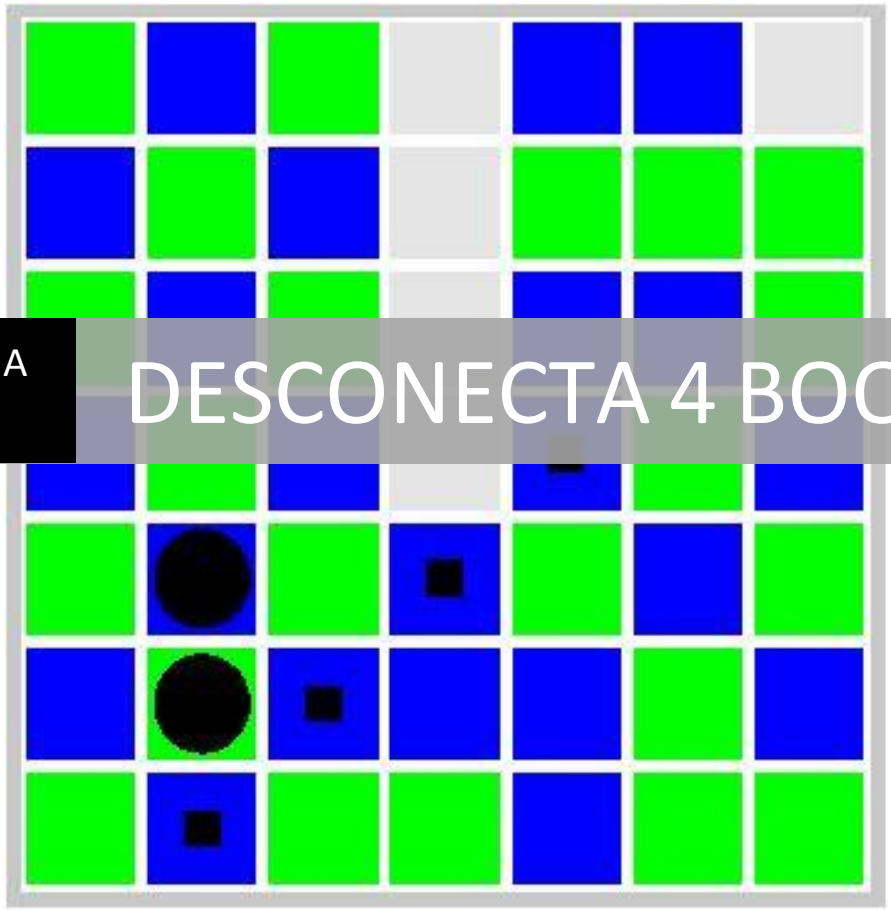
Junio de 2016

MEMORIA PRACTICA 3



INTELIGENCIA ARTIFICIAL

DESCONECTA 4 BOOM

Alberto Rodríguez Santana | DNI: 48950011P Grupo B1

Contenido

1. PRIMERAS IMPRESIONES - PROBLEMAS

En ésta tercera práctica de Inteligencia Artificial se han aplicado los conocimientos obtenidos en teoría acerca de las diferentes técnicas y métodos de búsqueda con adversario.

Entender estos métodos no ha sido una tarea tan difícil como el llevarlos a la práctica, que por una parte ha sido entretenido pero por otra parte un auténtico quebradero de cabeza.

En la clase de presentación de la práctica pareció ser una práctica fácil y entretenida a la cuál sabía que iba a tener que dedicarle muchísimo tiempo/esfuerzo después de la pésima nota obtenida en las dos anteriores, pero como he dicho 'pareció' puesto que de primeras empezaron los problemas que voy a comentar a continuación.

Pues el primer problema llegó a la hora de 'entender' la práctica, tarea no muy difícil, pero para lo que me han hecho falta aproximadamente más de 10 lecturas de la misma. En segundo lugar el mayor problema que he tenido ha venido después del correo recibido de parte del departamento cuyo contenido era una sentencia para comprobar la poda Alfa-Beta que ha sido la compatibilidad de la práctica en Linux en mi pc puesto que, después de bastantes vueltas al código, lo probé en el pc de un compañero y finalmente el problema estaba ahí, sin más, en el ordenador y no en el código (sin profundizar en el tema).

2. OBJETIVO

La práctica tiene como objetivo diseñar e implementar un agente deliberativo que pueda llevar a cabo un comportamiento inteligente dentro del juego DESCONECTA-4 que se explica a continuación.

El objetivo de DESCONECTA-4 BOOM es alinear cuatro fichas sobre un tablero formado por siete filas y siete columnas (en el juego original, el tablero es de seis filas). Cada jugador dispone de 25 fichas de un color (en nuestro caso, verdes y azules). Por turnos, los jugadores deben introducir una ficha en la columna que prefieran (de la 1 a la 7, numeradas de izquierda a derecha, siempre que no esté completa) y ésta caerá a la posición más baja. Gana la partida el primero que consiga alinear cuatro fichas consecutivas de un mismo color en horizontal, vertical o diagonal. Si todas las columnas están llenas pero nadie ha conseguido alinear 4 fichas de su color, entonces se produce empate.

A partir de estas consideraciones iniciales, el objetivo de la práctica es implementar un algoritmo MINIMAX con PODA ALPHA-BETA, con profundidad limitada (con cota máxima de 8), de manera que un jugador pueda determinar el movimiento más prometedor para ganar el juego, explorando el árbol de juego desde el estado actual hasta una profundidad máxima de 8 dada como entrada al algoritmo.

También forma parte del objetivo de esta práctica, la definición de una heurística apropiada, que asociada al algoritmo implementado proporcione un buen jugador artificial para el juego del DESCONECTA-4 BOOM.

3. IMPLEMENTACION PODA ALFA-BETA

El principal objetivo de esta práctica es la implementación de la poda Alfa-Beta, con profundidad limitada con una cota máxima de 8.

La poda alfa-beta es un algoritmo de búsqueda reduce el numero de nodos evaluados en un árbol de juego por el algoritmo minimax. Utiliza dos variables a lo largo del proceso de búsqueda en profundidad que son alfa y beta, en mi caso las he llamado alpha y beta de tipo doublé. Al principio del proceso de búsqueda, alfa toma el valor –infinito y beta toma el valor +infinito (inicializados al final de la función 'THINK').

En primer lugar comprobamos si estamos en un nodo terminal o el juego ya ha terminado y hacemos una valoración para ver el jugador que resulta ganador con la función ValoraciónTest con parámetros el tablero y el jugador.

```
if(profundidad == profundidad_max || tablero.JuegoTerminado()) {
                    return ValoracionTest(tablero,jugador);
}
```

Declaro la variable entera ult_act y la inicializo a -1 para la primera vez que llamo a GenerateNextMove() tal y como está indicado en el guión.

Con el bucle while vamos a recorrer el árbol de juego mientras ult_acc sea menor que 8 puesto que las acciones que se pueden realizar son únicamente de 0 a 7 siendo 0 PUT_1, 1 PUT_2…6 PUT_7, 7 BOOM.

Dentro de este bucle while voy a comprobar en primer lugar, que nos encontramos en un nodo MAX y asi iré comparando el valor obtenido en cada arco de los nodos que cuelgan con el valor actual de alfa, si el valor obtenido es mayor, este será el nuevo valor de alpha por tanto almacenará el mejor valor que encontremos desde el nodo en que estamos y solo seguiré la búsqueda desde ese nodo si se puede encontrar un valor mejor para alpha.

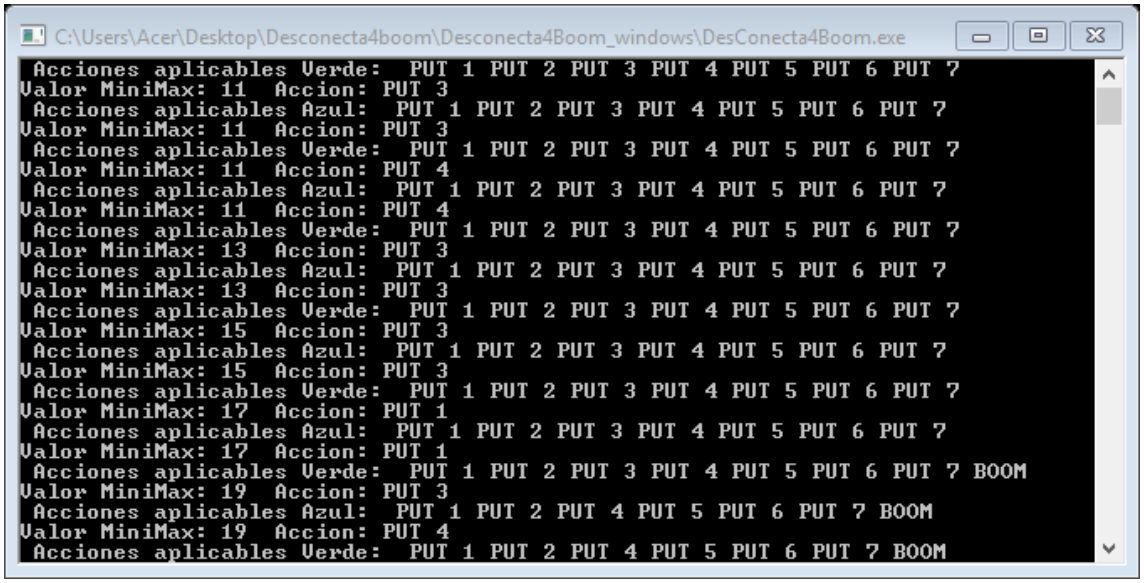
```
while(ult_act < 8){
        valor = Poda_AlfaBeta(hijo,jugador,profundidad+1,profundidad_max, accion_anterior,
alpha, beta);
        if (profundidad%2==0){  //Aquí es donde veo si el nodo es MAX
                if(valor > alpha) {
                        alpha = valor;
                        accion = static_cast <Environment::ActionType > (ult_act);
                }

                If(alpha>=beta){
                        return alpha;
                }
        }
```

En el caso del nodo MIN iré actualizando el valor de beta con el mismo proceso explicado anteriormente para alpha pero en este caso al contrario, si el valor encontrado es menor ese será el nuevo valor para beta.

```
else{ //Y ahora en este caso, estaríamos en un nodo MIN.
        if(valor < beta) {
                beta = valor;
                accion = static_cast <Environment::ActionType > (ult_act);
        }

        if(beta<=alpha){//condiccion de poda
                return beta;
        }

}
hijo = tablero.GenerateNextMove(ult_act);
```

y ahora por tanto guardamos en hijo (variable tipo Environment) el nuevo movimiento que el jugador al que le toca jugar sobre el tablero actual puede realizar.

La sentencia resultante al testear la poda Alfa-Beta en algunos pasos es la siguiente:



Al igual que la verificación recibida al correo:

Valor MiniMax: 11 Accion: PUT 3

Valor MiniMax: 11 Accion: PUT 3

Valor MiniMax: 11 Accion: PUT 4

Valor MiniMax: 11 Accion: PUT 4

Valor MiniMax: 13 Accion: PUT 3

Valor MiniMax: 13 Accion: PUT 3

Valor MiniMax: 15 Accion: PUT 3

Valor MiniMax: 15 Accion: PUT 3

Valor MiniMax: 17 Accion: PUT 1

Valor MiniMax: 17 Accion: PUT 1

Valor MiniMax: 19 Accion: PUT 3

Valor MiniMax: 19 Accion: PUT 4…

4.HEURISTICA (CÓDIGO)

```
//FUNCION MI PUNTUACION (Heuristica)
//Cuantas más fichas juntas, peor es.

double MiPuntuacion(int jugador, const Environment &estado){

   double suma=0;
   double fichas_seguidas=0;

   int enemigo; //Variable con la que sabremos el juegador contrincante con la siguiente
condicion.


   if(jugador==1)
      enemigo=2;
   else
      enemigo=1;



//-------Fichas seguidas para JUGADOR----------.

   //Fichas seguidas horizontales para jugador.
   for (int i=0; i<7; i++) {
     for (int j=0; j<7; j++){
        if (estado.See_Casilla(i,j)==jugador){
          fichas_seguidas++;
          suma=suma-fichas_seguidas;
        } else if (fichas_seguidas>0 && estado.See_Casilla(i,j)==enemigo) {
          suma=suma+fichas_seguidas;
          fichas_seguidas=0;
        } else {
          fichas_seguidas=0;
        }
     }
     fichas_seguidas=0;
   }

   //Fichas seguidas verticales para jugador.
   for (int i=0; i<7; i++) {
     for (int j=0; j<7; j++){
        if (estado.See_Casilla(j,i)==jugador){
          fichas_seguidas++;
          suma=suma-fichas_seguidas;
        } else if (fichas_seguidas>0 && estado.See_Casilla(j,i)==enemigo) {
          suma=suma+fichas_seguidas;
          fichas_seguidas=0;
        } else {
          fichas_seguidas=0;
```

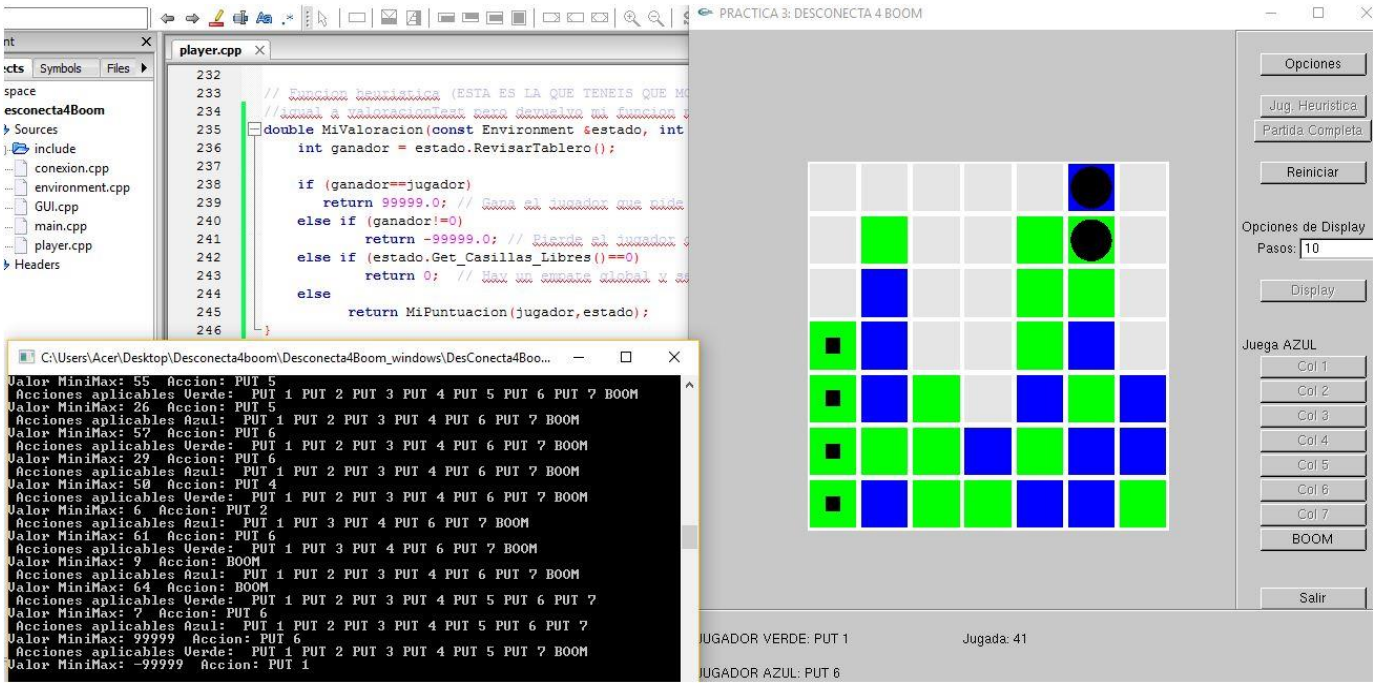
```
      }
    }
    fichas_seguidas=0;
  }


  // Diagonal  (derecha-abajo, izda-arriba)
  for (int i=0; i<4; i++) {
     for (int j=3; j<7; j++) {
        for(int k=0; k<4; k++) {
           if (estado.See_Casilla(i+k,j-k)==jugador) {
              fichas_seguidas++;
              suma=suma-fichas_seguidas;
           } else if (fichas_seguidas>0 && estado.See_Casilla(i+k,j-k)==enemigo) {
              suma=suma+fichas_seguidas;
              fichas_seguidas=0;
           } else {
              fichas_seguidas=0;
           }
        }
        fichas_seguidas=0;
     }
  }


  // Diagonal (derecha-arriba, izda-abajo)
  for (int i=0; i<4; i++) {
     for (int j=0; j<4; j++) {
        for(int k=0; k<4; k++) {
           if (estado.See_Casilla(j+k,i+k)==jugador) {
              fichas_seguidas++;
              suma=suma+fichas_seguidas;
           } else if (fichas_seguidas>0 && estado.See_Casilla(j+k,i+k)==enemigo) {
              suma=suma-fichas_seguidas;
              fichas_seguidas=0;
           } else {
              fichas_seguidas=0;
           }
        }
        fichas_seguidas=0;
     }
  }

//-------------------------------------------------------
```

```
//-------Fichas seguidas para ENEMIGO----------.

  // Fichas Horizontales para enemigo
  for (int i=0; i<7; i++) {
   for (int j=0; j<7; j++){
     if (estado.See_Casilla(i,j)==enemigo){
       fichas_seguidas++;
       suma=suma+fichas_seguidas;
     } else if (fichas_seguidas>0 && estado.See_Casilla(i,j)==jugador) {
       suma=suma-fichas_seguidas;
       fichas_seguidas=0;
     } else {
       fichas_seguidas=0;
     }
   }
   fichas_seguidas=0;
  }

  // Fichas seguidas Verticales para enemigo
  for (int i=0; i<7; i++) {
   for (int j=0; j<7; j++){
     if (estado.See_Casilla(j,i)==enemigo){
       fichas_seguidas++;
       suma=suma+fichas_seguidas;
     } else if (fichas_seguidas>0 && estado.See_Casilla(j,i)==jugador) {
       suma=suma-fichas_seguidas;
       fichas_seguidas=0;
     } else {
       fichas_seguidas=0;
     }
   }
   fichas_seguidas=0;
  }

  // Diagonal_1  (derecha-abajo, izda-arriba)
  for (int i=0; i<4; i++) {
     for (int j=3; j<7; j++) {
       for(int k=0; k<4; k++) {
         if (estado.See_Casilla(i+k,j-k)==enemigo) {
           fichas_seguidas++;
           suma=suma+fichas_seguidas;
         } else if (fichas_seguidas>0 && estado.See_Casilla(i+k,j-k)==jugador) {
           suma=suma-fichas_seguidas;
           fichas_seguidas=0;
         } else {
           fichas_seguidas=0;
         }
       }
       fichas_seguidas=0;
     }
  }
```

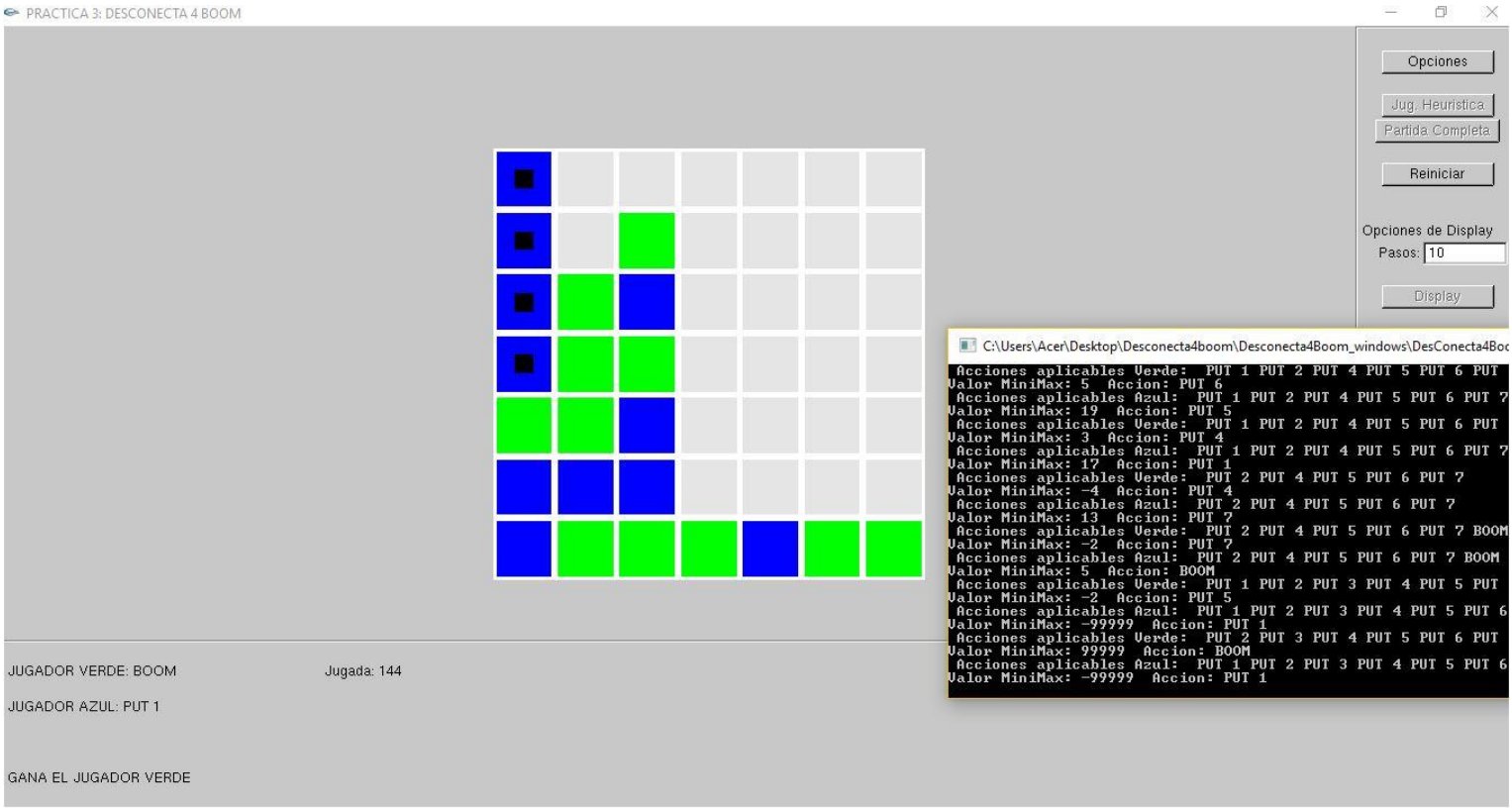
```
  // Diagonal_2  (derecha-arriba, izda-abajo)
   for (int i=0; i<4; i++) {
      for (int j=0; j<4; j++) {
         for(int k=0; k<4; k++) {
            if (estado.See_Casilla(j+k,i+k)==enemigo) {
               fichas_seguidas++;
               suma=suma+fichas_seguidas;
            } else if (fichas_seguidas>0 && estado.See_Casilla(j+k,i+k)==jugador) {
               suma=suma-fichas_seguidas;
               fichas_seguidas=0;
            } else {
               fichas_seguidas=0;
            }
         }
         fichas_seguidas=0;
      }
   }

//---------------------------------------------------------


   // FICHAS EN EL CENTRO
   for (int i=0; i<7; i++) {
     for (int j=0; j<7; j++){
        if (estado.See_Casilla(i,j)==jugador){
          if(j<3)
             suma=suma-j;
          else
             suma=suma-6;
        }
     }
   }

   return suma;
}
```

5. EJEMPLOS DE EJECUCIÓN

He vencido a NINJA!!



Jugada completa acaba ganando Azul en la jugada 41.



¡¡UNA LOCURA!! 144 JUGADAS, cuando solo hay 25 fichas…