CSCI–4150 Introduction to Artificial Intelligence, Fall 2005
Assignment 4 details (184 points)

Notes on this assignment

- There will be several support code files for this assignment. To start, you will need the files: `connect4.scm` and `a4code.com`. In addition, there will be a file `assign4.scm` that contains stubs for the procedures you must write.
- There will be two web-testers for this assignment:
  - The two web-testers for Problems 2 and 3 will be a regular web tester, as for previous assignments, with the standard electronic submission policy.
  - We will also set up a "pretournament system" so that you can run Connect 4 matches with other students' evaluation functions. You may upload a new evaluation function as many times as you want to this web-tester.

    There will be a CPU time limit for each player in these games which will determine the depth to which your evaluation function is run. See the Assignment 4 information web page for details.
- It is not appropriate for a student to give source code for his or her evaluation function to another student for purposes of playing a game between their evaluation functions. You will be able to play games against other students' evaluation functions through the pretournament system. In addition, there will be some compiled evaluation functions included in the support code.

Problems

Due October 6 (no late papers; solutions will be posted after class)

1. (24 points) Do the alpha-beta MINIMAX problem on the separate handout.

Due October 20

2. (36 points) Write the procedure `(create-c4-player eval-fn depth-cutoff)` in which you will implement the alpha-beta MINIMAX search.
3. (8 points) Write a "minimally competent" evaluation function (and opening database, if you wish). By "minimally competent", I mean that your evaluation function must beat one of our reference players. The source code for this reference player will be available to you. See Section A.2
4. (16 points) Do exercise 6.5 from the text [correctness of alpha-beta pruning]
5. (16 points) The following problem is based on Exercise 6.15 from our text.

   The most famous nonzero-sum game is the "Prisoner's Dilemma". (If you are not familiar with this game, see page 633 in Section 17.6.) In nonzero-sum games, each player has a separate, independent evaluation function (also called a "payoff" or "utility" function). In short, these are games for which "what is good for one player is bad for the other" does not necessarily hold, so it may be advantageous for players to cooperate.

   (a) In the prisoner's dilemma, the two players choose their action simultaneously, i.e., without knowledge of what the other player will do. Is it possible to have a nonzero-sum game with sequential moves (i.e., a player knows all previous moves when deciding upon the current move) where it is in both players' interest to cooperate? Justify your answer.

   (b) Suppose we have a two-player nonzero-sum game where players alternate turns making a move. Is it possible to modify alpha-beta MINIMAX search to determine the best move for a player more efficiently? If so, describe and explain the modified algorithm. If not, explain why.

Due October 27

6. (20 points) Final evaluation function due. No late evaluation functions accepted — we will take your most recent evaluation function in the "pretournament system" as of noon. Your score on this problem will be determined by your evaluation function's performance in a round-robin tournament and by how it fares against our "reference players."

7. (64 points) Writeup on your evaluation function. Details and guidelines for this writeup will be posted on the assignment web page. This will include a prose description of your evaluation function and some analysis of its performance.

A. Connect 4

The Connect 4 game is played on a "board" 6 rows high and 7 columns wide. Players alternately drop a piece from the top of any column, and it will fall to the lowest open row. At most 6 pieces can be in any column. Traditionally, one player is red and the other black, but for purposes of a text representation, we will use the letters "X" and "O" instead. Player "X" will always play first. The object is to get four pieces in a row, either horizontally, vertically, or diagonally.

The support code contains procedures that implement the Connect 4 game — printing, accessing, and manipulating boards, getting the children of a board state, running a game, and so on. Your tasks will be to implement the alpha-beta pruning algorithm and to write an evaluation function for the Connect 4 game.

You will need to read through much of the `connect4.scm` file. There is detailed documentation on much of the support code, some of which is repeated here. There is a section in this file on how to get started on this problem. One of the first things it suggests is to try playing against a random player:

```
(load "a4code")
(load "connect4")
(play-c4 random-player human-player)
```

A.1 Player procedures

In order to separate the evaluation function from the alpha-beta MINIMAX search and to independently control the search depth, you will write a procedure that returns a "player procedure." This procedure (for Problem 2) must be of the following form:

```
(create-c4-player eval-fn depth-cutoff)
```

where:

- `(eval-fn board current-player max-player)` returns a number corresponding to the quality of the game state for MAX. The argument `board` is a Connect 4 board state, and `current-player` and `max-player` are both symbols (either `X` or `O`) to indicate which pieces a player is playing.

  Although the arguments may seem redundant, it avoids confusion and may actually be necessary for more sophisticated evaluation functions. To be concrete, if MAX is playing `X`, evaluating a leaf node at at maximizing level is done by the call: `(eval-fn state 'X 'X)`. If a leaf node at a minimizing level is evaluated, the call is: `(eval-fn state 'O 'X)`.

- `depth-cutoff` is the depth at which the game tree will be cut off; leaf nodes of the tree will be at this depth (unless the game ends naturally at a shallower depth)

This problem is where you will implement alpha-beta MINIMAX search that will be embedded within a "player procedure" that it returns.

A player procedure is called with a board state and information about which player is up; it must run alpha-beta MINIMAX with the given evaluation function and depth cutoff and then returns a move. A player procedure must be of the form:

```
(player-fn board player-symbol)
```

where `board` is the current Connect 4 board state and `player` is the piece to be played, either (the symbol) `X` or `O`. This procedure must return a valid move, i.e. an integer between 1 and 7 inclusive.

Important note: your alpha-beta MINIMAX implementation must evaluate the children in the order given by the `c4-children` procedure, otherwise your code will not test properly. Also, your `create-c4-player` procedure must be able to work with any evaluation function, so you cannot hardcode any numerical values!

After you have written this procedure and your evaluation function, you can make the following call in order to play against it:

```
(play-c4 human-player (create-c4-player c4-eval 4))
```

The player procedure here uses the `c4-eval` function in its alpha-beta MINIMAX search up to a depth of 4.

A.1.1 Support code

There are a number of useful procedures in the `a4code.com` file:

- There is an example `create-c4-player` procedure given in the `assign4.scm` file for this problem. This procedure simply serves as a "front end" for the alpha-beta MINIMAX search that you will implement. See the file for details.
- `(other-player player-symbol)` — given an `'X` or an `'O`, it returns the other symbol
- There are a number of comparison procedures that work with numbers extended to include the symbols `pos-infinity` and `neg-infinity`:

  ```
  (inf:max a b)     (inf:> a b)     (inf:>= a b)     (inf:= a b)
  (inf:min a b)     (inf:< a b)     (inf:<= a b)
  ```

  For example:

  ```
  (inf:max 7 'neg-infinity)   ==> 7
  (inf:< 'pos-infinity 200)    ==> #f
  ```

A.2 Evaluation functions

For this assignment, you must create an original evaluation function for the game of Connect 4. This procedure must be called `c4-eval` and must be of the form:

```
(c4-eval board current-player max-player)
```

The arguments and return value of this procedure are described in Section A.1 under the `eval-fn` argument. Let me emphasize that your evaluation function must return a number.

A.3 Opening databases

You will be able to use an opening database for upto 8 moves (i.e., four moves for each player). In order to specify your opening database, define the variables `X-opening-db` and `O-opening-db`. These variables must be sorted lists of strings. The game state will contain a history of the moves made; assume this is represented as a string of digits. To look up move n from your opening database, the length $n-1$ move-history string is matched against the first $n-1$ characters of length n strings. The n^{th} character of the matching string is taken as the desired move.

For example, suppose the move history is: X played first in column 4, O played in column 3, and X played in column 7. For move 4, player O is to make a move, so we search all length 4 strings to find the one that whose first three characters are `"437"`. If it matched the string `"4372"`, then player O's move will be in column 2.

The strings in `X-opening-db` and `O-opening-db`should be sorted. Here is a simple example:

```
(define X-opening-db
  '("4" ; X's first move should be in column 4
    ; specify some of X's second moves
    "412" "425" "432" "443" "455"))

(define O-opening-db
  '(; if X doesn't play 2 on the first move, then O should take it
    "12" "23" "32" "42" "52" "62" "72"
    ; some of O's second moves
    "5231" "5245" "5252" "5267" "5276" "7223" "7233" "7246"))
```

Note that `X-opening-db` consists of only odd-length strings while `O-opening-db` contains only even-length strings.

Your opening database does not have to specify all possible combinations of moves. If a move is not found, then alpha-beta MINIMAX will be run to determine a move.

Note that this representation is redundant: the move histories `"542"` and `"245"` result in the same board state, so your opening database would presumably specify the same next move for player O for both. If you wish, you may "order" the state histories in your opening database by sorting each player's moves. Player X's and player O's moves must still alternate in the string, however. For example, the opening database entry `"4215413"` would be transformed into `"1142453"` because player X's moves (4, 1, and 4) and player O's moves (2, 5, and 1) appear in sorted order. If you choose to order the state histories, you can define the variable `ordered-state-histories` to `#t`, and the opening database will be searched more efficiently.