COMP30024 Artificial Intelligence: Assignment 2 Report

Jamie A.C Marks (student no. 1382410) and Joshua Munday (student no. 1453141)
The University of Melbourne, Semester 1 2025

Section 1: Our Approach to '*Freckers*'

As an adversarial game, creating an agent to play *Freckers* was a challenging task that balanced computational depth, compute time-management, algorithmic optimisation, and storage limits. We sought to build the strongest possible agent within these bounds – in so doing, we performed broad experimentation across multiple algorithms and ultimately chose a hybrid strategy of minimax and Monte Carlo Tree Search as our final deliverable.

Before getting into this, we first discuss the agent and board classes – fundamental building blocks upon which our algorithms were built.

The 'Bitboard'

Having an efficient internal representation of the state of the game is an essential component of a good agent. This is because making an effective adversarial agent (in both the Minimax and Monte Carlo cases) depends heavily on how quickly we can simulate game playouts and perform move generation.

To have the most efficient use of space and time we took inspiration from chess agents to use bitboards (Frey & Levy, 1994). This means that the state of each of the following piece categories - lily pads, red frogs and blue frogs - is tracked using only one unsigned 64-bit integer each. These states are manipulated using bitmasks and bit operations. This results in an extremely efficient representation that significantly improves the rate of move generation and state updating. This serves to avoid most of the overhead accrued from using the provided classes and dictionary representation that are both memory and time intensive.

In practice this meant getting from only around 200 simulations per move in the Monte Carlo tree search with a more naïve state representation that used the provided classes heavily, to something like 1200 simulations per turn option with a significantly decreased memory usage.

Equivalently, we were also able to search further in minimax (from 4-ply up to 5/6-ply on average), which allowed the system to make more educated move choices.

Agent Architecture

During implementation we realized that all the agents we were developing shared common characteristics. We therefore decided to streamline our agents into a general *Strategy* class, which we used as a base class for other agents. This allowed us to ensure important abstract functions were always defined (critically, the *best_action* function), which made it easier to swap between agents on the fly. This became especially useful while experimenting with mixing Monte Carlo and Minimax strategies.

Section 2: Our algorithms

Monte Carlo Tree Search

A Monte Carlo Tree search has a few important design decisions that drastically effect the resulting agent. The following discussion will outline the design choices that we made in our implementation of the Monte Carlo agent and the justification for these choices.

2.1.1) Tree policy

The first important design choice is the tree policy. This describes how the next level of possible moves is explored (i.e., the nodes from which you run simulations and gather statistics). We decided to use a novel variation of the traditional UCB1-based tree policy, incorporating progressive widening.

Usually, this process involves exploring the game tree layer wise, fully expanding a layer before traversing down the tree, choosing new nodes based off the UCB1 formula until we find a layer that hasn't been fully expanded yet. However, when the branching factor is very large – like in the midgame – the requirement of total expansion leads to myopic behavior.

To address this, we used a progressive widening approach (dynamically expanding children based on node visit counts). This meant that at each node, we only expanded a new child once the parent has been visited enough times relative to the total explored children. This ultimately led to a more informative game tree that gave generally better results, especially when the branching factor was high.

2.1.2) Rollout policy

After choosing a node to run simulations from we needed to decide how these rollouts (simulations) would be done. We were faced with a trade-off between having *useful* simulations and having *enough* simulations. We found that if you use a stripped-down rollout policy like completely random move choice, most simulations resulted in a draw. As such, these simulations were essentially useless in informing move-choice.

Conversely, if we strongly guided the rollouts with a heuristic, we were more likely to get rollouts that ended up with a decisive outcome (win or loss). However, the extra computation decreased the total number of simulations we could do at each turn choice by a considerable amount, thereby weakening the key strength of Monte Carlo Tree Search: having tangible simulation evidence.

Through extensive testing we found that using a light epsilon-greedy heuristic led to the best outcome. This is a compromise between quality and quantity of simulations. About 80% of the time, we chose a truly random move (prioritizing the simulation strengths of the Monte Carlo strategy), the rest of the time we greedily chose what the best move in our current situation is. This approach was chosen ultimately because of how strong a purely random greedy agent is, we saw this during experimentation.

2.1.3) Choosing the next move

After running all these simulations from the node, we chose using the tree policy discussed above. However, we also needed some way to decide what move the agent should take next, given limited information. The way we decided to do this is using a slight variation of the UCB1 formula. This looks like:

$$\text{UCB1}_{\text{altered}} = \text{argmax}_{m \epsilon move\_nodes} \left[\text{winrate}(m) + c \sqrt{\frac{\ln(\text{parent\_visits}(m))}{\text{visits}(m)}} + \frac{\text{bias}(m)}{\ln(\text{visits(m)})} \right]$$

Here, the bias term corresponds to some prior idea of how good a certain move is based on the criterion:

1. Frog cohesion (how dispersed are the frogs after the move in question)
2. Vertical gain from the move
3. The factor growth in possible moves that we get from committing to this move.

As a node is visited more, this bias term reduces, meaning the equation converges to the usual UCB1 formula. This helps early in the game when we don't have a lot of simulation 'evidence' to go off. We then encoded our domain knowledge into the bias term to improve move-choice in such situations, evidentially this results in a much stronger early game agent that makes fewer questionable moves.

2.1.4) MCTS Takeaways

During our experimentation we found that MCTS only performed well when we included a heuristic for *both* rollouts and action selection, as discussed above. A pure Monte Carlo strategy with truly random rollouts and no infused domain knowledge often led to situations where the agent was wasting moves and routinely losing to a random greedy agent. We took this result as evidence that a minimax strategy (which fundamentally relies on heuristics and evaluation functions) may be a promising option in making a smarter agent. Despite this, we ended up with an extremely strong Monte Carlo agent that won 97% of the time versus a random greedy agent and made what we believe to be the strongest possible move sequence winning by around the 45-50 turn count.

Minimax with α-β pruning

Having developed our initial (traditional) MCTS strategy, we were frustrated by inconsistent results against a greedy agent. We hoped that minimax might be able to better make such compromises, since it is able to perform the same maximization that greedy does, with better insight into which specific moves are maximally advancing.

While minimax is a useful strategy, it also falls victim to the massive branching factor of *Freckers* (as with many other adversarial games); on a given move, there can be up to 30 moves available, so over a 3ply search you could be dealing with up to 27,000 move-execution operations – each of which involving move caches, move execution, move findings, and board evaluation.

2.2.1) Algorithmic Choice

To reduce the computational load and allow us to search deeper, we implemented alpha beta cutoff search; this is a strategy which eliminates parts of the search tree which are guaranteed to return suboptimal results against perfect play. α-β beta minimax can do this in two ways:

1. By finding all terminal states, assigning values based on the winner, and pruning based on guaranteed wins/losses. This method, while technically ideal, is infeasible – we could not possibly expect our algorithm to find terminal states more than 7/8 moves into the future, due to the game tree's exponential growth.
2. A heuristic function (α-β minimax + cutoff). As a compromise for brute-force search, the algorithm can search to a certain depth (i.e a cutoff) and then assess the quality of a given position via a heuristic. The cutoff search then seeks to choose the path which maximizes the value of this heuristic (if unable to find actual terminal states).

2.2.2) Heuristic Tuning

On paper, this is a great optimization, because if we can immediately tell whether a position is "good" or "bad", then we can shave off huge amounts of computing-power. In practice, however, game states can be complex and difficult to characterize. For example, a heuristic that purely prioritizes advancement may fail to evaluate a strong blocking move. Equally, we may know that double jumps are important, but the question of **how important** is difficult to answer.

To find an optimal balance, we implemented a hill-climbing approach, wherein we fed our system the raw heuristics, and let a program test different weights, reinforcing them following wins, and weakening them following losses. A major challenge of this approach was improvement-attribution. To address this, we used something called an advantage-vector; essentially, we tracked how much each heuristic changed over the course of a game and then attributed results based on how "influential" each heuristic was to the outcome. Specifically, after every move, we stored the values of every heuristic (including their weights) before and after moving and used the total variance of each heuristic as an input.

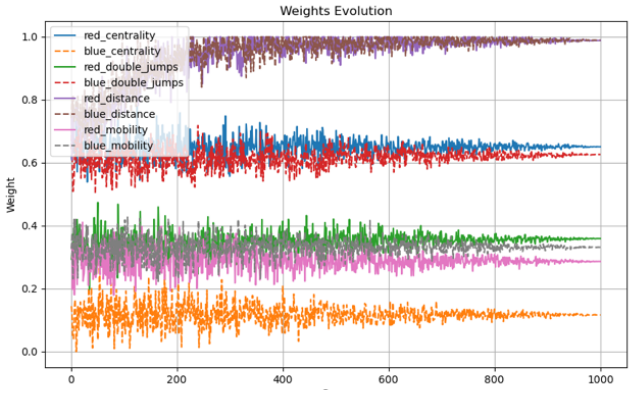

Figure 2: Hillclimbing heuristic weight tuning

While this strategy was somewhat effective, but it ultimately struggled to derive significant signals from the data. The stochastic nature of outcomes made it incredibly difficult to reinforce most signals, except for the strongest – vertical distance gained. Ultimately, distance trended to a weight of 1, while all other heuristics were unaffected. As such, we ultimately opted for a simple distance-based evaluation function, with

the aim that a simplified evaluation step would allow us to see deeper – more useful than a complex eval function.

2.2.3) Neural-Network Evaluation

We then began looking for alternative methods to evaluate the board; one compelling method was the use of a convolutional neural network to classify board states as "winning" or "losing". To build a dataset, we made use of our "bitboard class", which used bitmasks to store the entirety of the board within the space of 3 integers. We then simulated hundreds of self-play games, resulting in a dataset of over 20,000 labelled positions. While we achieved strong performance in training (≈95% classification success of winning/losing positions), the neural net was not very effective in minimax. We surmised that this was due to evaluative errors, which caused strong moves to be erroneously pruned.

After some reading, we learnt that neural nets were better adapted to Monte Carlo, which deals with the probabilistic labels of the net far more capably. This proved true; MCTS was able to comfortably beat the greedy agent while evaluated on the neural network. However, the major caveat was conversion of the pytorch neural net into a python-interpretable object. Since we had to use manual operations for each layer, the computation-cost of the evaluation function (even for small networks) reduced simulations enormously. The alternative (rewriting the conversion in C and porting over to python) was too time expensive. As such, we opted to use the abovementioned distance eval function instead.

2.1) MCTS-Minimax Hybrid

After experimentation, we made the interesting observation that while Minimax was stronger in the latter parts of the game, MCTS was generally better at creating a favorable arrangement of frogs in the early game. This is likely a result of minimax having foresight into the long-term effects of early moves by simulating game outcomes. Minimax, conversely, lacks this foresight as it relies heavily on a shallow game tree and simplified evaluation function to guide itself.

Fortunately, we had already prepared for this via the Strategy class (section 1), which allowed us to hot-swap different game playing agents during a playthrough. Our hybridization strategy was very simple: after a set depth, we swapped agents. This ultimately formed our final (submitted) agent.

This agent proved to be our strongest yet, beating our ensemble of agents (section 3.1.1) with an 80.2%-win rate, usually winning in under 60 moves. This is extremely impressive considering that this ensemble included pure Monte Carlo and Minimax strategies that are very strong.

Section 3: Performance Evaluation

3.1) Our Ensemble of Agents

Evaluating the performance of our agent effectively is deceptively hard. We had to try not to 'overfit' to a certain opponent strategy while also ensuring that our agent would be able to hold up to smart adversarial agents.

3.1.1) The 'test set' of agents

To test our adversarial agent comprehensively we used a suite of agents that we developed these include the following:

1. Minimax with alpha beta pruning (as described above)
2. Monte Carlo Tree Search agent (as described above)
3. A truly random agent: choses a random move from the set of legal moves
4. A random greedy agent: choses a random move over a weighted distribution of the set of legal moves (the weights being the vertical gain from that move).
5. A hybrid Minimax/Monte Carlo agent

That way for both our Minimax and Monte Carlo strategies we had 4 other agents to benchmark the performance of the agents, identify pitfalls in our implementation and prototype new ideas.

A striking observation in this testing phase was that the random greedy agent is surprisingly strong. Its strong bias for multi-jumps meant that the agent could slot frogs in the target row very quickly, (ignoring the lack of foresight) which turned out to be a surprisingly strong strategy. This supports our intuition about the importance of the distance gained heuristic in Minimax and Monte Carlo agents.

3.2) Implementing Self-Play

Since we sought to maximize performance, it naturally followed that we needed to evaluate different algorithms against each other. However, it was difficult to 'eyeball' which of two agents was a stronger player. Furthermore, the stochastic nature of playouts meant that even watching playouts manually for extended periods of time did not necessarily tell the full story.

Instead, we used a separate file – simulate.py – to repeatedly play agents against each other. We also needed a way to record the winner of playout games, and we used a separate tracking file **eval.txt**, to record outcomes over a series of playouts. We could then set a pairing to run for several hours at a time, or overnight, and thereby gain much stronger estimates of performance. Self-play was also instrumental in other parts of our development, including developing our neural network evaluation function (2.2.3)

Over the assignment period, we simulated thousands of games for various purposes, which allowed us to make important strategic decisions with confidence. We also used this self-play mechanism for tuning heuristics – for example, when combining Minimax and Monte Carlo Tree Search, the switching depth was difficult to gauge; instead, we simulated a suite of playouts between agents with different max depths and chose the optimal switching time.

Section 4: Optimizations

While we have made note of some of our optimizations in previous sections, there were a few key improvements that we will highlight again here. The first is an important observation for Minimax:

4.1.1) Randomness

We originally assumed the greedy agent's strength came solely from prioritizing advancement. However, while workshopping minimax, we had the insight that it might also be the random selecting that made it a strong agent. This is because our program had a very rigid move-processing order, which meant that Minimax was forming homogenous positions that cramped its position. Once we randomised the input order, however, Minimax became significantly stronger.

4.1.2) Lazy move generation

Using this random move generation also allowed us to utilize lazy move generation. This meant that a move is returned as soon as it is found, after randomizing the order of move directions considered. After this we chose between the move we just found and a *GrowAction* by sampling from a distribution that is representative of the actual distribution of *GrowAction*'s and *MoveAction*'s. This has (very close to) the same result as generating all moves and choosing a random one, without a lot of the computational work.

This practically means that we can avoid generating all moves during minimax effectively decreasing the computational work by a factor around 40 and increasing the simulations done in MCTS by from around 400 to 800 per move.

4.1.3) Time budgeting

Budgeting time was also a significant yet crucial challenge for getting the most out of our agent. The way that we decided to do this is by having a heavily early-game skewed time allocation throughout the play of our agent.

This meant that we were assigning large time chunks to our agent in the early game which geometrically decreased in size leading up to the end game. We did this as a way of 'hedging our bets'. We know that our agent is strong and wins in sub 60 turns about 80% of the time, so we can allocate most of our time to the early game where we **know** we are going to use the time. However, if we get unlucky and the game goes for all 150 turns, we will still have enough time to make informed decisions at each one of those turns. This improved the performance of our agent significantly.

4.1.4) Move caching

We used a simple Least Recently Used move cache to store moves within the *Bitboard* class based off the state of the board. We did this to save as much computational time as possible as MCTS often considers the same board state multiple times and having a cache cuts down the total time spent in move generation drastically.

References

Frey, P. W., & Levy, D. N. L. (1994). *Programming a chess computer*. Springer Science & Business Media.