Práctica 4 Inteligencia Artificial

Pareja 09 (Lucía Asencio y Juan Riera)

Mayo 2018

1 Preguntas sobre el código entregado

`Pregunta C1.` 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.

El razonamiento a seguir para crear la función de evaluación ha utilizado conceptos del aprendizaje automático similar al utilizado para entrenar redes neuronales, mezclados con otras estrategias que surgen naturalmente del conocimiento de las reglas del juego. Un ejemplo de esto es que nuestra heurística es capaz de medir si el jugador puede realizar una captura de fichas del oponente, sin embargo el valor devuelto en tal caso es un parámetro variable.

Así, más en detalle, nuestra fucnión de evaluación es la combinación de tres estrategias distintas. La primera es una simple multiplicación del vector de numeros de semillas en cada hoyo por un vector de ponderaciones. La segunda consiste en detectar en cada hoyo si al sembrar desde él se llega al kalaha, si se pasa, si no se llega, o si se pueden capturar fichas. En cualquiera de estos casos de devuelve un valor también variable que se toma de un vector de parámetros. Se hace lo mismo con el lado del oponente. La tercera es el resultado de sumar todas las fichas del lado del jugador y restarle la suma de las fichas del lado del oponente, y multiplicar el resultado por un peso. Por otro lado la heurística hace una comprobación de si es un estado de final de juego y comprueba quién es el ganador. Si es el jugador devuelve un valor muy elevado, y uno muy negativo si el ganador es el oponente

En cuanto a las pruebas realizadas para hallar los valores de todos los parámetros mencionados en el párrafo anterior, hemos procurado no ceñirnos a una única estrategia para que nuestro jugador sea lo más general posible. Por ello hemos utilizado scripts en bash y python que lo enfrentaban contra los jugadores aleatorios proporcionados en el enunciado, o contra jugadores aleatorios generados por nosotros con la misma estrategia anterior. Hemos así organizado torneos locales entre nuestros jugadores y hemos ido seleccionando.

`Pregunta C2.` 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.

Lo cierto es que el jugador bueno tiene un punto flaco, que se aprecia cuando se acerca a un estado de victoria. Si el estado de victoria está a profundidad dos con respecto al estado actual, el jugador regular evaluará el estado por la diferencia en el número de fichas de cada jugador, dando como resultado un número positivo si es un estado de victoria y uno negativo si es un estado de derrota. Sin embargo, el jugador bueno intenta expandir un nodo con la función $generar - sucesores$, que si recibe un estado de final de partida devuelve nil.

Al devolver nil, si bueno evaluara el mapcar que tiene en su código, lo haría sobre una lista vacía y devolvería el valor 0, siendo un estado crucial en la partida, ya que es el de derrota o de victoria.

Sin embargo esto nunca llega a ocurrir, porque la heurística de bueno comprueba antes de ejecutar el mapcar si el estado es de final de partida y devuelve -50 en tal caso. Esto supone un error grave, ya que logra que bueno evite los estados de final de juego, mientras que regular no distingue si un estado es de final de juego o no, sino

si es favorable para él.

Una mejora cosiderable se alcanza si se introduce que en el caso de ser un estado de final de juego, en vez de devolver -50, se compruebe si el jugador sale victorioso y, en tal caso devolver un número muy alto (del orden de 10000), o devolver un número muy bajo en otro caso (del orden de -10000). Esta mejora es suficiente para ganar al regular. Así, la heurística de Bueno quedaría así:

```
(defun f-eval-Bueno (estado)
  (if (juego-terminado-p estado)
      (if
        (<
          (suma-fila (estado-tablero estado)
                     (estado-lado-sgte-jugador estado))
        (suma-fila (estado-tablero estado)
                     (lado-contrario (estado-lado-sgte-jugador estado))))
        -10000
        10000)                          ;; Condicion especial de juego terminado
    ;; Devuelve el maximo del numero de fichas del lado enemigo menos el numero de propias
    (max-list (mapcar #'(lambda(x)
                          (- (suma-fila (estado-tablero x) (lado-contrario (estado-lado-sgte-jugador x)))
                             (suma-fila (estado-tablero x) (estado-lado-sgte-jugador x))))
                (generar-sucesores estado)))))
```

2 Evaluación de la calidad de la Función de Evaluación

Para una mejor compresión del código, conviene dividirlo en secciones que se suceden entre sí.

Heurística y función f-eval-ponderation

$F - eval - ponderation$ es la función principal que se encarga de llamar a las demás, y que es llamada con los argumentos correspondientes, por *heuristica*. Suma las ponderaciones de cada lado (explicadas más en detalle un poco más adelante), con el resultado de f-eval-ponderation-2, con una comprobación de si gana o pierde en este turno (que toma el valor 1000 si gana,-1000 si pierde y 0 si no es un estado de final de partida), con el resultado de restarle al número de fichas de mi lado el número de fichas del lado del oponente.

```
   (defun heuristica (estado)
    (f-eval-ponderation estado '((0 -68 -4 16 112 108) (52 96 24 -4 -60 -100))
    '((-104 72 80 -8) (-80 76 -56 56)))) ; funcion de evaluacion heuristica a implementar

(defvar *alias* '|habia_una_vez|) ; alias que aparecera en el ranking

(defun f-eval-ponderation (estado ponderations parameters)
    (+ (ponderate 0 (first ponderations)
          (lado-contrario (estado-lado-sgte-jugador estado))
          (estado-tablero estado))
       (ponderate 0 (second ponderations)
          (estado-lado-sgte-jugador estado)
```

```
        (estado-tablero estado))
  (if  (juego-terminado-p estado)
     (if
        (< (suma-fila
              (estado-tablero estado)
              (estado-lado-sgte-jugador estado))
            (suma-fila
              (estado-tablero estado)
              (lado-contrario (estado-lado-sgte-jugador estado))))
      -1000
      1000)
     0)
       (f-eval-ponderation-2 estado parameters)
   (- (suma-fila (estado-tablero estado) (estado-lado-sgte-jugador estado))
    (suma-fila (estado-tablero estado) (lado-contrario (estado-lado-sgte-jugador estado))))))
```

Función ponderate

Esta función multiplica cada elemento de la lista de números 'ponderation' recibida como segundo argumento por el número de fichas de cada hoyo del lado recibido como tercer argumento en el tablero recibido como segundo argumento. Para utilizarla de forma recursiva se ha añadido el primer argumento, que indica la posición actual. Así, en realidad suma el resultado de multiplicar el hoyo con la posicion indicada por el primer elemento de la ponderación, con el resultado de continuar la recursión con el resto de las ponderaciones y con la posicion siguieinte.

```
    (defun ponderate (position ponderation lado tablero)
   (if (null ponderation)
       0
       (+ (* (first ponderation)
           (get-fichas tablero lado position))
           (ponderate (+ position 1)
               (rest ponderation)
               lado tablero))))
```

Función f-eval-ponderation-2

Recibe como argumentos un estado y una lista que contiene dos listas de cuatro números cada una. Suma el resultado de llamar a calc-ponderations con cada una de estas listas y cada lado, con una comprobación de si es un estado final de la partida, esta comprobación toma el valor 10000 si es un estado de victoria, y -10000 si es uno de derrota.

```
 (defun f-eval-ponderation-2 (estado parameters)
   (+
     (apply
       '+
       (calc-ponderations
         (estado-tablero estado)
         (first parameters)
```

```
      (estado-lado-sgte-jugador estado)
      0))
    (apply
      '+
      (calc-ponderations
        (estado-tablero estado)
        (second parameters)
        (lado-contrario (estado-lado-sgte-jugador estado))
        0))
    (if
      (juego-terminado-p estado)
      (if
        (<
          (suma-fila (estado-tablero estado)
                     (estado-lado-sgte-jugador estado))
          (suma-fila (estado-tablero estado)
                     (lado-contrario (estado-lado-sgte-jugador estado))))
        -10000
        10000)
        0)))
```

Función calc-ponderations

Esta función recibe una lista llamada parameters, un tablero, un lado y una posición, y de forma recursiva recorre los hoyos del lado. Para cada hoyo devuelve el primer elemento de parameters si al sembrar desde ese hoyo nos pasaríamos del kalaha, devuelve el segundo elemento si nos quedáramos cortos, el tercero si llegaramos de forma exacta (repitiendo turno), y si podemos capturar fichas, el cuarto parámetro multiplicado por el número de fichas que se podrían capturar.

```
(defun calc-ponderations (tablero parameters side posicion)
  (if
    (equal posicion 6)        ; Si ya hemos ponderado las 5 posiciones, terminamos lista
    ()
    (cons                     ; Si no, devolvemos un cons de :
      (calc-ponderation       ; La ponderacion correspondiente a ese lado y posicion
        tablero
        parameters
        side
        posicion)
      (calc-ponderations      ; Y la lista ponderacion correspondiente a demas posiciciones
        tablero
        parameters
        side
        (+ 1 posicion)))))

(defun calc-ponderation (tablero parameters side posicion)
  (let
    ((num-fichas (get-fichas tablero side posicion)))
    (cond
```

```
;Si podemos efectuar un robo 4to coeficiente * numero de fichas robables
((and (< num-fichas (- 6 posicion))
      (eql (get-fichas tablero side (+ posicion num-fichas)) 0))

      (* (get-fichas tablero (lado-contrario side) (- 5 (+ posicion num-fichas)))
         (fourth parameters)))
; Si numfichas > 6 - posicion, 1er coeficiente
((> num-fichas (- 6 posicion))
 (first parameters))
; Si numfichas < 6 - posicion, 2o coeficiente
((< num-fichas (- 6 posicion))
 (second parameters))
; Si numfichas = 6 - posicion, 3er coeficiente (maxima ganancia/perdida)
(t
 (third parameters)))))
```

Esta es la heurística que más éxito ha tenido de las que hemos entregado. Volviendo a la explicación de la pregunta C1, se aprecia que tiene como vector de ponderaciones $((0-68-416112108)(529624-4-60-100))$, como vector de parámetros $((-1047280-8)(-8076-5656))$ y no aparece el peso mencionado, que se multiplicaba por el resultado de restar las fichas del lado del jugador oponente a las del actual. No aparece porque no logramos encontrar jugadores con el mismo éxito con las heurísticas que utilizaban tal estrategia. Así se podria decir que el peso es 1.

En cuanto a la calidad de la heurística en términos de eficiencia, a pesar de tardar menos que el jugador bueno proporcionado en el enunciado, es una heurística que posiblemente se podría optimizar, no lo hemos hecho porque ello acarrearía una menor legibilidad del código, y reduciría la modularización que nos ha permitido grandes avances.

Por otro lado el jugador parece bastante eficaz en su cometido, al haber logrado un 0.69 en el ránking, a pesar de no ser tan eficaz como otros. También cabe destacar el elevado número de empates que ha obtenido en relación a otros jugadores.