Project Part B Report

A - Search Strategy

Our agent uses two complementary search-based decision-making strategies: **Monte Carlo Tree Search (MCTS)** and **Minimax** search. This setup enables us to compare the trade-offs between selective simulation-based search (MCTS) and depth-limited exhaustive search (minimax).

1 - Monte Carlo Tree Search

Motivation

Since MCTS adaptively deepens its search along promising lines, it is well-suited for Freckers, where strategic advantages—such as setting up multi-jump chains—often require several turns to unfold. The value of these actions may not be immediately apparent, but their impact accumulates over time, eventually leading to a winning advantage.

Algorithm Description

At each decision point, MCTS runs a number of simulations starting from the current game state to determine the most promising action. Each simulation proceeds by traversing the tree using the **Policy-based Upper Confidence bound applied to Trees (PUCT)** formula. At each state s, the agent selects the action a that maximizes:

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)}$$

where $Q(s,a)$ is the average value of action a at state s; $P(s,a)$ is the prior probability of action a, which guides initial exploration; $N(s)$ is the total number of times state s has been visited; $N(s,a)$ is the number of times action a has been taken from state s; c_{puct} is an exploration constant controlling the trade-off. This balances exploitation of actions with high average value $Q(s,a)$ and exploration of actions with high prior $P(s,a)$ but low visit count.

The search continues recursively until it reaches a leaf node—a state that has not yet been expanded or is terminal. Its value is computed and then backpropagated through the search path to update action statistics:

$$Q(s,a) \leftarrow \frac{N(s,a)\cdot Q(s,a) + v}{N(s,a)+1}$$ where v is the newly backpropagated evaluation value.

To avoid redundant computation, the agent caches key information in dictionaries:

- For each pair of state s and action a: $Q(s,a)$ and $N(s,a)$.
- For each state s: $N(s)$, the prior policy vector $P(s)$, the terminal value of the game $E(s)$, the set of valid actions $V(s)$, and the predicted win probability $W(s)$.

After simulating, the algorithm selects and returns the action from the root state with the highest visit count.

Modifications & Creative Aspects

Policy prior biasing

A key improvement of PUCT over UCB is the incorporation of policy priors $P(s, a)$, which guide the search toward promising actions, which is especially valuable under a limited rollout budget. At the root of the MCTS tree, we combine both **model-informed** and **heuristic-based** priors.

For each valid action a, let s' be the resulting state after applying a to the current state s. We initially set $P(s, a) = f(s')$, where f is the machine-learned evaluation function described in the next section.

To encode strategic preferences, we introduce a set of action categories C_k, ordered from most to least desirable as follows: (1) Jump toward the goal over an opponent; (2) Jump toward the goal over a teammate; (3) Move (without jumping) toward the goal; (4) Grow lily pads; (5) Move sideways.

Each category C_k is assigned a multiplier m_k (e.g., $m_1 = 5$, $m_2 = 3$, $m_3 = 2$, $m_4 = 1.5$). For a given action a, which may consist of a sequence of jumps, we define the adjusted prior as:

$$P(s, a) = f(s') \cdot m(a) \text{ where } m(a) = \prod_{i,k} m_k^{I(a_i \in C_k)}$$

Here, a_i refers to each monotonic component of the action sequence. This formulation allows a sequence of jumps to accumulate priority multiplicatively, leading to strong (exponential) promotion of aggressive moves.

To discourage inefficient exploration under tight simulation constraints, we assign $P(s, a) = 0$ for sideways moves unless no higher-priority options are available.

The resulting policy vector $P(s)$ is then renormalized to sum to 1.

Leaf Evaluation

When the search reaches an unexpanded leaf node, the agent queries a trained model to predict the probability of winning from the current board state, rather than simulating the game to the end. This machine-learned evaluation allows for faster assessments under tight runtime constraints and leverages strategic knowledge acquired through self-play. Details of the evaluation model are presented in section B.

Search Scaling Across Game Phases

To allocate runtime more effectively, our agent dynamically adjusts the number of simulations and the exploration constant c_{puct} based on the game phase.
We allocate the most simulations (around 200) during the **midgame** (from the 10th turn to the 50th turn), where interactions between teams are most complex. This is when players attempt to capitalize on opponent positions—such as setting up or defending against multi-jump chains—so decisions must be carefully considered. A higher simulation count helps reduce noise and explore a range of possible outcomes.
In the **early game** (before the 10th turn), fewer simulations (around 100) are used because the game can unfold in many different ways, and deep simulations may not be very informative due to the high variability in how the match could develop. However, we still allocate enough to begin shaping a strong strategic position for the midgame.
In the **endgame** (after the 50th turn), the number of simulations is also lower (around 100). By this point, actions are typically straightforward, and earlier simulations have already covered many relevant state-action pairs, allowing us to reuse cached evaluations.

2 - Minimax

Motivation & Algorithm Description

Since Frecker is a deterministic, perfect-information game, Minimax is well-suited. We implemented a Minimax agent with alpha-beta pruning to rival MCTS. Given the large state space, our algorithm limits search to a fixed depth and uses an evaluation function to estimate which player is favored.

The evaluation is a weighted sum of five manually tuned features: number of frogs at the goal, average row distance, number of jumps, internal distance, and number of frogs without available moves. The final score is computed as red's score minus blue's, allowing red to maximize and blue to minimize.

Modification & Creative Aspects

Transposition table

Since only one of six frogs moves per turn, the search often revisits previous board states. To optimize performance, we use a transposition table, which is a Zobrist hash table that stores evaluated states along with their depth, pruning status (alpha or beta), and return value. If a known state is revisited at a depth less than or equal to the stored one, the evaluation result is reused to avoid redundant computation.

Using a transposition table, our Minimax agent extends its look-ahead depth from 4 to 6 plies while exploring nearly half as many nodes,with a cache hit rate of approximately 25%.

Other improvements

Branching Factor Reduction: To reduce the search space, we excluded wall moves (unless they are jumps) as they limit future options.

Move Ordering: We prioritized promising moves like jumps and forward moves to appear earlier in the action list, enhancing alpha-beta pruning by cutting off weak branches sooner.

Dynamic Depth Adjustment: Search depth was adjusted based on game state; for example, frogs in central rows (e.g., row 3 for red) allow deeper searches due to higher interaction potential, improving time efficiency.

B - Evaluation Function

1 - Overview

To evaluate game states during search, both our MCTS and Minimax agents use shared handcrafted features. MCTS uses an XGBoost evaluation model that takes a total of 39 features as input and predicts the probability that the player will eventually win in the given state.

2 - XGBoost Features

We extract 15 features for each team, capturing how well red and blue are performing individually. However, since winning also depends on relative advantage, we include 9 additional features representing the difference between red and blue's corresponding feature scores.

Feature Group	Feature Name	Motivation & Description
Team-specific features	frogs at goal row	To indicate progress towards winning (more frogs at goal = more likely to win).
	avg row dist to goal, avg min euclidean dist to goal	To measure how close all frogs are to the goal row in both cases where the goal lily pad is available or not (all need to be at goal to win).
	near goal ratio	To measure how many frogs are near the goal row (ensuring none are too far away).
	jumpable ratio, target jump ratio	To evaluate how many frogs have immediate jumping options and how many jumps can be made (more jumps = likely reach goal faster)
	interaction score = avg row dist * spread root-mean-square	To account for spread needs as row distance changes: frogs stay close when row distance is high, but must spread out when it's low to reach different goals.
	edge position ratio	To determine how many frogs are at board edges.
	col centrality score	To measure frog distribution across the board's columns, which is important in early moves.
	avg col	To measure the frogs' overall horizontal position in one value or whether frogs are clustered near the center or one side.
	spread variance	To quantify frog dispersion (higher variance = more spread).
	assistable ratio	To check for jump support among frogs.
	avg reachable pads	To measure frog mobility (average number of reachable pads).
	grow needed ratio	To assess when "grow" actions are required (can be an advantage/disadvantage depending on the situation).

	goal pad ratio	To measure the abundance of goal pads (more = less "grow", faster progress).
Delta features	delta #frogs at goal	To evaluate which side is gaining a frog advantage at the goal (more frogs than oppenent = likely to win).
	delta avg euclid dist	To evaluate which side is reducing their distance to the goal (less distance than opponent = likely to win).
	delta target jump ratio	To evaluate which side is gaining more jumping opportunities (more jumps than opponent = likely to reach goal faster).
	delta avg col	To measure how horizontally close two players’ frogs are.
	delta avg reachable pad	To evaluate which side is having more move options.
	delta col centrality	To evaluate which side is closer to the center column of the board.
	delta interaction	To assess which side has better strategy for the current game state.
	delta near goal ratio	To assess which side has more frogs near the goal.
	delta grow needed ratio	To assess which side needs more grow actions.

Table 1: Features description

3 - Model

Model Selection

The model used for prediction is an **XGBoost classifier,** which is a type of ensemble model that combines many shallow decision trees (weak learners) in a gradient boosting framework. It is suitable for modeling complex, non-linear relationships between features and outcomes, while remaining relatively fast to train and evaluate. In addition, since we work with a relatively large set of features, XGBoost is attractive for its ability to perform automatic feature selection, focusing only on the most informative features during training.

We selected XGBoost over other models we experimented with, such as logistic regression and feedforward neural networks, for both performance and practicality.

Logistic regression was too simplistic as it assumes a linear relationship between features and output and lacks the ability to capture **conditional interactions** between features. For instance, if our frogs are still far from the goal, we prefer them to be closely packed (low spread variance) to assist one another through jumps. However, when frogs are closer to the goal, we instead prefer them to be more spread out to efficiently occupy separate goal cells. These types of strategic dependencies are not easily captured by a linear model.

That said, a tradeoff exists: linear models like logistic regression can easily capture **delta features** (e.g., the difference in average distance to goal between teams) due to their additive structure. Tree-based models like XGBoost, which split on one feature at a time, do not naturally capture such interactions. To address this, we explicitly include delta features in our model input.

On the other end of the complexity spectrum, **neural networks,** while attractive for their ability to perform automatic feature engineering, require an enormous number of self-play games to generate sufficient training diversity. Moreover, they are harder to tune, slower to train, and less interpretable, which poses challenges under our limited compute and memory capacities.

Training Pipeline

Our training procedure follows a self-play and iterative improvement loop inspired by DeepMind's AlphaZero (Silver et al., 2017). We run a total of 40 training iterations, where in each iteration, a new XGBoost model is trained on approximately 45,000 samples. These samples are collected from 10 previous iterations, with each contributing data from 100 self-play games played by the strongest MCTS + XGBoost agent available at that point.

Each training sample consists of a 39-dimensional feature vector, extracted from a board state, and a binary label indicating the game outcome from the red team's perspective (1 for win, 0 for draw or loss).

Once a new model is trained, it is evaluated by playing 14 head-to-head games against the current best agent , 7 as red and 7 as blue. The new model replaces the current best if it achieves a win rate above 60%.

By continually generating stronger training data with the current best agent and discarding the oldest data, the training set theoretically improves over time. This ensures each XGBoost model is trained on increasingly relevant examples. In addition, it is only accepted if it meaningfully outperforms its predecessor, which reinforces a cycle of self-improvement.

Online Prediction

Due to environment constraints, we converted the XGBoost model into a NumPy array of trees. The model is first saved in JSON format, which is then parsed and transformed into a NumPy-based representation during loading. Each tree is stored as a dictionary of arrays with key fields: left_children and right_children store indices of the left and right child nodes respectively, using -1 to indicate leaf nodes; split_indices contains the feature indices used for splits; split_conditions holds the threshold values for splitting; default_left is a boolean array specifying whether to go left when values are missing; and base_weights provides the prediction values at each node, primarily used at the leaves.

The nested JSON format for each tree was transformed into a set of parallel arrays describing the tree structure, making traversal significantly more efficient. We programmed a speed test for making 100 predictions using both the original JSON structure and the new NumPy array structure. The new format is approximately 100 times faster.

C - Performance Evaluation

1 - Features & Model Evaluation

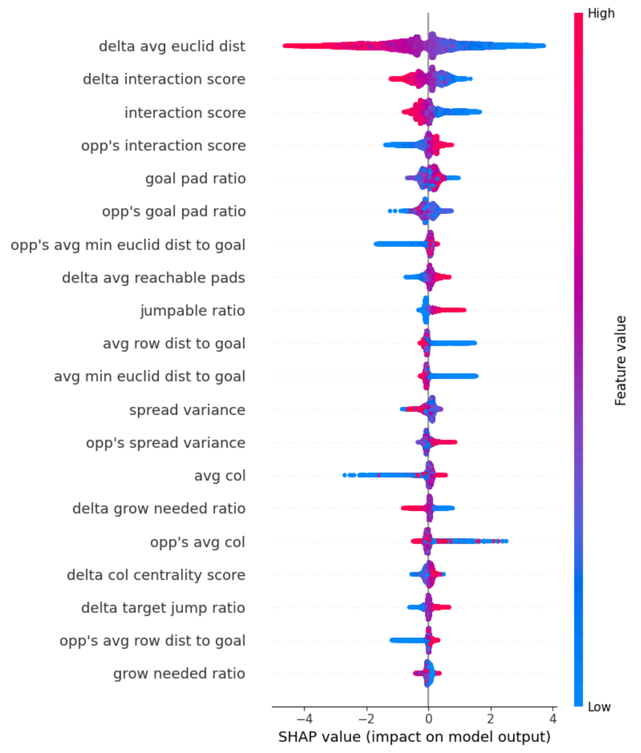

Figure 1: Feature importance (top 20)

Metric	Train	Test
Log loss	0.346	0.426
Accuracy	84.2%	78.2%
Brier score	0.110	0.142

Table 2: Model performance metrics

From figure 1, the most predictive feature is delta avg euclid dist, which captures how much closer the player is to the goal than the opponent. As expected, a lower value strongly correlates with higher win probability. This is followed by interaction scores, which measure how scattered and distant the frogs are from the goal, which are indicators of poor team coordination and lower success rates. The high importance of delta-based features (e.g., delta avg reachable pads, delta grow needed ratio) suggests that relative positioning matters more than absolute team-specific features.

From Table 2, the model achieves a test accuracy of 78.2%. This is a solid predictive performance, especially considering that early-game positions, which account for roughly 25% of the dataset, are inherently more uncertain. A Brier score of 0.142 indicates well-calibrated probability estimates, which is important in this setting, since MCTS relies on comparing win probabilities between states, not just binary outcomes. The moderate gap between training and test metrics suggests good generalization with limited overfitting.

2 - Agent Evaluation

When selecting weights for Minimax and training MCTS with different features and opponents, we compared each new model to the previous one and accepted it only if it won from both sides. To evaluate the final performance of our MCTS with XGBoost agent, we tested it against five agents: a random agent, a smarter random agent that always randomly chooses a forward move if possible, and a greedy agent using the XGBoost evaluation function. For deterministic agents like Minimax, we played two games, switching sides. For agents with randomness, five games were played to account for randomness. Running on our machine, the results showed that:

Agent	Opponent	Winrate	Notes
Minimax (3-5 ply)	Random Agent	100%	Minimax took an average of 60 turns and 12s.
	Smarter Random Agent	100%	
	Greedy Agent	100%	
MCTS+XGBoost (40-80 iterations)	Random Agent	100%	MCTS took an average of 66 turns and 25s.
	Smarter Random Agent	100%	
	Greedy Agent	100%	
MCTS+XGBoost (40-80 iterations)	Minimax 3-5 ply	50%	All games lasted about 58 turns. MCTS took roughly the same time as Minimax with 5-6 ply.
	Minimax 5-6 ply	50%	

To conclude, our MCTS (40-80 iterations) with XGBoost shows the same performance as Minimax with 5 to 6 ply. Moreover, we can increase the number of iterations in MCTS with XGBoost to 100-200 while still staying within the time constraints. In contrast, Minimax with 5-7 ply must reduce its depth to 3-5 after 40 turns to maintain acceptable runtime. Therefore, we selected MCTS+XGBoost as our final agent.

References

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. and Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, [online] 550(7676), pp.354–359. doi:https://doi.org/10.1038/nature24270.

Thakoor, Shantanu and Nair, Surag and Jhunjhunwala, Megha (2017). GitHub - suragnair/alpha-zero-general: A clean implementation based on AlphaZero for any game in any framework + tutorial + Othello/Gobang/TicTacToe/Connect4 and more. [online] GitHub. Available at: https://github.com/suragnair/alpha-zero-general.