Assignment 4

Title: Nim Game Analysis with Minimax and Alpha-Beta Pruning Algorithms

Name: Brennen Cramp

Date: 11/2/2025

Introduction

Nim is a game where there are two players aiming to take the final object(s) out of the last, populated heap. A player, in turns, can take one or more objects out of only one of the heaps until all the heaps are depleted. In addition, a player must always take at least one object from one of the heaps each turn. There can be a varied number of heaps with a varied number of objects in each of the heaps.

As an example, a two-player Nim game tree with two heaps both containing two objects has been provided below in *Figure 1*. This game tree shows all the possible player moves and their outcomes at each level of the tree where the terminal nodes ([0]: 0, [1]: 0) are highlighted red.

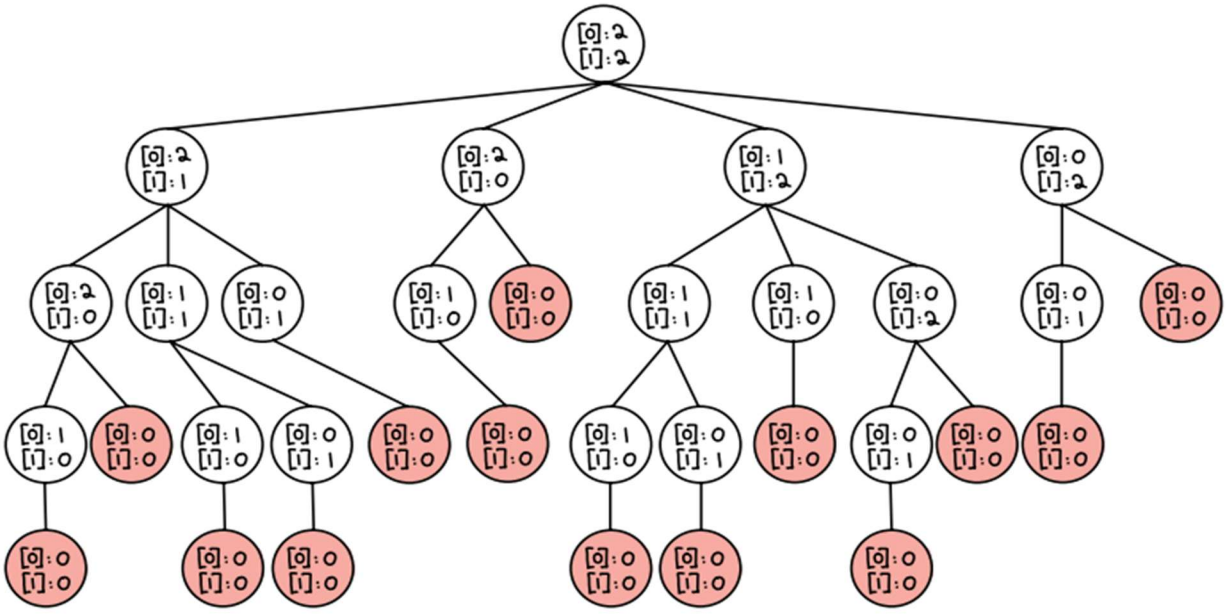


Figure 1: Nim game tree w/ 2 heaps w/ 2 objects in each heap

Algorithm Analysis

When applying the minimax algorithm to the game tree (seen in *Figure 2*), the optimal move for each player is shown along with the depth after each move has been made. Minimax is effective in this case provided in *Figure 2* since the player who is trying to maximize their moves (player one) won by first taking the maximum of the minimum values provided ([0]: 1, [1]: 2) with a value of 7. The next move was made by player two who took the minimum of the new moves provided ([0]: 1, [1]: 0) which was still 7. However, this led to player one winning by achieving the terminal state of ([0]: 0, [1]: 0) first; thus player one who was trying to maximize their possibility of winning had succeeded. If the heaps involved in the game expanded past the current two, there would be exponentially more moves which would provide more complexity in the evaluation since the minimax algorithm expands into every node. The same would apply if there were more objects in the piles which would allow for more variety in the move set. Minimax proves very effective and is guaranteed to find the optimal move in smaller state sets due to the two-player game nature of the algorithm by always expanding into every path through its Depth-First Search (DFS) design. This comes at a cost once more piles or objects in the piles increases since the DFS algorithm that minimax uses has a time complexity of $O(b^d)$, thus exponentially adding more move possibilities makes the algorithm not efficient in large state sets.

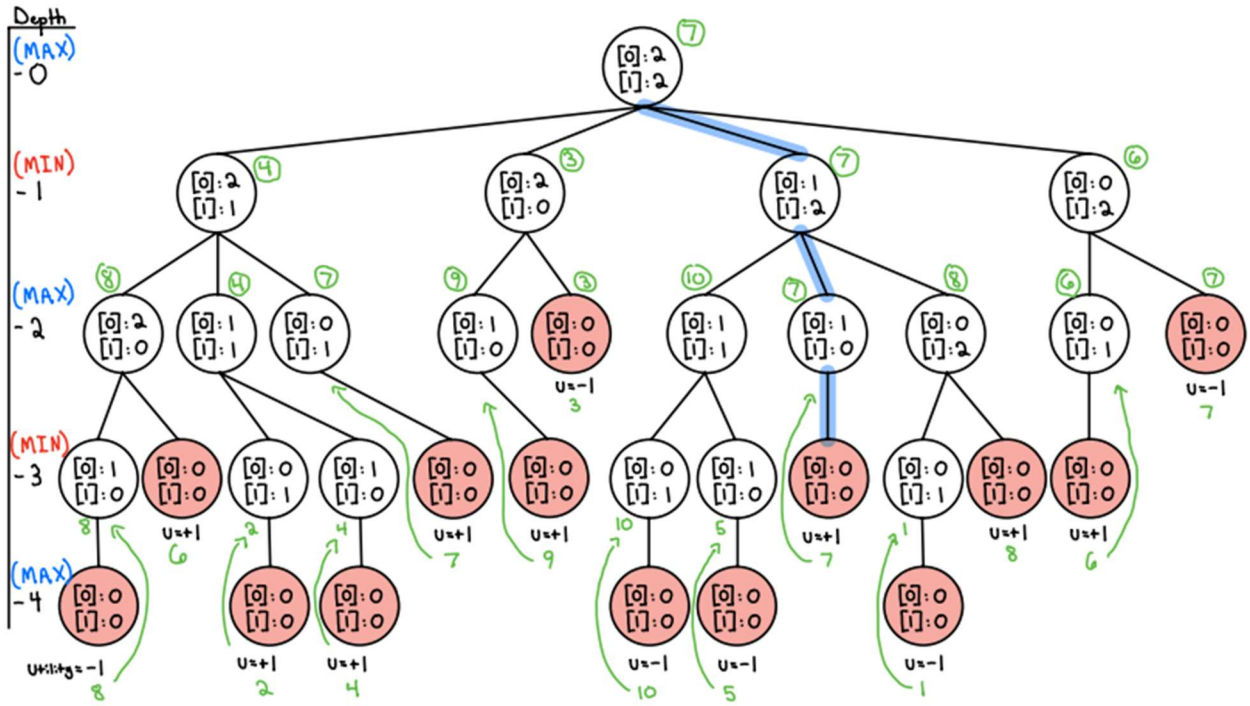


Figure 2: Nim game tree w/ Minimax Applied

In contrast to minimax, the alpha-beta pruning algorithm will prune/remove expanding nodes deemed not possible. This evaluation varies on the type of node at each depth, meaning if the node is a "max" node, there will be an evaluation of $\alpha \geq \beta^{ancestor}$ and if the node is a "min" node, there will be an evaluation $\alpha^{ancestor} \geq \beta$. If either of these checks are true at their specific nodes, then the path will be pruned and will not be expanded upon. These events can be witnessed below, in *Figure 3*, where the paths that have been pruned have been scribbled out in red to show that the tree will not expand into those nodes. This type of pruning allows for more efficiency, especially in larger state sets (i.e. chess and games with massive state sets) where there could be one node that is deemed not a viable option to expand upon, which will save time in finding the most optimal path. However, as seen below, this matters on the utility function and the heuristic assigned to each node while alpha-beta pruning guarantees the most optimal path,

same as the minimax algorithm, this evaluation led to the minimizing player (player two) to win the game.

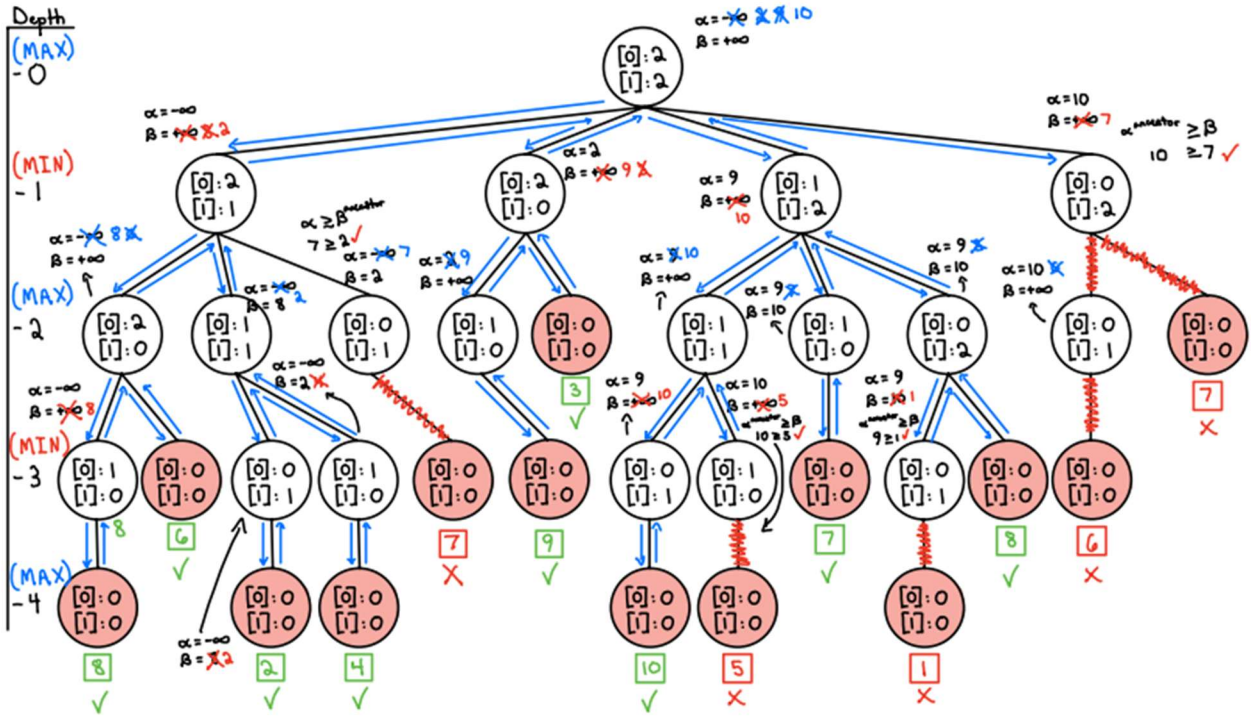


Figure 3: Nim game tree w/ Alpha-Beta Pruning Applied

Optimization Techniques

Additional optimizations for the minimax algorithms would be desired especially for larger game trees. A technique that could be utilized can be the Iterative-Deepening Search (IDS) where, as the algorithm states, is a combination of Breadth-First Search (BFS) and DFS where it can be optimal if all the step costs are 1 and has a time complexity of $O(b^d)$. IDS will check for the goal node by iterating through more depths if the goal has still not been found. While it may not be optimal for minimax due to the differing step costs, it could find the goal quickly as seen in *Figure 4*. While performing IDS, minimax will be executed at each depth where it will perform a depth-limited search, evaluating the game tree and once the desired goal has been found, it will return the optimal move found for the player at that depth; effectively cutting down searching the

whole tree. The benefits of IDS might seem minimal for a two-player game, however, in larger games, the starting depth search could provide move ordering that can be invaluable especially for the information provided for the following depth searches and if alpha-beta pruning was utilized, more branches can be reduced/pruned to dramatically reduce computation time.

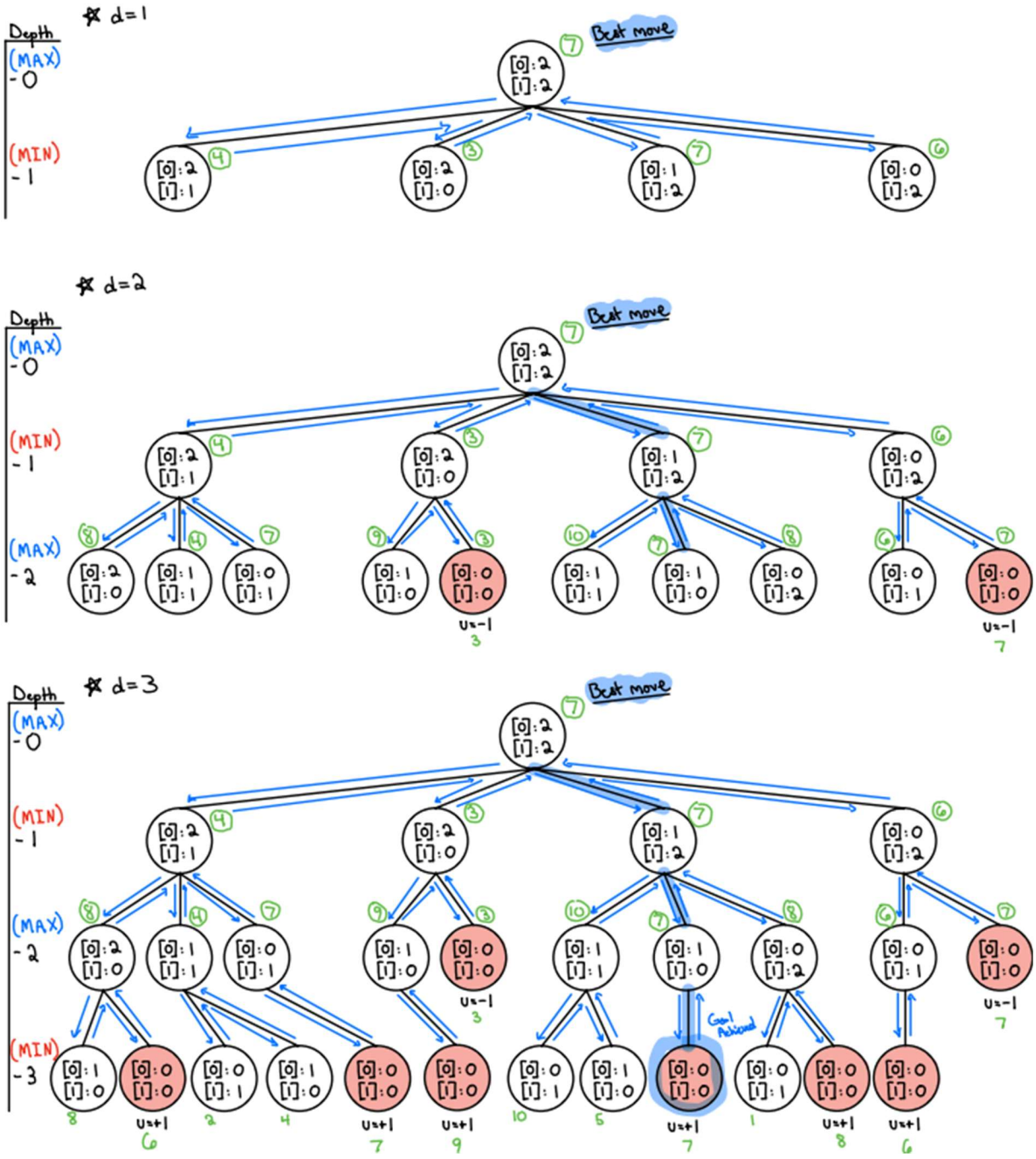


Figure 4: Nim game tree w/ Minimax and IDS Applied

Heuristic Evaluation

A heuristic function that could be used is a relatively simple one where an evaluation of the state's position in the game. For instance, in Nim, if one of the piles still has remaining objects while all the other piles are empty, this is a losing position for the previous player and a winning position for the current player. This can be determined using a Boolean for whether the state contains all empty piles but one non-empty pile, where each pile has a Boolean saying if it still contains objects meaning each state would have either a zero or one to denote if the state is a winning state or not a winning state. The Boolean will then determine if the state has a positive 1 or a negative 1 based on the favorability of that position for the player. For example, Nim will be played on a smaller subset to emphasize the heuristic working as seen in *Figure 5*, the maximizing player will pick the positive 1 route and inevitably won.

When comparing the standard minimax algorithm against the heuristic-based minimax algorithm, completeness, performance, and what the algorithm guarantees should be evaluated. The completeness of the standard minimax algorithm in a finite space will always find the optimal move, given the entire tree can be explored while the heuristic-based minimax algorithm might not always find the optimal move since the heuristic itself needs to be complex and effective along with applied throughout the tree instead of strictly at the terminals to dictate the best move. The performance for a standard minimax algorithm in large state sets or infinitely expanding trees is especially not ideal in this case but extremely effective in smaller state sets, guaranteeing that the optimal move will be found. The performance for the provided heuristic-based minimax algorithm is very efficient with reduced computational time since it will not be exploring the entire tree, in contrast to the standard minimax algorithm. Due to the clear winning and losing positions that are in Nim, the simple heuristic-based minimax algorithm can guarantee

that the optimal move will always be found due to the simplicity of Nim. However, in more complex games where full exploration is impossible, a heuristic will need to be more complex and well-designed to greatly improve performance. IDS could be implemented along with the heuristic-based minimax algorithm to keep the state set in small chunks to find optimal moves.

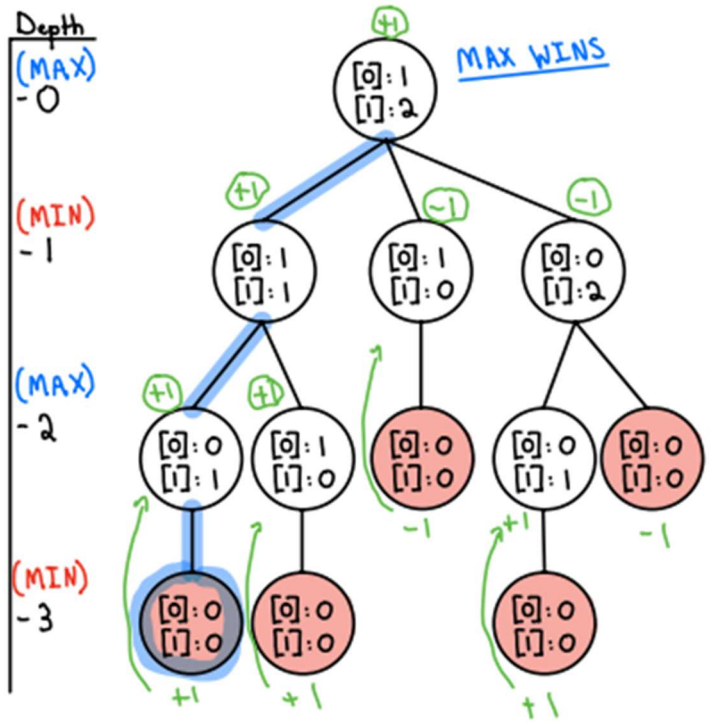


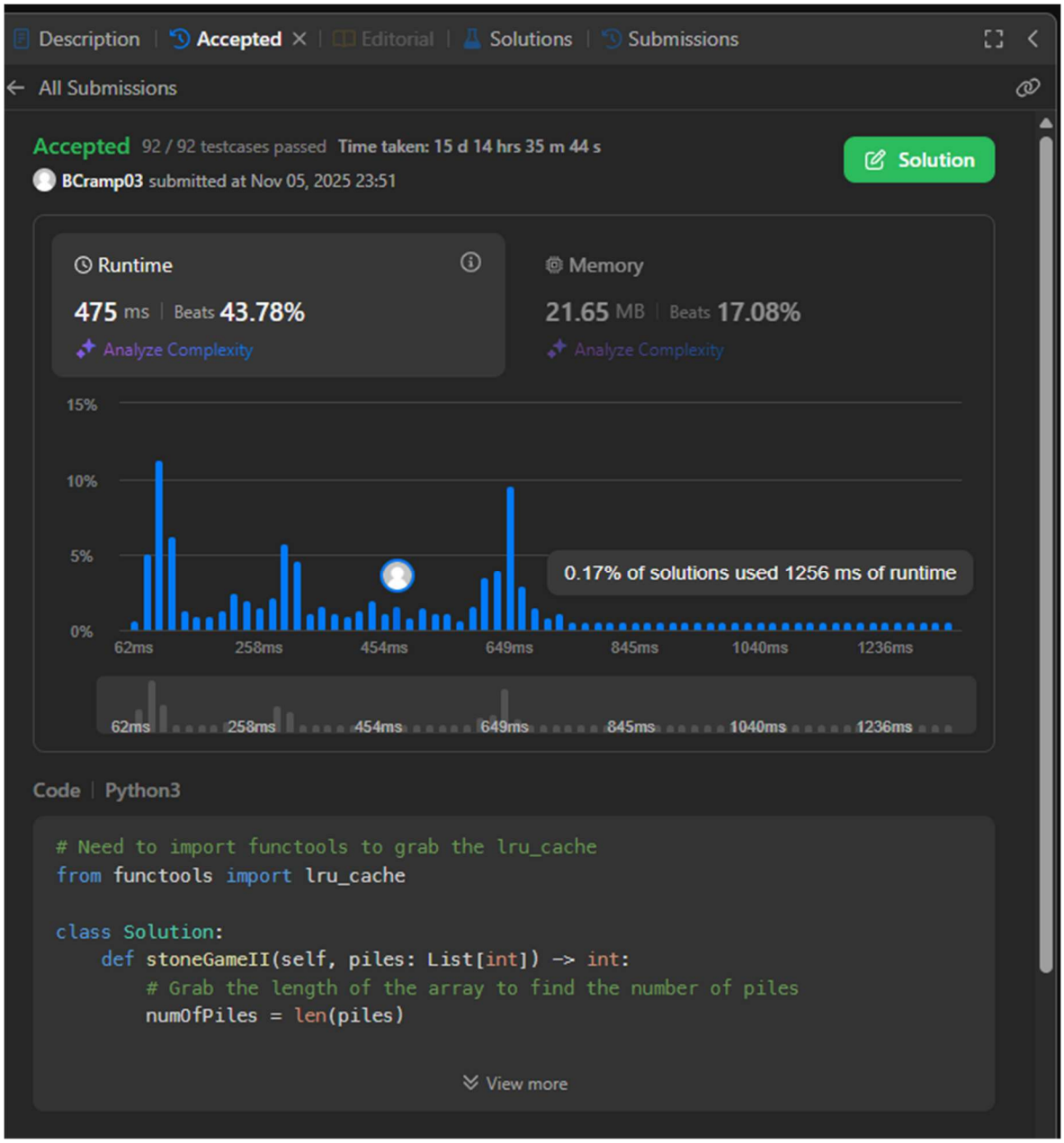
Figure 5: Smaller Nim game tree w/ Simple Heuristic Applied

Conclusion

In essence, the use of alpha-beta pruning proves to be more effective than the minimax algorithm and if a well-designed heuristic was implemented for the Nim game tree alongside alpha-beta pruning, it would greatly improve computational time. Alpha-beta pruning is an optimization that significantly decreases the computational time for the search by pruning routes of the game tree that will not be travelled, while still finding the optimal move as opposed to minimax's exploration of the entire game tree, leading to longer computational times and less performance.

Extra Credit / Grad Student Problem

Leet code accepted submission for the “Stone Game II (1140)” problem:



I used Minimax in my solution by first creating a list of sums that counted each pile from the current position where I could keep track of the most amount one of the players could add to their total amount from that current position. The next step was to call my minimax function that would determine if Alice (the maximizing player = true) played optimally, given Bob was to minimize the maximum number of points she could get, through calculating the maximum points based on Bob’s turn of minimizing her next turn during his turn. I used an if-else statement to determine whose move it was and would calculate Alice’s max if she was the current player and

Bob picking the smallest amount she could get from the possible next moves where the recursive structure broke down the problem to the base case and unraveled so each player would get the other's set of moves, either maximizing or minimizing their new set of options.