CSCI 4150 Introduction to Artificial Intelligence, Fall 2002
Assignment 4 (164 points): out Friday September 27, various due dates (see below)

Introduction

This assignment is on game playing. In the first half, you will implement the MINIMAX search and the "Coin-strip" game. In the second half, you will implement the MINIMAX search with alpha-beta pruning and use this with an evaluation function of your own design in order to play the Connect 4 game.

Problems 1–4 and 6 are ordinary problems: do the problem and then turn it in by the deadline. Problem 5, your Connect 4 evaluation function, is different. You will need to turn in a working evaluation function (meaning that it must be able to play an entire game) by the deadline. You will then have an additional week to improve your evaluation function. We will run a Connect 4 tournament with all students' final evaluation functions. Part of your score for Problem 5 will be based on how your entry performs in this tournament.

Notes on this assignment

- This is a large assignment, so there are three deadlines:
  - Friday October 4:
    - Problem 1: the MINIMAX implementation
    - Problem 3: the written alpha-beta pruning problem
  - Friday October 11:
    - Problem 2: Coin-strip game implementation
    - Problem 4: alpha-beta MINIMAX implementation for Connect 4
    - Problem 5: a working evaluation function for Connect 4
  - Friday October 18:
    - Problem 5: your final evaluation function for Connect 4
    - Problem 6: a written description and analysis
- There will be four files for this assignment:
  - `a4code.com` — compiled support code for Nim, playing games with MINIMAX and some support code for Connect 4
  - `connect4.scm` — support source code for Connect 4
  - `assign4-12.scm` and `assign4-45.scm` — files containing stubs for your procedures.
- There will be four web testers for this assignment:
  - The webtesters for Problems 1, 2, and 4 will be regular web testers as for previous assignments with the standard electronic submission policy.
  - The webtester for Problem 5 will be set up to run Connect 4 matches between students. Submission of a working evaluation function will follow the regular late policy. You may update your evaluation function on this web tester as many times as you like. Only the most last uploaded evaluation function will be used in the Connect 4 tournament.

    There will be a CPU time limit for each player in these games. If you run out of CPU time, your player forfeits the match. We'll have more details when this web tester is up.
- It is not appropriate for a student to give 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 web tester.

A. MINIMAX & Nim

For simple games, such as Nim and Coin-strip, we can search the entire game tree. In this problem, you will write MINIMAX search procedures to do this. So you can test your code, the support code contains implementation of the game "Nim".

Problem 1 (24 points) Write the procedure

```
(minimax start-state get-children game-end?)
```

where:

- `start-state` is the game state from which MAX will make the first move
- `get-children` is a procedure of one argument which given a *state* returns a list of the states can result from a legal move
- `game-end?` is a function of one argument which given a *state* returns `#t` if the game is over and the current player loses. The games we will use with your `minimax` procedure have the convention that the last player to make a move wins, i.e. the first player that doesn't have a valid move loses.

Your procedure should return a list of the form `(value final-node)` where:

- `value` is the value of the game from MAX's perspective; it should be 1 if MAX wins and -1 if MAX loses. The games we will play in this part of the assignment do not have "draws."
- `final-node` is a leaf node on the game tree corresponding to `value`. A node is a list of the form: `(state parent-node)`. Note that this recursive definition means that `final-node` contains a history of the "optimal" game that led to the value `value`. The parent of the root node is represented by a null list.

A.1 Some advice

- I strongly suggest you implement this problem using three procedures:
  - `minimax` — which calls `max-player`
  - `max-player` — which calls `min-player` (except for a leaf node)
  - `min-player` —- which calls `max-player` (except for a leaf node)

  Both `max-player` and `min-player` should return the same information as `minimax`.
- You may be tempted to take advantage of the fact that the score of a game is either +1 or -1 and thus stop trying to evaluate other children once you have found a +1 for MAX or a -1 for MIN. I advise against this, because it is actually more complicated than simply evaluating all the children. Furthermore, if you don't take this "shortcut," you will find it easier to modify your `minimax` procedure to create the alpha-beta MINIMAX procedure later in this assignment.
- In general, there will be more than one optimal move at some point in the game. It does not matter which one you pick, but your sequence of states may be different than the examples here. The value of the game at each step, however, should be the same!

A.2 Playing Nim with your MINIMAX procedure

The support code contains an implementation of a "multiple-pile version" of Nim for you to use with your MINIMAX search. At the beginning of the game, there are several piles of objects. There can be any number of piles with any number of objects in each. At each turn, a player can remove 1 or 2 objects from a single pile. The person who takes the last object wins.

The representation of the game state used in the provided implementation is a list of numbers corresponding to the number of objects in each pile. Once all the objects from a pile have been taken, the number 0 remains in this list to represent the pile.

The following procedures implement this game:

- `(nim-end? state)`
- `(nim-chilren state)`

As an example, suppose we play a 3-pile game of Nim starting with piles of 2, 3, and 2 objects:

```
(minimax '(2 3 2) nim-children nim-end?)
Evaluated 1682 states.
;Value 1: (-1 ((0 0 0)
               ((0 0 1)
                ((0 1 1)
                 ((0 3 1)
                  ((1 3 1)
                   ((1 3 2)
                    ((2 3 2) ()))))))))
```

Note that the leaf node contains the history of the entire game. Your `minimax` procedure may produce a different "optimal game," however the value of the node (and of every parent node) should be the same. My implementation prints out the number of states evaluated during a run, but yours need not.

There is also a procedure to print a more readable version of the information returned by the `minimax` procedure:

```
(print-nim-game (minimax '(2 3 2) nim-gc nim-end?))
Evaluated 1682 children.

The value of this game is -1
which means MAX will lose if MIN play optimally

Starting state is: (2 3 2)
MAX makes the move (1 0 0) after which the state is (1 3 2)
MIN makes the move (0 0 1) after which the state is (1 3 1)
MAX makes the move (1 0 0) after which the state is (0 3 1)
MIN makes the move (0 2 0) after which the state is (0 1 1)
MAX makes the move (0 1 0) after which the state is (0 0 1)
MIN makes the move (0 0 1) after which the state is (0 0 0)
```

And finally, there is a procedure that will allow you to play against your `minimax` procedure:

- `(play-nim start-state)` — computer plays first
- `(play-nim start-state #t)` — you play first

B. Coin-strip

In this problem, you will implement the "coin-strip" game to use with your MINIMAX search from Problem 1.

This game is played on a semi-infinite strip divided into boxes. Assume that we number the boxes starting with 1 and the numbers increase going to the right. Any number of coins can be placed, one per box, on the strip. Two players alternate moving one coin to the left (to a lower-numbered box). However, coins cannot pass each other, and at most one coin may occupy any box. Therefore, the coin may be moved any number of spaces to the left but only up until the next coin

or the end of the strip. The game ends when the n coins are in boxes 1 through n. The last player to move a coin wins (i.e. the first player who does not have a move loses).

Here's an example game:

```
                                    1  2  3  4    Scheme representation
Starting state:                        *     *          (2 4)
Player 1 moves a coin from 4 to 3:     *  *             (2 3)
Player 2 moves a coin from 2 to 1:  *     *             (1 3)
Player 1 moves a coin from 3 to 2:  *  *                (1 2)
Player 2 loses
```

We will represent this game with a list of numbers (in increasing order) which indicate the position of each coin. Your implementation should be able to handle an arbitrary number of coins.

Problem 2 (24 points) Write the following procedures:

- `(coin-end? state)` — returns `#t` if the coins are in boxes 1 through n. Note that you do not need to know what n is in order to write this function!
- `(coin-children state)` — given a state of the game, returns a list of the possible states that result from a legal move.

B.1 Playing coin-strip with your MINIMAX procedure

Once you have implemented the game, you can use your minimax procedure to evaluate a game state. For example, suppose we start with coins in boxes 3, 5, and 6:

```
(minimax '(3 5 6) coin-children coin-end?)
Evaluated 481 states.
;Value 2: (1 ((1 2 3)
              ((1 2 4)
               ((1 3 4)
                ((1 3 5)
                 ((1 4 5)
                  ((1 4 6)
                   ((1 5 6)
                    ((3 5 6) ()))))))))))
```

As with the game of Nim, there is support code for playing Coin-strip against your MINIMAX procedure:

- `(play-coin start-state)` which let you play against your MINIMAX procedure. (Give a second optional argument of `#t` in order to go first.)
- `(print-coin-game return-val)` which will nicely print out the game corresponding to a leaf node returned by MINIMAX.

C. MINIMAX with alpha-beta pruning

Alpha-beta pruning allows a MINIMAX search to prune certain branches of the game tree, resulting in faster evaluation of the tree.

Problem 3 (28 points) A separate handout/worksheet will have a written alpha-beta pruning problem. You will indicate which nodes are evaluated, give the value of the game tree, and indicate an "optimal" path from the root node to a leaf node of the tree.

D. 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)
```

D.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." 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.

Problem 4 (36 points) Write the procedure:

```
(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 this 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 is done by the call: `(eval-fn state 'X 'X)` When MIN plays `'O'` and evaluates a leaf node, 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)

Your procedure must return a "player procedure" that uses MINIMAX search with alpha-beta pruning (to the given depth cutoff and with the given evaluation function) for the Connect 4 game. This 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 player, either (the symbol) `X` or `O`. This procedure must return a valid move, i.e. an integer between 1 and 7 inclusive.

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

As with the MINIMAX problem, I strongly advise writing the alpha-beta MINIMAX using three procedures.

D.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 `assign4-45.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)
  ;Value: 7
  > (inf:< pos-infinity 200)
  ;Value: #f
  ```

D.2 Evaluation functions

An evaluation function returns a number that indicates how good the board state is for MAX.

> **Problem 5** (40 points) Write a procedure:
>
> ```
> (c4-eval board current-player max-player)
> ```
>
> The arguments and return value of this procedure are described in Problem 4 under the `eval-fn` argument. Part of your score on this problem will be based on the performance of your evaluation function.

There is also a written component to this part:

> **Problem 6** (12 points) Turn in a concise but detailed prose description of your evaluation function. Run it against one of our "reference" evaluation functions and analyze the resulting game: where did your evaluation function cause the computer to make good moves or bad moves, and suggest some possible ways your evaluation function could be improved with additional work.

D.2.1 Support code

Most of support code in `connect4.scm` falls into one of the following categories:

- Board representation and accessors — basic procedures to find out what piece (if any) is at a given board location
- Game implementation procedures
- Feature detectors — procedures that look for certain patters in the board