EE570—Artificial Intelligence
Fall 2013
Problem Set #4
Due: Friday 11–October–2013

This problem set is an exercise in *game-tree (adversarial) search*. You will implement the *min-max* procedure with *alpha/beta pruning* for playing two-person, multiple-move, zero-sum, deterministic, complete-information games. And you will instantiate this procedure for playing Tic Tac Toe. When solving this problem set, it may be useful to refer to the lecture slides for lectures 7 and 8, available on the course Web site.

Tic Tac Toe is played on an $n \times n$ board. The two players are X and O. Players alternate turns with player X playing first. The board is initially empty. At each turn, a player marks an empty square with his or her identity (i.e. X or O). The first player to mark n squares in a row (horizontally, vertically, or diagonally) with his or her mark wins.

We will represent player X as 1 and player O as -1. We will represent an $n \times n$ board as a list of n rows, from top to bottom. We will represent a row of an $n \times n$ board as a list of n positions, from left to right. Each position will be 1, indicating a mark by player X, -1, indicating a mark by player O, or 0 indicating an empty position.

We want you to implement the following procedures:

`initial-board` n [*Procedure*]

n is a positive integer. Returns b^0, the initially empty $n \times n$ Tic Tac Toe board.

`moves` b [*Procedure*]

b is a board. Returns $m(b)$, the set (represented as a list) of legal moves for player $p(b)$ in board b. You can choose whatever representation you wish for moves so long as it is accepted by `make-move` and produced by `optimal-moves~`.

`make-move` m b *[Procedure]*

m is a move. b is a board. Returns $b'(m, b)$, the board that results when player $p(b)$ takes move m in board b. You can choose whatever representation you wish for moves so long as it is what is produced by `moves` and `optimal-moves~`.

`win` b *[Procedure]*

b is a board. Returns $w^0(b)$, i.e. 1 if player 1 has won in board b, -1 if player -1 has won in board b, and 0 if neither player has won.

`optimal-moves~` k b *[Procedure]*

k is either a nonnegative integer or ∞. b is a board. k is a search-depth bound. If $k = \infty$, returns $\hat{m}(b)$, the set (represented as a list) of optimal moves for player $p(b)$ in board b. If $k \neq \infty$, returns $\tilde{m}^k(b)$, a set (represented as a list) of moves. You can choose whatever representation you wish for moves so long as it is accepted by `make-move` and produced by `moves`. Note that there is a bug in the definition of $\tilde{m}^k(b)$ in the original version of the lecture slides that has been fixed in the new version on the Web site.

If $k = \infty$, there is no search-depth bound. In this case, we want you to find the optimal moves by searching to the end of the game. No static board evaluator is necessary in this case. If $k \neq \infty$, we want you to search k moves into the future and then employ a static evaluator $\tilde{w}^0(b)$. You are free to choose your own heuristic static evaluation function, though it should return nonintegral values (see slide 16 of lecture 8) in some cases.

We want your implementation of `optimal-moves~` to perform alpha/beta pruning.

While not strictly necessary, you will probably find it helpful to implement procedures for $p(b)$, $w^*(b)$, $w_l^*(b)$, and $\tilde{w}^k(b)$ as defined in the lecture slides. You will probably also find it helpful to implement a procedure for $\tilde{w}_l^k(b)$ which is not defined in the lecture slides but is a simple and obvious extension of what is defined there.

To help debug and test your implementation, we have provided the GUI `(p4)`. Clicking on +N or −N increments or decrements the board size n respectively. Clicking on +K or −K increments or decrements the depth bound k respectively, if it is not ∞, and sets it to zero if it is ∞. Clicking on W* sets k to ∞. W* is highlighted if $k = \infty$. If $k \neq \infty$, +K and −K indicate the current value of k. Clicking on New Game initializes b to b^0. Currently, the GUI is hardwired so that the human plays X and the computer plays O. The human plays by clicking on a square. The message pane indicates when the game ends in a win or draw, the computer resigns, or the human has made an illegal move. Clicking on Quit exits the GUI.

Good luck and have fun!