

Adversarial search in tic-tac-toe

Due 19 March

The problem: Given a state of a tic-tac-toe game, determine the best move and its value. Specifically, you will write the functions “Minimax-Decision” and “Alpha-Beta-Search” from Chapter 6. These could be used to run a tic-tac-toe game that always makes the best moves.

This can be done as an adversarial search problem: the best move for X is the one that leads to the highest value (to X) of the subsequent board position – which is the ones that lead to the lowest value for O’s best move in the subsequent state. As you saw in class, this can be seen as a max-min search.

For the terminal situations, let the values be 1 for X winning, -1 for O winning, and 0 for a tie. X will be the maximizer here, O the minimizer. (you could generalize this if you would like).

An easy optimization: if you are maximizing, you can stop when you see a 1, similarly a -1 if you are minimizing. Taking advantage of this is easy and worthwhile.

Tic-tac-toe (or Naughts and Crosses for the anglophiles among you) has a small enough search space so this is practical – you need to search through to an end state. Still, you would be wise to start out with some simple cases – like with the board partially filled.

So what is the state of a t-t-t game? It is the 3 by 3 grid, with some cells filled with X’s and O’s—given X goes first you could determine whose move it is, and how many moves have been made. I am supplying you with code for dealing with boards and states, (also available on the web page), that provides the following functions:

make-state (*board player nmoves*) Takes a board, the player ('x or 'o) to play, and an integer (the number of moves taken) and returns a game state. (**make-state** (**make-board**) 'x 0) would return the initial game state.

state-board (*state*) Takes a state, returns its board.

state-player (*state*) Takes a state, returns its player.

state-nmoves (*state*) Takes a state, returns the number of moves taken.

make-board () Makes an empty board.

board-pos (*row col board*) Returns the token at a given row and column (each ranges from 0 to 2).

add-move (*token row col board*) Returns a copy of the board, with move added – a token at row,col location.

display-board (*str board*) If *str* is true, displays a board on screen.

initialize-state-from-file(*filename*) Reads in a description of a partial game (a set of moves in form (token row col) from a file, returns a state. Nice for testing.

add-move is well-behaved, in that it does not modify its parameter; rather it returns a new board with the change added.

How to attack this problem: Start small and simple, and write the utilities that you need. For example, having a function that determines whether the game has been tied (easy!) or won (also easy, but ugly). *Important: Use the provided board and state utilities to create, access, and modify states and boards.* Look at the code in Figure 6.3, and determine what your functions should return (and how things need to be done in LISP). Do Minimax-decision before Alpha-beta.

As always, turn in a transcript and a listing.

Good luck, and enjoy! The utility code is listed on the next page. Remember to implement both functions. Stick with the algorithm in 6.3 and 6.7, but you may be able to combine some functions in elegant ways if you are clever.

```

;; states

(defun make-state (board player nmoves)
  (list board player nmoves))

(defun state-board (state) (first state))
(defun state-player (state) (second state))
(defun state-nmoves (state) (third state))

;; board

;; board uses the elt function to provide random access -- can
;; access the nth position of list lst using (elt lst n), can
;; change the nth position's value to value of item by
;; using (setf (elt lst n) item)
;; be aware that these still require traversing lists so are not
;; cheap like array-cell access.

(defun make-board ()
  '(- - - - -))

(defun add-move (token row col board)
  (add-move-at-position token board (+ (* row 3)col)))

(defun add-move-at-position (token lst pos)
  (cond ((= pos 0)(cons token (rest lst)))
        (t (cons (first lst)
                  (add-move-at-position token (rest lst) (- pos 1))))))

(defun board-pos (row col board)
  (elt board (+ (* row 3)col)))

(defun display-board (str board)
  ;; str used in format, so can be easily used in debugging
  (format str " ~s ~s ~s~%" (elt board 0)(elt board 1)(elt board 2))
  (format str " ~s ~s ~s~%" (elt board 3)(elt board 4)(elt board 5))
  (format str " ~s ~s ~s~%" (elt board 6)(elt board 7)(elt board 8)))

(defun initialize-state-from-file(filename)
  ;; returns a state, no error checking
  ;; simple file format--moves of the form (token row column), for example
  ;; (x 0 0)(o 2 2)(x 1 2)
  ;; with or without line breaks as you prefer
  (let ((move nil) ;; temporary variables
        (nmoves 0)
        (board (make-board)))
    (with-open-file (str filename)
      (loop while (not (eq (setq move (read str nil :eof)) :eof))
            do (setq nmoves (+ nmoves 1))
                (setq board
                      (add-move (first move) (second move)(third move) board))))
    (make-state board (if (= (mod nmoves 2) 0) 'x 'o) nmoves)))

```