Project Task 13:
Minimax Part 3- Minimax Trees
By Carrie Corcoran

This is the thirteenth task in my semester-long project in creating heuristic mancala players. In this task, the code selects a promising subset of the found moves from the previous step and analyzes them using a height 3 minimax tree function. The static evaluation function was modified during this process. Instead of generating different values for each player, a single value is generated. Player A favors high values and Player B favors negative values.

A good deal of frustrating troubleshooting occurred for this code, in which I was painfully reminded of the importance of local variables. Initially, the minimax function would occasionally return invalid values, usually moves for the other player. This was due to the function overwriting the values saved in a previous layer. This has since been updated and the code has been tested thoroughly.

Code:

```
;helper function to select a subset of up to 50 moves
(defun random-move-subset (l)
  (cond
    ((< (length l) 50)
       l
     )
     (t
       (recursive-move-selection l 50 '())
     )
  )
)


;recusively selects random moves
(defun recursive-move-selection (l num sublist &aux which new-
element)
  (cond
    ((= num 0)
```

```
      sublist
    )
    (t
      (setf which (random (length l)))
      (setf new-element (nth which l))
     (setf l (remove new-element l))
      (setf sublist (cons new-element sublist))
      (recursive-move-selection l (- num 1) sublist)
    )
  )
)

;selects the possible moves by finding the longest from the
sample
(defun possible-move-selection (lst new-list &aux item min)
  (cond
    ((= (length lst) 0)
      new-list
    )
    ((eql new-list '())
      (setf new-list (cons (car lst) '()))
      (setf lst (cdr lst))
      (possible-move-selection lst new-list)
    )
    (t
      (setf item (car lst))
      (setf lst (cdr lst))
      (setf new-list (cons item new-list))
      (cond
```

```
          ((> (length new-list) 10)
             (setf min (car new-list))
             (loop for x in new-list do
               (cond
                  ((>(length min)(length x))
                    (setf min x)
                  )
                )
             )
             (setf new-list (remove min new-list))
          )
        )
        (possible-move-selection lst new-list)
      )
   )
)

;creates a node object containing a move list
(defun get-node (move-list move-num s player &aux side)
  (cond
    ((= move-num (length move-list))
       (create-node player s move-list)
     )
     (t
      (load-state s)
       (cond
        ((eql player 'a)
          (setf side *side-a-copy*)
```

```
        )
          (t
            (setf side *side-b-copy*)
          )
        )
       (single-move-test player (nth move-num move-list))
        (setf s (save-state nil))
        (get-node move-list (+ move-num 1) s player)
     )
  )
)

;converts a list of moves to a list of nodes
(defun get-nodes (lst num st player node-list)
  (cond
    ((= num (length lst))
       node-list
     )
     (t

       (setf node-list (cons (get-node (nth num lst) 0 st
player) node-list))
       (get-nodes lst (+ num 1) st player node-list)
     )
  )
)
```

```
;looks through list of nodes to find max eval value
(defun find-max (lst num max)
  (cond
    ((= num (length lst))
      max
     )
     ((= num 0)
       (setf max (nth 3 (car lst)))
       (find-max lst (+ num 1) max)
     )
     (t
       (cond
         ((> (nth 3 (nth num lst)) max)
            (setf max (nth 3 (nth num lst)))
          )
       )
       (find-max lst (+ num 1) max)
     )
  )
)

;looks through list of moves to find the min eval value
(defun find-min (lst num min)
  (cond
    ((= num (length lst))
      min
     )
     ((= num 0)
```

```
        (setf min (nth 3 (car lst)))
        (find-min lst (+ num 1) min)
      )
      (t
        (cond
          ((> (nth 3 (nth num lst)) min)
             (setf min (nth 3 (nth num lst)))
           )
        )
        (find-min lst (+ num 1) min)
      )
  )
)



;MINIMAX FUNCTION looks at a list of moves and looks ahead to
evaluate
;which move will be the most productive based on a similarity
metric
(defun minimax (level moves player &aux num new-player
evaluated-moves open-moves
  lists-of-moves new-moves min max)
  (cond
     ;at level 2, evaluate the list of moves and return the
needed value
    ((= level 2)
       (cond
         ((eql player 'a)
            (setf num (find-max moves 0 0))
          )
```

```
          (t
            (setf num (find-min moves 0 0))
          )
        )
        num
      )
      ;otherwise, generate list of future moves and recursive
call
      (t
        (cond
          ((eql player 'a)
            (setf new-player 'b)
          )
          (t
            (setf new-player 'a)
          )
        )
        (setf evaluated-moves '())
        (loop for x in moves do
          (load-state (nth 1 x))
          (setf open-moves (non-empty-spaces new-player 0 '()
t))
          ;if there are future moves from this state, evaluate
them
          (cond
            ((> (length open-moves) 0)
              (setf lists-of-moves '())
              (loop for y in open-moves do
                (setf lists-of-moves  (cons (cons y '()) lists-
of-moves))
```

```
              )
             (setf new-moves (find-moves (create-node new-
player(nth 1 x) open-moves )
                '() lists-of-moves 500))
              (setf new-moves (random-move-subset new-moves))
              (setf new-moves (possible-move-selection new-moves
'()))
              (setf nodes (get-nodes new-moves 0 (nth 1 x) new-
player '()))
              (setf num (minimax (+ level 1) nodes new-player))
             )
             ;otherwise, set a terminal value
             (t
               (cond
                  ((= level 0)
                    (cond
                       ((eql player 'a)
                         (setf num 50)
                       )
                       (t
                         (setf num -50)
                       )
                     )
                  )
                  (t
                    (cond
                       ((eql player 'b)
                         (setf num 50)
                       )
                       (t
```

```
                        (setf num -50)
                      )
                    )
                  )
                )
              )
            )
             (setf evaluated-moves (cons (cons num x) evaluated-
moves))
          )
          ;Checks through list of evaluated moves to find best move
          (cond
            ((= level 1)
               (cond
               ((eql player 'b)
                  (setf min (car (car evaluated-moves)))
                  (loop for x in evaluated-moves do
                    (cond
                       ((< (car x) min)
                         (setf min (car x))
                       )
                     )
                   )
                 min
                 )
                 (t
                 (setf max (car (car evaluated-moves)))
                  (loop for x in evaluated-moves do
                    (cond
```

```
            ((> (car x) max)
              (setf max (car x))
            )
          )
        )
         max
      )
    )
  )
  (t
    (cond
     ((eql player 'b)
        (setf min (car evaluated-moves))
        (loop for x in evaluated-moves do
          (cond
             ((< (car x) (car min))
               (setf min x)
             )
          )
        )
         min
      )
      (t
      (setf max (car evaluated-moves))
       (loop for x in evaluated-moves do
         (cond
             ((> (car x) (car max))
               (setf max x)
```

```
                    )
                  )
                )
                 max
               )
            )
          )
        )
      )
    )
)

;-------------------------------------Supplemental code to
generate full minimax move


;takes list of potential-moves and makes them each their own
sublist of -copy moves
;(formatting for minimax)
(defun real-to-copy-list (moves num lst &aux new-move)
  (cond
    ((= num (length moves))
       lst
     )
     (t
       (cond
         ((eql (nth num moves) 'a1)
            (setf new-move 'a1-copy)
          )
         ((eql (nth num moves) 'a2)
```

```
   (setf new-move 'a2-copy)
 )
((eql (nth num moves) 'a3)
   (setf new-move 'a3-copy)
 )
((eql (nth num moves) 'a4)
   (setf new-move 'a4-copy)
 )
((eql (nth num moves) 'a5)
   (setf new-move 'a5-copy)
 )
((eql (nth num moves) 'a6)
   (setf new-move 'a6-copy)
 )
((eql (nth num moves) 'b1)
   (setf new-move 'b1-copy)
 )
 ((eql (nth num moves) 'b2)
   (setf new-move 'b2-copy)
 )
 ((eql (nth num moves) 'b3)
   (setf new-move 'b3-copy)
 )
 ((eql (nth num moves) 'b4)
   (setf new-move 'b4-copy)
 )
 ((eql (nth num moves) 'b5)
   (setf new-move 'b5-copy)
```

```
          )
          ((eql (nth num moves) 'b6)
            (setf new-move 'b6-copy)
          )
        )
        (setf lst (cons (cons new-move '()) lst))
        (real-to-copy-list moves (+ num 1) lst)
      )
  )
)

;takes -copy values and returns the same list without the suffix
(defun copy-to-real (moves num lst &aux new-move)
  (cond
    ((= num -1)
      lst
    )
    (t
      (cond
        ((eql (nth num moves) 'a1-copy)
          (setf new-move 'a1)
        )
        ((eql (nth num moves) 'a2-copy)
          (setf new-move 'a2)
        )
        ((eql (nth num moves) 'a3-copy)
          (setf new-move 'a3)
        )
```

```
((eql (nth num moves) 'a4-copy)
  (setf new-move 'a4)
)
((eql (nth num moves) 'a5-copy)
  (setf new-move 'a5)
)
((eql (nth num moves) 'a6-copy)
  (setf new-move 'a6)
)
((eql (nth num moves) 'b1-copy)
  (setf new-move 'b1)
)
((eql (nth num moves) 'b2-copy)
  (setf new-move 'b2)
)
((eql (nth num moves) 'b3-copy)
  (setf new-move 'b3)
)
((eql (nth num moves) 'b4-copy)
  (setf new-move 'b4)
)
((eql (nth num moves) 'b5-copy)
  (setf new-move 'b5)
)
((eql (nth num moves) 'b6-copy)
  (setf new-move 'b6)
)
((eql (nth num moves) 'a1-copy)
```

```
            (setf new-move 'a1)
          )
        )
        (setf lst (cons new-move lst))
        (copy-to-real moves (- num 1) lst)
      )
  )
)

;Runs a recursive move for a given player, assuming current game
state
(defun minimax-move (player verbose &aux s n open m moves nodes
result
  move-list)
  (setf s (save-state t))
  (setf n (create-node player s 'a1))
  (setf open (non-empty-spaces player 0 '() nil))
  (setf m (real-to-copy-list open 0 '()))
  (setf moves (find-moves n '() m 500))
  (setf moves (random-move-subset moves))
  (setf moves (possible-move-selection moves '()))
  (format t "Potential moves: ~A~%" moves)
  (setf nodes (get-nodes moves 0 s player '()))
  (setf result (minimax 0 nodes player ))
  (setf move-list (copy-to-real (nth 3 result)(- (length (nth 3
result)) 1) '()))
  (cond
    (verbose
       (format t "Minimax Player's selected move: ~A~%" move-
list)
```

```
      )
  )
  (recursive-move player move-list verbose)
)

;Runs a list of moves recursively
(defun recursive-move (player lst verbose &aux current future)
  (cond
    ((> (length lst) 0)
       (setf current (car lst))
       (setf future (cdr lst))
       (single-move player current verbose)
       (recursive-move player future verbose)
     )
  )
)

(defun minimax-test ()
  (format t "Choosing Player A's opening move using minimax:
~%")
  (setf s (save-state t))
  (setf n (create-node 'a s 'a1))
  (setf m '((a1-copy)(a2-copy)(a3-copy)(a4-copy)(a5-copy)(a6-
copy)))
  (setf moves (find-moves n '() m 500))
  (setf moves (random-move-subset moves))
  (setf moves (possible-move-selection moves '()))
  (format t "Potential moves in copy format: ~A~%" moves)
  (setf nodes (get-nodes moves 0 s 'a '()))
  (setf result (minimax 0 nodes 'a))
```

```
  (format t "~%Chosen move: ~A with a static evaluation score of
~A~%" (nth 3 result) (nth 4 result) )

)
```

Demo:

```
[266]> (minimax-test)

Choosing Player A's opening move using minimax:

Potential moves in copy format:

((A2-COPY A2-COPY A3-COPY A5-COPY A1-COPY A3-COPY A5-COPY A4-
COPY)

 (A2-COPY A2-COPY A3-COPY A6-COPY A1-COPY A2-COPY A3-COPY A2-
COPY)

 (A2-COPY A2-COPY A3-COPY A6-COPY A1-COPY A2-COPY A1-COPY A2-
COPY)

 (A2-COPY A2-COPY A3-COPY A5-COPY A1-COPY A3-COPY A2-COPY A1-
COPY)

 (A2-COPY A2-COPY A3-COPY A5-COPY A1-COPY A2-COPY A3-COPY A2-
COPY)

 (A2-COPY A2-COPY A3-COPY A6-COPY A1-COPY A3-COPY A2-COPY A4-
COPY)

 (A6-COPY A3-COPY A2-COPY A5-COPY A3-COPY A6-COPY A5-COPY) (A2-
COPY A2-COPY A3-COPY A1-COPY A2-COPY A1-COPY A2-COPY)

 (A3-COPY A5-COPY A5-COPY A5-COPY A3-COPY A5-COPY A6-COPY) (A6-
COPY A6-COPY A3-COPY A3-COPY A5-COPY A2-COPY A6-COPY))


Chosen move: (A2-COPY A2-COPY A3-COPY A6-COPY A1-COPY A3-COPY
A2-COPY A4-COPY) with a static evaluation score of 20

NIL
```