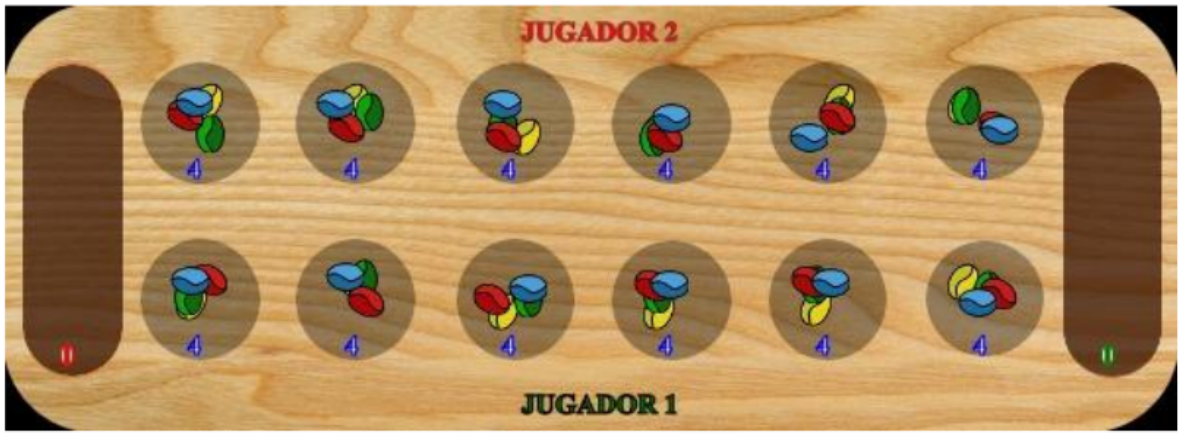
Autor: José Luis Pedraza Román
Grupo: A3

INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

Práctica 3

Agentes en entornos con adversario



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

MEMORIA MANCALA

INTRODUCCIÓN A LA SOLUCIÓN

Como ya sabemos el Mancala es un juego bipersonal con información perfecta, lo que quiere decir que juegan 2 jugadores, no se basa en el azar(es predecible) y conocemos en todo momento el estado del juego.

Para diseñar e implementar un agente deliberativo que sea capaz de jugar a este juego, primero planteé resolverlo con un árbol Minimax, algoritmo mediante el cual se crea el árbol con todas las posibilidades y se explora hasta llegar a los estados finales del juego. Este método tiene el inconveniente de que se exploran todos los nodos del árbol, y el juego que tenemos plantea muchas alternativas posibles por turno, lo que conlleva que superemos el límite de tiempo por turno y que copemos gran cantidad de memoria.

Debido a esto, opto resolverlo por la opción lógica, la poda Alfa-Beta limitando la profundidad de búsqueda y aplicando una heurística sencilla pero eficiente, la cual permitirá al bot decidir el mejor movimiento para el turno actual.

Ficheros JosBot.h y JosBot.cpp:

No requiero más que declarar en la parte privada del .h las variables “yo” y “contrincante” para establecer claramente los roles del juego, o sea, si el agente se encuentra en la posición del J1 o J2 (esto lo veremos más adelante en la descripción del método “quiensoy”); y por supuesto, definir todos los métodos de la clase necesarios y que paso a explicarlos a continuación.

Orden con las que se han ejecutado las pruebas posteriormente mostradas:
(según J1óJ2)

java -jar MancalaNoGUI.jar -p1 ./MancalaEngine/JosBot -p2 ./GreedyBot/GreedyBot -t 2

DISEÑO Y DESCRIPCIÓN DE LA SOLUCIÓN PLANTEADA

-Move nextMove():

En este método es donde establecemos la profundidad_maxima para la poda Alfa-Beta; lo establezco a 6 porque es la profundidad óptima para que funcione eficientemente.
Acto seguido hago una llamada a la función “quiensoy()” la cual nos establecerá los roles actuales, como se explica a continuación.
Llamamos a Alfa-Beta que ejecutará el algoritmo recursivo para obtener el movimiento más oportuno.
Devolvemos el siguiente movimiento para nuestro agente.

-void quiensoy():

Método desarrollado para establecer los roles de la partida actual(saber si estamos jugando como jugador1 o como jugador2):
Utiliza la función getPlayer llamada desde el estado actual.
Entendemos como "yo" a nuestro bot, que juega en el lugar de J1 o J2 dependiendo de la partida y como "contrincante" a nuestro oponente (independientemente que sea J1oJ2).
Debemos saberlo para el correcto desarrollo del algoritmo de Poda Alfa-Beta, el cual deberá saber en cada llamada si es el jugador1 o el jugador2 para actuar como MIN o como MAX según el estado del juego.
Este método es utilizado justo al principio del método que devuelve el siguiente movimiento (nextMove) después de ser calculado por Alfa-Beta.

-Move AlfaBeta()

Esquema del método encargado de devolver el movimiento más óptimo para cada estado del juego, llamando al método recursivo de la poda Alfa-Beta.

```
Move AlfaBeta(const GameState & state, int profundidad_maxima) {
  inicializamos alfa=-inf
  inicializamos beta=+inf
  inicializamos puntostotal=-inf
  inicializamos acc_act = M_NONE

  para cada casilla i{
    Si hay semillas entonces{
      Genero el hijo simulando el movimiento i
      valorhijo = AlfaBeta(hijo, contrincante, 0, profundidad_maxima, alfa, beta) //profundidad
0(primera jugada)
      Si valorhijo > puntostotal entonces
        accionactual = i
        puntostotal = valorhijo
    }
  }
  return accionactual
}
```

-int AlfaBeta()
Método recursivo encargado de evaluar la puntuación del nodo actual.

```
int AlfaBeta (const GameState & state, Player jos, int profundidad, int profundidad_maxima, double
                    alfa, double beta) {
   Si state es terminal ó profundidad=profundidad_maxima entonces{
      Si yo==ganador entonces
       devuelvo +inf
      Si contrincante ==ganador entonces
       devuelvo -inf
      Si ganador=0 entonces
       devuelvo 0
      Si no estoy en un nodo final entonces profundidad==profundidad_maxima
       devuelvo la puntuación del nodo (heuristicaa)
   }
   Si no{  //recorremos los hijos
    Si yo==MAX entonces{
     puntosmax=-inf
     para cada hijo del nodo actual{
       Si hay semillas en el hijo entonces{
         Genero el siguiente hijo simulando el moviemiento
         puntoshijomax=AlfaBeta(hijo, contrincante, profundidad+1, profundidad_maxima, alfa,
beta)
         Si puntoshijomax > puntosmax entonces
          actualizamos puntosmax=puntoshijomax
         Si puntosmax > alfa entonces
          actualizamos alfa=puntosmax
         Si beta <= alfa entonces
          PODAMOS
       }
      }
    }
    devuelvo puntosmax
   }
   Si no yo==MIN entonces{
    puntos_min=+inf
    para cada hijo del nodo actual{
      Si hay semillas en el hijo entonces{
       Genero el siguiente hijo simulando el moviemiento
       puntoshijomin=AlfaBeta(hijo, yo, profundidad+1, profundidad_maxima, alfa, beta)
       Si puntoshijomin < puntosmin entonces
        actualizamos puntosmin=puntoshijomin
       Si puntosmin < beta entonces
        actualizamos beta=puntosmin
       Si beta <= alfa entonces
        PODAMOS
      }
    }
   }
   devuelvo puntosmin
  }
}
```

-heuristicaa()

Este método es llamado en el caso que estemos en la profundidad_maxima (y esta no sea un nodo hoja ó terminal), o sea, que no haya ganador ni perdedor ni se empata, para obtener una valoración acerca de qué movimiento ó qué camino es el mejor para nuestro agente.
Como he comentado antes, la heurística que he aplicado es sencilla pero eficaz, ya que he hecho pruebas con heurísticas más complejas pero todas obteniendo peores resultados.
Simplemente se basa en contar las piezas de todas las posiciones de cada jugador ("yo" y mi "contrincante") en dos variables por separado, a estas le sumo la puntuación que suma el granero de cada uno respectivamente para así obtener los totales de semillas "totalSem1" y "totalSem2" para el movimiento en cuestión.
Una vez tengo el numero de semillas de cada jugador, llamamos al método "puedorobar()" al cual le pasamos como argumento el estado actual, y el valor de la diferencia de la suma de las semillas de cada jugador (totalSem1-totalSem2). Nuestra relación o fórmula que devuelve el método es:
(totalSem1-totalSem2) + semillaspuedorobar

-puedorobar()

Este método es llamado por la función "heuristicaa()", y podemos ver dos subsecciones(dos bucles), una para "Min" y otra para "Max".
He definido dos submétodos "roboMax()" y "roboMin()" los cuales se llaman para cada posible movimiento que pueda
ocasionar un robo (según estemos en nodo Min o en nodo Max), comprobando con la condición si tenemos posibilidad de robar (tenemos alguna casilla vacía y la correspondiente de nuestro oponente no). Con esto último debemos tener cuidado, ya que hay que invertir la el valor de la posición que esté vacía para nuestro agente y no vacía para su contrincante, para lo cual he definido otro método auxiliar para que calcule la posición que le corresponde mirar del contrincante "posContrincante()", implementada sencillamente con un switch(para saber qué posición que tenemos enfrente en cualquier momento que podamos robar).
Finalmente sumamos a la diferencia de semillas anteriormente calculada (totalSem12) el numero de semillas que le robaríamos a nuestro adversario(solo en caso de que se pueda robar) para así dar prioridad a los movimientos que mas semillas puedan robar.

CAPTURAS DE PANTALLA QUE DEMUESTRAN SU CORRECTO FUNCIONAMIENTO

(adjunto la salida completa en el .zip, en la captura sólo mostraré los estados finales de cada enfrentamiento por tema de espacio)

-P1=JosBot; P2=RandomBot;

```
[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 2.
[Simulador]: Estado actual del tablero:
        J1: Granero=8 C1=0 C2=6 C3=6 C4=7 C5=12 C6=1
        J2: Granero=7 C1=1 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 6.
[Simulador]: Estado actual del tablero:
        J1: Granero=8 C1=0 C2=6 C3=6 C4=7 C5=13 C6=0
        J2: Granero=7 C1=1 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 1.
[Simulador]: Estado actual del tablero:
        J1: Granero=40 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=8 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Fin de la partida.

-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (JosBot): 40
Tiempo del jugador 1 (JosBot): 350 milisegundos.
Puntos del jugador 2 (RandomBot): 8
Tiempo del jugador 2 (RandomBot): 400 milisegundos.
Ganador: Jugador 1 (JosBot)
```

-P1=RandomBot; P2=Josbot;

```
[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 2.
[Simulador]: Estado actual del tablero:
        J1: Granero=14 C1=6 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=2 C1=2 C2=11 C3=0 C4=2 C5=10 C6=1

[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 4.
[Simulador]: Estado actual del tablero:
        J1: Granero=14 C1=6 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=2 C1=2 C2=12 C3=1 C4=0 C5=10 C6=1

[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 1.
[Simulador]: Estado actual del tablero:
        J1: Granero=15 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=33 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Fin de la partida.

-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (RandomBot): 15
Tiempo del jugador 1 (RandomBot): 500 milisegundos.
Puntos del jugador 2 (JosBot): 33
Tiempo del jugador 2 (JosBot): 400 milisegundos.
Ganador: Jugador 2 (JosBot)
```

-P1=JosBot; P2=GreedyBot;

```
[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 5.
[Simulador]: Estado actual del tablero:
        J1: Granero=17 C1=0 C2=11 C3=0 C4=0 C5=0 C6=3
        J2: Granero=15 C1=0 C2=0 C3=1 C4=1 C5=0 C6=0

[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 6.
[Simulador]: Estado actual del tablero:
        J1: Granero=19 C1=0 C2=11 C3=0 C4=1 C5=1 C6=0
        J2: Granero=15 C1=0 C2=0 C3=1 C4=0 C5=0 C6=0

[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 3.
[Simulador]: Estado actual del tablero:
        J1: Granero=31 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=17 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Fin de la partida.

-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (JosBot): 31
Tiempo del jugador 1 (JosBot): 650 milisegundos.
Puntos del jugador 2 (GreedyBot): 17
Tiempo del jugador 2 (GreedyBot): 750 milisegundos.
Ganador: Jugador 1 (JosBot)
```

-P1=GreedyBot; P2=GreedyBot;

```
[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 2.
[Simulador]: Estado actual del tablero:
        J1: Granero=16 C1=1 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=7 C1=8 C2=1 C3=1 C4=13 C5=1 C6=0

[Simulador]: Turno del jugador 2.
[Simulador]: El jugador 2 realiza el movimiento 2.
[Simulador]: Estado actual del tablero:
        J1: Granero=16 C1=1 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=7 C1=9 C2=0 C3=1 C4=13 C5=1 C6=0

[Simulador]: Turno del jugador 1.
[Simulador]: El jugador 1 realiza el movimiento 1.
[Simulador]: Estado actual del tablero:
        J1: Granero=17 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0
        J2: Granero=31 C1=0 C2=0 C3=0 C4=0 C5=0 C6=0

[Simulador]: Fin de la partida.

-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (GreedyBot): 17
Tiempo del jugador 1 (GreedyBot): 850 milisegundos.
Puntos del jugador 2 (JosBot): 31
Tiempo del jugador 2 (JosBot): 450 milisegundos.
Ganador: Jugador 2 (JosBot)
```

Como se puede comprobar nuestro algoritmo funciona correctamente contra los otros dos (veremos qué pasa contra MuerteAlosCobardes!) ya que es capaz de ganarles en unos tiempos más que razonables (dado a que la profundidad máxima establecida para la exploración es óptima) con una buena puntuación.