AI Part B - Winston and Jason

Methodology

We know that Freckers is a deterministic game with perfect information, much like chess or checkers; hence, it seems appropriate to apply the following approaches shown in the lectures:

- Minimax
    - Adding Alpha-Beta Pruning
- Monte Carlo Tree Search

We begin our project by first coding a basic implementation of these algorithms. After successfully developing an algorithm suited to the scope and domain of ‘Freckers’, we are then able to modify and equip our algorithms with various optimisation strategies, seeking ultimately to improve their efficiency and tailor the program to specific game mechanics and concomitant strategies.

Game Playing Approach - Minimax

Our primary agent is built around the Minimax algorithm, enhanced with alpha-beta pruning to reduce the search space significantly. We selected Minimax because Freckers is a deterministic, perfect information game akin to chess or checkers, where lookahead strategies are highly effective.

We utilised a depth-limited Minimax with initial tests of 4-ply, hoping that it would be sufficiently deep to extract insightful results whilst still allowing us to satisfy the time and memory constraints. This was, however, unsuccessful as we ran into resource issues, which limited the agent around the 40th move. From this, we decided to tune the algorithm down to a 3-ply solution. The tradeoff here is that we get faster and less expensive computation, but have a solution that looks slightly less deep into the future, meaning it may miss some deeper insights into the game. While the 4-ply model took on average 22 seconds per move, the 3-ply model took only 1 - 2 seconds. Granted, this was manually measured with a phone’s stopwatch, but we hope the evident disparity in time was enough to ignore the granular accuracy that a proper time measurement algorithm might have provided.

While discussing this trade-off between different depth values, the idea of changing the depth value depending on the stage of the game was also considered as part of our strategy. This was a modification to the original Minimax algorithm, where the depth value usually remained unchanged. Our logic was that we can reduce the depth of the “opening” moves because our first few moves are so minimally affected by what the opponent does that looking 3 - 4 moves in advance seems largely inconsequential. By reducing the depth to 2 in the opening moves, we are able to save complexity and performance, which we could choose to allocate towards the mid and end game, where looking more steps in advance becomes more valuable.

An extension of this idea was briefly considered, where we hoped to identify the stage of the game (Opening, Midgame, Endgame) by calculating the distance between the frogs of the players via the findGameStage() function. The idea was to identify two thresholds that would indicate the transitioning of Opening -> Mid-game or Mid-game -> Endgame. Depending on the stage of the game, we could then allocate a depth value more efficiently. For example, we deemed the mid-game to benefit the most from looking further moves down the line compared to the opening and end-game stages. While this strategy would likely work in principle, we found there to be too many variations in the board states such that a single factor of distance would not be able to accurately identify the correct state of the game.

Above all else, our main strategy comes from the method in which we implemented our evaluation function, and the features the function considers as a characteristic of a good move. Ideally, the strength of a move should be evaluated by considering a range of factors, equipped with a weighting distribution to reflect the importance of each feature in evaluating the move.

We formed our heuristic evaluation function based on the following rules:

1. Reward the player for moving frogs closer to the end while giving priority to frogs that are already closer to the beginning
2. Penalise opponent frog advancement
3. Reward the player for setting up jump chains*
4. Reward a grow move based on the number of lily pads created
5. Heavily rewarding winning states

*Jump chains are where a frog, in one move, jumps more than once to a destination

Our justifications for each rule:

1. The state of the board can be naively evaluated by simply considering the total vertical distance between the frogs and the end of the board. The player who is currently winning is most commonly the one whose frogs are closer to the end of the board than the other player's. This serves also to reward moves that make the frogs progress forward as opposed to perpetually moving side by side.

   Additionally, we thought to give more priority to the frogs that are towards the beginning (less developed) of the board to prevent them from being isolated and cut off from the other frogs.

2. This penalty is an extension of rule 1, whereby the evaluation of the board state should also depend on whether the opponent is close to winning. This is also the rule that influences the action determined by minValue.

3. Since chaining jumps seems to be the most efficient way to progress forwards to the end of the board, we thought that our evaluation function should be most dependent on the move's ability to set up jump chain possibilities. Hence, we created a countJumpChains function that returns the number of jump chains present within that state of the board. Board states that have more jump chains will be rewarded more greatly.

When we had just started, the model's heuristic evaluation function only rewarded the player for moving the frogs closer to the end. This ended up with some issues, primarily being that the agent was incentivised to push through the frogs one at a time, whilst not considering any sort of jump chaining or grouping strategy. As a result of this, the agent was very inefficient and scarcely utilised jump chaining and was very inefficient with its move usages. A peculiarity from this stage of our heuristic function was that at the start of the game, the agent would move all the pieces forward before growing, then it would only move the left side frogs forward a few squares before considering the right side. In any case, this was not particularly effective as a heuristic, causing us to add some new conditions.

Our strategy in the game is to ideally have each of the frogs travel in a more or less group formation. This can be achieved by giving priority to frogs that are closer to the beginning (as seen in rule 1), rather than the end, meaning the frogs will generally be on similar rows. In our evaluation function, this was reflected by giving a multiplier bonus to the frogs that were less developed.

This works well because when there is a jump chain that a frog can take, the rest of the group will still be able to move forward as the frog that is closer to the end is already more or less at the other side and so further moving of that piece will not help that much in getting the rest of the group across.

4. We needed a logic for rewarding a good growth action, as it does not assist in strictly progressing the frogs forward, and so is never deemed as beneficial by the current set-up of the evaluation function.

   The way we approached rewarding a growth move is by checking how many lilypads the growth move created. The more lilypads created within that one action, the more valuable the evaluation function will evaluate it. It follows that we should also punish excessive growth actions, as needlessly growing is almost never a good strategy and can even be beneficial for the opponent and hence harmful to us.

5. If the winning move can be found in the next move, then the algorithm should pick this move as it is simply always the best move that the agent can make. This is reflected by how the evaluation function gives a massive score of +10000 to the evaluation to guarantee that the algorithm will not overlook this winning move.

Other features were considered; however, upon experimenting with them, it was found that they actually decreased the performance of our agent, either by actually failing to win or increasing the number of turns before it would win.

Alternative Design Ideas

The Monte Carlo Tree Search (MCTS) agent developed for the Freckers game was evaluated against a standard minimax-based agent. In testing, the MCTS agent consistently lost within approximately 60 turns, demonstrating a significant performance gap. This result highlights several structural weaknesses in the MCTS implementation and its unsuitability in its current form for the competitive environment of the game.

Firstly, the agent performs a fixed number of simulations (500 per move), which is insufficient for producing reliable action evaluations in a game with a branching factor as large as Freckers. MCTS relies on the law of large numbers: the more simulations it performs, the more accurate its value estimates become. With such a low simulation budget, the search tree remains shallow and underexplored, limiting the agent's ability to distinguish between strong and weak moves.

Secondly, the agent replaces standard random rollouts with a heuristic evaluation of a single state reached after expansion. While this approach is computationally cheaper, it sacrifices the core strength of MCTS - averaging outcomes over simulated gameplay. Using a single heuristic evaluation per simulation leads to biased and noisy estimates, particularly if the heuristic fails to capture key positional nuances. Moreover, the heuristic function currently applies the same evaluation formula across all game phases, without dynamic weighting or context awareness.

Another issue lies in the move selection mechanism. During the backpropagation phase, the agent uses a heuristic score as a proxy for win probability, and during final move selection, it defaults to choosing the most visited node (rather than the one with the highest win rate), potentially favouring frequently explored but suboptimal branches.

In contrast, the minimax agent evaluates the game tree deterministically and uses alpha-beta pruning to consider deeper consequences of each move, making it more strategic under limited computational budgets. In our head-to-head tests, we found that the MCTS agent consistently won against a random agent, however lost against the minimax agent regardless of which side it was placed on.

Overall, the MCTS agent's shallow exploration, reliance on static heuristics, and suboptimal move selection explain its poor performance relative to the minimax agent.

Testing Against Agents

In order to better evaluate the performance of our agent, we simulated tests wherein our agent would face off against other agents across multiple trials. From these tests, we were able to deduce a clear winner: Minimax with Alpha Beta Pruning.

The Minimax with Alpha Beta Pruning agent consistently beat our other agents, as evidenced in the following results:

- Random Agent - consistently beat at around turn 40 over 20 games
- MCTS Agent - consistently beat at around turn 60

Additionally, when testing computational efficiency of the Minimax Agent against the MCTS Agent, we observed that the Minimax Agent was significantly faster due to the 3-ply constraint that we had set on it, in combination with the large branching factor of Freckers, which made MCTS much slower, whilst making worse decisions.