Inteligencia Artificial

PL2

Tema: Kalah (juego)

Integrantes:

Ana Cortés Cercadillo

Carlos Javier Hellín Asensio

Daniel Ferreiro Rodríguez

Curso: 2020-2021

Índice

Enunciado

Mancala es un juego de mesa ancestral, cuyas raíces datan aparentemente del antiguo Egipto, y que aún hoy se juega, especialmente en África y Asia. Existen múltiples variantes de las reglas de este juego, aquí utilizaremos las reglas de la variante Kalah, que se indican en los siguientes enlaces:

• https://brainking.com/es/GameRules?tp=103
• https://www.thesprucecrafts.com/how-to-play-mancala-409424

A continuación se resumen las reglas del juego:

• Movimiento de un jugador: Cada jugador se sitúa a un lado del tablero, controlando la fila de 6 agujeros que esté a su lado. Cada turno, el jugador selecciona un agujero que tenga al menos una ficha de entre los 6 agujeros de su lado, cogiendo las semillas que haya en él (dejándolo vacío) y distribuyéndolas una a una en los siguientes agujeros, así como en la casa del jugador, en sentido antihorario.
• Casa del oponente: Si al ir colocando las semillas se llegase a la casa del oponente, esta se saltará, ya que un jugador no puede colocar semillas en la casa del oponente.
• Turno extra: Si al realizar un movimiento, el último agujero en el que se coloca una semilla es la casa del jugador, este ganará un turno extra seguidamente, pudiendo realizar un movimiento adicional.
• Robo: Si al realizar un movimiento, el último agujero en el que se coloca una semilla estaba vacío y pertenece a la fila del propio jugador, este realizará un robo del agujero opuesto en la fila del oponente, llevando todas las semillas que hubiese en dicho agujero directamente a la casa del jugador.
• Fin de partida: Si un jugador no puede realizar un movimiento porque todos los agujeros de su fila están vacíos, la partida finalizará. Al finalizar, todas las semillas que queden en la fila del oponente se moverán inmediatamente a la casa del oponente. Ganará el jugador que más semillas tenga en su casa.

Retos realizados

Los 5 retos se han desarrollado **completamente** ajustándose a las especificaciones pedidas en esta práctica.

¿Cómo abordamos el problema?

1. Leímos la práctica en repetidas ocasiones para tener claro lo que se nos pedía.
2. Antes de programar nada, estudiamos a fondo el funcionamiento en el papel la estructura del Kalah, la casa y los agujeros del jugador1 y jugador2 etc.
3. Nos remitimos a la bibliografía recomendada para estudiar las heurísticas a implementar.
4. Realizamos algunas partidas entre nosotros en un juego online para comprender las reglas del juego.
5. Puesta en marcha. ¡**A programar**!

Implementación

Robo

En el **robo** hemos implementado las reglas del Kalah. Si el agujero donde se va a depositar la última semilla de ese movimiento está vacío y tiene semillas el agujero de enfrente de este, se procede al robo realizando este cálculo:

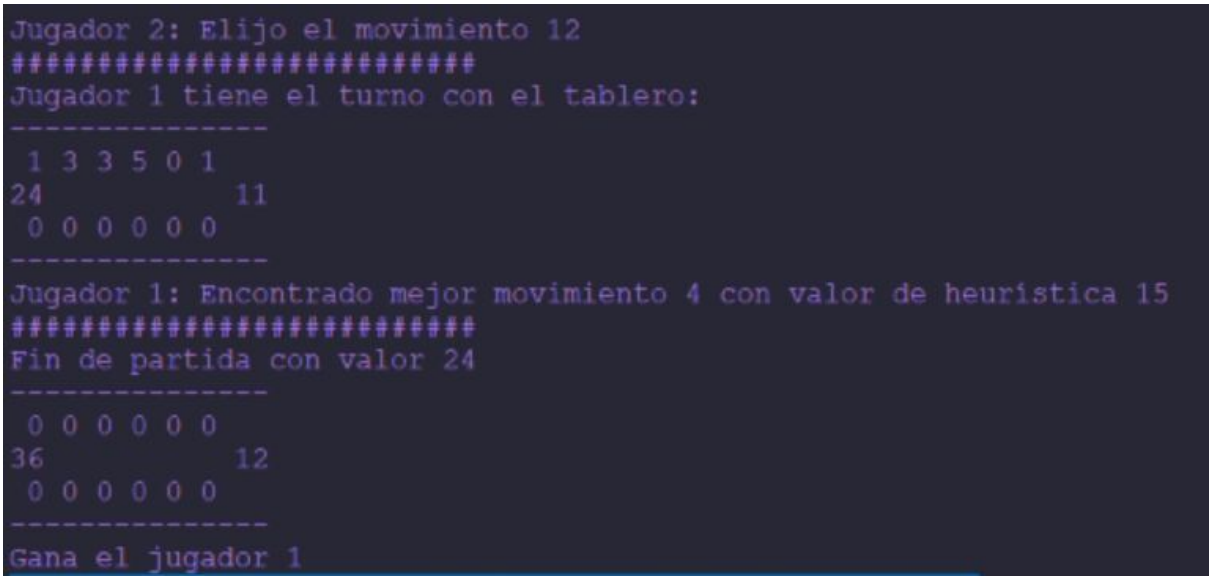
T = VC + AF + AC

T = Valor de la casa del jugador que roba
VC = Número de semillas que tiene la casa del jugador que roba.
AF = Una semilla (la que debió estar en el último agujero del movimiento).
AC = Número de semillas del agujero del jugador contrario, dicho agujero se pone 0.

Mejora en el tiempo

Si un jugador tiene en su casa más de 24 semillas gana la partida automáticamente ya que su oponente no podrá alcanzar una cifra de semillas mayor.

Resultado



Algoritmo minimax y alpha-beta

Hemos implementado el **minimax** por el libro “Artificial Intelligence: A Modern Approach ” página 166.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

La función **alpha-beta:**

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Argmax

En la función estrategia-minimax se usa **argmax** de Racket para devolver el primer elemento de la lista que maximiza el resultado del **proc**.

La heurística simple

Si es el turno del jugador1, la cantidad de semillas de su casa se restan con las de la casa del jugador2.
Si fue el turno del jugador2, se invertirá el orden de las casas en la resta.

Turno extra

Para que un jugador pudiera hacer **un turno extra** en el algoritmo minimax se le resta la profundidad máxima.

Lambda

La función jugar recibe las funciones de estrategia como parámetro y se invoca:

```
(define (jugar tablero jugador-inicial estrategia-jugador1 estrategia-jugador2 debug)
  (let (
        [posiciones (obtener-posiciones (list) 0 (t-agujeros tablero jugador-inicial) jugador-inicial)]
        )
    (if (or (> (t-casa-semillasj1 tablero) 24) (> (t-casa-semillasj2 tablero) 24) (empty? posiciones))
        (resultado tablero jugador-inicial debug)
        (let* (
               [posicion (estrategia-jugador1 tablero posiciones jugador-inicial debug)]
               [semillas (t-semillas tablero posicion)]
               [movimiento (mover (coger-semillas tablero posicion) (+ posicion 1) semillas jugador-inicial)]
               [tablero-final (car movimiento)]
               [posicion-final (cdr movimiento)]
               )
          (if (= posicion-final (casa jugador-inicial))
              (jugar tablero-final jugador-inicial estrategia-jugador1 estrategia-jugador2 debug)
              (jugar tablero-final (siguiente-jugador jugador-inicial) estrategia-jugador2 estrategia-jugador1 debug)
          )
        )
    )
  )
)
```

La profundidad y la heurística van incorporadas en la propia función de estrategia minimax mediante funciones lambda:

```
(define (estrategia funcion profundidad heuristica)
  (lambda (tablero jugador-inicial posiciones debug)
    (funcion tablero jugador-inicial posiciones heuristica profundidad debug)
  )
)
```

Después, llamamos a jugar de la siguiente manera:

```
(define (prueba-random)
  (displayln "Comienza el jugador 1, ambos con estrategia random")
  (jugar tablero 1 estrategia-random estrategia-random 1)
)
(prueba-random)
```

Poda alfa-beta

Para calcular el tiempo que tarda la simulación de prueba-multi-minimax-random se usó la función **current-milliseconds**.
En la función de la poda se le pasó como parámetro un booleano para analizar los resultados, si es true se poda y si es false no se podará.

Resultado

Sin poda:

```
Se jugarán 100 con el jugador 1 usando minimax con profundidad
4 y el jugador 2 usando una estrategia aleatoria, la mitad empezando el jugador 1 y la otra mitad empezando el jugador 2
Tras 100, el jugador 1 ha ganado 18, ha empatado 1 y ha perdido 81
Tiempo 7890
>
```

Con poda:

```
Se jugarán 100 con el jugador 1 usando minimax con profundidad
4 y el jugador 2 usando una estrategia aleatoria, la mitad empezando el jugador 1 y la otra mitad empezando el jugador 2
Tras 100, el jugador 1 ha ganado 26, ha empatado 2 y ha perdido 72
Tiempo 2424
>
```

La poda alfa beta es una técnica de búsqueda que reduce el número de nodos evaluados en un árbol de juego por el algoritmo minimax, por este motivo se puede observar que la simulación con poda tarda menos la ejecución.

Resultados entre la heurística simple y la compleja

Resultado

Heurística simple

```
Se jugarán 1000 con el jugador 1 usando minimax con profundidad
1 y el jugador 2 usando una estrategia aleatoria, la mitad empezando el jugador 1 y la otra mitad empezando el jugador 2
Tras 1000, el jugador 1 ha ganado 942, ha empatado 12 y ha perdido 46
```

Heurística compleja

```
Se jugarán 1000 con el jugador 1 usando minimax con profundidad
1 y el jugador 2 usando una estrategia aleatoria, la mitad empezando el jugador 1 y la otra mitad empezando el jugador 2
Tras 1000, el jugador 1 ha ganado 984, ha empatado 5 y ha perdido 11
```

Como se puede apreciar con la heurística compleja se obtiene una mejora sustancial con respecto a la heurística simple ya que se logra hacer movimientos más interesantes como conseguir un turno extra, ya que la heurística compleja usa la heurística simple con un peso de 0.4, otra heurística para prevenir robos con un peso de -0.1 y por último, la del turno extra con un peso de 0.5.

Problemas encontrados

Reto 1

Robo indebido: Cuando el agujero de enfrente estaba vacío y el agujero donde se deposita la última semilla de ese movimiento estaba vacío, se robaba sumando 1 a la casa del jugador correspondiente y dejando a 0 dicho dicho agujero.
Turno extra no realizado: Cuando la última semilla de un movimiento del jugador2 terminaba en su casa, pasaba el turno al jugador1 cuando debía tener este un turno extra.
Motivo: Problema al restar la posición, daba -1 cuando debía ser 0.

Resultado de la función “prueba-multi-random” para 1000 partidas.

```
Se jugarán 1000 partidas con agentes aleatorios, la mitad empezando el jugador 1 y la otra mitad empezando el jugador 2
Tras 1000, el jugador 1 ha ganado 488, ha empatado 57 y ha perdido 455
```

Reto 2

Se usaba el resultado, cuando teníamos que haber sacado el valor de las casas.
No entendíamos qué hacer con la raíz y esta se usa para hacer cada uno de los movimientos.

Reto 3

Nos hubiera gustado hacer una heurística que contara el número de robos que se pudiera hacer. Hemos hecho que prevenga el robo del contrario.

Reto 4

No hubo complicaciones en la implementación.

Reto 5

Dificultad en los test de las funciones que recibían como parámetros otras funciones.

Explicación de las funciones programadas

juego.rkt

- **(t-j1 t)** Del tablero se extrae los 6 agujeros y la casa del jugador1.
- **(t-j2 t)** Del tablero se extrae los 6 agujeros y la casa del jugador2.
- **(t-agujerosj1 t)** Se extrae los 6 agujeros jugador1.
- **(t-agujerosj2 t)** Se extrae los 6 agujeros jugador2.
- **(t-agujeros t jugador)** Recibe un jugador y devuelve sus agujeros.

- **(t-semillas t indice)** Se le pasa un índice y devuelve el número de semillas de ese agujero.
- **(t-casa-semillasj1 t)** Devuelve el número de semillas que tiene la casa del jugador1.
- **(t-casa-semillasj2 t)** Devuelve el número de semillas que tiene la casa del jugador2.
- **(casa jugador-actual)** Recibe un jugador y devuelve la cantidad de semillas que tiene su casa.
- **(cambiar-semillas tablero posicion valor)** Cambia el valor de las semillas que se encuentra en la posición del agujero pasada como parámetro.
- **(agregar-semilla tablero posicion)** Suma 1 al valor de las semillas que se encuentra en el agujero pasado como parámetro.
- **(coger-semillas tablero posicion)** Coloca un 0 en el agujero que se indica por parámetro.
- **(calcular-p-final tablero posicion)** permite darle el carácter circular-antihorario a las posiciones a las que se visita los agujeros.
- **(mover tablero posicion semillas jugador-actual)** Se le pasa el agujero y el número de semillas a mover teniendo en cuenta el jugador que es y también se comprueba si puede robar.
- **(calcular-posicion posicion jugador-actual)** Si es el jugador1 devuelve el valor de las posiciones y si es el jugador2 suma 7 a la posición pasada como parámetro.
- **(roba-semillas tablero posicion)** Si puede robar cola a 0 el agujero donde cayó y la que está al frente de esta.
- **(suma-semillas tablero semillas casa)** Suma 1 semilla al valor de la casa que se pasa por parámetro.
- **(semillas-oponente tablero posicion)** Devuelve el número de semillas que tiene un agujero en concreto.
- **(es-posicion-jugador? tablero posicion jugador-actual)** Comprobará si la posición del agujero pertenece al jugador pasado por parámetro.
- **(puede-robar tablero posicion jugador-actual)** Comprueba si el jugador que está moviendo puede robar.
- **(siguiente-jugador jugador-inicial)** Política de turnos.
- **(suma agujeros resultado)** Cuenta el número de agujeros que se le pasa como parámetro.
- **(pintar-agujero agujeros)** Dibuja los agujeros.
- **(pintar tablero jugador-actual cabecera)** Pinta la cabecera del jugador actual y el estado del tablero.
- **(resultado tablero jugador-actual debug)** Resta el valor de la casa del jugador 1 con la del jugador 2. Si el resultado es positivo, gana el jugador 1, si es negativo gana el jugador 2 y si es 0 hay un empate.
- **(jugar tablero jugador-inicial estrategia-jugador1 estrategia-jugador2 debug)** Función principal del programa que recibirá el tablero, el jugador que iniciará la partida, las estrategias de cada jugador y se especifica el modo (debug o silencioso).
- **(multi-jugar n jugador-inicial ganadas perdidas empatadas estrategiaj1 estrategiaj2)** Se simula el número deseado de partidas y al finalizar mostrará el resumen de resultado de la simulación que se ejecutará en el modo silencioso.

estrategia-random.rkt

- **(estrategia-random tablero posiciones jugador-actual debug)** Se le especifica el modo y las posiciones de los agujeros del jugador en ese momento. Muestra la cabecera y pinta el tablero. En cada movimiento se elige la posición del agujero de forma pseudo-aleatoria.
- **(prueba-random)** llama a la función jugar(), pasándole la misma estrategia a cada jugador en el modo depuración.
- **(prueba-multi-random n)** se simulará tantas partidas como se desea con estrategia random. La mitad de ellas las hará el jugador 1 y la otra el jugador2.

estrategia-minimax.rkt

- **(estrategia funcion profundidad heuristica)** Se especifica una estrategia especificando la profundidad y la heurística.
- **(heuristica-simple tablero jugador posicion-final)** Si se trata del jugador1 se restará la semillas de su casa con las semillas de la casa del jugador2. Hará lo contrario si es jugador2.
- **(heuristica-turno-extra tablero jugador posicion-final)** Si la última posición del movimiento realizado por un jugador es la casa, devuelve un 1 si hay un turno extra.
- **(posicion-vacia agujeros vacios posicion)** Devuelve una lista con la posición de los agujeros que no tiene semillas.
- **(heuristica-prevenir-robos tablero jugador)** Cuenta el número de robos que puede realizar el jugador contrario.
- **(heuristica-compleja tablero jugador posicion-final)** Se hace una combinación de la heurística-simple, heurística-prevenir-robos y heuristica-turno-extra aplicando un % de 0.45, -0.1 y 0.1 respectivamente.
- **(max-valor tablero valor posiciones jugador-actual min-max funcion profundidad-max heuristica poda alfa beta)** Determina el valor máximo de la heurística min-max.
- **(max-value tablero jugador-actual profundidad-max heuristica posicion-final poda alfa beta)** Devuelve la lista de posiciones que tiene un máximo valor en la heurística min-max.
- **(min-valor tablero valor posiciones jugador-actual min-max funcion profundidad-max heuristica poda alfa beta)** Determina el valor mínimo de la heurística min-max.
- **(min-value tablero jugador-actual profundidad-max heuristica posicion-final poda alfa beta)** Devuelve la lista de posiciones que tiene un máximo valor en la heurística min-max.
- **(minimax tablero resultados jugador-actual posiciones profundidad-max heuristica poda alfa beta)** Devuelve el resultado de la partida realizada con la heurística, profundidad, poda alfa beta del jugador que se le pasa como parámetro.
- **(estrategia-minimax tablero posiciones jugador-actual heuristica profundidad-max debug)** Devuelve la posición del agujero con la que se puede hacer el mejor movimiento con la heurística y profundidad especificada.
- **(prueba-minimax jugador-empieza profundidadj1 profundidadj2 heuristicaj1 heuristicaj2 debug)** Podrá empezar el jugador que se desee, se especificará la profundidad y la heurística de cada jugador.

- **(prueba-multi-minimax-random n profundidad heuristica)** Simula un determinado número de partidas con la heurística y profundidad especificada.

Ficheros entregados

El fichero zip de la entrega posee la siguiente distribución:

- juego.rkt: código principal del programa y sus test unitarios.
- estrategia-minimax.rkt implementación de la estrategia minimax
- estrategia-random.rkt implementación de la estrategia aleatoria
- Documentación.pdf: memoria del proyecto.
- Dos carpetas: coverage-conprueba y coverage-sinprueba.

Carpeta **coverage-conprueba**: Obtenemos mayor porcentaje, pero el resultado no es tan exacto porque incluye las funciones de imprimir por pantalla cuando no le hemos hecho tests.
Carpeta **coverage-sinprueba**: Obtuvimos menor porcentaje comparado con el coverage-conprueba, ya que sólo tiene en cuenta las funciones testeadas y obvia las de imprimir por pantalla. Las funciones de imprimir por consola se ponen en rojo, sino sería el 100%.

Bibliografía

https://github.com/tmhdgsn/mancmancala
https://en.wikipedia.org/wiki/Minimax
https://fiasco.ittc.ku.edu/publications/documents/Gifford_ITTC-FY2009-TR-03050-03.pdf
https://docs.racket-lang.org/
Libro "Artificial Inteligence: A Modern Approach, 3a edición, Russel s. & Norvig P."