MEMORIA DE LA PRÁCTICA 3
El Parchís (con un twist)
INTELIGENCIA ARTIFICIAL

José Alberto Hoces Castro 3ºDGIIM
Créditos por la portada y memes a Lorena Cáceres Arias

1. EXPLICACIÓN DE MI HEURÍSTICA

En esta única sección de la memoria voy a explicar poco a poco cómo trabaja la heurística que he programado como solución al problema que se nos ha propuesto en esta práctica. Mi heurística hace uso de la programación modular, es decir, para no repetir código he diseñado métodos que se apoyan en otros, por lo que iré de arriba a abajo. Describiré el código empezando por la función más arriba en la jerarquía e iré explicando las otras funciones auxiliares que usan, como si de matrioskas se tratase.

1.1. Método Heurística

```
double AIPlayer::Heuristica(const Parchis &estado, int jugador){

    int ganador = estado.getWinner();
    int oponente = (jugador+1) % 2;

    // Si hay un ganador, devuelvo más/menos infinito, según si he ganado yo o el oponente.

    if (ganador == jugador)
    {
        return gana;
    }
    else if (ganador == oponente)
    {
        return pierde;
    }
    else {
        double puntuacion_jugador = PuntuacionJugador(estado, jugador);
        double puntuacion_adversario = PuntuacionJugador(estado, oponente);
        return puntuacion_jugador - puntuacion_adversario;
    }
}
```

La función principal y que desencadena la llamada del resto que voy a explicar es *Heurística*. A esta se le pasa el estado actual del Parchís y el identificador del jugador para que determine cómo de favorable es el tablero para el jugador cuyo id se pasa como parámetro. Este es el método que se usa para evaluar los nodos en la poda alfa-beta.

Antes que nada, se usa el método *getWinner* para ver si el tablero *estado* tiene a algún jugador que tenga tres fichas de un mismo color en la meta. También se calcula el id del *oponente*. En caso de que haya ganador y este sea el jugador cuyo id es el parámetro, se devuelve *gana* que es una variable global con un valor máximo. Si el ganador es el oponente, se devuelve otra variable global *pierde* que es un valor mínimo. En caso de que no haya ganador, se calcula la puntuación del jugador y su oponente y se hace la diferencia. Si la diferencia es positiva, el tablero es favorable. Si es negativa, es desfavorable. Si es 0, el tablero presenta una situación neutra para ambos jugadores.

Para el cálculo de las puntuaciones de los jugadores cuando no hay ganador se hace uso del método *PuntuacionJugador*, que es la siguiente matrioska a analizar.

1.2. Método PuntuacionJugador

```
double AIPlayer::PuntuacionJugador(const Parchis &estado, int jugador){

    double puntuacion = 0;
    vector<color> colores;
    vector<double> puntuacion_color(2);

    colores = estado.getPlayerColors(jugador);
    puntuacion_color[0] = PuntuacionColor(estado, colores[0]);
    puntuacion_color[1] = PuntuacionColor(estado, colores[1]);

    // ÚLTIMO CRITERIO - Dados especiales

     vector<int> sdices_jugador = estado.getAvailableSpecialDices(jugador);

     for(int i = 0; i < sdices_jugador.size(); i++){

         switch(sdices_jugador[i]){

            case 101:
                puntuacion += 35;
                break;
            case 102:
                puntuacion += 30;
            case 103:
                puntuacion += 40;
                break;
            case 104:
                puntuacion += 25;
                break;
            case 105:
                puntuacion += 35;
                break;
            case 106:
                puntuacion += 8;
                break;
            case 107:
                puntuacion += 35;
                break;
            case 108:
                puntuacion += 30;
                break;
            case 109:
                puntuacion += 20;
                break;
            case 110:
                puntuacion += 5;
                break;
         }
     }

    if(puntuacion_color[0] > puntuacion_color[1]){
        puntuacion += 1.8*puntuacion_color[0] + 0.5*puntuacion_color[1];
    } else {
        puntuacion += 0.5*puntuacion_color[0] + 1.8*puntuacion_color[1];
    }

    return puntuacion;
}
```

En este método hay dos partes principales a analizar. Tras varios pensamientos y consejos de nuestros estupendos profesores, muchos decidimos que al calcular las puntuaciones de los jugadores, debíamos calcular por separado puntuaciones por colores. Esto nos sirve para priorizar a un color a la hora de jugar, ya que normalmente habrá un color más aventajado que otro. Para calcular la puntuación de los colores tuve que programar la tercera y última matrioska, *PuntuacionColor*. La analizaré más adelante, pero básicamente es la función que hace el mayor trabajo computacional.

En mi caso, al sumar las puntuaciones de los colores a la puntuación del jugador que se va a devolver, hago una comparación de las puntuaciones obtenidas y dependiendo de cuál sea mayor, pondero en dicha suma por 1.8 la mejor puntuación y por 0.5 la peor (0.5 es para no dar por muerto al color menos aventajado, ya que nos puede servir de ayuda en ocasiones concretas). El 1.8 es simplemente un valor al que he llegado a base de hacer intentos con el bot de la asignatura, no le puedo dar una justificación teórica pero sí empírica, pues vence a los 6 ninjas y además obtiene buenos resultados en los torneos nocturnos.

La segunda parte principal del método *PuntuacionJugador* es la valoración de los dados especiales. Al igual que las ponderaciones de las puntuaciones de los colores (0.5 y 1.8), los valores que les he dado a los dados especiales ha sido a base de muchas pruebas. Esta parte no va incluida en el código de *PuntuacionColor* ya que los dados especiales no se cuentan por color sino por jugador (los dos dados especiales que puede poseer un jugador como máximo pueden ser usados con el color que más le convenga de los dos que le corresponden). A la bala y el champiñón les he dado 40 y 8 respectivamente ya que es lo que me hacen avanzar como mínimo en el tablero. Con los objetos que pueden causar daño a fichas de un mismo jugador si son usados mal les he dado puntuaciones algo más bajas por ser conservador, salvo la estrella, el caparazón azul y el megachampiñón que son bastante útiles para arrasar con los rivales. De todas formas, para regular que esto no pase (di no al suicidio), en *PuntuacionColor* se regula el no atacar a fichas de otro color que sean del mismo jugador salvo en pequeños casos en los que viene bien (comerse una ficha del color que va peor nos sirve para avanzar 20 casillas por ejemplo). Finalmente en puntuación se suma la suma ponderada de las puntuaciones de los colores junto con la puntuación de los dados especiales que se poseen. Pasamos a analizar la última matrioska.

1.3. Método PuntuacionColor

En este método he considerado 5 criterios:

- Distancia de las fichas a la meta
- Fichas en la meta
- Fichas en casa
- Comer fichas
- Fichas destruidas en el turno anterior

Como el método es largo, adjunto poco a poco capturas del código donde se programa cada criterio. Empezamos con el primer criterio:

1.- Distancia de las fichas a la meta

```
// PRIMER CRITERIO - La distancia a la meta

double distancia = 0;
for(int i = 0; i < 3; i++){

        distancia += 100 - estado.distanceToGoal(c,i);
}
```

Para las fichas del color que se quiere puntuar, se hace la suma de 100 menos la distancia de cada ficha a la meta (al fin y al cabo consideramos que poca distancia es algo positivo, de ahí que reste a 100 ya que cuanto más pequeña sea, más sumamos de esos 100 posibles).

2.- Fichas en la meta

```
// SEGUNDO CRITERIO - Fichas en la meta

double en_meta = estado.piecesAtGoal(c)*50;
```

Aquí lo único interesante es que con el método piecesAtGoal me quedo con el número de fichas en casa y este lo multiplico por 50. Es de las ponderaciones más altas que tengo en el código ya que después de todo, como diría la ilustre Dakota Tarraga, lo importante no es participar, es ganar. Las fichas

en la meta es lo que necesitamos para ganar así que es una situación deseable.

3.- Fichas en casa

```
// TERCER CRITERIO - Fichas en casa

double en_casa = estado.piecesAtHome(c)*50;
```

Este criterio es de un razonamiento análogo al de antes. Las piezas en casa es algo que queremos evitar, así que para que la IA reaccione y saque las fichas de casa, pondero por 50 cada ficha en casa. Es una situación muy desfavorable y que nos aleja del Chanelazo. Al final del método lo que se hace para que cuente como algo negativo es restar la variable *en_casa*.

4.- Comer fichas

```
// CUARTO CRITERIO - Comerse fichas

double comer = 0;

tuple<color, int, int> last_action = estado.getLastAction();

switch(get<0>(last_action)){
    case red:
    case yellow:

        if(estado.eatenPiece().first == blue or estado.eatenPiece().first == green){
                comer += 15;
        } else {
            if(estado.eatenPiece().first != none){
                comer -= 15;
            }
        }

        break;
    case blue:
    case green:

        if(estado.eatenPiece().first == red or estado.eatenPiece().first == yellow){
                comer += 15;
        } else {
            if(estado.eatenPiece().first != none){
                comer -= 15;
            }
        }

        break;

    case none:
    break;
}
```

La idea principal de este código es favorecer los movimientos en los que se coman a fichas del otro jugador y no a las nuestras. La ponderación de -+15 se debe a que cuando comemos una ficha, avanzamos 20 casillas adicionales. De esta forma, a veces no se evalúa como negativo comerse una ficha del otro color de un mismo jugador si ello supone dar un salto importante hacia la meta (sobre todo cuando el color que se come es al que le corresponde la ponderación 0.5). De todas formas, con los if-else controlo un poco esto para que no abuse de dicha técnica.

El código puede resultar confuso, pero tras varios debates contigo razonamos de la siguiente forma: si en el turno anterior una ficha era del color actual y se comió a una ficha de los otros dos colores del rival (se obtiene si hay alguna ficha comida con el método *eatenPiece*), sumamos 15. Si la comida es de uno de los dos colores de mi jugador, se resta 15 a la puntuación.

Por último, pasamos a analizar el quinto y último criterio que implementé ayer y que ha supuesto una mejora significativa en mis resultados en el torneo:

5.- Fichas destruidas

```
// QUINTO CRITERIO - Fichas destruidas

vector<pair<color, int>> fichas_destruidas = estado.piecesDestroyedLastMove();
double destruidas = 0;

for(int i = 0; i < fichas_destruidas.size(); i++){

    switch(fichas_destruidas[i].first){
        case red:
        case yellow:

            if(c == red or c == yellow){
                destruidas -= 15;
            } else {
                destruidas += 15;
            }

            break;
        case blue:
        case green:

            if(c == blue or c == green){
                destruidas -= 15;
            } else {
                destruidas += 15;
            }

            break;
    }
}
```

Con el método piecesDestroyedLastMove obtengo las fichas que fueron destruidas en el turno anterior. Se recorre el vector de fichas destruidas y se comprueba una a una si su color es de las fichas del color actual o de su color capybara (capybara, el animal que es amigo de todos, para que se entienda la referencia). Pondero con +15 si las fichas destruidas son de los colores adversarios y con -15 si son de los colores que nos interesa favorecer. De esta forma, logro quitarme ese miedo a que las fichas de dos colores que van juntos usen objetos que atacan perjudicándose entre ellas y pude subir algo más las ponderaciones de esos dados especiales.

Finalmente, se calcula la puntuación con las 5 variables calculadas para sus respectivos criterios:

```
puntuacion = distancia + en_meta - en_casa + comer + destruidas;
```

Como ya habíamos dicho, el único criterio negativo es el número de fichas que están casa, de ahí que se reste en la suma.

Conclusiones

Estos últimos días los dos criterios que he añadido por razones competitivas son los dos últimos explicados. Comer y destruir fichas era algo clave para que mi IA tuviese un comportamiento más agresivo y pudiese ganar más partidas. Realmente, para vencer las 6 partidas contra los ninjas, con los tres primeros criterios me bastó.

Con esto, me despido de las prácticas de IA y espero que te haya sido tan ameno leer esta memoria como a mí me ha sido ameno hacer esta práctica 🙂. Gracias por tu empeño.