Alba Casillas Rodríguez
2ºA (A3)

PRÁCTICA 3

DESCONECTA 4-BOOM

3 de junio, 2019

La memoria de la práctica se dividirá en dos partes fundamentales: la explicación de la poda alfa-beta y la de la heurística utilizada.

Poda alfa-beta

Es una técnica de búsqueda que mejora al algoritmo **minimax;** ya que podemos calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juego.

El principio general se basa en considerar un nodo n en el árbol, de forma que el jugador tiene una opción de movimiento a ese nodo. Si el jugador tiene una mejor selección m en el nodo padre de n o en cualquier punto más lejano (teniendo en cuenta un valor de n decidido tras examinar a algunos de sus descendientes), podemos podarlo.

Parámetros que describen los límites sobre lo valores hacia atrás que aparecen a lo largo del camino:

α = el valor de la mejor opción (el más alto) que hemos encontrado hasta ahora en cualquier punto elegido a lo largo del camino para MAX.

β = el valor de la mejor opción (el más bajo) que hemos encontrado hasta ahora en cualquier punto elegido a lo largo del camino para MIN.

La búsqueda alfa-beta actualiza el valor de α y β según va recorriendo el árbol y poda las ramas restantes en un nodo tan pronto como el valor del nodo actual es peor que el actual valor α o β para MAX o MIN, respectivamente.

double Player::poda_AlfaBeta (const Environment & actual, int jugador, int profundidad, Environment::ActionType & accion, double alpha, double beta, bool& primerAnalisis);

Siendo esta la cabecera de nuestro algoritmo, inicialmente comprobamos para el primer estado justo después de que el rival haya colocado una ficha, si podemos obtener la victoria al **explotar la bomba** de tenerla, y si es así e algoritmo directamente devuelve la acción de la bomba y no recorre el árbol entero.

De no ser así, mediante el uso de un booleano evitamos que se vuelva a realizar esta comprobación y la poda alfa beta se realiza con normalidad.

A continuación, verificamos si el juego ha terminado o si ya se ha disminuido toda la profundidad establecida; de ser así, se llama a la **heurística** (explicada más adelante); si no, comprobamos si somos el nodo MAX
(**jugador == actual.JugadorActivo()**) o el nodo MIN.

En ambos casos, se generarán los hijos mediante un bucle while hasta que **no queden más hijos**, haciendo uso de una variable contador, o hasta que se cumpla la **condición de poda**
(**while (cont < hijosPosibles && beta > alpha){...}**).

Condición de poda: si α <= β, podar.

Los **hijos** se generan: ***Environment nodo;***
nodo = actual.GenerateNextMove(accion);

Este método, a parte de generar un hijo, aplica la siguiente acción que se puede hacer (si no puede hacerse ninguna acción más, devuelve el propio nodo que hace la llamada).

Tras esto, si es un **nodo MAX** llamamos a una **función auxiliar BuscaMax**, y si es un **nodo MIN**, a **BuscaMin**; para **actualizar** (si se puede) los valores de **α y β,** valores que devuelve el algoritmo.

```
BuscaMax (&valorMax, valor, &accion, mejor_accion)
   si valor > valorMax entonces:
      valorMax = valor;
      accion = casting(mejor_accion);
   Fin // si
Fin
```

```
BuscaMin(&valorMin, valor, &accion, mejor_accion)
        si valor < valorMin entonces:
          valorMin = valor;
          accion = casting(mejor_accion);
        Fin // si
Fin
```

La "**mejor_acción**" devuelve la última acción que menos pérdida produce.

En la poda alfa-beta, la variable acción tendrá al final de la ejecución del algoritmo el punto de partida que llevará al nodo de mejor ganancia, por eso se va actualizando.

La llamada de este método se realiza en el método principal de la IA:

Environment::ActionType Player::Think();

Devolviendo la acción elegida en cada turno de mi jugador.

Heurística

La idea principal es, tras asignar un valor al jugador y al rival, comprobar para cada uno el **número de fichas consecutivas** (buscando combinaciones de cuatro, tres y dos fichas) para, más tarde, calcular la diferencia entre el valor de mi situación y la del rival.

Para ello, llamamos seis veces (3 combinaciones * 2 jugadores) a la función:

double encuentraConsecutivas(const Environment &estado, int jugador, int nConsecutivas);

la cual, para cada ficha del tablero que corresponda al jugador recibido, suma las fichas consecutivas verticales, horizontales y diagonales; llamando a otras tres funciones auxiliares.

Dentro de las funciones auxiliares, comprobamos que la fila y columna no sobrepasen el tope del tablero, para evitar violaciones de segmento y tras esto, mediante un bucle for que itera **nConsecutivas** (es decir, el número de fichas que se quiere ver si son consecutivas) veces, comparamos si el color de nuestra ficha actual es el mismo que el de las fichas siguiente en vertical, diagonal u horizontal, sumando uno a una variable contador si se cumple la condición.

Ejemplo: Para comprobar las fichas consecutivas verticales:

```
for (int i = 0; i < nConsecutivas && sigue; i++){
       if (estado.See_Casilla(fila,columna) == estado.See_Casilla((fila+i),columna)){
       cuenta_consec++;
 }else{
        sigue = false;
  }
 }
```

Si el valor del contador es el mismo que el número de nConsecutivas pasado por argumento retornamos 1, y si no: 0.

Tras obtener los valores de las seis ejecuciones del método encuentraConsecutivas, comprobamos si el estado de juego ha finalizado, de forma que si somos ganadores devolveremos un valor muy alto (y si perdemos, uno muy bajo); si no, se **evalúan** las fichas consecutivas de cada jugador:

```
yo = (yo_cuatros*10 + yo_tres*100 + yo_dos*100000);
oponente = (rival_cuatros*100000 + rival_tres*100 + rival_dos*10);
return (oponente - yo);
```

Como nuestro **objetivo** es que el oponente conecte cuatro fichas de su color, **aumentamos la valoración del rival** de forma muy positiva en aquellas situaciones en las que **conecte más fichas** y, por contrario, al jugador se le dará menos valoración cuantas más fichas conecte.

El cálculo final de la heurística se realiza restando la valoración del oponente a la del jugador ya que el resultado positivo para el jugador está condicionado a que el oponente haga más lineas consecutivas de fichas; de esta forma, cuanto mayor sea la diferencia, mejor es el valor de la heurística.

Obtener un valor negativo en esta heurística supone tener muchas fichas consecutivas del color del jugador, estando más próximos a la derrota.

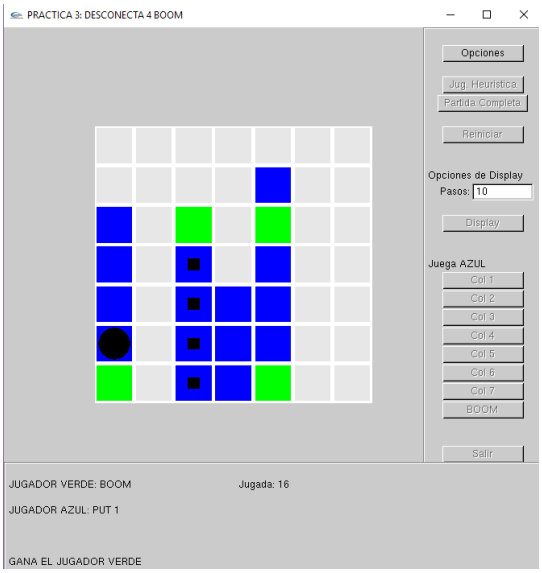
A pesar de que esta es la heurística definitiva, durante el desarrollo de la práctica he intentado implementar distintas reglas para mejorar las jugadas.

La **heurística inicial** consistía en la misma idea que he mantenido hasta el final: comparar fichas consecutivas. Sin embargo, aunque ganaba al ninja 3, no podía vencer a los primeros.

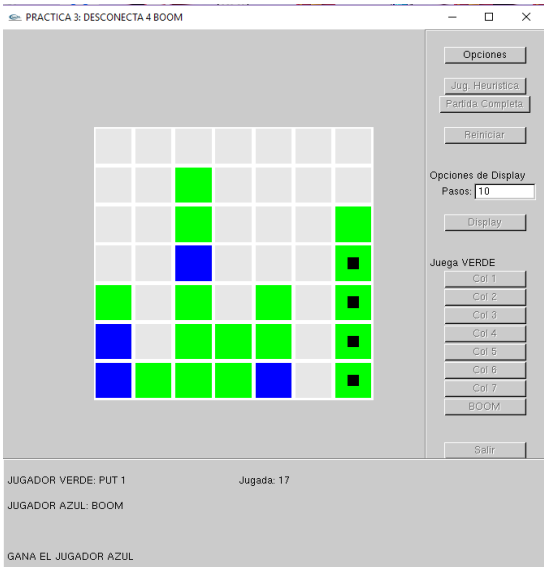
Para arreglar esto, he intentado **“jugar” con el uso de la bomba**, haciendo una “simulación” en la que explotaba; de forma que si al explotar mi jugador no perdía ni generaban tríos de fichas de mi color seguidos, se le asignaba un valor alto a la jugada. Al tener dificultades para implementar esto dentro de la función que ejecuta la heurística (a la hora de “arrastrar” la acción hacia arriba), incluí esta funcionalidad dentro del método **think()**, de manera que o se ejecutaba la explosión de la bomba, o se ejecutaba el algoritmo poda alfa-beta.

Aunque esto mejoró considerablemente el vencer a todos los ninjas, hubo uno de los casos (jugar contra el ninja 2 siendo el jugador 2), en el que tras numeras jugadas comenzaba a no buscar una solución.

Finalmente, el algoritmo es capaz de vencer a los tres ninjas tanto siendo el jugador1 como el jugador2.



Ganando al ninja 3 siendo el jugador 1



Ganando al ninja 3 siendo el jugador 2

- **Bibliografía**: S.Russell, P.Norving, Artificial Intelligence: A modern Approach, Segunda Edición, Ed. Pearson, 2004.