Trabalho 2: Adversarial Games

Marc Clascà Ramírez   Joaquim Ferrer Sagarra

April 2022

1 Introduction

This assignment is based on experiments regarding adversarial games and different approaches to solve them. In this kind of problems each agent needs to consider the actions of other agents and how they affect its own welfare [1, Section 5.1]. Our goal in this kind of problems is search in all possible states the one that scores the best in order to move forward and win the game against one adversarial. To do that we implemented 3 algorithms: MiniMax, Alpha-Beta pruning and Monte Carlo Tree Search. We gave to our program the possibility to play against the user or between them.

The game chosen to perform the assignment is the *Connect-Four Game*. This game is based on a 7 columns and 6 rows matrix in which players can place its token (x or o) in any of the columns. The goal is to reach a line of four equal tokens in any possible direction. More information about the game can be found in section 3.

2 Algorithms

This section aims to describe the algorithms used to solve the game. The main goal of all them is finding the best move to be done starting from the current state of the game. All them use an *evaluation function* who's in charge of evaluating a game state and giving a number that evaluates it. This number can be positive considering player A is winning, 0 considering no player is winning or negative, considering player B is winning. More information about the function used in this assignment can be found in section 3.

MiniMax MiniMax is the simplest strategy to be done in order to solve the game. Its goal is to build a tree taking in account all possible moves of the player, then all possible moves of the opponent, from them all possible moves of the player, then the opponent and so one. When arriving to a certain depth or to a terminal state (game finished or no more possible moves) the algorithm gets the evaluation of the movement and returns it to the parent in the tree. Doing that, the algorithm chooses, for every child from a given state, the one that gives a best evaluation to the player taking in account the opponents best move. So, for a given state, MiniMax will choose the best movement to do (the one that gives the best evaluation) taking in account that the opponent will choose after the also the best movement he can play (also the one with the best evaluation). [1, Section 5.2]

Figure 1 illustrates how MiniMax tree works. In this example, it is the red player turn which wants to maximize its score (so a positive value of the evaluation function means that blue is winning). So it builds the tree to a certain depth and starts looking at the evaluation scores of the movements to consider which is the best play taking in account the best play that the opponent may do.

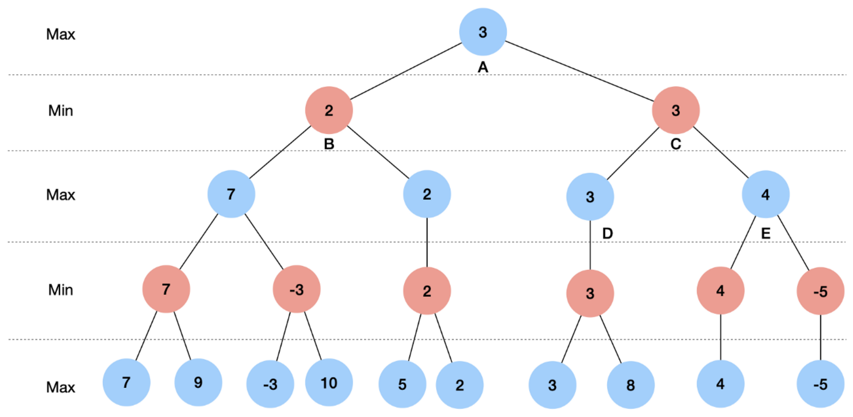

Figure 1: MiniMax tree. Source: https://towardsdatascience.com/

Alpha-Beta Pruning Alpha-beta Pruning is an optimization for the MiniMax algorithm. Its goal is to find branches in the tree that are not needed to visit. For example, taking in account the tree shown in Figure 1, we can easily realize that when visiting the last level of the tree after finding a number -3 was not reason to keep searching in the other tree leaves. That's because -3 is lower than 7. The node above -3 will choose always the minimum which will be -3 or lower and upper node will choose the maximum which will be for sure the 7 against the -3 or lower gotten from the other branch. The way to detect these cases is keeping track of the highest and the lowest number found in a certain path with the parameters named α and β. If the flow is in a maximizing node and finds one child with higher score than the minimum or in a minimizing node and finds one child with lower score than the maximum is not needed to keep exploring the other children, this path will not be the one chosen by the algorithm. [1, Section 5.3]

This algorithm can be even more optimized reordering the way in which the nodes are visited. If the flow is in a maximizing node is important to visit first the children that have the highest score and the opposite for the minimizing nodes. In this way is easier to prune branches and so creating and visiting less nodes in the tree.

Monte Carlo Tree Search The Monte Carlo Tree Search (MCTS) strategy aims to overcome the problems that alpha-beta encounters with games that have large branching factors or that heuristics are not very useful. In the basic MCTS strategy, the value of a state "is estimated as the average utility over a number of simulations of complete games starting from the state. A simulation (also called a playout or rollout) chooses moves first for one player, then for the other, repeating until a terminal position is reached." [1]

3 Connect Four

Connect Four Game is a classic game in which two players aim to connect a line of four tokens in a 6x7 matrix. In every turn, each player chooses a column to place its token. The token falls to the first free row in the desired column.

In each round, there's 7 possible moves regarding the 7 columns in which a token can be placed. The algorithms mentioned in section 2 have been used to select the best column to place the token (x or o) evaluating the possibilities using an *evaluation function.* This evaluation function gives a number positive or negative depending in which player is winning the game. To compute the evaluation from a given state, all segments of four tokens in all directions (horizontal, vertical and both diagonals) have to be taken in account. The evaluation is computed summing the score of all these segments obtained using the values of Table 1. A win of x scores 512 and a win of o -512 despite the segments score.

-50	3 'o', no 'x'
-10	2 'o', no 'x'
-1	1 'o', no 'x'
0	'o' and 'x' mixed
1	1 'x', no 'o'
10	2 'x', no 'o'
50	3 'x', no 'o'

Table 1: Score for segments in Connect Four Game

4 Algorithms applied to Connect Four

This section will give a description about how the algorithms have been implemented, results and performance analysis for each of them.

Evaluation Function The evaluation function for all algorithms had been the same which is the one described in section 3.

Programming Language and Runtime We chose C++ as the programming language for our program. We think it is a good choice because it is a compiled language, which gives better performance compared to interpreted languages (like Python) or languages that run over virtual machines (like Java). C++ allows compiling the source code to a binary that makes use of the features of the processor, like vector instructions, allowing fast computing of processes like large math operations. For Artificial Intelligence applications it is very important to be able to fine tune the program and the code, to use only the instructions needed and get a good time efficiency. In terms of spatial efficiency, C++ also allows to control better the memory usage and create data structures with the exact size needed. This is important when a large number of the same data structure is created and needed to work, like with nodes in this case. C++ allows fine tuning the allocation of dynamic memory, with the drawback of a more complicated programming, compared to other languages like Python.

Data Structures The main data structures involved in problem solving are the following:

- *Tabuleiro* This class contains a specific state of the table. Because it will be created to represent the different states during the game. It contains an array of 6x7 bytes, to represent all positions in the table and the token placed in each place. This class contains methods to make moves, consult the utility, generate all possible new movements and print the table in different formats (list of numbers or formatted as a matrix).

- *Player* This class is an abstract class used to instantiate all different players. It keeps the assigned token to the player and a function to make a new move which will perform depending the algorithm choosen.

Code Structure Each game is represented with the class `jogo`. It contains the two players instantiated and a function to make them play a round. In every round, both algorithms play their turn. Each player in each round starts from the current state of the game and returns a number regarding the position in which to move. Then tabuleiro is changed according this position by placing the correct token in the desired position. This sequence is being repeated untill the end of the game, when a player wins or when there is no more space for placing new tokens.

MiniMax Implementation Algorithm MiniMax is implemented with its own class made from class `Player`. It starts with a wrapper function that performs the first level of the MiniMax tree keeping the best movement. Then, for every child a recursive MiniMax is performed returning the values of the evaluation of each possible path. All possible children from a given state are stored in a vector and iterated in order to explore the tree using a DFS strategy. A limit is imposed and it can be changed through the configuration file.

Alpha Beta Purning Our implementation of Alfa Beta Pruning is quite similar to the MiniMax with some changes to improve its performance. The first difference is while implementing the pruning itself. The recursion has two parameters α and β keeping respectively the highest and the lowest value found in the path to perform the pruning as described in section 2. To improve its performance, the children nodes of a given state are kept in a *multimap* structure indexed by the evaluation of the state and ordered. With this structure we can order the children nodes ascending or descending to improve performance having the highest value first when maximizing and the lowest one first when minimizing.

Monte Carlo Tree Search Our implementation of the MCTS algorithm consists of two main elements: first, the player class, which contains the structure of the MCTS algorithm, a function that preforms iterations on the same state, and a function that performs the 4 differentiated phases of the algorithm. Then we have the `TreeNode` class, which is a custom tree structure to work with the MCTS strategy, using a UCB1 policy for selecting nodes. It contains all the parameters needed to traverse the tree, store the value of each move, and control the children that has (or not) been expanded.

4.1 Results

To test the algorithms and its performance we analyzed how they were performing in a game having means of time used in each turn and number of nodes used.

To test MiniMax performance in front of Alpha Beta we made them play together using different deeps. Table 2 shows the scalability the algorithms had when playing against each other. When using a depth of 6, MiniMax was not able to end while Alpha Beta performed quite good. The data of this table has been ploted in Figure 2 and Figure 3 in which we can see clearly the different way to scale the resources needed using each algorithm.

	Depth	Time per turn (ms)	Nodes per turn
MiniMax	2	1.23882	45
Alpha Beta	2	1.11335	42
MiniMax	3	8.81459	518
Alpha Beta	3	6.70318	213
MiniMax	4	119.105	7799
Alpha Beta	4	44.9473	1686
MiniMax	5	836.713	51878
Alpha Beta	5	223.231	5810
MiniMax	6	–	–
Alpha Beta	6	493.205	15776

Table 2: Comparison between MiniMax and Alpha beta

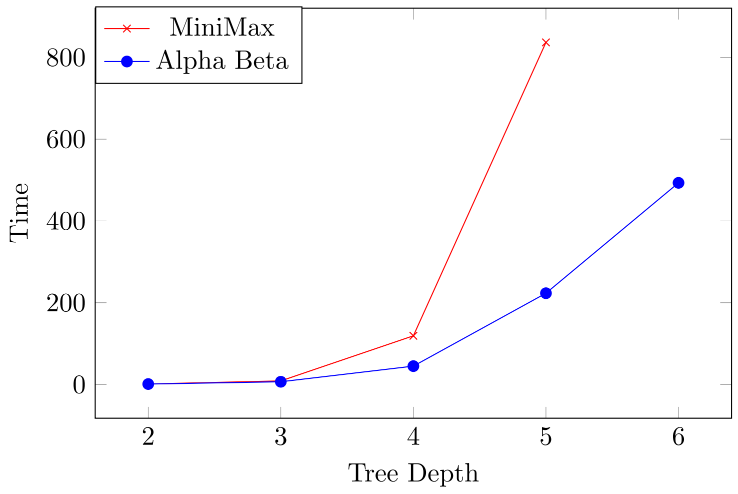

Figure 2: Execution time of MiniMax and Alpha Beta in different tree depths

References

[1] Russell, S. J., Norvig, P. (1995). Artificial intelligence: A modern approach. Englewood Cliffs, N.J: Prentice Hall.

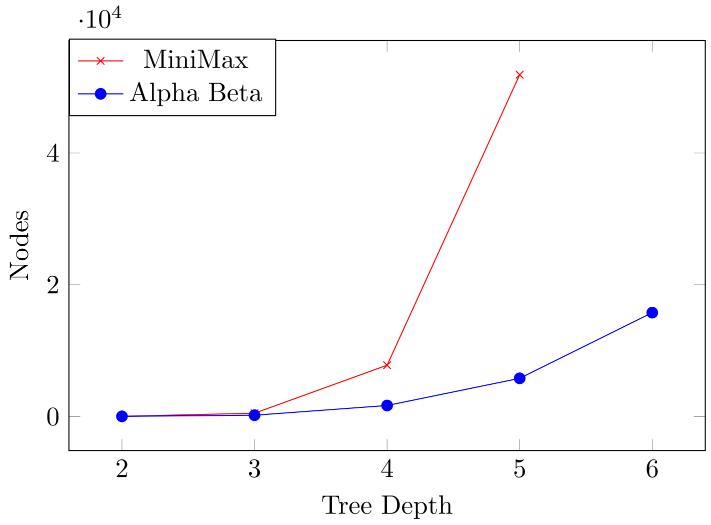

Figure 3: Nodes used by MiniMax and Alpha Beta in different tree depths