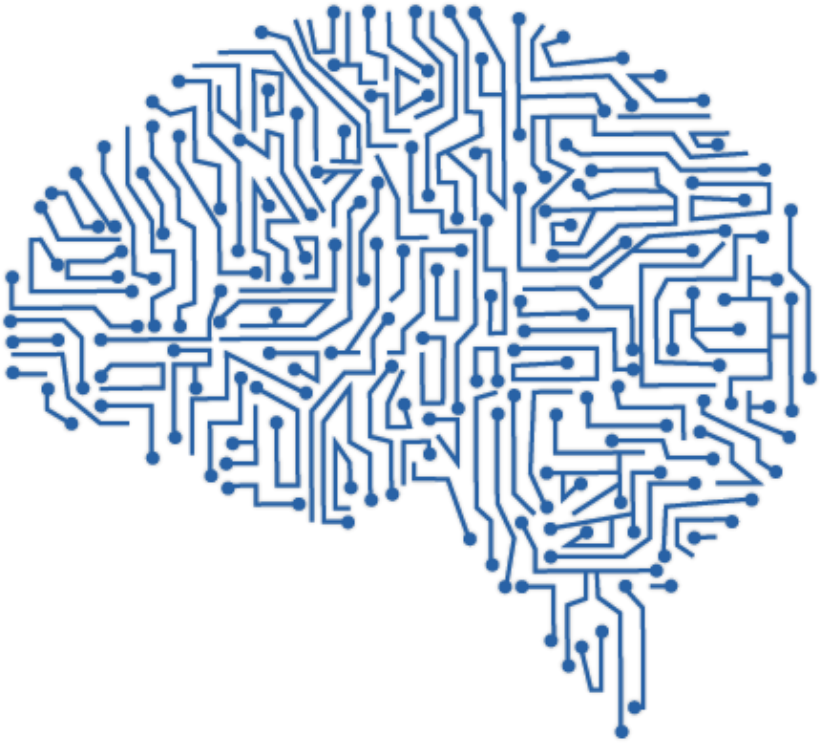
Práctica 4

Inteligencia Artificial



Alfonso Bonilla Trueba – David García Fernández
PAREJA 4 | GRUPO 2313 | EPS-UAM

PARTE C: Preguntas sobre el código entregado

Pregunta C1.

Explique el código negamax-a-b entregado: a. Utilice ejemplos de evaluaciones que ilustren para qué sirve cada función. b. Comente línea a línea las funciones implicadas en la implementación del algoritmo negamax.

```
(defun negamax-alfa-beta (estado profundidad devolver-movimiento profundidad-max f-eval alfa-valor alfa-
sucesor)
 (cond ((>= profundidad profundidad-max)
        (unless devolver-movimiento  (funcall f-eval estado)))
       (t
        (let ((sucesores (generar-sucesores estado profundidad))
              (mejor-valor +min-val+)
              (mejor-sucesor nil))
          (cond ((null sucesores)
                 (unless devolver-movimiento  (funcall f-eval estado)))
                (t
                 (loop for sucesor in sucesores do
                   (let* ((result-sucesor (- (negamax-alfa-beta sucesor (1+ profundidad)
                                             nil profundidad-max f-eval (- alfa-valor) alfa-sucesor))))
                     ;(format t "~% Nmx-1 Prof:~A result-suc ~3A de suc ~A, mejor=~A" profundidad result-
sucesor (estado-tablero sucesor) mejor-valor)
                     (when (>= (- alfa-valor) (- result-sucesor))
                       (return
                         (if devolver-movimiento alfa-sucesor alfa-valor)))
                     (when (> result-sucesor mejor-valor)
                       (setq mejor-valor result-sucesor)
                       (setq mejor-sucesor  sucesor)
                       (setq alfa-valor result-sucesor)
                       (setq alfa-sucesor  sucesor))))
                 (if  devolver-movimiento mejor-sucesor mejor-valor)))))))
```

Tomado como base el código de Moodle, hemos añadido dos parámetros para guardar el valor alfa y el sucesor alfa, de esta manera conseguimos tener sus valores en cualquier momento de la recursión. Estos valores se inicializarán en la primera iteración del primer bucle que se ejecute. Si en sucesivas iteraciones, incluidas las de otras iteraciones de otras instancias de la función, se localiza un valor que es menor que el valor alfa en un nivel superior (ambos valores negados por ser el algoritmo negamax), quiere decir que el nivel superior al actual no va a coger dicho valor por ser peor y que el actual solo va a poder elegir, a partir de ahí, valores mejores para él y peores para el jugador del nivel superior. Por esta razón, al no poderse encontrar información útil, se 'poda' el nodo devolviendo la información del nodo alfa en un return dentro del loop. La información del nodo alfa se actualizará junto a la del mejor-sucesor y mejor-valor del código base.

Nota: En el algoritmo negamax no tenemos dos jugadores max y min que intenten maximizar y minimar sus posibilidades, si no que tenemos jugadores max que van maximizando las suyas propias, por eso no hemos encontrado la necesidad de usar una variable adicional para el valor de beta.

Pregunta C2.

Compare el tiempo que tarda un jugador utilizando negamax y utilizando negamax con poda alfa-beta. Comente los resultados.

Como podemos observar el tiempo empleado por negamax con poda alfa beta es notablemente inferior al negamax normal, esto es debido a que la poda evita tener que generar los sucesores de determinados nodos, ahorrándose todos los cálculos que se tendrían que hacer sobre cada uno de ellos.

Esta prueba la hemos realizado usando al jugador bueno contra sí mismo con una profundidad de 2.

Usando el algoritmo negamax sin poda obtenemos lo siguiente:
; cpu time (non-gc) 987 msec user, 344 msec system
; cpu time (gc) 685 msec user, 0 msec system
; cpu time (total) 1,672 msec user, 344 msec system
; real time 2,072 msec

Usando el algoritmo negamax con poda alfa-beta obtenemos:
; cpu time (non-gc) 861 msec user, 328 msec system
; cpu time (gc) 717 msec user, 0 msec system
; cpu time (total) 1,578 msec user, 328 msec system
; real time 1,914 msec

Se puede observar que, para la misma partida con los mismos jugadores y la misma profundidad, pero con distinto algoritmo, el tiempo varía. El algoritmo con poda alfa-beta tarda menos en ejecutarse que el algoritmo sin poda.

(987+344) / (861+328) = 1.189

Por tanto, el algoritmo con poda alfa-beta es un 18.9% mejor que el algoritmo sin poda.

Pregunta C3.

Modifique el orden el que se exploran las jugadas. Comente el efecto que tiene en la poda alfa-beta modificar dicho orden.

El efecto que tiene en la poda alfa beta es dependiente del árbol generado, es decir, si se ejecuta el mismo algoritmo en orden normal y en orden inverso será más eficiente aquel que encuentre antes los valores que permitan la poda del resto de los subárboles.

Pregunta C4.

Explique los fundamentos, el razonamiento seguido, la bibliografía utilizada y las pruebas realizadas para la entrega de la/s función/es de evaluación entregada/s en el torneo.

Jugador 1

- Código:

```
(defun mi-f-ev (estado)
  (if (juego-terminado-p estado)
      (if (> (cuenta-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 0) 18)
          10
        -10)

        (* 10 (- (get-fichas (estado-tablero estado)(estado-lado-sgte-jugador estado) *long-fila*)
                 (get-fichas (estado-tablero estado)(lado-contrario (estado-lado-sgte-jugador estado))
                                                                    *long-fila*)))))
```

- Fundamentos-Razonamientos:
  La heurística diferencia dos casos, final de partida o juego normal. Si es el final de partida y el número de fichas de nuestro jugador es superior lo valorara positivamente en una magnitud de 10, si no, lo valorara en -10. Por otro lado, si no es fin de partida, calculará quien tiene más fichas en la última

posición de la fila, dando mejores resultados según el número de ventaja que tenga nuestro jugador sobre el oponente.

- Pruebas: Evaluación de la heurística jugando contra los oponentes aportados en el código base y con los jugadores anteriormente codificados.

Jugador 2: **ElButanero**

- Código:

```
(defun mi-f-ev (estado)
  (- (+ (get-pts (estado-lado-sgte-jugador estado))
        (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 6))
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 4)4)
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 5)5)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 4)-2)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 5)-4)
     (+ (get-pts (lado-contrario (estado-lado-sgte-jugador estado)))
        (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 6))))
```

- Fundamentos-Razonamientos
  La heurística utilizada consiste en crear como punto de referencia positivo la suma de la puntuación del jugador más el número de fichas en el kalaha, estando este número multiplicado por seis para aumentar su importancia. A esta suma de referencia se le va a restar el mismo dato, pero del oponente, de esta manera tratamos de maximizar que nosotros tengamos más puntos y fichas en el kalaha que el oponente. Además, también se sumará el número de fichas que tenga el oponente en las posiciones 4 y 5 y se restaran las del jugador en las mismas posiciones facilitando las jugadas en las que el oponente tenga fichas que podamos adquirir cerca de su kalaha y nuestro jugador no tenga ninguna. Estos datos se multiplicarán por un factor menor al de las fichas del kalaha por considerarlo menos importante (el factor de multiplicación dependerá de su posición y del jugador).

- Pruebas
  Evaluación de la heurística jugando contra los oponentes aportados en el código base y con los jugadores anteriormente codificados.

Jugador 3: **DavAlfPRO**

- Código

```
(defun evaluacion (estado)
  (- (+ (get-pts (estado-lado-sgte-jugador estado))
        (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 6))
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 4)4)
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 5)5)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 4)-2)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 5)-4)
     (+ (get-pts (lado-contrario (estado-lado-sgte-jugador estado)))
        (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 6))))

(defun suma (estado lista)
  (mapcar #'(lambda(x)
              (- (evaluacion estado)
                 x))lista))

(defun mi-f-ev (estado)
  (if (juego-terminado-p estado)
```

```
  -50                              ;; Condicion especial de juego terminado
;; Devuelve el maximo del numero de fichas del lado enemigo menos el numero de propias
(max-list (suma estado
                (mapcar #'(lambda(x)
                            (evaluacion x))
                  (generar-sucesores estado))))))
```

- Fundamentos-Razonamientos:
  Este jugador utiliza la misma heurística que el jugador 'ElButanero' pero tiene en cuenta un nivel más de profundidad.

- Pruebas:
  Evaluación de la heurística jugando contra los oponentes aportados en el código base y con los jugadores anteriormente codificados.

Pregunta C5.

El jugador llamado bueno sólo se distingue del regular en que expande un nivel más. ¿Por qué motivo no es capaz de ganar al regular? Sugiera y pruebe alguna solución que remedie este problema

El jugador bueno lo que hace es realizar una heurística sobre el siguiente movimiento al actual, es decir, que escoge un camino en el árbol para obtener un beneficio en un turno posterior al actual. Además, tal y como esta codificado el algoritmo negamax, presupone que su oponente realiza la misma estrategia cuando no es ni mucho menos cierto, ya que el jugador regular escoge una estrategia a corto plazo, desbaratando así las previsiones del jugador bueno en cada turno, a la vez que maximiza su ganancia.

Una solución para el jugador bueno, sería añadir a la heurística el valor del camino recorrido hasta el momento, para que tenga en cuenta las fases que llevan hasta su objetivo y sepa valorar cual es el camino más adecuado para la victoria a la vez que mejora su nivel de predicción sobre el oponente.

Código sugerido:

```
(defun evaluacion (estado)
  (- (+ (get-pts (estado-lado-sgte-jugador estado))
        (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 6))
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 4)4)
     (* (get-fichas (estado-tablero estado) (estado-lado-sgte-jugador estado) 5)5)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 4)-2)
     (* (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 5)-4)
     (+ (get-pts (lado-contrario (estado-lado-sgte-jugador estado)))
        (get-fichas (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado)) 6))))

(defun suma (estado lista)
  (mapcar #'(lambda(x)
              (- (evaluacion estado)
                 x))lista))

(defun mi-f-ev (estado)
  (if (juego-terminado-p estado)
      -50                              ;; Condicion especial de juego terminado
    ;; Devuelve el maximo del numero de fichas del lado enemigo menos el numero de propias
    (max-list (suma estado
                    (mapcar #'(lambda(x)
                                (evaluacion x))
                      (generar-sucesores estado))))))
```