CSCI 4150 Introduction to Artificial Intelligence, Fall 1999
Assignment 4: out Thursday September 23, due Thursday October 7

Turn in electronically to `/dept/cs/ai/submit4` your procedures: `minimax`, `coin-end?`, `coin-gc`, `ab-minimax`, and `othello-eval`. Turn in your solution to Problem 3a (on paper) in class.

1 Minimax (& Nim)

Write a procedure (minimax start-state get-children game-end?) which performs MINIMAX search on a game specified by the three arguments. This minimax search should search to the bottom of the game tree. Assume MAX plays first. Your function should return a list where the first element is the minimax value of the game and the second element is the final node (not state) of the path found by minimax.

For this problem, a node consists of a list where the first element is the state at that node, and the second element is the parent *node*. Note that the final node of the best path found by MINIMAX contains a history of the "optimal" game.

The three functions given to your minimax search are:

- `start-state` — the state of the game from which MAX will make the first move
- `get-children` — a function of one argument which given a *state* returns a list of the states that can result from a legal move.
- `game-end?` — a function of one argument which given a *state* returns `#t` or `#f` if the given state finishes the game.

Assume that the games we will use with this procedure are such that the player who does not have a move loses. The value of the game is 1 or -1 and should be reported from the perspective of MAX. Thus if `(game-end? state)` is true and it is MAX's turn, the value of the game is -1.

In order for you to test your MINIMAX search, I am providing an implementation of Nim. The game of Nim is played by starting with a number of piles of objects. At each turn, a player can remove 1 or 2 objects from one pile. The person who takes the last object wins.

The following functions implement Nim for arbitrarily many piles:

- `nim-end?` — determines if the game has ended, i.e. there are 0 objects in all of the piles. When this function returns true, the current player loses the game.
- `nim-gc` — given a state, returns a list of the child states, i.e. all states that can result from a legal move.

The state of the game is a list of numbers which represent the number of objects in each pile.

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-gc 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) ()))))))))
```

The value of the game (-1) indicates that MAX will lose this game if MIN plays opimally. Note the structure of the final node of the game tree; the root node of the game tree has `'()` as its parent node. My implementation prints out the number of states evaluated during a run, but yours need not. At any step, if there are multiple states with the identical highest or lowest (depending on who is playing) values, the actual sequence of states reported will depend upon which one is chosen.

I will also provide a `(play-nim start-state)` which will let you play Nim against your MINIMAX procedure and a mechanism for counting children. See the Assignment 4 web page for details.

I suggest you implement the `minimax` function in three parts: a main function that calls a procedure `max-player` which calls a procedure `min-player` which calls the `max-player` procedure which calls the `min-player` procedure and so on ...

2 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 to a box, on the strip. At any given move, a player can move one coin any number of spaces to the left (to a lower-numbered box) up to the box adjacent to the box with the next "lowest" coin. Obviously, the game will end when the n coins are in boxes 1 through n. The last person to move a coin wins (or the first person who does not have move loses).

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.

Write the following procedures to implement this game:

- `(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-gc state)` — given a state of the game, returns a list of the possible states that result from a legal move. Similarly, you do not need to know n to implement this function.

For example, suppose we start with coins in boxes 3, 5, and 6:

```
(minimax '(3 5 6) coin-gc 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) ()))))))))))
```

I will provide a procedure `(play-coin start-state)` to let you play against your MINIMAX procedure.

3 Alpha-beta pruning & Othello

In this problem, you will create a new version of your MINIMAX procedure that implements alpha-beta pruning (with a depth cutoff) and use it to play Othello. You will also write an evaluation function for the game of Othello. I will provide the support functions for Othello — generating children, printing boards, etc.

Othello is the commonly known version of the traditional game Reversi. Othello is played on an 8 by 8 array (though my implementation will handle any square array with an even number of boxes on each side). The playing pieces are two sided: black on one side, white on the other. The first four pieces are played in the 2 by 2 square in the middle of the board, two black pieces in opposite corners and two white pieces in the remaining corners.

At each move a player puts down a piece of their color. A legal move must be adjacent to an opponent's piece, and this move must "surround" at least one of the opponent's pieces, i.e. there must be at least one row, column, or diagonal where one of the player's pieces is on the other side of a sequence of the opponent's pieces. Any pieces thus surrounded by a move are flipped to the player's color. The game ends when neither player has a legal move (which is often but not always when the board has been filled).

Please see the Assignment 4 web page for more details about this problem. Here are the three parts to this assignment:

(a) Written alpha-beta pruning problem: there will be a sample game tree on the web page (also handed out in class). Indicate which nodes will be evaluated by a MINIMAX search using alpha-beta pruning.

(b) Write the `ab-minimax` procedure which implements the MINIMAX search with a depth cutoff and alpha-beta pruning. I will provide a (simple) sample evaluation function for Othello, along with other support code to access the board, play interactively with your MINIMAX function, etc.

(c) Write an evaluation function `othello-eval` to use in playing Othello.