Memoria Práctica 3
Búsqueda con Adversario (Juegos)
CONECTA-4 BOOM

César Muñoz Reinoso

2020-2021

1. Análisis del problema:

El problema a solucionar es el clásico 4 en raya pero con alguna modificación. Tenemos un tablero de 7x7 posiciones en el que ambos jugadores van colocando sus fichas:

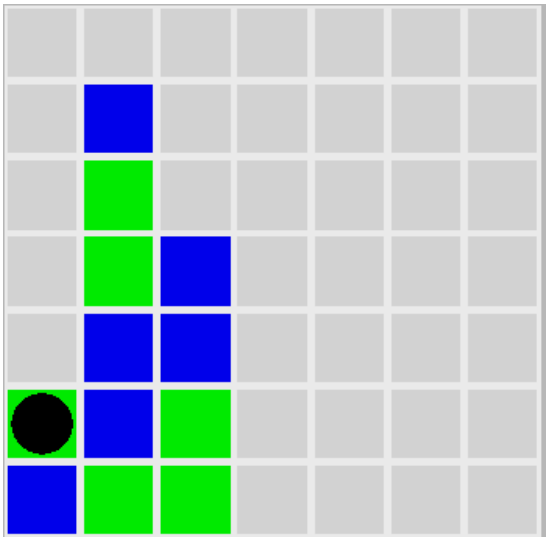
- El primer jugador coloca una ficha, luego el siguiente jugador coloca dos fichas, y a partir de ese momento, se alternan colocando 2 fichas cada uno.

- Cada 4 jugadas, cada jugador coloca una “ficha bomba”, que puede explotar gastando un turno de colocación. Esta ficha cuenta como una ficha normal para hacer 4 en raya, pero si la explota, elimina dicha ficha y todas las fichas del adversario en esa fila. Solo puede haber una ficha bomba de cada jugador en el tablero. Para volver a poner una, deberá explotarla antes.

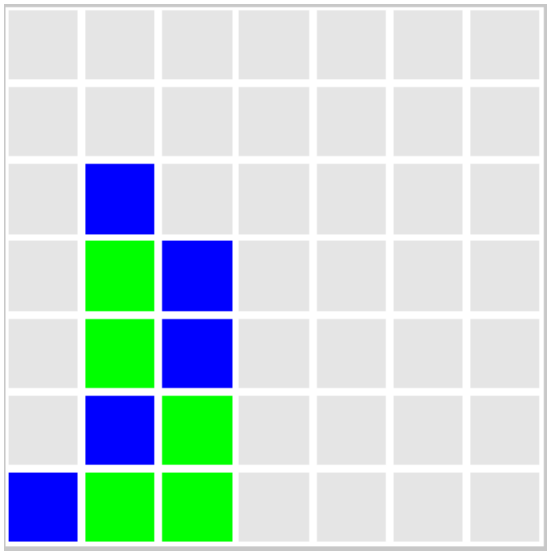
- En cada colocación de ficha, cada jugador puede colocar una ficha en alguna posición libre, de las 7 columnas posibles, o puede explotar la ficha bomba, si esta colocada.

- Gana el jugador que consiga hacer 4 fichas seguidas en filas, columnas o diagonal. Si se ocupan todas las casillas sin que nadie consiga ganar, se produce un empate.

ANTES



DESPUÉS



2. Descripción de la solución planteada:

2.1 Algoritmo Poda Alfa - Beta:

Para solucionar el problema, he optado por un algoritmo Poda Alfa-Beta, aportado en las transparencias de teoría.

Algoritmo ALFA-BETA

Para calcular el valor V(J,alfa,beta), hacer lo siguiente:

1. Si J es un nodo terminal, devolver V(J)=f(J). En otro caso, sean $J_1,\ldots,J_k,\ldots,J_b$ los sucesores de J. Hacer k ←1 y, si J es un nodo MAX ir al paso 2; si J es un nodo MIN ir al paso 5.
2. NODO MAX
   1. Hacer alfa ← max(alfa, $V(J_k$,alfa,beta)).
   2. Si alfa >= beta devolver beta (**¡criterio de poda!**); si no, continuar
   3. Si k=b, devolver alfa; si no, hacer k ← k+1 y volver al paso 2.
3. NODO MIN
   1. Hacer beta ← min(beta, $V(J_k$,alfa,beta)).
   2. Si beta <= alfa devolver alfa (**¡criterio de poda!**); si no, continuar
   3. Si k=b, devolver beta; si no, hacer k ← k+1 y volver al paso 5.

Para el cual se utiliza un árbol de juego, cuyos nodos representan los tableros del juego y las ramas representan las acciones de los jugadores,` de las 8 posibles que tenemos (1-7 casillas o BOOM).

El algoritmo Poda Alfa-Beta reduce el número de nodos evaluados en un árbol de juego por el algoritmo Minimax , donde se minimiza la perdida máxima, con la variante de utilizar dos parámetros alfa y beta para podar nodos.

- α es el valor de la mejor opción hasta el momento a lo largo del camino para MAX, esto implicará por lo tanto la elección del valor más alto
- β es el valor de la mejor opción hasta el momento a lo largo del camino para MIN, esto implicará por lo tanto la elección del valor más bajo.

En mi implementación, compruebo si me encuentro en un nodo de profundidad máxima establecida (profundidad 8 en nuestro caso) o si el juego ha concluido, en ese caso evalúo el nodo en cuestión con mi heurística, la función Valoración().

En otro caso, calculo el numero de nodos hijos posibles que tendrá el nodo actual. Dependiendo de cual sea el jugador actual, sabemos si es un nodo MAX o MIN. Vamos generando los nodos hijos J1, …, Jb. Para cada uno, vemos si el valor devuelto es mayor que alfa (en MAX) o si es menor que beta (en MIN), entonces asignamos el nuevo valor a el

parámetro y si estamos en el nodo raíz, asignamos esa acción para que sea devuelta a el jugador.

Una vez hemos visto todos los nodos hijos, devolvemos al padre el valor de alfa (en MAX) o beta (en MIN). Si en algún momento se produce la condición de poda, (alfa>= beta), se devuelve el valor de beta (en MAX) o alfa (en MIN).

double Player::Poda_AlfaBeta(Environment &actual, int jugador, int profundidad, const int profundidad_maxima, Environment::ActionType &accion, double alfa, double beta){

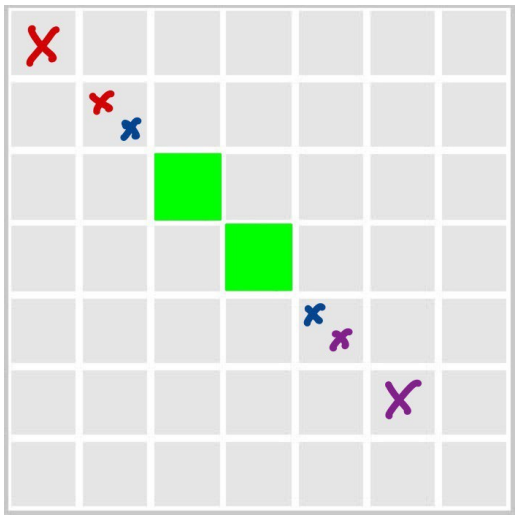
```
 int act = -1;
 Environment nodo_hijo;
 double valor_hijo = 0;
 Environment::ActionType accion_hijo;
 bool aux[8];
 int movimientos_totales = 0;

 if(profundidad == profundidad_maxima or actual.JuegoTerminado()){
  return Valoracion(actual, jugador);
 }else{

  movimientos_totales = actual.possible_actions(aux);
  nodo_hijo = actual.GenerateNextMove(act);

  if(jugador == actual.JugadorActivo()){ //Nodo MAX

   for(int i = 0; i < movimientos_totales; i++){

    valor_hijo = Poda_AlfaBeta(nodo_hijo, jugador, profundidad + 1, profundidad_maxima, accion_hijo, alfa, beta);
    if(alfa < valor_hijo){
     alfa = valor_hijo;
     if (profundidad == 0){
      accion = static_cast< Environment::ActionType > (act);
     }
    }
    if(alfa >= beta){
     return beta; //poda
    }
    nodo_hijo = actual.GenerateNextMove(act);
   }
   return alfa;
  }else{ //Nodo MIN

   for(int i = 0; i < movimientos_totales; i++){

    valor_hijo = Poda_AlfaBeta(nodo_hijo, jugador, profundidad + 1, profundidad_maxima, accion_hijo, alfa, beta);
    if(beta > valor_hijo){
     beta = valor_hijo;
     if (profundidad == 0){
      accion = static_cast< Environment::ActionType > (act);
     }
    }
    if(beta <= alfa){
     return alfa; //poda
    }
    nodo_hijo = actual.GenerateNextMove(act);
   }
   return beta;
  }
 }
}
```

2.2 Heurística utilizada:

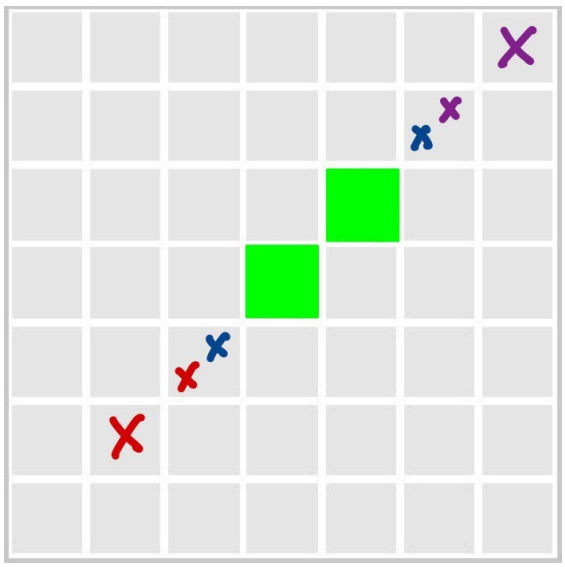
Como Heurística implementada hemos valorado lo siguiente:

Vamos viendo todas las casillas del tablero donde tenemos una ficha de nuestro color (función ValorCasilla), viendo 4 casos:

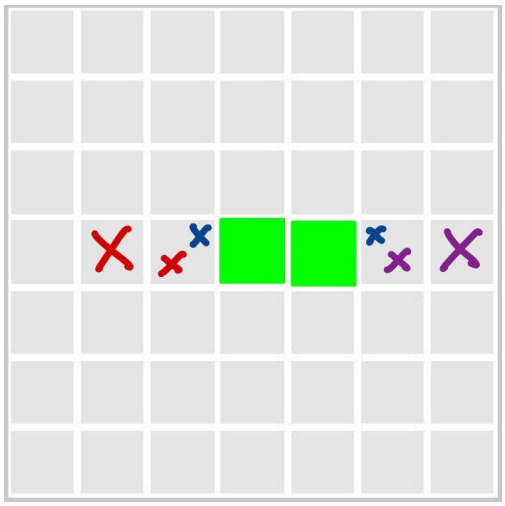
- Función Diagonales: vemos en la diagonal izquierda cuantas fichas tenemos. Si tenemos 2 fichas, vemos si las 2 casillas siguientes en ambas direcciones están libres y si la siguiente y la anterior están libres, en cuyo caso suman 1 a la variable valor.



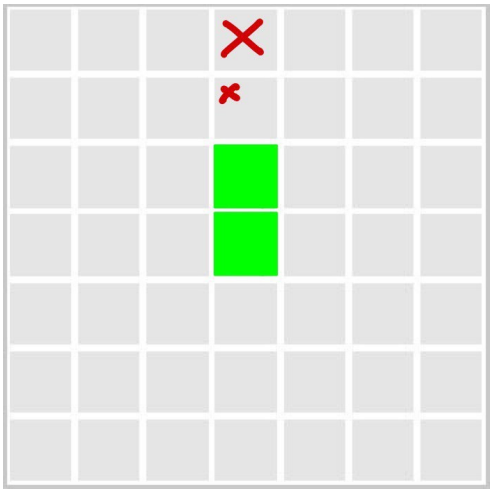
- Función Diagonales2 : vemos en la diagonal derecha cuantas fichas tenemos. Si tenemos 2 fichas, vemos si las 2 casillas siguientes en ambas direcciones están libres y si la siguiente y la anterior están libres, en cuyo caso suman 1 a la variable valor.



- Función Fila: vemos en la fila actual cuantas fichas tenemos. Si tenemos 2 fichas, vemos si las 2 casillas si las siguientes en ambas direcciones están libres y si la siguiente y la anterior están libres, en cuyo caso suman 1 a la variable valor.



- Función Columna: vemos en la columna actual cuantas fichas tenemos. Si tenemos 2 fichas, vemos si las 2 casillas de arriba están libres , en cuyo caso suman 1 a la variable valor.



Se repite el procedimiento para cuando tenemos 3 fichas seguidas en cuyo caso se ve la siguiente y la anterior ficha, salvo en la función Columna, que solo se ve la casilla de arriba

La función Valoración devuelve el valor del jugador actual menos el valor del jugador contrario.

return (ValorCasilla(estado, jugador) - ValorCasilla(estado, abs(jugador – 3));)

2.2 Ninjas:

El algoritmo Poda Alfa - Beta consigue ganar a ninja 1 como jugador verde y azul y a ninja 2 solo como jugador verde.