Laboratory 2

Variant #3
Mariya Zacharneva and Ruslan Melnyk

Introduction

The task is to implement the Nim game, where the user plays against a bot. The bot must use the Minimax algorithm with alpha-beta pruning.

Minimax is a decision-making algorithm primarily used in two-player turn-based games like chess, tic-tac-toe, and checkers. Winning such a game implies solving the adversarial search problem.

Adversarial search is used in competitive environments where two (or more) players/agents have conflicting goals. It applies to games without a win-win outcome: when one player succeeds, the other one will always fail. These games are also often called zero-sum games. In this system, the Max player aims to maximize the score, while the Min player tries to do the opposite and minimize it.

To implement the algorithm, we consider all possible ways the game can unfold and assign a utility value to each outcome:

- If player one wins (who is trying to maximize the score), the utility is **1**.
- If player two wins (who is trying to minimize the score), the utility is **-1**.
- If the game ends in a draw (if possible), the utility is **0**.

Let's say I am player one, and it's my turn. Since I am trying to maximize the score:

1. I will consider all the possible moves (or actions) I can make from that position.
2. Then I will think from my opponent's perspective and consider what can they do from this position.
3. Similarly, my opponent, trying to minimize the score, would consider their own possible actions and see what I would do next.
4. This process continues recursively until we reach the terminal state (the end of the game), where we can determine the winner based on the utility value of the terminal state.
5. Now, backtracking, I will select actions that maximize my score, and my opponent will choose actions that minimize it.

Alpha-beta pruning is the optimization of the mini-max algorithm. It helps improve efficiency by "pruning" (cutting off) branches that do not need to be explored, reducing computational time.

The algorithm maintains two values:

- Alpha (α): The best value the Max player can guarantee.
- Beta (β): The best value the Min player can guarantee.

During tree traversal, if a branch's outcome is worse than what a player can already guarantee, the branch is pruned (ignored). However, even when utilizing alpha-beta pruning, the algorithm is still not perfect for more complex games. In games like chess, there are an enormous number of possible moves and game outcomes. For some initial state, the mini-max algorithm considers all the possible actions, and for each of them – all the possible actions for the second player, and so on until we get to the end of the game. It can become infeasible for the algorithm to select a move in a reasonable amount of time.

This is where the depth limit comes into play. After a certain number of moves ahead, the algorithm will stop and not consider the moves that will come after that. Since we haven't reached the end of the game, the algorithm must have an evaluation heuristic function that will say how good that state is, using incomplete data.

Advantages of a minimax algorithm:

- Easy to understand.
- If given enough depth, it always finds the optimal move.
- Works well with depth limit and alpha-beta pruning for a good evaluation function.

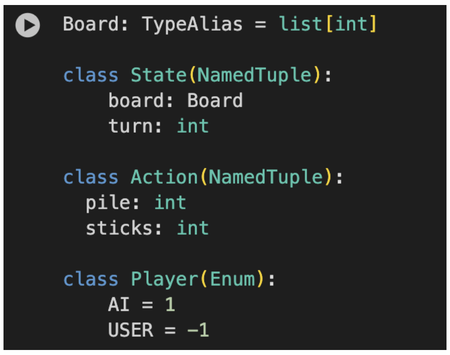
Disadvantages:

- Without the depth limit, it explores all possible moves, making it unusable even for the Nim game.
- With the depth limit, its performance relies on the evaluation function that must be a good heuristic
- Not Suitable for Stochastic Games – it cannot be used for chance-based games (e.g., backgammon) since it assumes perfect determinism.

Implementation

We chose AI (bot) to play maximizing and the User to play minimizing.

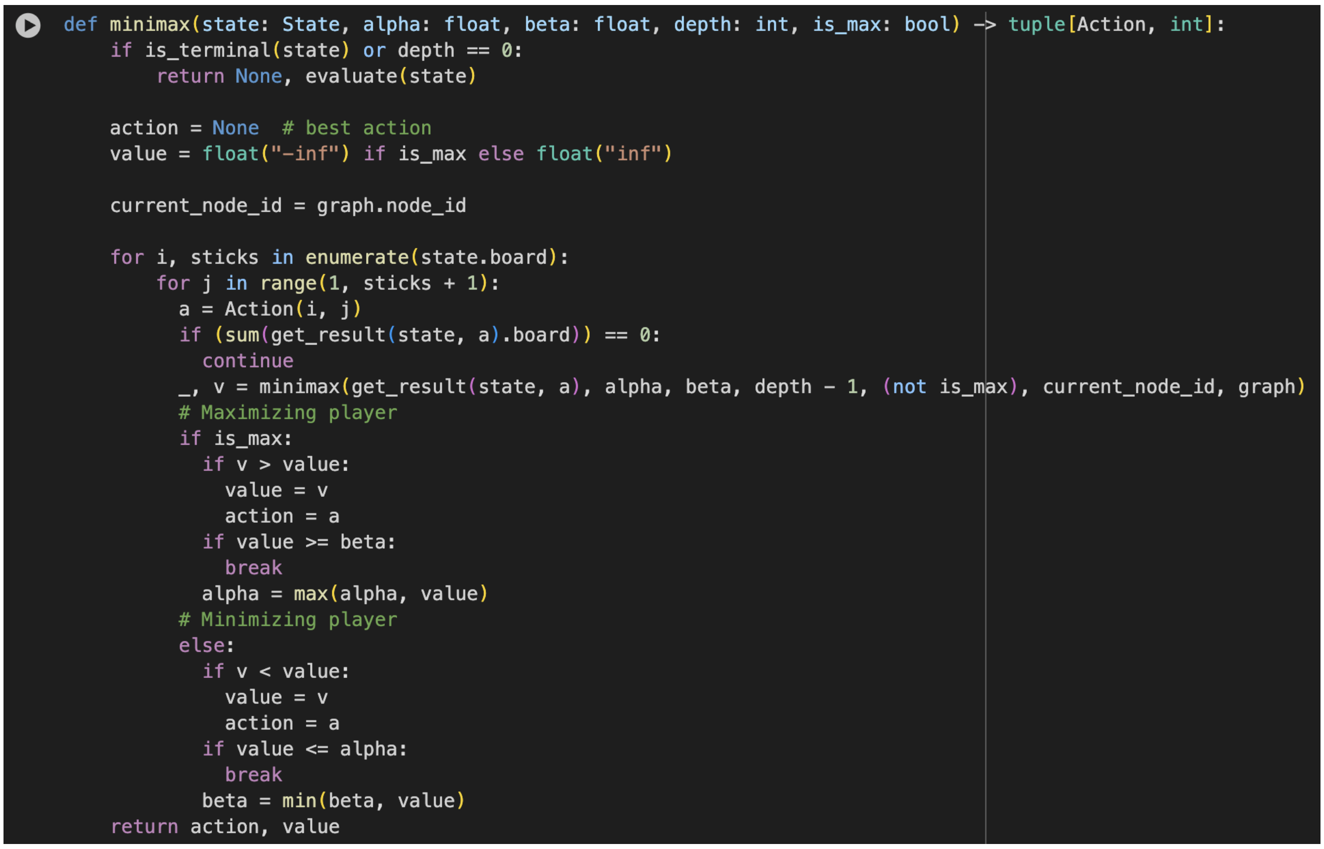
The `Board` of the game is a list, each index of which represents a pile, and the value represents the number of sticks in the pile. State contains the specific state of the board and the turn number. `Action` represents a move made by the player: removing some sticks from a particular pile.



```
Board: TypeAlias = list[int]

class State(NamedTuple):
    board: Board
    turn: int

class Action(NamedTuple):
  pile: int
  sticks: int

class Player(Enum):
    AI = 1
    USER = -1
```

The game starts by prompting the user to set the initial board. The user can enter the number of piles and a number of sticks in each pile.

The main function is a `minimax` function:

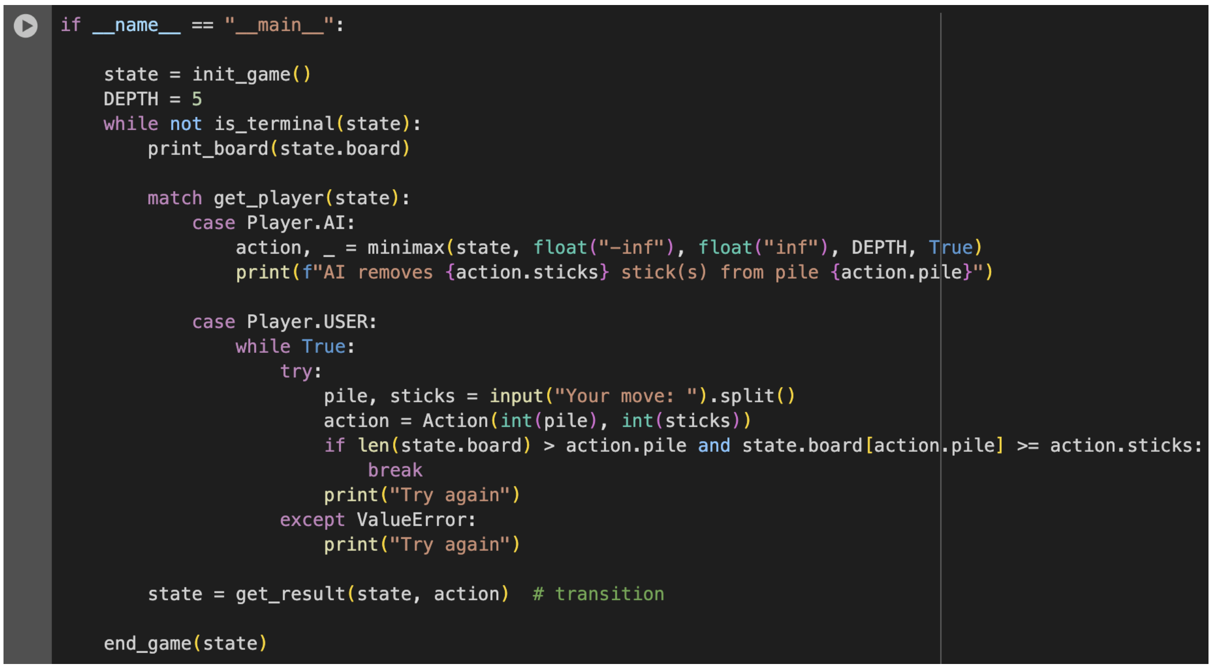


```python
def minimax(state: State, alpha: float, beta: float, depth: int, is_max: bool) -> tuple[Action, int]:
    if is_terminal(state) or depth == 0:
        return None, evaluate(state)

    action = None  # best action
    value = float("-inf") if is_max else float("inf")

    current_node_id = graph.node_id

    for i, sticks in enumerate(state.board):
        for j in range(1, sticks + 1):
          a = Action(i, j)
          if (sum(get_result(state, a).board)) == 0:
            continue
          _, v = minimax(get_result(state, a), alpha, beta, depth - 1, (not is_max), current_node_id, graph)
          # Maximizing player
          if is_max:
            if v > value:
              value = v
              action = a
            if value >= beta:
              break
            alpha = max(alpha, value)
          # Minimizing player
          else:
            if v < value:
              value = v
              action = a
            if value <= alpha:
              break
            beta = min(beta, value)
    return action, value
```

1. If the `state` is already terminal, we return the result of evaluating it: for a terminal state it is either 1 or -1, depending on who won the game.
2. Then we consider all possible actions we can take from that state. For each action, we see what the opposite player would achieve if we took that action. We want to pick the action which gives the best (highest) value.
3. Let's consider pruning for the maximizing player. If we find a value greater than or equal to beta, we can prune branches because the min player (who could have called "minimax" in a previous step) will never allow this path to be chosen. Additionally, we update alpha to keep track of the best value found so far, ensuring we maximize pruning in future iterations.
4. The situation is similar for the minimizing player (User in our case). But instead of the highest, we pick the action that produces the lowest possible value from the next possible max player moves. If we find the value less than or equal to alpha, we also prune branches because the max player (who could have called "minimax" in a previous step) will never allow this path to be chosen.

The user always moves first, then the AI bot, and then the user again, AI again… until we reach the terminal state.

Main game loop:

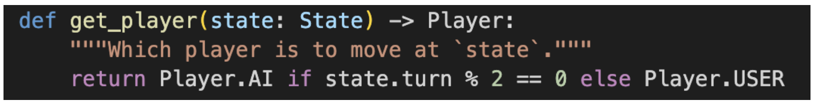


```python
if __name__ == "__main__":

    state = init_game()
    DEPTH = 5
    while not is_terminal(state):
        print_board(state.board)

        match get_player(state):
            case Player.AI:
                action, _ = minimax(state, float("-inf"), float("inf"), DEPTH, True)
                print(f"AI removes {action.sticks} stick(s) from pile {action.pile}")

            case Player.USER:
                while True:
                    try:
                        pile, sticks = input("Your move: ").split()
                        action = Action(int(pile), int(sticks))
                        if len(state.board) > action.pile and state.board[action.pile] >= action.sticks:
                            break
                        print("Try again")
                    except ValueError:
                        print("Try again")

        state = get_result(state, action)  # transition

    end_game(state)
```

We set the depth limit to be 5. However, it can be configured to a different value.

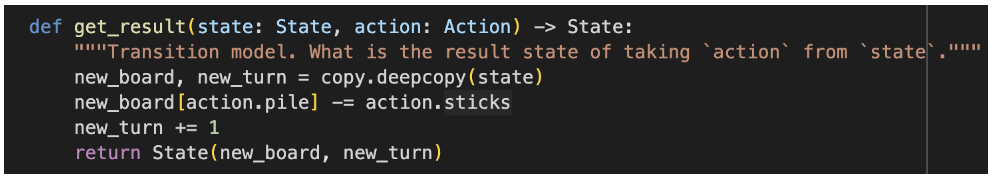
Helper functions

The `get_player` function returns whose turn to move in a particular state:



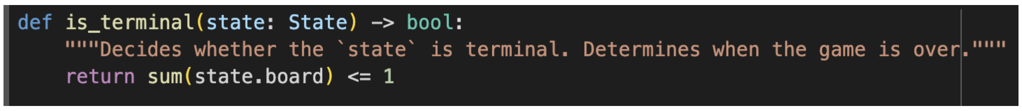
```python
def get_player(state: State) -> Player:
    """Which player is to move at `state`."""
    return Player.AI if state.turn % 2 == 0 else Player.USER
```

The `get_result` function defines the transition model. It returns the result of taking a particular action from a particular state.



```python
def get_result(state: State, action: Action) -> State:
    """Transition model. What is the result state of taking `action` from `state`."""
    new_board, new_turn = copy.deepcopy(state)
    new_board[action.pile] -= action.sticks
    new_turn += 1
    return State(new_board, new_turn)
```

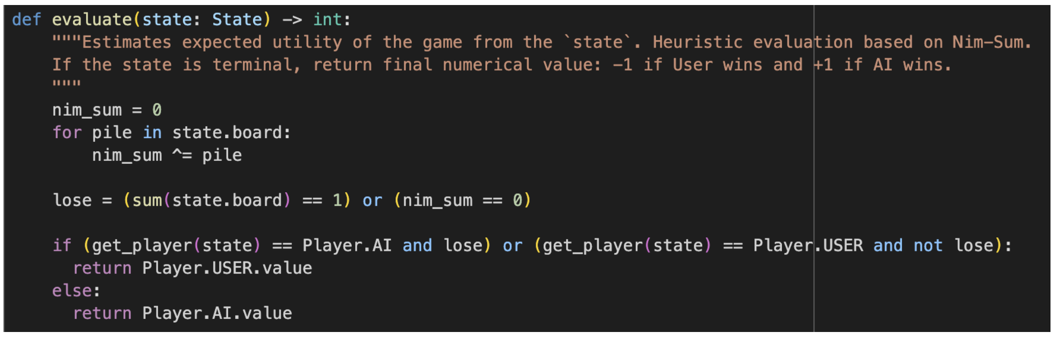
The `is_terminal` function decides whether the state is terminal (final state if the game is over). The game finishes if 0 or 1 stick is left.



```python
def is_terminal(state: State) -> bool:
    """Decides whether the `state` is terminal. Determines when the game is over."""
    return sum(state.board) <= 1
```

The `evaluate` function estimates the expected outcome of the game based on the current state. It uses a heuristic evaluation based on the Nim-Sum. This sum is part of the most well-known winning strategy for the Nim game. In a standard Nim game, the player who makes the first move has a winning strategy if and only if the Nim-Sum (the XOR of all heap sizes) is not zero. If the Nim-Sum is zero, then the second player has a winning strategy.

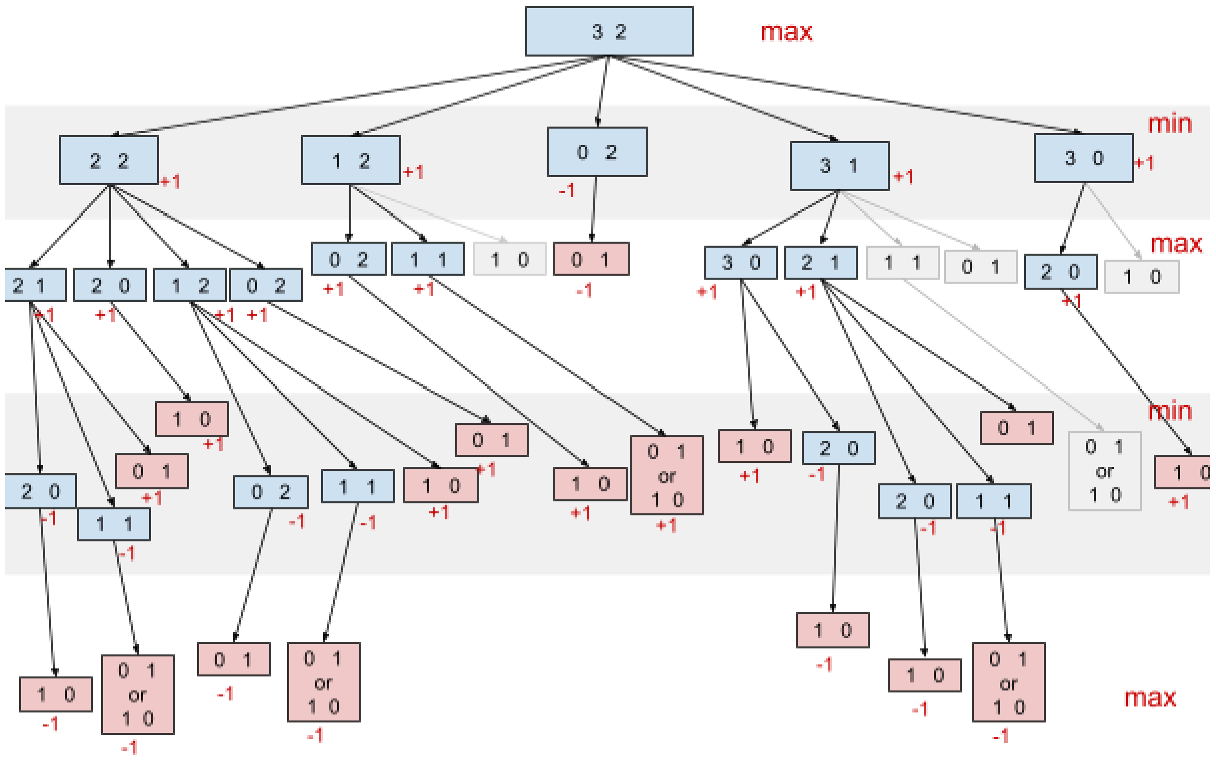
If the state is terminal, the function returns the final numerical value, depending on who has won.



```python
def evaluate(state: State) -> int:
    """Estimates expected utility of the game from the `state`. Heuristic evaluation based on Nim-Sum.
    If the state is terminal, return final numerical value: -1 if User wins and +1 if AI wins.
    """
    nim_sum = 0
    for pile in state.board:
        nim_sum ^= pile

    lose = (sum(state.board) == 1) or (nim_sum == 0)

    if (get_player(state) == Player.AI and lose) or (get_player(state) == Player.USER and not lose):
      return Player.USER.value
    else:
      return Player.AI.value
```

Examples

Here is an example of a whole decision tree with a starting condition of 2 piles, having 2 and 3 sticks. Pruned nodes are marked gray.

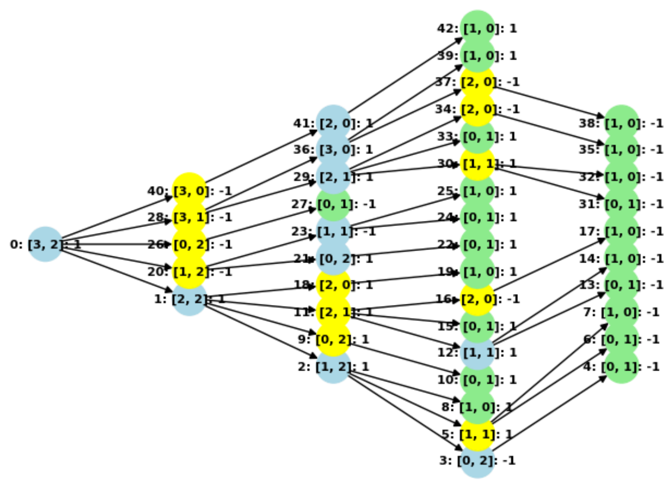
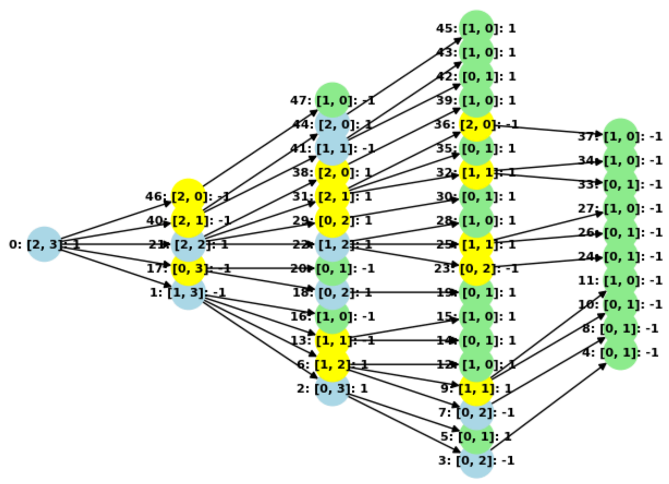
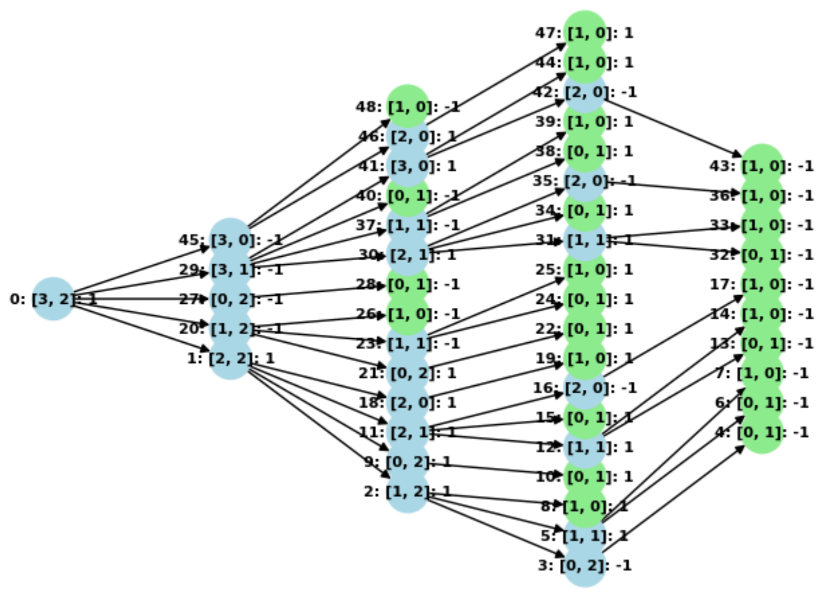


For such small examples, pruning doesn't yet provide much optimization. Below you can see the generated[1] simulation graphs for this example. Pruned nodes are marked with yellow, however, for most of them pruning happens when all childs are actually considered already. This is because most nodes have just 2 or 3 children, which is not enough to fully see the pruning optimization power.

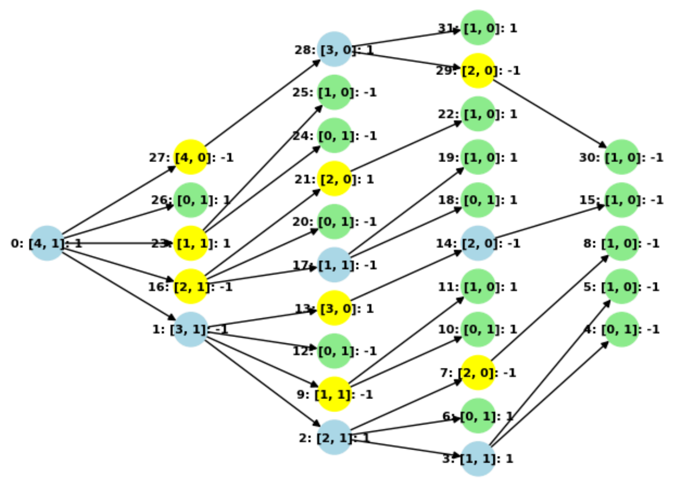
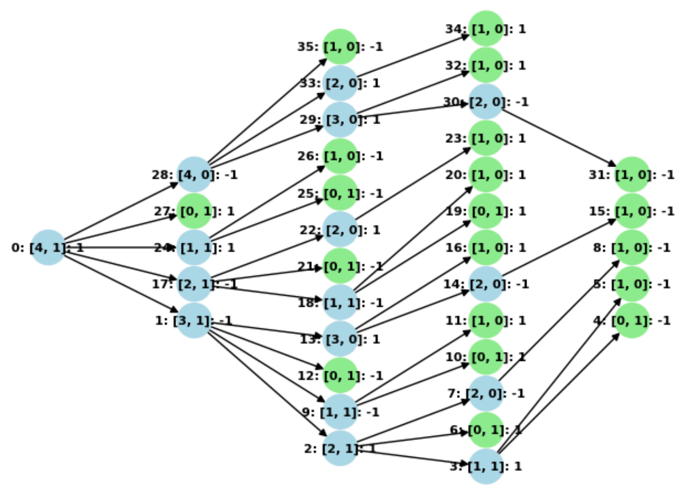
It is also worth noting that the optimization depends on an input state - even on the order of piles: for example, for initial state [3, 2] pruning eliminates 6 nodes as shown below, but for an initial state [2, 3] only 1 node gets pruned. This effect is explained by the

[1] If you would like to generate more simulation, you can access Google Colab here: mini-max for nim.ipynb . Note, that the notebook does not allow you to play interactively; it uses our functions only to simulate the decision tree for the first move of the bot.

implementation details: the algorithm considers children of a node in a particular order, so in some cases it can find a “good” node earlier or later (“good” node meaning the one which updates alpha\beta value and later allows to prune more nodes).

Initial state [3, 2] With pruning: 43 nodes considered	Initial state [2, 3]With pruning: 48 nodes considered
	
Initial state [2, 3] or [3, 2] Without pruning: 49 nodes considered	
	

Another example: decision tree for a starting condition of 2 piles with 1 and 4 sticks:

With pruning: 32 nodes considered	Without pruning: 36 nodes considered
	

Discussion

The evaluation function for the AI bot in the Nim game is based on the Nim-Sum (the bitwise XOR of all pile sizes). This is a well-known heuristic in combinatorial game theory because it helps determine whether the current player is in a winning or losing position.

How it works:

1. The function computes the Nim-Sum by XOR-ing all piles.
2. If nim_sum == 0, the position is a losing state (the opponent can force a win).
3. If nim_sum != 0, the position is a winning state (the current player has a guaranteed strategy to win).

While the Minimax algorithm itself can work with games like checkers or chess, the same evaluation function (based on Nim-Sum) cannot be used for those games, because of the following reasons:

1. The games as checkers or chess use a board, not piles.
2. These games require analyzing piece positions, control of key areas, and strategy (like center control, king safety, and mobility). The Nim-Sum only works for structured, mathematical games like Nim, where the winning strategy is purely based on pile sizes.

3. The Nim-Sum method is discrete (win/lose only), making it unsuitable for dynamic board games where one position may be more preferable to the other (some positions may have a heuristic value of 10, but others a value of 20).

Conclusion

We implemented the Nim game using the Minimax algorithm with alpha-beta pruning and a depth limit. This helped us better understand adversarial search and how AI makes decisions in turn-based strategy games.

One of the biggest challenges was optimizing the algorithm. The most difficult part was implementing alpha-beta pruning. Even though the idea is simple, it becomes tricky when translating it into code.