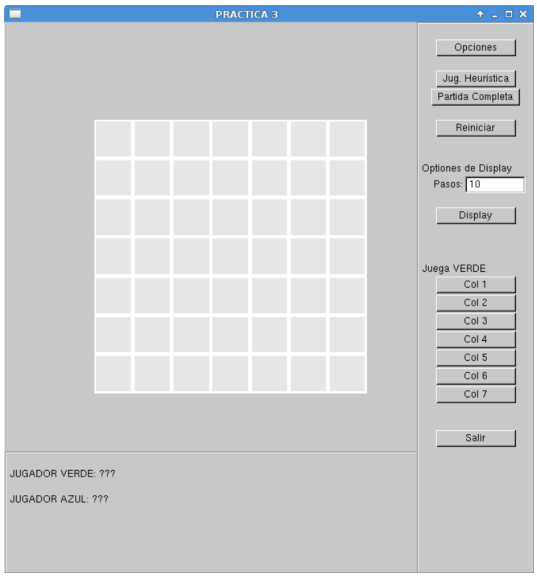
Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIONES

DESCONECTA-4 BOOM



Melguizo Marcos, Iñaki

1 Introducción

El problema que se plantea consiste en la implementación de técnicas de búsqueda con adversario en un entorno de juegos. En mi caso he implementado el algoritmo de la **poda alfa-beta** para que el jugador artificial de este juego esté dotado de un comportamiento deliberativo y sea capaz de vencer a su adversario (ser capaz de que su adversario coloque cuatro fichas consecutivas).

En resumen, he implementado poda alfa-beta con **profundidad de 8** para que un jugador pueda determinar el movimiento más prometedor para ganar la partida basado en una heurística que he considerado apropiada y que explicaré más adelante.

2 Implementación de poda alfa-beta

El algoritmo de poda alfa-beta es una técnica de búsqueda con adversario en un entorno de juegos. Con el algoritmo alfa-beta se puede obtener el mismo resultado que el algoritmo minimax pero con **menos esfuerzo computacional**.

Al principio, antes de iniciar el proceso de búsqueda he inicializado alfa a menos infinito y beta a más infinito.

```
double alpha, beta;
alpha=menosinf;
beta=masinf;
```

Habrá dos tipos de nodos, los **nodos MAX** que serán aquellos donde iremos actualizando el valor de alfa a medida que vayamos explorando cada arco que cuelgue de ese nodo y los **nodos MIN** donde actualizaremos el valor de beta. Identificaremos los nodos MAX porque serán aquellos en los que al jugador al que le toca “colocar una ficha” es el jugador maximizador y los nodos MIN serán aquellos en los que al jugador que le toca colocar una ficha no es el jugador maximizador.

```
if (maximizador){
best=max(best, val);
alpha=max(alpha,best);
if (beta<=alpha)
    break;
```

```
else{
best=min(best, val);
beta=min(beta, best);
if (beta<=alpha)
    break;
```

Alfa y beta se inicializan a menos infinito y más infinito respectivamente para que a la hora de hallar el máximo o el mínimo entre alfa o beta y best (que es el mejor valor que devuelve la función recursiva) la primera vez que se llame a la función recursiva se actualicen los valores de alfa y beta.

Con el break de la condición de que beta sea menor o igual que alfa se consigue que el coste computacional del algoritmo sea menor.

3 Implementación de la función alfa-beta

La función recursiva de poda alfabeta devolverá un double y se utilizará para hallar la **mejor acción a realizar** (poner una ficha en la columna 3, explotar la bomba, etc.). Los parámetros de la función son:

- **Enviroment e**: Representa el estado del juego.
- **bool maximizador**: True si es el turno del jugador maximizador y False en caso contrario.
- **int jugador_**: Devuelve un 1 si es el turno del jugador verde y 2 si es el turno del jugador azul. Es necesario este parámetro ya que es un parámetro que utiliza la heurística.
- **int depth**: Indica la profundidad en la que estamos en el árbol de búsqueda alfa-beta. Se utiliza para saber si hemos llegado a los nodos hoja, en cuyo caso tenemos que llamar a la función heurística o para saber si estamos en el nodo raíz, escenario en el cual tenemos que seleccionar la mejor acción posible.
- **const int PROFUNDIDAD_ALFABETA**: Devuelve la profundidad máxima que en este caso es 8.

- **Environment::AccionType accion**: Se pasa por referencia ya que en esa variable se almacenará la acción a realizar cuando se haya terminado de explorar el árbol de búsqueda.
- **double alfa**.
- **double beta**.

```
double alphabeta(Environment e, bool maximizador, int jugador_, int depth , const int PROFUNDIDAD_ALFABETA,Environment::ActionType &accion, double alpha, double beta)
```

1. En caso de que sea el turno del **jugador maximizador**, se generarán todos los posibles movimientos que se almacenararán en un la variable *vect*.

```
vect=new Environment[8];
int num=e.GenerateAllMoves(vect);
```

   Si cada uno de esos movimientos no provoca el fin de la partida (porque alguno de los jugadores haya obtenido 4 en raya) se llamará a la función *alphabeta*.

```
if (vect[i].RevisarTablero()==0)
    val=alphabeta(vect[i], false,((jugador_%2)+1),depth+1, PROFUNDIDAD_ALFABETA, accion, alpha, beta);
else if (vect[i].RevisarTablero()==jugador_)
    val=masinf;
else
    val=menosinf;
```

   Si el jugador ganador es el jugador actual devolverá masinf (cuanto mayor sea el valor devuelto mejor) y en caso de que el jugador actual sea el jugador perdedor devolverá menosinf.

   Si estamos en el **nodo raiz** (profundidad==0) tenemos que ver que valor devuelto es el mejor, para seleccionar la acción que se realizará, que es la finalidad de este algoritmo.

```
if (depth==0)
    valores[i]=val;
```

   Cuando se hayan almacenado todos los valores devueltos por la función *alphabeta* miramos cual de los valores devueltos es el mayor, lo que determinará la acción a realizar.

```
if (depth==0){

    bool encontrado=false;
    for (int j=0; j< num &&!encontrado; j++){
        if (valores[j]==best){

            accion = static_cast< Environment::ActionType > (vect[j].Last_Action(jugador_));
            encontrado=true;
        }

    }
}
```

2. En el caso de que sea el turno del **jugador no maximizador**, se modificará el valor de la variable beta cuando sea necesario (visto anteriormente) y no hará falta almacenar los valores devueltos ya que la profundidad de los nodos será mayor o igual que 1.

   Generamos los posibles movimientos.

```
vect=new Environment[8];
int num=e.GenerateAllMoves(vect);
```

   - En caso de que el juego no haya terminado, llamamos a la función *alphabeta* de nuevo.

```
if (vect[i].RevisarTablero()==0)
    val=alphabeta(vect[i], true, ((jugador_%2)+1), depth+1,PROFUNDIDAD_ALFABETA, accion, alpha, beta);
else if(vect[i].RevisarTablero()==jugador_)
    val=menosinf;
else val=masinf;
```

   - En caso de que el ganador sea el jugador actual devolveremos el valor menosinf, ya que el jugador actual es el jugador minimizador.
   - En el caso en el que estemos en profundidad 8 (la PROFUNDIDAD_ALFABETA) llamaremos a la **función heurística** que hemos creado:

```
if (depth==PROFUNDIDAD_ALFABETA){

    return Valoracion(e,jugador_);
}
```

   En la siguiente sección analizaremos la heurística que hemos creado.

4 Implementación de la función heurística

La heurística que he creado es bastante simple y se basa en **contar el número de 3 y 2-fichas alineadas que tiene cada jugador en el tablero**. Por cada 3-fichas alineadas que tenga el jugador maximizador sobre el tablero se le restará 100 a una variable y por cada 2-fichas alineadas que tenga el jugador maximizador se le restará 10 a esa variable. En caso de que sea el jugador minimizador el que tenga 3 o 2 fichas en línea, se sumará 100 y 10 a esa variable. Por lo tanto, como el objetivo es no tener 4 en línea, cuanto mayor sea el valor de esa variable será un mejor tablero para el jugador maximizador.

En caso de que el jugador maximizador haya ganado la partida, el valor de esa variable será masinf y en caso de que lo haya hecho el jugador minimizador el valor de dicha variable será menosinf.

```
if (e.RevisarTablero()==jugador)
    val=masinf;
else if(e.RevisarTablero()==((jugador%2)+1))
    val=menosinf;
else{
    int num_dos_jug=0;
    int num_tres_jug=0;
    heuristicaEnlinea(jugador, num_tres_jug, num_dos_jug, e);
    int num_dos_otro_jug=0;
    int num_tres_otro_jug=0;
    heuristicaEnlinea(((jugador%2)+1), num_tres_otro_jug, num_dos_otro_jug, e);

    val=((num_dos_otro_jug-num_dos_jug)*10+(num_tres_otro_jug-num_tres_jug)*100);
```

Para contar el número de 3-fichas alineadas que tiene cada jugador he hecho un bucle anidado para analizar todas las posiciones del tablero.

```
for (int fila=0; fila < 7 ; fila++){
    for (int columna=0; columna<7 ; columna++){
```

Ahora, si la posición que estamos analizando tiene columna menor que 5 mira a ver si las posiciones (fila, columna), (fila , columna+1), (fila, columna+2) están ocupadas por fichas del jugador que se la pasa a la función heurística como parámetro. En ese caso, el número de tres en línea se aumentaría en 1.

```
if (columna<5){
  if ((e.See_Casilla(fila, columna)%3)   == (e.See_Casilla(fila, columna+1)%3) and
      (e.See_Casilla(fila, columna+1)%3) == (e.See_Casilla(fila, columna+2)%3) and
      (e.See_Casilla(fila, columna+2)%3) == jugador){
      //cout << "En la misma fila\n";
      ntres++;
  }
}
```

- En caso de que la fila de la posición que estamos analizando sea menor que 5,

```
if (fila<5){
```

  si hay 3 fichas del jugador que estamos analizando alineadas verticalmente empezando desde esa posición aumentamos la variable *ntres* en 1.

```
ntres++;
```

- En caso de que la fila y la columna de la posición que estamos analizando sea menor que 5,

```
if (fila<5 and columna<5){
```

  se comprueba si el jugador tiene 3 fichas alineadas en una diagonal con pendiente positiva. En caso positivo aumentamos la variable *ntres* en 1.

```
ntres++;
```

- En caso de que la fila sea mayor que 1 y la columna menor que 5,

```
if (fila>1 and columna<5){
```

  se comprueba si el jugador tiene 3 fichas alineadas en una diagonal con pendiente negativa. En caso positivo aumentamos la variable *ntres* en 1.

```
ntres++;
```

Así, tendríamos en número de 3 en raya que tendría el jugador que estamos analizando. El número de 2 en raya se calcularía de una forma similar aunque habría que restarle dos veces el número de 3 en raya, ya que por cada 3 fichas en línea tenemos que hay 2 veces 2 en raya.

```
ndos-=2*ntres;
```

Como mejora a la heurística, había pensado que sería una buena idea que se le sumara a la variable duevuelta por la heurística un determinado valor en caso de que el jugador maximizador tuviera una bomba en el tablero para explotar (ya que da cierta ventaja) y le restara el mismo valor en caso de que el que tuviera la bomba fuera el jugador minimizador. Sin embargo esto provocaría que en algunos casos críticos no se explotase la bomba y tomara una decisión equivocada o que se le pasaran los 5 turnos que tiene el jugador para explotar la bomba.