31-5-2014

PRÁCTICA 3 – CONECTA4

INTELIGENCIA ARTIFICIAL

Francisco Javier Bolívar Lupiáñez

2º GRADO EN INGENIERÍA INFORMÁTICA – GRUPO B1

Contenido

Análisis del problema

Descripción del problema

La práctica tiene como objetivo diseñar e implementar un agente deliberativo que pueda llevar a cabo un comportamiento inteligente dentro del juego **CONECTA-4** que se explica a continuación.

El objetivo de CONECTA-4 es alinear cuatro fichas sobre un tablero formado por siete filas y siete columnas (en el juego original, el tablero es de seis filas). Cada jugador dispone de 25 fichas de un color (en nuestro caso, verdes y azules). Por turnos, los jugadores deben introducir una ficha en la columna que prefieran (de la 1 a la 7, numeradas de izquierda a derecha, siempre que no esté completa) y ésta caerá a la posición más baja. Gana la partida el primero que consiga alinear cuatro fichas consecutivas de un mismo color en horizontal, vertical o diagonal. Si todas las columnas están llenas pero nadie ha conseguido alinear 4 fichas de su color, entonces se produce empate.

Objetivo

A partir de estas consideraciones iniciales, el objetivo de la práctica es implementar un algoritmo MINIMAX con **PODA ALPHA-BETA, con profundidad limitada** (con cota máxima de 8), de manera que un jugador pueda determinar el movimiento más prometedor para ganar el juego, explorando el árbol de juego desde el estado actual hasta una profundidad máxima de 8 dada como entrada al algoritmo.

También forma parte del objetivo de esta práctica, la definición de una heurística apropiada, que asociada al algoritmo implementado proporcione un buen jugador artificial para el juego del CONECTA4.

Diseño de la solución

Descripción del algoritmo minimax con poda alpha-beta

Pseudo-código de la solución:

```
alfabeta(nodo,jugador,profundidad,profmax,alpha,beta)
  IF prof == profmax || estado terminal THEN
    RETURN valoracion(nodo,jugador)
  genararmovimientos(nodo)
  IF jugador == max THEN
    FOR EACH movimiento posible
      alpha=max(alpha,alfabeta(hijo,jugador,profundidad+1,profmax,alpha,beta))
      IF beta <= alpha THEN
        RETURN alpha
     RETURN alpha
  ELSE
    FOR EACH movimiento posible
      beta=min(beta,alfabeta(hijo,jugador,profundidad+1,profmax,alpha,beta))
      IF beta <= alpha THEN
        RETURN beta
    RETURN beta
```

Descripción de la heurística usada

Mi heurística se podría dividir en tres subsecciones:

- **Fase ofensiva:** Por cada ficha en el tablero se sumará 1. Si forman dos seguidas, 2, y si forman tres, 3. Es decir, prima el tener las fichas juntas a tenerlas desperdigadas. Suponiendo que solo se observa una dirección (horizontal, vertical o una diagonal), si tienes en el tablero dos fichas juntas y una separada devolvería 4, pero si tienes las tres juntas, devolvería 6. También se tiene en cuenta si una fila de seguidas se acaba, es

decir, tienes dos seguidas y la siguiente ficha es del contrincante (XXO). En ese caso, se restará una unidad.

- **Fase defensiva:** Es inversamente proporcional a la ofensiva, se calcula la puntuación del enemigo, pero se resta en vez de sumar al total donde antes se sumaba y se suma donde antes se restaba.
- **Fichas en el centro:** Tener una ficha más orientada al centro da más posibilidad de crear cuatro en raya. Por tanto se premiará con tres unidades si la ficha la insertas en la columna 3, dos si la insertas en la 2 ó 4, uno si es en la 1 ó 5 y cero si es a los lados.

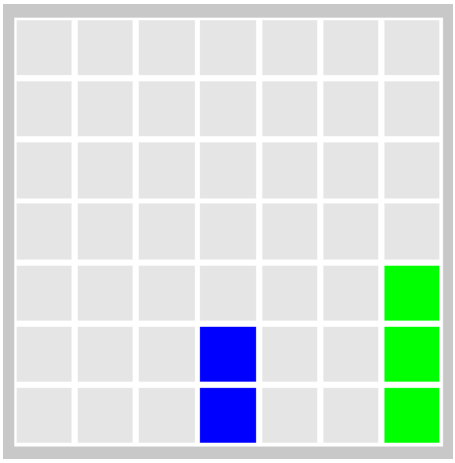
Pseudo-código de la solución:

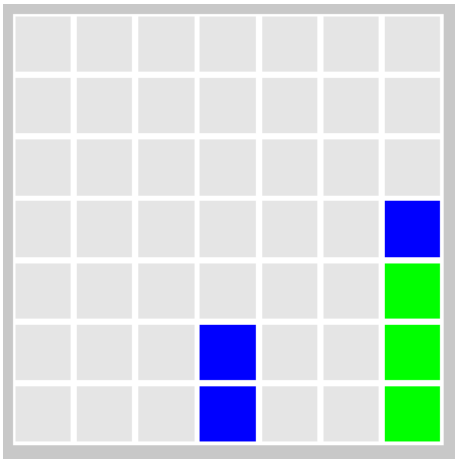
```
valoracion(nodo,jugador)
  ganador=revisartablero(nodo)
  IF ganador == jugador THEN
    RETURN 99999999
  ELSE IF ganador == enemigo THEN
    RETURN -99999999
  ELSE IF casillaslibres(nodo) == 0 THEN
    RETURN 0
  ELSE
  RETURN puntuacion(nodo,jugador)
```

```
puntuacion(nodo,jugador)
  suma=seguidas=0

  ##FASE OFENSIVA
  ##horizontal
  FOR EACH fila
    FOR EACH columna
      IF casilla == jugador THEN
        seguidas++
        suma+=seguidas
      ELSE IF casilla == enemigo and seguidas > 0 THEN
        suma-=seguidas
        seguidas=0
      ELSE
        seguidas=0
    seguidas=0
  ##Recorremos tambien vertical, horizontal y las dos diagonales

  ##FASE DEFENSIVA
  ##horizontal
  FOR EACH fila
    FOR EACH columna
      IF casilla == enemigo THEN
        seguidas++
        suma-=seguidas
      ELSE IF casilla == jugador and seguidas > 0 THEN
        suma+=seguidas
        seguidas=0
      ELSE
        seguidas=0
    seguidas=0
  ##Recorremos tambien vertical, horizontal y las dos diagonales

  ##FICHAS EN EL CENTRO
  FOR EACH fila
    FOR EACH columna
      IF casilla == jugador THEN
        IF columna<3 THEN
          suma+=columna
        ELSE
          suma+=(6-j)

  RETURN suma
```

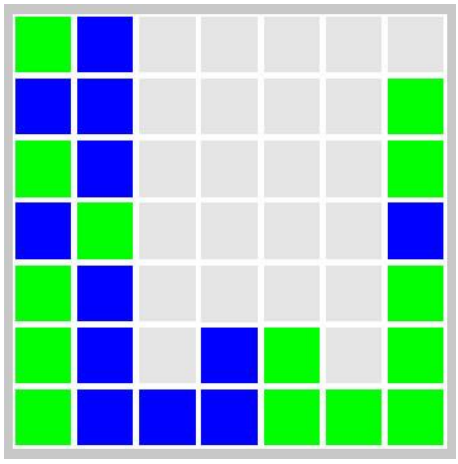
Ejemplos de ejecución

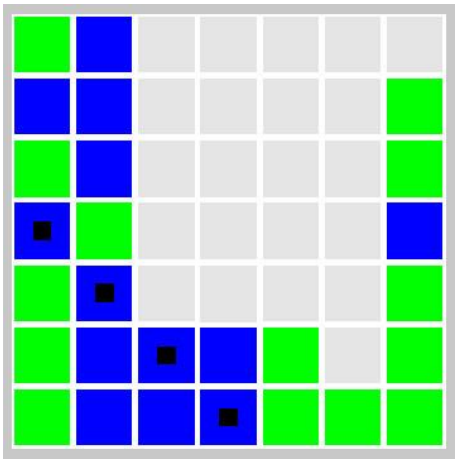
Supongamos el tablero de la izda. Lo lógico sería insertar la ficha en la última columna para **evitar que el oponente gane**, como vemos que hace:



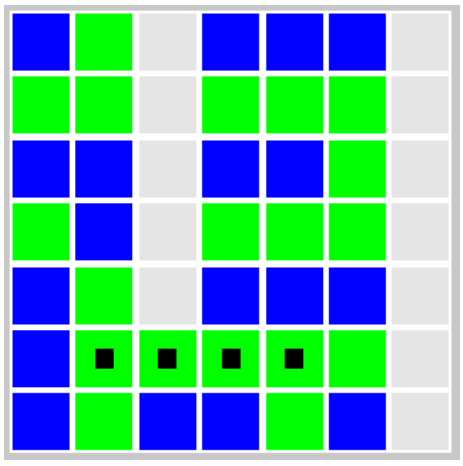


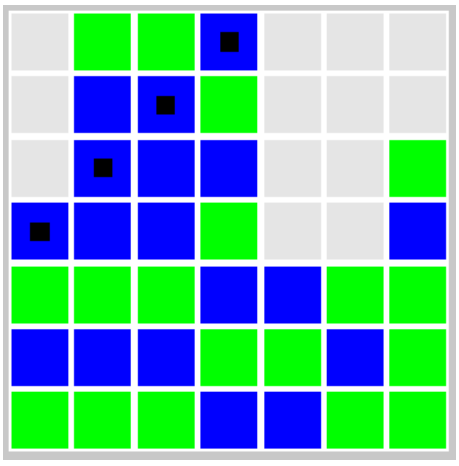
En esta otra sitación, se le presenta una **oportunidad de ganar y la aprovecha**.





Aquí vemos como también **vence a la heurística ninja** tanto como jugador 1 como jugador 2:





Código

player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include "environment.h"

class Player{
    public:
      Player(int jug);
      Environment::ActionType Think();
      void Perceive(const Environment &env);
    private:
      int jugador_;
      Environment actual_;
};
#endif
```

player.cpp

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <queue>
#include "player.h"
#include "environment.h"

using namespace std;

const double masinf=9999999999.0, menosinf=-9999999999.0;


// Constructor
Player::Player(int jug){
    jugador_=jug;
}

// Actualiza el estado del juego para el jugador
void Player::Perceive(const Environment & env){
    actual_=env;
}

double Puntuacion(int jugador, const Environment &estado){
    double suma=0;

    for (int i=0; i<7; i++)
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(i,j)==jugador){
            if (j<3)
               suma += j;
            else
               suma += (6-j);
         }
      }

    return suma;
}


// Funcion de valoracion para testear Poda Alfabeta
double ValoracionTest(const Environment &estado, int jugador){
    int ganador = estado.RevisarTablero();

    if (ganador==jugador)
       return 99999999.0; // Gana el jugador que pide la valoracion
    else if (ganador!=0)
            return -99999999.0; // Pierde el jugador que pide la valoracion
    else if (estado.Get_Casillas_Libres()==0)
            return 0;  // Hay un empate global y se ha rellenado completamente el
tablero
```

```
    else
          return Puntuacion(jugador,estado);
}

// ------------------- Los tres metodos anteriores no se pueden modificar

double MiPuntuacion(int jugador, const Environment &estado){
    double suma=0;
    double fichas_seguidas=0;
    int enemigo;
    (jugador==1) ? enemigo=2 : enemigo=1;

    // BUSQUEDA DE FICHAS SEGUIDAS

    // Horizontales
    for (int i=0; i<7; i++) {
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(i,j)==jugador){
            fichas_seguidas++;
            suma+=fichas_seguidas;
         } else if (fichas_seguidas>0 and estado.See_Casilla(i,j)==enemigo) {
            suma-=fichas_seguidas;
            fichas_seguidas=0;
         } else {
            fichas_seguidas=0;
         }
      }
      fichas_seguidas=0;
    }

    // Verticales
    for (int i=0; i<7; i++) {
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(j,i)==jugador){
            fichas_seguidas++;
            suma+=fichas_seguidas;
         } else if (fichas_seguidas>0 and estado.See_Casilla(j,i)==enemigo) {
            suma-=fichas_seguidas;
            fichas_seguidas=0;
         } else {
            fichas_seguidas=0;
         }
      }
      fichas_seguidas=0;
    }

    // Diagonal_1 / (derecha-abajo, izda-arriba)
    for (int i=0; i<4; i++) {
        for (int j=3; j<7; j++) {
            for(int k=0; k<4; k++) {
                if (estado.See_Casilla(i+k,j-k)==jugador) {
                    fichas_seguidas++;
                    suma+=fichas_seguidas;
                } else if (fichas_seguidas>0 and estado.See_Casilla(i+k,j-
k)==enemigo) {
                    suma-=fichas_seguidas;
                    fichas_seguidas=0;
                } else {
                    fichas_seguidas=0;
                }
            }
            fichas_seguidas=0;
        }
    }

    // Diagonal_2 \ (derecha-arriba, izda-abajo)
    for (int i=0; i<4; i++) {
        for (int j=0; j<4; j++) {
            for(int k=0; k<4; k++) {
                if (estado.See_Casilla(j+k,i+k)==jugador) {
                    fichas_seguidas++;
                    suma+=fichas_seguidas;
```

```
                } else if (fichas_seguidas>0 and
estado.See_Casilla(j+k,i+k)==enemigo) {
                    suma-=fichas_seguidas;
                    fichas_seguidas=0;
                } else {
                    fichas_seguidas=0;
                }
            }
            fichas_seguidas=0;
        }
    }


    // BUSQUEDA DE FICHAS ENEMIGAS SEGUIDAS

    // Horizontales
    for (int i=0; i<7; i++) {
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(i,j)==enemigo){
            fichas_seguidas++;
            suma-=fichas_seguidas;
         } else if (fichas_seguidas>0 and estado.See_Casilla(i,j)==jugador) {
            suma+=fichas_seguidas;
            fichas_seguidas=0;
         } else {
            fichas_seguidas=0;
         }
      }
      fichas_seguidas=0;
    }

    // Verticales
    for (int i=0; i<7; i++) {
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(j,i)==enemigo){
            fichas_seguidas++;
            suma-=fichas_seguidas;
         } else if (fichas_seguidas>0 and estado.See_Casilla(j,i)==jugador) {
            suma+=fichas_seguidas;
            fichas_seguidas=0;
         } else {
            fichas_seguidas=0;
         }
      }
      fichas_seguidas=0;
    }

    // Diagonal_1 / (derecha-abajo, izda-arriba)
    for (int i=0; i<4; i++) {
        for (int j=3; j<7; j++) {
            for(int k=0; k<4; k++) {
                if (estado.See_Casilla(i+k,j-k)==enemigo) {
                    fichas_seguidas++;
                    suma-=fichas_seguidas;
                } else if (fichas_seguidas>0 and estado.See_Casilla(i+k,j-
k)==jugador) {
                    suma+=fichas_seguidas;
                    fichas_seguidas=0;
                } else {
                    fichas_seguidas=0;
                }
            }
            fichas_seguidas=0;
        }
    }

    // Diagonal_2 \ (derecha-arriba, izda-abajo)
    for (int i=0; i<4; i++) {
        for (int j=0; j<4; j++) {
            for(int k=0; k<4; k++) {
                if (estado.See_Casilla(j+k,i+k)==enemigo) {
                    fichas_seguidas++;
```

```
                    suma-=fichas_seguidas;
                } else if (fichas_seguidas>0 and
estado.See_Casilla(j+k,i+k)==jugador) {
                    suma+=fichas_seguidas;
                    fichas_seguidas=0;
                } else {
                    fichas_seguidas=0;
                }
            }
            fichas_seguidas=0;
        }
    }

    // FICHAS EN EL CENTRO

    for (int i=0; i<7; i++) {
      for (int j=0; j<7; j++){
         if (estado.See_Casilla(i,j)==jugador){
            (j<3) ? suma+=j : suma+=(6-j);
         }
      }
    }

    return suma;
}

// Funcion heuristica
double Valoracion(const Environment &estado, int jugador){
    int ganador = estado.RevisarTablero();

    if (ganador==jugador)
       return 99999.0; // Gana el jugador que pide la valoracion
    else if (ganador!=0)
            return -99999.0; // Pierde el jugador que pide la valoracion
    else if (estado.Get_Casillas_Libres()==0)
            return 0;  // Hay un empate global y se ha rellenado completamente el
tablero
    else
          return MiPuntuacion(jugador,estado);
}

// Esta funcion no se puede usar en la version entregable
// Aparece aqui solo para ILUSTRAR el comportamiento del juego
// ESTO NO IMPLEMENTA NI MINIMAX, NI PODA ALFABETA
void JuegoAleatorio(bool aplicables[], int opciones[], int &j){
    j=0;
    for (int i=0; i<7; i++){
        if (aplicables[i]){
           opciones[j]=i;
           j++;
        }
    }
}

double Poda_AlfaBeta(Environment actual_, int jugador_, int profundidad, int
PROFUNDIDAD_ALFABETA, Environment::ActionType &accion, double alpha, double beta)
{

    if(profundidad==PROFUNDIDAD_ALFABETA || actual_.JuegoTerminado()  ) { //
Estado terminal
        return Valoracion(actual_,jugador_);
    }

    Environment V[7];
    int n_gen=actual_.GenerateAllMoves(V);
    double aux;

    if(profundidad%2==0) { // Nodo MAX

        for(int i=0; i<n_gen; i++) {
```

```
aux=Poda_AlfaBeta(V[i],jugador_,profundidad+1,PROFUNDIDAD_ALFABETA,accion,alpha,b
eta);
            if(aux>alpha) {
                alpha=aux;
                if(profundidad==0) accion=static_cast< Environment::ActionType
>(V[i].Last_Action(jugador_));
            }
            if(beta<=alpha) return alpha; // Poda
        }
        return alpha;

    } else { // Nodo MIN

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

aux=Poda_AlfaBeta(V[i],jugador_,profundidad+1,PROFUNDIDAD_ALFABETA,accion,alpha,b
eta);
            if(aux<beta) {
                beta=aux;
                if(profundidad==0) accion=static_cast< Environment::ActionType
>(V[i].Last_Action(jugador_));
            }
            if(beta<=alpha) return beta; // Poda
        }
        return beta;

    }

}

// Invoca el siguiente movimiento del jugador
Environment::ActionType Player::Think(){
    const int PROFUNDIDAD_MINIMAX = 6;  // Umbral maximo de profundidad para el
metodo MiniMax
    const int PROFUNDIDAD_ALFABETA = 8; // Umbral maximo de profundidad para la
poda Alfa_Beta

    Environment::ActionType accion; // acción que se va a devolver
    bool aplicables[7]; // Vector bool usado para obtener las acciones que son
aplicables en el estado actual. La interpretacion es
                        // aplicables[0]==true si PUT1 es aplicable
                        // aplicables[1]==true si PUT2 es aplicable
                        // aplicables[2]==true si PUT3 es aplicable
                        // aplicables[3]==true si PUT4 es aplicable
                        // aplicables[4]==true si PUT5 es aplicable
                        // aplicables[5]==true si PUT6 es aplicable
                        // aplicables[6]==true si PUT7 es aplicable



    double valor; // Almacena el valor con el que se etiqueta el estado tras el
proceso de busqueda.
    double alpha, beta; // Cotas de la poda AlfaBeta

    int n_act; //Acciones posibles en el estado actual


    n_act = actual_.possible_actions(aplicables); // Obtengo las acciones
aplicables al estado actual en "aplicables"
    int opciones[10];

    // Muestra por la consola las acciones aplicable para el jugador activo
    cout << " Acciones aplicables ";
    (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
    for (int t=0; t<7; t++)
      if (aplicables[t])
         cout << " " << actual_.ActionStr( static_cast< Environment::ActionType >
(t)  );
    cout << endl;
```

```
    //---------------------- COMENTAR Desde aqui
/*
    cout << "\n\t";
    int n_opciones=0;
    JuegoAleatorio(aplicables, opciones, n_opciones);

    if (n_act==0){
        (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
        cout << " No puede realizar ninguna accion!!!\n";
        //accion = Environment::actIDLE;
    }
    else if (n_act==1){
        (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
        cout << " Solo se puede realizar la accion "
             << actual_.ActionStr( static_cast< Environment::ActionType >
(opciones[0])  ) << endl;
        accion = static_cast< Environment::ActionType > (opciones[0]);

    }
    else { // Hay que elegir entre varias posibles acciones
        int aleatorio = rand()%n_opciones;
        cout << " -> " << actual_.ActionStr( static_cast< Environment::ActionType
> (opciones[aleatorio])  ) << endl;
        accion = static_cast< Environment::ActionType > (opciones[aleatorio]);
    }
*/
    //---------------------- COMENTAR Hasta aqui


    //---------------------- AQUI EMPIEZA LA PARTE A REALIZAR POR EL ALUMNO ------
-----------------------------------------


    // Opcion: Poda AlfaBeta
    // NOTA: La parametrizacion es solo orientativa
    valor = Poda_AlfaBeta(actual_, jugador_, 0, PROFUNDIDAD_ALFABETA, accion,
menosinf, masinf);
    cout << "Valor MiniMax: " << valor << "  Accion: " <<
actual_.ActionStr(accion) << endl;

    return accion;
}
```