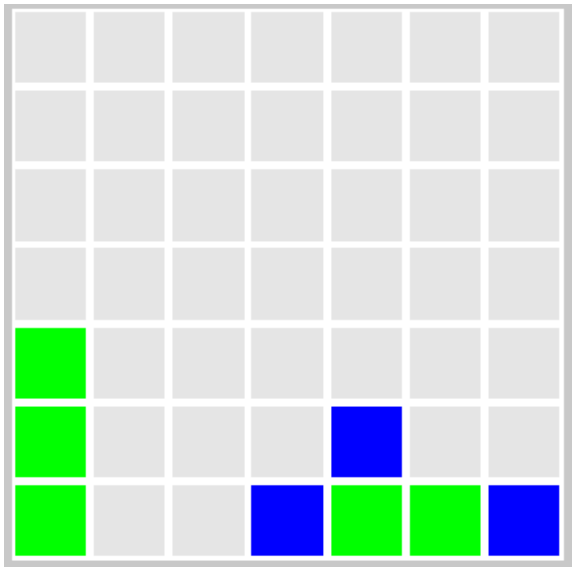
Práctica 3 IA
Desconecta 4 Boom

Doble Grado Informática y Matemáticas
Víctor Manuel Arroyo Martín

Elección de la heurística

La heurísitca más simple que se podría pensar es la de asignar un valor de -1 si se ha perdido, 1 si se ha ganado y 0 si se empata. Pero esta heurística es mala para discriminar entre buenas y malas soluciones ya que no es lo mismo que en un tablero empatado haya tres fichas tuyas en línea que tres del jugador contrario. Por ejemplo, en el siguiente tablero tendríamos más posibilidades de perder, pues tenemos más fichas en línea:



Así, llegué a la conclusión de que una heurística buena y a la vez sencilla sería aquella que cuente la cantidad de fichas en líneas de tres y de dos y las multiplique por un valor. En concreto, he usado los siguientes valores:

- 8 si el contrincante tiene tres fichas en línea
- 2 si tiene dos fichas en línea
- -2 si soy yo quien tiene dos en línea
- -8 si tengo tres

Estos valores se multiplican por el número de líneas de tres o de dos que hay en cada jugador y se suma todo. El valor será el valor del nodo en el árbol de jugadas.

Implementación de la función de valoración

Para contarlo, he creado una función llamada *NumFichasEnLinea* que es una variación de la función de Environment *EnLinea* haciendo que cuente sólo cuando hay tres o dos en línea. Respectivamente:

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

→ Ejemplo de contar cuántas líneas de tres hay, en este caso en la misma fila.

```
//En la misma Fila
if (columna<6){
  if ((estado.See_Casilla(fila,columna)%3)   == (estado.See_Casilla(fila,columna+1)%3)
      and (estado.See_Casilla(fila,columna+1)%3) == (jugador)){
      //cout << "En la misma fila\n";
    numdos++;
  }
}
```

→ Ejemplo de cuántas líneas de dos hay, en este caso en la misma fila.

El numdos-=2 se debe a que cuando está contando las fichas alineadas de dos, cuenta que hay dos veces dos alineadas por cada fila de tres así que hay que restarlo.

De esta forma, la función *Valoracion* queda como sigue:

```
double Valoracion(const Environment &estado, int jugador){
  int lineaCuatro=estado.RevisarTablero();
  if(lineaCuatro==jugador)
    return masinf;
  else if(lineaCuatro==(jugador%2)+1)
    return menosinf;
  int numtres=0, numdos=0, numdos_p=0, numtres_p=0; //Los _p son el otro jugador
  NumFichasEnLinea(estado, jugador, numdos, numtres);
  NumFichasEnLinea(estado, (jugador%2)+1, numdos_p, numtres_p);
  int valor3=-8, valor2=-2, valor3_p=8, valor2_p=2;
  return numdos*valor2+numtres*valor3+numtres_p*valor3_p+numdos_p*valor2_p;

}
```

RevisarTablero devuelve el jugador ganador o si hay empate, por lo que en caso que de ganemos nosotros devolvemos el mayor valor posible, pues esa secuencia de jugadas será muy buena para ganar. En caso de que perdamos devolvemos el menor valor, y si se empata o no se termina el juego devolvemos el valor que nos da la heurística.

Poda Alfa Beta

El algoritmo alfa beta es una variación del algoritmo minimax pero que poda ramas que no nos interesan porque ahí no se va a llegar a ninguna solución más prometedora. Se poda cuando beta es menor o igual que alfa. Así, si el algoritmo minimax es O(b^d), la poda alfa beta es O(b^d/2).
He optado por usar la poda alfa beta en vez del algoritmo minimax porque es más eficiente y no es mucho más difícil de implementar.
Lo he implementado de la siguiente manera:

```
double Poda_AlfaBeta(bool maximizingPlayer, const Environment &env, int jugador_, int depth , int profmax, Environment::ActionType &accion, double alpha, double beta){

  if(depth==profmax){
    return Valoracion(env,jugador_);
  }

  Environment *hijos;
  hijos=new Environment[8];
  int num_moves=env.GenerateAllMoves(hijos);
```

Primero, si la profundidad es la máxima, quiere decir que ese será un nodo del árbol al cual no se le generarán más hijos. Los hijos quedan guardados en un vector de hijos Environment generando todos los movimientos con la función que ya viene implementada *GenerateAllMoves.* El bool maximizingPlayer es para indicar si el jugador que se le pasa por parámetro somos nosotros (y por tanto maximizarlo) o es el otro jugador, que sería false.

```
for(int i=0; i<num_moves; i++){
  if(hijos[i].RevisarTablero()==0)
    val=Poda_AlfaBeta(false, hijos[i],(jugador_%2)+1, depth+1, profmax, accion, alpha, beta);
  else if(hijos[i].RevisarTablero()==jugador_)
    val=masinf;
  else
    val=menosinf;
    if(depth==0){
      valores0[i]=val;
    }
    best=max(best, val);
    alpha=max(alpha, best);
    best2=best;

    if(beta<=alpha){
      break;
    }
}
```

Ahora para todos los hijos hay que comprobar si el juego ha terminado y se usa el mismo criterio que en la función de valoración. En caso de empate o que no haya terminado, se procede con la recursividad para seguir construyendo el árbol.
Luego se actualizan las variables según el algoritmo alfa beta y si la beta es menor que la alfa, se realiza la poda, pues tendremos asegurado que los siguientes valores no van a mejorar en nada nuestra situación. Esto se hace con un break.

```
if(depth==0){
  bool encontrado=false;
  for(int j=0; j<num_moves && !encontrado; j++){
    if(valores0[j]==best2){
      encontrado=true;
      accion=static_cast<Environment::ActionType>(hijos[j].Last_Action(jugador_));
    }
  }
}
return best;
```

Como la jugada que queremos guardar es la que lleva la etiqueta de la raiz, comparamos en el nivel 0 cuál de todos los hijos tiene este valor y lo guardamos en una variable llamada acción, que será la que tomemos. Por último se devuelve el valor del nodo con return best.

```
}else{
  double best=masinf;
  double val;
  for(int i=0; i<num_moves; i++){
    if(hijos[i].RevisarTablero()==0)
      val=Poda_AlfaBeta(true, hijos[i],(jugador_%2)+1, depth+1, profmax, accion, alpha, beta);
    else if(hijos[i].RevisarTablero()==jugador_)
      val=menosinf;
    else
      val=masinf;
    best=min(best,val);
    beta=min(beta,best);

     if(beta<=alpha){
      break;
    }
  }
  return best;
}
```

Cuando el movimiento es del otro jugador, se hace lo mismo pero minimizando.

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

```
  Environment *hijos;
hijos=new Environment[8];
  int num_moves=actual_.GenerateAllMoves(hijos);
  bool ganadora=false;
  for(int i=0; i<num_moves && !ganadora; i++){
    if(hijos[i].RevisarTablero()==jugador_){
      ganadora=true;
      accion=static_cast<Environment::ActionType>(hijos[i].Last_Action(jugador_));
    }
  }



  // Opcion: Poda AlfaBeta
  // NOTA: La parametrizacion es solo orientativa
  if(!ganadora){
    double valor = Poda_AlfaBeta(true, actual_, jugador_, 0, PROFUNDIDAD_ALFABETA, accion, alpha, beta);
    cout << "Valor MiniMax: " << valor << "  Accion: " << actual_.ActionStr(accion) << endl;
}
delete [] hijos;


  return accion;
```

Por último, en el think, inicializamos alfa y beta a sus respectivos valores y llamamos a la función poda alfa beta para que nos devuelva la siguiente acción que debemos realizar. Antes de ello, si se puede hacer una jugada ganadora, ésta será la siguiente que se realice en vez de llamar a la poda.