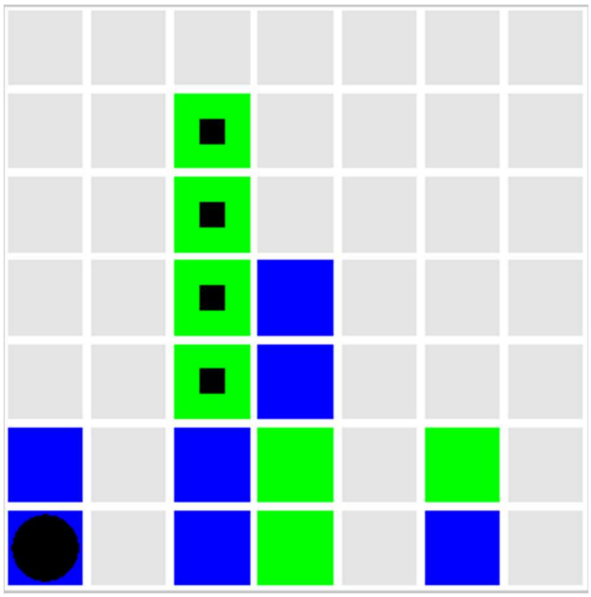
DESCONECTA4BOOM

Memoria de la Práctica 3. Inteligencia artificial.



Esther García Gallego

2ºB Ingeniería Informática.

1. ANÁLISIS DEL PROBLEMA

El problema planteado en esta práctica es la implementación de un agente deliberativo que juegue contra un adversario, siguiendo las técnicas de búsqueda como el algoritmo MiniMax o la Poda Alfa-Beta.

En este caso, nuestro entorno es un tablero de siete filas y siete columnas, en el que deben ir colocándose fichas. En cada turno, los jugadores (verde y azul) podrán colocar dos fichas de su color en cualquier columna del tablero que se encuentre disponible, excepto en la primera jugada, que constará de una única ficha verde (por ser éste el jugador que inicia el juego).

Además de fichas normales, hemos de notar la existencia de las fichas bomba. Éstas permiten al jugador eliminar las fichas del oponente que se encuentren en la misma fila mediante una "explosión".

El juego concluye cuando alguno de los jugadores consigue hacer "cuatro en raya", es decir, coloca cuatro fichas de su color en línea. Esto puede suceder de forma vertical, horizontal o diagonal. En caso de que se llene el tablero por completo, se considera un empate.

2. SOLUCIÓN PLANTEADA

Como solución, se ha implementado el algoritmo de Poda Alfa-Beta siguiendo el esquema mostrado en el código 1, aunque se han incluido algunas modificaciones para una mejor legibilidad y una menor cantidad de código.

```
función alfa-beta(nodo //en nuestro caso el tablero, profundidad,
α, β, jugador)
    si nodo es un nodo terminal o profundidad = 0
        devolver el valor heurístico del nodo
    si jugador1
        para cada hijo de nodo
            α := max(α, alfa-beta(hijo, profundidad-1, α, β,
jugador2))
            si β≤α
                romper (* poda β *)
        devolver α
    si no
        para cada hijo de nodo
            β := min(β, alfa-beta(hijo, profundidad-1, α, β,
jugador1))
            si β≤α
                romper (* poda α *)
        devolver β
```

Código 1. Esquema de la poda Alfa-Beta.

En primer lugar, se comprueba si se ha llegado a la profundidad límite, establecida por la variable PROFUNDIDAD_ALFABETA que se proporciona por parámetros o si se ha alcanzado un nodo terminal (lo que daría por finalizado el juego).

```
if (profundidad == PROFUNDIDAD_ALFABETA || actual.JuegoTerminado()) // Si hemos llegado a la profundidad MAX
    return Valoracion(actual,jugador);
```

Ilustración 1. Condición de parada.

Posteriormente, se inicia un bucle que concluye cuando todos los hijos han sido explorados o cuando beta es menor o igual a alfa. Dentro del mismo se calcula el valor del nodo mediante una llamada recursiva al método de poda. Seguidamente, comprueba si el nodo a explorar es MIN o MAX, es decir, si el jugador actual es el introducido por parámetro o es el oponente, respectivamente.

En el primer caso, si el valor del nodo es mayor a alfa, se actualizará tanto el valor almacenado en la variable Alpha. Si el que juega es el oponente y se determina que el valor del nodo es más pequeño que beta, se actualizará beta. En ambos casos se debe modificar la variable estado al estado del hijo actual.

```
for(int i = 0; i< num_hijos; i++) {
    valor_nodo = Poda_AlfaBeta(hijo[i], jugador, profundidad+1, PROFUNDIDAD_ALFABETA, accion, alpha, beta);

    if (actual.JugadorActivo()==jugador) { //Estamos en un nodo MAX

        if (valor_nodo > alpha) {
            alpha = valor_nodo;
            estado = hijo[i];
        }

    } else { // Si no, estamos en un nodo MIN

        if (valor_nodo < beta) {
            beta = valor_nodo;
            estado = hijo[i];
        }
    }

    if (beta <= alpha) // Si beta es menor que alpha hemos terminado
            break;

    if(profundidad == 0){   // Control de errores
        accion = static_cast <Environment::ActionType>(estado.Last_Action(jugador));
        cout << "\tValor: " << valor_nodo << "\tAccion: " << accion +1<< endl;
    }
}
```

Ilustración 2. Implementación del bucle.

2.1 Heurística

La función heurística utilizada en este caso es muy sencilla. Primero se revisa el tablero y se establece un ganador según el valor retornado. Si el resultado es 1, ganamos nosotros, si es 2, perdemos y si es 0 se ha llegado a un empate global. En

cualquier otro caso, es decir, mientras dure la partida, se aplica la función heurística que explicaremos a continuación.

```
// Funcion heuristica (ESTA ES LA QUE TENEIS QUE MODIFICAR)
double Valoracion(const Environment &estado, int jugador){

    int ganador = estado.RevisarTablero();

    if (ganador==jugador)
        return 10000.0;  // Gana el jugador que pide la valoracion
    else if (ganador != 0)
        return -10000.0; // Pierde el jugador que pide la valoracion
    else if (estado.Get_Casillas_Libres()==0)
        return 0;  // Hay un empate global y se ha rellenado completamente el tablero
    else
        return func_heur(estado,jugador);
}
```

Ilustración 3. Función Valoración

La función heurística devuelve la suma de todos los valores de las casillas del tablero, cosa que calcula mediante la función ValorCasilla. Ésta consiste en asignarle un número (un valor) a todas las casillas del tablero en función de lo que tengan a su alrededor. Primero se determina a quién pertenece la casilla seleccionada, si es 1 o 4 pertenecerá al jugador 1, si es 2 o 5, al jugador 2 y si es 0 la casilla estará vacía, así que consideramos que es "nuestra" por el momento.

Posteriormente, analizaremos las casillas que rodean a la seleccionada. En este punto tuve un pequeño problema ya que no sabía cuántas casillas considerar; tras varios ensayos de prueba y error, encontré que analizar las casillas desde tres filas y tres columnas *(fil-3, col-3)* antes hasta tres filas y tres columnas más que cada casilla *(fil+3, col+3)* me proporcionaba los mejores resultados (pude ganarle a 4 de los 6 ninjas).

```
// Analizamos las casillas que rodean a la seleccionada ----------------------
    for (int j=col-3; j<=col+3; j++){
        for (int i=fila-3; i<=fila+3; i++){

            if((i!=fila || j!=col) && i>=0 && i<7 && j>=0 && j<7){  //Si la ca
                casillaCercana = estado.See_Casilla(i,j)%3;          //Almace

                if(casillaCercana == casilla && casilla == jugador)//Si la cas
                        valor+=4;
                else //En caso contrario reducimos la valoracion. Ya que estar
                        valor--;
            }
        }
    }
```

Ilustración 4. Bucle para calcular el valor de la casilla

Como podemos observar en la ilustración 6, primeramente se comprueba que la casilla cercana a la seleccionada sea correcta, es decir, que esté dentro de los límites y no sea la que estamos valorando. Posteriormente se almacena y se le hace módulo 3 para que, en caso de que sea una bomba, la considere como ficha verde (1) o azul (2), respectivamente. Luego se comprueba si es del mismo color que la que estamos

analizando y, en caso afirmativo, aumentamos su valor. Si es de un color distinto significaría que estamos ante una casilla del adversario, así que reducimos su valor, de forma que se le da prioridad a formar grupos de casillas del mismo color.

No quise hacer una heurística muy complicada que comprobase todas las diagonales, verticales y horizontales del adversario porque no lo vi necesario, pienso que la sencillez y la legibilidad del código son igualmente importantes y esta sencilla implementación hacía su trabajo igual de bien que otras mucho más extensas. Para mayor eficiencia, apliqué cambios en las variables en los bucles para que se recorrieran antes las columnas que las filas ya que en c se almacenan de esa manera.

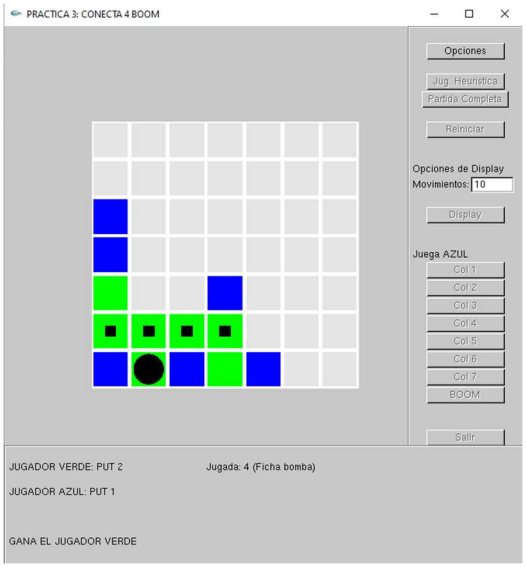


Ilustración 7. Contra ninja 1 como j1

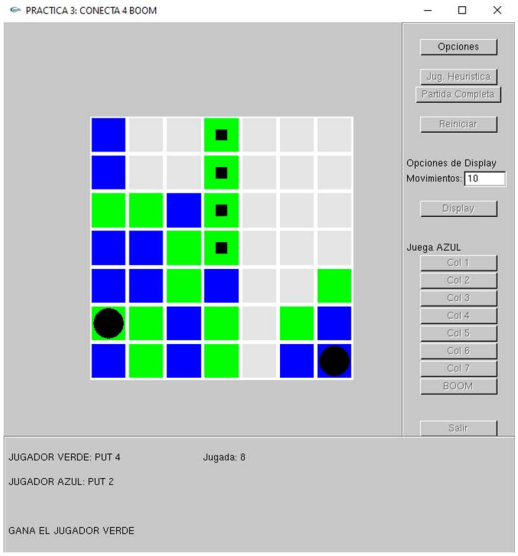


Ilustración 8. Contra ninja 2 como j1

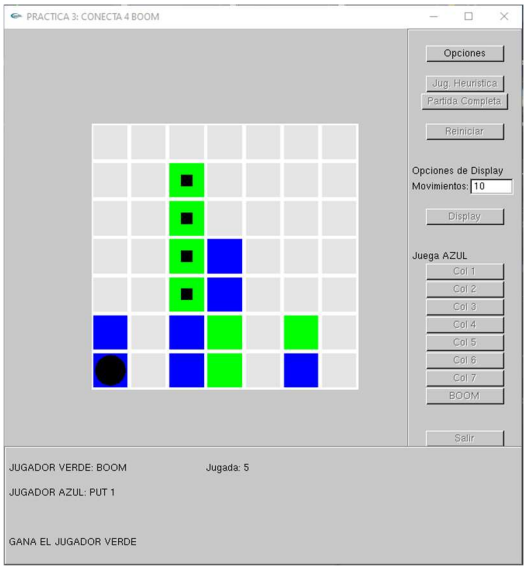


Ilustración 9. Contra ninja 3 como j1

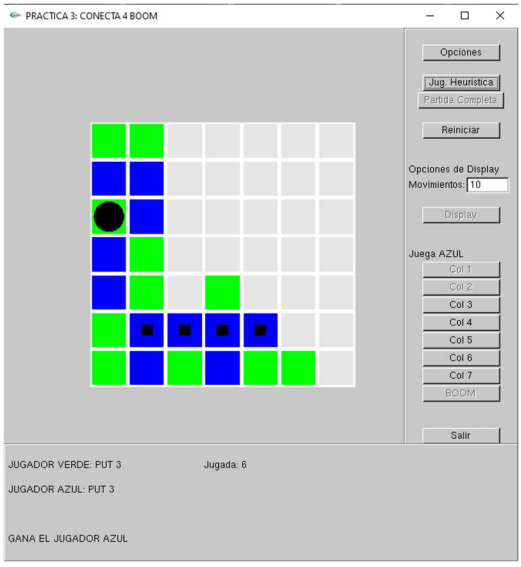


Ilustración 10. Contra ninja 2 como j2