Artificial Intelligence Project B Report

Approach:

Overall, we utilised Minimax Algorithm instead of Monte Carlo Tree Search (MCTS) to design our agent, while both algorithms are suitable for adversarial games, Minimax algorithm is more suitable for fully observable and deterministic games. Furthermore, MCTS may not be feasible and can be difficult to fine tune the agent. Hence we chose the first one and made many optimisations to enhance the depth and computational efficiency. Moreover, our evaluation function encourages our agent to advance more aggressively and hinders opponents movement.

To take advantage of the 180 seconds and 250mb constraints, we further improved our agent with various techniques, such as Alpha-Beta Pruning to reduce the number of nodes needing to be explored, Zobrist Hashing (GeeksforGeeks, 2016, Logic Crazy Chess, 2014) to store evaluated boards and refactoring codebase to combat performance bottleneck. Additionally, we employed a dynamic Min-Max exploration depth algorithm to allocate computational power to key decision-making situations. Lastly, to eliminate the cases where the frog cannot reach the final row, we modified the A Star pathfinding from project A to only consider the movement cost instead of returning the list of actions.

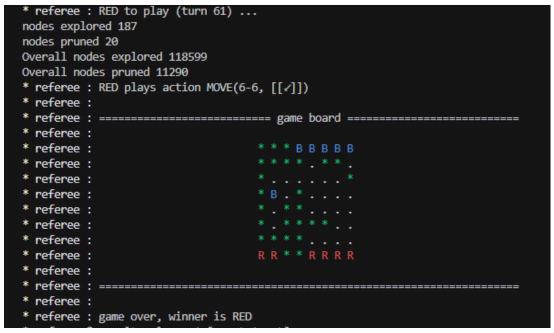
Minimax Algorithm:

The algorithm starts at the **get_best_move** Method, which evaluates all possible moves when the terminal_test condition is met or delves deep to evaluate whether it is the optimal decision in the future. We identified 3 possible **Actions**: Grow, Move, and Jump, where each type is considered individually. First, it evaluates Grow, which expands the movable space around the frogs. The agent sets the min_max_value of the current board state as the best value. Then, it evaluates Move, which simply enumerates in all possible directions and evaluates them accordingly to pick the min-max values and compare with previous one to keep the best action. At last checking jumps which utilised a depth first search to simulate all possible jumps, including those intermediate steps to minimise moves that greatly favours opponent and again keeps the best action possible.

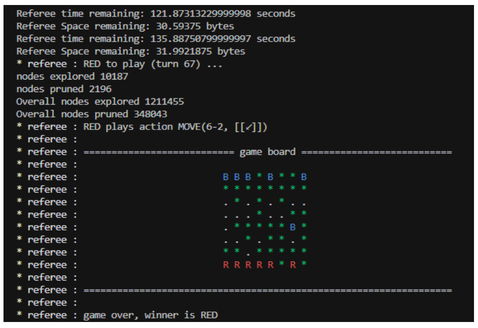
Alpha-beta Pruning:

In **evaluate_min_max** and **get_best_move** methods, we applied alpha-beta pruning after each evaluation to eliminate branches that will not affect the outcome of the decision, avoiding computational overhead. Where alpha represents the best evaluation that favours our agent and beta is the worst evaluation for the opponent. The agent maintains the values and passes them to the next depth in order to

maximise and minimise the players move respectively. When the beta <= alpha condition is met, it breaks out of the loop.
(Example below: Early stage, fixed depth without hashing)



```
* referee : RED to play (turn 61) ...
nodes explored 187
nodes pruned 20
Overall nodes explored 118599
Overall nodes pruned 11290
* referee : RED plays action MOVE(6-6, [[↙]])
* referee :
* referee : ========================== game board ==========================
* referee :
* referee :                         * * * B B B B B
* referee :                         * * * * . * * .
* referee :                         * . . . . . . *
* referee :                         * B . * . . . .
* referee :                         * . * * . . . .
* referee :                         * . * * * * . .
* referee :                         * * * * . . . .
* referee :                         R R * * R R R R
* referee :
* referee : ================================================================
* referee :
* referee : game over, winner is RED
```



```
Referee time remaining: 121.87313229999998 seconds
Referee Space remaining: 30.59375 bytes
Referee time remaining: 135.88750799999997 seconds
Referee Space remaining: 31.9921875 bytes
* referee : RED to play (turn 67) ...
nodes explored 10187
nodes pruned 2196
Overall nodes explored 1211455
Overall nodes pruned 348043
* referee : RED plays action MOVE(6-2, [[↙]])
* referee :
* referee : ========================== game board ==========================
* referee :
* referee :                         B B B * B * * B
* referee :                         * * * * * * * *
* referee :                         . * . * . * . .
* referee :                         . . . * . . * *
* referee :                         . * * * * * B *
* referee :                         . . * . * * . *
* referee :                         * * . * * * * *
* referee :                         R R R R R * R *
* referee :
* referee : ================================================================
* referee :
* referee : game over, winner is RED
```

	With pruning:	**WITHOUT pruning:**
time remaining (RED)	145.45s	105.46s
time remaining (BLUE)	150.89s	109.40s
Overall nodes explored	118599	296313
Overall nodes pruned	11290	

The data shows a portion of nodes being pruned by the algorithm, showing significant performance improvement. The agent is able to save 46.7% of time and pruned around 10% of total nodes after experimenting many trials. (RED taking 35s to win vs 75s to win). Later, with Zobrist Hashing, the number of nodes pruned jumped to 20%.

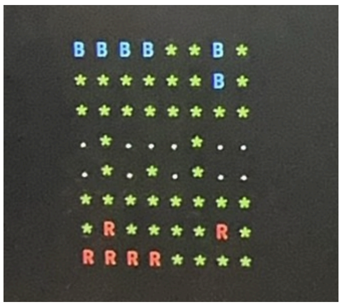
Evaluation Function:

Evaluation Value = MyFrogScore - EnemyFrogScore

Current implementation is dominated by the positions of frogs, that is the closer to the goal row, the greater the total score. Each row is weighted exponentially to encourage the frogs to advance as much as possible. In particular, scenarios in which the frog jumps over multiple frogs in one turn are considerably rewarding. On the other hand, when the opponents gain significant advantages in one action, the agent will try to minimise the opponent's advantage, such as sabotaging the opponent's multiple jumps, which, as a drawback, results in more turns overall.

Weights = [1, 2, 4, 8, 16, 32, 64, 200, 850] # 67 turns

After experimenting with the weights, we picked the combination that requires the least turns to complete the game, although this has some limitations, for instance the frogs ignore the jumps that just can advance multiple rows but with a less score.



```
B B B B * * B *
* * * * * * B *
* * * * * * * *
. * . . . * . .
. * . * . * . .
* * * * * * * *
* R * * * * R *
R R R R * * * *
```

Figure1. Red frogs stuck on finding the goal

A* Pathfinding

In our observation, there are scenarios in which the frog is unable to reach the final row due to the limitation of distance based evaluation function. As the figure shows, red frog (6, 7) is not able to move to (7, 4) because the intermediate steps have the same evaluation values, as a consequence, they are pruned instead of continuing to explore. Thus, we introduced A Star Pathfinding to evaluate the distance of the frogs to its nearest destinations. Lastly, because the path is always blocked at the beginning of the game, the agents will only start to evaluate the distance when the distance to the goal_row is between 1 and 4 units.

Transposition table and Zobrist Hashing:

Although the evaluation of the boards will always be the same, many boards are re-evaluated multiple times. So we applied zobrist hashing to store evaluated boards in a transposition table, and retrieve them when evaluating the same board. To adapt the changes, we modified the board from a dictionary based board to a 2D list based – list[list[str]], and made changes to `apply_action` Method.

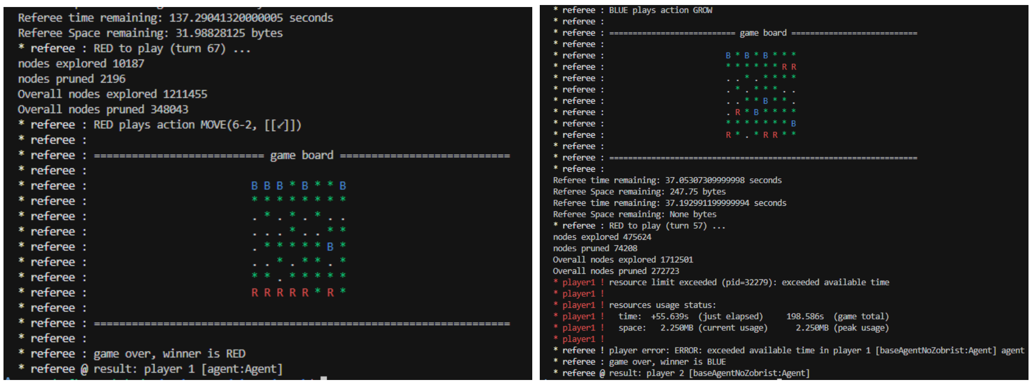
XOR Computation

The dictionary implementation pops and inserts changes to the board, whereas the new method uses XOR binary computation, which is extremely fast for a computer to compute. This significantly improved the overall performance.

Zobrist Hashing:

First, we assign a unique random number to every possible state for each cell (192 unique integers), which runs only once every game. Then it creates boards using the 2D list of str. Initially, the starting board is hashed by XOR-ing every lilypad, and frogs by enumerating the entire board, later, only the ones getting updated will be XOR. This way, we created unique hash keys to access all evaluated boards.

Base agent: minimax with alpha-beta pruning, dynamic depth with default depth 4.
Base agent with Zobrist Hashing:



```
Referee time remaining: 137.29041320000005 seconds
Referee Space remaining: 31.98828125 bytes
* referee : RED to play (turn 67) ...
nodes explored 10187
nodes pruned 2196
Overall nodes explored 1211455
Overall nodes pruned 348043
* referee : RED plays action MOVE(6-2, [[↙]])
* referee :
* referee : ========================== game board ==========================
* referee :
* referee :                          B B B * B * * B
* referee :                          * * * * * * * *
* referee :                          . * . * . * . .
* referee :                          . . . * . . * *
* referee :                          . * * * * * B *
* referee :                          . . * . * * . *
* referee :                          * * . * * * * *
* referee :                          R R R R R * R *
* referee :
* referee : ==================================================================
* referee :
* referee : game over, winner is RED
* referee @ result: player 1 [agent:Agent]
```

```
* referee : BLUE plays action GROW
* referee :
* referee : ========================== game board ==========================
* referee :
* referee :                          B * B * B * * *
* referee :                          * * * * * * R R
* referee :                          . . * . * * * *
* referee :                          . * . * * * . .
* referee :                          . . * * B * * .
* referee :                          . R * B * * * *
* referee :                          * * * * * * * B
* referee :                          R * . * R R * *
* referee :
* referee : ==================================================================
* referee :
Referee time remaining: 37.05307309999998 seconds
Referee Space remaining: 247.75 bytes
Referee time remaining: 37.192991199999994 seconds
Referee Space remaining: None bytes
* referee : RED to play (turn 57) ...
nodes explored 475624
nodes pruned 74208
Overall nodes explored 1712501
Overall nodes pruned 272723
* player1 ! resource limit exceeded (pid=32279): exceeded available time
* player1 !
* player1 ! resources usage status:
* player1 !   time:  +55.639s  (just elapsed)     198.586s  (game total)
* player1 !   space:   2.250MB (current usage)      2.250MB (peak usage)
* player1 !
* referee ! player error: ERROR: exceeded available time in player 1 [baseAgentNoZobrist:Agent] agent
* referee : game over, winner is BLUE
* referee @ result: player 2 [baseAgentNoZobrist:Agent]
```

Referee time remaining: 124.22 seconds
Referee Space remaining: 31.99 megabytes

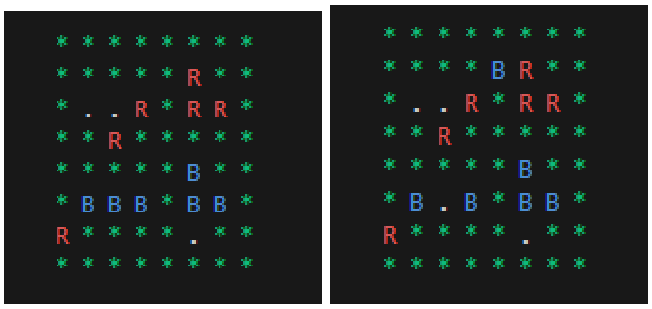
Base agent without Zobrist Hashing: (Right figure)
Did not manage to finish, failed the timing test when the game was still in a halfway stage. Did not utilise the use of space provided at all. From the stats, it clearly acknowledged the choice of using Zobrist Hashing especially when combined with dynamic depth approach and the need to look furthermore than base depth in many stages during the game.

Dynamic Minimax Depth:

After applying Zobrist hashing, the agent can complete the game faster, which leaves room for improvements by delving deeper boards, however, setting to depth 6 can result in risk of being unable to complete the game on time and 5 can be too shallow. Therefore, we adopted a dynamic depth algorithm that evaluates the positions of the frogs to determine the stage of the game and adjust the depth of Minimax search to make the most of the time and space constraint. **(at_critical_pos, in_late_game, get_dynamic_depth)** It avoids unnecessary deep searches in early game stages and adjusts to the opponent's strategies by focusing computational resources on critical moments.

Depth First Search (DFS)

In order to consider all possible moves, we used DFS to enumerate all possible jumps, including those intermediate jumps, this allows us to consider cases where jumps that could potentially maximise our score and minimise opponents scores.



```
* * * * * * * *
* * * * * R * *
* . . R * R R *
* * R * * * * *
* * * * * B * *
* B B B * B B *
R * * * * . * *
* * * * * * * *
```

```
* * * * * * * *
* * * * B R * *
* . . R * R R *
* * R * * * * *
* * * * * B * *
* B . B * B B *
R * * * * . * *
* * * * * * * *
```

Figure 3. referee : BLUE plays action MOVE(5-2, [[→], [↗], [↑], [←]])

Data Structure:

- MyBoard: board: list[list[str]], red_frogs_positions: set(), blue_frogs_positions: set(), hash_key: int
- MyGame Board Class is created to encapsulate board environments, such as hash_key, frog position and lilypad position, represented using ‘l’, ‘r’, ‘b’ respectively. To increase access speed, the frog's positions are stored in a set. and copied whenever a new board is created.
- Priority Queue: is used in A* pathfinding
- Dictionary: for minimum path costs in A*
- Set: to get frogs’ positions. O(1)
- Stack: for dfs, in get_all_possible_jumps
- Cache: dict[int, BoardState]
- Queue (least recently used board) : to remove cached boards

Performance Evaluation:

Time Complexity

The performance is primarily dominated by the number of nodes needing to be evaluated.
Minimax Search:
Worst time complexity O(b^m), where b is the branching factor and m is the depth of the search.

- b ≈ possible_moves * number_of_frogs + possible jumps * number_of_frogs
- m = 5-7, dynamic depending on the state of the board.

Each player has six frogs on board, each frog can move in five directions or play a GrowAction. So the worst case branching factor is 31 not considering jumps.
After pruning: O(b^(m/2)) on average, and in the worst case it’s still O(b^m) when no nodes get pruned.
Zobrist Hashing: O(1) because XOR computation cannot exceed the board width.
Transposition Table Retrieval: O(1) because the hashing key is astronomically unique for a random number of 64 bits.
A * Pathing: O(b^m) for the worst case, where b is the number of possible moves and m is the depth of the search, for our optimised solution, m cannot exceed BOARD_N.

Space Complexity

Minimax : O(bm) = O(possible_moves * frogs * depth * number of turns)
Size of Transposition Table = size_of_board * size_of_board * cell_states

- Which is really small as it will be computed once.

Size of Cached Board State = max_cache_size * board_state_size

- Which is essentially a dictionary that has hash_key and a float (Explained in supporting work) as value.

my_board: MyBoard = board size + set for red/blue frogs + hashing size

Other aspects:

Zobrist Hashing Optimisation: Only the first board is hashed entirely and later boards only update through xoring, which improves the computational efficiency.
Setting the depth to a lower one may cause its loss against the random agents despite that it is able to 'finish' the game fast enough. And setting the depth to higher usually causes long responding time in every stage of the game and ultimately leads to the breaching of time limit. Hence after many trials, the base depth of 4 and further on that the dynamic depth brings together the balance during searching for winning.

Supporting work

Evaluation Function : evaluate_free_tiles

We tried to take the number of free tiles into consideration, however, this had no effect on improving the number of turns.

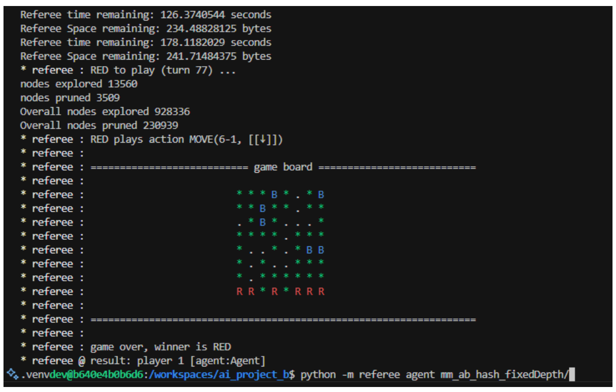
Evaluation Function : Dynamic weighting

Current implementation uses hard coded numbers, which can be helpful in the early stage, however, the frogs that are left behind are reluctant to advance in the later game as the weights are insignificant. This suggests a dynamic weighting algorithm and be implemented to enhance the competency of the agent.

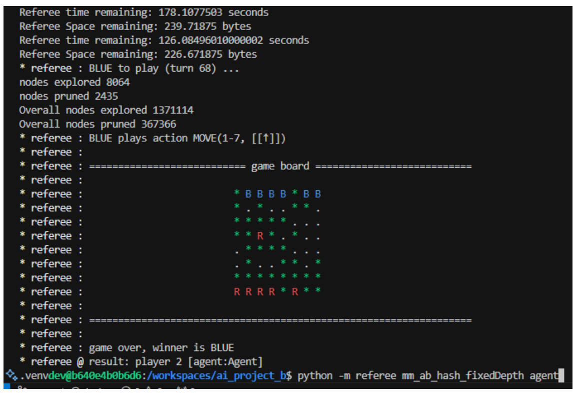
Potential Further improve the performance with Zobrist Hashing

We tried to make use of zobrist hashing by storing the actions and alpha beta values, so the game can continue from the optimal path instead of simply retrieving stored evaluation values. This was proposed in the BoardState Class, however, the attempt was not successful.

Agent (mm_ab_hash_dynamicDepth) vs mm_ab_hash_fixedDepth



```
Referee time remaining: 126.3740544 seconds
Referee Space remaining: 234.48828125 bytes
Referee time remaining: 178.1182029 seconds
Referee Space remaining: 241.71484375 bytes
* referee : RED to play (turn 77) ...
nodes explored 13560
nodes pruned 3509
Overall nodes explored 928336
Overall nodes pruned 230939
* referee : RED plays action MOVE(6-1, [[↓]])
* referee :
* referee : =========================== game board ===========================
* referee :
* referee :                            * * * B * . * B
* referee :                            * * B * * . * *
* referee :                            . * B * . . . *
* referee :                            * * * * . * * *
* referee :                            * . . * . * B B
* referee :                            * . * . . * * *
* referee :                            * . * * * * * *
* referee :                            R R * R * R R R
* referee :
* referee : ==================================================================
* referee :
* referee : game over, winner is RED
* referee @ result: player 1 [agent:Agent]
.venvdev@b640e4b0b6d6:/workspaces/ai_project_b$ python -m referee agent mm_ab_hash_fixedDepth/
```



```
Referee time remaining: 178.1077503 seconds
Referee Space remaining: 239.71875 bytes
Referee time remaining: 126.08496010000002 seconds
Referee Space remaining: 226.671875 bytes
* referee : BLUE to play (turn 68) ...
nodes explored 8064
nodes pruned 2435
Overall nodes explored 1371114
Overall nodes pruned 367366
* referee : BLUE plays action MOVE(1-7, [[↑]])
* referee :
* referee : =========================== game board ===========================
* referee :
* referee :                            * B B B B * B B
* referee :                            * . * . . * * .
* referee :                            * * * * * . . .
* referee :                            * * R * . * . .
* referee :                            . * * * * . . .
* referee :                            . * . . * * . *
* referee :                            * * * * * * * *
* referee :                            R R R R * R * *
* referee :
* referee : ==================================================================
* referee :
* referee : game over, winner is BLUE
* referee @ result: player 2 [agent:Agent]
.venvdev@b640e4b0b6d6:/workspaces/ai_project_b$ python -m referee mm_ab_hash_fixedDepth agent
```

Used the final agent with base depth 4 to compete with minimax + pruning + hashing + fixed depth 4. At turn 77, the final agent is able to win the game. When we switch the order the agent is still able to win the game.

References:

Technique Reference

- Logic Crazy Chess. (2014, June 24). Transposition Tables & Zobrist Keys - Advanced Java Chess Engine Tutorial 30. YouTube. https://www.youtube.com/watch?v=QYNRvMolN20
- GeeksforGeeks. (2016, October 10). Minimax Algorithm in Game Theory | Set 5 (Zobrist Hashing). GeeksforGeeks. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/

ChatGPT prompt used:

- Basic questions to understand the fundamentals and feasibility of each algorithm (Minimax, MCTS).
- Possible optimisation techniques we could employ to enhance the competency of adversarial agents.
- What are the important aspects to consider to win the game?
- Some trivial questions regarding python.