Nick Zywalewski

Artificial Intelligence: Homework 4

Purpose: Exploring the minimax algorithm for a game of Nim, alpha-beta pruning, and potential optimization techniques.

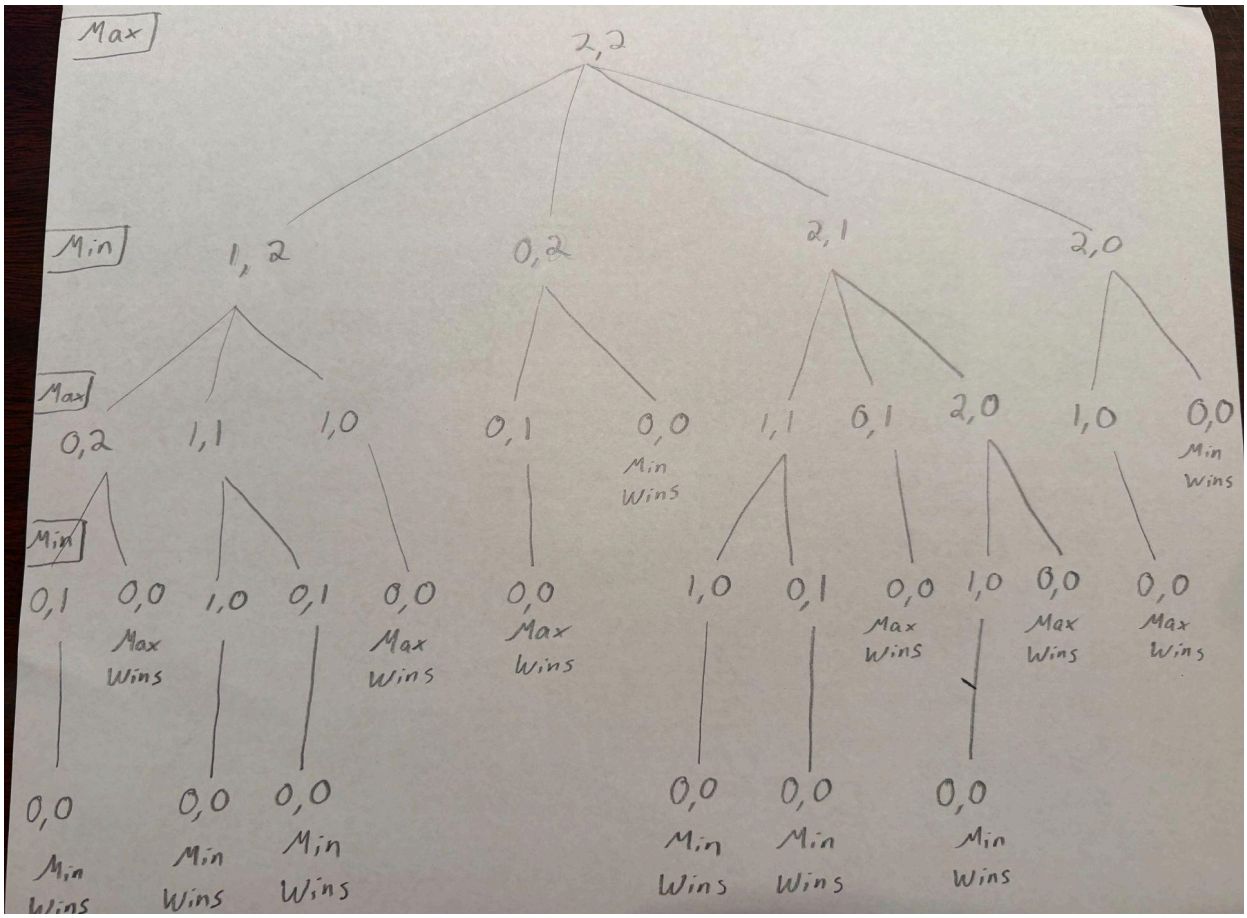
Problem 1:

Question 1: Game Description: Explain the rules of the game of Nim, including how many heaps there are, how many objects are in each heap at the start of the game, and how players take turns removing objects from the heaps.

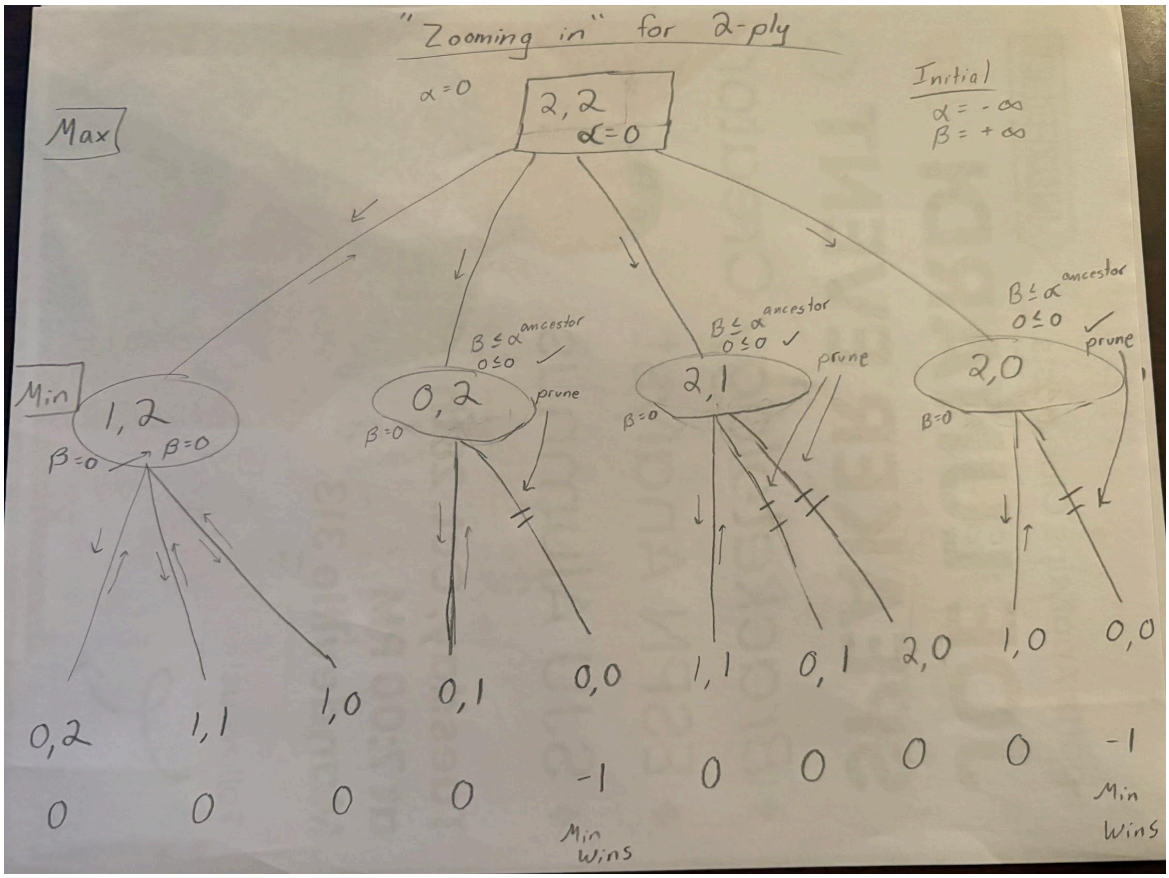
Nim is a two-player turn-based game in which there are objects in multiple different piles, called heaps. On each turn, a player must choose a heap and select at least one object from the heap (can remove as many objects from the heap as they choose, even all of them). The game continues with the turns alternating between each of the two players. The game is over when all objects have been removed from all of the heaps, and the player that removes the final object wins the game of Nim.

Below, I analyze a game beginning with two heaps, and each heap contains 2 objects. This is represented by the state (2,2). At each stage, the number of objects remaining is denoted as (x,y), where x is the number of objects in the first heap and y is the number of objects in the second heap. Player MAX takes the first move, and MAX is able to remove 1 object from the first heap, 2 objects from the first heap, 1 object from the second heap, or 2 objects from the second heap. The game proceeds until the state reaches (0,0), denoting no remaining objects in either heap. At this point, the player (MAX or MIN) that removed the final object wins the game.

Question 2: Draw the game tree for a simple case of Nim, such as a game with two heaps containing 3 and 4 objects, respectively. Show all possible moves and their outcomes at each level of the tree.



Questions 3-4: **Minimax Algorithm:** Apply the minimax algorithm to the game tree to determine the optimal move for each player at each level. Show the values assigned to each node in the tree by the minimax algorithm in a 2-ply game. **Alpha-Beta Pruning**: Apply the alpha-beta pruning algorithm to the game tree to reduce the number of nodes that need to be evaluated. Show the alpha and beta values at each level of the tree and indicate which nodes were pruned.



In the 2-ply minimax evaluation of (2,2), MAX sets alpha=0 after evaluating (1,2) and this causes pruning in the MIN nodes (0,2), (2,1) and (2,0) because once beta reaches 0 in those nodes, the (current beta <= ancestor's alpha) equation triggers pruning because 0 <= 0. This eliminates the branches that cannot change the final decision: the second branch out of (0,2), the second and third branches out of (2,1), and the second branch out of (2,0).

Question 5: Discuss the effectiveness of the minimax algorithm and alpha-beta pruning in solving the game of Nim. Consider factors such as the size of the game tree, number of heaps, number of objects in each heap, the number of nodes pruned by alpha-beta pruning, and the complexity of the game.

The main limitation of using 2-ply minimax search in Nim is that it can only look two moves ahead, so it cannot always see the long-term consequences of a position/move. This means that a move that appears reasonable at a shallow depth may actually lead to a forced

loss later on once there are more moves made. This is because the algorithm is cutting off the search early and relying on estimated values rather than seeing the full end of the game. If we were able to search deeper, the results would be more accurate because we would see more of the actual outcomes instead of approximations. However, searching deeper becomes difficult since the game tree grows very quickly with more heaps and larger size heaps. Minimax still gives us a logical way to decide which moves place the current player in a stronger position, and alpha-beta pruning makes this process more efficient due to the cutting off of branches that do not affect the final decisions/values. This is because once alpha and beta values show that a move will be unable to change the value or improve the outcome, there is no more reason to explore the branch. In the example above, we were able to reduce the number of branches we had to evaluate, which is able to show that alpha-beta pruning can save time while keeping the same result.

Problem 2:

Question 1: Explore additional optimization techniques for the minimax algorithm, such as the use of iterative deepening, to improve its performance in analyzing larger game trees. Use a 2-ply game to analyze and explain your strategy.

One way to improve the performance of minimax in larger game trees is to use iterative deepening, which means that the algorithm gradually increases how far it searches instead of jumping straight to a deep search all at once or stopping a defined shallow depth. This is helpful because searching the entire game tree at a large depth can become overwhelming very quick, especially as the number of possible moves grows. By starting with the shallow search, the algorithm can get an early sense of which moves appear promising, and this early information can then guide the order of exploration when the search goes farther. This means that when the depth is increased, the algorithm is more likely to explore strong moves first. This also means that we are able to explore the strongest branches earlier, and weaker branches can be identified and avoided sooner. Essentially, this would lead to fewer nodes being evaluated and makes alpha-beta pruning more effective, which would improve speed and accuracy.

In the context of the 2-ply NIm example from the previous problem, the first shallow search would provide a basic understanding of which moves seem reasonable from the starting position. If we then applied iterative deepening, we could extend the search to 3-ply or 4-ply and focus our attention on the branches that looked most useful from the earlier search process. This would help us reveal deeper consequences of certain moves while still managing the complexity of the game tree. However, as we continue to search deeper, the number of states still grows quickly, especially with larger heap sizes, so there is always a balance between going deep enough to improve the accuracy of the algorithm and staying shallow enough to keep the algorithm from doing a ton of work (computationally better).

Another factor that can influence the efficiency of the algorithm is the order in which the children nodes are explored. This is because it would have an effect on how early/late we are able to prune certain branches. If the algorithm explores a strong move early, it can set alpha or beta to a useful value early on. This would allow more of the branches to be pruned. However, if the children are explored in a less favorable way, then pruning will not be able to occur until later

on in the search. This would mean that more nodes would have to be evaluated, and therefore, the algorithm would not be as efficient as it could be.

In the 2-ply example from the previous problem, evaluating a move like (1,2) first helped set alpha to be 0 early on. This allowed branches under (0,2), (2,1), and (2,0) to be pruned. If we had explored a weaker move (like (0,2)) first, alpha would have stayed lower (at -1) until a stronger branch was reached, delaying when pruning could begin. This would have meant that pruning would have been delayed. This shows that the ordering of the children nodes could potentially act as an optimization technique in and of itself due to its effect on how much pruning can occur.

Question 2: Develop a heuristic evaluation function to estimate the value of game states in Nim, and compare the expected performance of heuristic-based minimax with the standard minimax algorithm. Discuss you approach using a 2-ply game.

For the heuristic evaluation in a 2-ply game, a scoring rule that favors equal heaps could be used. For example, if the heaps are an equal size, then the heuristic would produce a positive value of some sort. If the heaps are unequal in size, then the heuristic would produce a negative value for that particular node. This is because equal heap sizes appear to give the player more control, since these positions often allow the player to respond to the opponent's moves in a mirrored way. This would keep the game balanced in their favor. When the heaps are unequal, however, the situation becomes more vulnerable because the opponent can steer the game toward a forced winning path.

Using this heuristic would be beneficial, as long as the heuristic is aware of whose turn it is. Equal heaps are only beneficial when the current player is the one who creates them. This means that if Max makes a move that results in equal heaps, the heuristic should score the position positively, but if Min makes the move to create equal heaps, the heuristic should score the position negatively, since Min would then gain the ability to mirror Max's moves. For example, if Max makes a move that results in (1,2), then Min can easily make a move that results in (1,1). Max would then be forced to take the rest of one pile, and Min would win on the next move. However, if Max makes a move that can result in an even number in each heap, then Max would be in a position of power.

Unfortunately for Max in the (2,2) example above, there is no move that can result in a pair of even heaps for Max. However, in a different setup to the game, this could be a beneficial heuristic to use when making informed decisions, and it would keep computation to a manageable level.

With this heuristic, it is also important to note that the (0,0) states are not automatically positive just because the heap sizes are equal. Instead, their values depend on who made the last move. These terminal win/loss outcomes (the (0,0) states) would need to be represented as extremely large values, such as positive infinity and negative infinity. This ensures that (0,0) states are treated separately from the heuristic because they are representations of guaranteed wins or losses rather than estimated values.

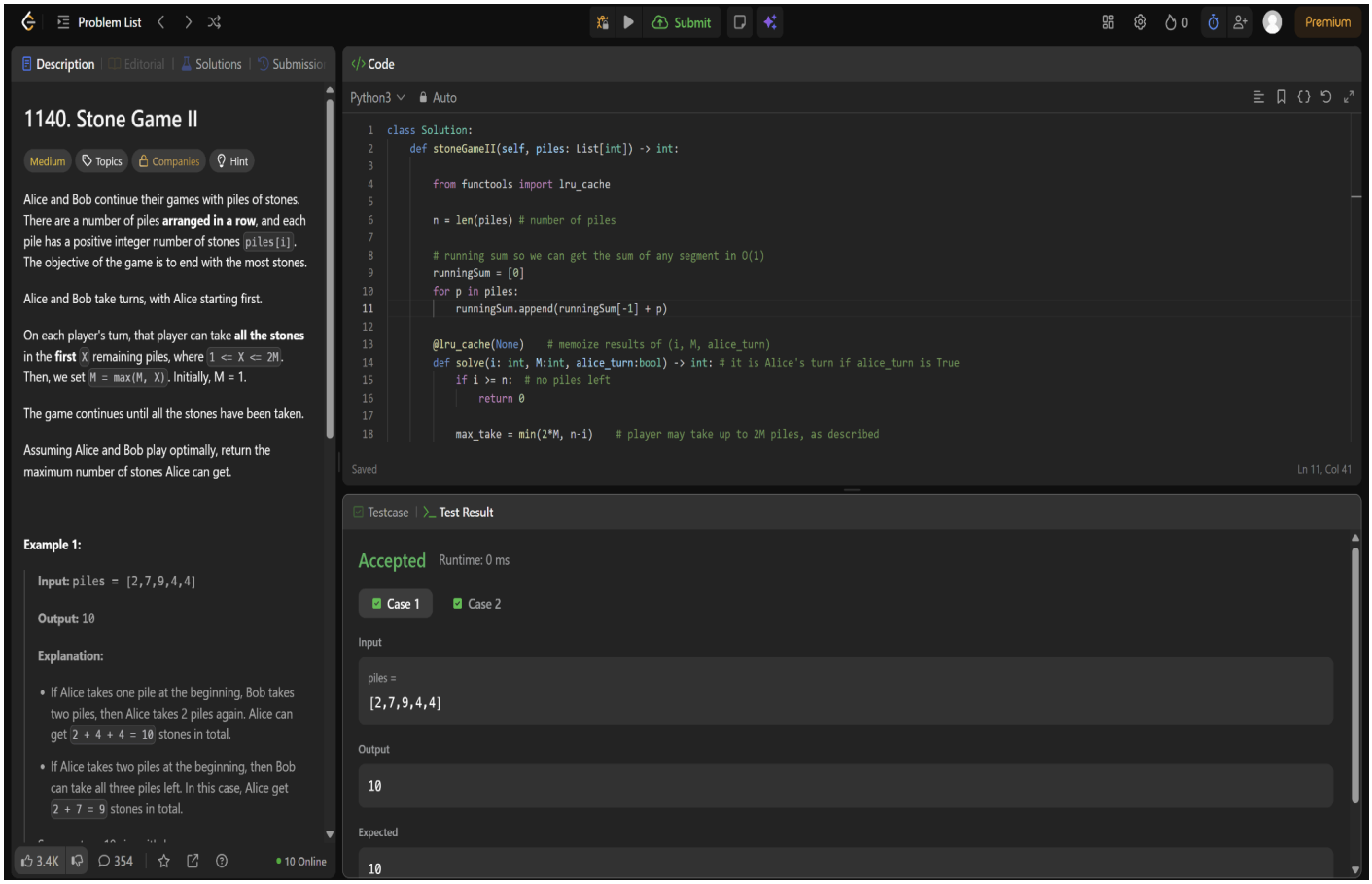
In comparison, the standard 2-ply minimax approach without a heuristic assigned many of these same frontier states a value of 0, which means that it could not effectively distinguish which positions were stronger. By using this equal-heap heuristic, the algorithm would be able to rank some states as more favorable than others even without searching deeper, allowing it to

make a more informed decision as the 2-ply cutoff. This improves performance because the algorithm does not need to explore the full game tree to identify promising moves, even though the result could sometimes be less accurate than a full-depth search since the evaluation is based upon an approximation rather than guaranteed outcomes.

Leetcode Stone Game II:

1140. Stone Game II

Screenshot of accepted code:



Code used:

```
class Solution:
    def stoneGameII(self, piles: List[int]) -> int:

        from functools import lru_cache

        n = len(piles) # number of piles

        # running sum so we can get the sum of any segment in O(1)
        runningSum = [0]
        for p in piles:
            runningSum.append(runningSum[-1] + p)

        @lru_cache(None)    # memoize results of (i, M, alice_turn)
        def solve(i: int, M:int, alice_turn:bool) -> int: # it is Alice's turn if alice_turn is True
            if i >= n:  # no piles left
                return 0

            max_take = min(2*M, n-i)    # player may take up to 2M piles, as described

            if alice_turn: # Alice tries to maximize her total
                best=0
                for X in range(1, max_take + 1):    # try taking X piles
                    taken = runningSum[i + X] - runningSum[i]     # stones Alice gets now
                    # future result if Bob plays next
                    best = max(best, taken + solve(i+X, max(M,X), False))
                return best

            else:   # Bob tries to minimize how many stone Alice ends with
                best = float('inf')
                for X in range(1, max_take + 1): # Bob chooses X to hurt Alice
                    # Bob does not add to Alice's score
                    # pass turn to Alice
                    best = min(best, solve(i + X, max(M,X), True))
                return 0 if best == float('inf') else best

        return solve(0,1,True) # start pile at 0, M at 1, Alice's turn
```

Explanation of the use of Minimax:

To approach this problem, I aimed to treat Alice as the MAX player and I treated Bob as the MIN player. On each turn, the recursion explores all allowed moves, and Alice (MAX) chooses the move that maximizes the stones that she can get. Bob (MIN) is choosing moves that minimize Alice's eventual total. No alpha-beta pruning was needed because the entire state space is relatively small and memoization was used to avoid redundant calculations.