Assignment report

Minimax with Alpha-Beta pruning and Iterative-deepening, Transposition Table (AB Agent With Cache)

Motivation

At first, we chose pure Minimax as the algorithm of our agent since Minimax and its variations are the most popular decision-making algorithms in the world of 2-player turn-based games. However, due to the high-branching-factor nature of the game, minimax struggled to make several lookahead moves without sacrificing its effectiveness. Therefore, we implemented Alpha-Beta Pruning to reduce the number of explored branches and transposition table to cache the states that have already been explored, which reduces the time of re-exploring game states. Moreover, we updated the algorithm with Iterative Deepening so that it can explore deeper depths if the time permits, which improves the performance of the agent. The detailed explanation and results obtained from using these techniques are provided below.

Algorithm explanation

Pure Minimax

The algorithm simulates all moves made by both players up to 2 lookaheads. This is because as we progress the game and explore deeper depth of the trees, the number of branching factors increases significantly. This leads to a large growth in the number of nodes the algorithm must evaluate, which slows down computation time. As a result, the player will lose due to time constraint.

Alpha-Beta Pruning

To optimize Minimax, we implemented alpha-beta pruning to reduce nodes evaluated. The algorithm aids our agent in terms of performance. When compared it with Pure Minimax with depth 3, the time required for the agent to select the next move reduces by a half (from 156 seconds on average to 70 seconds). This allows us to increase the number of lookaheads for Minimax to 3, enabling the agent to make more informed decisions.

Transposition Table

One of the disadvantages of minimax is the fact that the board state is evaluated many times throughout the search of the tree. Therefore, to address this problem, we have implemented a transposition table, which caches the states of the board with its associated evaluation value in a dictionary.

Data structure used:

The transposition table is a dictionary where the key is the Zobrist Hash of the board's position. Each entry of the table consists of:

- **Depth**: The depth of which the Agent found the board state
- **Value**: It could be the exact evaluation score of the board, the upper bound (beta), or the lower bound (alpha) of which caused the cutoff when doing alpha-beta pruning
- **Best move**: The best move that caused the beta cutoff. This aids the move ordering of the search
- **Flag**: The entry's flag which helps identify if the value stored is an exact score, upper bound value (beta), or a lower bound value (alpha)

Creation

To create the table, we need a table random ID, which uniformly chosen from 0 to 2^{64}. Since the table is an 8 row by 8 columns table, and each cell can have 3 states: **red frog**, **blue frog**, and **Lilypad** (excluding the empty cell). This way ensures each cell state has its own unique ID.

Caching

To compute the hash of the table, we used Zobrist Hashing: For every cell on the board, we get its correspondence ID of that cell state. We initially set the hash value to be 0, then iteratively XOR it to the correspondence ID of each cell state iteratively. The process will halt until all the cells are scanned.

The board will check whether the board state has been cached or not. If the table yields a cache hit, it will return the value of the evaluation value of that state. If not, the evaluation will be calculated and then be added into the table.

Collision Handling

Since the table uses 64 bits to hash the board state, collisions can happen as it is not guaranteed to produce unique ID for every state of the board. In fact, there are around 4^{56} states (including invalid) for Frecker, while there's only 2^{64} unique IDs. Therefore, to reduce the probability of error, the agent will only use the data from the cache if its remaining depth equals the cache's remaining depth. While this does not eliminate hash collision totally, it helps reduce the errors.

Cache Invalidation

Without cache invalidation, the search cannot get the most up-to-date result from the table, which introduces bugs and might degrade the agent's performance. Therefore, the table is cleared after every move is made.

Uses:

In an Alpha-beta tree, there are 3 kind of nodes which are the result of the alpha beta pruning searching (*Node Types - Chessprogramming Wiki*, 2015):

- PV-node (Exact): The node whose score inside the searching window (alpha < score < beta), or the value return is the exact value of the evaluation
- Cut-node (Fail High): The node in which a beta-cutoff was performed, or the value return is the lower bound (Beta) of the search window
- All-node (Fail Low): The node in which a no move's score exceeded alpha, or the value return is the upper bound of the search window

With each cutoff, the value of the evaluation score is saved with its appropriate flag in the table entry.

Iterative-Deepening

In our playout of our Minimax with Alpha-Beta Pruning Agent, the agent can only effectively look 3 steps ahead without running out of time in the game.

One of the optimizations for Alpha-Beta Pruning is the use of iterative-deepening. The action search function will take in the board state and the time limit. The time limit is calculated by the average time an agent can spend searching for appropriate action (around 2.5 seconds) divided by the number of moves.

Then, if there is enough time, the function will iteratively increase the depth after every Minimax search. With the help of the transposition table, at a given depth, if the board has been cached, the value will be instantly returned, and the depth can be increased. The deepening has helped the agent look up as deep as 6 levels.

Evaluation function

Minimax Algorithm requires a good evaluation of the game board. We decided to use an 8x8 matrix,

each representing the positional weight for each frog. The matrix will start with the starting row, which all have positional weight as 0 because it is their start position. Then the weight is further increased to make the agent know that they should focus on getting to the other side of the board.

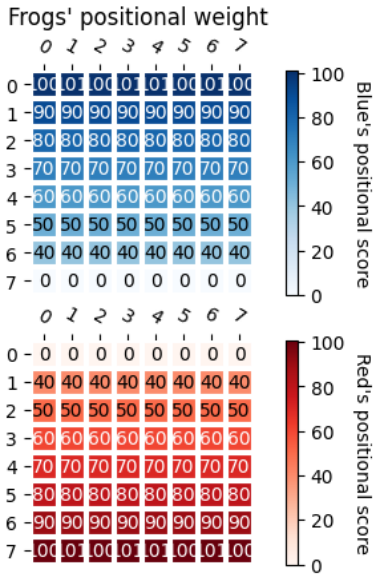

Figure 2: Frog's positional scoring with respective colours

With each team, the score is the sum of all the weight of their respective frog's colours.

$$\text{Score}_{\text{player colour}} = \sum_{\text{player's frogs position}} \text{BoardWeight}(\text{Frog's position})$$

Then the evaluation will be the difference of score the current player got to its opponent's

$$\text{Eval}(\text{board}) = \text{Score}_{\text{current player}} - \text{Score}_{\text{opponent}}$$

One special feature about this board is at the very end row of each respective board, the weights are assigned 100 and 101 alternatively. This is to prevent a deadlock where a row of frogs clumped into a cluster, preventing the frog who has not reached the destination from getting into the final row

For example, this boards shows the blue frogs are clustered as a group on the left-hand side while on the second row there's a frog on the left.

Figure 1: An example of a board where blue frogs are clustered in the corner

Unless the algorithm can evaluate until the 8th depth (as the second-row frog must do at least 4 action (Right, Right, Grow, Up Right) and evaluating its opponent's frogs), any action that could be taken results in the same evaluation score. As a result, the first-row frog stays stationary until the end of the game. With the above optimisation, the frogs who has arrived at the end of the board will be more likely to have a space in-between, therefore there's aways space for the other frogs to jump in.

Performance evaluation

How we evaluate the agents:

We have created a script which modified the run function of the referee to extract insights of the winning agent. We then compare 5 agents against each other and themselves. The information collected includes the resource usage: peak

memory usage and time elapsed, and the result of the match.

The 6 agents we have tested are:

Number	Name	Heuristics / Parameters
1	Random Agent	None (Select moves at random)
2	Greedy Agent	Greedy Agent will try to only maximize the number of frogs to the finish line
3	Minimax	Lookahead Depth 3
4	Iterative Deepening Minimax with Alpha Beta Pruning, Killer Moves and Custom Move Ordering	
5	Monte Carlo Tree Search Agent	Random Action Choosing Policy

Table 1: Implemented agents and their corresponding heuristic/parameters

As we have implemented several enhancements for our agent, we would like to see how much it has been improved compared to just Alpha-Beta Pruning enhancement to the Minimax Algorithm. Therefore, we simulated 20 playouts with our enhanced Alpha-Beta Algorithm against Pure Alpha-Beta pruning. The color assigned for each agent is changed alternatively.

Result

The Alpha Beta Pruning with Cache Agent consistently wins over 6 other agents.

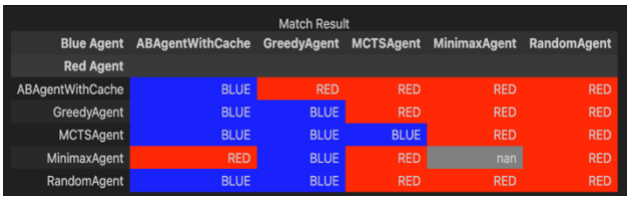

Figure 2: Result of the match with respective agents

Figure 2 showed that the AB Agent With Cache was able to win consistently against other agents. One interesting trend is that Blue were able to win 3 out of 5 playouts in match that uses the same-type agents. This suggests that Blue have a significantly better chance at winning. This could be because at the beginning of the game, there is no variation to the evaluation score of the red agent, therefore it plays an essentially a random move, or the move that it first found. Blue, however, has some variation to its evaluation function after Red makes it first move, therefore has a better time at evaluating when perform searching. Finally, Blue is more likely to be the last person to make the move, which could indicate the result of the game.

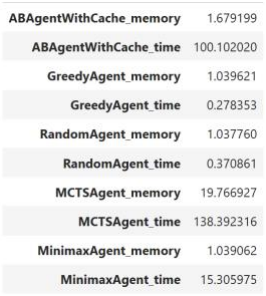

Figure 3: Average time spent by each agent each game

In the playouts with our enhanced Alpha-Beta pruning versus pure alpha beta implementation, our agent yields a significantly better result.

Agent name	Number of wins
AB With Cache and Move Ordering	16
Pure AB Agent	2

Figure 4: The result of the playout

	Without transposition table	*With Transposition table and Killer Moves*	*With transposition table and no Killer Move, with no sort*	*With transposition table, with no killer move, with sort*	*With Transposition table, killer move, and sort*
Median	11132	5490	5189	4029	3973.5
Total nodes	504493	221031	240881	180586	174719

The AB Agent with Cache has major performance improvement compared to traditional Alpha-Beta pruning agent. However, the time for the AB agent is significantly longer than AB Agent With Cache, which indicates a better performance while able to utilize resources more effectively. In fact, AB Agent With Cache was able to think on average 120 seconds per game, allowing agent to look ahead deeper.

Discussion

Transposition table

Theoretically, iterative-deepening helps the agent to see deeper into state tree while also utilising its time resources. However, this introduces performance overhead as the agent must re-evaluate the already-seen moves as the iteration starts from depth 0. Therefore, transposition table helps the agent to not re-evaluate already seen board state, allowing it to focus on pruning and evaluating deeper board states.

Move Ordering

Alpha-Beta pruning benefits significantly from appropriate move ordering as it allows the algorithm to prune more subtrees (*Alpha-Beta - Chessprogramming Wiki*, 2010). We attempted to sort the algorithm by the distance the frog moves, and the number of leaps jumps the frog can take as it allows the agent to gain more scores and therefore prune more nodes. Moreover, we also save the moves that caused beta pruning as it is considered as the best move (or a killer move) (*Killer Move*, 2021).

To validate our theory, we conducted 5 playouts of our agent as red, and the greedy agents as blue. We chose the greedy agents as it has the most similar moves between playouts.

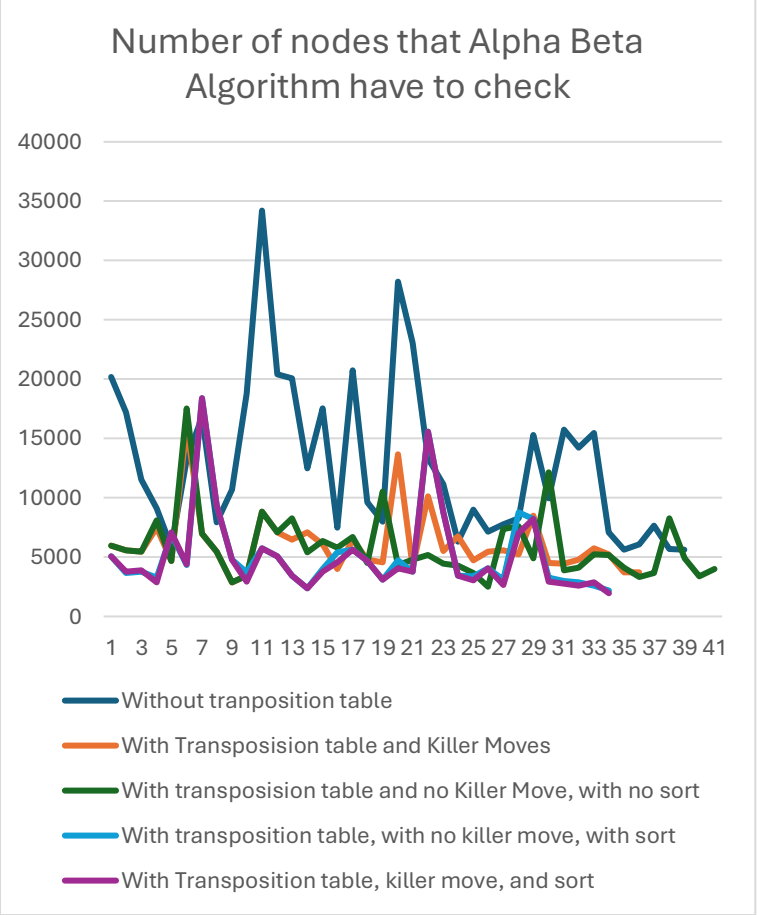

Figure 5: Number of nodes that alpha beta algorithm must check

Without any supports for move ordering and transposition table, the number of nodes assessed is significantly higher than the any its counterparts. Transposition table, killer move implementation, and move sorting reduces 75% of the total number of nodes.

Table 2: Median and the total number of nodes encountered

Grow Action Elimination

Frecker allows the agent to play grow move even when the spaces surrounding its frogs are already expanded. This led to our agent continuously played grow action and leads to the agent failing.

To address the problem, before assembling legal moves, the agent will check if the grow action makes any changes to the board. If there are changes, the grow action will be added, else the legal moves will only consist of move actions.

Supporting Works

Monte Carlo Tree Search Agent

One of the ways for us to evaluate our agent's evaluation function, we have created Monte Carlo Tree Search Agent (MCTS Agent). However, the MCTS Agent showed an underwhelming performance comparing to our Minimax Agent. This could be because the time we allowed the agent to run each move is too short (2.5 seconds), which makes the agent to only simulate around 20 to 30 times each moves. Moreover, the agent only chooses action randomly from the list of legal actions, therefore learning slowly. As a result, our MCTS Agent was scraped as it falls behind our Minimax Implementation.

The utilisation of the referee program to aid evaluation works

Referee module which was given by the teaching teams came with a lot of features, which allowed us to extract relevant information of playout. However, there were some modifications needed to be made to help our evaluation:

We created a new GameEnd data class, which we saved the agents names, remaining memory, time used, and the result of the playout. Then we modified the `_run()` and `main()` methods accordingly to be compatible with the new `GameEnd` class.

- Our evaluation (`evaluation.py`) script imports the referee class and use them as an API to help conduct playouts between different agents. Then the data collected will be dumped into a JSON file.
- We created a Jupiter notebook (`notebook.ipynb`) which we used Pandas to analyse the data and extract information.

References

Alpha-Beta - Chessprogramming wiki. (2010). Chessprogramming.org. https://www.chessprogramming.org/Alpha-Beta

Killer Move. (2021, July 20). Chess Programming Wiki. https://www.chessprogramming.org/Killer_Move

Node Types - Chessprogramming wiki. (2015). Chessprogramming.org. https://www.chessprogramming.org/Node_Types