Memoria Practica3

Daniel Redondo Castilla
Jose Manuel de Frutos Porras
Pareja 9
Grupo 2301

Pregunta B1

Apdo B1.1:

Tres principios básicos gobiernan la construcción de una función heurística.
(1) Los estados terminales deben ser evaluados igual que lo haría una función de utilidad.
Los estados victoriosos deben evaluar mejor que los empates y estos últimos mejor que las derrotas. De otro modo un agente podría tomar como decisión elegir una jugada de empate o derrota aunque pueda también elegir justo una jugada que conduce inmediatamente a la victoria.

(2) Deben ser sencillas y rápidas. El objetivo de la función evaluación es buscar rápido y no tener que expandir todo el árbol.

(3) Para los estados no terminales, la evaluación debe estar fuertemente correlada con la probabilidad actual de ganar.

Como documentación hemos tomado el documento https://www.ittc.ku.edu/publications/documents/Gifford_ITTC-FY2009-TR-03050-03.pdf, el cual indicaba seis heurísticas básicas que hemos tomado como referencia, y el libro indicado como referencia en el enunciado:
Russel, S. and Norvig, P. Imperfect Real-Time Decisions. In Artificial Intelligence: A Modern Approach, Chapter 6, pp. 171-175. New Jersey, USA, 2003.

Para probar las heurísticas realizadas mediante combinaciones lineales de estas heurísticas hemos utilizado como rivales a las anteriores heurísticas creadas y a las proporcionadas por el código dado. Los jugadores que iban consiguiendo mejores resultados eran subidos al torneo.

Heurística utilizada:

```
(defun aux1-snape (estado)
  (- (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 6)
     (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 6)))

(defun aux2-snape (estado)
  (- (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 5)
     (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 5)))


(defun aux3-snape (estado)
  (- (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 0)
     (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 0)))

(defun f-eval-snape (estado)
  (+ (* 5 (aux1-snape estado)) (* 2 (aux2-snape estado)) (aux3-snape estado)))


(setf *jdr-mmx-snape* (make-jugador
                       :nombre   '|Ju-Mmx-snape|
                       :f-juego  #'f-j-mmx
                       :f-eval   #'f-eval-snape))
```

COMENTARIO:
Hemos usado una función que asigna pesos secuencialmente y realiza partidas contra un jugador

pasado por argumento. Si con una secuencia de pesos resulta que el nuevo jugador vence al pasado como argumento se imprime los pesos que han dado éxito.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ajustar-peso
;;;
;;; Funcion que ajusta los pesos de ciertas medidas heuristicas y crea un jugador cuya heuristica esta
;;; dada por la proporcion de los pesos.
;;; Cuando uno de los jugadores vence tanto empezando antes como despues al pasado como argumento se imprime la combinacion de pesos
;;; por pantalla de este nuevo jugador.
;;;
;;; Objetivo:
;;;  Buscar el maximo dado por la combinacion de heuristicas básicas.
;;;
;;; ATENCION: Se recomienda compilar el codigo para ejecutar esta funcion.


(defun ajustar-peso (jugador)
 (setf path (make-pathname :name "pesos"))
  (setf str (open path :direction :output
                   :if-exists :supersede))

  (setf i 0)
  (setf j 0)
  (setf start 0)
  (defun f-eval-p (estado)
     (+ (* i (aux1-snape estado)) (* j (aux2-snape estado))))

  (setf *jdr-mmx-p* (make-jugador
                       :nombre   '|Ju-Mmx-p|
                       :f-juego  #'f-j-mmx
                       :f-eval   #'f-eval-p))
  (do ((i start (+ i 1)))
    ((> i 100))
    (do ((j start (+ j 1)))
      ((> (+ i j) 100))
        (when (and (= (partida 0 2 (list jugador *jdr-mmx-p*)) 2) (= (partida 1 2 (list jugador  *jdr-mmx-p*)) 2))
          (format str "~D ~D" i j))))
    (close str))



;;; (ajustar-peso *jdr-mmx-snape*)
```

Sin embargo no hemos obtenido resultados significativos.

<u>Apdo B1.2:</u>

El principal problema de jugador bueno reside en que a un estado terminal le asigna el valor -50. Pero hay estados victoriosos a los que no necesariamente se llega cuando el jugador max termina la partida.
La solución consistiría en distinguir entre estados terminales victoriosos y no victoriosos por este motivo.

Pregunta B2

<u>Apdo B2.1:</u>
(a)
Ejemplo de ejecución de la función generar-sucesores:
Creamos un estado muy sencillo con solo tres sucesores:

```
(setq mi-posicion (list '(1 0 1 1 0 0 7) (reverse '(7 0 0 0 1 1 1))))
(setq estado (crea-estado-inicial 0 mi-posicion))
(generar-sucesores estado)
```

Salida:

```
---- Gen.Sucesores de #2A((1 0 1 1 0 0 7) (1 1 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 2)  => Ultima ficha en 3, con 1, contra: 1: (10 10)
Juega (R 3)  => Ultima ficha en 4, con 0, contra: 1 => Captura 1 y termina,
(#S(ESTADO :TABLERO #2A((0 1 1 1 0 0 7) (1 1 1 0 0 0 7)) :LADO-SGTE-JUGADOR 1 :DEBE-PASAR-TURNO NIL :ACCION (R 0))
 #S(ESTADO :TABLERO #2A((1 0 0 2 0 0 7) (1 1 1 0 0 0 7)) :LADO-SGTE-JUGADOR 1 :DEBE-PASAR-TURNO NIL :ACCION (R 2))
 #S(ESTADO :TABLERO #2A((1 0 1 0 0 0 9) (1 0 1 0 0 0 7)) :LADO-SGTE-JUGADOR 1 :DEBE-PASAR-TURNO NIL :ACCION (R 3)))
```

La salida es la esperada el jugador uno solo tiene tres posibles opciones:
(1) mover la ficha mas a la izquierda y depositarla una casilla más a la derecha (estado sucesor 1)
(2) mover la ficha en la casilla 2 y depositarla una casilla 3 (estado sucesor 2)
(3) mover la ficha en la casilla 3 y capturar las fichas del contricante y depositarlas en el kalaha (estado sucesor 3)

Ejemplos de la función minimax:
La función minimax lo único que hace es leer y modificar las variables de debug y llamar a la función minimax-1, que es la que tiene toda la lógica del algoritmo. Por ello probamos sólo esta última:

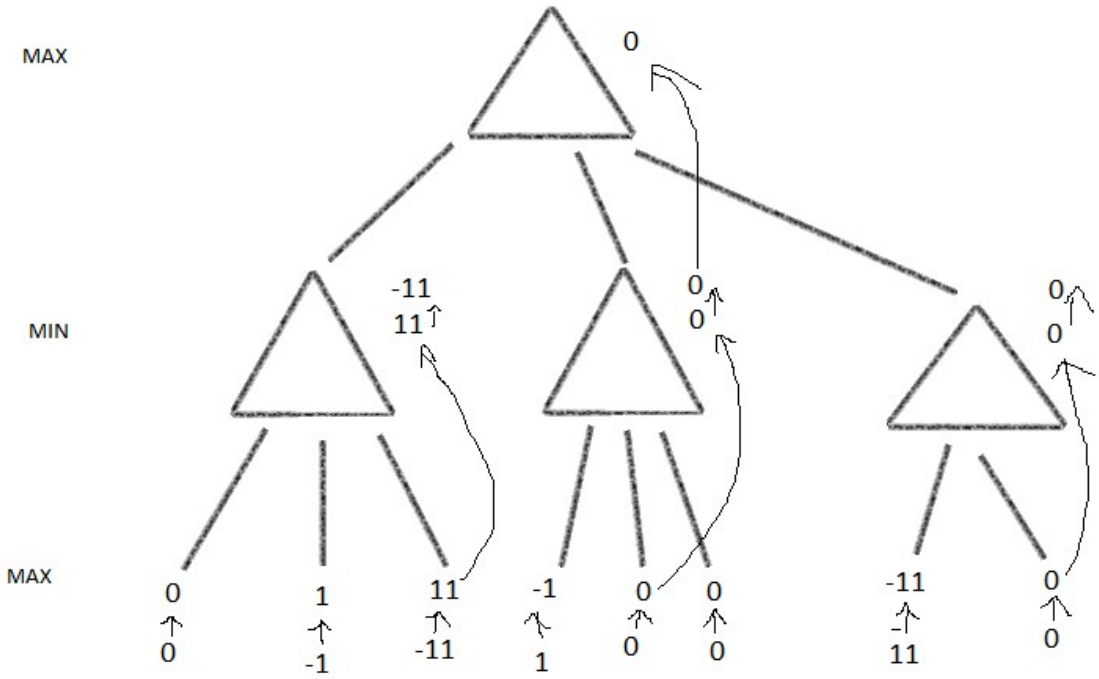
Para profundidad 2 (ejemplo de profundidad par):

```
(minimax-1 estado 0 t 2 #'f-eval-snape)
```

Salida:

```
 ---- Gen.Sucesores de #2A((1 0 1 1 0 0 7) (1 1 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 2)  => Ultima ficha en 3, con 1, contra: 1: (10 10)
Juega (R 3)  => Ultima ficha en 4, con 0, contra: 1 => Captura 1 y termina,
 ---- Gen.Sucesores de #2A((0 1 1 1 0 0 7) (1 1 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 1, contra: 0: (10 10)
Juega (R 1)  => Ultima ficha en 2, con 1, contra: 1: (10 10)
Juega (R 2)  => Ultima ficha en 3, con 0, contra: 1 => Captura 1 y termina,
0
-1
-11
 ---- Gen.Sucesores de #2A((1 0 0 2 0 0 7) (1 1 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 1, contra: 0: (10 10)
Juega (R 1)  => Ultima ficha en 2, con 1, contra: 2: (10 10)
Juega (R 2)  => Ultima ficha en 3, con 0, contra: 0
1
0
0
 ---- Gen.Sucesores de #2A((1 0 1 0 0 0 9) (1 0 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 2)  => Ultima ficha en 3, con 0, contra: 1 => Captura 1 y termina,
11
0
#S(ESTADO :TABLERO #2A((1 0 0 2 0 0 7) (1 1 1 0 0 0 7)) :LADO-SGTE-JUGADOR 1 :DEBE-PASAR-TURNO NIL :ACCION (R 2))
```

Hemos modificado la función de evaluación para que también imprima el valor de la heurística.
Entonces se ve que aplicando minimax al algoritmo con la versión negamax implementada se toma como elección el segundo movimiento posible (R 2).
Como la profundidad es par los valores impresos se corresponden con el valor asociado a min del estado.



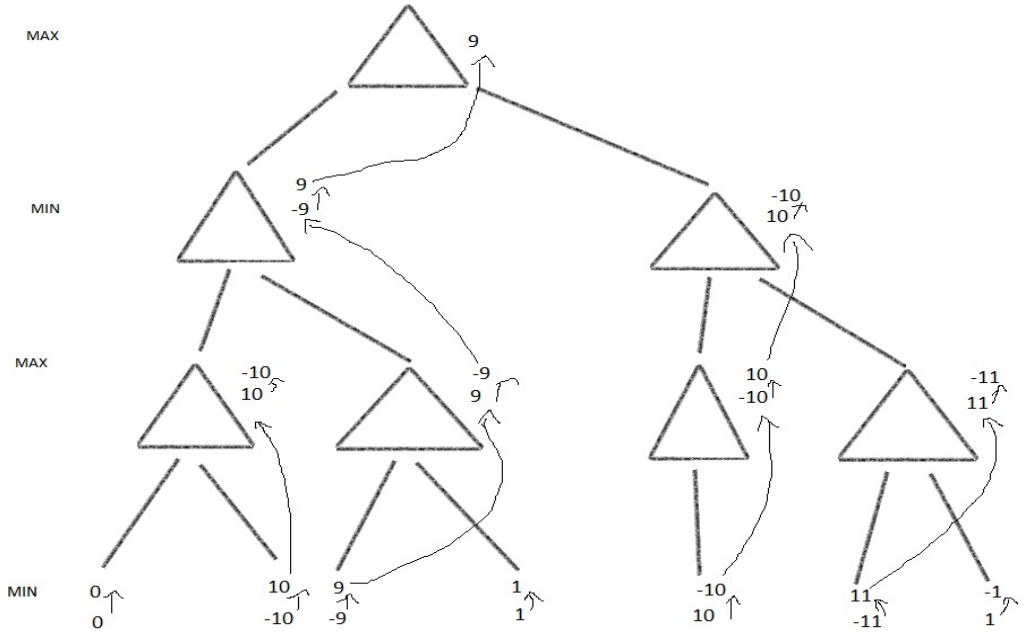
Profundidad 3 (ejemplo de profundidad impar):

(setq mi-posicion (list '(1 0 0 1 0 0 7) (reverse '(7 0 0 0 1 0 1))))
(setq estado (crea-estado-inicial 0 mi-posicion))
(minimax-1 estado 0 t 3 #'f-eval-snape)

Salida:

```
 ---- Gen.Sucesores de #2A((1 0 0 1 0 0 7) (1 0 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 3)  => Ultima ficha en 4, con 0, contra: 0
 ---- Gen.Sucesores de #2A((0 1 0 1 0 0 7) (1 0 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 2)  => Ultima ficha en 3, con 0, contra: 0
 ---- Gen.Sucesores de #2A((0 1 0 1 0 0 7) (0 1 1 0 0 0 7))
Juega (R 1)  => Ultima ficha en 2, con 0, contra: 0
Juega (R 3)  => Ultima ficha en 4, con 0, contra: 1 => Captura 1 y termina,
0
-10
 ---- Gen.Sucesores de #2A((0 1 0 1 0 0 7) (1 0 0 1 0 0 7))
Juega (R 1)  => Ultima ficha en 2, con 0, contra: 1 => Captura 1 y termina,
Juega (R 3)  => Ultima ficha en 4, con 0, contra: 0
-9
1
 ---- Gen.Sucesores de #2A((1 0 0 0 1 0 7) (1 0 1 0 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 1 => Captura 1 y termina,
Juega (R 2)  => Ultima ficha en 3, con 0, contra: 0
 ---- Gen.Sucesores de #2A((1 0 0 0 0 0 7) (0 0 1 0 0 0 9))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
10
 ---- Gen.Sucesores de #2A((1 0 0 0 1 0 7) (1 0 0 1 0 0 7))
Juega (R 0)  => Ultima ficha en 1, con 0, contra: 0
Juega (R 4)  => Ultima ficha en 5, con 0, contra: 1 => Captura 1 y termina,
1
-11
#S(ESTADO :TABLERO #2A((0 1 0 1 0 0 7) (1 0 1 0 0 0 7)) :LADO-SGTE-JUGADOR 1 :DEBE-PASAR-TURNO NIL :ACCION (R 0))
```

Mismo caso que el comentado anteriormente pero el análisis se propaga hasta profundidad 3.
Como la profundidad es impar los valores impresos se corresponden con el valor asociado a max del estado.
La acción tomada es el primer movimiento, que como vemos en el árbol es la acción que teóricamente se tomaría.



(b)
Hemos comentado la función minimax entregada:

```
(defun minimax (estado profundidad-max f-eval)
  (let* ((oldverb *verb*) ; Si la bandera de debug esta puesta modo verboso en la ejecucion de minimax. Si no modo vervoso desactivado
        (*verb* (if *debug-mmx* *verb* nil))
        (estado2 (minimax-1 estado 0 t profundidad-max f-eval)) ; Llamada a minimax desde la profundidad inicial 0 (modo verboso o no). Es decir
                                                                  empezamos la busqueda como tal.
        (*verb* oldverb)) ;Restablecemos el valor de *verb*.
    estado2))


(defun minimax-1 (estado profundidad devolver-movimiento profundidad-max f-eval)
  (cond ((>= profundidad profundidad-max) ;En caso de llegar a la profundidad maxima estabelcida. Paramos la busqueda y evaluamos el tablero.
        ;;Si la bandera devolver-movimiento esta a nil devolvemos el valor del estado del tablero segun nuestra heuristica.
        ;;En caso contrario la evalucion del estado.
         (unless devolver-movimiento  (funcall f-eval estado)))
        (t
         (let* ((sucesores (generar-sucesores estado)) ;Generamos todos los tableros sucesores y realizamos la busqueda
              (mejor-valor -99999) ;El valor de un estado es inicialmente para max -infinto y para min +inf
               (mejor-sucesor nil)) ;No hay inicialmente un mejor movimiento, pues todavia no se ha explorado
           (cond ((null sucesores) ;Si ya no quedan sucecesores en la lista hemos llegado al final de nuestra exploracion y evaluamos el estado.
                  (unless devolver-movimiento  (funcall f-eval estado)))
                 (t
                  ;; Para cada sucesor realizamos minimax.
                  (loop for sucesor in sucesores do
                    ;;Obtenemos el valor minimax de cada sucesor
                    (let* ((resultado-sucesor (minimax-1 sucesor (1+ profundidad)
                                                         nil profundidad-max f-eval))
                           ;; En los nodos hijos es el turno del contricante opuesto, busca minimizar la ganancia de su rival. max(x,y) = - min (-x, -y)
                           (valor-nuevo (- resultado-sucesor)))
                      (when (> valor-nuevo mejor-valor) ; Maximizamos el valor obtenido.
                        (setq mejor-valor valor-nuevo)
                        (setq mejor-sucesor  sucesor ))))
                  (if  devolver-movimiento mejor-sucesor mejor-valor)))))) ;Devolvemos el valor optimo o el sucesor que concude a ducho tablero con valor optimo.
```

(c)

En minimax el jugador max busca maximizar la función de utilidad y el jugador min minimizarla. Como este juego es de suma cero, el estado que tenga valor heurístico x para max tendrá para min valor heurístico -x.
Basándonos en esta expresión matemática:

Min(x,y) = -Max(-x, -y)

se implementa minimax para juegos de suma cero mediante la versión negamax.
De esta forma cambiando el signo del valor heurístico de un estado podemos simplificar el algoritmo maximizando en cada nodo el valor.
Las diferencias de ejecución para distintas profundidades ya están explicadas en el apartado a).
Mediante estos ejemplos se ve que no es necesario tener en cuenta la profundidad del estado en la heurística.

Apdo B2.2:

Nuestra función de minimax con poda alfa-beta es:

```
(defun minimax-a-b (estado profundidad-max f-eval)
   (let* ((oldverb *verb*) ; Si la bandera de debug esta puesta modo verboso en la ejecucion de minimax. Si no modo vervoso desactivado
      (*verb* (if *debug-mmx* *verb* nil))
      (estado2 (minimax-1-a-b estado 0 t profundidad-max -99999 99999 f-eval)) ; Llamada a minimax desde la profundidad inicial 0 (modo verboso
                                                                                o no). Es decir empezamos la busqueda como tal.
      (*verb* oldverb)) ;Restablecemos el valor de *verb*.
   estado2))


 (defun minimax-1-a-b (estado profundidad devolver-movimiento profundidad-max alfa beta f-eval)
  (cond ((>= profundidad profundidad-max) ;En caso de llegar a la profundidad maxima estabelcida. Paramos la busqueda y evaluamos el tablero.
       (unless devolver-movimiento  (funcall f-eval estado))) ;Si la bandera devolver-movimiento esta a nil devolvemos el valor del estado del tablero
segun nuestra heuristica.
       (t
       (let* ((sucesores (generar-sucesores estado))
            (mejor-sucesor nil))
         (cond ((null sucesores) ;Si ya no quedan sucecesores en la lista hemos llegado al final de nuestra exploracion y evaluamos el estado.
              (unless devolver-movimiento  (funcall f-eval estado)))
              (t
              (loop for sucesor in sucesores do     ;iteramos por cada sucesor generado
                (let* ((resultado-sucesor (minimax-1-a-b sucesor (1+ profundidad)  ;Calculamos su valor minimax
                            nil profundidad-max (- beta) (- alfa) f-eval))
                     (valor-nuevo (- resultado-sucesor)))   ;Y lo negamos (negamax)
                  (when (> valor-nuevo alfa)  ;Actualizamos el alfa al ser nodo max (todos lo son)
                    (setq alfa valor-nuevo)
                    (setq mejor-sucesor  sucesor))
                  (when ( <= beta alfa)       ;si alfa es mayor que beta, podamos (no exploramos las demas ramas de ese nodo)
                    (return (if  devolver-movimiento mejor-sucesor alfa)))))
              (if  devolver-movimiento mejor-sucesor alfa)))))))
```

Para probar nuestra función, hemos jugado varias partidas con jugadores con las mismas heurísticas, primero utilizando minimax sin poda, y después nuestra poda alfa-beta. Hemos comprobado que bajo las mismas condiciones iniciales y con heurísticas deterministas se obtenían los mismos resultados con ambas versiones.

Apdo B2.3:

Para una misma heurística para todos los jugadores, hemos ejecutado dos partidas, una con dos jugadores sin poda y otra con dos jugadores con poda. Estos son los resultados a profundidad 3:

Sin poda:

```
FIN DEL JUEGO por VICTORIA en 58 Jugadas
```

Marcador: JUGADOR-MINIMAX-1 17 - 19 JUGADOR-MINIMAX-2

; cpu time (non-gc) 497 msec user, 16 msec system
; cpu time (gc) 486 msec user, 0 msec system
; cpu time (total) 983 msec user, 16 msec system
; real time 996 msec

Con poda:

FIN DEL JUEGO por VICTORIA en 58 Jugadas
Marcador: JUGADOR-MINIMAX-2-A-B 17 - 19 JUGADOR-MINIMAX-1-A-B

; cpu time (non-gc) 388 msec user, 0 msec system
; cpu time (gc) 408 msec user, 0 msec system
; cpu time (total) 796 msec user, 0 msec system
; real time 795 msec

Y para profundidad 5:

Sin poda:

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-1 19 - 17 JUGADOR-MINIMAX-2

; cpu time (non-gc) 3,527 msec user, 0 msec system
; cpu time (gc) 4,460 msec user, 0 msec system
; cpu time (total) 7,987 msec user, 0 msec system
; real time 8,037 msec

Con poda:

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-2-A-B 19 - 17 JUGADOR-MINIMAX-1-A-B

; cpu time (non-gc) 1,423 msec user, 15 msec system
; cpu time (gc) 1,884 msec user, 0 msec system
; cpu time (total) 3,307 msec user, 15 msec system
; real time 3,352 msec

Se aprecia que la poda alfa-beta acelera notablemente el proceso. Esta diferencia de tiempo es más significativa cuanto mayor es la profundidad a la que se evalúa.

Apdo B2.4:

Hemos implementado dos versiones adicionales del minimax con poda, una ordena los nodos de mayor a menor de acuerdo a la estrategia y la otra de menor a mayor.
La poda alfa-beta es óptima cuando los mejores valores se exploran primero, por lo tanto la versión que ordena de mayor a menor debería dar mejores resultados de tiempo, y la otra versión debería ser similar, e incluso mayor por el tiempo de ordenación, a la versión sin poda de minimax en cuanto a tiempo de ejecución.

Para profundidad 3:

Poda normal

FIN DEL JUEGO por VICTORIA en 58 Jugadas
Marcador: JUGADOR-MINIMAX-1-A-B 17 - 19 JUGADOR-MINIMAX-2-A-B

; cpu time (non-gc) 343 msec user, 15 msec system
; cpu time (gc) 609 msec user, 0 msec system
; cpu time (total) 952 msec user, 15 msec system
; real time 968 msec

Poda y ordenación mayor a menor

FIN DEL JUEGO por VICTORIA en 59 Jugadas
Marcador: JUGADOR-MINIMAX-1-SORT 17 - 19 JUGADOR-MINIMAX-2-SORT

; cpu time (non-gc) 547 msec user, 0 msec system
; cpu time (gc) 452 msec user, 0 msec system
; cpu time (total) 999 msec user, 0 msec system
; real time 1,014 msec

Poda y ordenación de menor a mayor

FIN DEL JUEGO por VICTORIA en 58 Jugadas
Marcador: JUGADOR-MINIMAX-1-UNSORT 17 - 19 JUGADOR-MINIMAX-2-UNSORT

; cpu time (non-gc) 624 msec user, 0 msec system
; cpu time (gc) 967 msec user, 0 msec system
; cpu time (total) 1,591 msec user, 0 msec system
; real time 1,613 msec

Para esta profundidad pequeña, ordenar los nodos añade demasiado tiempo a la ejecución para compensar el ahorro de tiempo en la poda ya que no se generan excesivos nodos. La ordenación inversa añade aún más tiempo de ejecución.

Para profundidad 5:

Poda normal

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-1-A-B 19 - 17 JUGADOR-MINIMAX-2-A-B

; cpu time (non-gc) 1,701 msec user, 0 msec system
; cpu time (gc) 2,527 msec user, 0 msec system
; cpu time (total) 4,228 msec user, 0 msec system
; real time 4,255 msec

Poda y ordenación de mayor a menor

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-1-SORT 19 - 17 JUGADOR-MINIMAX-2-SORT

; cpu time (non-gc) 811 msec user, 0 msec system
; cpu time (gc) 1,436 msec user, 0 msec system
; cpu time (total) 2,247 msec user, 0 msec system
; real time 2,267 msec

Poda y ordenación de menor a mayor

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-1-UNSORT 19 - 17 JUGADOR-MINIMAX-2-UNSORT

; cpu time (non-gc) 5,230 msec user, 0 msec system
; cpu time (gc) 6,330 msec user, 0 msec system
; cpu time (total) 11,560 msec user, 0 msec system
; real time 11,702 msec

Sin poda

FIN DEL JUEGO por VICTORIA en 40 Jugadas
Marcador: JUGADOR-MINIMAX-1 19 - 17 JUGADOR-MINIMAX-2

; cpu time (non-gc) 4,104 msec user, 15 msec system
; cpu time (gc) 6,349 msec user, 0 msec system
; cpu time (total) 10,453 msec user, 15 msec system
; real time 10,536 msec

Se ve que el ahorro de tiempo de ejecución en este caso es muy significativo cuando los nodos se ordenan de manera que la poda sea más eficiente. Cuando se ordenan al contrario el tiempo aumenta con respecto a la poda sin ordenación, y de nuevo es mayor al tiempo del minimax sin poda, debido al tiempo de ordenación.

Versión de minimax con poda y ordenación beneficiosa:

```
(defun minimax-a-b-sort (estado profundidad-max f-eval)
  (let* ((oldverb *verb*) ; Si la bandera de debug esta puesta modo verboso en la ejecucion de minimax. Si no modo vervoso desactivado
      (*verb* (if *debug-mmx* *verb* nil))
      (estado2 (minimax-1-a-b-sort estado 0 t profundidad-max -99999 99999 f-eval)) ; Llamada a minimax desde la profundidad inicial 0 (modo verboso o no). Es decir empezamos la busqueda como tal.
      (*verb* oldverb)) ;Restablecemos el valor de *verb*.
  estado2))
```

```
(defun minimax-1-a-b-sort (estado profundidad devolver-movimiento profundidad-max alfa beta f-eval)
```

```
  (cond ((>= profundidad profundidad-max) ;En caso de llegar a la profundidad maxima estabelcida. Paramos la busqueda y evaluamos el tablero.
        (unless devolver-movimiento  (funcall f-eval estado))) ;Si la bandera devolver-movimiento esta a nil devolvemos el valor del estado del tablero
segun nuestra heuristica.
        (t
        (let* ((sucesores (sort (generar-sucesores estado) #'< :key f-eval))
              (mejor-sucesor nil))
          (cond ((null sucesores) ;Si ya no quedan sucecesores en la lista hemos llegado al final de nuestra exploracion y evaluamos el estado.
                (unless devolver-movimiento  (funcall f-eval estado)))
                (t
                (loop for sucesor in sucesores do     ;iteramos por cada sucesor generado
                  (let* ((resultado-sucesor (minimax-1-a-b-sort sucesor (1+ profundidad)  ;Calculamos su valor minimax
                                    nil profundidad-max (- beta) (- alfa) f-eval))
                        (valor-nuevo (- resultado-sucesor)))   ;Y lo negamos (negamax)
                    (when (> valor-nuevo alfa)  ;Actualizamos el alfa al ser nodo max (todos lo son)
                      (setq alfa valor-nuevo)
                      (setq mejor-sucesor  sucesor))
                    (when ( <= beta alfa)        ;si alfa es mayor que beta, podamos (no exploramos las demas ramas de ese nodo)
                      (return (if  devolver-movimiento mejor-sucesor alfa)))))
                (if  devolver-movimiento mejor-sucesor alfa)))))))
```

COMENTARIO: aunque el sentido de la desigualdad daría a pensar en que estamos ordenando de menor a mayor valor heurístico los nodos, debido al cambio de signo inherente a negamax, en realidad el orden es justo el opuesto.

Versión de minimax con poda y ordenación perjudicial:

```
(defun minimax-a-b-unsort (estado profundidad-max f-eval)
   (let* ((oldverb *verb*) ; Si la bandera de debug esta puesta modo verboso en la ejecucion de minimax. Si no modo vervoso desactivado
        (*verb* (if *debug-mmx* *verb* nil))
        (estado2 (minimax-1-a-b-unsort estado 0 t profundidad-max -99999 99999 f-eval)) ; Llamada a minimax desde la profundidad inicial 0 (modo
verboso o no). Es decir empezamos la busqueda como tal.
        (*verb* oldverb)) ;Restablecemos el valor de *verb*.
   estado2))


 (defun minimax-1-a-b-unsort (estado profundidad devolver-movimiento profundidad-max alfa beta f-eval)
  (cond ((>= profundidad profundidad-max) ;En caso de llegar a la profundidad maxima estabelcida. Paramos la busqueda y evaluamos el tablero.
        (unless devolver-movimiento  (funcall f-eval estado))) ;Si la bandera devolver-movimiento esta a nil devolvemos el valor del estado del tablero
segun nuestra heuristica.
        (t
        (let* ((sucesores (sort (generar-sucesores estado) #'> :key f-eval))
              (mejor-sucesor nil))
          (cond ((null sucesores) ;Si ya no quedan sucecesores en la lista hemos llegado al final de nuestra exploracion y evaluamos el estado.
                (unless devolver-movimiento  (funcall f-eval estado)))
                (t
                (loop for sucesor in sucesores do     ;iteramos por cada sucesor generado
                  (let* ((resultado-sucesor (minimax-1-a-b-unsort sucesor (1+ profundidad)  ;Calculamos su valor minimax
                                    nil profundidad-max (- beta) (- alfa) f-eval))
                        (valor-nuevo (- resultado-sucesor)))   ;Y lo negamos (negamax)
                    (when (> valor-nuevo alfa)  ;Actualizamos el alfa al ser nodo max (todos lo son)
                      (setq alfa valor-nuevo)
                      (setq mejor-sucesor  sucesor))
                    (when ( <= beta alfa)        ;si alfa es mayor que beta, podamos (no exploramos las demas ramas de ese nodo)
                      (return (if  devolver-movimiento mejor-sucesor alfa)))))
                (if  devolver-movimiento mejor-sucesor alfa)))))))
```

COMENTARIO:
Hemos implementado una función que compara automáticamente tiempos de partidas con jugadores con el mismo tipo de minimax (sin poda, con poda y poda con ordenación beneficiosa) para varias profundidades, e imprime el resultado en un fichero.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; estadistico
;;; funcion que imprime en un fichero "nombre" el tiempo en ciclos de reloj de las partidas con profundidad 1 hasta max-prof entre
;;; (i) jugadores normales sin poda
;;; (ii) jugadores con poda
;;; (iii) jugadores con poda y ordenacion
;;;
;;; Todos los jugadores usan la misma heuristica y estan definidos antes
;;;
```

```
;;; Cuestiones de diseno:
;;;  Se podria haber creado a los jugadores dentro del cuerpo de la funcion pero como en el fondo todo este fichero es de prueba y conclusiones
;;;  sobre el algoritmo minimax pensamos que no es execivamente importante crear una funcion autonoma sino las conclusiones que se pueden sacar
;;;  de los resultados.
;;;
;;; Foramto de impresion:
;;;  profundidad i : tiempo-partida-sin-poda tiempo-partida-con-poda tiempo-partida-con-poda-y-ordenacion
;;;
;;; Objetivo ver como evoluciona el algoritmo con poda y la ordenacion de los sucesores con la profundidad
;;;

(defun estadistico( fichero max-prof)

  (setf path (make-pathname :name fichero))
  (setf str (open path :direction :output
                       :if-exists :supersede))
  (setf start 1)
  (do ((i start (1+ i )))
    ((> i max-prof))
      ;; Tiempo sin poda
      (setf before (get-internal-run-time))
      (partida 0 i (list *mi-jugador-pruebas-1* *mi-jugador-pruebas-2*))
      (setf time1 (- (get-internal-run-time)  before))
      ;; Tiempo con poda sin ordenacion
      (setf before (get-internal-run-time))
      (partida 0 i (list *mi-jugador-pruebas-a-b-1* *mi-jugador-pruebas-a-b-2*))
      (setf time2 (- (get-internal-run-time)  before))
      ;; Tiempo con poda y ordenacion
      (setf before (get-internal-run-time))
      (partida 0 i (list *mi-jugador-pruebas-a-b-1-sort* *mi-jugador-pruebas-a-b-2-sort*))
      (setf time3 (- (get-internal-run-time)  before))
      (format str "Profundidad ~D: ~D ~D ~D ~%" i time1 time2 time3))
      (close str))

;; Profundidades muy grandes pueden generar tiempos de espera muy grandes incluso caidas del sistema. La busqueda es un problema exponencial.
;(estadistico "estadisticas" 5)
```

Bibliografia:

Inteligencia artificial: fundamentos, práctica y aplicaciones
Capt 5 Juegos pag 151-164 Autor: Alberto García

Russel, S. and Norvig, P. Imperfect Real-Time Decisions. In Artificial Intelligence: A Modern Approach, Chapter 6, pp. 171-175. New Jersey, USA, 2003.

Chris Gifford, James Bley, Dayo Ajayi, and Zach Thompson.
Searching & Game Playing: An Artificial Intelligence Approach to Mancala
https://www.ittc.ku.edu/publications/documents/Gifford_ITTC-FY2009-TR-03050-03.pdf