COMP30024: Report Project Part B - Hanzel and GretAIl

Our Approach

Our agent utilises a minimax search strategy with iterative deepening, alpha-beta pruning, and hard coded opening moves. We made this selection in consideration of Freckers' environmental factors, being a fully observable, deterministic, sequential and discrete game. The typical constraining factor of a minimax agent, for scenarios where consistently reaching terminal depth is not possible, is the accuracy of its evaluation function in estimating the ordering of best moves. However with the environmental factors of Freckers, we can consider all consequential features of any state of the game, meaning an effective evaluation function can be designed.

We also reasoned that minimax's reasonable performance when implemented for chess, a game with a greater branching factor than Freckers, would allow us to achieve an adequate search depth. Our agent uses a hard-coded sequence of opening moves rather than minimax for the first seven moves as among other factors, the other player's choices have minimal impact in this stage of the game. We developed this sequence by compiling an opening book of moves and identifying the most successful strategy.

Our implementation of minimax features slight additional modifications (see 'Other Aspects' for further details) to improve the likelihood of pruning during search.

While we could have incorporated machine learning in conjunction with minimax for strategies such as TDLeaf(Lambda) or Temporal Difference Learning, the details of their implementation were beyond our knowledge and we reasoned the payoff would be insignificant in comparison to the time investment. Instead, we tweaked the weighting of the factors of our evaluation function manually by tracking.

With more time, we would have liked to experiment with forward pruning strategies to achieve a greater search depth, such as by decreasing depth on poor performing branches (late move reduction) or prematurely narrowing search to the most promising candidates (beam search).

Additionally, also inspired by chess programming, we wanted to incorporate a transposition table into our solution, but ran out of time. A transposition table stores the evaluation of boards we have already seen, speeding up our search significantly, but even more importantly allowing us to re-use information from our iterative deepening search to pick more promising moves first, greatly speeding up our search since we can prune far more branches.

Performance Evaluation

We believe our agent performs very well overall. After watching many of its games, we aren't sure either of us could beat it consistently. It appears to perform very well at all stages of the game and almost never makes obviously horrible moves.

It always wins against all earlier versions of our agent, including others incorporating minimax search, though with a slightly different evaluation function and no opening book. When we were fine-tuning our agent's parameters and opening sequence, we found the one we submitted wins consistently even against very similar agents.

We search for around 1.25 seconds per move and this usually gives us around a ~4ply search from post-opening book, increasing to up to 10ply for the last couple moves of the game. We believe this is sufficient for the agent to make consistently strong moves with our evaluation function.

The only issues we appear to have with our agent is due to the fact we don't directly consider lilypad positions in our evaluation function. This leads to the occasional situation where the agent strands some frogs behind some water and has difficulty getting them to the back row efficiently. The LONELY_WEIGHT parameter in our model partially addresses this by discouraging the agent from leaving frogs behind by providing an incentive to move frogs further back. While it certainly significantly improved our agent, the problem still persists. We could address it by using information about available moves and potentially paths for individual frogs to the end row in our evaluation function but we thought it would be more effort and take more computation time than the benefit it would bring so did not pursue it.

Other Aspects

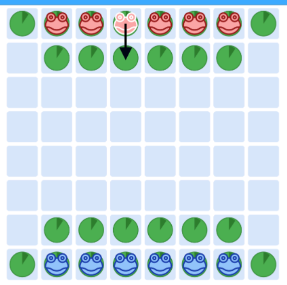
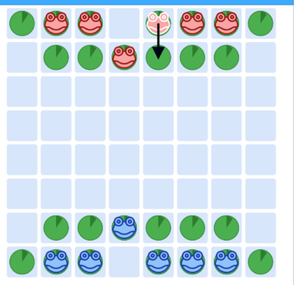
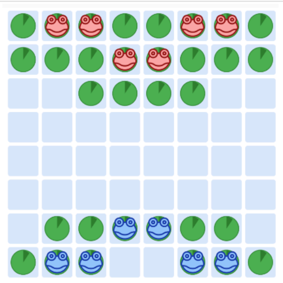
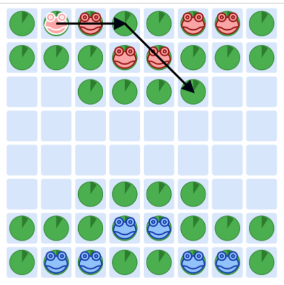
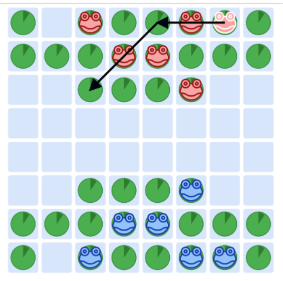
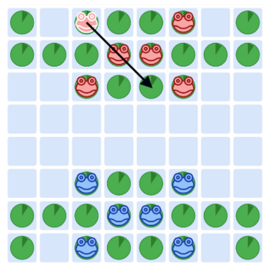
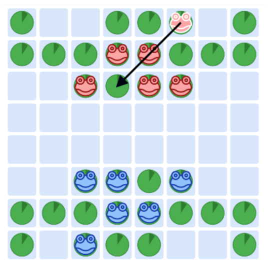
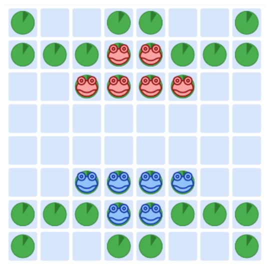
We represented the state of the game using a GameState object. This contains a 64 length character array representing the board ('R' for red frogs, 'B' for blue frogs, 'L' for lily pads, '*' for water), and two arrays containing Frog objects to store information about the frogs, most importantly their location which allows us to avoid having to loop over the entire board to find the frogs every action. We also have a DirectionOffset enum class assisting this, which is analogous to the Direction class the referee uses but tailored to this board.

Our GameState class contains lots of very useful utility methods, including one of retrieving adjacent squares that are in bounds, applying grow and move actions, and calculating the evaluation of the board. This allows our code to be much more readable and efficient than if we used the referee's board implementation, and made it significantly easier to implement our search algorithm

We had our own representations of the Actions in the game. There is an abstract Action superclass, together with Grow and Move subclasses, alongside Step and Hop subclasses of Move. All Actions contain an evaluation attribute, which stores the evaluation of the board post taking the action. Move objects also store the start and end index of the moves, alongside the direction(s) that make up the move. This allows us to easily and efficiently mutate our board during our minimax search. We also have a method to convert each of our actions to the corresponding GrowAction / MoveAction for returning the move we decide on during the search each turn.

Since we are using alpha-beta pruning, we sought to search moves that were likely better first. Because of this, when generating the moves to search for each turn, we search all hop moves, followed by step moves, followed by grow actions, since we believe that in general hop moves are very good, with grow usually only useful when there are few other options. This facilitates pruning so ideally our algorithm is significantly more efficient.

Inspired by chess AIs, we hardcoded an optimal opening sequence into our solution (pictured on the right). We reasoned this would be effective as for the first ~10 moves of the game, the players are not directly interacting so have little influence on each other. Furthermore, our minimax agent cannot see to a great enough depth to make good decisions this early into the game. We experimented with a number of opening sequences, comparing otherwise identical agents against each other until we found one that we are happy with that outperforms everything else we could think of. Since we calculate how much time to spend on moves by dividing the remaining time by the remaining moves, hardcoding the first few moves also allows us to spend more time searching on critical moves later in the game.

Move Number	Play	Move Number	Play
1		2	
3		4	
5		6	
7		N/A Final State	

We made a number of other small optimisations, including not generating moves for frogs when they reached the end row, allowing us to search even deeper in the end game since there are far less moves to search, since moving a frog already in the final row would almost never be optimal.

Supporting Work
We did not make significant changes to the driver program. We added a few print statements to the action() method in program.py, including ones to print the depth searched, time searched for, and current evaluation of the board which were helpful in debugging and improving our agent.

Initially our evaluation function only took into consideration the sum of the distance the frogs had moved up the board for each colour. We incrementally improved this by noting issues we had with our agent, and adding specific terms to combat this. For example, we had a very significant problem with individual frogs being left behind, so we added the LONELY_WEIGHT to directly combat this, which greatly improved the performance of our agent.

We chose the optimal opening sequence by generating a lot of options which seemed decent by hand, then running agents set to use these sequences against each other until we found one that consistently one against every other opening.

Similarly, we finalised the weights of our evaluation function towards the end of the development process by making small changes then running ~10 games between our old version and new version and picking the one that won more times. We believe this process allowed us to get very close to the optimal weights for our evaluation function though it was not the most efficient way to arrive at these results.

We also manually set the time spent on each move until close to our submission, so we usually just used a low value (around 0.3s usually) so comparing different versions wasn't too tedious.