against other agents, previous iterations of the agent, and a head-to-head comparison of the same agents.

Time Performance

For the iterations of our Minimax-based models, the average time to complete evaluate a game round in a head-to-head match was considered when the depth was fixed. This allows us to compare two models for their efficiency in obtaining a solution, and was primarily for ensuring that our alpha-beta optimisations were effective. However, for a more informative perspective on this game, the comparison was then changed to how far the iterative deepening can explore the game tree within the same set amount of time. We found that for the same time allocation function:

1. Without alpha-beta pruning, the mid rounds would only reach a depth of 3, never 4 or higher.
2. With basic alpha-beta pruning, the mid rounds would more often reach a depth of 4, occasionally reaching a depth of 5.
3. With optimised alpha-beta pruning, the mid rounds would consistently reach a depth of 4, with a high rate of reaching a depth of 5.

Average Number of Rounds

With a fixed depth search and no time restrictions, iterations of our Minimax models behaved the same under the same heuristic, which was to be expected. However, with time restrictions, the unoptimised models took more rounds on average to beat other baseline models.

The MCTS agent required the highest average number of rounds to complete a game besides a random agent. The result reflects the agent's tendency to make less optimal moves, often those that don't move frogs closer to the end row (side moves). The tendency could be attributed to MCTS not having enough simulations, and thus not enough data to evaluate more optimal moves, but it shows slight learning.

The Minimax agent had the lowest average number of moves, beating out a greedy agent as it was able to recognise opponent counters and potential move set-ups it can produce, making it more comprehensive and able to make more progress more efficiently. Playing against itself, the enhanced Minimax model was generally able to complete a game within 55-65 rounds. The less optimised models under a time limit generally took 65-75 rounds. This improvement is substantial, as being able to win in a shorter number of rounds is crucial to winning a game of Freckers.

Win Rate

To further the performance evaluation, a series of head to head simulations were conducted. The simulations involved each agent playing multiple games against each agent.

The MCTS agent returned the lowest win rate amongst the agents, slightly beating out a random agent, but not by a substantial margin, reinforcing that it just did not have sufficient time to properly learn what moves were best. Since it was not able to make substantial progress, it was beaten out by the greedy agent that picked to make the most progress at any given point.

The fixed-depth standard Minimax agent frequently outperformed the all other agents. The exhaustive search and evaluation function allowed the agent to make more strategic moves than MCTS in most cases. However, with an iterative deepening, timed approach, its win rate was much lower than our final optimised minimax agent. This indicates that at a certain depth it was still able to make good moves, but it often missed opportunities that the optimised agent was able to explore.

The enhanced Minimax agent had the highest win rate amongst the agents. The superior performance can be attributed to the deeper searches in the more complex phase of the game, allowing it to see moves that the other agents cannot. The win rate suggests the enhancements made to the standard Minimax agent result in better gameplay decisions. It also shared an even split between Red and Blue wins when both agents were the same, indicating that there was no inherit bias by colour.

4 Supporting Work

Debug Logs

When debugging the agents, the team utilised a series of logs to an external file to evaluate and monitor the expected performance of the agent at each stage in the decision-making process.

This allowed for a way to view the game being played in real time, and then go back and understand why certain decisions by the agents were made. This greatly helped in debugging and the development of the heuristic for Minimax, as well as the further optimisations that were made.

When developing Minimax, the entire game tree and every evaluation and decision made along the way was printed to the log with appropriate indentations to make it readable. This allowed us to manually backtrace a decision that we didn't understand and take action from there. The log files were also compared against each other for different Minimax agents to see how optimisations, such as those for alpha-beta pruning, made a concrete impact on the decision-making process.

When developing the MCTS approach, the amount of wins, draws and visits of each potential action were printed to the logs. This highlighted to the team that the MCTS was in fact selecting the correct move based on the conditions and led the team to identify the issue of choosing sub-optimal moves that occurred in the simulation phase of the algorithm.

This logging functionality was removed from the submitted code as it draws away from the readability and even though it can be commented out, the ability to read and write to disk has been disallowed within the restrictions of this project. Nevertheless, it does exist within an earlier commit within the repository.

5 Conclusion

After a thorough analysis of each agent developed by the team, the adversarial search algorithm Minimax coupled with both Alpha-Beta pruning and iterative deepening, proved to be the most effective and efficient strategy when designing a gameplay agent for Freckers, easily beating random-move and greedy agents, as well as earlier, simpler iterations of the Minimax agent.