Laboratory 2

Variant 3, Group 13

By Saumya Shah and Anas Tagui

Introduction

The task involves the game of Nim, a well-known combinatorial game where two players take turns removing sticks from piles. The player forced to remove the last stick loses the game. The objective is to implement an AI player using the Minimax algorithm with Alpha-Beta Pruning to determine the optimal moves.

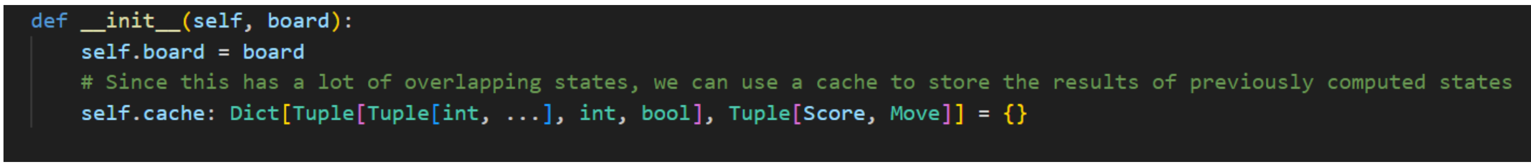
Algorithm Description

The Minimax algorithm is a decision-making algorithm used for turn-based games. It evaluates all possible game states recursively and determines the best move for the maximising player (AI) while considering the opponent's best possible moves.

- **Advantages:**
    - Guarantees the best possible move for the AI.
    - A structured way to explore all possible moves.
- **Disadvantages:**
    - Computationally expensive for large search spaces in games such as chess.
    - Requires optimisation techniques like alpha-beta pruning to improve efficiency.

Alpha-Beta pruning reduces the number of nodes evaluated by the Minimax algorithm, making it more efficient by eliminating branches that do not need to be explored.

Implementation



```python
def __init__(self, board):
    self.board = board
    # Since this has a lot of overlapping states, we can use a cache to store the results of previously computed states
    self.cache: Dict[Tuple[Tuple[int, ...], int, bool], Tuple[Score, Move]] = {}
```

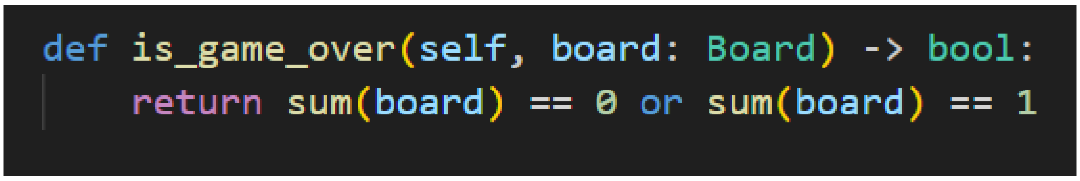
Initializes a Nim game with a given board state (a list of integers, each representing a pile of sticks).

Parameters:

- `board`: a list of integers, each representing a pile and the number of sticks in it.

Functionality:

- Stores the initial game board.
- Creates a cache (`self.cache`) to store previously computed results for optimization using **memoization** in the minimax function.

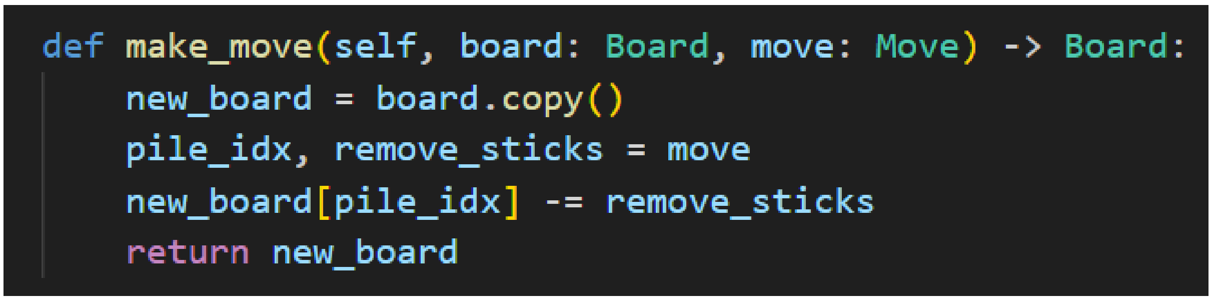


```python
def is_game_over(self, board: Board) -> bool:
    return sum(board) == 0 or sum(board) == 1
```

Checks whether the game has reached a terminal state.

Returns:

- `True` if the game is over (when the sum of sticks is 0 or 1).
- `False` otherwise.



```python
def make_move(self, board: Board, move: Move) -> Board:
    new_board = board.copy()
    pile_idx, remove_sticks = move
    new_board[pile_idx] -= remove_sticks
    return new_board
```

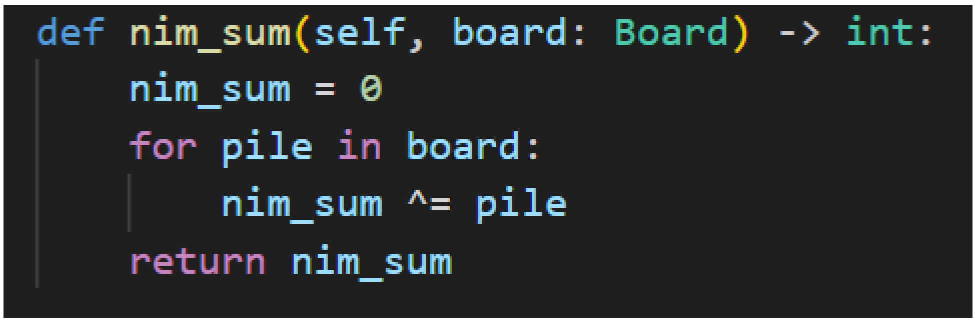
Returns a new board after making a move.

Parameters:

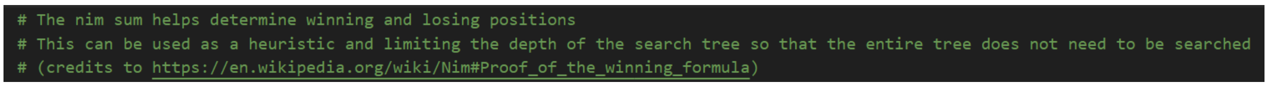
- `board`: current game board.
- `move`: tuple `(pile_idx, remove_sticks)` representing which pile and how many sticks to remove.

Returns:

- A **new** board (copy) after applying the move.



```python
def nim_sum(self, board: Board) -> int:
    nim_sum = 0
    for pile in board:
        nim_sum ^= pile
    return nim_sum
```



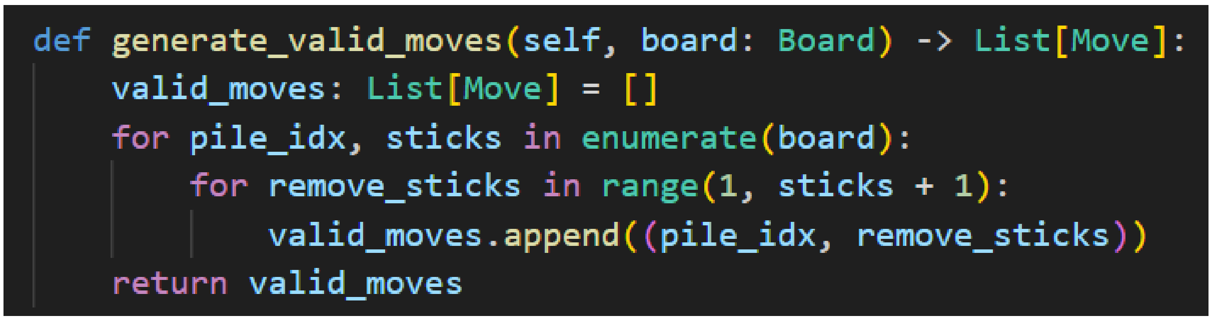
```python
# The nim sum helps determine winning and losing positions
# This can be used as a heuristic and limiting the depth of the search tree so that the entire tree does not need to be searched
# (credits to https://en.wikipedia.org/wiki/Nim#Proof_of_the_winning_formula)
```

Calculates the **nim sum**, which is the XOR of all pile sizes.

Returns:

- The XOR of all the piles.

In Nim, a non-zero nim sum indicates a **winning position**, and a zero nim sum indicates a **losing position** for the current player (according to game theory).



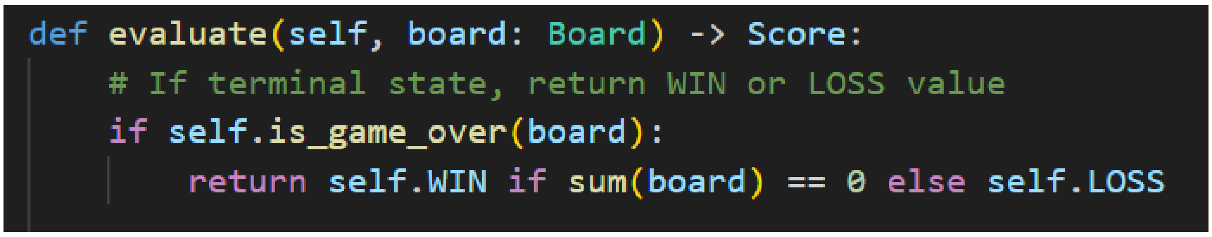
```python
def generate_valid_moves(self, board: Board) -> List[Move]:
    valid_moves: List[Move] = []
    for pile_idx, sticks in enumerate(board):
        for remove_sticks in range(1, sticks + 1):
            valid_moves.append((pile_idx, remove_sticks))
    return valid_moves
```

Generates all possible valid moves from the current board state.

Returns:

- A list of tuples, each representing a valid move `(pile_idx, remove_sticks)`.

These moves are used by the **minimax algorithm** to simulate all potential game states.



```python
def evaluate(self, board: Board) -> Score:
    # If terminal state, return WIN or LOSS value
    if self.is_game_over(board):
        return self.WIN if sum(board) == 0 else self.LOSS
```

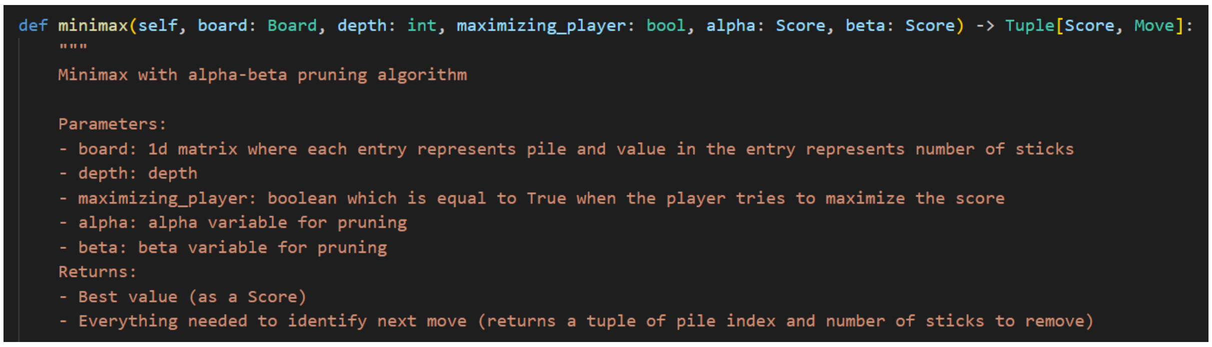
Evaluates the current board to assign a score based on how favorable it is.

Returns:

- A score (`WIN`, `LOSS`, or a fractional value if it's a non-terminal state).

How it works:

- If game is over:
  - Returns `WIN` if it's a winning position (i.e. sum is 0).
  - Returns `LOSS` if sum is 1 (you're forced to remove the last stick).
- Otherwise, evaluates based on **nim sum**.



```python
def minimax(self, board: Board, depth: int, maximizing_player: bool, alpha: Score, beta: Score) -> Tuple[Score, Move]:
    """
    Minimax with alpha-beta pruning algorithm

    Parameters:
    - board: 1d matrix where each entry represents pile and value in the entry represents number of sticks
    - depth: depth
    - maximizing_player: boolean which is equal to True when the player tries to maximize the score
    - alpha: alpha variable for pruning
    - beta: beta variable for pruning
    Returns:
    - Best value (as a Score)
    - Everything needed to identify next move (returns a tuple of pile index and number of sticks to remove)
```

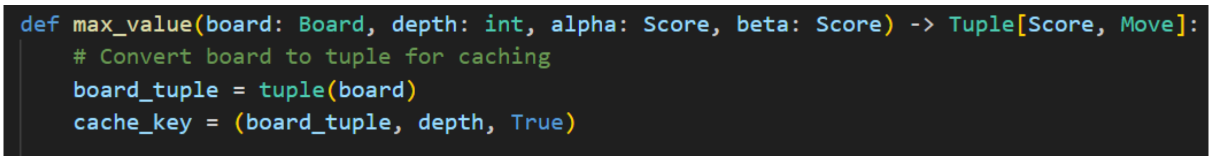
Implements the **Minimax algorithm** with **alpha-beta pruning** to choose the best move for the AI.

Parameters:

- `board`: Current game state.
- `depth`: Current depth in the game tree.
- `maximizing_player`: Whether AI is trying to maximize or minimize.
- `alpha`: Best score that maximizer can guarantee.
- `beta`: Best score that minimizer can guarantee.

Returns:

- Tuple of best score and best move `(score, move)`.

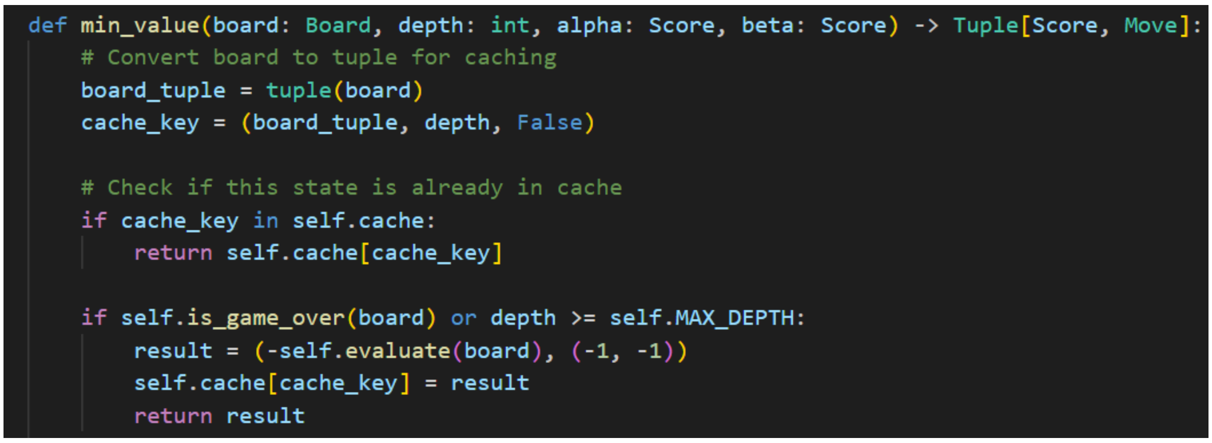


```python
def max_value(board: Board, depth: int, alpha: Score, beta: Score) -> Tuple[Score, Move]:
    # Convert board to tuple for caching
    board_tuple = tuple(board)
    cache_key = (board_tuple, depth, True)
```

We use it when AI is trying to maximize its score.

How it works:

- For every valid move:
    - Simulates the move.
    - Calls `min_value()` recursively.
    - Picks the move with the highest score.
- Uses `alpha-beta pruning` to avoid unnecessary calculations.



```python
def min_value(board: Board, depth: int, alpha: Score, beta: Score) -> Tuple[Score, Move]:
    # Convert board to tuple for caching
    board_tuple = tuple(board)
    cache_key = (board_tuple, depth, False)

    # Check if this state is already in cache
    if cache_key in self.cache:
        return self.cache[cache_key]

    if self.is_game_over(board) or depth >= self.MAX_DEPTH:
        result = (-self.evaluate(board), (-1, -1))
        self.cache[cache_key] = result
        return result
```

Used When Opponent (human) is minimizing the AI's score.

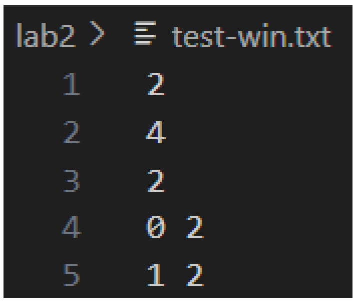
How it works:

- Similar to `max_value`, but tries to **minimize** the score.
- Uses alpha-beta pruning by updating `beta`.

Discussion

Test Cases:
The two following test cases were used and visualised (note: the test cases were visualised using SVG in HTML files, which are available in the repository)

test-win

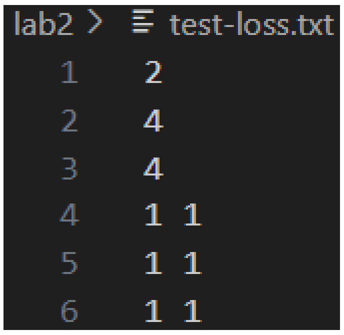


lab2 > test-win.txt

```
2
4
2
0 2
1 2
```

Start with two piles [4, 2]
(The visualisation for this may be found in `test_case_win.html`)

test-loss



lab2 > test-loss.txt

```
2
4
4
1 1
1 1
1 1
```

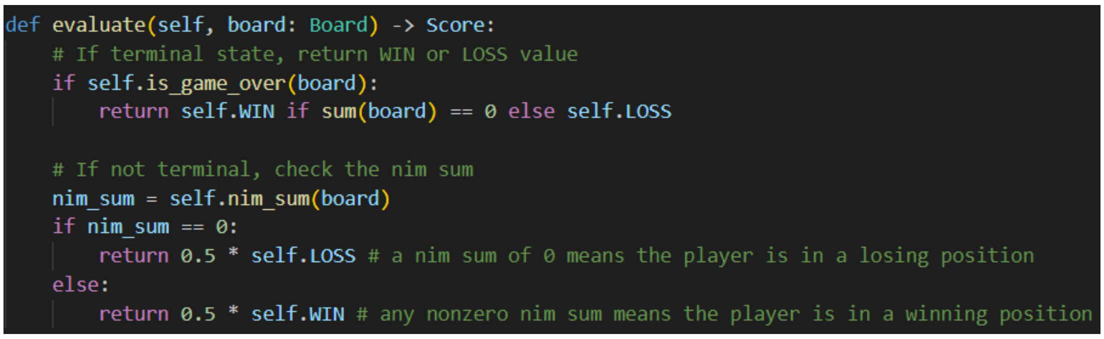
Start with two piles [4, 4]
(The visualisation for this may be found in `test_case_loss.html`)

These test cases were chosen to represent two fundamental scenarios in Nim: a starting position that is a winning position for the first player under optimal play, and a starting position that is a losing position for the first player against an optimally playing opponent.

In the first case, the game begins with two piles containing 4 and 2 sticks respectively. This can be represented as the state [4,2]. By taking two sticks from pile 0 (ie. state is [2, 2]), the maximising player can reach a position where only one stick is left after the AI agent moves (as shown in the [2, 2] branch of the visualisation). Therefore, an optimal play can defeat the AI agent in this case.

In the second case, the game begins with two piles both containing 4 sticks. This can be represented as the state [4, 4]. As seen in the visualisation, no evaluation score for any of the branches gives us a positive result. Even optimal play cannot beat the AI agent in this case. In this case (as shown in the test case visualisation), taking one stick at a time from either pile can delay the loss, but there is no way to win against an optimal player.

Evaluation Function:
The evaluation function is as follows:

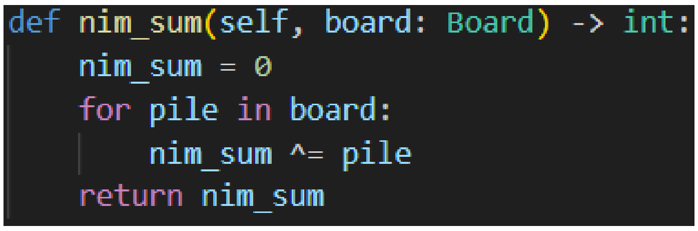


```python
def evaluate(self, board: Board) -> Score:
    # If terminal state, return WIN or LOSS value
    if self.is_game_over(board):
        return self.WIN if sum(board) == 0 else self.LOSS

    # If not terminal, check the nim sum
    nim_sum = self.nim_sum(board)
    if nim_sum == 0:
        return 0.5 * self.LOSS # a nim sum of 0 means the player is in a losing position
    else:
        return 0.5 * self.WIN # any nonzero nim sum means the player is in a winning position
```

If it is the end of the game, then the sum of the sticks is checked; if it is 0, then it returns the `WIN` value since the other player was forced to take the last stick. If it is 1, then it returns the `LOSS` value since you will be forced to take the last stick.

For non-terminal states (ie. when max depth is reached), the nim-sum is calculated as follows:



```python
def nim_sum(self, board: Board) -> int:
    nim_sum = 0
    for pile in board:
        nim_sum ^= pile
    return nim_sum
```

If the nim-sum is 0, the position is losing under optimal play, as the opponent can always respond in a way that maintains a winning strategy. However, since the position is not already a loss, it is assigned a score of half the `LOSS` value. If the nim-sum is nonzero, the position is winning, meaning the current player has a move that forces the opponent into a losing position. Similarly, it is assigned a score of half the `WIN` value.
(A full proof of this is available at our source here)

This will not work for chess or checkers at all for other games such as chess or checkers. This is because:
- The boards for each game are different; there is no direct mapping from a chess or checkers board to a Nim board or vice-versa.
- The terminal conditions are different: counting the total number of pieces on the board does not correspond to any terminal conditions in chess or checkers (the number of pieces by colour must be counted for checkers).
- The nim-sum heuristic is specific to nim and unlikely to work for other games.

Conclusion

Learning Outcomes

- We understood and implemented the **Minimax algorithm** and how **alpha-beta pruning** optimizes it.
- We learned how to evaluate game states using **Nim Sum**, and how to adapt logic for **normal vs misère** modes.
- We improved our **Python programming skills**, especially in recursion, logic structuring, and input handling.

Challenges

- Handling edge cases, especially in **misère mode**, where the logic is different when few sticks remain.
- Debugging and maintaining code clarity while integrating **minimax with pruning and different difficulty levels**.
- Making the AI feel “human” without making it unbeatable.

Improvements

- Add a **GUI** for better user interaction.
- Improve **code structure** using object-oriented programming.
- Include **more AI levels** or randomness to make the game more engaging for beginners.