Exercise: Searching Game Trees

In this exercise you will implement one method and prove the correctness of several others.

The program that the exercise focuses on plays a simple game called Connect Four. The following description of the game is from Wikipedia (http://en.wikipedia.org/wiki/Connect_Four). The game is played on a board with 7 columns and 6 rows, which is placed in a vertical position. The players have 21 discs each, distinguished by color (say black and white; White plays first). The players take turns in dropping discs in one of the non-full columns. The disc then occupies the lowest unoccupied square on that column. A player wins by placing four of their own discs consecutively in a line (row, column or diagonal), which ends the game. The game ends in a draw if the board is filled completely without any player winning.

The game itself is represented by two classes, `ConnectFourBoard` and `ConnectFourMove`. The second class represents a move in the game. In this game, a move is characterized by the column into which the disc is dropped, so this class is very simple: it represents an integer that we understand to be a column index.

The first class, `ConnectFourBoard`, represents the state of the game. It provides several methods:

- `move` is a query that returns a new game state (an object of the same class), the state that the current game would be in after a given move by one of the players. The move and the identity of the player are given as arguments to the method.
- `legalMoves` returns all the legal moves in the state of the game represented by the current object.
- `isGameOver` is a query that returns a value that indicates the state of the game. A score of zero means that the white won. A score of 1 means that the black won. A score of 1/2 means that the game ended in a draw (the game ended with no winner). A score of −1 means that the game has not ended yet.
- `score` is a query that returns a `double` value between 0 and 1 (exclusive!) that indicates how bad the current game state is to the white player. Values near 0 should indicate that the White's position is better than the Black's, and values near 1 the opposite. Scores should be monotonic: the worse the situation for White, the higher the score should be. Informally, think about them as an estimate for the probability that White will win.

To allow the computer to play against a human, the program includes two other classes. They both perform the same function, but one is less efficient but easier to

understand. Given a board situation, object from these classes search the space of possible continuations of the game to find the optimal move for the player whose turn it is. Since searching the space of possibilities until the game ends can be too expensive, these objects treat game situations that are `depth` moves away (for a given parameter `depth`) as terminations of the game. If the search algorithm reaches a game situation that is `depth` moves away, it computes the score of the situation, and uses that as a measure of who well each player does. The objective of the search algorithm is to find a move that minimizes the score that White can achieve if it's her turn and maximize the score that Black can achieve if it's her turn. The algorithm assumes that both players play optimally. For example, if a white move can lead to a white win if Black makes a mistake but to a white loss if Black plays well, this is considered a losing move for white; White assumes that Black makes no mistakes.

Under this rules, we can view the game as a tree, in which nodes represent game positions and edges represent possible moves. The root is the empty board, the initial state of the game. The edges from the root to its children represent all the possible White moves (7 in this game, unless some of the columns are full). Edges from children of the root to their children represent Black moves, and so on. Leaves of the tree are game-end positions. Our search algorithm only considers sub-trees rooted at the current board and extending `depth` moves down. We can label such a sub-tree with scores as follows

- Leaves of the subtree are labeled with their score, as computed by `ConnectFourBoard.score()`, or by 1 and 0 if the leaf represents a game-end situation.
- The score of non-leaves in which it is White's turn to play is the minimum over the score of the node's children; White plays to minimize the score. This is called the min-rule.
- The score of non-leaves in which it is Black's turn to play is the maximum over the score of the children. This is the max-rule.

Such trees are called mini-max trees or game-search trees. The program plays by computing the scores of the children of the current game situation and selecting the move to the min child if the program plays the White (or the move to the max child if the program plays black).

The simpler game-searching class, `ExhaustiveGameSearch`, finds the optimal move by recursively expanding the sub-tree it needs to search and using the min-max rules directly. The other class, `AlphaBetaGameSearch`, uses a cleverer algorithm.

Complete the following tasks:

1. Implement `ConnectFourBoard.isGameOver()` so that it satisfies its contract. You can add fields to the class if you need to.
2. Implement `ConnectFourBoard.score()` so that it satisfies its contract. You can start with a very simple implementation that always returns 0.5 or a random value. This will allow you to test the program (and in particular, to test your solution for the previous part of the exercise). But eventually, your implementation must return a range of values that reflects at least how close

each player is to connecting four disks. You can implement a better score function (for example, a method that also counts how many opportunities to connect four each player has, etc.), but it will not contribute to your grade (but it can be fun, especially if the program beats you at the game). It is not hard to write a score method that will make the program a strong player that is hard to beat.

3. Prove that `ConnectFourBoard.move()` is correct (satisfies its contract).
4. Prove that `ExhaustiveGameSearch.optimalMove` is correct. You will need to analyze its helper method, of course.
5. Prove that `AlphaBetaGameSearch.optimalMove` is correct. This is hard. We will give this part of the program only 1/10 of the grade.

Write proofs in comments directly above the methods whose correctness you prove. Do not attach extra files or hand-written proofs on paper.

There are many possible improvements to this program, and trying some of them is fun. For example, you can try to modify the search algorithm so that the program plays the shortest path to winning once it finds a winning strategy. Ordering the children of a node by score can improve the search efficiency of `AlphaBetaGameSearch`, which will allow you to search deeper without spending much more time.