Práctica 3: Desconecta4Boom.

Práctica realizada por Néstor Rodríguez Vico.

Índice:

1. Descripción del problema.

La práctica tiene como objetivo diseñar e implementar un agente deliberativo para el juego Desconecta4Boom. El objetivo de dicho juego es conseguir que el oponente alinee cuatro fichas sobre un tablero formado por siete filas y siete columnas. Por turnos, los jugadores deben introducir una ficha en la columna que prefieran o, de ser posible, explotar la ficha bomba. Gana la partida el primero que consiga que el oponente alinee cuatro fichas consecutivas de un mismo color en horizontal, vertical o diagonal. Si el tablero esta lleno pero no se ha dado la situación anterior, la partida termina en empate.

2. Objetivo.

El objetivo de la práctica es implementar la poda Alfa-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 juego del Desconecta4Boom.

3. Diseño de la solución: Algoritmo de poda Alfa-Beta.

```
double PodaAlfaBeta(const Environment & T, int jugador, int profundidad, double alpha, double beta,
                    Environment::ActionType & accion, bool maxORmin, double & numNodos) {
  si T es terminal o profundidad = 0
    numNodos++
    Valoracion(T, jugador)

  int ultacc = -1;
  Environment::ActionType acc_ant;
  double valor;
  Environment hijo = T.GenerateNextMove(ultacc);

  si maxORmin { (* Soy jugador MAX *)
    para cada hijo de T {
      valor = PodaAlfaBeta(hijo, jugador, profundidad - 1, alpha, beta, acc_ant, false, numNodos);
      if (valor > alpha){
        alpha = valor; accion = static_cast <Environment::ActionType > (ultacc);
      }
      if (beta <= alpha) break (* poda β *)
    }
    devolver alpha
  }
  si no { (* Soy jugador MIN *)
    para cada hijo de T {
      valor = PodaAlfaBeta(hijo, jugador, profundidad - 1, alpha, beta, acc_ant, true, numNodos);
      if (valor < beta) {
        beta = valor; accion = static_cast <Environment::ActionType > (ultacc);
      }
      if (beta <= alpha) break (* poda α *)
    }
    devolver beta
  }
}
```

La llamada inicial sería:

```
valor = PodaAlfaBeta(actual_, jugador_, PROFUNDIDAD_ALFABETA, infnito, -infnito, accion, true, numNodos);
```

4. Diseño de la solución: Función heurística y función valoración.

Dada la similitud de dicha versión del juego con la original (Conecta4) la función heurística usada es la misma que la que se usaría en el juego original pero con la valoración heurística opuesta. Es decir, si en el juego original nos interesaba que la función heurística para un tablero en el que se podría formar un 4 en raya fuese alta, aquí nos interesa todo lo contrario, que sea baja, ya que esto supondría una derrota. La función de valoración sería la siguiente:

```
double Valoracion(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 {
    //Si no he ganado ni perdido ni empatado aplicamos la heurística
    return Heuristica(jugador,estado);
  }
}
```

Dado que la explicación de la función heurística es más clara sabiendo lo que hacen las funciones auxiliares usadas (Horizontal, Vertical, Diagonal) empecemos primero por dichas funciones.

Estas tres funciones son las encargadas de contabilizar la cantidad de fichas seguidas que hay seguidas en horizontal, vertical y diagonal (las fichas bomba también son contadas en esta tarea). En función del número de fichas seguidas otorgamos un mayor o menor valor a la puntuación de dicho tablero. El siguiente pseudocódigo hace más ilustrativa dicha idea:

```
for(int i = 0; i < 7; i++) {
      for(int j = 0; j < 7; j++){
         if(estado.See_Casilla(i,j) == jugador || estado.See_Casilla(i,j) == bomba){
            seguidas++;
            if(seguidas == 2)
               puntuacion = puntuacion + 2 * seguidas;
            else if(seguidas == 3)
               puntuacion = puntuacion + 4 * seguidas;
            else
               puntuacion += seguidas;
         }
         else if(estado.See_Casilla(i,j) == enemigo || estado.See_Casilla(i,j) == bomba_enemigo){
            if(seguidas == 2)
               puntuacion = puntuacion – 2 * seguidas;
            else if(seguidas == 3)
               puntuacion = puntuacion – 4 * seguidas;
            else
               puntuacion -= seguidas;
```

```
            seguidas = 0;
          }
          else {
            seguidas = 0;
          }
        }
        seguidas = 0;
    }

    return puntuacion;
```

(Este fragmento ilustra el recorrido de la matriz de forma horizontal, es decir, recorriendo las filas. La manera de proceder es la misma para el recorrido por columnas y por diagonales.)

La idea es puntuar de mejor manera aquellos tableros en los que se produzca una alineación de dos o tres fichas contiguas del mismo jugador. Si el número de fichas seguidas es 2, la puntuación aumenta en 2*numero_seguidas. Si el número de fichas seguidas es 3, la puntuación aumenta en 4*numero_seguidas. En caso de que sólo sea una ficha, la puntuación aumenta en uno. En una primera aproximación mi función de puntuación terminaba aquí, pero tras diversas pruebas no funcionaba como esperaba. Para mejorar el funcionamiento se añadió la parte en la que se resta puntuación si la ficha es del enemigo (parte en negrita del código anterior). Esto es para que, por ejemplo, en el caso de que haya tres fichas seguidas en horizontal y a continuación una ficha del enemigo, la puntuación no varíe, ya que no nos interesaría colocar sobre la ficha del enemigo.

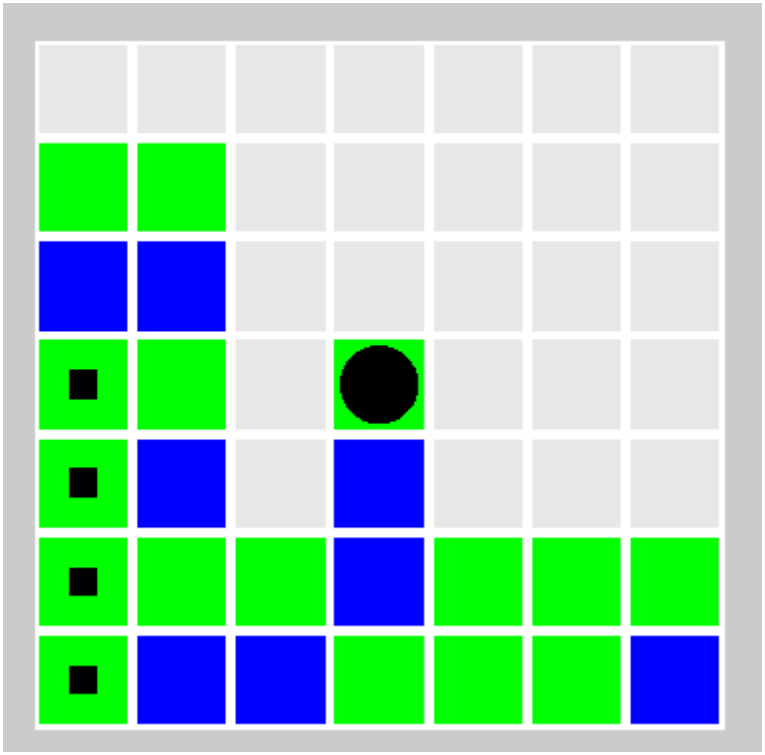
Una vez ha quedado clara el cometido de dichas funciones “auxiliares”, veamos la implementación de la función heurística:

```
double Heuristica(int jugador, const Environment &estado){
    int enemigo;
    (jugador == 1) ? enemigo = 2 : enemigo = 1;

    int mi_h = Horizontal(jugador, estado);
    int op_h = Horizontal(enemigo, estado);
    int mi_v = Vertical(jugador, estado);
    int op_v = Vertical(enemigo, estado);
    int mi_d = Diagonal(jugador, estado);
    int op_d = Diagonal(enemigo, estado);

    return ((op_h + op_v + op_d) - (mi_h + mi_v + mi_d));
}
```

Dado que el objetivo del juego es hacer que el enemigo alinee cuatro fichas y nosotros no, lo que hago es puntuar mejor un tablero si hay mas opciones de que el jugador enemigo realice una combinación de fichas a que la haga yo. Por lo tanto, la suma de las puntuaciones en vertical horizontal y diagonal para el enemigo menos la suma de las puntuaciones en vertical horizontal y diagonal para mí es la puntuación del tablero.

5. Ejemplos contra el ninja.

Aquí podemos ver los resultados obtenidos contra el ninja. Primero, jugando mi heurística como jugador azul:



Y jugando como jugador verde: