UNIVERSIDAD
DE GRANADA

DOBLE GRADO EN
INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

INTELIGENCIA ARTIFICIAL

Práctica 3. CONECTA-4 BOOM

Autores
José Miguel González Cañadas

Profesor
Juan Luis Suárez Díaz

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, mes de junio de 2021

Análisis del problema

Esta práctica consiste en el diseño de alguna de las técnicas de búsqueda con adversario en un entorno de juego real, donde el objetivo principal es la implementación del algoritmo *Minimax* o del algoritmo de poda *Alpha-beta*, para dotar de comportamiento inteligente deliberativo a un jugador artificial, de manera que esté en condiciones de competir y vencer a sus adversarios.

Debemos alinear cuatro fichas sobre un tablero conformado por siete filas y siete columnas. Cada jugador dispone de veinticinco fichas de un color. Las jugadas son alternas, empezando siempre el jugador *uno*, y en cada jugada se realizan dos movimientos, excepto el jugador *uno* en la primera jugada, en la que sólo realiza uno. [1]

El juego mantiene todas las normas del *Cuatro en raya* tradicional, incluida una variante: en el cuarto turno de cada jugador, se coloca una ficha especial de su color que llamaremos *ficha bomba*, pudiendo tener como máximo cada jugador una ficha de este tipo en el tablero. Ésta, al igual que el resto, sirve para confeccionar una posible alineación de cuatro fichas y así ganar la partida, con la peculiaridad de que el jugador la puede *explotar* en uno de los movimientos de su jugada.

Si ninguno de los dos jugadores consigue alinear cuatro fichas de su color, perderá aquel que le toque comenzar su jugada y no pueda realizar ningún movimiento.

¿Cómo se explota una ficha bomba?
Situación. El jugador debe realizar un movimiento y posee una ficha bomba sobre el tablero.
Actuación. Explotar la bomba. Esta acción consume un movimiento.

¿Qué efecto produce la explosión? La explosión elimina la propia *ficha bomba* y las de la misma fila que sean del adversario. Las fichas situadas sobre las casillas afectadas caerían por *gravedad* hasta situarse en sus posiciones estables.

Por último, cabe destacar que para el correcto desarrollo de la práctica se ha proporcionado un simulador, el cual servirá de interfaz gráfica para el juego que nos ocupa.

[1] Un movimiento consiste en introducir una ficha en una columna (de la *uno* a la *siete*, numeradas de izquierda a derecha) siempre que no esté completa, y ésta caerá a la posición más baja. Gana la partida el primer jugador que consiga alinear cuatro fichas consecutivas en horizontal, vertical o cualquier diagonal. Si todas las columnas están ocupadas se produce un empate técnico.

Algoritmo de poda Alpha-Beta

La definición de nuestro algoritmo de poda *Alpha-Beta* es la que sigue.

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

En primer lugar, verificamos si la partida ha finalizado ya o si se ha descendido en el árbol de juego toda la profundidad establecida. En este caso, devolvemos el valor de *Valoración*.

```
if (actual.JuegoTerminado() || profundidad == 0 || hijos == 0)
      return Valoracion(actual, jugador);
```

En caso contrario, comprobamos si somos un nodo *MAX* o *MIN*. En ambos casos, expandimos el nodo hasta que no queden más hijos por generar, o hasta que se cumpla la condición de poda. Posteriormente, llamamos a la función *max* o *min*, respectivamente, con el fin de actualizar los valores de *alpha* y *beta*.

```
if (jugador == actual.JugadorActivo()) {
    int cont = 0;
    while (cont < hijos && beta > alpha){
        cont++;
        max(alpha, (poda_AlfaBeta(nodo, jugador, profundidad-1 ,
    ultima_accion, alpha, beta)), accion, mejor_accion);
        nodo = actual.GenerateNextMove(mejor_accion);
    }
    return alpha;
} else {
    int cont = 0;
    while (cont < hijos && beta > alpha){
        cont++;
        min(beta, (poda_AlfaBeta(nodo, jugador, profundidad-1 ,
    ultima_accion, alpha, beta)), accion, mejor_accion);
        nodo = actual.GenerateNextMove(mejor_accion);
    }
    return beta;
}
```

donde (*max* es análogo),

```
void min (double & min, double value, Environment::ActionType & act,
    int best_act){
   if(value < min){
      min = value;
      act = static_cast< Environment::ActionType > (best_act);
   }
}
```

La secuencia de acciones devuelta nos situará en el nodo del árbol de juego con mayor valoración para nuestro jugador, con un valor máximo de profundidad

```
const int PROFUNDIDAD_ALFABETA = 8;
```

Heurística

Para implementar nuestra heurística hemos pasado una serie de etapas. Iremos desgranando cada una de ellas a continuación.

```
double consecutivas(const Environment &estado,int jugador,int nConsec){
   int contador = 0;

   for (int i=0; i<7; i++){
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(i,j) == jugador or estado.See_Casilla(i
    ,j) == jugador+3){
            contador += consecutivasVertical(i,j,estado,nConsec,jugador
    );
            contador += consecutivasHorizontal(i,j,estado,nConsec,
    jugador);
            contador += consecutivasDiagonal(i,j,estado,nConsec,jugador
    );
         }
       }
   }
   return contador;
}
```

Encontrar el número de combinaciones verticales ganadoras

En primer lugar, hemos calculado para cada situación posible del tablero de juego durante nuestra partida el número de combinaciones verticales ganadoras existentes. El algoritmo implementado en este método es el que sigue:

- Para cada casilla *(i,j)* del tablero de juego donde haya una ficha de nuestro jugador
  - Comprobamos las combinaciones ganadoras partiendo de un 2 en raya
    - Si existen dos fichas de nuestro jugador dispuestas una sobre otra
      - Comprobamos si existen otras dos casillas consecutivamente por encima de ellas y están vacías
  - Comprobamos las combinaciones ganadoras partiendo de un 3 en raya
    - Si existen tres fichas de nuestro jugador dispuestas una sobre otra
      - Comprobamos si existe otra casilla consecutivamente por encima de ellas y está vacía

```
int consecutivasVertical(int fila,int col,const Environment &estado,int
     nConsec,int jugador){
   int consec = 0;
   if (fila + nConsec- 1 < 7) {
      for (int i=0; i<nConsec;i++) {
         if (estado.See_Casilla(fila,col) == estado.See_Casilla((fila+i
    ),col) or
             estado.See_Casilla(fila,col) == estado.See_Casilla((fila+i)
    ,col)+3) {
                consec++;
         } else {
            break;
         }
      }
   }
   if (nConsec == consec) {
      return adyacentesVaciasVertical(fila+nConsec-1,col,estado,nConsec
    ,jugador);
   }
   return 0;
}
```

Tratamos cada combinación vertical ganadora como una *unidad*, y devolvemos entonces el número de combinaciones verticales ganadoras en la situación actual del tablero.

Análogamente, introducimos la búsqueda de combinaciones horizontales y diagonales ganadoras.

Encontrar el número de combinaciones horizontales ganadoras

- Para cada casilla *(i,j)* del tablero de juego donde haya una ficha de nuestro jugador
    - Comprobamos las combinaciones ganadoras partiendo de un 2 en raya
        - Si existen dos fichas de nuestro jugador dispuestas consecutivamente en horizontal
            - Comprobamos si existen otras dos casillas horizontalmente consecutivas a ellas, y están vacías, a ambos lados
            - Comprobamos si existe al menos una casilla horizontalmente consecutiva a ellas, y está vacía, a ambos lados
    - Comprobamos las combinaciones ganadoras partiendo de un 3 en raya
        - Si existen tres fichas de nuestro jugador dispuestas horizontalmente, consecutivas
            - Comprobamos si existe otra casilla horizontal y consecutiva a ellas, y está vacía, a ambos lados

```
int consecutivasHorizontal(int fila, int col, const Environment &
    estado, int nConsec, int jugador){
   int consec = 0;
   if (col + nConsec - 1 < 7){
      for (int i=0; i<nConsec; i++) {
         if (estado.See_Casilla(fila,col) == estado.See_Casilla(fila,(
    col+i)) or
            estado.See_Casilla(fila,col) == estado.See_Casilla(fila,(
    col+i))+3) {
               consec++;
         } else {
            break;
         }
      }
   }
   if (nConsec == consec){
      return
         ( adyacentesHorizontalDcha(fila, col+nConsec-1, estado,
    nConsec, jugador)
         + adyacentesHorizontalIzda(fila, col, estado, nConsec, jugador
    )
         + adyacentesHorizontalAmbosLados(fila, col, estado, nConsec,
    jugador) );
   }
   return 0;
}
```

Encontrar el número de combinaciones diagonales ganadoras

- Para cada casilla *(i,j)* del tablero de juego donde haya una ficha de nuestro jugador
  - Comprobamos las combinaciones ganadoras partiendo de un 2 en raya
    - Si existen dos fichas de nuestro jugador dispuestas consecutivamente en diagonal en la dirección marcada por el *slash*, /
      - Comprobamos si existen otras dos casillas diagonalemente consecutivas a ellas en dicha dirección, y están vacías, a ambos lados
      - Comprobamos si existe al menos una casilla hdiagonalemente consecutivas a ellas en dicha dirección, y está vacía, a ambos lados
    - Si existen dos fichas de nuestro jugador dispuestas consecutivamente en diagonal en la dirección marcada por el *backslash*, \
      - Comprobamos si existen otras dos casillas diagonalemente consecutivas a ellas en dicha dirección, y están vacías, a ambos lados

        - Comprobamos si existe al menos una casilla hdiagonalemente consecutivas a ellas en dicha dirección, y está vacía, a ambos lados

- Comprobamos las combinaciones ganadoras partiendo de un 3 en raya
    - Si existen tres fichas de nuestro jugador dispuestas consecutivamente en diagonal en la dirección marcada por el *slash*, /
        - Comprobamos si existe otra casilla diagonalemente consecutiva a ellas en dicha dirección, y está vacía, a ambos lados
    - Si existen tres fichas de nuestro jugador dispuestas consecutivamente en diagonal en la dirección marcada por el *backslash*, \
        - Comprobamos si existe otra casilla diagonalemente consecutiva a ellas en dicha dirección, y está vacía, a ambos lados

```
int consecutivasDiagonal(int fila, int col, const Environment & estado,
     int nConsec, int jugador){
   int
      consec = 0,
      nCombinaciones = 0;
   if (fila + nConsec - 1 < 7 and col + nConsec - 1 < 7){
      for (int i=0; i<nConsec;i++){
         if (estado.See_Casilla(fila,col) == estado.See_Casilla((fila+i
    ),(col+i)) or
            estado.See_Casilla(fila,col) == estado.See_Casilla((fila+i)
    ,(col+i))+3) {
               consec++;
         }
         else break;
      }
   }
   if (nConsec == consec){
      nCombinaciones =
         ( adyacentesAntidiagonalDcha(fila+nConsec-1, col+nConsec-1,
    estado, nConsec, jugador)
         + adyacentesAntidiagonalIzda(fila, col, estado, nConsec,
    jugador)
         + adyacentesAntidiagonalAmbosLados(fila, col, estado, nConsec,
     jugador) );
   }
   consec = 0;
   if (fila - nConsec + 1 >= 0 and col + nConsec - 1 < 7){
      for (int i=0; i<nConsec;i++){
         if (estado.See_Casilla(fila,col) == estado.See_Casilla((fila-i
    ),(col+i)) or
            estado.See_Casilla(fila,col) == estado.See_Casilla((fila-i)
    ,(col+i))+3) {
```

```
                consec++;
            }
            else break;
        }
    }
    if (nConsec == consec){
        nCombinaciones +=
            ( adyacentesDiagonalPrincipalDcha(fila, col+nConsec-1, estado,
      nConsec, jugador)
            + adyacentesDiagonalPrincipalIzda(fila+nConsec-1, col, estado,
      nConsec, jugador)
            + adyacentesDiagonalPrincipalAmbosLados(fila, col, estado,
     nConsec, jugador) );
    }
    return nCombinaciones;
}
```

Hemos tenido en cuenta la diagonal principal de la matriz, en sentido decreciente con respecto a la primera coordenada, y la antidiagonal, en sentido creciente. No es necesario tener en cuenta las otras dos diagonales puesto que ya han quedado cubiertas las dos direcciones existentes.

Función de valoración de un nodo

Una vez explicada la primera parte de nuestra heurística, procederemos a tratar cómo se produce la asignación de una valoración a un nodo en nuestra implementación.

En primer lugar, aplicamos la primera parte de nuestra heurística a la situación de juego actual y calculamos el número total de combinaciones ganadoras que se nos presentarían en el caso de llegar a esta situación, tanto para el rival como para nuestro jugador.

```
int
    ganador = estado.RevisarTablero(),
    rival = 1;
double
    rival_tres, rival_dos, jug_tres, jug_dos,
    jug, oponente;

if (jugador == 1) {
    rival = 2;
}

if (ganador==jugador)
    return 99999999.0;
else if (ganador!=0)
    return -99999999.0;
else if (estado.Get_Casillas_Libres()==0)
    return 0;
else {
```

```
    rival_tres = consecutivas(estado, rival, 3);
    rival_dos = consecutivas(estado, rival, 2);
    jug_tres = consecutivas(estado, jugador, 3);
    jug_dos = consecutivas(estado, jugador, 2);

    jug = (jug_tres + jug_dos);
    oponente = (rival_tres + rival_dos);
```

Obviamente, nos interesarán aquellos casos donde se maximicen nuestras combinaciones ganadoras y minimicen las de nuestro rival, por lo que un buen punto de partida podría ser maximizar *result*, definido como sigue.

```
    int result = jug-oponente;
```

Por otra parte, nos interesa beneficiar el juego por el centro del tablero. De esto se encarga el método *Puntuacion* proporcionado por la propia práctica.

```
    result += Puntuacion(jugador, estado);
```

En última instancia, cabe aclarar que cuando calculamos combinaciones ganadoras de cuatro en raya durante el desarrollo de la primera parte de nuestra heurística no estamos calculando aquellas situaciones en las que sólo nos haga falta una ficha extra para ganar. Por el contrario, únicamente estamos maximizando aquellas situaciones ganadoras en las que necesitaremos *al menos una* ficha extra para fizalizar la partida con una victoria.

Como cabría entonces esperar, nuestra última parte de la heurística irá dirigida a maximizar aquellas situaciones ganadoras en las que necesitemos el menor número de fichas extras posible para alcanzar la victoria; idealmente, intentamos maximizar aquellas situaciones ganadoras en las que únicamente necesitemos una ficha extra.

En este punto resulta conveniente centrarse en aquellas situaciones relativamente avanzadas de la partida, aquellas en las que el número de fichas colocadas sobre el tablero sea *suficientemente elevado*. Una posible solución podría ser entonces valorar al alza las situaciones del tablero en las que la mayor parte de las columnas están suficientemente ocupadas por fichas de nuestro jugador, beneficiando a aquellas posiciones que tengan un menor índice de entrada en la fila, esto es, a aquellas fichas colocadas en posiciones inferiores del tablero de juego.

```
for (int j=0;j<7;j++) {
          if (estado.Get_Ocupacion_Columna(j)>4) {
             for (size_t i =0; i<7; i++){
                if (estado.See_Casilla(i,j)==jugador) {
                   result += (7-i);
                }
             }
          }
      }
```

Esta heurística nos llevó a derrotar a todos los ninjas propuestos por la práctica, tanto en condición de jugador *MAX* como *MIN*.