# INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

# Práctica 3

## Búsqueda con Adversario (Juegos)

## El Parchís (BOOM!)

**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL**
**UNIVERSIDAD DE GRANADA**
**Curso 2023-2024**

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

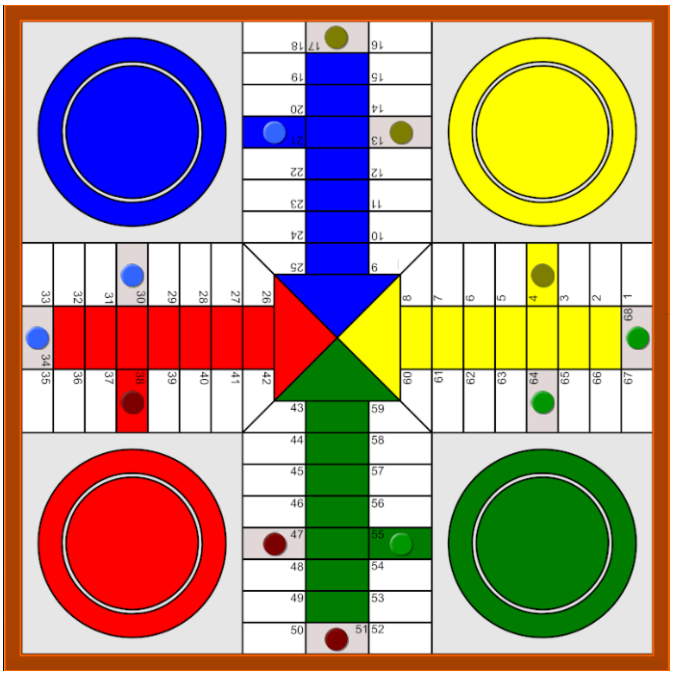
/ UGR / decsai

# 1. Introducción

## 1.1. Motivación

La tercera práctica de la asignatura ***Inteligencia Artificial*** consiste en el diseño e implementación de alguna de las técnicas de búsqueda con adversario en un entorno de juegos. Se trabajará con un simulador del juego **PARCHÍS,** que simula un Parchís determinista en el que cada jugador juega con dos colores alternos y los dados se van eligiendo entre un conjunto de dados disponibles.

El Parchís es un popular juego de mesa para 4 jugadores derivado del pachisi. En su versión original es un juego de 2 a 4 jugadores. Requiere un tablero específico formado por un circuito de 100 casillas y 4 "casas" de diferentes colores: amarillo, rojo, verde y azul. Cada jugador dispone de 4 fichas del mismo color que su "casa". El objetivo del juego es llevar todas las fichas desde su casa hasta la meta recorriendo todo el circuito, intentando "comerse" o capturar el resto de fichas en el camino. El primero en conseguirlo será el ganador.



Para esta práctica, contaremos con una versión especial del parchís. Además de ser determinista y poder elegir nosotros en cada turno el dado que queremos usar, dispondremos de una barra de energía y un dado especial para cada jugador. El funcionamiento del dado especial dependerá, en cada momento, del valor de la barra de energía, pudiendo tener superpoderes o hacer catástrofes.

Para la realización de esta práctica, el/la alumno/a deberá conocer en primer lugar las técnicas de búsqueda con adversario explicadas en teoría. En concreto, **el objetivo de esta práctica es la implementación del algoritmo MINIMAX o del algoritmo de PODA ALFA-BETA**, para dotar de comportamiento inteligente deliberativo a un jugador artificial para este juego, de manera que esté en condiciones de competir y ganar a sus adversarios.

A continuación, explicamos cuáles son los requisitos de la práctica, los objetivos concretos que se persiguen, el software necesario junto con su instalación, y una guía para poder programar el simulador.

## 2. Requisitos

Para poder realizar la práctica, es necesario que el/la alumno/a disponga de:

1. Conocimientos básicos del lenguaje C/C++: tipos de datos, sentencias condicionales, sentencias repetitivas, funciones y procedimientos, clases, métodos de clases, constructores de clase.
2. Instalar la librería SFML:
    a. En Ubuntu: sudo apt install libsfml-dev
    b. En MacOS: brew install sfml
3. El software para construir el simulador **PARCHÍS** está disponible en el GitHub de la asignatura.

## 3. Objetivo de la práctica

La práctica tiene como objetivo diseñar e implementar un agente deliberativo que pueda llevar a cabo un comportamiento inteligente dentro del juego **PARCHÍS** que se explica a continuación.

Para adaptar el popular juego Parchís a los requisitos de la asignatura, se sustituye el comportamiento aleatorio de tirar un dado por la elección del dado entre los dados disponibles. El conjunto de dados disponible será, a priori, 6 valores de un dado (1, 2, 4, 5, 6 y **especial**). Cada vez que se utilice uno de los dados ese valor se gastará, teniendo que elegir en el siguiente turno un dado diferente. Cuando se hayan gastado todos los valores del dado, se regenera por completo. Eventualmente, a los 6 valores del dado se añadirán valores especiales como 10 o 20 para ser utilizados en el momento.

Además, en lugar de elegir cada jugador un color pudiendo jugar entre 2-4 jugadores, en este caso siempre habrá 2 jugadores que jugarán con dos colores alternos. Cada vez que le toque a un jugador, con el dado que elija sacar podrá mover una ficha de cualquiera de sus dos colores. Aunque cada jugador controle a dos colores, estos colores podrán atacarse entre sí. Por ejemplo, aunque un jugador esté jugando con los colores amarillo y rojo, cuando se mueve una ficha amarilla a una casilla no segura

donde había previamente una roja, la ficha amarilla se come a la ficha roja aunque sean del mismo jugador.

El objetivo de **PARCHÍS** es conseguir meter TODAS las fichas de uno de nuestros colores en su casilla destino. **Gana** la partida **el primer jugador que consiga meter TODAS las fichas de CUALQUIERA de sus colores**, independientemente de dónde estén el resto de sus fichas.

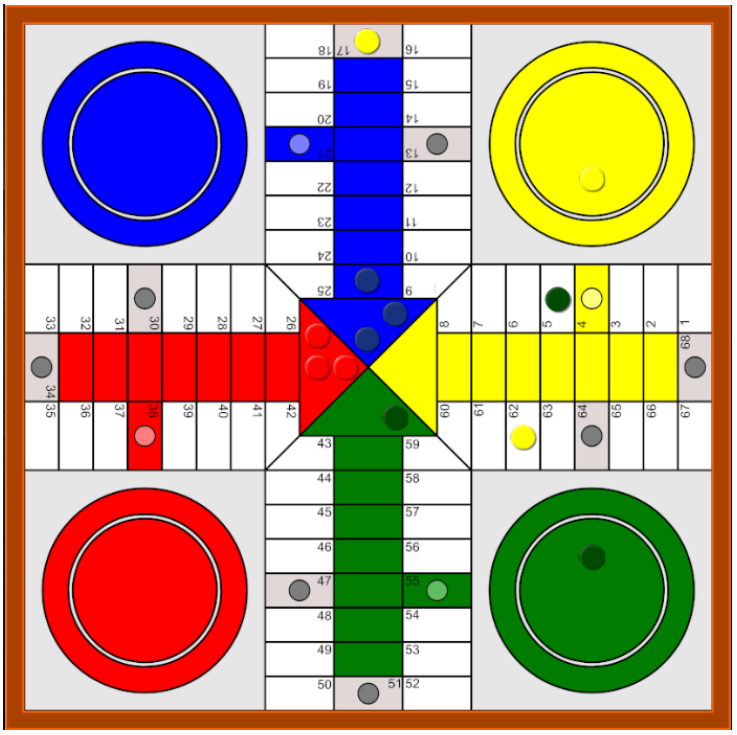


**Figura 1:** Imagen de una partida de PARCHIS en la que ha ganado el color rojo (jugador 1).

Casi seguro que todos/as hemos jugado al parchís alguna vez en nuestra infancia, y en mayor o menor medida todos/as conocemos las normas generales. Aunque es común que cada uno/a en nuestra casa hayamos jugado con nuestras propias reglas y puede que no tengamos una posición común en determinados aspectos, como por ejemplo, en el funcionamiento de las barreras. Además de esto, en esta versión del juego se han adaptado algunas de las reglas con la finalidad de hacer las partidas más fluidas y entretenidas. A modo de **resumen** (de las normas generales y de las adaptaciones), a continuación aclaramos las normas que se siguen en esta versión del juego:

UNIVERSIDAD DE GRANADA

/ UGR / decsai

Departamento de Ciencias de la Computación e Inteligencia Artificial

## REGLAS BÁSICAS

1. Juegan dos jugadores con dos colores cada uno. Jugador 1 (amarillo y rojo) vs Jugador 2 (azul y verde). Gana el primer jugador que lleve TODAS fichas de CUALQUIERA de sus dos colores a la meta. En esta versión del juego, el tablero dispondrá solo de **3 fichas** para cada color.

2. El orden de juego es Amarillo → Azul → Rojo → Verde.

3. En cada turno, **el jugador elige un dado** del color que toca y mueve la ficha que quiera de **cualquiera de sus dos colores**. **El dado elegido se gasta** y no puede volver a ser usado hasta que se gasten todos los dados. En ese momento reaparecen de nuevo todos los dados. Los dados disponibles inicialmente son los números del 1 al 6, salvo el número 3, y el dado **especial.**

4. Para sacar una ficha de la casa hay que sacar un 5.

5. Si se saca un 6 se vuelve a tirar.

6. Cuando se saca un 6 y todas las fichas de ese color están fuera de la casa, se avanza 7 casillas en lugar de 6.

7. No puede haber más de dos fichas en la misma casilla, salvo en las de casa o meta. Si ya hay dos, una tercera ficha no podría moverse a esa casilla con su tirada.

8. Dos fichas del mismo color en la misma casilla forman una barrera. Una ficha de otro color no puede pasar dicha barrera hasta que se rompa. Una ficha del mismo color sí puede saltarse la barrera.

9. Si se saca un 6 y hay alguna barrera del color que toca, es obligatorio romper dicha barrera. No se puede mover una ficha que no sea de una barrera.

10. Cuando una ficha llega a una casilla no segura donde hay una ficha de otro color se come esa ficha. Esa otra ficha vuelve a su casa. El jugador que se come la ficha se cuenta 20 con la ficha que desee de cualquiera de sus dos colores. Esto se gestiona en un turno adicional en el que el jugador solo tiene disponible el movimiento **+20**.

11. Los dos colores de un mismo jugador pueden comerse entre sí. Por ejemplo, una ficha amarilla puede comerse a una roja si se da el caso, aunque las dos sean del J1.

12. En las casillas seguras (marcadas con un círculo) pueden convivir dos fichas de distintos colores. No se puede comer una ficha que esté en esas casillas. Aunque haya dos fichas de distintos colores en una casilla segura no actúan como barrera, es decir, cualquier otra ficha puede saltarse esa casilla.

13. Para llegar a la meta, hay que sacar el número exacto de casillas que faltan para llegar. Si se saca de más, la ficha *rebota* y empieza a contar hacia atrás el exceso de casillas.

14. El número de rebotes totales que puede realizar un color a lo largo de una partida está limitado a 30. Si se supera ese número, el jugador pierde la partida. Esta es una regla artificial, cuya única finalidad es evitar que se produzcan partidas infinitas.

15. Cuando una ficha llega a la meta, se cuenta 10 con cualquiera de las otras fichas de cualquiera de sus dos colores. Esto se gestiona en un turno adicional en el que el jugador solo tiene disponible el movimiento **+10**.

16. En cada turno es obligatorio elegir un dado de los que no se hayan usado. Si para el valor del dado elegido no se puede mover ninguna ficha se puede gastar ese dado y pasar el turno sin que el jugador haga ningún movimiento.

17. Con el fin de agilizar las partidas, las fichas no aparecerán todas en casa al inicio de la partida, si no organizadas como vemos en la Figura 2.

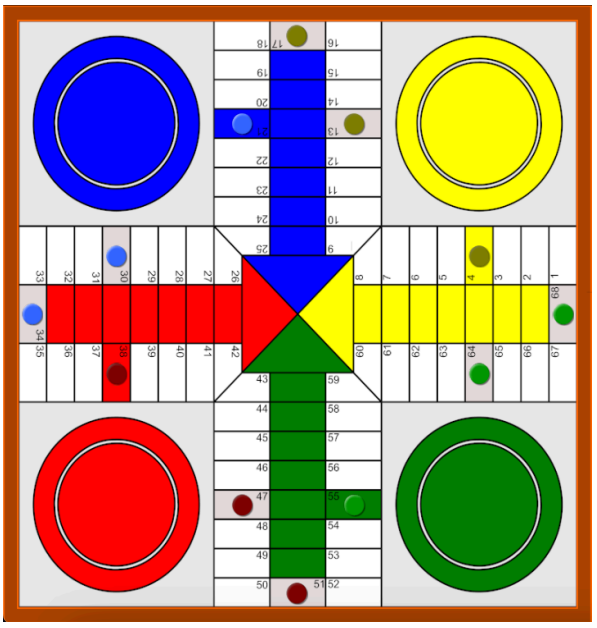


**Figura 2:** Configuración del tablero inicial.

Pero estas no son las únicas reglas que formarán parte de la versión del juego a la que vamos a jugar en esta práctica. Para hacer las partidas más divertidas, se ha incluido un dado **especial**, con el cual podremos hacer cosas inimaginables en una partida de parchís normal. El efecto de este dado dependerá, en cada momento, del valor de la **barra de energía** del jugador. Las acciones que puede llevar a cabo este dado **especial** están inspiradas en un popular juego de carreras, pero no os confiéis, la barra está algo estropeada y algunas acciones pueden ser perjudiciales para el propio jugador **(boom!)**.

## BARRA DE ENERGÍA

Cada jugador dispone de su propia **barra de energía**. El valor de la barra de energía se actualiza con cada movimiento:

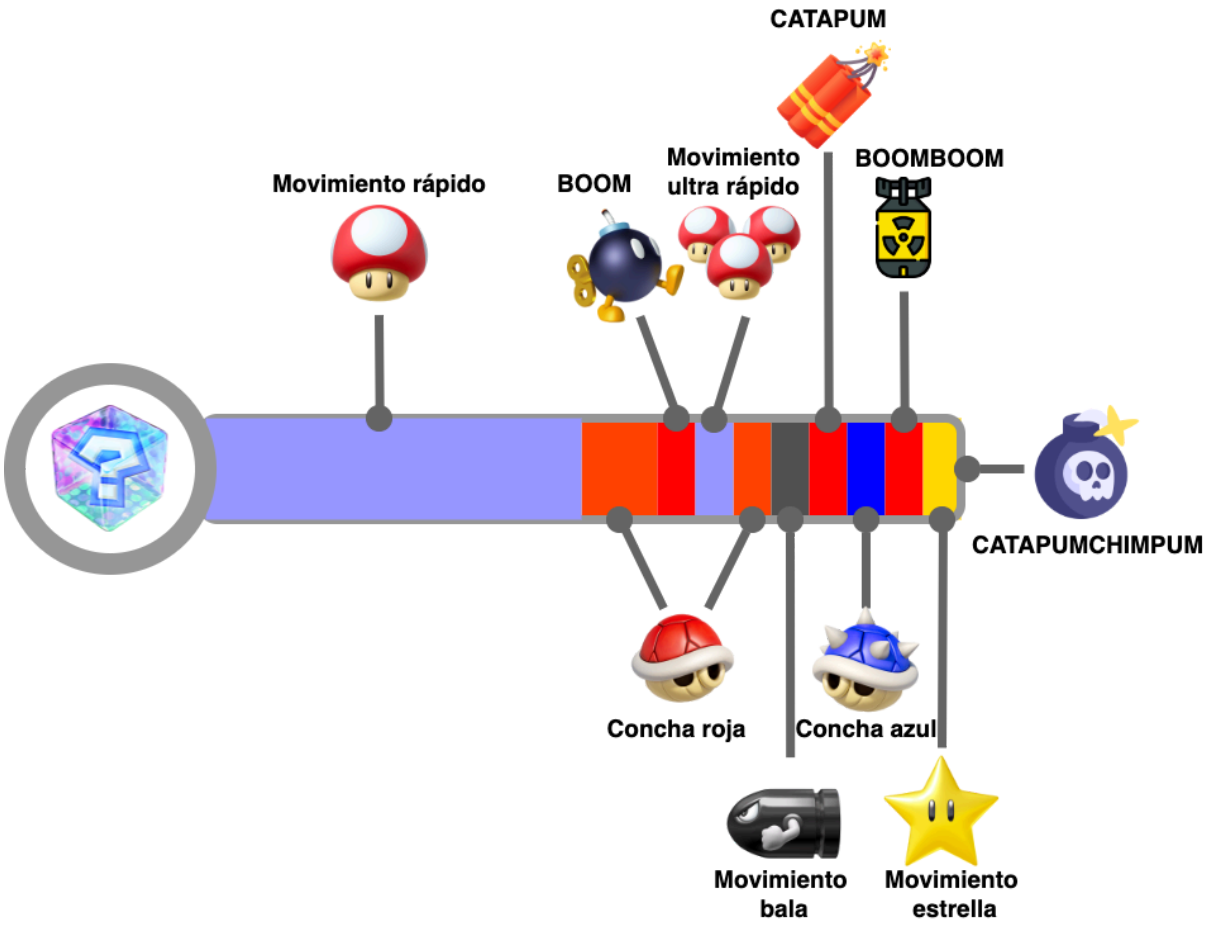
1. Cuando se hace uso del dado **especial**, se consume toda la **energía** quedando la barra de energía a 0 al finalizar ese turno.
2. Cuando se realiza cualquier otro movimiento, la **energía** se incrementa en tantos puntos como casillas se haya avanzado. Además, se puede sumar **energía** extra en las siguientes situaciones:
    a. +5 puntos cuando el movimiento acaba en casilla segura.
    b. +10 puntos cuando el movimiento forma una nueva barrera.
    c. +15 puntos cuando la ficha que se mueve se come a otra ficha.
3. La **energía** máxima se establece a 100 para cada jugador. Si con un movimiento se supera ese valor, la **energía** se truncará a 100.

## ACCIONES DADO ESPECIAL

A continuación, detallamos las acciones en función del valor de la **barra de energía**. Para llevar a cabo las acciones, habrá que seleccionar una ficha válida.

- **Movimiento rápido: [0, 50).** La ficha seleccionada avanza un total de ***7 + (energía / 7)*** **casillas**. Además, si la casilla de llegada estuviera ocupada, la ficha seguirá avanzando hasta una casilla libre. En caso de sobrepasar la meta con este movimiento rápido, la ficha no rebotará, se quedará en la meta. El movimiento ignora barreras y cualquier otro obstáculo.
- **Concha roja: [50, 60) y [70,75).** Si se utiliza el dado **especial**, se **mandará a casa automáticamente a la ficha más cercana de distinto color que esté delante de la ficha seleccionada**, ignorando barreras y/o casillas seguras. En caso de que haya varias a la misma distancia, eliminará a todas ellas.
- **BOOM: [60, 65).** Si, *entiendo que por error*, se utiliza el dado **especial**, se **mandará a casa automáticamente la peor ficha** (mayor distancia a meta, sin contar las que estén en casa) del jugador independientemente de la ficha seleccionada. La explosión generada afecta a todas las fichas en un radio de dos casillas, que vuelven a su casa también.
- **Movimiento ultra rápido: [65, 70).** La ficha seleccionada avanza un total de **25 casillas**. Además, si la casilla de llegada estuviera ocupada, la ficha seguirá avanzando hasta una casilla libre. En caso de sobrepasar la meta con este movimiento rápido, la ficha no rebotará, se quedará en la meta. El movimiento ignora barreras y cualquier otro obstáculo.
- **Movimiento bala: [75, 80).** La ficha seleccionada avanza un total de **40 casillas**. Además, si la casilla de llegada estuviera ocupada, la ficha seguirá avanzando hasta una casilla libre. En caso de sobrepasar la meta con este movimiento rápido, la ficha no rebotará, se quedará en la meta. El movimiento ignora barreras y cualquier otro obstáculo.
- **CATAPUM: [80, 85).** Si, *debido a una mala decisión*, se utiliza el dado **especial**, se **mandará a casa automáticamente la mejor ficha** (menor distancia a meta, sin contar las que estén ya en la meta) del jugador independientemente de la ficha seleccionada. La explosión generada afecta a todas las fichas en un radio de dos casillas, que vuelven a su casa también.

- **Concha azul: [85, 90).** Si se utiliza el dado **especial**, se **mandará a casa automáticamente a la ficha de distinto color que esté más cerca de la meta** (que no haya llegado aún), ignorando barreras y/o casillas seguras. En caso de que haya varias a la misma distancia, eliminará a todas ellas.
- **BOOMBOOM: [90, 95).** Si, *porque malas decisiones tomamos todos*, se utiliza el dado **especial**, se **mandará a casa automáticamente la mejor ficha de cada color** (menor distancia a meta, sin contar las que estén ya en la meta) del jugador independientemente de la ficha seleccionada. La explosión generada afecta a todas las fichas en un radio de dos casillas, que vuelven a su casa también.
- **Movimiento estrella: [95, 100).** La ficha seleccionada no se mueve en ese turno, pero pasa a ser **invencible** durante los próximos **5 movimientos del jugador**. No se ve afectada por movimientos de otras fichas ni explosiones. Cuando otra ficha pasa por su casilla, muere (se va a casa). Cuando se mueve, mata (manda a casa) a todas las fichas que se cruce por el camino. Además, cada vez que realice un movimiento, avanza 2 casillas más del dado seleccionado.
- **CATAPUMCHIMPUM: 100.** Si, *porque un mal día lo tiene cualquiera*, se utiliza el dado **especial**, se **mandará a casa automáticamente a todas las fichas del mejor color** (menor distancia a meta, sin contar las que estén ya en la meta) del jugador independientemente de la ficha seleccionada. La explosión generada afecta a todas las fichas en un radio de dos casillas, que vuelven a su casa también.



**OBJETIVO DE LA PRÁCTICA**

A partir de estas consideraciones iniciales, el objetivo de la práctica es implementar MINIMAX **(con profundidad máxima de 4)** o PODA ALFA-BETA **(con profundidad máxima de 6)**, 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 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 el juego del **PARCHÍS.**

Los conceptos necesarios para poder llevar a cabo la implementación del algoritmo dentro del código fuente del simulador se explican en las siguientes secciones.

## 4. Instalación y descripción del simulador

## 4.1. Instalación del simulador

El simulador **PARCHÍS** nos permitirá

- Implementar el comportamiento de uno o dos jugadores en un entorno en el que el jugador (bien humano o bien máquina) podrá competir con otro jugador software o con otro humano.
- Visualizar los movimientos decididos en una interfaz de usuario.

Para instalarlo, seguir estos pasos especificados en el GitHub de la asignatura.

## 4.2. Ejecución del simulador

Una vez compilado el simulador y tras su ejecución debe aparecernos la siguiente ventana:

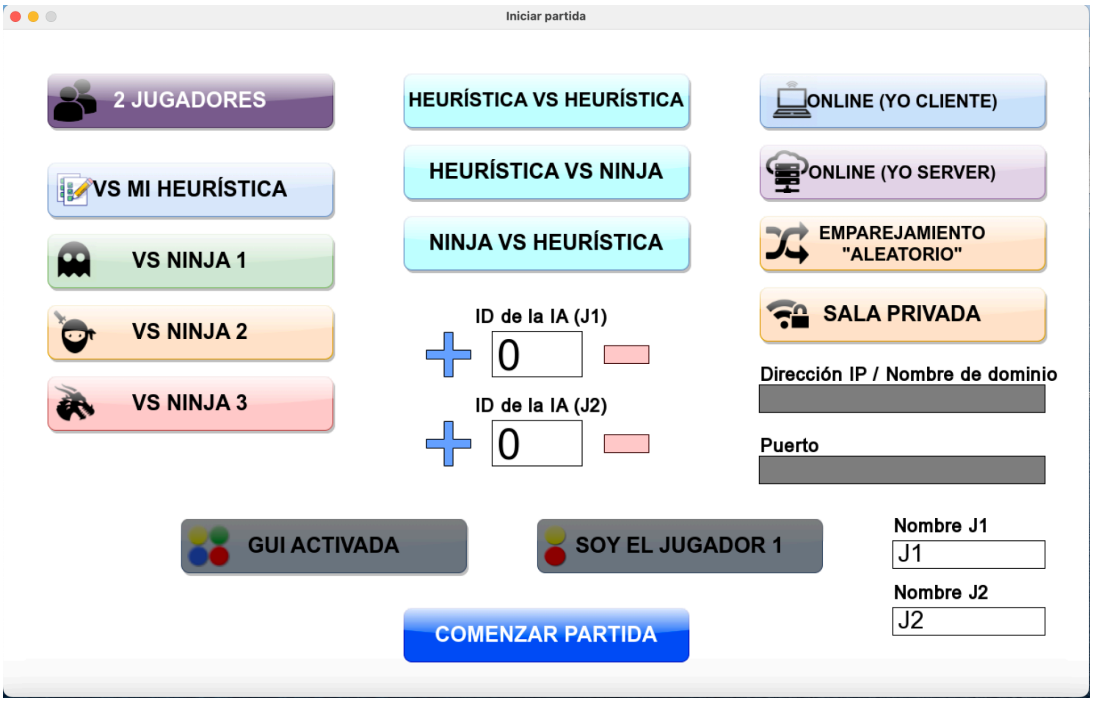



**Figura 3.** Ventana de inicio de juego.

Donde podremos elegir en qué modo de juego de entre los disponibles queremos elegir las diferentes configuraciones para la partida.

Las opciones configurables en el juego son las siguientes:

- "**Modo de Juego**": Establece la forma en la que se va a comportar el simulador. Se puede elegir entre 5 modos diferentes:
  - "*2 Jugadores*": En este modo de juego se jugará por los dos jugadores con GUI, pudiendo elegir entre hacer el movimiento manual, o con la heurística especificada por el ID de cada jugador en cada turno.
  - "*Vs mi heurística*": En este modo un jugador humano juega contra la heurística que él mismo ha programado en movimientos alternos. De igual forma, en el turno del humano se puede elegir utilizar la heurística asociada a su jugador.
  - "*Online (yo Server)*": En este modo se permite jugar por red con otro adversario. Obviamente, en este modo se requiere que un compañero tenga levantado su simulador,

elija el modo *Online (yo Cliente)* y ponga en *Dirección IP* la dirección IP de la máquina donde se encuentra el compañero en el modo servidor. Para poder actuar de servidor sería necesario que cliente y servidor estén dentro de una misma red privada, o que el servidor disponga de una IP pública o pueda acceder a ella mediante una redirección de puertos o mecanismo similar. El cliente será el que decida si es jugador 1 o 2. El servidor será el contrario.

- "*Online (yo Cliente)*": El complementario de lo descrito justo anteriormente para jugar con un compañero en red. La máquina servidora debe estar escuchando por el puerto que se especifique antes de iniciar la conexión. El cliente es el que decide si es jugador 1 o 2. El servidor será el contrario.
- "Vs Ninja 1", "Vs Ninja 2", "Vs Ninja 3" : Este es un modo especial de juego en red contra distintos jugadores automáticos. Se puede utilizar este modo para evaluar cómo de buena es la heurística que se ha implementado. **Es posible que para jugar a estos modos (y a los que se describen a continuación) sea necesario estar conectado a la VPN de la universidad.**
- "Emparejamiento 'Aleatorio'": En este modo de juego se puede jugar entre dos compañeros sin necesidad de que ninguno de los dos actúe como servidor. Una tercera máquina externa será la encargada de hacer de puente entre los dos jugadores. El "Aleatorio" está entre comillas ya que el método de emparejamiento consistirá en ir emparejando a cada par de jugadores en el orden en el que llegan, de forma que sea posible ponerse de acuerdo para coincidir en una misma partida con mayor facilidad. El número del jugador se determina por el orden de llegada. El jugador 1 será el que se conecte primero a la máquina puente.
- "Sala privada": Este modo de juego también permite conectar a dos clientes a través de una tercera máquina que hace de puente. En este caso, los jugadores podrán especificar una palabra con la que identificar a una sala a la que solo podrán entrar dos clientes que conozcan dicha palabra. Una vez se llene la sala comenzará la partida. De nuevo, el orden de juego de cada jugador se asignará según el orden de entrada a la sala.

- "**Jugador**": Decides qué jugador quieres ser. El jugador 1 siempre juega primero.
- "**ID de la IA (J1/J2)**": ID que define a la IA. Se podría utilizar para probar diferentes heurísticas. Los ninjas la tienen asignada por defecto.
- "**Dirección IP / Nombre de dominio**": La dirección IP o nombre de dominio del servidor contra el que quieres jugar en red.
- "**Nombre de la sala**": Nombre para identificar la sala en caso de elegir el modo de "Sala privada". Aparece cuando se selecciona dicho modo en lugar de la dirección IP.
- **"Puerto"**: El puerto en el que escucha el servidor contra el que quieres jugar en la red.

- **"GUI activada/desactivada"**: Especifica si el juego se lanza con interfaz gráfica o no. En algunos modos de juego está fijada por defecto.
- **"Nombre J1" / "Nombre J2":** Permite establecer nombres personalizados para cada jugador.

También encontramos tres formas de juego que realizan la partida completa entre dos heurísticas, pudiendo elegirse la versión con o sin GUI. Estas formas son:

- "*Heurística vs Heurística*": La heurística implementada compite contra sí misma, pudiendo llamarse con valores diferentes de ID y así probar.
- "*Heurística vs Ninja*" y "*Ninja vs Heurística*": se realiza una partida automática entre la heurística y el ninja seleccionados, pudiendo elegir qué jugador es el ninja y cuál la heurística según el botón seleccionado.

Tras darle al botón de *Comenzar partida* nos aparecerá el siguiente panel, donde tenemos los dados, el tablero de juego, varios botones para gestionar el audio, y los botones para mover. En cada turno con un jugador con GUI podrá:

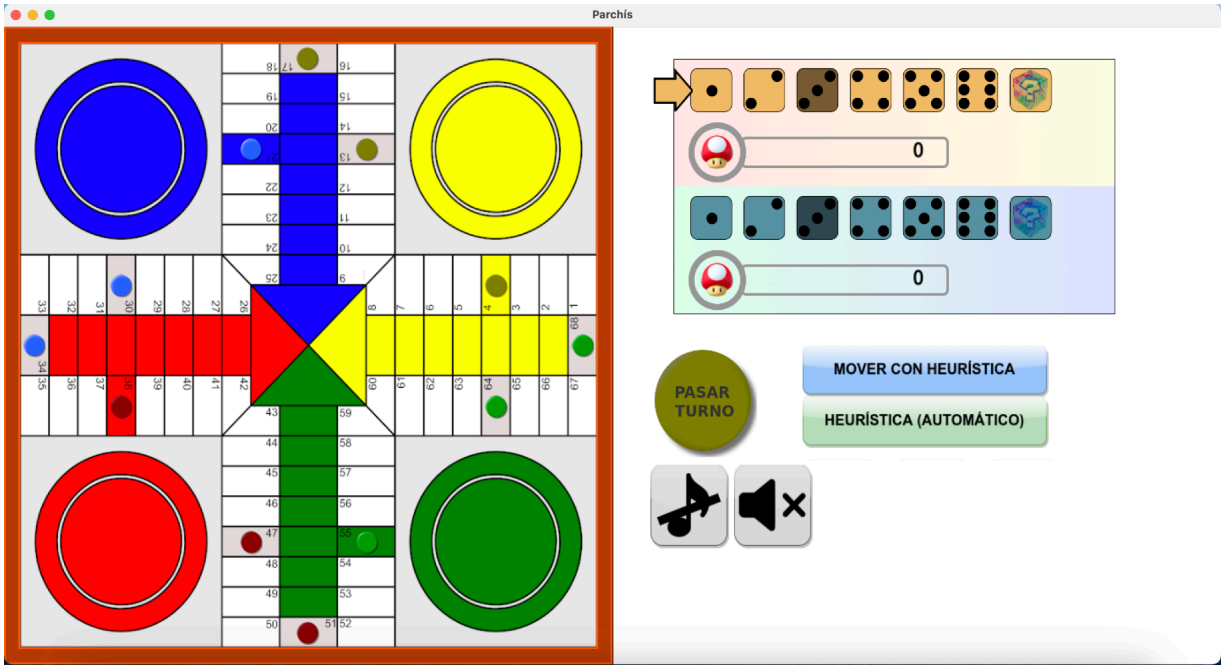



**Figura 4:** Ventana Principal del Juego.

1. "*Mover con heurística*": Se realizará el movimiento determinado como mejor según la heurística que se haya elegido para jugar.

2. Movimiento manual: Pulsar un dado de entre los disponibles de su color, y posteriormente una ficha. Si el movimiento no es válido, la ficha no podrá ser seleccionada.

Finalmente, encontramos la opción de "*Heurística (automático)*". Mientras esté activado este botón, todos los movimientos asociados a jugadores controlables mediante la interfaz gráfica se realizarán automáticamente llamando a la función heurística.

## 4.3. Ejecución del simulador desde línea de comandos

Con la finalidad de poder acceder de forma más directa a todas las opciones que presenta el simulador, se proporciona también una versión batch de **Parchís** con la que, directamente desde línea de comandos, se puede acceder a las distintas opciones que proporciona el juego. La sintaxis es como sigue:

```
./build/ParchisGame <opciones_normales> | <opciones_servidor> |
                <emparejamiento_aleatorio> | <sala_privada> (--no-gui)
  - <opciones_normales> = --p1 <opciones_p1> --p2 <opciones_p2>
                                (--ip direccion_ip=localhost) (--port=8888)
      - <opciones_p1> = [GUI|AI|Remote|Ninja] (id=0) (name=J1)
      - <opciones_p2> = [GUI|AI|Remote|Ninja] (id=0) (name=J2)
  - <opciones_servidor> = --server [GUI|AI] (id=0) (name=Server) (--port=8888)
  - <emparejamiento_aleatorio> = --random [GUI|AI] (id=0) (name=J1)
  - <sala_privada> = --private <nombre_sala> [GUI|AI] (id=0) (name=J1)
```

Por ejemplo, con el comando:

./build/ParchisGame **--p1 AI 0** Juanlu **--p2 Ninja 1** Nuria **--no-gui**

estoy diciendo que quiero una partida en la que el jugador 1 es mi heurística (**AI**) asociada al id número **0**, y juego contra el jugador 2, que es el Ninja **1** (**Ninja**). Además, **no** quiero que se abra la **interfaz gráfica**, la partida se jugará solo por terminal. Si no especifico el --no-gui la interfaz sí se abrirá.

De las opciones restantes para el jugador, GUI proporciona un jugador que puede mover las fichas desde la interfaz gráfica o usando los botones de las heurísticas, y Remote permitirá conectarnos como cliente a un jugador remoto que ha lanzado su Parchís en modo servidor.

Por último, en las partidas remotas se deben especificar las opciones --ip y --port para indicar IP y puerto cuando sea el caso.

En la siguiente sección se explica el contenido de los ficheros fuente y los pasos a seguir para poder construir la práctica.

## 5. Pasos para construir la práctica

Para implementar vuestra solución, el alumno deberá modificar únicamente:

- ***AIPlayer.h y cpp***, donde se implementa la clase ***AIPlayer*** usada para representar a cada uno de los dos jugadores inteligentes. Estos archivos son los únicos que modificaréis y contendrán TODA LA LÓGICA de vuestra solución.

Además, podéis consultar (no modificar) los siguientes archivos de utilidad para vuestra solución:

- ***Parchis.h y cpp***, donde se implementa la clase ***Parchis*** usada para representar los diferentes estados del juego y las interacciones entre ellos.
- ***Board.h y cpp y Dice.h y cpp***, donde se implementan algunas utilidades necesarias para la gestión del tablero y los dados de los jugadores.
- ***Attributes.h y cpp,*** donde se definen las casillas del tablero y los colores.

**IMPORTANTE: Los ficheros que se entregarán, y los únicos que se deben modificar para la práctica, son AIPlayer.h y AIPlayer.cpp!!!**

### 5.1. Clases auxiliares para la representación del juego (Clases Board, Dice y Box)

Los diferentes estados de juego estarán representados por un momento concreto del tablero, incluyendo la posición de las fichas y los dados disponibles para cada color. Las clases que se encargan de esta representación son las siguientes:

#### 5.1.1. Struct Box

En primer lugar definimos una **box** o casilla, que viene definida en el fichero *Attributes.h* de la siguiente forma:

Vemos que una casilla está definida por su número, color y tipo, y que se implementan simplemente los constructores y operadores de igualdad y el menor que.

```
//Struct para definir las casillas: número de casilla, tipo y color.
struct Box
{
   //Número de casilla:
   //{1, 2, ..., 68} para casillas normales (normal)
   //{1, 2, ..., 7} para casillas del pasillo a la meta (final_queue)
```

```
    //0 en caso contrario
    int num;
    //Tipo de la casilla
    box_type type;
    //Color de la casilla
    color col;

    /**
     * @brief Constructor de un nuevo objeto Box
     *
     * @param num
     * @param type
     * @param col
     */
    inline Box(int num, box_type type, color col){
        this->num = num; this->type = type; this->col = col;
    }

    /**
     * @brief Constructor por defecto de un objeto Box
     *
     */
    inline Box(){};
};
```

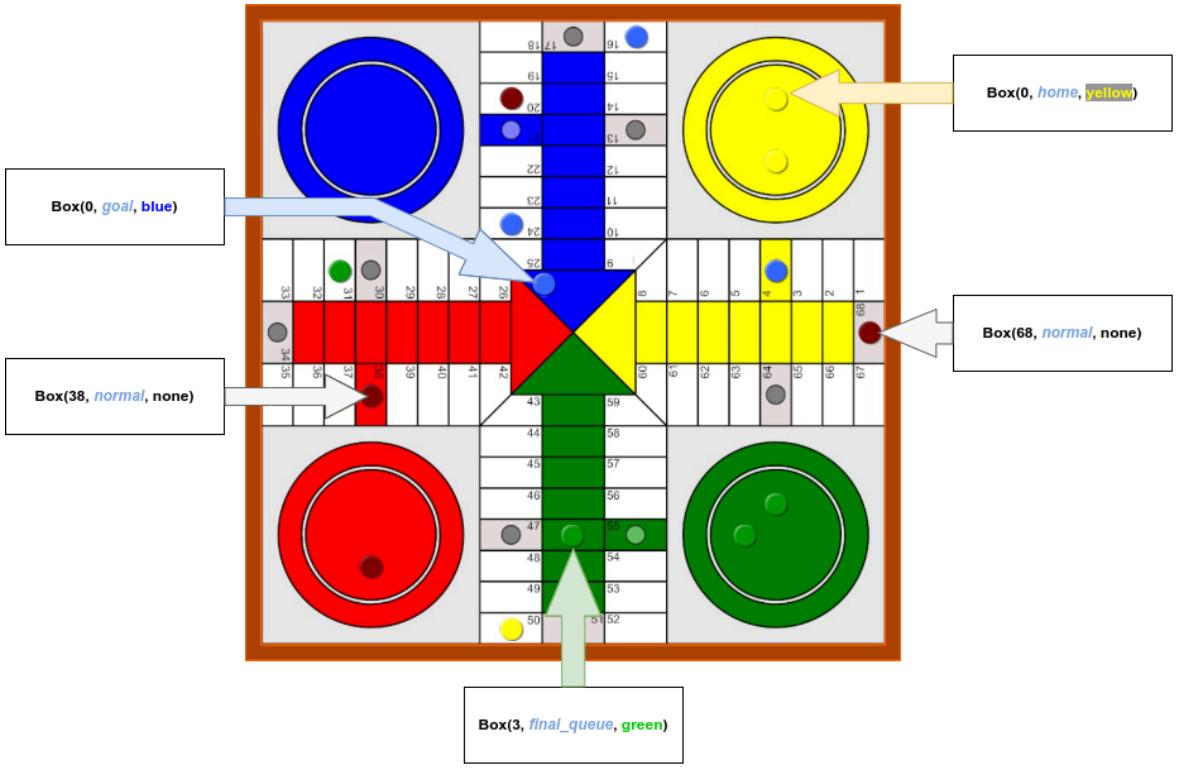
En el mismo fichero definimos los enumerados de posibles colores y tipos de casillas. Con respecto a los colores incluimos los 4 colores del parchís y el color vacío (para las casillas blancas), y el respectivo paso a string a partir de un objeto de tipo ***color.*** Finalmente, definimos en esta clase los 4 tipos de casillas que encontramos en el parchís:

```
//Enumerado con los diferentes colores: Azul, rojo, verde, amarillo y ninguno.
enum color {blue, red, green, yellow, none};

inline string str(color c){
    switch(c){
        case blue: return "Azul";
        case red: return "Rojo";
        case green: return "Verde";
        case yellow: return "Amarillo";
        case none: default: return "???";
    }
}
```

1. ***home***, para las casillas de inicio de cada color.
2. ***final_queue***, para el pasillo final hacia la meta de cada uno de los colores.
3. ***goal***, para la casilla destino o meta de cada uno de los colores.
4. ***normal***, para el resto de casillas.

La siguiente figura muestra cómo se traducen las casillas del tablero a nivel de programación usando la clase **Box** y los enumerados de **color** y **box_type**.




## 5.1.2. Clase Board

Esta clase representa el estado del tablero del juego. Un tablero está representado por la posición de sus fichas. Para ello, lo que hacemos es representarlo con el atributo de clase *pieces* que se define como un *map* en el que identificamos cada color con un vector de casillas. Así, almacenamos para cada uno de los 4 colores de fichas, dónde están cada una.

```
class Board{
   private:
       //Conjunto de todas las piezas del tablero
       //Se usa como identificador el color, y después un vector con las casillas en las que
       //está cada una de las 4 piezas.
```

```
map<color, vector<Piece> > pieces;
```

Esta clase implementa diferentes constructores: por defecto, a partir de un *map* con las posiciones de las fichas, y a partir de una configuración concreta del tablero.

También implementa funciones para obtener las casillas donde se encuentran determinadas fichas, o mover fichas dentro del tablero. Estas funciones se utilizan desde la clase ***Parchis,*** y también pueden ser útiles para consultar la posición de las fichas en el tablero.

```
/**
* @brief Función que devuelve el Box correspondiente a la ficha
* en la posición "idx" del vector de fichas de color "c".
*
* @param c
* @param idx
* @return Box
*/
const Piece & getPiece(const color c, const int idx) const;

/**
* @brief Función que devuelve el vector de Box del color "c".
*
* @param c
* @return Box
*/
const vector<Piece> & getPieces(const color c) const;
```

### 5.1.3. Clase Dice

De forma similar a la clase ***Board***, en esta clase almacenamos un *map* con los dados disponibles para cada color. Los dados disponibles se almacenan como un vector de enteros en varias capas. La primera capa para los dados disponibles de entre los dados clásicos {1, 2, 4, 5, 6} y el dado especial. Y otra capas para los dados forzados {10, 20}.

```
class Dice{
   private:
       /**
        * @brief Dados para cada jugador. Los dados se agrupan por capas.
        *
```

```
 * - Capa 1: Dados clásicos del 1-6 (quitando el 3) + dado especial.
 * - Capa 2: Dados forzados (mover 10 o 20)
 *
 */
map <color, vector<vector <int>> > dice;
```

Esta clase cuenta con constructor por defecto y a partir de un determinado *map* de dados, además de diferentes funciones para acceder a la información de *dice* y modificarla. Estas funciones se utilizan desde la clase ***Parchis***. También puede ser de utilidad la función getDice(color), que nos permite consultar los dados que puede usar un jugador en un momento determinado.

## 5.2. Representación del juego (Clase Parchis)

La clase ***Parchis*** gestiona toda la representación del juego. Para ello, hace uso de las clases definidas previamente.

Los atributos que definen esta clase son:

1. El tablero y el dado actuales: board y dice.
2. Variables para almacenar los últimos movimientos y dados obtenidos: last_moves, last_action y last_dice.
3. Variable que almacena el turno actual de juego: turn.
4. Y cuál es el color y el jugador actual: current_player y current_color.

```
class Parchis{
   private:
        //Tablero
        Board board;
        //Dados
        Dice dice;

        //Variables para almacenar los últimos movimientos
        //Últimos movimientos identificados por el color.
        vector<tuple <color, int, Box, Box>> last_moves;
        //Última acción identificada por el color.
        tuple <color, int, int> last_action;
        //Último dado utilizado.
        int last_dice;

        //Turno actual
        int turn;
```

```
//Jugadores y colorores actuales
//0: yellow & red, 1: blue and green.
int current_player;
color current_color;
```

5. Variables auxiliares para controlar las acciones de los jugadores y los movimientos especiales.

6. Variables para almacenar las casillas especiales para cada color.

## Métodos destacables de la clase Parchis

A continuación, se describen los métodos esenciales de esta clase, para la comprensión y la elaboración de la práctica.

1. **Obtener quién ha ganado:** Funciones que indican si ha terminado la partida y quien la ha ganado.

```
/**
* @brief Indica si la partida ha terminado.
*
* @return true
* @return false
*/
bool gameOver() const;


/**
* @brief Si la partida ha terminado, devuelve el índice del jugador ganador (0 o 1).
*
* @return int
*/
int getWinner() const;

/**
* @brief Si la partida ha terminado, devuelve el color del jugador ganador.
*
* @return color
```

```
*/
color getColorWinner() const;
```

2. **Número de fichas en la meta:** Función que devuelve el número de fichas de un determinado color que están ya en la meta.

```
/**
* @brief Devuelve el número de fichas de un color que han llegado a la meta.
*
* @return int
*/
int piecesAtGoal(color player) const;
```

3. **Funciones de distancia a la meta:** Estas funciones devuelven la distancia a la meta tanto de una ficha concreta como de una casilla para un jugador determinado.

```
/**
* @brief Función que devuelve la distancia a la meta del color "player" desde
* la casilla "box".
*
* La distancia se entiende como el número de casillas que hay que avanzar hasta
* la meta.
*
* @param player
* @param box
* @return int
*/
int distanceToGoal(color player, const Box & box) const;

/**
* @brief Función que devuelve la distancia a la meta de la ficha identificada
* por id_piece del jugador identificado por player.
*
* La distancia se entiende como el número de casillas que hay que avanzar hasta
* la meta.
*
* @param player
* @param id_piece
```

```
* @return int
*/
int distanceToGoal(color player, int id_piece) const;
```

4. **Casillas y fichas seguras:** Funciones que devuelven si una determinada casilla es segura, o si una determinada ficha se encuentra en una casilla segura.

```
/**
* @brief Función que devuelve si una determinada casilla es segura o no.
*
* @param box
* @return true
* @return false
*/
bool isSafeBox(const Box & box) const;

/**
* @brief Función que devuelve si una determinada ficha de un determinado está
* en una casilla segura o no.
*
* @param player
* @param piece
* @return true
* @return false
*/
bool isSafePiece(const color & player, const int & piece) const;
```

5. **Funciones para comprobar barreras:** Las siguientes funciones devuelven los colores de las barreras (en caso de haberlas) tanto en una determinada casilla, como en las casillas de un determinado recorrido.

```
/**
* @brief Función que devuelve el color de la barrera (en caso de haberla) en la casilla "b".
* Es decir, si en la casilla "b" hay dos fichas de un mismo color devuelve este color.
*
* @param b
* @return const color
*/
const color isWall(const Box & b) const;
```

```
/**
* @brief Función que devuelve el vector de colores de las barreras (en caso de haberlas) del
* camino entre b1 y b2.
*
* Esto es, se va recorriendo todas las casillas que habría que recorrer para ir de b1 y b2,
* y siempre que se encuentran dos fichas de un mismo color en una misma casilla se añade ese
* color al vector que se devuelve.
*
* Por ejemplo: si en la casilla 2 hay una barrera amarilla y en la 4 una azul, el
anywalls(1,6)
* devuelve {yellow, blue}
*
* @param b1
* @param b2
* @return const vector<color>
*/
const vector<color> anyWall(const Box & b1, const Box & b2) const;
```

6. **Acceso a las prolongaciones de Dice y Board:** estas funciones permiten acceder a una referencia constante al tablero y a los dados para realizar consultas. También dispone Parchis de algunas envolventes para funciones de dichas clases.

```
/**
* @brief Función que devuelve el atributo dice.
* @param player
* @return const vector<int>&
*/
const Dice & getDice () const;


/**
* @brief Función que devuelve el atributo board.
*
* @param player
* @return const vector<int>&
*/
```

```
const Board & getBoard () const;

/**
* @brief Función que devuelve todas las fichas de player que pueden
* hacer un movimiento según el valor del dado dice_number.
*
* Por ejemplo, si dice_number = 2, las fichas que se encuentran en home
* no aparecerán como disponibles.
*
* También se gestionan las barreras y otros casos particulares.
*
* @param player
* @param dice_number
* @return const vector<int>&
*/
const vector<tuple<color,int>> getAvailablePieces (color player, int dice_number) const;
```

```
/**
* @brief Obtener los números del dado disponibles para el jugador player.
*
* @param player
* @return const vector<int>&
*/
inline const vector<int> getAvailableNormalDices (int player) const{...}
```

7. **Otras funciones de consulta:** permiten comprobar si determinado movimiento es legal, si se puede pasar turno, si el último movimiento acabó con una ficha en la meta, siendo comida o rebotando, y el color actual.

```
/**
* @brief Función que comprueba si un movimiento es válido para las fichas de un determinado
* color en una determinada casilla. Tiene en cuenta barreras y otras particularidades.
*
*
```

```
* @param player
* @param box
* @param dice_number
* @return true
* @return false
*/
bool isLegalMove(const Piece & piece, int dice_number) const;

/**
* @brief Comprobar si el jugador puede pasar turno con el dado seleccionado (si no tiene
fichas para mover).
*
* @param player
* @param dice_number
* @return true
* @return false
*/
bool canSkipTurn(color player, int dice_number) const;

/**
* @brief Función que devuelve el valor del atributo eating_move
*
* @return true
* @return false
*/
inline const bool isEatingMove() const {...}
}

/**
* @brief Función que devuelve el valor del atributo goal_move
*
* @return true
* @return false
*/
inline const bool isGoalMove() const {...}
}

/**
* @brief Función que devuelve el valor del atributo goal_bounce
```

```
*
* @return true
* @return false
*/
inline const bool goalBounce() const{...}
}
```

## 5.2. Representación de los jugadores (Clase AIPlayer)

La clase ***AIPlayer*** se utiliza para representar un jugador (es la clase equivalente a *ComportamientoJugador* en la anterior práctica).

```
class AIPlayer: public Player{

   protected:
       //Id identificativo del jugador
       const int id;
   public:
       inline AIPlayer(const string & name):Player(name), id(1){};
       inline AIPlayer(const string & name, const int id):Player(name), id(id){};

       inline virtual void perceive(Parchis &p){Player::perceive(p);}
       virtual bool move();

       virtual void think(color & c_piece,  int & id_piece, int & dice) const;
       inline virtual bool canThink() const{return true;}

       static double ValoracionTest(const Parchis &estado, int jugador);
```

Contiene una variable protegida:

- ***id*** *(int)*: Almacena el identificador del jugador. Esta variable se puede utilizar dentro del método think para seleccionar diferentes heurísticas y así hacer pruebas rápidas de las heurísticas que se diseñen, o incluso enfrentar dos heurísticas diferentes entre sí.

Destacamos tres métodos:

1. ***think***: Que implementa el proceso de decisión del jugador para escoger la mejor jugada. El valor del mejor movimiento elegido por el jugador se almacena en las variables pasadas como referencia.
2. ***perceive,*** que implementa el proceso de percepción del jugador y que permite acceder al estado actual del juego que tiene el jugador. **Este método no puede ser modificado!!**
3. ***move*****,** es la función que se encarga de ejecutar la acción del jugador y de interactuar con el tablero de juego. **Este método no puede ser modificado!!**

IMPORTANTE**:**

**La invocación al MINIMAX o la PODA ALFA-BETA se debe hacer dentro del método** Think()

Todos los recursos necesarios para poder implementar un proceso de búsqueda con adversario se suministran fundamentalmente en la clase **Parchis**, y en su prolongación sobre **Board** y **Dice**. Podrán definirse los métodos que el/la alumno/a estime oportunos, pero tendrán que estar implementados en el fichero **AIPlayer.cpp**.

## 5.3. Versión Inicial del método think()

El estudiante al compilar por primera vez el software de la práctica se encontrará con la implementación en el método think() de un jugador aleatorio. La idea es ilustrar de forma práctica cómo se puede manipular la información que nos proporciona el tablero a través de la clase **Parchís**.

```
const double masinf = 9999999999.0, menosinf = -9999999999.0;
const double gana = masinf - 1, pierde = menosinf + 1;
const int num_pieces = 3;
const int PROFUNDIDAD_MINIMAX = 4;  // Umbral maximo de profundidad para el metodo MiniMax
const int PROFUNDIDAD_ALFABETA = 6; // Umbral maximo de profundidad para la poda Alfa_Beta
```

Antes de entrar en la descripción de este método, podemos observar en la parte superior del fichero AIPlayer.cpp las siguientes definiciones. Entre ellas tenemos las constantes **PROFUNDIDAD_MINIMAX = 4** y **PROFUNDIDAD_ALFABETA = 6**. Estás serán las profundidades máximas que podrá elegir el estudiante en cada caso. Es decir, que si implementa Minimax la profundidad máxima que podrá poner será 4, siendo 6 la que se puede usar en el caso de tener implementada la poda AlfaBeta.

Entrando ya en el método think() vemos que en primer lugar se asigna a la variable c_piece el color al que le toca mover en el juego en el turno actual. Esta variable es la que determina el color de la ficha a mover, por lo que en esta variante del juego la única posibilidad es elegir el color actual. Para

determinar la parte que nos queda del movimiento necesitamos elegir un número de dado y un id (del 0 al 3) para la ficha que queremos mover dentro de las de nuestro color.

```
void AIPlayer::think(color & c_piece, int & id_piece, int & dice) const{
   // IMPLEMENTACIÓN INICIAL DEL AGENTE
   // Esta implementación realiza un movimiento aleatorio.
   // Se proporciona como ejemplo, pero se debe cambiar por una que realice un movimiento
inteligente
   //como lo que se muestran al final de la función.

   // OBJETIVO: Asignar a las variables c_piece, id_piece, dice (pasadas por referencia) los
valores,
   //respectivamente, de:
   // - color de ficha a mover
   // - identificador de la ficha que se va a mover
   // - valor del dado con el que se va a mover la ficha.

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

Para determinar tanto el número de dado como el id de la ficha a mover vamos a obtener en primer lugar todos los dados disponibles en el turno actual. Para ello, consultamos la función getAvailableNormalDices(color). Y, dentro de los disponibles, nos quedamos con un dado al azar.

```
   // 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<tuple<color,int>> current_pieces;

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

Finalmente, una vez elegido el dado, miramos las fichas que se podrían mover con ese dado. Para ello tenemos la función getAvailablePieces(color, int). De nuevo elegimos una al azar, salvo que no pudiéramos mover ninguna ficha. En ese caso optamos por pasar el turno.

```
   // 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){
        int random_id = rand() % current_pieces.size();
        id_piece = get<1>(current_pieces[random_id]);
        c_piece = get<0>(current_pieces[random_id]);
    }
    else{
        // Si no tengo fichas para el dado elegido, pasa turno (la macro SKIP_TURN me permite
no mover).
        id_piece = SKIP_TURN;
    }
```

Una vez se han asignado los valores a las variables c_piece, id_piece y dice, queda completamente determinado el próximo movimiento a realizar por el jugador. Cuando finalice el método think dicho movimiento (en este caso aleatorio) procederá a ejecutarse.

## 5.4. Empezando a plantear el algoritmo de búsqueda

La última parte del método think() está comentada, y tendrá que ser descomentada cuando el estudiante haga su implementación del Minimax o de la Poda Alfabeta. En ese momento, se comentará en su lugar la parte del código relativa al jugador aleatorio.

```
 // El siguiente código se proporciona como sugerencia para iniciar la implementación del
agente.

double valor; // Almacena el valor con el que se etiqueta el estado tras el proceso de
busqueda.
double alpha = menosinf, beta = masinf; // Cotas iniciales de la poda AlfaBeta

// Llamada a la función para la poda (los parámetros son solo una sugerencia, se pueden
modificar).
valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece, dice,
alpha, beta, ValoracionTest);

cout << "Valor MiniMax: " << valor << "  Accion: " << str(c_piece) << " " << id_piece << " "
<< dice << endl;
```

La parametrización que se propone de la poda AlfaBeta es meramente indicativa. El estudiante es libre de decidir los parámetros que desea utilizar para realizar la búsqueda.

La primera tarea del estudiante en esta práctica es familiarizarse con el código y hacer una implementación de uno de los dos algoritmos de búsqueda. Para que resulte más fácil y el estudiante en este primer instante sólo piense en el algoritmo, se le ofrece una heurística para que la pueda utilizar, llamada **ValoracionTest**. Esta función simplemente se encarga de valorar de forma positiva las fichas de los colores de mi jugador que estén en casillas seguras o en la meta, a la vez que se penaliza por las fichas del oponente que estén en la misma situación. Con un algoritmo de búsqueda bien implementado, bajo esta heurística las fichas tenderán a moverse (siempre que puedan) a casillas en las que estén a salvo de los movimientos rivales.

Una vez verificado que el algoritmo implementado funciona correctamente, en la segunda tarea el estudiante tiene que definir una heurística propia (distinta de ValoracionTest) que permita hacer un jugador automático que juegue lo mejor posible contra otros compañeros y en especial contra los jugadores automáticos que se proponen junto al software.

Para la confección de dicha función heurística por parte del estudiante se pueden diseñar funciones con las mismas entradas y salidas que ValoracionTest, pero con distintos nombres, y se pueden diseñar tantas heurísticas como se deseen. Como pasaba con el método de búsqueda, en este caso, la parametrización de la función es sólo orientativa y el estudiante es libre de hacer la parametrización que crea oportuna. Todas las heurísticas implementadas pueden conservarse en el código e incluso enfrentarse entre ellas para que el estudiante pueda valorar cuál puede ser la más adecuada. Para ello se propone en la clase AIPlayer la variable de instancia **id**, la cual se puede establecer tanto por línea de comandos como por la interfaz gráfica antes de iniciar la partida. En función de este **id**, se podrían lanzar distintas heurísticas, procediendo de forma análoga o similar a como se muestra en la última sección comentada del método think():

```
// Si quiero poder manejar varias heurísticas, puedo usar la variable id del agente para usar
una u otra.
switch(id){
    case 0:
        valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece,
dice, alpha, beta, ValoracionTest);
       break;
    case 1:
        valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece,
dice, alpha, beta, MiValoracion1);
        break;
    case 2:
        valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece,
dice, alpha, beta, MiValoracion2);
        break;
}
```

```
cout << "Valor MiniMax: " << valor << "  Accion: " << str(c_piece) << " " << id_piece << " "
<< dice << endl;
```

**Importante: En ningún caso puede eliminarse ni alterarse la función ValoracionTest. Debe permanecer tal y como está cuando se produzca la entrega de la práctica. Esta función heurística será utilizada para validar el correcto funcionamiento de la implementación de la función de búsqueda realizada por el estudiante.**

## 5.5. Últimas consideraciones

En esta sección se describen brevemente ciertas consideraciones importantes a tener en cuenta a la hora de empezar a plantear esta práctica:

- En primer lugar, tanto si se opta por implementar Minimax como la Poda Alfa-Beta, va a ser necesario realizar una búsqueda en profundidad partiendo del estado actual del juego. En consecuencia, para cada nodo se hace necesario obtener todos los posibles tableros a los que se podría pasar, con cada uno de los posibles movimientos. Para ello, se ha implementado la clase **ParchisBros** que se encarga de la generación de los nodos hijos posibles a partir de un estado actual del parchis. Esta clase incluye un **iterador** que se usará para ir recorriendo los nodos hijos del nodo actual. Dado que desde un punto de vista estratégico, en muchas ocasiones un **valor de dado mayor** puede suponer una **mayor ventaja**, el operador++ del iterador se ha implementado recorriendo los hijos en orden descendente del dado.
- En segundo lugar, es importante destacar que en el juego propuesto **un turno se corresponde con un único movimiento de ficha, independientemente** de que ese movimiento sea repetido por el mismo jugador tras **sacar un 6**, que sea un movimiento de **contarse 10 o 20** tras llegar a la meta o comer, o que sea la acción que se ejecute mediante un dado especial. En consecuencia, como sucesor de un nodo MÁX podríamos encontrarnos de nuevo otro nodo MÁX (lo mismo para los MÍN). Por tanto debemos tener en cuenta que la secuencia de nodos no va a ir alternándose necesariamente en cada nivel. Tras sacar un 6 como nodo MÁX bajaríamos a un nuevo nodo MÁX, con 1 más de profundidad. Igualmente, tras comer ficha bajaríamos a un nuevo nodo del mismo tipo, en el que únicamente tendremos que elegir de entre nuestras posibles fichas con cuál contarnos 20. En cualquier caso, en todo momento podremos saber si somos un nodo MÁX o MIN, ya que conocemos el jugador que llama a la heurística y las funciones como **getCurrentPlayerId**, de la clase Parchis, nos indican a qué jugador le toca mover en cada turno. Recordemos que un nodo debería ser MÁX cuando el jugador que mueve es el que llamó al algoritmo de búsqueda.
- La **ramificación** del árbol de búsqueda **va a variar** de forma significativa **según la cantidad de dados** de los que dispongan los jugadores en cada turno. Por ello, es posible que el algoritmo de búsqueda sea **bastante lento inicialmente**, pero irá **aumentando su velocidad** conforme los jugadores vayan gastando dados, hasta que estos se renueven. También, conforme vaya avanzando la partida irán quedando menos opciones por mover, por lo que también tenderá a pensar más rápido conforme pasen los turnos.

- Las **clases descritas en la sección 5** disponen de funciones que pueden ser de mucha utilidad para el desarrollo de heurísticas. Es **importante echar un vistazo a las cabeceras** de las clases mencionadas para descubrir todas las posibilidades que se ofrecen. En el **tutorial** se experimenta con algunas de ellas para elaborar algunas estrategias simples para jugar al Parchís. Es **recomendable seguirlo** para adquirir manejo y posteriormente poder emplear las funciones de las clases proporcionadas en una heurística que pueda ser usada por el algoritmo de búsqueda.

## 6. Evaluación y entrega de prácticas

La **calificación final** de la práctica se calculará de la siguiente forma:

- Se entregará una memoria de prácticas (ver apartado 6.1 de este guión) al finalizar las tareas a realizar.
- La práctica se califica numéricamente de **0 a 10**. Se evaluará como la suma de los siguientes criterios:
  - La memoria de prácticas se evalúa de **0 a 3.**
  - La elección del método de búsqueda se evaluará entre **0 y 1**, siendo 0 si la elección ha sido Minimax y 1 si la elección ha sido la Poda Alfa-Beta, o valores intermedios en caso de haber algún error en la implementación de la misma.
  - La eficacia del algoritmo se evaluará de **0 a 6** puntos y estará basado en competir frente a tres jugadores ninja tanto de jugador 1 como de jugador 2. Con estas 6 partidas (3 como primer jugador y 3 como segundo jugador) se definirá la capacidad de la heurística desarrollada por el estudiante. La calificación será de **un punto por cada victoria**.
  - La nota final será la suma de los dos apartados anteriores. Es requisito necesario entregar tanto el software como la memoria de prácticas. Si alguna de ellas falta en la entrega se entenderá por no entregada.

- La **fecha de entrega de la práctica**

**Domingo 9 de junio antes de las 23:00 horas.**

### 6.1. Restricciones del software a entregar y representación.

Se pide desarrollar un programa (modificando el código de los ficheros del simulador **AIPlayer.cpp**, **AIPlayer.h**) que implemente el algoritmo MINIMAX (**en este caso la profundidad máxima permitida será de 4**) o la PODA ALFA-BETA (**donde la profundidad máxima en este caso será de 6**) en los términos en que se ha explicado previamente. Estos ficheros deberán entregarse en la plataforma docente PRADO, en un fichero ZIP que NO contenga carpetas separadas, es decir, todos los ficheros

aparecerán en la carpeta donde se descomprima el fichero ZIP. **No se evaluarán aquellas prácticas que contengan ficheros ejecutables o virus.**

El fichero ZIP debe contener una **memoria de prácticas** en formato PDF (no más de 5 páginas) que, como mínimo, contenga los siguientes apartados:

1. Análisis del problema.
2. Descripción de la solución planteada, detallando las heurísticas empleadas y las ventajas/inconvenientes de cada una de ellas. Detallar también qué heurística es la mejor (y por tanto, la que se quiere usar como solución). **IMPORTANTE: Para la evaluación se corregirá con id=1, así que asociad la mejor heurística a este id y valoracionTest con el id = 0.**

**Recordad que las prácticas son individuales!! Si hay sospecha de que esto no es así, se puede convocar a una defensa oral de la práctica.**

**Esperamos que os guste el juego** 🙂