COMP30024 Report for assignment part B

Group 5x4MJ_E_0L4:
Yanrunzhe Lyu, Zhenyiu Xu

1. Introduction

In the second part of the COMP30024 group project, we are tasked with developing an artificial intelligence agent to play the game Frecker—a modified version of the two-player Checkers game, featuring restricted movement destinations to position with pads and the movement to horizontal and forwardly vertical and diagonally directions, to move all pieces to the opposite side of the board.

Through this practical investigation into the logic of gameplay and specified model construction and selection for the agent, we aim to deepen our understanding of model architectures suitable for two-player, competitive, zero-sum games.

2. Approaches

To address the task of action selection in our game-playing program, we explore and implement two distinct search-based approaches: Monte Carlo Tree Search (MCTS) and Minimax. Although the performance of these models is unequal, they still provide us with insights into strategic trade-offs. Between the two models, we select the final one by sampling results from matches where the models play against each other and identifying which performs better. Similarly, we test the performance of each model after every adaptation against its previous version, determining whether the change is effective based on overall results.

Furthermore, a random agent is also developed as an external comparator to evaluate turn-wise efficiency and to check for overfitting to strategies tailored for structurally similar models.

2.1 Monte Carlo Tree Search (MCTS)

The first algorithm we choose is MCTS. Since Freckers is a complex game with an 8x8 board and 6 frogs for each player, the branching factor can be very high. MCTS finds the best action through numerous random simulations and can potentially explore most branches without consuming excessive time, which simplifies the decision process. In addition, a good evaluation function may not be developed easily for a complex game. MCTS utilises the number of winnings, which is more accurate than evaluation. We follow the standard process of the algorithm as the following steps: select, expand, simulate, and back-propagate. During the development and testing, we find that the simulation process takes a manageable amount of time, therefore, we use the standard simulation to reach a win or end state without limiting the simulation depth and avoid using an evaluation function. In the end, we allocate an iteration limit for the whole process of MCTS to manage running time and check the performance difference.

2.2 Minimax

Another approach that we select is the Minimax Algorithm with Alpha-Beta pruning, in search depth of 5, as Freckers is a fully observable, deterministic, and turn-based game, hence it is well-suited for Minimax.

Alpha-beta pruning significantly reduces the number of game states that need to be evaluated, allowing deeper lookahead without exceeding time limits. This makes the algorithm efficient and effective at planning optimal moves by anticipating and countering the opponent's responses. To improve the effectiveness of alpha-beta pruning, we order all the moves for a game state inside minimax through static heuristic and killer heuristic. The killer heuristic theory is adopted from the paper written by Huberman (1968).

While developing the algorithm, we have also finely adjusted an evaluation function in order to improve the agent's performance as much as possible.

2.2.1 Evaluation Function of Minimax

For the Minimax algorithm to function, we develop an evaluation function. With testing and adapting during the process, our final evaluation function of minimax has these well-selected variables

- **opponent_distances** and **player_distances**: Measures the total row distances to the target row of player frogs and the opponent frogs. Since the primary goal is to reach the opposite end of the board, the value of (opponent_distances – player_distances) directly reflects the likelihood of winning and hence has been assigned a weight of 15, indicating it is the most important variable.
- **line_value**: Assesses how favourable the destination row is for scoring. It rewards cells at the target row, lily pads (+4), empty spaces (+3), and discourages movement toward cells already occupied, +2 and +1 for opponent's frogs and player's frogs. This encourages the agent to plan toward realistic and available goal paths.
- **central_adjust**: Encourages frogs to remain closer to their starting row in the early and mid-game, preventing overextension to support safer play and better coordination, especially for setting up jumps or growing pads. It had been assigned a weight of 3, making it only significant when the move options' gain is close.
- **hop_opportunity** and **hop_forward**: Hop opportunity counts the number of potential legal hops to hop over a frog available at weight of 1, and hop_forward rewards actually executing forward hops (as opposed to just having them available) at 5 weights, as hops are efficient move generally, this encourages the agent to actively seek for hop chances and use them to gain advantage
- **uneven_frogs_penalty**: With a slight deduction weight -2, this penalizes scenarios where frogs are concentrated in a single or a few columns. Maintaining a balanced spread reduces traffic and increases pad growth and jump setup flexibility.
- **blocking _penalty**: Similarly, in negative weight -3, it penalises frogs moving into columns that are blocked at the destination row. This discourages wasted effort heading toward "dead ends," unless used as a tactical pass-through.

However, not every attempt in selecting variables is effective. During the adaptation, we had to abandon some of the variables that we once believed would be supportive of our model, but returned scarcely any improvement.

- **palyer_score** and **opponent_score**: these variables are designed to measure the number of frogs that have already reached the goal line. While this might seem like a good indicator of progress, it is too binary and becomes relevant only in the late game. In practice, it added little value during mid-game evaluation and led to suboptimal prioritization of early finishes at the expense of overall control.

- **pad_count**: Counts the number of lily pads adjacent to the player's frogs. This was considered a proxy for mobility, but in practice, it overlapped heavily with jump detection and positional spread. It also misrepresented temporary situations, misleading the frog to grow or get near temporary growth areas, without directly contributing to progress.
- **threat_level**: This variable estimates how many of the opponent's frogs are capable of jumping over our frogs. While this had potential value in defensive play, it made the model overly conservative and led to excessive hesitation in making aggressive or strategic forward moves. Its cost in missed opportunities outweighed its defensive utility and ignored the potential opportunity to jump over the opponent's frog at the adjacent.

2.2.2 Heuristic

In the static heuristic of our Minimax program, Grow actions are assigned a low value of 2, while Move actions are prioritised based on the length of movement, especially for consecutive hops and forward progress. This is because movement is generally more urgent than pad-growing unless movement is blocked, as Grow does not provide immediate positional gain—it only enables future Move actions. Therefore, Grow actions are given low but non-zero priority to ensure they are not completely ignored.

The heuristic also accounts for the length of movement chains, favouring longer sequences of hops. Since each jump in a chain advances the piece by 2 units, such mobility is often effective in reaching the bottom line, making these moves strategically valuable.

Finally, moves that include forward progress are further prioritised by awarding a score proportional to the row advancement, scaled by a forward_mult factor of 4. In contrast, horizontal moves are considered less effective under similar conditions, as they do not contribute directly to progressing toward the bottom line.

The killer heuristic stores the action that generates an alpha-beta pruning in other branches at the same depth as the current position in the minimax search tree, and uses it given that the action has a high possibility to generate a cutoff in the current position (Huberman, 1968). Our killer heuristic table stores two actions for each depth: one for the action that results in the most recent pruning with the highest priority, and one for an earlier action. The killer actions are given higher priorities than the static heuristic.

2.3 Random

Besides the minimax and MCTS agent program, instead of using a logical approach to select the most effective action, we construct a list of all legal actions and select one at random. This program is a supportive agent, expected not to be effective generally, it is used as a basic opponent to play against other models to gain insight into the models' general turn-wise effectiveness.

3. Performance

To develop a deeper understanding of the models, we evaluate their performance across various dimensions using multiple tests.

The baseline criterion for considering a model effective is its ability to consistently defeat both the random agent we developed and the default agent on Gradescope.

Beyond this, we assess the model's performance based on the average number of turns required to win against the random and default agents.

More importantly, we evaluate a model's strength through its win rate against alternative approaches. To eliminate any bias from the first-move advantage observed when using the same model on both sides, we conduct tests using both colours. The turn amount performance is also very supportive in considering the model's effectiveness, as it reflects the agent's ability to win efficiently or to delay loss when at a disadvantage. By setting models to compete against each other, we also gain a more intuitive understanding of their relative computational efficiency—an important factor, as inefficiency could lead to timeouts.

We assess the impact of any model modification using similar methods: comparing win rates and turn-wise performance of the updated model against its previous version, and checking the average number of steps needed to defeat a random agent. These outcomes help determine whether a change is beneficial and worth adopting.

3.1 Monte Carlo Tree Search (MCTS)

Player	Random	MCTS 200 iterations
MCTS 200 iterations	100%, 97.5 turns	60%, 97 turns
MCTS 300 iterations	100%,103.5 turns	50%, 78 turns

Table 3.1.1 MCTS with different iterations against Random and MCTS, displaying win rate and average turns to win in 10 turns (5 Red + 5 Blue) statistics

Based on Table 3.1.1, the overall effectiveness of MCTS was suboptimal. Although it successfully defeated both the random agent, it exhibited a significant number of moves to win. On average, MCTS needed around 100 turns or more to achieve a win over the random model, based on results such as 103.5 and 102 turns in 10-game samples, which is an indication of inefficiency in selecting optimal actions. This drawback placed MCTS at a disadvantage against stronger models, often resulting in losses when tested head-to-head against the Minimax algorithm.

Player	Random	MCTS C=2.0
MCTS C=0.5	100%, 102 turns	50%, 92 turns
MCTS C=1.0	100%, 99.5 turns	100%, 101.5 turns
MCTS C=1.5	100%, 103.5 turns	50%, 106 turns

Table 3.1.2 MCTS with different exploration constant C of UCB against Random and MCTS, displaying win rate and average turns to win in 10 turns (5 Red + 5 Blue) statistics

From Table 3.1.2, the MCTS agent behaves best with an exploration constant C equal to 1, as it can beat the agent with C equal to 2. However, different C values do not help the performance against a Random agent, the average turns to win remains approximately 100. From Table 3.1.1, the performance even worsens after the iteration limit increases, since the average turns to win and the win rate decreases. The problems can be attributed to a relatively small number of iterations allowed. MCTS may require a large amount of time or iterations to approximate a good action to choose.

This underperformance likely stems from the fact that MCTS is not well-suited to the specific characteristics of Freckers. MCTS excels in environments that are partially observable, involve stochastic elements, or have vast or uncertain state spaces. Its strength lies in simulating future game states to estimate win probabilities and inform decision-making. However, Freckers is derived from Checkers, and hence, a fully observable, deterministic, and discrete game does not present the kind of uncertainty where MCTS typically thrives. As a result, MCTS could not fully exploit its strengths in this setting.

Additionally, the theoretical performance of MCTS assumes access to infinite computational resources. In practice, the algorithm is constrained by runtime limits and must terminate early, often relying on heuristic evaluations rather than reaching terminal states through full simulations. This necessity compromises its decision quality and overall effectiveness.

Nevertheless, we believe that with further refinement, particularly through better heuristic design and parameter tuning, MCTS could still yield competitive performance within this domain.

3.2 Minimax

Player	Random	MCTS	Minimax D4	Minimax D5
Minimax D4	100%, 68 turns	100%, 63.5 turns	50%, 57 turns	
Minimax D5	100%, 57 turns	100%, 58.6 turns	100%, 57.5 turns	50%, 85 turns

Table 3.2 Minimax with different depth D against Random, MCTS with 200 iterations and Minimax, displaying win rate and average turns to win in 10 turns (5 Red + 5 Blue) statistics

Controversially, the agent program with Minimax presents a very satisfying performance, achieving consistent 100%-win rates against both random and MCTS agents, with significantly fewer turns required to win at an average of 57 in comparison to 103.5 from Table 3.2.
The program with the Minimax algorithm succeeded against our MCTS approach in both head-to-head win rates and average turn counts when defeating the random agent as expected. We measured depths 4 and 5 as smaller depths, like 2 or 3, are trivial, which may not lead to good performance, and larger depth like 6, can consume a lot of running time. From the table, the performance improves as the search depth rises; depth 5 leads to faster winning and a 100%-win rate against depth 4.

The results indicate the Minimax Algorithm would be a suitable model in Frecker or similar zero-sum competitive games, as long as the heuristic and evaluation function are carefully specified for the game.

Interestingly, we find that minimax always loses when playing as Blue against itself with the same depth. This may result from the first move advantage of Freckers in real game play, since the game tends to end earlier than 150 turns, helping the Red player lead.
Considering MCTS's longer average turn counts vs Minimax (103.5 vs. 57.5) and its 0-win rate in head-to-head comparisons with Minimax, we ultimately selected the Minimax algorithm as our final model, as it proved more effective in this context.

4. Other Aspects

To improve computational efficiency, we restructured the storage method for the game board in our agent. The board is stored as a 2D integer NumPy array with integers representing different cell states for fast indexing and copying. Furthermore, we additionally store the coordinates of red and green frogs for a board state through a set of integer tuples. These modifications allow movement to be reconstructed through tuple addition between the current coordinate and the movement direction, eliminating the need for the original Coordinate class's adding function, which we found incurred a lot of unnecessary runtimes after profiling. The modification is successful and results in a notable reduction in runtime per round.

We add two types of move ordering heuristics to promote the alpha-beta pruning in our minimax algorithm, one based on static evaluation, and the other is the killer heuristic. We profile the minimax program and the time of running minimax 10 times on a complex board state decreases from 17.497 seconds to 9.305 seconds, the efficiency increases by 80%.
Moreover, our minimax algorithm is finally implemented through negamax, adapted from Knuth and Moore (1975). It only involves one player and handles the Max and Min turn alternation by passing negated game utility or evaluation supported by the equation:

$$\max(x, y, z, ...) = -\min(-x, -y, z, \ldots).$$

Negamax reduces duplicate code and makes our code clean. In addition, we add iterative deepening (ID) to our minimax program. ID increases the time flexibility since the program can stop after an iteration if the time limit is reached with only a little additional running time overhead.

5. Supporting Work

We develop a script called profile_test.py utilising cProfile library to check the calling frequencies and running time of each function in our agent program, This helps improve the speed of our program to further enhance the performance, e.g., increasing the search depth of minimax. For example, the profiling of minimax is measured by running it multiple times automatically for the same hardcoded complex board state with a high branching factor.

6. Conclusions

After thoroughly investigating the playing logic of Freckers, we developed our final model using an adapted Minimax algorithm with alpha-beta pruning and a heuristic-based action selection function, as we found it to be more suitable than MCTS.

In the game Freckers, the Minimax algorithm outperforms MCTS because Freckers is a checkers-like game that is fully observable, deterministic, and discrete, with a manageable number of possible actions. These characteristics make it more amenable to Minimax, while posing challenges for MCTS. Specifically, MCTS requires time and many iterations to gain meaningful information, but due to computational constraints, it cannot fully realise its theoretical advantages in this context.

This superiority is reflected in the Minimax model's better time complexity per move and greater efficiency in reaching goals, resulting in a higher success rate.

Therefore, we have successfully constructed a robust agent program based on the Minimax algorithm.

7. References

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326. https://doi.org/10.1016/0004-3702(75)90019-3

Huberman, B. J. (1968, August 19). *A program to play chess end games* (Technical Report No CS 106). Stanford University, Department of Computer Science. https://apps.dtic.mil/sti/citations/AD0673971

Liu, Z., Zhou, M., Cao, W., Qu, Q., Yeung, H. W. F., & Chung, V. Y. Y. (2019). *Towards understanding Chinese Checkers with heuristics, Monte Carlo tree search, and deep reinforcement learning* (arXiv:1903.01747 [cs.LG]). arXiv. https://doi.org/10.48550/arXiv.1903.01747