COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS

Introduction to Artificial Intelligence
Exercise 6 - MiniMax

October 26, 2022

6. MINIMAX AND ALPHA-BETA PRUNING

The task is to program an agent that will play the TicTacToe game (and Gomoku), using MiniMax algorithm and Alpha-Beta pruning.

Program:

Program (and the description in this assignment) is copied from the first exercise. In the file *games.py* is defined a class *Game* and from that derived *TicTacToe* and *Gomoku*, which you do not need to modify. You will only work with *players_minimax.py*. Look for *YOUR CODE GOES HERE* and/or *EXAMPLES*.

Your job is to finish the function **choose_move(game, state)** in classes *MinimaxPlayer* and *AlphaBetaPlayer* to use MiniMax and Alpha-Beta pruning. These functions get two arguments:

- **game** - instance of class **Game** in which are already implemented all the functions regarding the game itself (see below)
- **state** - actual state of the game (state of the board, player on move, ...), on which we want to respond with the best possible action (best move). You don't need to parse the state, only use it as an argument into already prepared functions (see below).

Output of the function is an action (a move, which is to be played) – an integer from list of possible actions **game.actions(state)** for a given state **state**.

A few functions of class **Game**, which can be helpful:

- **game.player_at_turn(state)** – returns, which player is currently (in state *state*) on move ('X' or 'O')
- **game.other_player(player)** – as an argument gets a letter of a player and returns the letter of the other player
- **game.board_in_state(state)** – returns the state of the board *(list(list(char)))* in the state *state*
- **game.w** – width of the board
- **game.h** – height of the board

- **game.k** – the amount of characters in a row a player needs to have in order to win the game
- **game.actions(state)** – returns a list *(list(int))* of all possible actions (valid actions) from state *state*
- **game.state_after_move(state, a)** – returns new state, which arises from performing action *a* from state *state*
- **game.is_terminal(state)** – tests, if the state *state* is terminal (if it is a win/loss situation or a draw)
- **game.utility(state, player)** – returns the 'utility' of state *state* for player *player* : 1 if won, -1 if lost, 0 otherwise. Keep in mind, that the individual utilities are known only at the end of the game. There is no sense in asking for utility in state *state*, until it is not terminal.

Simple examples of these functions are included in codes.

At the end of the code the games are launched – uncomment whatever you will need. When debugging we recommend playing MiniMax vs human.
As for the sainty check, TicTacToe is an instance of a so-called m, n, k game with a lot of theoretical reserach. For us, it is important to know that in case of 3x3 board, an optimal player can not lose. So, if your agent is programmed correctly, it always wins or draws, regardles whether he goes first or second. If he plays against another optimal player (MiniMax vs. MiniMax), all games should end in a draw.

Assignment 1 (1p): Implement MiniMax algorithm in the MinimaxPlayer class. Pseudocode can be found for example in slides from lecture or HERE. Use recursion!
Assignment 2 (0.4p): As you can see, MiniMax is quite slow, especially if it goes first. The solution is to prune (i. e. not search) those branches that can not improve the solution - so-called Alpha-Beta pruning. Implement MiniMax with Alpha-Beta pruning in class AlphaBetaPlayer. (AIMA pseudocode)
Assignment 3 (0.6p): Standard 3x3 TicTacToe is a simple problem - the entire search space has only $3^9 = 19683$ states, so we can search the entire tree all the way to the leafs. Alternative version with 15x15 board - Gomoku - has $3^{255} \approx 4.6 \times 10^{121}$ states. In this case we cannot search the entire tree, we have to stop at certain depth. That state won't be terminal, so we will not know its real utility, so we need to estimate it using an evaluation function. It should evaluate a state based on how "promising" it looks. There is no perfect evaluation function - you have to come up with yours (you can even use your code from the first exercise). You don't need to find the best function, just use something sensible.
Implement tree pruning (at some fixed depth) and evaluation function in class AlphaBetaEvalPlayer. This agent should be able to play Gomoku reasonably well.