Middle East Technical University Department of Computer Engineering

# CENG 462

Artificial Intelligence

Spring 2023-2024

Assignment 3

## Regulations

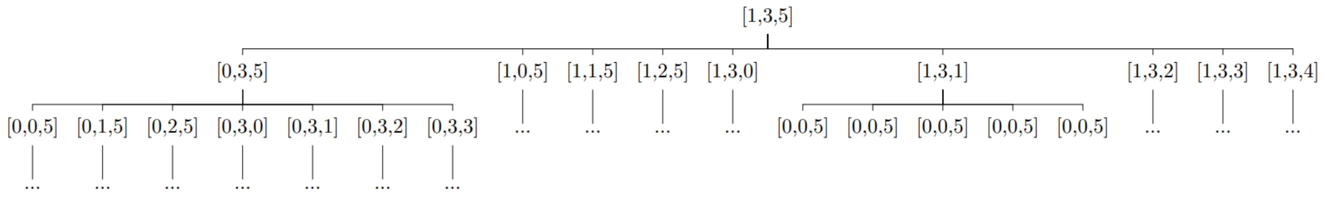
1. The homework is due by **23:55 on April 28th, 2024**. Late submission is not allowed.
2. Submissions are to be made via ODTUClass, do not send your homework via e-mail.
3. Upload your solution as a single **Python** file named *yourStudentId_HW3.py*. **Submissions that violate the naming convention will incur a grade reduction of 10 points.**
4. Send an e-mail to **garipler@metu.edu.tr** if you need to get in contact.
5. **This is an individual homework, which means you have to answer the questions on your own. Any contrary case including but not limited to getting help from automated tools, sharing your answers with each other, extensive collaboration etc. will be considered as cheating and university regulations about cheating will be applied.**

## Question

In this assignment, you are expected to write a **Python** program implementing the Minimax Algorithm and the Alpha-Beta Search method in order to solve the Game of Nim and other generic two-player games that are represented as trees.

Nim is a mathematical strategy game in which two players take turns removing (or "nimming") objects from distinct piles. On each turn, a player must remove at least one object, and is allowed to remove any number of objects provided they all come from the same pile. The goal of the game is to avoid taking the last object. That is, the player that takes the last object loses the game. There is no restriction on the number of piles and sizes of piless (i.e. there can be any number of piles each having any number of objects).

For this assignment, (initial configuration of) a nim game will be represented with a list of integers where each element of the list corresponds to a pile and denotes number of elements in that pile. To illustrate, consider the nim game represented with *[1,3,5]*. The number of piles is 3. The first pile has 1 object, the second has 3 objects, and the third pile has 5 objects. Further, the corresponding game tree is as below:



For the given example game [1,3,5], a possible action sequence may be as such:

- Player 1 chose to start with taking 1 object from the first pile. The game became [0,3,5]
- Then, Player 2 chose the third pile and took 1 object. The game became [0,3,4]
- Then, Player 1 chose the third pile and took 3 objects. The game became [0,3,1]
- Then, Player 2 chose to take 3 objects from the second pile. The game became [0,0,1]
- Then, Player 1 necessarily takes the last object since there is only one object in the third pile and no objects in other piles. That is, Player 1 loses, thus, Player 2 wins.

## Specifications

- Assume the player at the root state is the MAX player; and the terminal nodes are labeled with the utility scores for the MAX player.
- When a board configuration results in MAX's victory its utility value is 1. For a loss (i.e. when min player wins), it gets a utility value of -1.
- Implement your program in Python 3. Do not import any libraries.
- You are expected to implement a function with the following function signature:
  ***def SolveGame(method_name, game)***
  where the parameter ***method_name*** specifies which method to be run for the given problem (will necessarily have one of the values "Minimax" or "AlphaBeta");
  and the parameter ***game*** is the list specifing the initial configuration of the game (e.g. [1,3,5,]).
- While evaluating your submissions, only the function ***SolveGame(method_name, game)*** will be called. Therefore, that function must return the expected result.
- The result expected to be returned is a tuple ***(next_configuration,number_of_nodes)*** where ***next_configuration*** is the list denoting the configuration reached as the result of the optimal first move made by the MAX player, and ***number_of_nodes*** is the number of the visited/processed nodes of the subtree rooted at the optimal next state*.
- While expanding a node (i.e. during children construction) start from the leftmost pile. After covering all the cases for that pile from having 0 objects to having $k-1$ objects (where $k$ is the number of objects in that pile at the current configuration), continue with the next pile in the same fashion. To illustrate, while expanding the example game is $[1,3,5]$, the order of children construction should be as such:
  $[0,3,5]$, $[1,0,5]$, $[1,1,5]$, $[1,2,5]$, $[1,3,0]$, $[1,3,1]$, $[1,3,2]$, $[1,3,3]$, $[1,3,4]$.
- For some input, there can be more than one optimal next configuration. In such a case, any of those will be considered a correct answer.

*When you return the number of visited nodes, you need to return the number of possible moves made at the subtree rooted at the optimal next state. Let's say the game is [1,3,5]. The possible next configurations are [0, 3, 5], [1, 0, 5], [1, 1, 5], [1, 2, 5], [1, 3, 0], [1, 3, 1], [1, 3, 2], [1, 3, 3], [1, 3, 4]. After this step, the algorithms need to go deep for every state. In other words, the state [0, 3, 5] has 1735 processed nodes, the state [1, 0, 5] has 143, the state [1, 3, 2] has 446, and it goes so on... Since the optimal next state is found to be [1,3,2], you will return the 446.

## Sample I/O

- $>>>$ *SolveGame*("*Minimax*", [1, 3, 5])
  ([1 , 3 , 2] , 446)
- $>>>$ *SolveGame*("*AlphaBeta*", [1, 3, 5])
  ([1 , 3 , 2] , 110)
- $>>>$ *SolveGame*("*Minimax*", [1, 2, 4, 5])
  ([1 , 0 , 4, 5] , 41992)
- $>>>$ *SolveGame*("*AlphaBeta*", [1, 2, 4, 5])
  ([1 , 0 , 4, 5] , 2575)