Minimax & Alpha-Beta

Source: http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
Acknowledgment to Dr. Bruce Rosen at UCLA and © Bruce Rosen
Minimax algorithm

• The minimax algorithm is a way of finding an optimal move in a two player game.

• *Alpha-beta pruning* is a way of finding the optimal minimax solution while avoiding searching subtrees of moves which won't be selected.

• In the search tree for a two-player game, there are two kinds of nodes, nodes representing *your* moves and nodes representing *your opponent's* moves.

• Nodes representing your moves are generally drawn as squares (or possibly upward pointing triangles):
These are also called MAX nodes. The goal at a MAX node is to maximize the value of the subtree rooted at that node. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX node.

These are also called MIN nodes. The goal at a MIN node is to minimize the value of the subtree rooted at that node. To do this, a MIN node chooses the child with the least (smallest) value, and that becomes the value of the MIN node.
Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree that has already been seen. Specifically,

- **Alpha** is the *maximum lower bound* of possible solutions
- **Beta** is the *minimum upper bound* of possible solutions
- Thus, when any new node is being considered as a possible path to the solution, it can only work if: \[ \alpha \leq N \leq \beta \]
- where \( N \) is the current estimate of the value of the node.

Source: http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
• To visualize this, we can use a number line.
  • Alpha is the *maximum lower bound* of possible solutions
  • Beta is the *minimum upper bound* of possible solutions

• At any point in time, alpha and beta are lower and upper bounds on the set of possible solution values, like so:

• As the problem progresses, we can assume restrictions about the range of possible solutions based on min nodes (which may place an upper bound) and max nodes (which may place a lower bound). As we move through the search tree, these bounds typically get closer and closer together:
• This convergence is not a problem as long as there is some overlap in the ranges of alpha and beta.

  • Alpha is the *maximum lower bound of possible solutions*
  • Beta is the *minimum upper bound of possible solutions*

• At some point in evaluating a node, we may find that it has moved one of the bounds such that there is no longer any overlap between the ranges of alpha and beta:

• At this point, we know that this node could never result in a solution path that we will consider, so we may stop processing this node. In other words, we stop generating its children and move back to its parent node.

• For the value of this node, we should pass to the parent the value we changed which exceeded the other bound.
Example
Source: http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

• To demonstrate minimax with alpha-beta pruning, we use the following minimax tree as an example:
Example

• At the start of the problem, you see only the current state (i.e. the current position of pieces on the game board).

• As for upper and lower bounds, all you know is that it's a number less than infinity and greater than negative infinity. Thus, here's what the initial situation looks like:

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

And to the leaf node

Node with 3 to set for min limit to be 3 (or less)

Alpha is the *maximum lower bound* of possible solutions
Beta is the *minimum upper bound* of possible solutions
We pass this node back to the min node above. Since this is a min node, we now know that the minimax value of this node must be less than or equal to 3. In other words, we change beta to 3.
Example

Note that the alpha and beta values at higher levels in the tree didn't change. When processing actually returns to those nodes, their values will be updated. There is no real gain in propagating the values up the tree if there is a chance they will change again in the future. The only propagation of alpha and beta values is between parent and child nodes. If we plot alpha and beta on a number line, they now look like this:

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Next we generate the next child at depth 4, run our evaluation function, and return a value of 17 to the min node above.
Example

Since this is a min node and 17 is greater than 3, this child is ignored. Now we've seen all of the children of this min node, so we return the beta value to the max node above. Since it is a max node, we now know that it's value will be greater than or equal to 3, so we change alpha to 3.
Example

Notice that beta didn't change. This is because max nodes can only make restrictions on the lower bound. Further note that while values passed down the tree are just passed along, they aren't passed along on the way up. Instead, the final value of beta in a min node is passed on to possibly change the alpha value of its parent. Likewise the final value of alpha in a max node is passed on to possibly change the beta value of its parent. At the max node we're currently evaluating, the number line currently looks like this:

*Alpha is the maximum lower bound of possible solutions*
*Beta is the minimum upper bound of possible solutions*
Example

We generate the next child and pass the bounds along. Since this node is not at the target depth, we generate its first child, run the evaluation function on that node, and return it's value.

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

We generate the next child and pass the bounds along. Since this node is not at the target depth, we generate its first child, run the evaluation function on that node, and return its value:

**Alpha** is the *maximum lower bound* of possible solutions. **Beta** is the *minimum upper bound* of possible solutions.
Example

We generate the next child and pass the bounds along. Since this node is not at the target depth, we generate its first child, run the evaluation function on that node, and return its value: 2

Since this is a min node, we now know that the value of this node will be less than or equal to 2, so we change beta to 2
Example

We generate the next child and pass the bounds along. Since this node is not at the target depth, we generate its first child, run the evaluation function on that node, and return its value: 2

Since this is a min node, we now know that the value of this node will be less than or equal to 2, so we change beta to 2.

The number line now looks like this:

Alpha is the *maximum lower bound* of possible solutions
Beta is the *minimum upper bound* of possible solutions
Example

As you can see from the number line, there is no longer any overlap between the regions bounded by alpha and beta. In essence, we've discovered that the only way we could find a solution path at this node is if we found a child node with a value that was both greater than 3 and less than 2. Since that is impossible, we can stop evaluating the children of this node, and return the beta value (2) as the value of the node.

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

Now we move on to the parent min node. With the 3 for the first child value, we know that the value of the min node must be less than or equal to 3, thus we set beta to 3.

Now the graph of alpha and beta on a number line looks like this.

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

Since we still have a valid range, we go on to explore the next child. We generate the max node...

Alpha is the maximum lower bound of possible solutions.
Beta is the minimum upper bound of possible solutions.
Example

Since we still have a valid range, we go on to explore the next child. We generate the max node...

... it's first child min node ...
Example

Since we still have a valid range, we go on to explore the next child. We generate the max node... it's first child min node ... and finally the max node at the target depth. All along this path, we merely pass the alpha and beta bounds along. At this point, we've seen all of the children of the min node, and we haven't changed the beta bound. Since we haven't exceeded the bound, we should return the actual min value for the node. Notice that this is different than the case where we pruned, in which case you returned the beta value.
At this point, we've seen all of the children of the min node, and we haven't changed the beta bound. Since we haven't exceeded the bound, we should return the actual min value for the node. Notice that this is different than the case where we pruned, in which case you returned the beta value. The reason for this will become apparent shortly.
Example

Now we return the value to the parent max node. Based on this value, we know that this max node will have a value of 15 or greater, so we set alpha to 15. Now the graph of alpha and beta on a number line looks like this:

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Once again the alpha and beta bounds have crossed, so we can prune the rest of this node's children and return the value that exceeded the bound (i.e. 15). Notice that if we had returned the beta value of the child min node (3) instead of the actual value (15), we wouldn't have been able to prune here.
Example

Now the parent min node has seen all of its children, so it can select the minimum value of its children (3) and return.

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

Finally we've finished with the first child of the root max node. We now know our solution will be at least 3, so we set the alpha value to 3 and go on to the second child.

Alpha is the maximum lower bound of possible solutions. Beta is the minimum upper bound of possible solutions.
Example

Passing the alpha and beta values along as we go, we generate the second child of the root node...

Alpha is the maximum lower bound of possible solutions. Beta is the minimum upper bound of possible solutions.
Example

Passing the alpha and beta values along as we go, we generate the second child of the root node...

... and its first child ...

Alpha is the maximum lower bound of possible solutions.
Beta is the minimum upper bound of possible solutions.
Example

Alpha is the maximum lower bound of possible solutions. Beta is the minimum upper bound of possible solutions.

Passing the alpha and beta values along as we go, we generate the second child of the root node...

... and its first child ...
Example

Passing the alpha and beta values along as we go, we generate the second child of the root node...
... and its first child. Now we are at the target depth, so we call the evaluation function and get 2:

Alpha is the maximum lower bound of possible solutions. Beta is the minimum upper bound of possible solutions.
Example

Alpha is the maximum lower bound of possible solutions.
Beta is the minimum upper bound of possible solutions.

The min node parent uses this value to set its beta value to 2.
Example

The min node parent uses this value to set its beta value to 2.
Now the graph of alpha and beta on a number line looks like this:

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Once again we are able to prune the other children of this node and return the value that exceeded the bound. Since this value isn't greater than the alpha bound of the parent max node, we don't change the bounds.

Alpha is the maximum lower bound of possible solutions
Beta is the minimum upper bound of possible solutions
Example

From here, we generate the next child of the max node.

Alpha is the *maximum lower bound* of possible solutions.
Beta is the *minimum upper bound* of possible solutions.
Example

From here, we generate the next child of the max node.

Then we generate its child, which is at the target depth. We call the evaluation function and get its value of 3.
Example

Alpha is the maximum lower bound of possible solutions.
Beta is the minimum upper bound of possible solutions.

The parent min node uses this value to set its upper bound (beta) to 3.
Example

The parent min node uses this value to set its upper bound (beta) to 3. At this point the number line graph of alpha and beta looks like this
Example

The parent min node uses this value to set its upper bound (beta) to 3. At this point the number line graph of alpha and beta looks like this. In other words, at this point alpha = beta.

Should we prune here? We haven't actually exceeded the bounds, but since alpha and beta are equal, we know we can't really do better in this subtree.
Example

The answer is yes to prune, we should prune. The reason is that even though we can't do better, we might be able to do worse. Remember, the task of minimax is to find the best move to make at the state represented by the top level max node. As it happens we've finished with this node's children anyway, so we return the min value $3$. 
Example

The max node above has now seen all of its children, so it returns the maximum value of those it has seen, which is 3.
Example

This value is returned to its parent min node, which then has a new upper bound of 3, so it sets beta to 3.
This value is returned to its parent min node, which then has a new upper bound of 3, so it sets beta to 3:

Now the graph of alpha and beta on a number line looks like this
Once again, we're at a point where alpha and beta are tied, so we prune. Note that a real solution doesn't just indicate a number, but what move led to that number.

If you were to run minimax on the list version presented at the start of the example, your minimax would return a value of 3 (and 6 terminal nodes would have been examined).
Minimax Alpha-Beta Pruning Algorithm
Minimax Alpha-Beta Pruning Algorithm


01 function alphabeta(node, depth, α, β, maximizingPlayer)
02     if depth = 0 or node is a terminal node
03         return the heuristic value of node
04     if maximizingPlayer
05         v := -∞
06         for each child of node
07             v := max(v, alphabeta(child, depth – 1, α, β, FALSE))
08         α := max(α, v)
09         if β ≤ α
10             break (* β cut-off *)
11         return v
12     else
13         v := +∞
14         for each child of node
15             v := min(v, alphabeta(child, depth – 1, α, β, TRUE))
16         β := min(β, v)
17         if β ≤ α
18             break (* α cut-off *)
19         return v

(* Initial call *) alphabeta(origin, depth, -∞, +∞, TRUE)
Minimax Alpha-Beta Pruning Algorithm


```javascript
function alphabeta(node, alpha, beta, maximizingPlayer) {
    var bestValue;
    if (node.children.length === 0) {
        bestValue = node.data;
    } else if (maximizingPlayer) {
        bestValue = alpha;
        // Recurse for all children of node.
        for (var i=0, c=node.children.length; i< c; i++) {
            var childValue = alphabeta(node.children[i], bestValue, beta, false);
            bestValue = Math.max(bestValue, childValue);
            if (beta <= bestValue) {
                break;
            }
        }
    } else {
        bestValue = beta;
        // Recurse for all children of node.
        for (var i=0, c=node.children.length; i< c; i++) {
            var childValue = alphabeta(node.children[i], alpha, bestValue, true);
            bestValue = Math.min(bestValue, childValue);
            if (bestValue <= alpha) {
                break;
            }
        }
    }
    return bestValue;
}
```
Minimax Alpha-Beta Pruning Algorithm

http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/

```plaintext
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        v := -∞
        for each child of node
            v := max(v, alphabeta(child, depth – 1, α, β, FALSE))
            α := max(α, v)
        if β ≤ α
            break (* β cut-off *)
        return v
    else
        v := +∞
        for each child of node
            v := min(v, alphabeta(child, depth – 1, α, β, TRUE))
            β := min(β, v)
        if β ≤ α
            break (* α cut-off *)
        return v
    (* Initial call *)
    alphabeta(origin, depth, -∞, +∞, TRUE)
```
function alphaBeta(node, alpha, beta, maximisingPlayer) {
    var bestValue;
    if (node.children.length === 0) {
        bestValue = node.data;
    } else if (maximisingPlayer) {
        bestValue = alpha;
        // Recurse for all children of node.
        for (var i = 0, c = node.children.length; i < c; i++) {
            var childValue = alphaBeta(node.children[i], bestValue, beta, false);
            bestValue = Math.max(bestValue, childValue);
            if (beta <= bestValue) {
                break;
            }
        }
    } else {
        bestValue = beta;
        // Recurse for all children of node.
        for (var i = 0, c = node.children.length; i < c; i++) {
            var childValue = alphaBeta(node.children[i], alpha, bestValue, true);
            bestValue = Math.min(bestValue, childValue);
            if (bestValue <= alpha) {
                break;
            }
        }
    }
    return bestValue;
}
Minimax Alpha-Beta Pruning Algorithm

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

```plaintext
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
        α := max(α, value)
        if α ≥ β then
            break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
        β := min(β, value)
        if α ≥ β then
            break (* α cut-off *)
        return value
```

(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

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

(defun print-board (board)
  (format t "~%")
  (format t "    0|1|2~%")
  (format t " ~%")
  (format t "    ------~%")
  (format t "1   ~A|~A|~A~%" (token 3 board) (token 4 board) (token 5 board))
  (format t "    ------~%")
  (format t "2   ~A|~A|~A~%" (token 6 board) (token 7 board) (token 8 board))
  (format t " ~%"))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; used by print-board
(defun token (index board)
  (let ((item (nth index board)))
    (if item item " ")) ; converts nils to blanks, but leaves 'Xs and 'Os alone

;;; checks whether the board represents a draw
(defun is-draw (board) ; the game is a draw when the board is full
  (cond
    ((is-win board 'X) nil) ; unless it is a win for X
    ((is-win board 'O) nil) ; or it is a win for O
    (t (not (member nil board)))))

;;; checks whether the board represents a win for the given player
(defun is-win (board player)
  (let ((wins '((0 1 2) (3 4 5) (6 7 8) ; rows
                (0 3 6) (1 4 7) (2 5 8) ; columns
                (0 4 8) (2 4 6))) ; diagonals
    (check-for-wins wins board player)))

;;; checks whether the boards represents a game that is finished
(defun game-over (board)
  (or (is-win board 'X) (is-win board 'O) (is-draw board)))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; used by is-win
(defun check-for-wins (wins board player)
  (cond
    ((null wins) nil) ; got to the end of list and there were no wins
    ((is-a-win board player (car wins)) t)
    (t (check-for-wins (cdr wins) board player))))

;;; used by check-for-wins
(defun is-a-win (board player positions)
  (and (equal (nth (nth 0 positions) board) player)
       (equal (nth (nth 1 positions) board) player)
       (equal (nth (nth 2 positions) board) player)))

;;; converts 'O to 'X and vice versa
(defun other-player (player)
  (case player
    (X 'O)
    (O 'X)))
Minimax Alpha-Beta Pruning Algorithm
https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; returns a list of next moves for a given board (or nil if there are none)
(defun next-moves (board player)
  (if (is-win board (other-player player))
      nil
      (next-moves-aux nil board player)))

;;; constructs a list where there is a board corresponding to each nil
;;; slot in the original board.
(defun next-moves-aux (slots-checked slots-to-check player)
  (cond ((null slots-to-check) nil)
        ((null (car slots-to-check))
         (cons (append slots-checked (list player) (cdr slots-to-check))
               (next-moves-aux (append slots-checked '()())
                               (cdr slots-to-check) player)))
        (t (next-moves-aux (append slots-checked (list (car slots-to-check)))
                          (cdr slots-to-check) player))))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; plays tic-tac-toe
(defun play ()
  (let ((board (make-board))
         (search-level nil)
         (human-first nil)
         (player nil))
    (format t "~%Maximum search level (1-9)? ")
    (setf search-level (read))
    (format t "~%X or O? ")
    (setf player (read))
    (format t "~%Would you like to go first? (t or nil) ")
    (setf human-first (read))
    (when human-first
      (print-board board)
      (setf board (get-human-move board player))
      (print-board board))
    (do () ; (no do variables)
      (game-over board) ; until game over
      (cond ((is-draw board) 'draw)
            ((is-win board player) 'human-wins)
            (t 'machine-wins)))
    (setf board (get-machine-move board (other-player player)
                  search-level))
    (print-board board)
    (when (not (game-over board))
      (setf board (get-human-move board player))
      (print-board board))))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; asks for and executes the move of the human player
(defun get-human-move (board player)
  (let ((row nil)
        (column nil)
        (slot nil))
    (format t "Row: ")
    (setf row (read))
    (format t "Column: ")
    (setf column (read))
    (setf slot (+ (* 3 row) column))
    (cond ((or (< slot 0) (> slot 8)) (format t "Try again!~%") (get-human-move board player))
          ((null (nth slot board)) (place-token board player slot))
          (t (format t "No cheating!~%") (get-human-move board player))))

;;; used by get-human-move
(defun place-token (board player index)
  (replace-nth 0 board index player))

;;; used by place-token to make a move on a board
(defun replace-nth (i l index item)
  (cond ((equal i index ) (cons item (cdr l)))
        (t (cons (car l) (replace-nth (1+ i) (cdr l) index item))))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; the following function is not yet used
;;; you can use this function to order child nodes by
;;; supplying a list of pairs of boards and heuristic values, e.g
;;; (move-sort '((2 board1) (-1 board2) (0 board3))) => (board2 board3 board1);
;;; use the (reverse list) function on the result to sort in opposite order
(defun move-sort (items)
  (mapcar #'cadr ; extract boards from list of sorted pairs
            (sort items #'< :key #'car))) ; sort by the key in car of pair

;;; invokes minimax-search with alpha-beta pruning
(defun get-machine-move (board player max-search)
  (setf *nodes-generated* 0)
  (let ((board (minimax-alpha-beta board 0 max-search player
                                *infinity* (- *infinity*))))
    (format t "Number of nodes generated: ~S~" *nodes-generated*)
    board))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

(defun minimax-alpha-beta (board depth max-depth
  player use-thresh pass-thresh)
  (if (equal depth max-depth)
    ;; then
    (heuristic board player)
    ;; else
    (let ((successors (next-moves board player)))
      (setf *nodes-generated* (+ *nodes-generated* (length successors)))
      (if (null successors)
        ;; then
        (heuristic board player)
        ;; else
        (do ((new-value nil)
             (best-move (car successors))
             ;; when no more successors return pass-thresh or
             ;; best-move if at top level
             ((null successors) (if (= depth 0)
                                   best-move
                                   pass-thresh))
             ;; when no more successors return pass-thresh or
             ;; best-move if at top level
             ((null successors) (if (= depth 0)
                                   best-move
                                   pass-thresh)))
          ;; when no more successors return pass-thresh or
          ;; best-move if at top level
          ((null successors) (if (= depth 0)
                                best-move
                                pass-thresh)))
        (setf new-value
          (- (minimax-alpha-beta
              (car successors)
              (+ 1 depth)
              max-depth
              (other-player player)
              (- pass-thresh)
              (- use-thresh))))))
  (when (> new-value pass-thresh)
    (setf pass-thresh new-value)
    (setf best-move (car successors)))
  (if (>= pass-thresh use-thresh)
    ;; then (do the pruning)
    (setf successors nil) ; terminate the loop
    ;; else
    (setf successors (cdr successors))))))))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

(defun heuristic (board player)
  (cond
    ((is-win board player) *infinity*)
    ((is-win board (other-player player)) (- *infinity*))
    ((is-draw board) 0)
    (t (+ (* 10 (count-pairs board player))
         (* 3 (count-corners board player))
         (check-center board player))))
)

;;; counts the number of adjacent pairs player has on the board
(defun count-pairs (board player)
  (let ((pairs '((0 1) (1 2) (3 4) (4 5) (6 7) (7 8) ; rows
                  (0 3) (3 6) (1 4) (4 7) (2 5) (5 8) ; columns
                  (0 4) (4 8) (2 4) (4 6)))) ; diagonals
    (add-pairs pairs board player)))

;;; adds the number of pairs player has on the board
(defun add-pairs (pairs board player)
  (cond
    ((null pairs) 0) ; end of pair list
    (t (+ (is-a-pair (car pairs) board player)
         (add-pairs (cdr pairs) board player)))))

;;; checks if the pair is on the board
(defun is-a-pair (pair board player)
  (cond
    ((and (equal (nth (nth 0 pair) board) player)
          (equal (nth (nth 1 pair) board) player)) 1)
    (t 0)))
Minimax Alpha-Beta Pruning Algorithm

https://en.wikipedia.org/wiki/Alpha–beta_pruning

;;; counts the number of corners player has on the board
(defun count-corners (board player)
  (let ((corners '(0 2 6 8)))
    (add-corners corners board player)))

;;; adds the number of corners
(defun add-corners (corners board player)
  (cond
    ((null corners) 0)
    (t (+ (is-a-corner (car corners) board player)
         (add-corners (cdr corners) board player)))))

;;; checks if the corner position is on the board
(defun is-a-corner (corner board player)
  (cond
    ((equal (nth corner board) player) 1)
    (t 0)))

;;; checks if player has the board center
(defun check-center (board player)
  (cond
    ((equal (nth 4 board) player) 1)
    (t 0)))

;;; *EOF*