# INTELIGENCIA ARTIFICIAL

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

# Práctica 3


## Búsqueda con Adversario (Juegos)

## El ParCheckers

**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL**
**UNIVERSIDAD DE GRANADA**
**Curso 2024-2025**

UNIVERSIDAD DE GRANADA

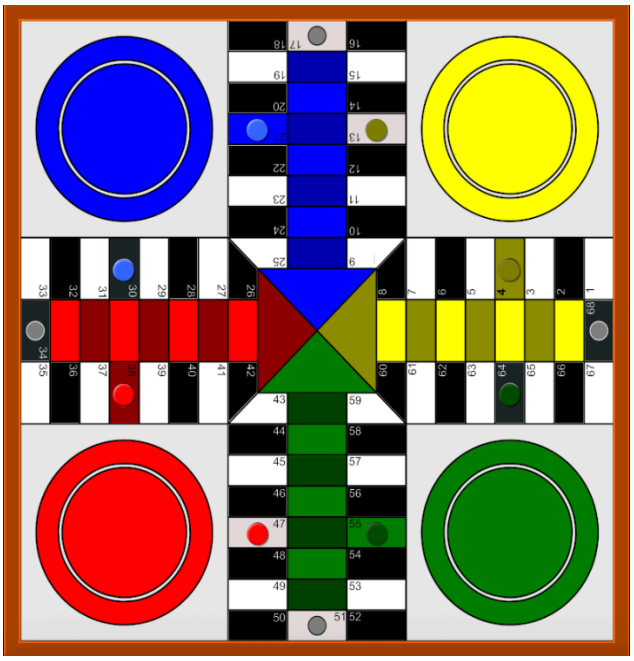
Departamento de Ciencias de la Computación e Inteligencia Artificial

/ UGR / decsai

# 1. Introducción

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 **ParCheckers,** que simula un Parchís determinista en el que hay dos jugadores que juegan con dos colores cada uno y los dados se van eligiendo entre un conjunto de dados disponibles. Además, las casillas del tablero son de dos colores alternos: blancas y negras.

El Parchís es un popular juego de mesa 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, habrá dos tipos de casillas en el camino: blancas y negras. Por la disposición del tablero, cada ficha empezará el juego en una casilla de un tipo (blanca o negra) determinado, y los movimientos por el tablero se harán sin cambiar de color, a no ser que se use el dado **yinyang**.

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 de PODA ALFA-BETA junto con su heurística**, **así como ser capaz de proponer e implementar mejoras del mismo** 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 **ParCheckers** 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 **ParCheckers** 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 **yinyang**). 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. 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 dos colores, estos colores podrán atacarse entre sí. Por ejemplo, aunque un jugador esté jugando con los

colores amarillo y verde, cuando se mueve una ficha amarilla a una casilla no segura donde había previamente una verde, la ficha amarilla se come a la ficha verde aunque sean del mismo jugador.

El objetivo de **ParCheckers** es conseguir meter TODAS las fichas en su casilla destino. **Gana** la partida **el primer jugador** que consiga **meter todas las fichas de sus dos colores en sus respectivas metas.**

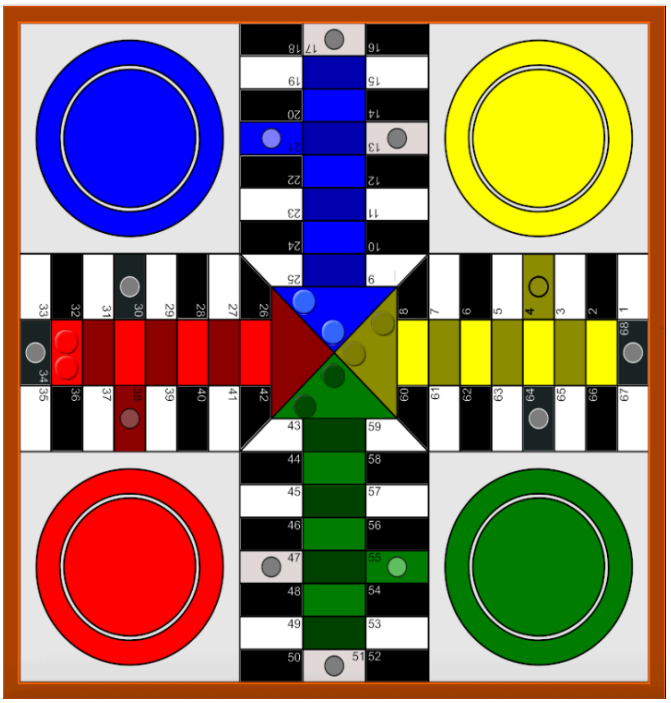


**Figura 1:** Imagen de una partida de **ParCheckers** en la que ha ganado el Jugador 1 (amarillo y verde).

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 **verde**) vs Jugador 2 (**azul** y **rojo**). Gana el primer jugador que lleve TODAS sus fichas (de ambos colores) a la meta. En esta versión del juego, el tablero dispondrá solo de **2 fichas** para cada color.

2. El orden de juego es **J1** → **J2** → **J2** → **J1** → **J1** → **J2**… Es decir, el primer turno comienza con el Jugador 1, y después van jugando dos turnos consecutivos cada jugador.

3. En cada turno, **el jugador elige un dado** de los disponibles 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 **yinyang** (se describe más adelante)**.** El dado **yinyang** también debe gastarse para que reaparezcan todos.

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.

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 verde 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.

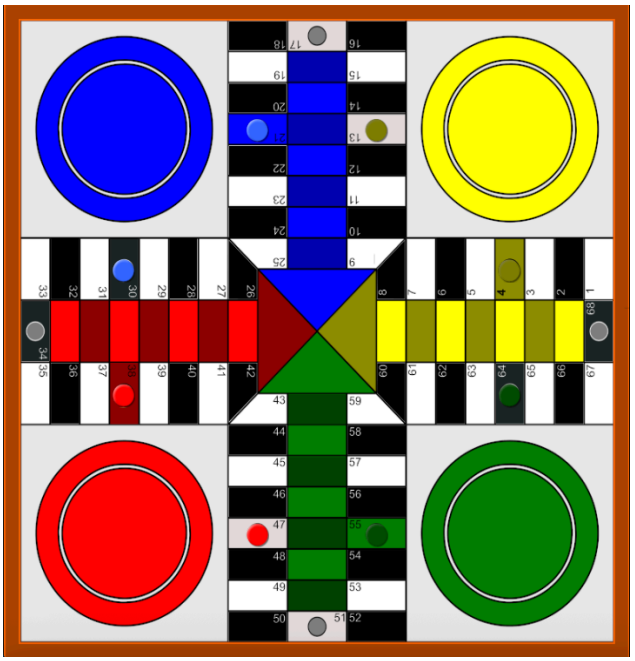


**Figura 2:** Configuración inicial del tablero.

**REGLAS ESPECIALES**

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 cambiado el diseño del tablero y se ha incluido el dado **yinyang**.

El nuevo diseño del tablero consiste en tener casillas de dos tipos (blancas y negras). Por la estructura del tablero, cada ficha aparece en un determinado tipo de casilla. En este diseño del tablero, las fichas solo pueden moverse por el tipo de casillas en el que están (a priori). Esto se traduce en la siguientes reglas con respeto a los movimientos:

1. Los movimientos se realizan solo contando las casillas de un determinado color. Esto es, si una ficha se encuentra en una casilla de tipo "blanco" y elige el dado 3, avanza 3 casillas blancas (sin contar las negras que haya en el camino), y viceversa.
2. El tipo de casilla por el que se puede mover una ficha está determinado, en un principio, por el tipo de casilla que ocupa en la configuración inicial (ver Figura 2) y, en caso de ser comida, por el tipo de casilla de salida de la casa de su color.
3. Aunque las fichas solo se mueven por casillas de su color (blanco o negro), le afectan todas las posibles barreras que haya en las casillas intermedias, tanto blancas como negras.
4. Claramente, para poder comerse una ficha, hay que estar desplazándose por el mismo tipo de casilla para poder llegar a la casilla donde se encuentra la ficha.
5. De igual forma, las casillas de meta de cada uno de los colores también serán de tipo negra o blanca, y no todas las fichas de cada color aparecen en el mismo tipo de casilla que su meta, por lo que tendrán que ser capaces de cambiar de tipo de casilla.

Entonces… ¿Es posible comerse las fichas que se encuentran en el otro tipo de casillas o acceder a algunas metas? ¡Por supuesto que sí! Si no, esto sería demasiado aburrido. Aquí es donde entra en juego el dado **yinyang**. Este dado nos permite cambiar el tipo de casillas por el que se desplaza una ficha concreta. Para ello, avanza a la siguiente casilla (equivalente a avanzar con un 1 en el Parchís normal). Esto es, si está en una casilla negra, la ficha pasa a la siguiente casilla blanca, y viceversa. Una vez terminada esta acción, la ficha se seguirá moviendo con normalidad por su nuevo tipo de casillas. Este dado se puede aplicar tantas veces como se quiera, siempre y cuando esté disponible.

## 4. Objetivos de la práctica

A partir de las consideraciones iniciales, y de la implementación del algoritmo MINIMAX que os proporcionamos, el objetivo de esta práctica es **implementar sucesivas mejoras del algoritmo MINIMAX** de forma que se haga una búsqueda más eficiente manteniendo siempre tanto un **número**

**límite de nodos evaluados** (con la heurística) + **nodos generados** (los hijos de los nodos evaluados) **de 2500000** como un **límite de tiempo de 60s** por movimiento. Si este límite se alcanza, se debe devolver la mejor solución que ha sido capaz de encontrar hasta dicho momento. Si el límite se excede, la partida se dará por perdida automáticamente.

Se valorarán las mejoras implementadas sobre la poda, con el fin de conseguir resultados más prometedores manteniendo (o disminuyendo) el número de nodos generados y evaluaciones de los mismos.

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 **ParCheckers**. Una heurística será apropiada cuando sea capaz de representar fielmente una situación de juego. De esta forma, para una situación de juego mejor que otra, la heurística deberá ser mayor. También es **recomendable** que la heurística sea **simétrica**. Esto es, que lo que es bueno para un jugador, sea proporcionalmente malo para el contrincante. Se anima a diseñar diferentes heurísticas, comprobar el funcionamiento, y analizar el por qué de los resultados.

A continuación se detallan las diferentes versiones del algoritmo de búsqueda propuestas.

## 4.1. Poda Alfa-Beta

La poda alfa-beta es una **optimización** del algoritmo minimax que **evita explorar ramas del árbol que no pueden influir** en la decisión final. Es decir, recorta partes del árbol que sabemos que no serán elegidas. En la poda alfa-beta, usamos dos variables llamadas **alfa (α)** y **beta (β)** para representar los límites de lo que vale la pena seguir evaluando.

Con respecto a la cota alfa:

- Es la **mejor puntuación que el jugador MAX puede garantizar** hasta ese punto en la búsqueda.
- Se **inicializa en -∞** y se va actualizando cuando MAX encuentra una opción mejor.
- Cuanto **mayor sea alfa**, menos opciones quedan para MIN que valgan la pena.

Con respecto a la cota beta:

- Es la **mejor puntuación que el jugador MIN puede garantizar** hasta ese punto.
- Se **inicializa en** +∞ y se actualiza cuando MIN encuentra una opción peor para MAX.
- Cuanto **menor sea beta**, menos opciones quedan para MAX.

Durante la búsqueda si **alfa ≥ beta**, **se detiene** la exploración de esa rama. Esto indica que el jugador actual **ya tiene una mejor opción disponible** por otro camino, por lo que continuar evaluando no cambiará la decisión final.

Estas cotas se **pasan de nodo en nodo** a medida que el algoritmo desciende por el árbol:

- En nodos MAX, se actualiza **alfa**.
- En nodos MIN, se actualiza **beta**.

Esto permite **detectar lo antes posible** cuándo se puede hacer una poda, y así evitar evaluaciones innecesarias.

El algoritmo alfa-beta produce el **mismo resultado** que MINIMAX (si se aplica correctamente), pero con **menos nodos evaluados**. Por tanto, con esta mejora se podrá explorar a más profundidad con el mismo coste de nodos generados + nodos evaluados.

**La implementación de esta mejora será obligatoria pues el resto de mejoras parten de esta versión del algoritmo de búsqueda.**

**Para esta versión básica de la poda, con el límite de 2500000 nodos generados + evaluados, la profundidad máxima recomendada es 7-8. A partir de las mejoras de la poda explicadas a continuación, esta profundidad se puede aumentar (manteniendo el número de nodos) para mejorar la búsqueda de la mejor acción.**

## 4.2. Profundidad dinámica basada en ramificación

El objetivo es aprovechar mejor el límite de nodos generados y evaluados, **aumentando la profundidad de búsqueda cuando el árbol es menos ramificado**. En el algoritmo minimax (con o sin poda alfa-beta), el **número de nodos crece exponencialmente** con la profundidad, según el **factor de ramificación**. Si ese factor es alto, se deben limitar los niveles explorados. Pero cuando la **ramificación es baja**, es **posible y deseable ir más profundo**, porque:

- Se pueden explorar más jugadas futuras relevantes.
- Se mejora la calidad de las decisiones sin sobrepasar el límite de nodos.

Los pasos a seguir para la correcta implementación de estas mejora son:

1. **Medir la ramificación del nodo actual**. Antes de decidir si avanzar más en profundidad, se **generan todos los hijos posibles** del nodo actual. En nuestro caso concreto, la ramificación máxima para un nodo concreto será *dados_disponibles x fichas_disponibles*.
2. **Ajustar la ramificación** según ese número. Por ejemplo:
    - Si la ramificación es muy baja, aumentar la profundidad.
    - Si la ramificación es alta, mantener la profundidad.
3. **Continuar la búsqueda** con esa nueva profundidad.

❖ **Consejo práctico:** para conocer la ramificación a partir de un nodo actual, tienen que generarse todas sus ramificaciones a priori (para poder contarlas). Por tanto, cuidado con no generar los

hijos del nodo actual dos veces. Se recomienda generarlos todos, y usando la misma estructura contarlos y, a posteriori, recorrerlos.

## 4.3. Ordenación de movimientos

El objetivo es **ordenar los movimientos antes de expandirlos**, de modo que **los más prometedores** se examinen **primero**. Esto aumenta la eficacia de la poda alfa-beta y puede **reducir enormemente el número de nodos evaluados.**

La poda alfa-beta depende **en gran parte del orden en que se exploran los nodos**:

- Si los **mejores movimientos se exploran primero**, se podan muchas más ramas.
- Si los **peores se exploran primero**, el algoritmo puede comportarse casi igual que minimax puro (sin poda).

¿Cómo se implementa?

1. **Antes de expandir hijos**, evaluar cada uno **rápidamente** con una función heurística.
2. **Ordenar los movimientos** según esa evaluación (mayor valor si el nodo es MAX, menor si el nodo es MIN).
3. Expandir los nodos en ese orden.

❖ **Consejo práctico:** La diferencia principal se nota en el **primer nivel** de búsqueda:
  ➢ Si en la raíz ya se elige primero una jugada muy buena, eso permite podar mucho más profundamente en las ramas siguientes.
  ➢ No es necesario (ni eficiente) hacer un ordenamiento exhaustivo en todos los niveles: basta con enfocarse en los primeros (especialmente la raíz).

## 4.4. Poda probabilística

El objetivo es reducir aún más el número de nodos evaluados, permitiendo **podas más agresivas** cuando no parece que el resultado actual vaya a superar significativamente el mejor hasta el momento.

En la poda alfa-beta tradicional:

- Se poda si **alfa ≥ beta**, es decir, si se demuestra que la jugada **no puede ser mejor** que lo que ya tenemos.

En la **poda probabilística**, relajamos esa condición:

- Podamos **aunque alfa < beta**, si la diferencia es **pequeña** y poco prometedora.
- Ejemplo: si beta = 20, y alfa llega solo hasta 19.5, podemos asumir que **probablemente no vale la pena seguir buscando**.

Se puede establecer un **umbral de tolerancia (ε)** que defina cuánto más debe mejorar un valor para que sigamos explorando. Por ejemplo:

```
if (beta - alpha < epsilon){

    // Poda anticipada: se asume que no habrá mejora significativa

    return valor_actual;

}
```

Con un valor de *epsilon* prefijado o proporcional a los valores de alfa y beta.

Esta versión de la poda alfa-beta **aumenta la velocidad de búsqueda**, permitiendo una **mayor profundidad** bajo la hipótesis de un número limitado de nodos generados. Sin embargo, puede descartar una jugada ligeramente mejor de forma prematura, ya que no garantiza soluciones óptimas, a diferencia de la poda alfa-beta tradicional. Por eso se llama **poda probabilística**: asume que es **poco probable** que la mejora valga la pena. El riesgo asumido será mayor o menor en función del valor *epsilon* elegido.

## 4.5. Búsqueda de quietud

El objetivo de esta variación es evitar evaluaciones **superficiales o engañosas** que se producen al detener la búsqueda en estados **inestables**, donde están ocurriendo (o por ocurrir) eventos importantes para el juego como **comidas de fichas**, **creación de barreras** o **salida/entrada de casillas seguras**.

Cuando el algoritmo alfa-beta alcanza la **profundidad límite**, normalmente evalúa la posición usando una **heurística estática**. Pero si el estado actual es **muy inestable** (por ejemplo, una ficha está por ser comida), esta evaluación puede **no ser representativa**.

La *Quiescence Search* **extiende selectivamente la búsqueda** en estos casos inestables, para **"dejar que se resuelva el caos"** y así obtener una evaluación más representativa.

Los pasos a seguir para la correcta implementación de estas mejora son:

1. **Al llegar a la profundidad límite**, no se evalúa inmediatamente.

2. Se comprueba si el estado es **"quieto" (quiescente)**, es decir, que no haya movimientos que puedan causar inestabilidad en la partida. Por ejemplo:
   a. No hay comidas inmediatas disponibles.
   No hay jugadas amenazantes claras (como entrar en barrera o moverse a una casilla segura).

3. Si el estado **no es quiescente**, se generan **solo ciertos movimientos tácticos relevantes** (como comidas o barreras) y se sigue buscando *más allá* del límite normal.

4. Cuando finalmente se alcanza un estado "quieto", se aplica la heurística.

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

## 5. Instalación y descripción del simulador

### 5.1. Instalación del simulador

El simulador **ParCheckers** 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, debe seguir los pasos especificados en el GitHub de la asignatura.

## 5.2. Ejecución del simulador

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

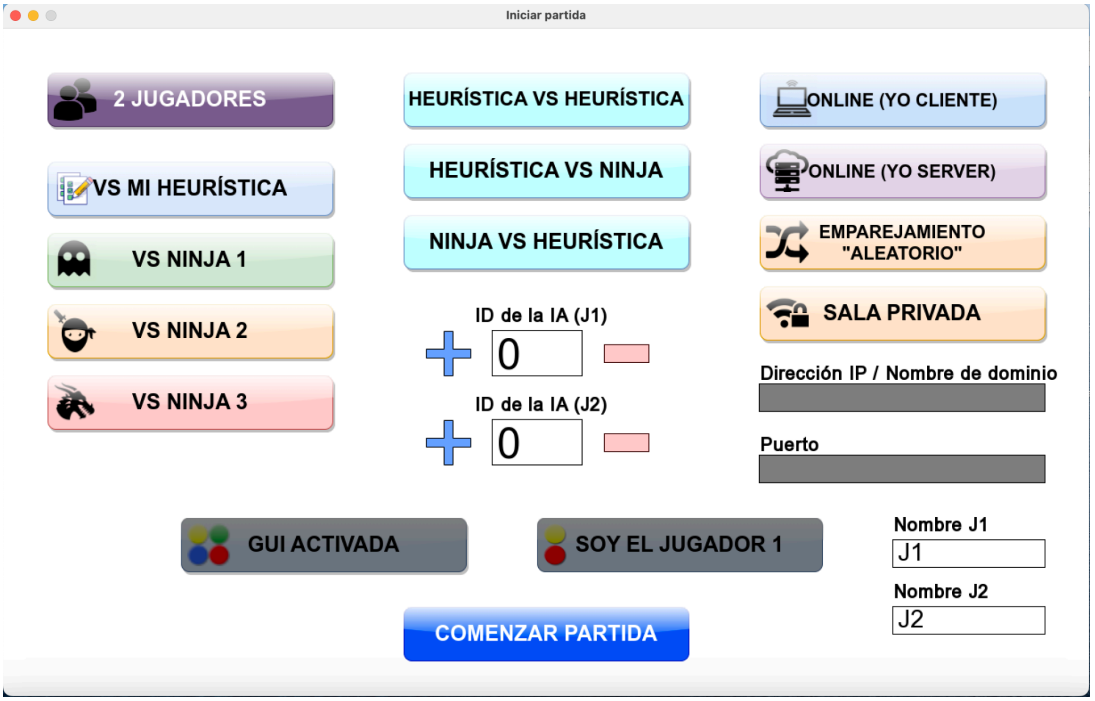



**Figura 3.** Ventana de inicio de juego.

En dicha pantalla de inicio 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 7 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:

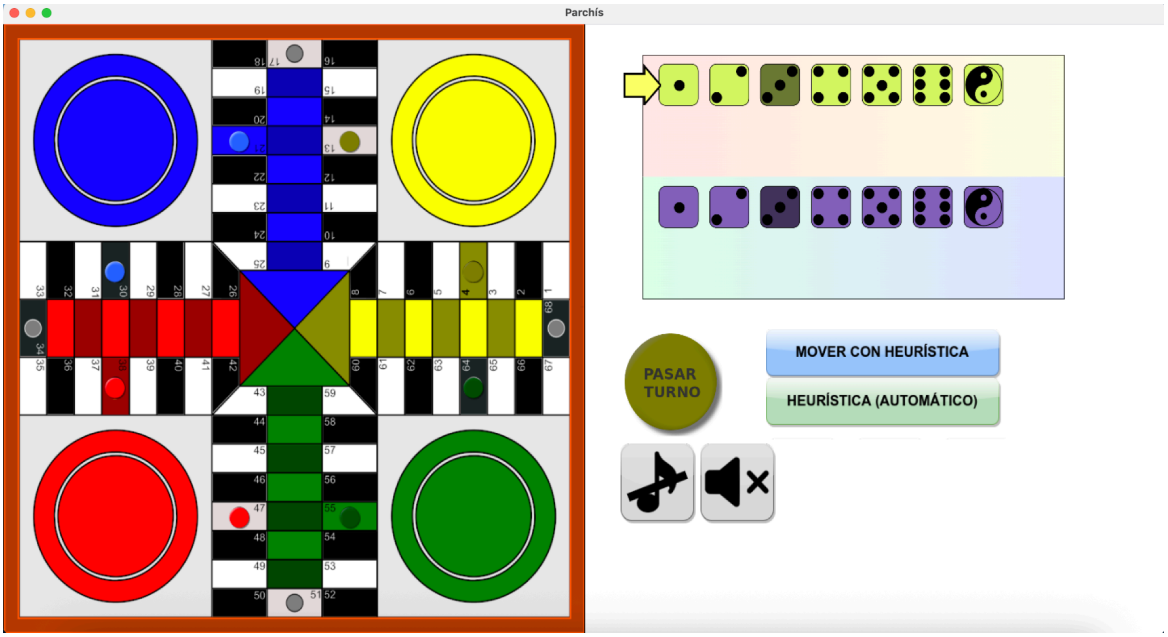



**Figura 4:** Ventana Principal del Juego.

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.

## 5.3. 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 **ParCheckers** con la que, directamente desde línea de comandos, se puede acceder a las distintas opciones que proporciona el juego. La sintaxis es como sigue:

```
./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:

./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 **ParCheckers** 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.

## 6. 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.

Una descripción más detallada de las funciones que os pueden ser de utilidad de dichos archivos se encuentran en el Anexo de la práctica.

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

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

La clase ***AIPlayer*** se utiliza para representar un jugador (es la clase equivalente a los agentes de las clases en *ComportamientosJugador* 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;}
```

Contiene una variable protegida:

- ***id** (int)*: Almacena el identificador del jugador. Esta variable se puede utilizar dentro del método think para seleccionar diferentes búsquedas alfa-beta que se diseñen, o incluso para enfrentar diferentes algoritmos 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 a los algoritmos de búsqueda 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**.

## 6.2. Entendiendo el método think()

El estudiante, al compilar por primera vez el software de la práctica, se encontrará con un método think() que implementa un jugador aleatorio. Esta implementación inicial ilustra de forma práctica cómo se puede manipular la información que nos proporciona el tablero a través de la clase **Parchis**. Dicha implementación inicial se explica detalladamente en el tutorial de la práctica.

Al igual que en la práctica anterior, el método think() define el comportamiento de nuestro agente (el jugador de la partida) ante un entorno concreto (la situación actual de la partida). El entorno concreto viene definido por la situación actual de la partida, un elemento de la clase Parchis, y un entero que define qué jugador en la partida es el que está pensando (0 si es el jugador 1 y 1 si es el jugador 2). Ambos elementos forman parte de la clase AIPlayer, y vienen definidos, respectivamente, como **Parchis**

***actual** e **int jugador**. Desde el método think() podremos acceder directamente a cualquiera de estos atributos ya definidos escribiendo los nombres de estas variables.

Por otra parte, una vez el agente efectúa su comportamiento en el método think(), debe tomar una decisión, que será la acción que se vaya a realizar. En este caso, el próximo movimiento a efectuar en la partida. Un movimiento viene definido por 3 variables:

- El valor del dado con el que se va a mover (1, 2, 4, 5, 6 o el dado **yinyang**, que viene definido por la constante **yinyang** que podemos usar directamente).
- El color de la ficha que se va a mover (se representa mediante un enumerado denominado **color**, que contiene los valores **blue, red, green, yellow** y **none**.
- El número de ficha que se mueve dentro del color seleccionado (se representa mediante un entero: la ficha 0 o la ficha 1 del color seleccionado).

Al tratarse de una acción compuesta, la acción que se calcula en el método **think** no se devuelve a través de un return, si no a través de sus 3 parámetros, que se pasan por referencia:

```
virtual void think(color& c_piece, int& id_piece, int& dice) const;
```

Dentro del think() deberemos actualizar estas tres variables para que una vez se salga del método tengan asignada la acción exacta que queremos realizar para la partida.

## 6.3. Un solo think() pero múltiples comportamientos

En la versión inicial que se proporciona del método think se ofrece un código comentado en el que se propone una forma de plantear las distintas implementaciones que se hagan a lo largo de la práctica:

```
  // Si quiero poder manejar varios comportamientos, puedo usar la variable id del agente
para usar una u otra.
  switch (id)
  {
  case 0:
     // Mi implementación base de la poda con ValoracionTest
     valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece,
dice, alpha, beta, &valoracionTest);
     break;
  case 1:
     // Mi implementación definitiva con la que gano a todos los ninjas.
     valor = Poda_Final2DefinitivaAhoraSi(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece,
id_piece, dice, alpha, beta, &miValoracion3);
     break;
  case 2:
```

```
      // Las distintas pruebas que he realizado (primera prueba)
      valor = Poda_AlfaBeta_Mejorada(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece,
id_piece, dice, alpha, beta, &miValoracion1);
      break;
  case 3:
      // Las distintas pruebas que he realizado (segunda prueba)
      valor = Poda_AlfaBeta_SegundaMejora(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece,
id_piece, dice, alpha, beta, &miValoracion1);
      break;
   // ...
```

El switch anterior tiene en su condición la variable **id**. Esta variable también viene ya definida en la clase del jugador, y toma su valor del parámetro de ID del jugador que pasamos al iniciar la partida, bien a través de la caja de texto en la interfaz de selección de partida, o bien a través de los argumentos de la línea de comandos (--p{i} <Tipo de jugador> **<ID del jugador>** <Nombre del jugador>). De esta forma, en función del ID que se seleccione al comenzar la partida podremos jugar con uno de nuestros comportamientos o con otro, o incluso enfrentar a varios de nuestros comportamientos entre sí o contra los ninjas.

Es recomendable guardar los distintos comportamientos que se hayan elaborado de esta forma, hayan sido o no exitosos, ya que también se valorará el trabajo realizado, y no únicamente los resultados de las partidas. Reserva el ID 0 para la poda con ValoracionTest, que utilizaremos para verificar el correcto funcionamiento del algoritmo, y el ID 1 para el comportamiento definitivo con el que ganes a más ninjas. A partir de ahí, en cada ID puedes asignar un comportamiento diferente en función de las diferentes mejoras de la poda y otras heurísticas que hayas ido probando.

## 6.4. Cómo implementar un comportamiento

En el ejemplo anterior, también podemos ver cómo en cada caso se implementa un comportamiento diferente. Como comportamiento entenderemos a la combinación de un algoritmo de búsqueda (que en esta práctica serán los algoritmos minimax, la poda alfa-beta y las variantes de esta que se implementen) y la heurística o heurísticas que se usen para evaluar los nodos.

Sobre el algoritmo de búsqueda, se propone una cabecera para implementar la poda alfa-beta, que se basa en la cabecera proporcionada para el minimax que se puede consultar en el tutorial:

```
float Poda_AlfaBeta(const Parchis &actual, int jugador, int profundidad, int profundidad_max,
color &c_piece, int &id_piece, int &dice, float alpha, float beta, Heuristic *heuristic);
```

Es importante comprender bien que la poda alfa-beta es un algoritmo de búsqueda en profundidad y su implementación es recursiva. En cada llamada a la poda partimos de una nueva situación de partida (a la que se ha llegado desde la situación raíz de la primera llamada de la poda), en un nuevo nivel de profundidad y con unos nuevos parámetros alfa y beta que nos vienen de las llamadas anteriores. A la salida de esta función debemos devolver la acción que se debe realizar en el siguiente turno de la partida. Analicemos uno a uno los argumentos de esta cabecera:

- **const Parchis &actual**: es una referencia al tablero de la partida desde el que se parte para la búsqueda. En la primera llamada le debemos pasar el tablero que recibe el jugador actual y sobre el que tiene que mover. En las llamadas recursivas se irá llamando con los nodos hijos de ese tablero.
- **int jugador**: se corresponde con el jugador que invoca a la poda, el mismo que está ejecutando el método think. Es muy útil para determinar si un nodo es min o max. En todas las llamadas a la poda debe ser constante.
- **int profundidad:** la profundidad en la que nos encontramos en cada llamada a la poda. Inicialmente a 0 en la llamada original, se debe incrementar a cada llamada.
- **int profundidad_max:** la profundidad límite a la que queramos llegar, en el caso de la implementación básica con profundidad fija. La podemos comparar con **profundidad** para saber si debemos terminar de bajar y evaluar el nodo actual.
- **color &c_piece, int &id_piece, int &dice**: el movimiento (color y número de ficha + valor del dado) que se debe realizar. Se pasan por referencia y se deben actualizar cuando estamos en el nodo raíz (a profundidad 0) cada vez que encontremos un nodo mejor, para actualizar el movimiento que debe hacer el jugador. Al igual que en el método think(), cuando finaliza la poda en estos valores tendremos almacenada la acción deseada.
- **float alpha, float beta**: las cotas alfa y beta del algoritmo. Inicialmente deben ser menos infinito y más infinito, respectivamente. En los nodos max se actualizará alfa y en los min beta cada vez que se encuentre un nodo con mejor valoración. En las nuevas llamadas a la poda se irán enviando los alfas y betas actualizados. En la vuelta hacia arriba (al hacer el return) se irán actualizando los alfa en los nodos max y los beta en los nodos min.
- **Heuristic *heuristic**: un puntero a la heurística que queramos utilizar para valorar los nodos. Una heurística será una subclase de **Heuristic** sobre la que definiremos la función evaluadora. Veremos los detalles en la siguiente subsección.

En el tutorial se puede ver cómo se utilizan estos parámetros en un caso muy similar: el algoritmo minimax. La adaptación a la poda a partir de ese caso es bastante inmediata.

**IMPORTANTE: Estos parámetros se proponen para la implementación básica de la poda. A la hora de hacer mejoras de la poda, si consideras que necesitan parámetros adicionales o diferentes puedes añadir los que consideres oportunos.**

## 6.5. Valorando los nodos con la clase Heuristic

Como hemos comentado en el apartado anterior, además del método de búsqueda utilizado, será importante disponer de una heurística que valore los nodos adecuadamente. La forma de valorar los nodos debe hacerse mediante la clase **Heuristic**.

La clase **Heuristic** es una clase abstracta que tiene un método, **float getHeuristic(const Parchis &estado, int jugador) const**, que no está implementado. Cada vez que queramos elaborar una heurística nueva, solo tendremos que definir una subclase de **Heuristic** e implementar este método **getHeuristic**. Luego esta subclase se la podemos pasar como puntero a nuestra función de búsqueda, como se explica en la sección anterior. El polimorfismo hace que podamos usar heurísticas diferentes con una misma función de búsqueda, facilitando la creación de comportamientos diferentes.

En la implementación inicial se proporciona una heurística de prueba, **ValoracionTest**, con un comportamiento muy simple: valora positivamente fichas en casillas seguras o en la meta de mi jugador y negativamente las del oponente. Podemos usarla como referencia para ver cómo se define una heurística cuando queramos definir una nueva. Como podemos ver en **AIPlayer.h**, solo tenemos que crear una clase **ValoracionTest** que herede de **Heuristic** y que redefina el método **getHeuristic:**

```
class ValoracionTest: public Heuristic{
  public:
     virtual float getHeuristic(const Parchis &estado, int jugador) const;
};
```

Después, definimos en nuestro **AIPlayer.cpp** el código que deseemos para evaluar con esta heurística:

```
float ValoracionTest::getHeuristic(const Parchis& estado, int jugador) const{
  // Implementación de la heurística (la de ValoracionTest la puedes ver en el código de la práctica)
}
```

Y con la clase ya definida, podemos pasársela a nuestra función de búsqueda en la posición del parámetro de tipo **Heuristic *** como vimos en la sección previa:

```
ValoracionTest valoracionTest; // La valoración que voy a usar
valor = Poda_AlfaBeta(*actual, jugador, 0, PROFUNDIDAD_ALFABETA, c_piece, id_piece, dice,
alpha, beta, &valoracionTest); // Cómo pasársela al método de poda.
```

Y cada vez que queramos evaluar un nodo dentro del algoritmo de búsqueda, lo podemos hacer con el método **float evaluate(const Parchis & estado, int jugador)** de nuestra heurística **(IMPORTANTE:**

**debemos usar el método evaluate para valorar las heurísticas dentro del algoritmo de búsqueda, a pesar de que el método que hemos redefinido fuera el getHeuristic)**:

```
float Poda_AlfaBeta(const Parchis &actual, int jugador, int profundidad, int profundidad_max,
color &c_piece, int &id_piece, int &dice, float alpha, float beta, Heuristic *heuristic)
{
 // ... MIS COSAS DE PODAR ...
 valor = heuristic->evaluate(nodo, jugador); // Evaluando algún nodo dentro del código
 // ... MÁS COSAS DE PODAR ...
}
```

De nuevo, en la implementación del algoritmo minimax que se ofrece en el tutorial, se puede consultar cómo usar las heurísticas (en ese caso, de nuevo ValoracionTest) con más detalle.

**CONSEJO: como hemos mencionado antes, ValoracionTest es una heurística de prueba que se os da para poder comprobar el funcionamiento de minimax y poda con una referencia, pero es una heurística mala. A la hora de pensar en una heurística buena debemos tener en cuenta que, a nivel general, en el Parchis, cuanto más cerca de la meta, mejor. Y sobre esta base, ya podemos ir pensando en otros factores relevantes como casillas seguras, barreras o ventajas posicionales. La clase Parchis tiene dos funciones para medir distancias en el tablero (distanceToGoal y distanceBoxToBox) que pueden ser útiles para la base de la heurística. Estas funciones están explicadas con más detalle en el anexo y en la descripción del código.**

**MUY IMPORTANTE: Aunque se permite que la implementación de la poda sea flexible en cuanto a diseño y parámetros, en cuanto a la/s heurística/s, es obligatorio implementarla/s como subclase/s de la clase Heuristic. Cualquier otro tipo de implementación de las heurísticas que no se haga mediante este método no se evaluará.**

## 6.6. Otras clases relevantes

Debemos destacar algunas clases más que serán relevantes para la construcción de los algoritmos de búsqueda que se piden para la práctica:

- El iterador de **ParchisBros**: dado un tablero concreto de juego de la clase **Parchis**, el método **getChildren** de esta clase nos devolverá un iterador que permite recorrer todos los tableros (hermanos) a los que se puede llegar desde el original. Con este iterador recorrer todos los hijos se convierte en un bucle for clásico, donde para pasar de un hermano (bro) a otro solo tendremos que hacer un **++iterador**.
- El alter-ego del iterador es la lista completa de posiciones hijas (**ParchisSis**) de una posición concreta. En algunas variantes que se pueden implementar puede ser interesante tener la lista completa de situaciones hijas para compararlas y tomar una decisión teniendo todas. Aunque si

no es el caso, suele resultar más sencillo iterar hijo a hijo con **ParchisBros** y no generar nodos innecesarios.

- Por último, puesto que el juego controla en cada turno el total de nodos que se generan y se evalúan, podremos (y deberíamos) controlar desde nuestro algoritmo de búsqueda cuándo llegamos a ese límite, y cortar el algoritmo al llegar (si no, perderemos la partida automáticamente). La clase **NodeCounter** se encarga de ello. Podemos acceder estáticamente a ella desde cualquier parte de nuestro código sin necesidad de instanciarla, y consultar cuántos nodos llevamos generados, o si hemos alcanzado el límite. Para este último caso, bastaría con poner la instrucción en nuestro código `NodeCounter::isLimitReached().`

En el tutorial se describen todos estos casos con más detalle y se utilizan en las implementaciones que se ofrecen. Se recomienda estudiar bien dicho código.

## 6.7. 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, es importante destacar que debido a la dinámica de turnos de juego en la que a partir del segundo turno los jugadores juegan dos turnos seguidos, **el sucesor de un nodo MÁX no siempre será un nodo MIN (ni viceversa)**. Además, 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. 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.

- Las **clases descritas en el anexo de la práctica** 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 **ParCheckers**. 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.

## 7. Evaluación y entrega de la práctica

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

- Se entregará una memoria de prácticas al finalizar las tareas a realizar. Es importante que en la memoria se refleje todas las versiones de la poda alfa-beta y heurísticas implementadas, así como los éxitos o fracasos obtenidos (y los motivos por los que se piensa que se han obtenido). La memoria contendrá una tabla resumen de resultados de cada una de las versiones de la poda y de la heurística contra los ninjas disponibles. **Queda terminantemente prohibido introducir fragmentos de código en la memoria.**
- 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 2.** Se evaluará la claridad de las explicaciones así como la originalidad de las heurísticas contempladas.
  - La variedad de las versiones de la poda alfa-beta implementadas se evaluará de **0 a 2.** Siendo obligatorio haber implementado al menos 3 de las 5 versiones propuestas (incluyendo la versión clásica de poda alfa-beta que es obligatoria). Además, se evaluará positivamente con hasta **1 punto extra** el proponer mejoras no contempladas en el documento. Esta puntuación dependerá de la originalidad, la factibilidad y el éxito contra los ninjas.
  - 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 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 1 de junio antes de las 23:00 horas.**

## 7.1. 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 la poda alfa-beta y sus sucesivas mejoras, así como diferentes heurísticas diseñadas para el juego de **ParCheckers** 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.**

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

/ UGR / decsai

El fichero ZIP debe contener una **memoria corta de prácticas (¡sin código!)** en formato PDF que, como mínimo, contenga los siguientes apartados:

1. Listado y breve descripción de todas las mejoras de la poda alfa-beta desarrolladas.
2. Listado y breve descripción de las heurísticas propuestas.
3. Tabla comparativa de todas las podas y heurísticas propuestas contra los ninjas. Además, se recomienda comparar entre sí soluciones propias. En la tabla se debe especificar con que ID está asociada cada propuesta (combinación de la versión poda + heurística).
4. Análisis de los resultados obtenidos en la tabla comparativa. Ventajas e inconvenientes de las mejoras de la poda y las heurísticas contempladas. Se valorará positivamente que tanto las mejoras de la poda como las heurísticas se hayan hecho de forma progresiva intentando solucionar las debilidades de versiones anteriores.
5. Opcionalmente, se puede incluir una reflexión personal acerca de las dificultades que se han encontrado a lo largo del desarrollo de los algoritmos.

**IMPORTANTE: Para la competición contra los ninjas se corregirá con id=1, así que asociad la mejor heurística y versión de la poda a este id y valoracionTest con la poda alfa-beta básica 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** 🙂