

You will turn in this assignment electronically to the AI web tester. Stay tuned for details.

1 Minimax (& Nim)

(35 points) For this problem, you will implement a MINIMAX search that will search the entire game tree. Consequently, we will only play simple games with this implementation!

Your procedure (`minimax start-state get-children game-end?`) takes the following arguments:

- `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 is the end of the game, i.e. whomever played last won the game.

Your procedure should return a list where the first element is the value of the game and the second element is the final *node* (not state) of the path found by MINIMAX.

The value of the game should be either +1 or -1 and be reported from the perspective of MAX. If (`game-end? state`) is true and it is MAX's turn, the value of the game is -1 because MAX has lost.

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

1.1 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 type of information as `minimax`.

You will 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. However, I advise against this, because I think it's more complicated than simply evaluating all the children. Furthermore, it will be more similar to the implementation of alpha-beta MINIMAX that you'll write for Problem 3.

There will generally 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!

1.2 Testing your MINIMAX search with Nim

I am providing an implementation of Nim for you to test your MINIMAX search. 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. if 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 optimally. 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. (I will provide some procedures to help count the number of states.)

Once your `minimax` procedure is working, you can then call the function `(play-nim start-state)` which will let you play Nim against your MINIMAX procedure. See the Assignment 4 information page for a transcript of a game played using `play-nim`.

2 Coin-strip

(25 points) 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. 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).

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. See the Assignment 4 information page for a transcript of a game played using `play-coin`.

3 Alpha-beta pruning & Connect 4

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 Connect 4. You will also write an evaluation function for this game. I will provide the support functions for generating children, printing boards, rudimentary analysis, etc.

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. 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. The object is to get four pieces in a row, either horizontally, vertically, or diagonally.

- a. (15 points) Written alpha-beta pruning problem: there will be a sample game tree handed out with this assignment (and will also be available on the web page). Indicate which nodes will be evaluated by a MINIMAX search using alpha-beta pruning and give the value of the tree. Show your work!
- b. (35 points) Write a procedure (`create-ab-minimax-player eval-fn depth-cutoff`) which should return a “player function.” A player function takes two arguments: (`player-fn board player`) where `board` is a Connect 4 board as defined in the support code comments and `player` is either (the symbol) X or O.

I am asking you to write a function that returns a player function 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 support functions for operations on the real numbers extended to include the symbols `pos-infinity` and `neg-infinity`.

- c. (25 points) Write an evaluation function (`c4-eval board player`) for the Connect 4 game. You should be able to test this with your `make-ab-minimax-player` function, but I will test it using my solution. Your score for this problem will be based on the performance of your evaluation function.

I am providing quite a bit of support code for this problem, and it is pretty well documented in the comments. See the Assignment 4 information page for more details.