Laboratory 2

Variant 3

Group 3

By Juan Manuel Aristizabal Henao and Krzysztof Kotowski

Exercise 2: Two-player deterministic games

Introduction

The task was to do a program that played the game called **Nim**. The program must use the **min-max algorithm with alpha-beta pruning** for choosing its moves. The game has a board that is a list of piles, their value represents the number of sticks in that pile.

How to play Nim:

- The user chooses the number of piles and the number of sticks in each pile.
- The user makes the first move by specifying the pile and the number of sticks to take from that pile then the agent does the same, interleaving their turns.
- The player that takes the last stick of the board loses the game.
- The game ends when one or no sticks are left on the board.

Min-max algorithm with alpha-beta pruning

The min-max algorithm is used in two-player deterministic games like Nim, chess, etc., for powering the decision-making of programs that play these games. This algorithm aids in determining the best possible move by considering all the future possible moves of the players (e.g. two players).

In this algorithm, the program (e.g. AI) tries to maximize the value of its movements while the user (e.g. human player, opponent) tries to minimize the value of the movements. The whole game can be represented by a tree, where each of the nodes is a game state and its edges are the possible moves to that game state, and the leaves are terminal states

where using an evaluation function a value is assigned to it. At each level of the tree, the player responsible for the move at that level decides which game state is followed, depending on whether they are the maximizing or the minimizing player and the values of the children of that node. The algorithm decides the path from bottom to root of the tree, visiting each of the nodes and returning the state of the game that the player responsible for that level chooses.

A pseudo-code representing the process for the min-max algorithm can be seen in the picture below:

Algorithm Min-Max

```
Input: position, depth, isMax
Output: bestValue
Procedure MinMax(position, depth, isMax):
    if depth = 0 or position is terminal then
        return the value of position;
    end
    if isMax then
        bestValue ← −∞; for move in possible moves from position do
            value ← MinMax(position after move, depth-1, False); bestValue ← max(bestValue, value);
        end
    end
    else
        bestValue ← ∞; for move in possible moves from position do
            value ← MinMax(position after move, depth-1, True); bestValue ← min(bestValue, value);
        end
    end
    return bestValue;
end
```

** Picture taken from EARIN Lecture slides: “Lecture 4 – Two player deterministic games and Constraint Satisfaction Problem.pdf” of 25L semester.

Alpha-beta pruning is an optimization of the min-max algorithm, in which game states that will certainly not be taken are ignored by the search algorithm. Hence the algorithm simply does not continue visiting all the children of that node, if it is deemed that from that node there is already a clear candidate for a path. This algorithm works by keeping alpha and beta values, which represent the best value that the maximizer or the minimizer respectively, can guarantee. If at any moment a node has a value for alpha greater or equal to the beta, then the rest of the nodes of that branch can be safely pruned from the search.

This optimization of the algorithm reduces the number of computations and memory used, by ignoring these branches that are irrelevant for the optimal decision, and therefore, not needing to call the min-max algorithm in those ignored nodes. The drawbacks of this algorithm it is that even with optimizations, its scalability is limited to

the complexity of the game, requiring a limit for the depth of the search since the number of possible moves on complex games grows exponentially with the depth.

A pseudo-code depicting an implementation of the min-max algorithm with alpha-beta pruning can be seen on the picture below:

Algorithm Alpha-Beta Pruning

```
Function AlphaBeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value ← −∞
        for each child of node do
            value ← max(value, AlphaBeta(child, depth - 1, α, β, false))
            α ← max(α, value)
            if β ≤ α then
                break
        return value
    else
        value ← +∞
        for each child of node do
            value ← min(value, AlphaBeta(child, depth - 1, α, β, true))
            β ← min(β, value)
            if β ≤ α then
                break
        return value
```

** Picture taken from EARIN Lecture slides: “Lecture 4 – Two player deterministic games and Constraint Satisfaction Problem.pdf” of 25L semester.

Implementation

The Nim game implementation with the min-max algorithm with alpha-beta pruning was done using a provided template for the specific variant of our laboratory exercise.

Min-max function

```
def minimax(self, board: list[int], depth: int, maximizing_player: bool,
            alpha: int | float, beta: int | float) -> tuple[int | float, tuple[int, int]]:
```

Takes the following parameters:

- Board: 1d matrix where each entry represents pile and value in the entry represents number of sticks
- Depth: depth to reach in the possible moves tree.
- 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 a tuple containing the best value and a tuple indicating the best move

```
if depth == 0 or sum(board) <= 1:
    return self.eval_func(sum(board), maximizing_player), self.NIM_EMPTY_MOVE
```

Stopping conditions for the recursion and call to the evaluation function, returning the value of that end node given by that evaluation function, and a tuple indicating empty move.

```
if maximizing_player:
    value = -Nim.NIM_INF
    for iter_pile_index in range(0, len(board)):
        for iter_sticks_taken_from_pile in range(1, board[iter_pile_index] + 1):
```

Maximizing player level decision node. Setting its initial value to negative infinity and iterating through each of its children by the order of the pile and the number of sticks that can be taken from that pile.

```
tempBoard = copy.deepcopy(board)
tempBoard[iter_pile_index] = tempBoard[iter_pile_index] - iter_sticks_taken_from_pile
```

A copy of the current state game is made and edited to match the game state of the child of the current node.

```
value = max(value, self.minimax(tempBoard, depth - 1, maximizing_player: False, alpha, beta)[0])
```

The minimax function is called for the recursion with the child board, the depth reduced by one, the minimizing player, and the current values of alpha and beta. The returned best value from that child is then compared with the current value, and the maximal value between them is set as new current value.

```
if value > alpha:
    alpha = value
    best_move = (iter_pile_index, iter_sticks_taken_from_pile)
```

Then, the current value is checked if it surpasses the current alpha value, and if it does, the current value is set as the new alpha value and the best move is set to be the tuple made from the current pile number and the value of the amount of sticks to be taken from that pile.

```
if beta <= alpha:
    break
```

Finally, if the value of beta is lower or equal to the value of alpha, then the iterations over that node are stopped.

```
else:
    value = Nim.NIM_INF
    for iter_pile_index in range(0, len(board)):
        for iter_sticks_taken_from_pile in range(1, board[iter_pile_index] + 1):
            tempBoard = copy.deepcopy(board)
            tempBoard[iter_pile_index] = tempBoard[iter_pile_index] - iter_sticks_taken_from_pile
            value = min(value, self.minimax(tempBoard, depth - 1, maximizing_player: True, alpha, beta)[0])
            if value < beta:
                beta = value
                best_move = (iter_pile_index, iter_sticks_taken_from_pile)
            if beta <= alpha:
                break
```

Minimizing player level decision node. Similar code flow, but the differences are that:

- the value is initialized as positive infinity,

- calls the minimax recursion to the maximizing player,
- checks if the returned best value from minimax is lower than the current value, and assigns it to the current value,
- Checks if the current value of beta is lower than the current value, and if so, it assigns this current value to beta.

```
return value, best_move
```

Finally, the minimax function returns the tuple containing the best value and the best move.

Evaluation Function

```
def eval_func(self, sum_board: int, maximizing_player: bool) -> int:
```

Takes parameters for evaluating the value of the leaf node depending on the end game state and which is the current player at that moment.

```
if sum_board == 0:
    if maximizing_player:
        return self.NIM_WIN_VALUE
    else:
        return self.NIM_LOSS_VALUE
elif sum_board == 1:
    if maximizing_player:
        return self.NIM_LOSS_VALUE
    else:
        return self.NIM_WIN_VALUE
```

Return leaf evaluated value. This value depends on which player takes the last stick, which is determined as follows:

- If the end game state is 0:
    - Returns the win value (1) if it is the turn of the maximizing player.
    - Returns the loss value (-1) if it is the turn of the minimizing player.
- If the end game state is 1:
    - Returns the loss value (-1) if it is the turn of the maximizing player.
    - Returns the win value (1) if it is the turn of the minimizing player.

Function: agentDecision

```
def agentDecision(self):
    """
    Calls the min-max algorithm to choose the agents move.

    Returns:
        A tuple representing the agent's move, containing the pile number and the sticks amount.
    """
    _, next_move = self.minimax(self.board, sum(self.board),  maximizing_player: True, -self.NIM_INF, self.NIM_INF)
    return next_move
```

Function in charge of calling the min-max algorithm implementation with proper initial parameters and return the tuple containing the move of the agent.

Function: makeaMoveWithEndHandling

```
def makeAMoveWithEndHandling(self, pile_nr: int, sticks_amount: int, isPlayer: bool) -> int:
```

Takes the specified pile number and the sticks amount on the move including the player that makes the move.

```
if pile_nr < 0 or pile_nr >= len(game.board):
    if not isPlayer:
        raise Exception(f"ERROR! Problem with agent! move {pile_nr}, {sticks_amount} is invalid!")
    print("Pile with number " + str(pile_nr) + " doesn't exist, pick another")
    return -1
if sticks_amount < 1:
    if not isPlayer:
        raise Exception(f"ERROR! Problem with agent! move {pile_nr}, {sticks_amount} is invalid!")
    print("You need to take at least one stick (you wanted to take " + str(sticks_amount) + ")!")
    return -1
if game.board[pile_nr] < sticks_amount:
    if not isPlayer:
        raise Exception(f"ERROR! Problem with agent! move {pile_nr}, {sticks_amount} is invalid!")
    print("You can take at most " + str(game.board[pile_nr]) + " sticks from pile " + str(
        pile_nr) + ", while you wanted to take " + str(sticks_amount) + "!")
    return -1
```

Checks the validity of the move, prints an appropriate message when the move is invalid and returns -1 (indicates an invalid move).

```
if not isPlayer:
    print("Agent's move: pile chosen = " + str(pile_nr + 1) + ", sticks taken =" + str(sticks_amount))
self.board[pile_nr] -= sticks_amount
```

Prints the move of the agent and updates the sticks amount in the specified pile.

```
if sum(self.board) == 0:
    print(self.board)
    if isPlayer:
        print("You lose")
    else:
        print("You win")
    return 1
elif sum(self.board) == 1:
    print(self.board)
    if isPlayer:
        print("You win")
    else:
        print("You lose")
    return 1

return 0
```

Checks if a game end state has been reached, displays an appropriate message if so and returns 1 (indicating a game end state), or returns 0 (if no game end state has been reached yet).

Main game loop

All the code prior to the main loop was kept as in the provided template. The main loop of the game is implemented as follows:

```
# Initial variables for handling game loop
hasGameEnded = False
isPlayersTurn = True
while hasGameEnded is False:
    print("Pile state %s" % game.board)
```

Initial variables for checking if a game end state has been reached (hasGameEnded) and which player turn is (isPlayerTurn), the latter is True for the user's turn and False for the agent's turn.

```
while True:
    pile_nr, sticks_amount = None, None

    if isPlayersTurn:
        user_input = input("Your move (<pile_nr> <sticks_amount>): ")
        try:
            a = [int(x) for x in user_input.split()]
            if len(a) != 2:
                print("Wrong number of input parameters! Provide exactly 2 integers")
                continue
        except Exception as e:
            print("Invalid input for the move, please try again")
            continue
        pile_nr, sticks_amount = a[0] - 1, a[1]
    else:
        # Agent's turn
        pile_nr, sticks_amount = game.agentDecision()

    move_result = game.makeAMoveWithEndHandling(pile_nr, sticks_amount, isPlayersTurn)

    if move_result == -1:
        continue
    elif move_result == 1:
        hasGameEnded = True

    break
```

This is a middle loop, in which the user is asked for its move and its input is parsed and checked. For the agent‘s turn, its function agentDecision is called returning its move. Then, the move of the current player is checked, and depending on the output of makeaMoveWithEndHandling: the user is asked for a different move if the move is invalid, the hasGameEnded is set true if a game end state has been reached, or the game continues and it breaks from this middle loop to continue the main one.

```
# After move is made for current player, switch
# the current-turn player
isPlayersTurn = not isPlayersTurn
```

Final instruction in the main loop, the player is switched to the next one.

Discussion

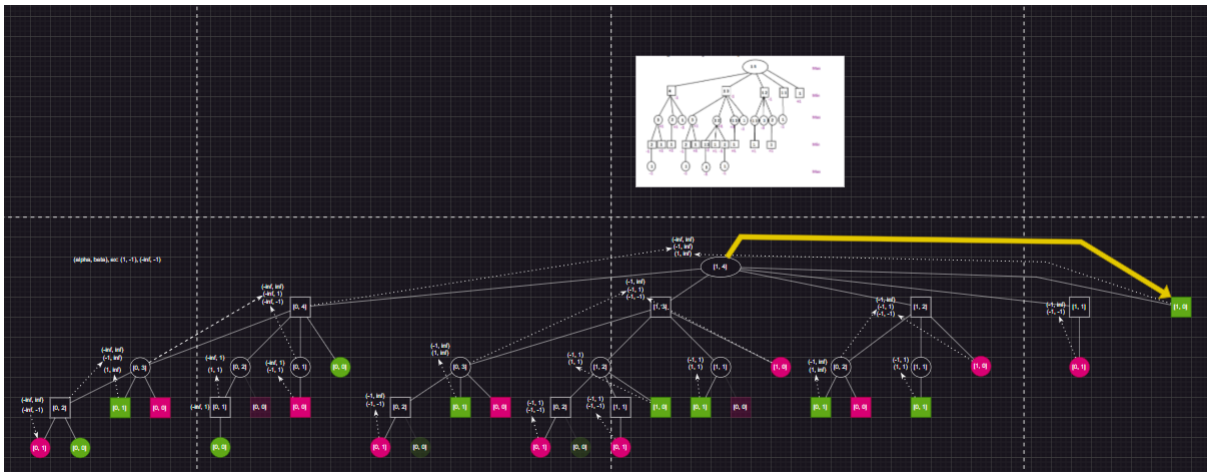
Evaluation function

Our evaluation function assigns the value of +1 to winning terminal nodes and –1 to losing ones. If we applied the same principle to eval-functions for chess or checkers, then it would be very inefficient, because reaching the “terminal node” in them requires not only making players do a significant number of turns, but also the number of possible moves with available pieces is great. Therefore, it is not realistically possible to reach the win/lose state at any point at the beginning rounds of the game, which renders this method useless. If the function would assign values to early moves, then this would be actually useful.

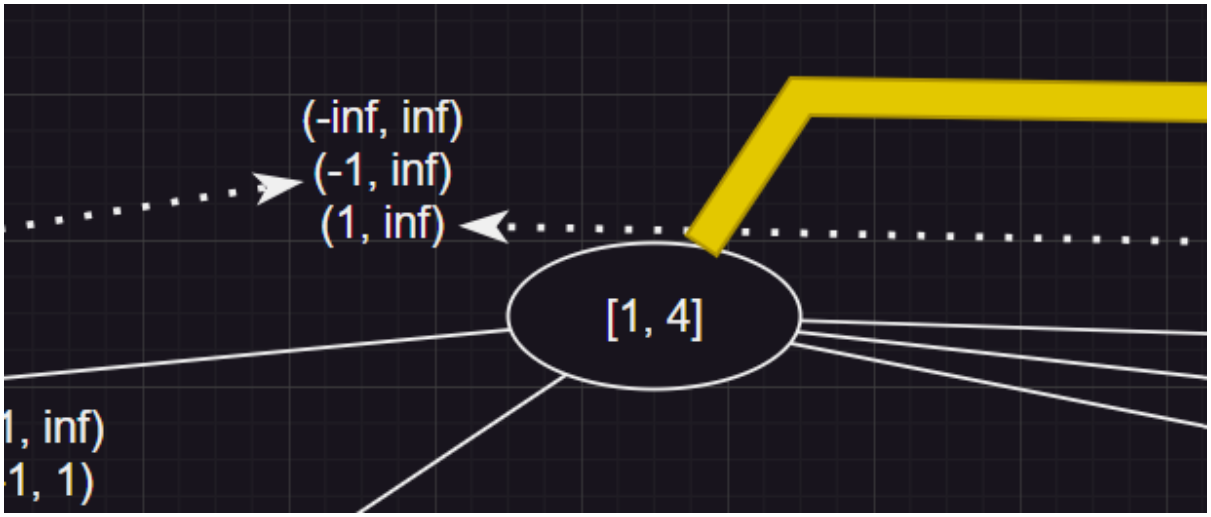
Case studies:

Agent starting with board [4, 1]

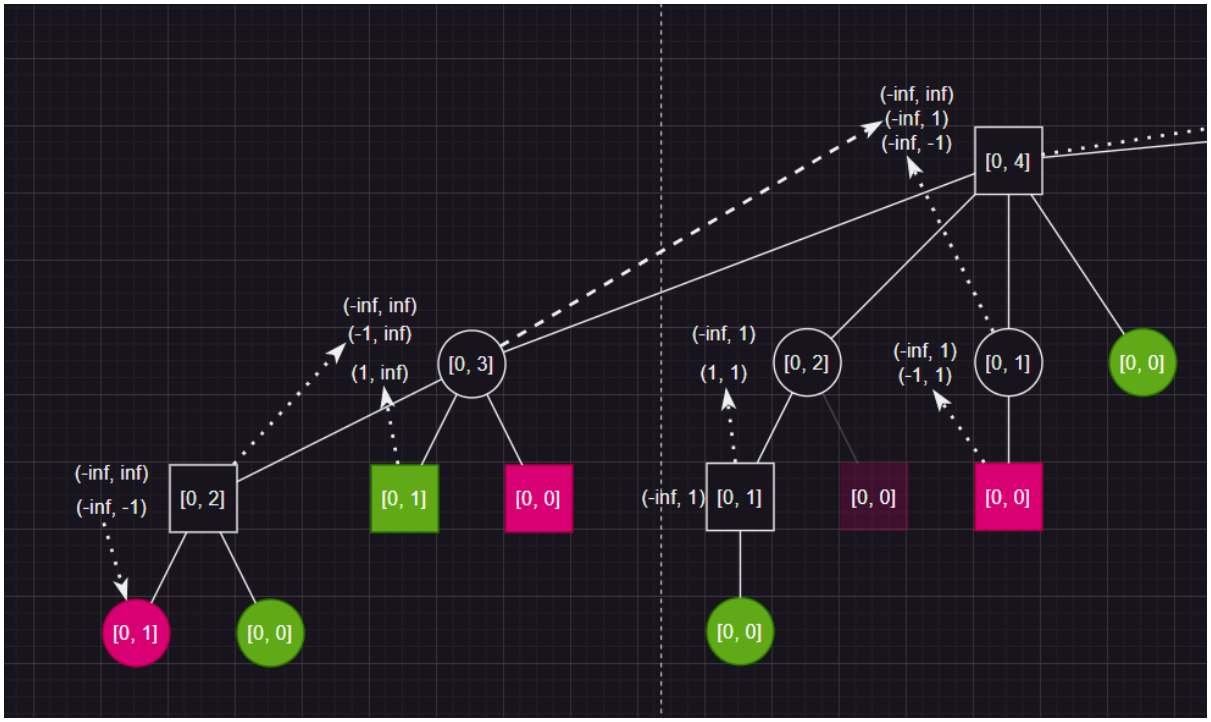
The whole tree for piles with sticks of [4, 1]:



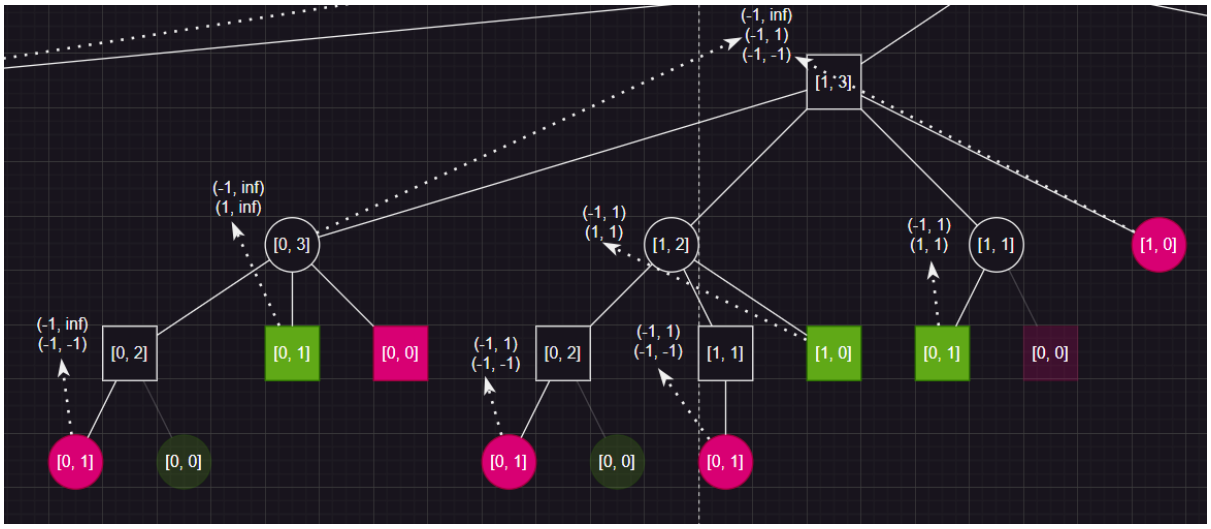
Starting node:



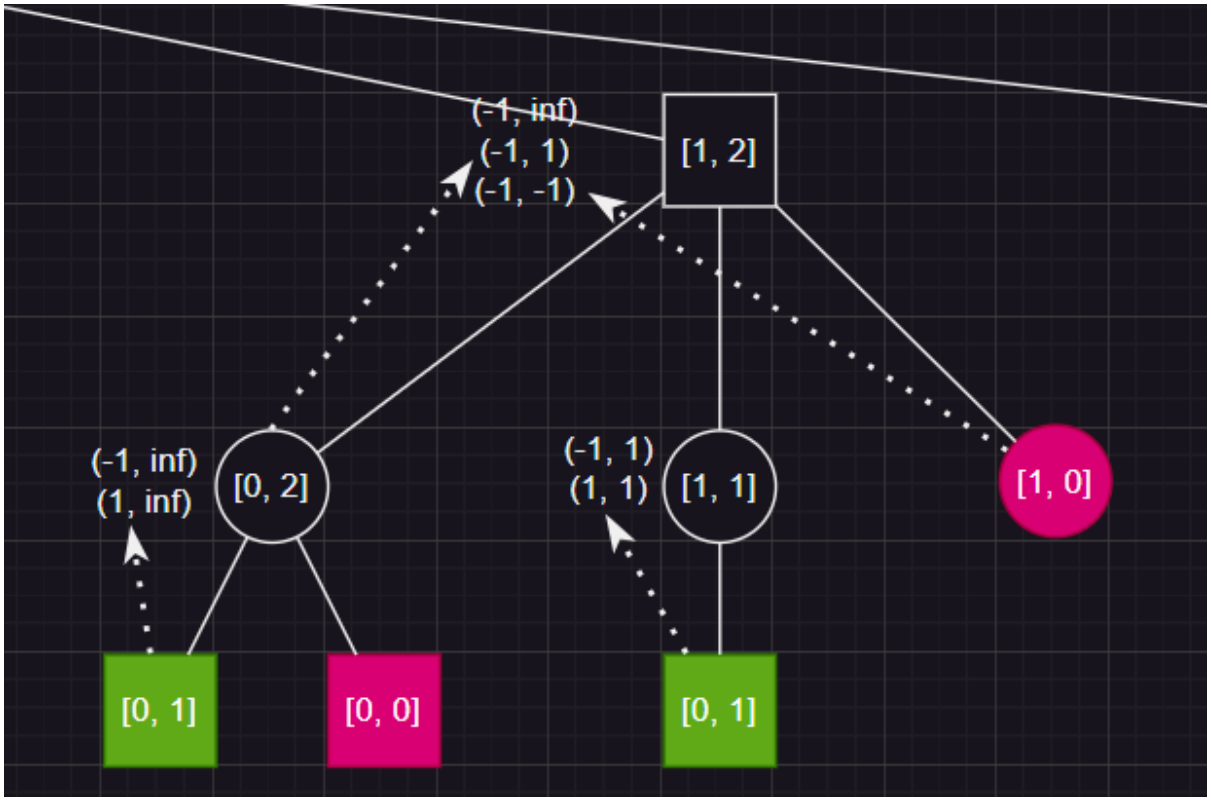
Agent chooses to take 1 stick from pile 1 -> [0, 4]



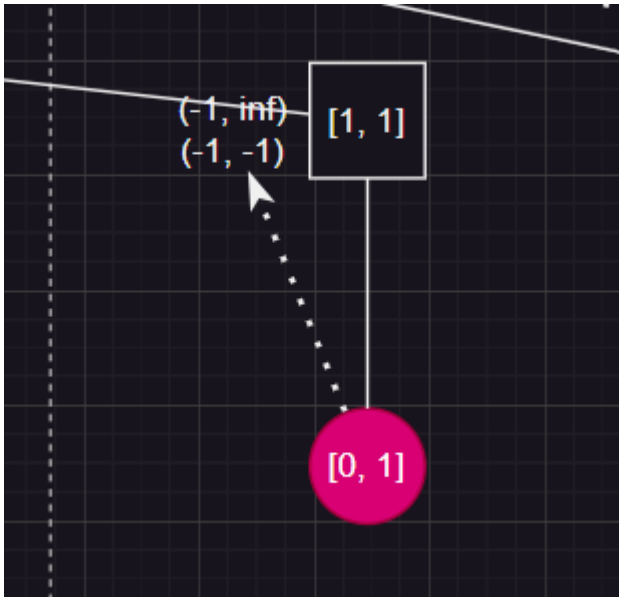
1 stick from pile 2 -> [1,3]



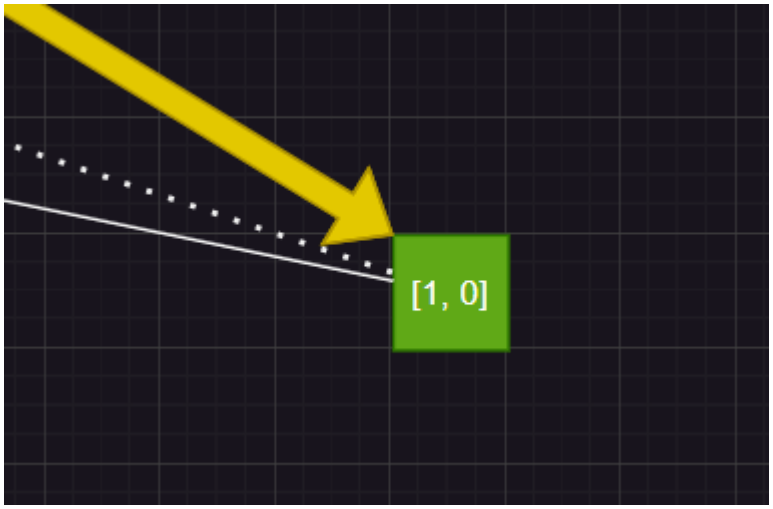
2 sticks from pile 2:



3 sticks from pile 2 -> [1,1]:



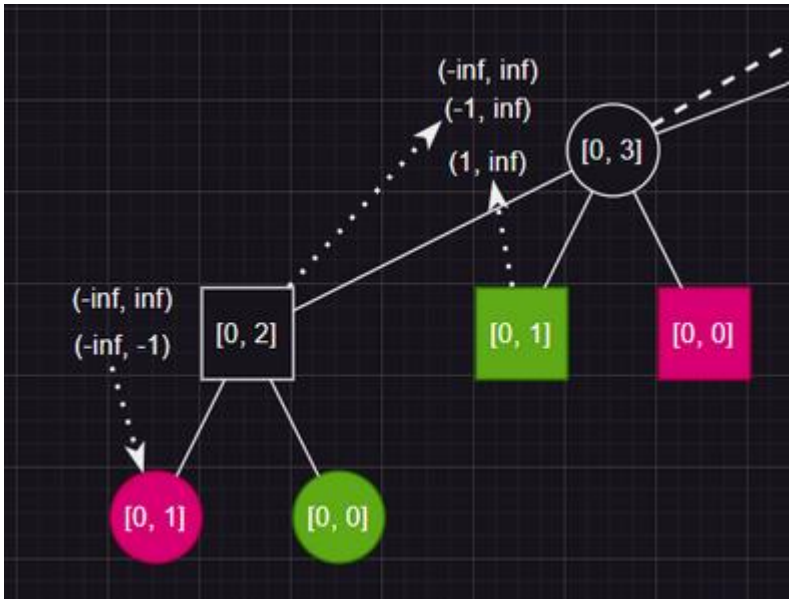
4 sticks from pile 2 -> [1, 0]



If we let the agent end up in situaltion where the board is [4, 1], then it will take 4 sticks out of the second pile, leading to player's failure:

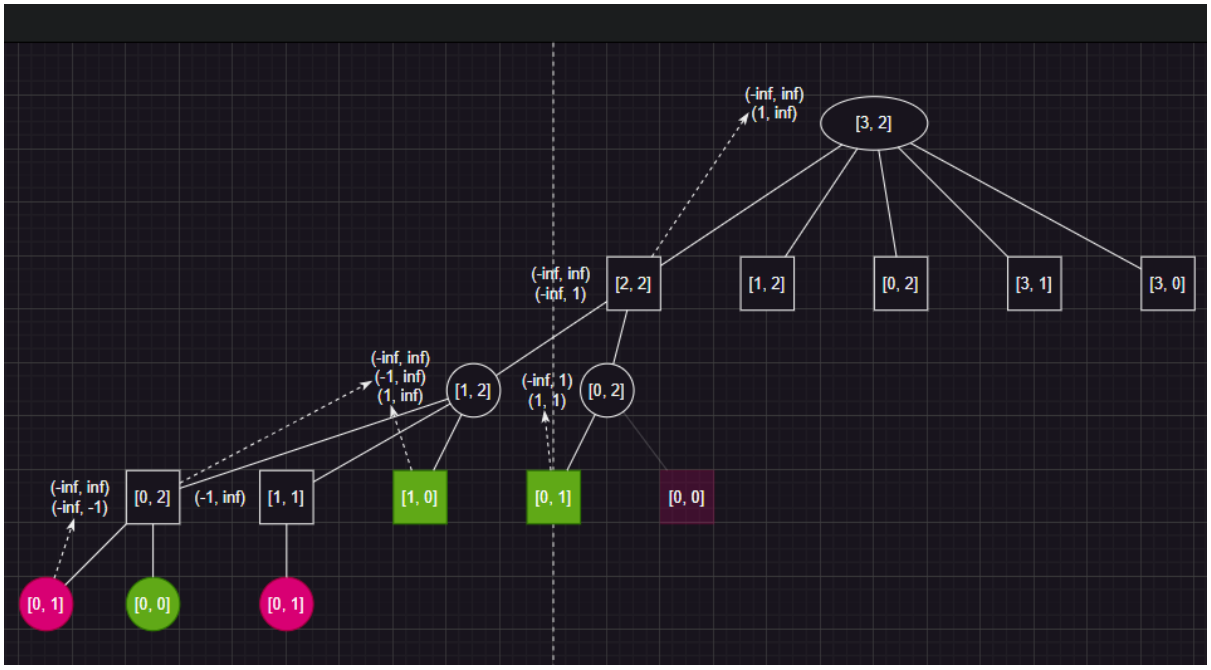
```
Pile state [4, 2]
Your move (<pile_nr> <sticks_amount>): 2 1
Pile state [4, 1]
Agent's move: pile chosen = 1, sticks taken =4
[0, 1]
You lose
```

If agent ends up with a board of [0, 3], he will take 2 sticks from pile 2, exactly how it was calculated in the diagram:




```
Pile state [1, 3]
Your move (<pile_nr> <sticks_amount>): 1 1
Pile state [0, 3]
Agent's move: pile chosen = 2, sticks taken =2
[0, 1]
You lose
```

Agent starting with board [3, 2]:



Player taking out 1 stick while board is [2, 2]:

```
Pile state [3, 3]
Your move (<pile_nr> <sticks_amount>): 1 1
Pile state [2, 3]
Agent's move: pile chosen = 2, sticks taken =1
Pile state [2, 2]
Your move (<pile_nr> <sticks_amount>): 1 1
Pile state [1, 2]
Agent's move: pile chosen = 2, sticks taken =2
[1, 0]
You lose
```

```
Pile state [3, 3]
Your move (<pile_nr> <sticks_amount>): 1 1
Pile state [2, 3]
Agent's move: pile chosen = 2, sticks taken =1
Pile state [2, 2]
Your move (<pile_nr> <sticks_amount>): 1 2
Pile state [0, 2]
Agent's move: pile chosen = 2, sticks taken =1
[0, 1]
You lose
```

Player taking out 2 sticks while board is [2, 2]:

```
Pile state [3, 3]
Your move (<pile_nr> <sticks_amount>): 1 1
Pile state [2, 3]
Agent's move: pile chosen = 2, sticks taken =1
Pile state [2, 2]
Your move (<pile_nr> <sticks_amount>): 1 2
Pile state [0, 2]
Agent's move: pile chosen = 2, sticks taken =1
[0, 1]
You lose
```

```
Pile state [3, 3]
Your move (<pile_nr> <sticks_amount>): 2 1
Pile state [3, 2]
Agent's move: pile chosen = 1, sticks taken =1
Pile state [2, 2]
Your move (<pile_nr> <sticks_amount>): 2 2
Pile state [2, 0]
Agent's move: pile chosen = 1, sticks taken =1
[1, 0]
You lose
```

If player ends up with board of [2, 2], it would be a guaranteed loss

Conclusion

It was an interesting experience to program an agent for playing games. It taught us a lot of interesting concepts:

- Some states, that seem possible to win, are actually unwinnable when other player plays his/hers moves right, for example in NIM if I'll get board of [2, 2] it's actually a game over for me, because no matter what moves I'll make, agent will beat me up. Therefore, these states can be extended as "extended terminal nodes" - losing node in the case of [2, 2]
- Modifying brute-force algorithms with even small changes can yield great results: simple MinMax is an exponentially growing algorithm with a lot of redundant checking. Just adding two variables for smallest and biggest possible value for each node is enough to cut off a lot of redundancy
- Maybe not related to AI, but I (Krzysztof Kotowski) made a lot of mistakes in my code, because I didn't bother to closely check the duplicate of my code I copied: I initially made a simple MinMax function (without alpha-beta) just to check how it works. Then I copied its code and made a minmax with alpha-beta pruning. However, I made a mistake, where the a-b minmax called simple minmax recursively, because I didn't notice, that I left out calls to the simple minmax.

The problem with the algorithm was for complex more extensive boards, as it couldn't go over the entire board. There should be smaller depth chosen for those boards.