Difficulty Implementation Document

Our AI's algorithm is relatively simple. First, at the begging of the single-player game, a decision tree is generated. Each Node of the tree has a Board-object that stores a possible combination of pieces on the board. The root of the tree is a Node with an empty board, and every single possibility of how the game can go on is somewhere down the branches of the tree.

When prompted to make a move, our AI is given a Board-object, that represents the current board state of the game. First task is to find a Board-object on the decision tree that has the same board. Then the AI performs a minimax algorithm on each of its children, and as the next move it chooses the child-board with the highest (or lowest) score.

Our minimax algorithm is a recursive search for up to d levels down the decision tree, where d is a passed variable. The depth of the search is strictly connected to number of moves AI sees ahead, so we always passed in 9-we always wanted our AI to exhaust the search. At the end of the search a StaticEvaluator's object evaluate() is invoked. Evaluate() function plays a key role in the efficiency of our AI. If implemented correctly, it should return 0 for a drawn position, positive number for an X-winning position, and a negative number for an O-winning position. That way, our AI determines which path down the decision tree the game will go on, given an optimal play from both players.

The way we toggle our difficulty is to simply switch the StaticEvaluator object that is responsible for board evaluation when AI is prompted for a move. If we give our AI a good evaluator, that determines the winning, losing, and drawn position correctly (AdvancedEvaluator), our AI will always play optimally. On the other hand, if we give it a bad one, that evaluates any board randomly (RandomEvaluator), our AI will also be playing random moves.