Memoria Practica 3 IA:

David Martínez Díaz GII-ADE

Para iniciar con la práctica, lo primero que hice fue mirarme el guion para poder tener una ligera idea de lo que iba consistir el proyecto del parchís, y una vez leído me dispuse a comenzar con el tutorial.

Para ello, en primer lugar, mientras que me leía el documento, también tenia al lado los códigos del jugador, y poder ir hilando algunos conceptos y tener las cosas mas claras.

Entonces me dispuse a realizar la primera toma de contacto con la práctica, el método thinkAleatorio, que simplemente utilizaba los dados de la forma más aleatoria.

```
void AIPlayer::thinkAleatorio(color & c_piece, int & id_piece, int & dice) const{

    // El color de ficha que se va a mover
    c_piece = actual->getCurrentColor();

    // Vector que almacenará los dados que se pueden usar para el movimiento
    vector<int> current_dices;
    // Vector que almacenará los ids de las fichas que se pueden mover para el dado elegido.
    vector<int> current_pieces;

    // Se obtiene el vector de dados que se pueden usar para el movimiento
    current_dices = actual->getAvailableDices(c_piece);
    // Elijo un dado de forma aleatoria.
    dice = current_dices[rand() % current_dices.size()];

    // Se obtiene el vector de fichas que se pueden mover para el dado elegido
    current_pieces = actual->getAvailablePieces(c_piece, dice);

    // Si tengo fichas para el dado elegido muevo una al azar.
    if(current_pieces.size() > 0){
        id_piece = current_pieces[rand() % current_pieces.size()];
    }
    else{
        // Si no tengo fichas para el dado elegido, pasa turno (la macro SKIP_TURN me permite no mover).
        id_piece = SKIP_TURN;
    }
}
```

Lo siguiente que implemente fue el thinkAleatorioMasinteligente, que básicamente consistía en lo mismo sin embargo tenia en cuenta solo las piezas que se podían mover que dichos dados:

```
void AIPlayer::thinkAleatorioMasInteligente(color & c_piece, int & id_piece, int & dice) const{

   // El color de ficha que se va a mover
    c_piece = actual->getCurrentColor();

    // Vector que almacenará los dados que se pueden usar para el movimiento
    vector<int> current_dices;
    // Vector que almacenará los ids de las fichas que se pueden mover para el dado elegido.
    vector<int> current_pieces;

    // Se obtiene el vector de dados que se pueden usar para el movimiento
    current_dices = actual->getAvailableDices(c_piece);


    vector<int> current_dices_que_pueden_mover_ficha;

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


        current_pieces=actual->getAvailablePieces(c_piece,current_dices[i]);

        if(current_pieces.size() > 0 ){
            current_dices_que_pueden_mover_ficha.push_back(current_dices[i]);
        }

    }

    if(current_dices_que_pueden_mover_ficha.size() == 0){
        dice = current_dices[rand() % current_dices.size()];

        id_piece=SKIP_TURN;
    }
    else{
        dice=current_dices_que_pueden_mover_ficha[rand() % current_dices_que_pueden_mover_ficha.size()];

        current_pieces=actual->getAvailablePieces(c_piece,dice);

        id_piece=current_pieces[rand() % current_pieces.size()];
    }
}
```

Luego buscando unos movimientos mas inteligente y coherentes nos creamos la función de FichaMasAdelantada, que simplemente trata de como de una vez has escogido cual va ser el dado ha utilizar, seleccionas la ficha que este mas adelantada para que pueda llegar cuanto antes a la meta:

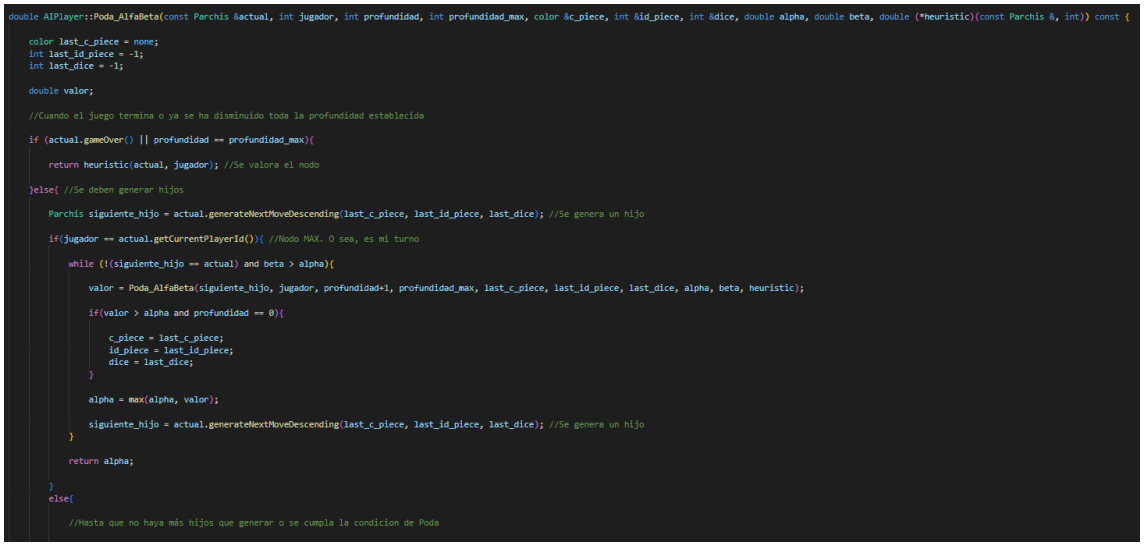
```
void AIPlayer::thinkFichaMasAdelantada(color & c_piece, int & id_piece, int & dice) const{

    thinkAleatorioMasInteligente(c_piece, id_piece, dice);

    vector<int> current_pieces = actual->getAvailablePieces(c_piece, dice);

    int id_ficha_mas_adelantada = -1;
    int min_distancia_meta = 9999;

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

        int distancia_meta = actual->distanceToGoal(c_piece, current_pieces[i]);
        if(distancia_meta < min_distancia_meta){
            min_distancia_meta = distancia_meta;
            id_ficha_mas_adelantada = current_pieces[i];
        }
    }

    if(id_ficha_mas_adelantada == -1){
        id_piece = SKIP_TURN;
    }
    else {
        id_piece = id_ficha_mas_adelantada;
    }

}
```

Por ultimo, para finalizar con el seguimiento del guion nos creamos la función de MejorOpcion, donde aquí ya se nos empieza a plantear el uso de arboles y diferentes estados de nodos para poder plantear cual es la mejor opción, utilizando en este caso las funciones auxiliares ya creadas como isEatingMove o isGoalMove entre otras:

```
void AIPlayer::thinkMejorOpcion(color & c_piece, int & id_piece, int & dice) const{

    color last_c_piece = none;
    int last_id_piece = -1;
    int last_dice = -1;

    Parchis siguiente_hijo = actual->generateNextMove(last_c_piece, last_id_piece, last_dice);
    bool me_quedo_con_esta_accion = false;

    while(!(siguiente_hijo == *actual) && !me_quedo_con_esta_accion){

        if(siguiente_hijo.isEatingMove() or siguiente_hijo.isGoalMove() or (siguiente_hijo.gameOver() and siguiente_hijo.getWinner() == this->jugador)){
            me_quedo_con_esta_accion = true;
        }
        else{
            siguiente_hijo = actual->generateNextMove(last_c_piece, last_id_piece, last_dice);
        }
    }

    if(me_quedo_con_esta_accion){

        c_piece = last_c_piece;
        id_piece = last_id_piece;
        dice = last_dice;
    }
    else{
        thinkFichaMasAdelantada(c_piece, id_piece, dice);
    }


}
```

Una vez realizado el tutorial, ya podemos continuar con la practica nosotros solos una vez tenemos claros mas o menos las funciones básicas para poder plantear alguna estrategia ganadora para enfretarnos a los ninjas, donde se nos plantean dos opciones, la poda alfa-beta o la función minimax.

En mi caso, como en el archivo con extensión header se nos facilito la cabecera de la poda Alfa-beta, decidí decantarme por esta opción, y tras informarme en tutoriales de como realizar dicha poda y algunos ejemplos de esta, consegui crear dicha implementación adaptada a nuestro caso al parchis:



```
double AIPlayer::Poda_AlfaBeta(const Parchis &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 {

    color last_c_piece = none;
    int last_id_piece = -1;
    int last_dice = -1;

    double valor;

    //Cuando el juego termina o ya se ha disminuido toda la profundidad establecida

    if (actual.gameOver() || profundidad == profundidad_max){

        return heuristic(actual, jugador); //Se valora el nodo

    }else{ //Se deben generar hijos

        Parchis siguiente_hijo = actual.generateNextMoveDescending(last_c_piece, last_id_piece, last_dice); //Se genera un hijo

        if(jugador == actual.getCurrentPlayerId()){ //Nodo MAX. O sea, es mi turno

            while (!(siguiente_hijo == actual) and beta > alpha){

                valor = Poda_AlfaBeta(siguiente_hijo, jugador, profundidad+1, profundidad_max, last_c_piece, last_id_piece, last_dice, alpha, beta, heuristic);

                if(valor > alpha and profundidad == 0){

                    c_piece = last_c_piece;
                    id_piece = last_id_piece;
                    dice = last_dice;
                }

                alpha = max(alpha, valor);

                siguiente_hijo = actual.generateNextMoveDescending(last_c_piece, last_id_piece, last_dice); //Se genera un hijo
            }

            return alpha;

        }
        else{

            //Hasta que no haya más hijos que generar o se cumpla la condicion de Poda
```

```
            while (!(siguiente_hijo == actual) and beta > alpha){

                valor = Poda_AlfaBeta(siguiente_hijo, jugador, profundidad+1 , profundidad_max, last_c_piece, last_id_piece, last_dice, alpha, beta, heuristic);

                if(valor < beta && profundidad == 0){
                    c_piece = last_c_piece;
                    id_piece = last_id_piece;
                    dice = last_dice;
                }

                beta = min(valor, beta);

                siguiente_hijo = actual.generateNextMoveDescending(last_c_piece, last_id_piece, last_dice);//Se genera un hijo

            }

            return beta;

        }
    }
```

Al principio, no me funcionaba del todo bien, porque realiza unos máximos y mínimos innecesarios que me alteraba tanto el valor del alfa y como de la beta, pero posteriormente se me resolvió dicha duda y ya funcionaba correctamente, consiguiendo probar dicha poda con la función que nos facilitaban que era ValoracionTest frente al Ninja 0.

Una vez, comprobé el correcto funcionamiento de la poda, decidí ponerme con la heurística, donde al principio me decanté por intentar mejor la función que nos facilitaban.

Sin embargo, fue todo un fracaso ya que no era capaz de ganar a ningún ninja, por lo que decidí cambiar de estrategia y pensé como jugaría yo en las situaciones que se presentaban, donde al final pensé en utilizar uno de mis colores como sacrificios y para crear muros mientras que otro se dedicaba a avanzar e intentar llegar a la meta pasando por los sitios seguros.

Para ello, distinguí los colores de cada jugador y realicé una comprobación previa de quien iba mas adelantado de los colores tanto míos como de mi oponente, así podía dar diferentes valores a cada uno dependiendo del contexto en el que se situaran.

Dicha comprobación lo realizaba a través de la función DistanceToGoal() por lo que cuanto menos les quedase para llegar a la meta mas prioridad iba a tener dicho color.

Luego simplemente comprobaba al igual que ValoracionTest algunos contextos en los que se podía situar cada pieza, ya sea que se encontrase en la meta, en la cola final para llegar a la meta o en casa.

Además, decidí darle prioridad a las casillas que fueran seguras, para así no poder ser comidos por el oponente, y les daba color según si eran el color prioritario o el secundario.

Por último, añadí una cierta puntación por si la pieza que había sido comido era del oponente o en este caso mía para poder darle más puntación a dicha situación o restarle dependiendo de la ficha comida, donde finalmente resto la puntación de ambos jugadores y consigo un valor mas o menos coherente para poder ir realizando la poda correctamente, consiguiendo ganar a los 3 ninjas tanto de Jugador 1 como de Jugador 2.

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

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

    if (ganador == jugador)
    {
        return gana;
    }
    else if (ganador == oponente)
    {
        return pierde;
    }
    else{

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

        int puntuacion_jugador = 0;
        int puntuacion_oponente = 0;

        int my_puntacion_colores[2] = {0, 0};
        int op_puntacion_colores[2] = {0, 0};

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

            color c = my_colors[i];
            color c_oponente = op_colors[i];

            for (int j = 0; j < num_pieces; j++){

                my_puntacion_colores[i] += estado.distanceToGoal(c, j);
                op_puntacion_colores[i] += estado.distanceToGoal(c_oponente, j);
            }
        }

        //-----------------------------------------
```

```
color my_color_primario, my_color_secundario;
color op_color_primario, op_color_secundario;

if(my_puntacion_colores[0] > my_puntacion_colores[1]){
    my_color_primario = my_colors[1];
    my_color_secundario = my_colors[0];
}
else{
    my_color_primario = my_colors[0];
    my_color_secundario = my_colors[1];
}

//----------------------------------------

if(op_puntacion_colores[0] > op_puntacion_colores[1]){
    op_color_primario = op_colors[1];
    op_color_secundario = op_colors[0];
}
else{
    op_color_primario = op_colors[0];
    op_color_secundario = op_colors[1];
}

//----------------------------------------

    for (int j = 0; j < num_pieces; j++)
    {

        //-----------------------------------
        // Jugador
        //-----------------------------------

        // Color primario

        if(estado.getBoard().getPiece(my_color_primario, j).type == goal){
            puntuacion_jugador+=20;
        }
        else if(estado.getBoard().getPiece(my_color_primario, j).type == final_queue){
            puntuacion_jugador+=15;
        }
        else if (estado.getBoard().getPiece(my_color_primario, j).type != home){
            puntuacion_jugador+=5;
        }
```

```
        if (estado.isSafePiece(my_color_primario, j))
        {
            puntuacion_jugador += 10;
        }

        /*
        if(estado.boxState(estado.getBoard().getPiece(my_color_primario, j)).size() == 2){
            puntuacion_jugador += 10;
        }
        */
        puntuacion_jugador += 100 - estado.distanceToGoal(my_color_primario, j) / 4;

        // Color secundario

        if(estado.getBoard().getPiece(my_color_secundario, j).type == goal){
            puntuacion_jugador+=10;
        }
        else if(estado.getBoard().getPiece(my_color_secundario, j).type == final_queue){
            puntuacion_jugador+=5;
        }
        else if (estado.getBoard().getPiece(my_color_secundario, j).type != home){
            puntuacion_jugador+=2;
        }

        if (estado.isSafePiece(my_color_secundario, j))
        {
            puntuacion_jugador += 4;
        }
        /*
        if(estado.boxState(estado.getBoard().getPiece(my_color_secundario, j)).size() == 2){
            puntuacion_jugador += 10;
        }
        */
        puntuacion_jugador += 100 - estado.distanceToGoal(my_color_secundario, j) / 8;

        //---------------------------------------
        // Oponente
        //---------------------------------------

        // Color primario
```

```
        if(estado.getBoard().getPiece(op_color_primario, j).type == goal){
            puntuacion_oponente+=20;
        }
        else if(estado.getBoard().getPiece(op_color_primario, j).type == final_queue){
            puntuacion_oponente+=15;
        }
        else if (estado.getBoard().getPiece(op_color_primario, j).type != home){
            puntuacion_oponente+=5;
        }

        if (estado.isSafePiece(op_color_primario, j))
        {
            puntuacion_oponente += 10;
        }

        /*
        if(estado.boxState(estado.getBoard().getPiece(op_color_primario, j)).size() == 2){
            puntuacion_oponente += 10;
        }
        */
        puntuacion_oponente += 100 - estado.distanceToGoal(op_color_primario, j) / 4;

        // Color secundario

        if(estado.getBoard().getPiece(op_color_secundario, j).type == goal){
            puntuacion_oponente+=10;
        }
        else if(estado.getBoard().getPiece(op_color_secundario, j).type == final_queue){
            puntuacion_oponente+=5;
        }
        else if (estado.getBoard().getPiece(op_color_secundario, j).type != home){
            puntuacion_oponente+=2;
        }

        if (estado.isSafePiece(op_color_secundario, j))
        {
            puntuacion_oponente += 4;
        }

        /*
        if(estado.boxState(estado.getBoard().getPiece(op_color_secundario, j)).size() == 2){
            puntuacion_oponente += 10;
        }
        */
```

```
            puntuacion_oponente += 100 - estado.distanceToGoal(op_color_secundario, j) / 8;

        }
        if(estado.eatenPiece().first == op_color_primario){
            puntuacion_jugador+=15;
        }
        else if (estado.eatenPiece().first == op_color_secundario){
            puntuacion_jugador+=5;
        }
        else if (estado.eatenPiece().first == my_color_primario){
            puntuacion_oponente+=15;
        }
        else if (estado.eatenPiece().first == my_color_secundario){
            puntuacion_oponente+=5;
        }


    return puntuacion_jugador - puntuacion_oponente;
}
```