CSCI 4150 Introduction to Artificial Intelligence, Fall 2001
Assignment 4 (180 points): out Thursday October 4, various due dates (see below)

# Notes on this assignment

- There are 5 problems on this assignment which will be due over the next 3 weeks as follows:
  - Problem 3 (written) due Thursday October 11
  - Problems 1 and 2 due Thursday October 18
  - Problems 4 and 5 due Thursday October 25
- Support code for this assignment is only available for MIT Scheme
- The programming portions of this assignment will only be accepted through the web tester.

# Minimax & Nim

For the problems in this section, you will implement a general MINIMAX search that will search the entire game tree. I will provide procedures that implement the game of Nim for you to test your `minimax` procedure. You will also use your MINIMAX search with Welter's game later in this assignment.

**Problem 1** (40 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 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.

## 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 I'll provide. However, the value of the game at each step should be the same!

## Playing Nim with your MINIMAX procedure

I am providing 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 functions 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-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) ()))))))))
```

Please note that 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. (I will provide some procedures you can use to count the number of states.)

I have also provided a procedure to print a more readable version of the information returned by `minimax`:

```
(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)
```

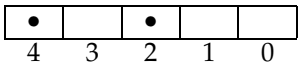
```
MAX makes the move (1 0 0) after which the state is (0 3 1)
MIN makes the move (0 2 0) after which the state is (0 1 1)
MAX makes the move (0 1 0) after which the state is (0 0 1)
MIN makes the move (0 0 1) after which the state is (0 0 0)
```

And finally, I am providing a procedure that will allow you to play against a computer player that uses your `minimax` procedure:

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

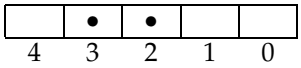
## Welter's game

Welter's game is played on a semi-infinite array of cells. Assume that the first cell is cell 0 and that they increase in number going to the left. At the start of the game, any number of coins are placed in cells, one coin per cell. We will represent the state of this game by a list of numbers in decreasing order which correspond to the location of each coin. For example:

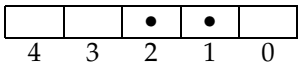



`(4 2)`

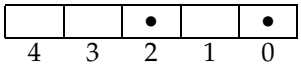
At each turn, a player can move one coin to the right any number of cells to any unoccupied cell. For example, the child states of the above state are:



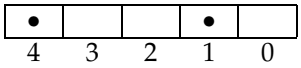

`(3 2)`



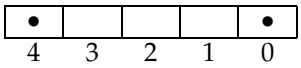

`(2 1)`




`(2 0)`




`(4 1)`




`(4 0)`

Eventually, $N$ coins will occupy cells 0 through $N-1$, and there will be no valid move. The winner is the player that makes the last move.

**Problem 2** (40 points) Write the procedures:

- `(w-children state)` which returns a list of states that can result from a valid move from the given state.
- `(w-end? state)` which returns `#t` if the game is over, i.e. the coins are in cells 0 through $N-1$, so the current player loses.

As with Nim, I am providing a number of procedures to help you test and play with your implementation of Welter's game:

- `(print-w-game value)` takes a value returned by your `minimax` procedure and prints a readable description of the "optimal game" from the starting state.
- `(play-welters start-state)` and `(play-welters start-state #t)` use your procedures for Problem 2 and your `minimax` procedure to let you to play the computer. The first invocation lets the computer play first; the second allows you to play first.

## Connect 4 & alpha-beta pruning

Connect 4 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 visual clarity on a text screen, we'll 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.

I will provide support code that implements the Connect 4 game — printing, accessing, and manipulating boards, getting the children of a board state, printing boards, 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.

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

> **Problem 3** (20 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.

Although our text gives an algorithm for alpha-beta pruning, it returns only the value of the tree. To use this algorithm in a computer game player, you must be able to return MAX's best move from the root node of the game tree. We will discuss in class how to modify the algorithm to do this — it is not as straightforward as it may seem!

> **Problem 4** (40points) Write a 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 either (the symbol) `X` or `O`. This procedure must return a move, i.e. an integer between 1 and 7 inclusive.

### Notes on your `create-c4-player` procedure

Your `create-c4-player` procedure must specialized for the game of Connect 4 — you'll have to use the `c4-children` procedure, for example. Implementing this procedure will be much easier assuming you code will only be used for the Connect 4 game instead of being more "general purpose." However, *your*

`create-c4-player` *procedure cannot be specialized for your evaluation function!* Your procedure must be able to run with any evaluation function for purposes of testing.

I am asking you to write a procedure that returns a "player procedure" because I will test your underlying alpha-beta MINIMAX search using my evaluation function and will also need to control the depth cutoff. This also allows you to use any node representation that you want. I will provide a sample `create-ab-minimax-player` function; this will simply act as a front end to your alpha-beta minimax implementation.

As for the regular MINIMAX search, I suggest you implement the alpha-beta MINIMAX in three functions: `ab-minimax`, `ab-max-player`, and `ab-min-player`. I will provide the following procedures for operations on the real 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)
```

## Creating an evaluation function

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.

The support code provides a number of "feature detectors" that you can use to formulate an evaluation function. There is also enough detail and examples that you can write your own feature detectors if you wish.

## Getting started with Connect 4 support code

- There will be two files for this assignment:
  - `a4code.com` — compiled support code for Nim, Welter's game, and some Connect 4 support code
  - `connect4.scm` — source code (reasonably well commented) that contains procedures for the board state, feature detectors, and more…
- First, try playing a game using the support code. I am only providing a "random player," but this should familiarize you with the game. After loading the above files, type:

  ```
  (play-c4 random-player human-player)
  ```

  The `play-c4` procedure takes two player procedures, the first of which plays `X`. When you have your evaluation function and `create-c4-player` working, you can evaluate:

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

  to play against the computer, limiting it to looking ahead at most 4 moves.
- Read through the `connect4.scm` file to familiarize yourself with the Connect 4 board state representation and how to access it.