

MEMORIA PRÁCTICA 3:

Agentes en entornos con adversario

Se ha implementado el algoritmo *Mini-Max* acompañado de la poda *alfa-beta* para realizar la búsqueda del mejor movimiento a escoger frente a nuestro rival en el juego *Mancala*.

En el bot implementado, Helheim, se han utilizado únicamente los tipos de datos aportados. Se han añadido métodos y datos miembros de ámbito privado respecto a la implementación base que nos había sido aportada:

➤ Datos miembro:

- ***const int profundidad_max = ... ;***
Delimita la máxima profundidad a la que podrá ahondar el algoritmo en busca del estado más conveniente. Visto desde otra perspectiva, restringe el número de llamadas recursivas (al método *valorMIN_MAX()*).
- ***Player turno;***
Esta variable contendrá el número de jugador (J1 o J2) que se nos ha asignado en la partida. Se inicializará en el método *nextMove()*.
- ***Player oponente;***
Esta variable contendrá el número de jugador (J1 o J2) que se le ha asignado a nuestro contrincante en la partida. Se inicializará en el método *nextMove()*.

➤ Métodos:

- ***int valorMIN_MAX(const GameState &state,
const GameState &simulacion,
int profundidad,
int alfa,
int beta);***

Devuelve el beneficio de un estado.

Este es el método recursivo. Su caso base es cuando llegamos a la profundidad máxima o cuando llegamos a un estado final (ya sea ganar o perder). En el caso base se limita a calcular el beneficio que nos aporta ese estado y lo devuelve.

En cualquier otro caso realiza una simulación de la partida para cada movimiento válido disponible y se calcula el beneficio que aporta la nueva situación. Dependiendo de si es nuestro turno(*MAX*) o el del adversario(*MIN*) actualiza el valor a devolver así como *alfa* o *beta*. Si se trata de una jugada del oponente actualiza *valor* y *beta* si el nuevo beneficio calculado es menor que el anterior guardado. Si la jugada nos corresponde a nosotros actualizamos *valor* y *alfa* si el nuevo beneficio calculado es mayor que el anterior guardado. Por último devolvería el último beneficio actualizado en la variable *valor*.

Si al actualizar los valores de *alfa* y *beta* llegamos a la situación en que *alfa >= beta*

se procedería a podar todas las ramas restantes por explorar (sale del bucle)

- ***Move MINI_MAX(const GameState &state);***

Este es el método inicial que se encarga de llamar a *int valorMIN_MAX(...)* para devolver la mejor opción de movimiento disponible. Según el beneficio que vayamos obteniendo vamos actualizando el mejor movimiento a tomar. Funciona como *MAX* en el algoritmo Mini-Max, es decir, actualizamos si el nuevo beneficio obtenido es mejor que el anterior.

- ***int calcularUtilidad(const GameState &simulacion);***

Este es el método que implementa la heurística para calcular el beneficio de un estado de la partida determinado. Realmente es muy sencillo:

$$\text{beneficio} = \text{semillas en mi granero} - \text{semillas en su granero}$$

Indica si vamos ganando o perdiendo en ese estado. Cuando mayor es beneficio mayor será nuestra ventaja sobre el rival y viceversa.

Podemos dar una explicación de la implementación del algoritmo siguiendo el orden de ejecución del código:

El método llamado por el simulador es *nextMove(...)*. En este método inicializamos los datos miembros *turno* y *oponente*. Por último devolvemos el movimiento calculado por el método *MINI_MAX(...)*.

En *MINI_MAX(...)* inicializamos las variables *mejor_movimiento* = *M_NONE* (mejor movimiento asociado al mejor beneficio calculado hasta el momento. Al principio asignamos un movimiento nulo, no válido), *mejor_utilidad* = -48 (mejor beneficio calculado hasta el momento. Inicializado al menor valor alcanzable, es decir, perder la partida por el máximo de puntos), *simulacion* donde guardaremos los posibles estados al aplicar un movimiento al estado actual de la partida, *alfa* = -48 y *beta* = 48. A continuación iteramos por todos los posibles movimientos y si es factible según las reglas del juego reproduciremos una simulación de la partida tomando ese movimiento y calcularemos el beneficio que nos aporta. Cada vez que se acabe de estudiar un movimiento válido se actualizará, si corresponde, *mejor_utilidad* y *mejor_movimiento*.

En *valorMIN_MAX(...)* nos encargamos de calcular el beneficio de los movimientos iterados en *MINI_MAX(...)*. Inicializamos una variable *valor* que será la que devuelva el beneficio asociado al estado y una variable *es_mi_turno* que nos permitirá identificar si realizar las operaciones asociadas a un estado *MAX* o a un estado *MIN*. Si es nuestro turno inicializamos *valor* a -48 y si no lo es a +48 (mínimo y máximo valor alcanzable en nuestra heurística). Al igual que antes creamos una variable *simulacion* por el mismo motivo. Comprobamos si es el caso base para devolver el valor que nos proporcione el método *calcularUtilidad(...)* y sino inicializamos una variable *salir* a *false* que usaremos para indicar si *alfa* >= *beta*, es decir, para indicar si debemos podar las ramas restantes del árbol. En este estado de la partida también disponemos de movimientos posibles así que deberemos iterar por ellos y si es factible repetir el proceso. Cuando se llegue a un caso base comenzará la regresión de los valores calculados a través de las llamadas recursivas. Al recibir un dato actualizamos *valor* y *alfa* si este dato es mayor que el que teníamos guardado anteriormente y estamos en nuestro turno; actualizamos *valor* y *beta* si este dato es menor que el que teníamos guardado anteriormente y es el turno del oponente.