Practical 2: Adversarial Games

If practical 1 versed on the search algorithms on a one-sided variant of chess game, typical chess is possibly one of a kind of adversarial games, in which two opponents confront each other's ability at finding the smartest strategy to check mate each other.

The goal of this practical is to familiarize yourself with strategies to play adversarial games, with the basic search algorithms of AI. To this end, your task will consist of implementing several adversarial game search algorithms to attain specific goals on a chess board.

The Simulator
As for practical 1, you will use the very same simulator you used for the previous practical, consisting of four hierarchical classes: aichess, chess, board, piece. The latter three implement the dynamics of a chess game, which may be run by two players. Please, run the main on the class Chess to this end. Finally, the Aichess class is a capsule of the former three, with the purpose of implementing AI algorithms to analyse and alter the dynamics of the chess game.

A few **notes of interest**:

- The Chess class incorporates two instances of the Board class: Board and BoardSim. The idea is that of clean-cut separating a board in which you may simulate your moves, and another one in which you may make your effective moves towards the "target" whenever you are certain of what the correct strategy is.
- In brief, you must use these two boards to simulate your moves, effectively moving the pieces, as a means to synchronize the aichess exploration of the state of the board.
- Unlike in the case of search algorithms, adversarial games, both white and black pieces may move ahead. Therefore, to answer the questions described next you will have to

The Definition of the State and the Initial State

- The definition of the state ought to include the positions and piece types. Although you may define two independent black/white states, the problem state includes both.
- Each team moves in alternation. You will have to implement an algorithm to move whites/blacks as requested.

Your Work
As customary in the AI practical, during this game you are provided with a specific starting board configuration (pre-programmed): it contains a black king, in position [0,5], and a black rook in position [0,0], and two white pieces, the movement of which you will have to program: a king in position [7,5] and a rook in position [7,0]. Your goal will vary from question to question, but keep in mind that, in chess, whites move first.

You are instructed to find that state by implementing algorithms described next. In all cases, please do provide a list of the visited states from the origin to the target and the minimal depth necessary to reach the target.

1. Whites start moving. Implement the dynamics of a game in which both, whites and blacks, follow the same Minimax algorithm to try to check-mate each other. Assume that both implement minimax with a depth of 4 moves (2p)
    a. Once implemented, run the same game 20 times. How many times do whites win?
    b. Why is that?

2. Now run the same simulations, but varying the depth of the minimax algorithm from 1 to 5 moves both for whites and blacks. Run each possible combination of depths 10 times (2p)
    a. Plot the percentage of white wins over the total for each depth value.
    b. Is the result symmetric. Why is that?

3. Implement the alfa-beta pruning for the blacks only, whites still play with minimax (2p).
    a. Using an equal depth of 4, run 10 simulations. Who wins the most?
    b. Justify your result.

4. Both whites and blacks use the same alfa-beta pruning. Run ten simulations each while varying the depth with which each team plays (1-5) (2p)
    a. Plot the proportion of wins for whites and blacks.
    b. Comment on the result.

5. Implement the expectimax algorithm for whites and blacks. (2p)
    a. Whites play expectimax, blacks alfa-beta pruning.
        i. Run 10 simulations each and plot the proportion of wins for whites/blacks.
        ii. Who wins the most. Why is that?
    b. Now whites play alfa-beta pruning, blacks expectimax.
        i. Run again 10 simulations each and plot the proportion of wins for whites/blacks.
        ii. Who wins the most. Why is that?
    c. If there are differences between a. and b., please, do comment why.

6. The situation generated by confronting a white king and a black king plus a rook each may be considered even (3p)
    a. Is it really the case? Justify your answer.
    b. In your opinion, what makes this situation of particular interest for the study of adversarial games?

Documentation

The python code provided is documented, and provides a straightforward structure for you to study and analyse. Although you do not have to understand every single detail, you need to build sufficient intuition about the structure of the underlying classes to be able to solve the problems listed next.

In this practical we will view chess as a game of search (of the check-mate board position starting from an origin position), which you will have to find out with your search algorithms. To facilitate your implementation work, the AIClass has several variables of interest, listed next:

- CurrentState. The list of positions and pieces of the White pieces on the board.
- ListNextStates. The list of states that hang from the current state in the tree.
- ListVisitedStates. The list of already visited states in the tree.
- PathToTarget. The sequence – python list of states from the origin to the target state yielded by your algorithm.
- Depth. The necessary depth to reach that the target (check-mate) state.

It is fundamental that you implement the **state** as a list of elements containing the positions and types of pieces you move. Each position has a numeric identifier you may find in the initialization of the board class. For example, the initial position could be: [[7,0,2],[7,4,6]]. This indicates that a white rook is in board position [7,0] and the king on position [7,6]. As you explore the decision tree, you will have to modify this state.

Finally, the Aiclass already has an implementation of the getListNextStateW(*thisState*) function, which returns the list of states you may potentially reach from *thisState.*