PRÁCTICA E2

INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

DAVID MUÑOZ SÁNCHEZ
07256819C Grupo B1

Análisis del problema

En esta práctica, se presenta una variación del juego típico del parchís en el que solo juegan dos jugadores, cada uno con dos colores distintos.
Esta práctica nos presenta dos retos:

1. Implementar el algoritmo MINIMAX con una profundidad máxima de cuatro niveles, o implementar la PODA ALFA-BETA con una profundidad máxima de seis niveles.
2. Diseñar una heurística que asigne valoraciones a los tableros de juego que vayamos generando con el algoritmo mencionado en el punto número 1. Esta heurística lo debe hacer lo mejor posible enfrentándose a varios ninjas con heurísticas propias, como primer jugador y como segundo jugador.

Descripción de la solución planteada

```
double AIPlayer::Poda_AlfaBeta(const Parchis &st_actual, int jugador, int
profundidad, int profundidad_max, color &c_piece, int &id_piece, int &dice,
double alpha, double beta, double (*heuristic)(const Parchis &, int)) const
{
   double valor_heuristica;
   //Si estamos en la profundidad máxima o si el juego ha terminado
   //quiere decir que estamos en un nodo hoja y lo tenemos que evaluar con
la
   //heurística correspondiente
   if (profundidad == profundidad_max or st_actual.gameOver())
       return heuristic(st_actual, jugador);

   //Variables para generar los hijos de cada nodo
   int id_piece_auxiliar = -1;
   int dice_auxiliar = -1;
   color c_piece_auxiliar = none;

   //Se genera siguiente hijo
   Parchis nextMove = st_actual.generateNextMoveDescending(c_piece_auxiliar,
id_piece_auxiliar, dice_auxiliar);

   while (!(nextMove == st_actual))
   {
       //Comprobacion para los casos en que aparezcan nodos max o min
seguidos
       if (this->jugador != st_actual.getCurrentPlayerId())
       {
           //Llamamos recursivamente a la poda para evaluar el hijo que
hemos generado
```

```
            valor_heuristica = Poda_AlfaBeta(nextMove, jugador, profundidad +
1, profundidad_max, c_piece_auxiliar, id_piece_auxiliar, dice_auxiliar,
alpha, beta, heuristic);

            beta = min(valor_heuristica, beta);

            if (beta <= alpha)
            {
                break;
            }
        }
        else
        {
            valor_heuristica = Poda_AlfaBeta(nextMove, jugador, profundidad +
1, profundidad_max, c_piece_auxiliar, id_piece_auxiliar, dice_auxiliar,
alpha, beta, heuristic);

            double aux = alpha;

            alpha = max(valor_heuristica, alpha);

            if (profundidad == 0 && aux != alpha)
            {

                dice = dice_auxiliar;
                c_piece = c_piece_auxiliar;
                id_piece = id_piece_auxiliar;
            }
            if (alpha >= beta)
            {
                break;
            }
        }
        nextMove = st_actual.generateNextMoveDescending(c_piece_auxiliar,
id_piece_auxiliar, dice_auxiliar);
    }

    if (jugador == st_actual.getCurrentPlayerId())
        return alpha;
    else
        return beta;
}
```

Tras declarar variables y hacer diversas comprobaciones comentadas en el código, entramos en lo que sería el análisis del árbol de decisión.

Cuando hacemos la llamada recursiva de la poda, evaluamos el hijo que hemos generado y este valor devuelto hay que tratarlo de forma diferente dependiendo de si es un nodo MAX o MIN:

- Si es MAX, comparamos alpha con el alpha actual que tenemos. Si es mayor, actualizamos el valor de alpha. Si la profundidad es 0, asignamos el movimiento.
  Por último, si beta es menor o igual que alpha dejamos de evaluar los siguientes hijos a generar, porque podemos realizar una poda.
- Si es MIN, comparamos beta con el beta actual. Si es menor, actualizamos el valor de beta. Si la profundidad es 0, asignamos el movimiento y si beta es menor o igual que alpha, podamos, igual que en el punto anterior.

En cuanto a la heurística implementada, he implementado una sola heurística que he asociado al id 0 y que tiene en cuenta diversos factores para asignar una valoración a un tablero. Para ello, hago uso de algunos de los métodos propuestos en el guión. Mi heurística se llama heurística y su implementación es la siguiente:

```
double AIPlayer::heuristica(const Parchis &st_actual, int jugador)
{
   double asignado = 0;
   int oponente = (jugador + 1) % 2;

   int ganador = st_actual.getWinner();
   if (ganador == jugador)
   {
       return gana;
   }
   else if (ganador == oponente)
   {
       return pierde;
   }
   else
   {

       vector<color> my_colors = st_actual.getPlayerColors(jugador);
       vector<color> op_colors = st_actual.getPlayerColors(oponente);

       //Por cada pieza que tengamos segura sumamos 1 a la valoración del tablero
       for (int i = 0; i < my_colors.size(); i++)
       {
           color c = my_colors[i];
           for (int j = 0; j < num_pieces; j++)
           {
               if (st_actual.isSafePiece(c, j))
               {
                   asignado += 1;
               }
           }
```

```
        }

        //Por cada pieza asegurada del oponente le quitamos 1, ya que el tablero no
es tan beneficioso
        //para nosotros.
        for (int i = 0; i < op_colors.size(); i++)
        {
            color c = op_colors[i];
            for (int j = 0; j < num_pieces; j++)
            {
                if (st_actual.isSafePiece(c, j))
                    asignado -= 1;
            }
        }

        // valor_asignado = valor_asignado + ValoracionTest(st_actual,jugador);

        //Le quito la media de distancias de mis fichas a la meta, mientras mas lejos
estén,
        //más se restara y el tablero será peor.
        asignado -= calcular_media(st_actual, jugador, my_colors[0]);
        asignado -= calcular_media(st_actual, jugador, my_colors[1]);
        //Mientras más piezas tenga en casa el oponente mejor es el tablero
        asignado += 4 * st_actual.piecesAtHome(op_colors[0]);
        asignado += 4 * st_actual.piecesAtHome(op_colors[1]);
        //Mientras más piezas tenga en la meta el oponente peor es el tablero
        asignado -= 3 * st_actual.piecesAtGoal(op_colors[0]);
        asignado -= 3 * st_actual.piecesAtGoal(op_colors[1]);
        //Si tengo piezas en casa quito el núero de piezas por 3, el tablero es peor
        asignado -= 4 * st_actual.piecesAtHome(my_colors[0]);
        asignado -= 4 * st_actual.piecesAtHome(my_colors[1]);
        //Si las tengo justo en la meta hago lo contrario, ya que mi tablero es mejor
        asignado += 3 * st_actual.piecesAtGoal(my_colors[0]);
        asignado += 3 * st_actual.piecesAtGoal(my_colors[1]);
        //Mientras más lejos estén de la meta las fichas del oponente mejor sera el
tablero
        asignado += calcular_media(st_actual, jugador, op_colors[0]);
        asignado += calcular_media(st_actual, jugador, op_colors[1]);


        return asignado;
    }
}
```

Lo primero para asignar una valoración en la variable asignado es sumar 1 unidad por cada ficha que tengamos segura, para lo que se usa el método **isSafePiece.** También hay que tener en cuenta que las fichas que tenga el oponente aseguradas hacen peor el tablero de juego desde el punto de vista del jugador, por tanto, por cada pieza asegurada del oponente, quitamos 1 unidad.

Ahora pasamos a tener en cuenta más consideraciones, como por ejemplo, las piezas que tengo en casa y las piezas que tiene el oponente en casa.
Que el oponente tenga piezas en casa nos viene bien, por tanto, se sumará a asignado el número de piezas en casa multiplicado por 4. Sin embargo, se hará lo mismo pero restando con las piezas que tengamos nosotros en casa, ya que es más difícil ganar con piezas en casa. Se hace uso de la función **piecesAtHome.**
Otro dato a tener en cuenta son las piezas en la meta. Se añadirá a la valoración el número de piezas en la meta del jugador multiplicado por 3 y se quitará el número de piezas en la meta del oponente multiplicado por 3. Se hace uso de la función **piecesAtGoal.**

Por último, cabe mencionar el uso de la función **calcular_media**, que devuelve la media de las distancias de las fichas a la meta.
Le quitaremos a la valoración la media de nuestras fichas, es decir, si están muy lejos, el tablero pasará a ser peor. Sin embargo, añadiremos a la valoración la media de las fichas del oponente, ya que mientras más lejos estén, mejor será el tablero desde nuestro punto de vista. Dentro de **calcular_media**, se hace uso del método **distanceToGoal** para averiguar la distancia de cada ficha a la meta según el color.

Lo más trabajoso y a la vez la mayor ventaja de esta heurística, es que modificando las valoraciones según el sector al que le queramos dar más importancia, podemos obtener un comportamiento u otro. A base de pruebas de ensayo y error, he descubierto que esas son las mejores valoraciones, donde por ejemplo, se da más importancia a no tener fichas en casa que a tener fichas en la meta.