Práctica 3: Desconecta4Boom

David Gómez Hernández – 2ºB

Ingeniería Informática – Grado Ingeniería Informática

ÍNDICE

1.- Objetivos de la práctica

La práctica consistía en diseñar un agente para el juego Desconecta4Boom. En este juego el objetivo era ganar no haciendo 4 en raya (tanto en diagonal, vertical y horizontal). El agente debía implementar un algoritmo de poda Alfa-Beta que recorriera todo el árbol de juego y buscara el mejor movimiento para poder ganar al rival.

Dicha poda utiliza una heurística diseñada e implementada de tal forma que los movimientos sean concisos y estén destinados a la victoria por parte del agente. Esta heurística será probada contra unas heurísticas del servidor de la UGR llamados ninjas: ninja 1, ninja 2 y ninja 3. A medida que aumentamos de ninja, más dificultad tiene. La heurística tiene que ser capaz de ganarle a los 3.

2.- Algoritmo de poda Alfa-Beta

La poda alfa-beta es una técnica de búsqueda que reduce el número de nodos evaluados en un árbol de juego con implementación MiniMax.

El problema del MiniMax es que el número de estados a explorar es exponencial al número de movimientos. Partiendo de esto, la poda alfa-beta trata de eliminar partes del árbol aplicando MiniMax, de forma que se devuelva el mismo movimiento que usaría este pero habiendo podado una gran cantidad de ramas.

```
double PodaAlfaBeta(const Environment &E, int jugador, int profundidad, Environment::ActionType &accion,
double alpha, double beta){
    si E es terminal o la profundidad es 0
        devolver valor heurístico del nodo (funcion Valoracion)

    si jugador es el que queremos maximizar (jugador == 1){
        valor:= menosinf;
        para cada hijo de nodo
            valor:=maximo(valor,PodaAlfaBeta(hijo,jugador,profundidad-1,ultima_accion,alpha.beta))
            si PodaAlfaBeta(hijo,jugador,profundidad-1,ultima_accion,alpha,beta) > valor
                accion:=  static_cast<Environment::ActionType>(accion_anterior);
            alpha:=max(alpha,valor);
            si alpha >= beta
                break; (*Podar Beta*)

        return valor; (*Devolvemos alpha*)
    }
    sino
        valor:= masinf;
        para cada hijo de nodo
            valor:=min(valor,PodaAlfaBeta(hijo,jugador,profundidad-1,ultima_accion,alpha.beta))
            si PodaAlfaBeta(hijo,jugador,profundidad-1,ultima_accion,alpha,beta) < valor
                accion:=  static_cast<Environment::ActionType>(accion_anterior);
            beta:=min(alpha,valor);
            si alpha >= beta
                break; (*Podar alpha*)

        return valor; (*Devolvemos beta*)
}
```

En nuestro caso nuestro algoritmo empieza comparando si la profundidad es 0(es decir, ha llegado al límite) o si ha llegado a un nodo terminal. Seguido comprueba de si el jugador es que el hay que maximizar (nosotros) y devuelve el valor en menos o más en función de dicho valor.

Seguidamente compara si el jugador es el que hay que maximizar, para podar alfa. Recorre todos los nodos hijos y realiza una llamada a recursiva de la misma poda, cambiando el estado por el estado del hijo, de tal manera que va recorriendo las ramas de los nodos y va devolviendo un valor, llamado minimax.

El minimax se compara con el valor menosinf, que era el valor que tenía al principio el valor alpha. Si lo supera, el valor se iguala al del minimax y la acción heurística se iguala a la acción anterior. Es decir, elegimos el máximo entre el minimax y el valor.

Después se comprara el valor con alpha, y si es menor alpha se iguala a dicho valor (máximo entre alfa y beta). Finalmente si alpha es mayor o igual que beta, se poda alpha.

Para podar beta se hace de una forma más o menos similar. El valor se cambia por el masinf (el primer valor que toma beta) y se vuelven a recorrer los hijos. Esta vez el minimax se comprueba de que sea menor que valor, si es así el valor se iguala al minimax (se coge el mínimo entre valor y minimax).

Seguidamente se comprueba de que valor sea menor que beta. Si esto ocurre se iguala beta a valor (se coge el mínimo entre beta y valor). Finalmente si alpha es mayor o igual que beta, se poda beta.

3.- Funciones heurísticas

En el desarrollo de la práctica se nos pide que hagamos una heurística para que el agente tenga una base a la hora de elegir un nodo en la poda Alfa-Beta. Para ello me he apoyado de dos funciones. ContadorFichas y Valoración

```
double contadorFichas(const Environment &E,int jugador){
    int rival, bomba, bomba_rival;
    int contiguas = 0, contiguas_rival = 0;
    double valor = 0;

    si jugador==1
        bomba = 4;
        rival = 2;
        bomba_rival = 5;
    sino
        bomba = 5;
        rival = 1;
        bomba_rival = 4;

    (*Suponemos el algoritmo para las filas horizontales, para evitar repeticiones, ya para los demás ejes es lo mismo*)
    para i:=0 hasta 7
        para j:=0 hasta 7
            si E.SeeCasilla(i,j) == 0
                valor:= valor - contiguas_rival;
                valor:= valor + contiguas
                contiguas:=0;
                contiguas_rival:=0
            sino si E.SeeCasilla(i,j) == jugador or E.SeeCasilla(i,j) == bomba
                contiguas:=contiguas + 1;
                valor:= valor + contiguas_rival;
                contiguas_rival:=0;
            sino si E.SeeCasilla(i,j) == rival or E.SeeCasilla(i,j) == bomba_rival
                contiguas_rival:=contiguas_rival + 1;
                valor:= valor - contiguas;
                contiguas:=0;

    (*Realizamos el mismo algoritmo para el resto de ejes*)
    ....
    return valor;
}
```

.

En contadorFichas se le pasa por parámetro el entorno y el jugador. Lo que realiza esta función es contar las fichas en horizontal, diagonal y vertical. Pongamos de ejemplo de que hay 2 fichas verdes (siendo nosotros) en horizontal y luego una ficha azul (siendo el rival).

El algoritmo va contando las fichas contiguas verdes hasta que encuentra la azul. En el momento en que encuentra la azul el algoritmo le resta al valor final de la heurística el número de fichas contiguas (ya que lo que quieres es que se maximice, es decir, que esté lo más lejos de 4 posible, ya que si hace 4 pierde).

Realiza lo mismo las casillas del rival (es decir, si encuentra 2 azules y 1 verde) solo que lo suma en vez de restar, ya que para ganar necesitas que el rival haga 4.

También si encuentra una casilla vacía al elemento valor le suma las contiguas del rival y le resta las contiguas del jugador, igualándolas a 0.

Esto lo hace con cada uno de los ejes, modificando el valor en función de las fichas tanto suyas como las del rival. Finalmente el valor que ha ido modificando lo devuelve.

```
double Valoracion(const Environment &estado, int jugador){
    int rival, ganador = estado.RevisarTablero();
    si jugador == 1 rival = 2 sino rival = 1;

    si  ganador == jugador
        devolver 99999999.0;
    sino si ganador = rival
        devolver -99999999.0;
    sino si estado.getCasillasLibres() == 0
        devolver 0;
    sino
        devolver contadorFichas(estado,jugador);
}
```

La segunda función es la de Valoración, a la cual se le pasa el entorno y el jugador. Esta función comprueba quien ha ganado de los 2 jugadores en función del jugador que esté jugando (nosotros).

Si ha ganado el jugador devuelve un entero positivo, si ha ganado el rival un entero negativo, si no ha ganado nadie devuelve 0 y si no se da ninguno de estos casos, llama a la función heurística contadorFichas que devuelve el valor en función de las fichas (explicada arriba).

De esta forma el agente utiliza la poda AlfaBeta usando esta función, que a su vez usa la función contadorFichas para poder devolver un resultado preciso, de tal forma que realice bien las podas para lograr un resultado óptimo.

Utilizando todo esto, el agente es capaz de ganarle a los 3 ninjas tanto de verde como en azul.