Assignment 2 Report COMP30024

Yongyou (Lucas) Yu
Zhaoyu (Joey) He

May 12, 2025

1 Foundation of our agent

Before doing anything creative, we started by building some basic agents that act as a benchmark, assessing how well our newer agents perform.

We built a naive priority-based agent. The core of this agent is to get all valid moves for all frogs for the current player. Then, it will categorise valid moves into jumps forward, sideways, and grow. It will then choose a move category based on the order: jumps→forward→sideways→grow, and choose one move from that category randomly. This is not meant to be smart, but is at least more consistent than an agent acting purely randomly, which will not help in a benchmark. By building this simple agent, we also gained valuable experience in how we could build a skeleton for an agent for Freckers games.

In one word, we built a simple agent to compare with any sophisticated algorithm we implement later. Obviously, we can always compare our agent with a new feature to the old agent which does not have the new feature, and see if we have improved based on the benchmark.

2 Our basic Minimax alpha-beta

Based on our research, we found a very similar game called Checkers. We then looked up the advanced AI agent playing checkers, Chinook (we also realise this is what the lecture mentioned), to get inspiration. Chinook used a parallel alpha-beta search algorithm, with a lot of enhancements (https://webdocs.cs.ualberta.ca/~jonathan/publications/ai_publications/aimag96.pdf). We decide to follow this approach.

We also found a chess agent using minimax called Stockfish https://github.com/official-stockfish/Stockfish.git, and it uses a more recent and advanced neural network-based evaluation function called efficiently updatable neural network (NNUE). Due to time constraint and the steep learning curve, we focused more on studying minimax-related techniques.

In our research, a parallel alpha-beta search is different from minimax alpha-beta, mainly in its parallelisation feature, where multiple cores were used instead of sequential execution. We can't really do a multiple-core implementation since we don't have a good understanding of our marking environment.

We decided to begin by using a minimax alpha-beta first without the parallelisation, and then optimise the performance after a basic minimax alpha-beta agent is developed by coming up with enhancements.

For our basic minimax alpha-beta agent. We started by creating an internal representation of the board with default lilypads and frogs. We then identified frogs for the current player, and generate all possible moves. We categorised these moves into jump moves, forward moves and sideways move. Our jump sequence are found using DFS, which is implemented using what we did in project part A.

Before minimaxing, we did a basic move ordering, in which we placed jump moves→forward →sideways→grow to make alpha-beta pruning more efficient. Then we started the standard minimax with alpha-beta pruning by recursively evaluating game states to a search depth, and using alpha-beta bounds to prune unpromising branches. We returned our best move based on score.

Our evaluation function was fairly simple but solid. We awarded 100 points for frogs reaches the destination row, and 10 points for each row a frog advances towards its goal row. The only creative thing here was that we also awarded 5 points for frogs that were in the centre of the board (columns 2

- 5). The evaluation function was linear because it just looped through all coordinates of the current state, and summed up the score based on the frogs position and the current player's colour.

Dynamic depth adjustment was used, referring to time management. We estimate how many moves are left (was 150 turns but then set to 80 because on average games end between 50-70), and then get seconds per move on average, to decide our depth used for this turn. For our basic minimax we capped search depth at 3 because thats what worked the best in empirical testing. We found odd number usually work better, and 3 worked better than 5 due to the efficiency of searching, this is a limitation of our basic searching, which we addressed later in enhancements. Our agent also includes strategic time constraint checks throughout the recursion to ensure the agent responds within allowable time limits, with a fallback mechanism to always return a valid move even under extreme time pressure. Overall, the time is set to be loose because it might work differently in different marking environments.

The time complexity of our minimax implementation with alpha-beta pruning is approximately $O(b^{d/2})$ where: b is the branching factor (average number of moves available, usually 20-30 in mid-game and way less in opening/endgame), d is the search depth. As noted in lectures, this optimal bound requires perfect move ordering. Our ordering is not perfectly efficient in empirical testing, but still a significant improvement over standard minimax but with room for enhancement through more sophisticated move ordering techniques, which we will talk about later on.

The space complexity is fairly small. Recursive call stack will need $O(d)$ space, where d is the depth. Everything in the call such as alpha-beta values, score are linear. The board/state we used is a temporary dictionary, and it doest not contribute to space complexity. For the current searching level, we have $O(b)$ moves, which b is the branching factor. b and d are both small since in Freckers, we usually have up to 30 moves for one state, and we only search up to depth 5-7 at most. So we have $O(b + d)$ for space complexity.

This basic minimax alpha-beta was what our enhancements are based on. And as you can see there are already some enhancements such as a good evaluation function, a good move ordering for pruning and dynamic search depth. It already worked really well against non-sophisticated agents like our naive agent, but we are aiming for better AI agents!

3 Major Enhancements

1. **Optimized Move Ordering:**

   In our lecture, we covered 'Good move ordering improves effectiveness of pruning'. We adapted this in our function `get_moves`, as it returns a list of possible moves for minimax to search from the first one in the list. We started improving the ordering by sorting the jump moves by the length of the sequence, assuming longer jumps are usually better moves. Then, we ordered forward moves based on proximity to the destination row, prioritising frogs closer to the goal. This gives us moves in the order the most promising moves.

   We also implemented a simple dynamic move ordering based on game phase, which in early games, we prioritised grow action just after jumps. And after round 19 we switch to normal priority.

   By improving on move ordering, the new agents started to beat the old agent in every game, by a few steps to many steps, whether as blue or red. We can also see that fewer states were explored by the new agent, confirming that the improved ordering led to more efficient alpha-beta pruning, which bringing us closer to the theoretical $O(b^{d/2})$ time complexity.

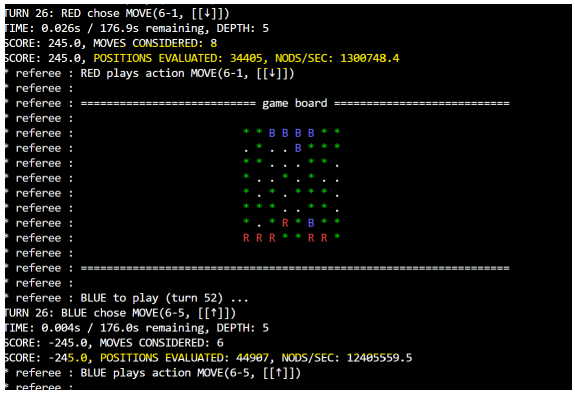

Figure 1: less state are explored = better pruning

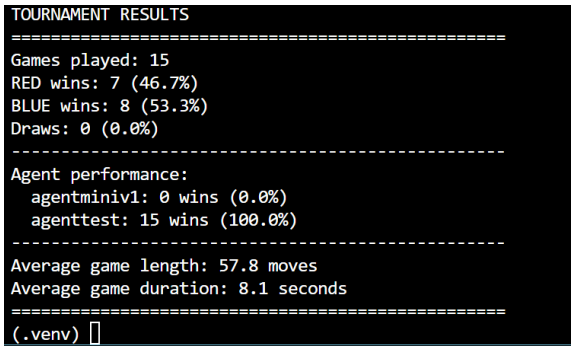

Figure 2: new agent had a 100% winrate against basic minimax agent

2. **Quiescence search:**

   We got inspiration from Chinook and the lecture to use a quiescent search in our minimax search. It worked by continuing searching for 'tactical moves' after search-depth is reached. Allowing a more in-depth search without a huge search cost.

   When our regular minimax search reached depth 0, instead of stop and returning a best move, it would continue searching by invoking the quiescence search, which extends up to 5 additional depths. It would only search for 'tactical (very important) moves' instead of all moves. In our `_get_tactical_moves` program, it only considered moves that are from either a jump sequence or moves from frogs in their 2nd last row to goal. It then sorted the jump sequence based on length like we did in move ordering in pruning. Before exploring tactical moves, we will keep the evaluation score of the current position, we will use it if no tactical moves is better or found, preventing the extended search actually worsen the performance.

   Unlike classic quiescence search, which only invoke the quiescence search in unstable nodes, we searched all leaf nodes. It was quite hard for us to define a unstable node in Freckers, like in chess, besides we found that it worked really well without unstable-nodes-only search.

   For time complexity, it will be $O(b^{d/2} * bt^{dt})$, in which bt is the tactical branching factor and dt is the depth for tactical. Note that bt and b are complete two different variables, and same for d and dt. bt stands for branching of tactical, and dt stands for depth. It looks scary but in practice tactical moves are rare, bt^{dt} will be fairly small, and close to a constant.

   Space complexity is similar except it goes up more depth, at a cap of 5, but still, it is very small. And it remains $O(b + d + dt)$ in terms of space complexity, which dt is the extra tactical moves, that is very close to a constant.

   Quiescence search allowed us to search to a great depth without having a huge performance cost, because tactical moves are very rare, resulting in a very small branching factor after regular search depth.

   After implementing a quiescence search, the new agent was able to beat the old agent with a few steps, both as blue and red.

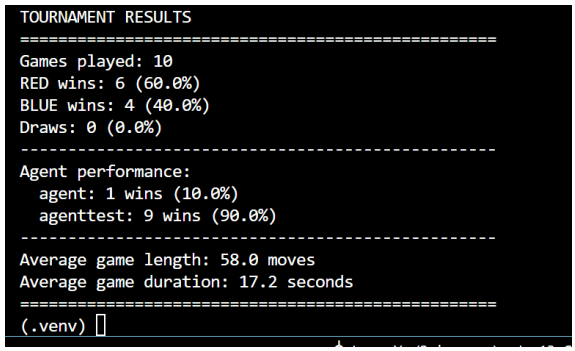

Figure 3: 90% winrate against old agent

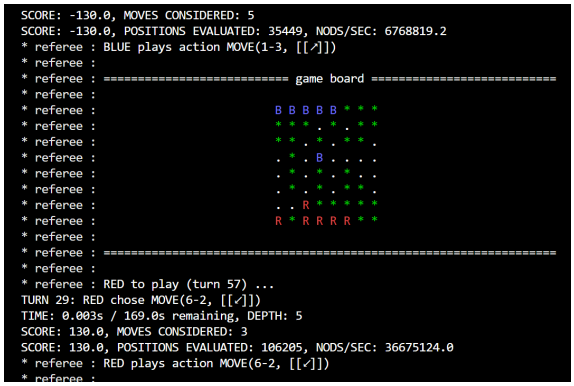

Figure 4: We explored way more states,with a high depth (5+5),when maintaining reasonable time

3. **Iterative deepening:**

   Iterative deepening search is a very common technique in chess AI. We believed this was a must for our enhancement because we didn't use all of our time effectively, which wouldn't let some of our performance optimisations pay off. The idea was straightforward, as we complete minimax depth by depth and go up as deep as possible when time allows. We started at depth 1, then depth 2, until the time constraint is reached. We store the best move from the previous depth. So in case a depth is not searched completely, we use the best move from the previous depth.

   We tuned time management once again, allocating more time budget between turns 10 to 40, because in our observations, that's where better agents start to win against other agents, making it the most crucial game phase.

   We also tuned the move ordering for pruning efficiency. Instead of ordering moves that 'seem' promising based on heuristics, we ordered moves based on their actual evaluation scores from previous depths. This evidence-based approach dramatically improved alpha-beta pruning efficiency.

   Our implementation allowed searches up to a depth of 15, and in practice, it typically reached depths of 5-7, significantly deeper than our previous agent's fixed depth of 3.

   Space complexity: similar to our previous agent except it tracks the moves score in each depth, so it is still $O(b + d + dt)$.

   Time complexity: similar to our previous agent except the depth can go up, but still at a constant level, which is about 5-7, rarely 10. And ideally it will bring the time complexity closer to our expected bound since the move ordering is closer to 'perfect'.

   The results were impressive, as our iterative deepening agent consistently defeated earlier versions by several moves. From our observations, the new agent started to gain advantage in the mid-game phase where its deeper search capability proved most valuable.

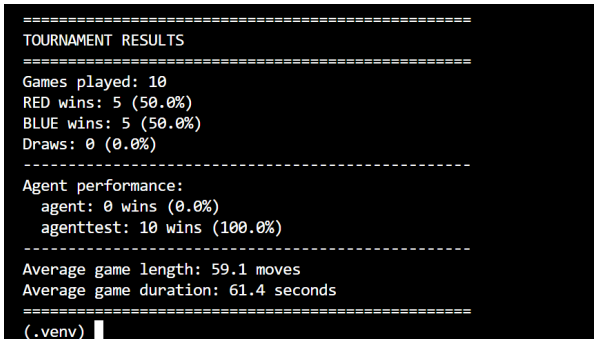

Figure 5: Iterative deepening search win-rate

Code efficiency was also crucial for making our agent stronger. We made many small improvements throughout development that boosted performance. For example, we replaced expensive `deepcopy` operations with faster regular copies. While these optimizations might seem minor individually, together they allowed our agent to evaluate many more positions in the same amount of time. This meant our agent could think "deeper" without any fundamental algorithm changes, simply by being more efficient with computational resources.

4 Performance Evaluation

We already included our performance evaluation analysis on the agents we made and the enhancements throughout the report. We will discuss our main judging factors on performance evaluation.

The efficiency of code itself is a big factor, as it allows more states to be searched, which we can look at the states and depths it searched in a given turn.

A hidden factor is a evaluation function which is hidden and can only be seen by actual play testing.

To provide a benchmark for them, the most straight-forward approach was to let different agents play against each other to determine performance. For supporting work, we also developed a simple program called `simple_tournament.py` in the root directory. This would use the referee and two agents, letting them play with each other, and alternate their colours in the game for a given number of games. It then recorded the results and print information such as win-rate, which is a main factor we consider whether something is an actual improvement.

For pruning effectiveness, we assume that with the same amount of depth searched. If one searched less states, it means it pruned more states, saving time and potentially search more states later.

5 Enhancements that didn't work in our journey

With the amount of enhancements that performed excellently in our search agent, we have more that were not working during exploring. In reality, most of the options we tried didn't work out/achieve our goal. We will highlight those that are worth mentioning, and keep it brief.

Evaluation function focused:

The evaluation function is the most fundamental feature in our search. With a better evaluation function, every search will be more accurate.

1. Mobility Assessment: We tried to add scores for positions with more moves as a bonus score, but this performed worse than our old agent.

2. Jump Opportunity Assessment: Since jumping is usually good move, we use the amount of jumping we can do in one state as a bonus score, and it failed to improve our agent.

3. Progressive Weighting for Forward Movement: We expected this to be a more accurate evaluation function because it assigns a higher score to positions near the end by adding more values each time it progresses, instead of a single linear value. However, in actual tests, it actually performed worse than the original agent. By observing the actual playout, we found that it tends to get several frogs to the end while having a few frogs in the starting line. This might benefit the opponent as they can use it as a jumping medium.

Search focused: Null-move search:

Stockfish used null-move search alongside minimax alpha-beta, so we had high hopes for it. It basically worked by giving the oppoenent a move, and if the position is so good that the opponent makes two moves and still advantageous, this is likely to be a good position. It can then prune other moves, which eliminate large sections of the search tree. However, during actual testing, it didn't outperform our old agent, and in fact had worse performance. It pruned a lot more states though, which makes the search more efficient, by suffering accuracy.

Monte Carlo Tree Search (MCTS):

Although it was listed in the specification that MCTS is hard to implement and is not required to get a decent mark, our fearless team member decided to go on this perilous path anyway.

We started by implementing the general search agent structure and adopted all the helper functions, including those for game setup, move classification. Then, we built the MCTS structure by defining nodes and coding the 4 main components of the MCTS: Selection, Expansion, Simulation, and Backpropagation.

1. **Selection:**
   Starting from the root node, we select child nodes according to the UCB1 policy, until we have reached a node that is not fully expanded or is a leaf.

2. **Expansion:**
   We expand the selected node by adding an untried child node.

3. **Simulation:**
   We play random moves until the game ends and return the result. The # of games we play is a hyperparameter set by the player, and we experimented with this number a lot to balance the performance and the time constraint. During simulations, we had a priority of jump-overs→forward moves→sideway moves→grow action.

   In our early versions, we set the # of simulations to be too high, such that the max time limit is exceeded in just turn 16. The solution was that we tuned down this parameter, and we used the referee's `time_remaining` to limit set the safe time limit per move. However, the result ended up being the model performing much worse than our most naive model, showing that achieving a balance between time efficiency and performance is difficult.

4. **Backpropagation:**
   Update the visit reward counts for all nodes on the path from the leaf back to the root.

Then we did some improvements. We first added a simple evaluation function after getting beaten by our minimax agent. Then, we added move ordering that varies as the game progresses. Based on play-outs, we sorted the forward moves by how much it helps the frog advances forward. We also prioritised grow action much more in the early phase of the game. If it is late game, we made sure that the frogs move to the opposite side first. Further more, we made sure that the agent simulate more times as we get closer to endgame, which is inspired by iterative deepening search.

As it was still getting beaten up by the minimax agent, we further added a reward shaping section that incentivises the frogs to move forward. In addition, we tuned the hyperparameters, namely the exploration constant, # of iterations and the simulation depth. To be creative, we even tried to use a hybrid approach, such as using MCTS to select a few promising moves, then let minimax to pick the best among them.

Unfortunately, none of the MCTS versions performed better than the current minimax agent if we refer to their winrates, so we called it off and decided to focus on what we have built beforehand.

```
==================================================
TOURNAMENT RESULTS
==================================================
Games played: 5
RED wins: 2 (40.0%)
BLUE wins: 3 (60.0%)
Draws: 0 (0.0%)
--------------------------------------------------
Agent performance:
  agent_mcts: 0 wins (0.0%)
  agent: 5 wins (100.0%)
--------------------------------------------------
Average game length: 69.2 moves
Average game duration: 120.1 seconds
==================================================
```

Figure 6: 0% winrate for MCTS vs. minimax $\alpha - \beta$

References

We researched and got inspiration from many chess-playing AI agents that used a minimax search alongside with other enhancements, for example Chinook and Stockfish.

LLMs were used for code benchmarking messages printing, and debugging message printing (although they were mostly removed in the submissions). They were also used for formatting and commenting to provide better readability.

In the end, we used LLM to create a file called `simple_tournament.py`, which runs a referee for a given number of times and calculates win rate of two agents competing.

AI-powered IDE such as Cursor was also adopted to develop the MCTS version of the agent `agent_mcts`.