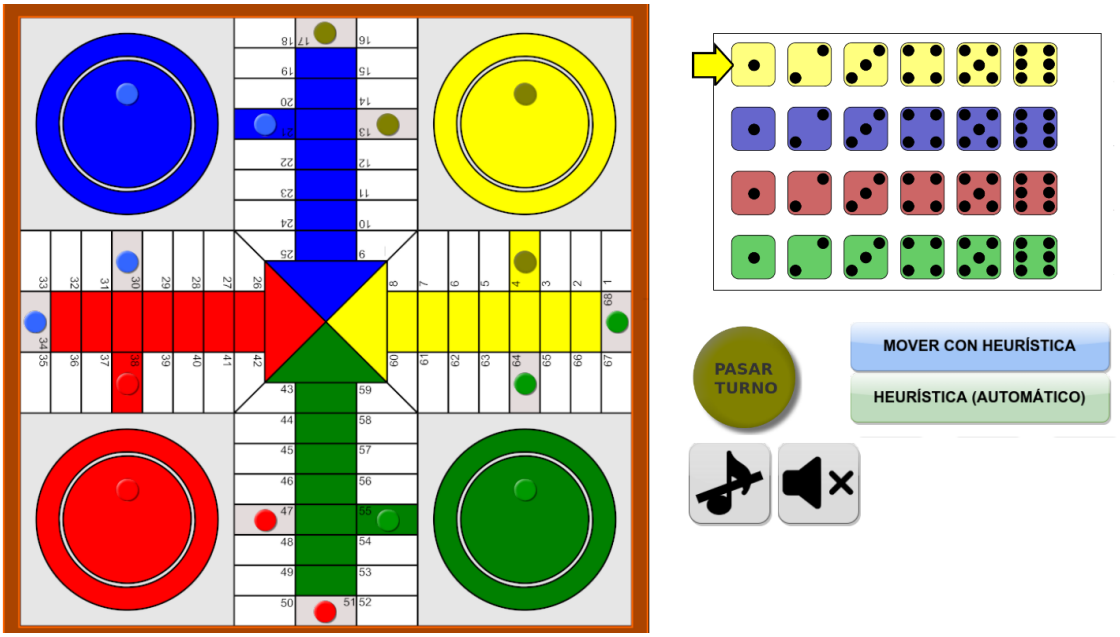
Documentación Práctica 3



1. Algoritmos de Búsqueda (MiniMax y Alfa Beta)

Para esta práctica, se pedía implementar (al menos) uno de los dos algoritmos para búsqueda en juegos del tema 4 de teoría. En mi caso, he implementado ambos algoritmos, tanto el MiniMax como el Poda Alfa Beta. En ambos la implementación era sencilla, puesto que se nos proporcionaba la función para generar sucesores. Bastaba con entender el algoritmo y adaptar las plantillas de teoría a nuestro problema.

Ambos funcionan correctamente, por lo que a la hora de escoger uno he optado por el de Poda Alfa Beta, ya que es más rápido (gracias a la poda) y permite recorrer más nodos. Además, para mejorar su velocidad de ejecución, he generado en ese los hijos de forma descendente para, en muchos casos (sobre todo los casos del nodo MAX) se pudieran podar los nodos de una forma más óptima.

```
// Algoritmo Poda Alfa Beta
double AIPlayer::PodaAlfaBeta(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 {
   if(actual.gameOver() || profundidad==profundidad_max) {
       return heuristic(actual, jugador);
   }
   // Guardan la accion a realizar para llegar al k-ésimo hijo
   color last_c_piece = none;  // El color de la última ficha que se movió.
   int last_id_piece = -1;     // El id de la última ficha que se movió.
   int last_dice = -1;         // El dado que se usó en el último movimiento.
   // Auxiliares, son variables estáticas (sólo se declaran una vez, más eficiente)
   double aux = 0;
   color aux_c_piece;
   int aux_id_piece;
   int aux_dice;
   Parchis siguiente_hijo = actual.generateNextMoveDescending(last_c_piece,
last_id_piece, last_dice);
```

```
    if (jugador == actual.getCurrentPlayerId()) {    // Nodo MAX
        while(!(siguiente_hijo == actual)) {
            aux = PodaAlfaBeta (siguiente_hijo, jugador, profundidad + 1,
                                profundidad_max, c_piece, id_piece,
                                dice, alpha, beta, heuristic);
            if (aux > alpha) {
                alpha = aux;
                aux_c_piece = last_c_piece;
                aux_id_piece = last_id_piece;
                aux_dice = last_dice;
            }
            if(alpha>=beta) {
                return beta;
            }
            siguiente_hijo = actual.generateNextMoveDescending(last_c_piece,
last_id_piece, last_dice);
        }
        if (profundidad==0) {
            c_piece = aux_c_piece;
            id_piece = aux_id_piece;
            dice = aux_dice;
        }
        return alpha;
    } else {    // Nodo MIN
        while(!(siguiente_hijo == actual)) {
            aux = PodaAlfaBeta (siguiente_hijo, jugador, profundidad + 1,
                                profundidad_max, c_piece, id_piece,
                                dice, alpha, beta, heuristic);
            if (aux < beta) {
                beta = aux;
                aux_c_piece = last_c_piece;
                aux_id_piece = last_id_piece;
                aux_dice = last_dice;
            }
            if(alpha>=beta) {
                return alpha;
            }
            siguiente_hijo = actual.generateNextMoveDescending(last_c_piece,
last_id_piece, last_dice);
        }
        if (profundidad==0) {
            c_piece = aux_c_piece;
            id_piece = aux_id_piece;
            dice = aux_dice;
        }
        return beta;
    }
}
```

2. Funciones Heurísticas

A la hora de diseñar funciones heurísticas para la práctica, he tomado como base de toda función, la estructura de ValoracionTest, y a partir de ahí he ido añadiendo nuevas estrategias y cambiando las ponderaciones de cada sección.

En la práctica he dejado 3 heurísticas implementadas, siendo la función myHeuristic1 la escogida puesto que proporciona mejores resultados que el resto. Paradójicamente, esta es también la más simple de todas y la que menos recursos computacionales requiere. En ella, al igual que en ValoracionTest se recorren los colores y las piezas de cada jugador y se van

estableciendo puntuaciones acerca de ciertas características que estas cumplan, y al final se devuelve la diferencia de puntuaciones de los jugadores como valor de dicho estado (en el caso de los estados terminales se devuelve más infinito si gana MAX, y menos infinito si gana MIN).

Las características y puntuaciones por estas se pueden ver en la siguiente tabla (son iguales tanto para MAX como para MIN).

Característica	Valoración
Está en una barrera (de su color)	+ 0.5 por cada ficha
Está en un seguro	+ 1 por cada ficha
Está en la meta	+ 5 por cada ficha
Está en el pasillo final	+ 2 por cada ficha
Está en casa	- 10 por cada ficha
Ha comido una ficha (de su oponente)	+ 10 por color
Ha alcanzado la meta con una ficha	+ 2 por color

La implementación de la heurística queda de la siguiente manera:

```
double AIPlayer::myHeuristic1(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 {
       // Colores que juega mi jugador y colores del oponente
       vector<color> my_colors = estado.getPlayerColors(jugador);
       vector<color> op_colors = estado.getPlayerColors(oponente);

       // Recorro todas las fichas de mi jugador
       int puntuacion_jugador = 0;
       // Recorro colores de mi jugador.
       for (int i = 0; i < my_colors.size(); i++) {
           color c = my_colors[i];
           // Recorro las fichas de ese color.
           for (int j = 0; j < num_pieces; j++) {
               if (estado.isWall(estado.getBoard().getPiece(c, j)) == c) {
                   // Debe ser una mitad, puesto que si está en una barrera,
                   // la puntuación se incrementará dos veces.
                   puntuacion_jugador += 0.5;
               } else if (estado.isSafePiece(c, j)) {
                   puntuacion_jugador ++;
               } else if (estado.getBoard().getPiece(c, j).type == goal) {
                   puntuacion_jugador += 5;
               } else if (estado.getBoard().getPiece(c, j).type == final_queue) {
                   puntuacion_jugador += 2;
               } else if (estado.getBoard().getPiece(c, j).type == home) {
                   puntuacion_jugador -= 10;
               }
           }
           // Nuestro color ha comido pieza
           if (estado.isEatingMove() && estado.getCurrentColor() == c) {
```

```
                if (estado.eatenPiece().first != my_colors[(i + 1) % 2]) {
                    puntuacion_jugador += 10;
                }
            } else if (estado.isGoalMove() && estado.getCurrentColor() == c) {
                puntuacion_jugador += 2;
            }
        }

        // Recorro todas las fichas del oponente
        int puntuacion_oponente = 0;
        for (int i = 0; i < my_colors.size(); i++) {
            color c = op_colors[i];
            // Recorro las fichas de ese color.
            for (int j = 0; j < num_pieces; j++) {
                if (estado.isWall(estado.getBoard().getPiece(c, j)) == c) {
                    puntuacion_oponente += 0.5;
                } else if (estado.isSafePiece(c, j)) {
                    puntuacion_oponente ++;
                } else if (estado.getBoard().getPiece(c, j).type == goal) {
                    puntuacion_oponente += 5;
                } else if (estado.getBoard().getPiece(c, j).type == final_queue) {
                    puntuacion_oponente += 2;
                } else if (estado.getBoard().getPiece(c, j).type == home) {
                    puntuacion_oponente -= 10;
                }
            }
            if (estado.isEatingMove() && estado.getCurrentColor() == c) {
                if (estado.eatenPiece().first != op_colors[(i + 1) % 2]) {
                    puntuacion_oponente += 10;
                }
            } else if (estado.isGoalMove() && estado.getCurrentColor() == c) {
                puntuacion_oponente += 2;
            }
        }
        // Devuelvo la puntuación de mi jugador menos la puntuación del oponente.
        return puntuacion_jugador - puntuacion_oponente;
    }
}
```

Las otras dos heurísticas (myHeuristic2, myHeuristic3) son algo más complejas y en teoría deberían funcionar mejor que la escogida (puesto que representan estrategias más complejas que deberían de ayudar a ganar). Aunque en la práctica producen peores resultados frente a los ninjas que la escogida. Al tener en cuenta más factores, (como la distancia hasta la meta, o los dados disponibles), he tenido que aumentar el valor de otras características y se ha convertido en una tarea difícil y tediosa el equilibrar las ponderaciones para estas funciones. No obstante, he decidido dejarlas en el código puesto que pueden resultar bastante interesantes si se consigue establecer bien las ponderaciones. Están explicadas con más detalle en su cabecera e implementación.

3. Resultados

Concluyendo con la memoria de esta práctica, los resultados obtenidos son los descritos a continuación, con el algoritmo Poda Alfa Beta (con profundidad máxima 6) y la heurística myHeuristic1.

	Ninja 1	Ninja 2	Ninja 3
IA J1	Victoria	Derrota	Victoria
IA J2	Victoria	Derrota	Derrota