PRÁCTICA 3:
AGENTES EN ENTORNOS CON ADVERSARIO

Sergio Vela Pelegrina

Inteligencia artificial

Grupo C

ÍNDICE

1. INTRODUCCIÓN

La tercera práctica de la asignatura consiste en el diseño e implementación de un agente deliberativo desplegado en un entorno multi-agente competitivo en el que se encuentran dos agentes con objetivos contrapuestos que escogen sus acciones por turnos.

1.1. Objetivos de la práctica

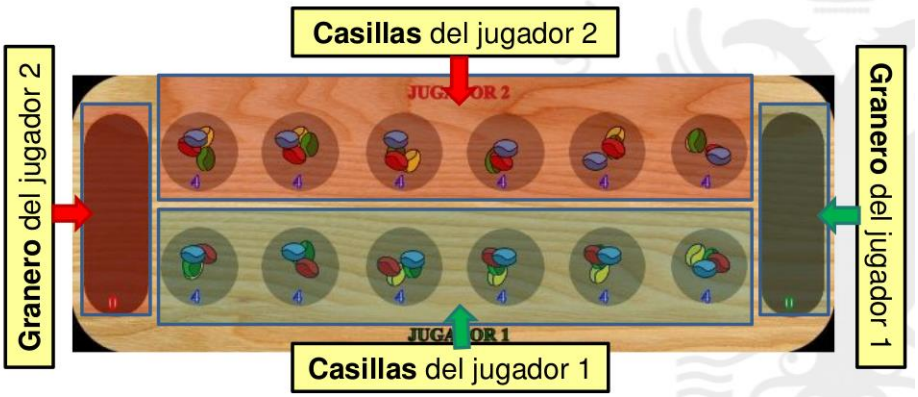
- Familiarizarse con una aplicación concreta de inteligencia artificial dentro del ámbito de los agentes desplegados en entornos con adversario.
- Aprender a usar técnicas de búsqueda basadas en algoritmos para juegos.
- Implementar un agente para resolver un problema concreto en este ámbito: **el juego del Mancala.**

1.2. Mancala

El **Mancala** es un juego de tablero para dos jugadores en que se juega por turnos. Procede de África y Ásia. Existen muchas versiones del juego, nosotros nos quedaremos con con la más popular.

1.2.1. Elementos del juego

- Hay **12** hoyos de cultivo o **casillas** (6 para cada jugador), en dos filas de 6.
- **48** semillas, inicialmente repartidas equitativamente entre las **12 casillas**, (4 semillas por casilla al principio del juego).
- **2 graneros**, situados en los extremos del tablero. Cada uno pertenece a un jugador.



1.2.2. Dinámica del juego

- El **Mancala** se juega secuencialmente, por turnos, **comenzando por el jugador 1.**
- En cada turno, el jugador que mueve selecciona un **movimiento de siembra**, escogiendo **una de sus casillas**.
- La **siembra** consiste en quitar todas las semillas de la casilla seleccionada e ir colocando una única semilla en las casillas inmediatamente inferiores, secuencialmente, **incluido en su granero si es posible**.
- Si tras colocar todas las semillas aún quedaran, estas se seguirán colocando de **una en una** en las casillas del **contrario**, de **mayor a menor** número de casilla.
- **Nunca colocaremos una semilla en el granero del contrincante**.
- Si llegamos a poner una semilla en la casilla **1** del **contrario**, y aún siguen quedando semillas, comenzaremos a poner de nuevo **una semilla en cada una de nuestras casillas, en orden descendente**.

1.2.3. Objetivos del juego

- Obtener mayor número de semillas en nuestro granero que nuestro contrincante.
- El juego terminará cuando alguno de los dos jugadores no tenga semillas en ninguna de sus casillas.
- Si se llega al final del juego y uno de los jugadores aún tiene semillas en sus casillas, automáticamente las recogerá y las sumará a su granero.

1.2.4. Reglas especiales

Turno extra: Cuando un jugador siembra una casilla y la última semilla la deposita en su granero, gana un turno extra y vuelve a jugar.

Robo: Si al realizar una siembra, la última semilla del jugador cae en una de sus casillas que esté vacía, pero la opuesta del contrincante no, entonces el jugador lleva directamente esa semilla y las de la casilla del jugador contrario a su granero.

Inmolación: Si un jugador intenta hacer un movimiento no permitido (sembrar una casilla que no tiene semillas), automáticamente perderá la partida por 48 semillas a 0.

1.2.5. Conclusión

Para concluir esta primera parte de la práctica, deberemos implementar un jugador artificial inteligente para el juego del **Mancala que trate ganar**. Nuestro jugador deberá poder jugar como **jugador 1** ó **jugador 2** instintivamente. Deberemos implementar un comportamiento deliberativo basado en búsqueda en entornos con adversario (**poda α-β**). Tenemos un límite de tiempo en cada turno de **2 segundos**.

2. DISEÑO DEL ALGORITMO

Para poder comenzar la práctica, lo primero que debía hacer era pensar cómo iba a abordar el problema. Tenía dos posibilidades:

- Algoritmo **Minimax**
- Algoritmo **Poda α-β**

En mi caso, decidí implementar de primeras la **poda α-β**, ya que permite descartar caminos del árbol de juego, ahorrando tiempo para poder explorar otros caminos más prometedores, mientras que el algoritmo **minimax** se recorre todo el árbol de juego directamente. Por tanto, explicaré en qué consiste mi algoritmo y cómo lo he diseñado.

De primeras, mi algoritmo recibía como parámetros una variable de tipo **GameState** que nombré como "**estado**" y otra variable de tipo **Move** que nombré como "**mov**", entre otras variables que comentaré más adelante.

Cuando empecé con la implementación, me di cuenta que, me resultaba incómodo trabajar con las variables anteriormente nombradas, es ese el motivo por el cual decidí declarar un **struct** "**Nodo**" el cual tuviera las dos variables anteriormente nombradas.

```
struct Nodo{
GameState estado;
Move mov;
};
```

Dicho struct está declarado en el **.h** y me servirá para saber en cada **estado** el **movimiento** que he hecho.

Comentado esto, el método en el que implemento el algoritmo de **poda α-β** se llama **PodaAlfaBeta** y recibe como parámetro un **Nodo** "**mi_nodo**" pasado por referencia, una variable de tipo entero llamada **profundidad**, la cual limitará la profundidad que queremos estudiar en el desarrollo del árbol.

También le paso un jugador de tipo **Player** por referencia y constante "**j**", que **hará referencia a mi jugador en la partida**, y dos variables de tipo entero llamadas **alfa** y **beta**. En un principio, **alfa** tendrá el valor **-∞** y beta el valor **+∞**. Ahora comentaremos el motivo.

El método comienza con una condición la cual comprueba que no estemos en un nodo "**hoja**" o hayamos alcanzado el límite de **profundidad** que le pasamos como parámetro al método.

```
if((profundidad > 0) && !mi_nodo.estado.isFinalState())
```

En caso de que se cumpla la condición, como hemos comentado, significará que ni estamos en un nodo **hoja** ni hemos alcanzado el límite de profundidad, lo siguiente que debemos comprobar es si es un nodo **max**, en cuyo caso significará que es **nuestro turno**, o si es un nodo **min**, en cuyo caso significará que es **turno de mi adversario**.

Para dicha comprobación, uso el siguiente condicional:

```
if(j == mi_nodo.estado.getCurrentPlayer())
```

La variable "**j**" hace referencia al **jugador** que le pasamos como parámetro al método, que como he comentado previamente, **será mi jugador**. "**mi_nodo**" hace referencia al **Nodo** que le pasamos a la función que como sabemos, tiene un **estado** y un **movimiento**.

En este caso comprobamos si es **nuestro turno**, en cuyo caso la condición se cumplirá y entraremos dentro del condicional. Estaremos por tanto en un nodo **max**.

Suponiendo que sí que es mi turno y por tanto, entramos en el condicional, defino dos variables, "**mejor_valor**" inicializada a **-∞**, y otra variable de tipo **Move** llamada "**mejor_movimiento**".

Definimos también una variable booleana "**podar**" en un principio inicializada a **false**. Dicha variable nos servirá para **podar** cuando sea necesario, o sea, cuando estemos explorando un camino poco prometedor.

Después tenemos un bucle **for**, **for(int i = 1; i < 7 && !podar; i++)** desde 1 a 6, que son los **graneros** que tenemos en el juego. El bucle se ejecutará hasta llegar a **6** o hasta que haya que **podar**.

Es entonces cuando defino un **Nodo** "**hijo**" al cual le asignamos el **estado** que tendría el **nodo padre** o en mi caso, "**mi_nodo**", de ejecutar el **movimiento** dado en cada una de las iteraciones del bucle. Para simular dicho **estado**, hago uso de la función que se nos da definida llamada **simulateMove**. Tras lo cual defino una variable de tipo entero llamada "**valor**" y le asigno el resultado obtenido de la llamada recursiva a mi función, en este caso, los valores que le paso son el **nodo** "**hijo**", la **profundidad-1**, el jugador **j**, **alfa** y **beta**.

Una vez que tengo dicho **valor** de la llamada a la función, compruebo si ese **valor** es más grande que el **mejor valor** almacenado hasta el momento. (Como he comentado anteriormente, **mejor valor** inicialmente vale **-∞**).

Dicho condicional es el siguiente: **if(valor > mejor_valor)**, en caso de que se cumpla la condición, actualizaremos **mejor_valor** con dicho **valor** y asignaremos a la variable **mov** de **mi_nodo** el movimiento de esa iteración del bucle.

En resumidas cuentas, en **mejor_valor** tendremos el valor más grande obtenido hasta el momento y en **mi_nodo.mov** el mejor movimiento hasta el momento.

Después asignaremos a **alfa** el valor **mayor** entre **alfa** y **mejor_valor**. Para ello, he definido una función llamada **CalculaMaximo**, a la cual simplemente le pasamos dos enteros y nos devuelve el más grande. Esta comprobación nos hará falta para verificar si tenemos que realizar la **poda**.

Por tanto, la siguiente comprobación será si **beta <= alfa**, en caso de que se cumpla, el camino que estamos examinando es poco prometedor y debemos **podar**, por lo que la variable booleana **podar** la pondremos como **true**, saldremos del bucle **for** y devolveremos **mejor_valor**.

Aquí termina nuestra primera parte del algoritmo, en la que básicamente comprobamos el **mejor valor posible** que podemos obtener siendo **nuestro turno** y estando por tanto en un nodo **max**.

Ahora hay que comprobar la parte en la que el turno es de nuestro **adversario** (nodo **min**).

En esta parte, básicamente, hacemos las mismas comprobaciones que en el apartado anterior pero al contrario, es decir, como nos encontramos en un nodo **min**, empezaremos inicializando la variable **mejor_valor** a **+∞**. El resto de las declaraciones es exactamente igual que en anterior caso, el bucle **for** y la llamada recursiva al método también, la diferencia es que en este caso, deberemos comprobar si el **valor** obtenido de la llamada recursiva es **MENOR** que **mejor_valor** (estamos en un nodo **min**). **if(valor < mejor_valor)**, en caso de que se cumpla, repetiremos el proceso del anterior caso, asignamos a **mejor_valor** el **valor** obtenido, y lo mismo con **mi_nodo.mov**. A la variable **beta** le asignaremos el **menor** valor entre **beta** y **mejor_valor**, para ello hago uso de otro método auxiliar llamado **CalculaMinimo**, que, pasándole dos enteros, nos devolverá el más pequeño. En caso de que **beta <= alfa**, estaremos recorriendo un camino poco prometedor, por lo que la variable **podar** la hacemos **true**, salimos del bucle **for** y devolvemos **mejor_valor**.

Quiero hacer énfasis en que si nos encontramos en un nodo **max** (es nuestro turno) el **mejor_valor** será el más alto posible, mientras que si nos encontramos en un nodo **min** (turno del adversario) **mejor_valor** será el valor más pequeño posible.

Al comienzo de la explicación del algoritmo, he comentado que la primera condición era **if((profundidad > 0) && !mi_nodo.estado.isFinalState())** con la que comprobamos si estamos en un nodo **hoja** o si hemos alcanzado el límite de **profundidad** establecido.

Entonces, ¿qué debemos hacer si estamos en un nodo **hoja** o hemos alcanzado dicho límite de **profundidad**?

La respuesta parece evidente, deberemos asignarle una "**puntuación**" o **valor** a dicho **nodo hoja**. Es por tanto que en esta parte definiremos la **heurística** de nuestro algoritmo. Es muy importante, ya que cuanto mejor sea, mejores resultados obtendremos contra nuestros adversarios. Para el cálculo de la heurística, he definido un método auxiliar al cual he llamado EvaluaNodo, el cual recibe como parámetro el **nodo** pasado por referencia y el **jugador**.

En un principio, implementé una **heurística** muy básica, que consistía en sumar las semillas que se encontraban en mis casillas y le restaba las semillas que se encontraban en las casillas del adversario. Dicha diferencia de semillas es la que devolvía. Básicamente lo que hacía era lo siguiente:

```
for(int i = 0; i < 7; i++){
mis_semillas += mi_nodo.estado.getSeedsAt(j, (Position) i);
semillas_contrario += mi_nodo.estado.getSeedsAt(contrario, (Position) i);
}

int semillas_totales = mis_semillas - semillas_contrario;
return semillas_totales;
```

Dicha heurística la implementé con el único fin de comprobar que funcionara mi algoritmo de **poda**, pero para mi sorpresa, comprobé que funcionaba bastante bien y conseguía ganar a **Greedy** fácilmente como veremos a continuación:

- Como jugador 1:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VelaBot): 37
Tiempo del jugador 1 (VelaBot): 450 milisegundos.
Puntos del jugador 2 (GreedyBot): 11
Tiempo del jugador 2 (GreedyBot): 500 milisegundos.
Ganador: Jugador 1 (VelaBot)
```

- Como jugador 2:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (GreedyBot): 12
Tiempo del jugador 1 (GreedyBot): 600 milisegundos.
Puntos del jugador 2 (VelaBot): 36
Tiempo del jugador 2 (VelaBot): 800 milisegundos.
Ganador: Jugador 2 (VelaBot)
```

El problema fue cuando competí contra bots de mis compañeros y vi que la básica **heurística** que utilizaba no daba tan buenos resultados como podemos ver a continuación:

- Como jugador 1:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VelaBot): 22
Tiempo del jugador 1 (VelaBot): 900 milisegundos.
Puntos del jugador 2 (PPBot): 26
Tiempo del jugador 2 (PPBot): 1000 milisegundos.
Ganador: Jugador 2 (PPBot)
```

- Como jugador 2:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (PPBot): 28
Tiempo del jugador 1 (PPBot): 1900 milisegundos.
Puntos del jugador 2 (VelaBot): 20
Tiempo del jugador 2 (VelaBot): 1600 milisegundos.
Ganador: Jugador 1 (PPBot)
```

Es entonces cuando comencé a estudiar posibles jugadas y a intentar mejorar la heurística. Comencé a tener en cuenta cuántos **posibles movimientos** tenía (número de casillas distintas de 0). También comencé a tener en cuenta el **número de semillas** que tenía en mi **granero**.

Otra heurística que implementé fue calcular la diferencia entre las semillas de mi granero, menos las semillas del granero de mi adversario más 24 - las semillas del granero de mi adversario.

También implementé otra en la que, si yo era el jugador ganador de la partida, devolvía un valor alto, al contrario que si mi oponente era dicho ganador.

Pero para mi sorpresa, la heurística que mejor resultados me ha dado ha sido en la que calculo las semillas que hay en mis casillas, les resto las del oponente y a esta diferencia, le sumo mi puntuación. La implementación sería la siguiente:

```
for(int i = 0; i < 7; i++){
mis_semillas += mi_nodo.estado.getSeedsAt(j, (Position) i);
semillas_contrario += mi_nodo.estado.getSeedsAt(contrario, (Position) i);
}

int mi_puntuacion = mi_nodo.estado.getScore(j);

int semillas_diferencia = mis_semillas - semillas_contrario;

return (semillas_diferencia + mi_puntuacion);
```

Con dicha heurística, ya consigo vencer al oponente anterior y a varios compañeros con los que he competido como veremos a continuación:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VelaBot): 35
Tiempo del jugador 1 (VelaBot): 750 milisegundos.
Puntos del jugador 2 (PPBot): 13
Tiempo del jugador 2 (PPBot): 550 milisegundos.
Ganador: Jugador 1 (VelaBot)
```

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (PPBot): 20
Tiempo del jugador 1 (PPBot): 800 milisegundos.
Puntos del jugador 2 (VelaBot): 28
Tiempo del jugador 2 (VelaBot): 650 milisegundos.
Ganador: Jugador 2 (VelaBot)
```

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VelaBot): 34
Tiempo del jugador 1 (VelaBot): 1450 milisegundos.
Puntos del jugador 2 (AlejandroGarciaPerez_Bot): 14
Tiempo del jugador 2 (AlejandroGarciaPerez_Bot): 1050 milisegundos.
Ganador: Jugador 1 (VelaBot)
```

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (AlejandroGarciaPerez_Bot): 20
Tiempo del jugador 1 (AlejandroGarciaPerez_Bot): 900 milisegundos.
Puntos del jugador 2 (VelaBot): 28
Tiempo del jugador 2 (VelaBot): 850 milisegundos.
Ganador: Jugador 2 (VelaBot)
```

Aunque hay bots que me ganan cuando son **J1**:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VitoEscopeta): 26
Tiempo del jugador 1 (VitoEscopeta): 1700 milisegundos.
Puntos del jugador 2 (VelaBot): 22
Tiempo del jugador 2 (VelaBot): 900 milisegundos.
Ganador: Jugador 1 (VitoEscopeta)
```

Pero les gano yo también cuando soy el **J1**:

```
-------FIN DE LA PARTIDA
-------
Puntos del jugador 1 (VelaBot): 39
Tiempo del jugador 1 (VelaBot): 900 milisegundos.
Puntos del jugador 2 (VitoEscopeta): 9
Tiempo del jugador 2 (VitoEscopeta): 600 milisegundos.
Ganador: Jugador 1 (VelaBot)
```

Ahora que he explicado mi **heurística**, nos queda una cosa por comentar, ¿desde donde llamamos a nuestro método **PodaAlfaBeta**?

Para ello, tenemos el método **nextMove**, en el cual tenemos una variable tipo **Player** a la que he llamado "**yo**", y la inicializamos haciendo uso de la función **getPlayer** si somos el jugador **1** o el **2**.

Primero, definimos **alfa** y **beta** inicializadas a **-∞** y **+∞** respectivamente, después, establecemos el nivel de profundidad, que en mi caso es **11** por último, creamos un **nodo n** inicializado al **estado** que se pasa como argumento al método y como **movimiento M_NONE**.

Es entonces cuando a una variable de tipo entero llamada **movimiento** le asignamos el valor que nos devuelve la llamada al método **PodaAlfaBeta**.

Este método devolverá el mejor **movimiento** que se le ha asignado al nodo en el método **PodaAlfaBeta**. (**return n.mov**).

Para terminar, me gustaría comentar el motivo por el cual he establecido el límite de profundidad a **11**. El motivo es sencillo, al tener un límite de tiempo de 2 segundos, teniendo **13** de profundidad, por ejemplo, en algunos casos me pasaba de tiempo. Si le pongo menos de **11** de profundidad, mi bot no obtiene resultados tan buenos y entre **11** y **12** me he decantado por **11** porque obtengo mejores resultados y no me arriesgo a pasar el tiempo límite que tenemos para cada jugada.