INTELIGENCIA ARTIFICIAL

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

Tutorial Práctica 3

Búsqueda con Adversario (Juegos)

El Parcheckers

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

1. Introducción

Este documento se proporciona como punto de partida para que los estudiantes empiecen a familiarizarse con el software de la práctica 3. Al realizar este tutorial se espera que los alumnos y alumnas comprendan cómo pueden programar comportamientos inteligentes para el jugador de **ParCheckers** usando el simulador, e inicien el descubrimiento de algunas de las muchas funciones que se proporcionan para consultar los aspectos del juego en la clase Parchis. El tutorial finaliza con el **algoritmo minimax implementado.**

Se recomienda, antes de empezar el tutorial, leer detenidamente el guion de la práctica, destacando principalmente el objetivo de la práctica, las reglas del juego y la introducción al software.

2. El punto de partida

Como se comenta en la sección 6.2 del guion, el simulador trae implementado por defecto el comportamiento de un agente que elige de forma aleatoria tanto el valor del dado como la ficha que quiere mover. El agente se implementa en la clase **AIPlayer**, el cual tendrá acceso en todo momento, a través de variables de instancia, al estado actual de la partida (la variable **actual**), y a su id de jugador (la variable **jugador**), que valdrá 0 para el primer jugador (amarillo y verde) y 1 para el segundo (azul y rojo).

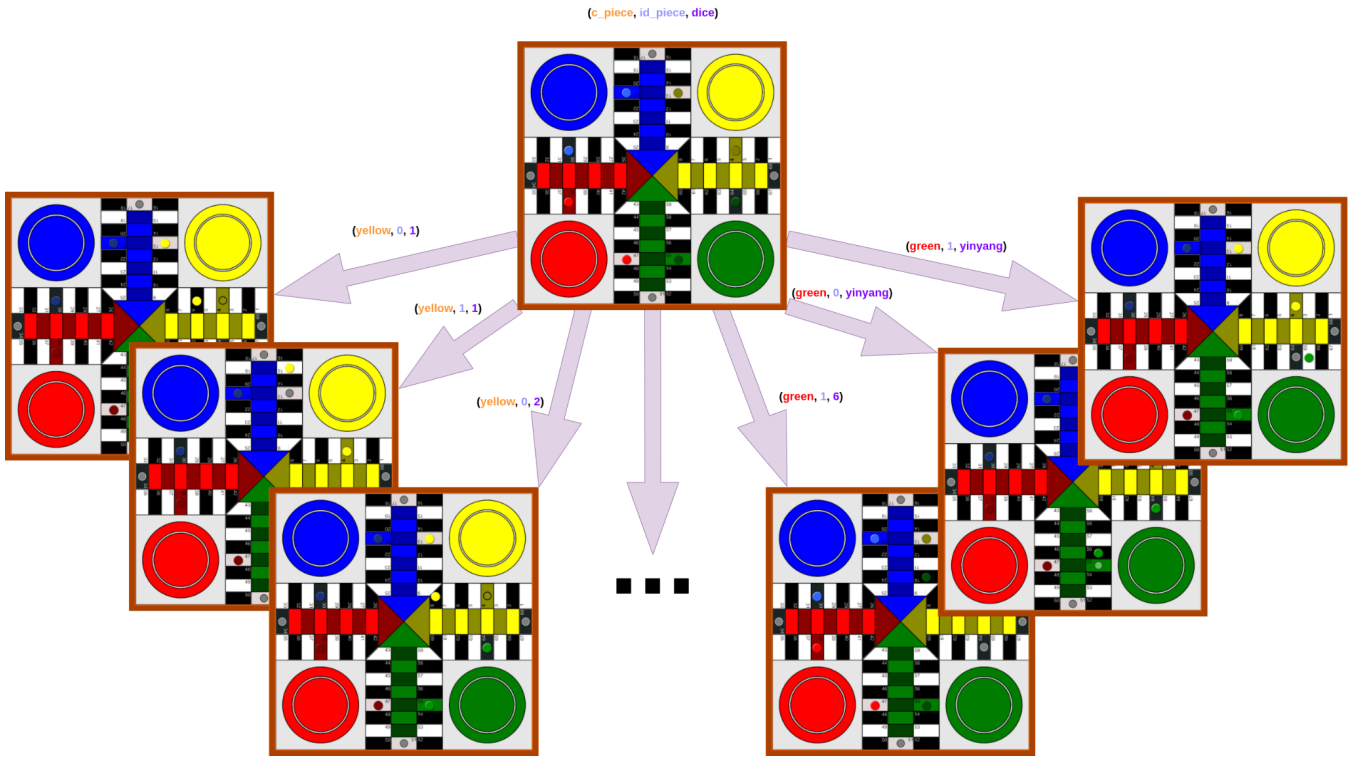
```
class Player{

   protected:

       // ...

       // Son heredados en la clase AIPlayer

       Parchis *actual;

       int jugador;

       // ...

};
```

El método **think**, situado en la clase **AIPlayer**, es el encargado de, como en la práctica anterior, a partir de la información de la que dispone del juego el agente en el turno que le toque, determinar la acción a realizar. En este caso, la acción a realizar viene determinada por 3 parámetros:

- El **color de la ficha** que se mueve. En esta versión del Parchís, en su turno, **cada jugador podrá elegir mover una ficha de cualquiera de sus dos colores**. Este color se representa como un valor del enumerado **color** (ver el fichero *"Attributes.h"*), que puede tomar los valores **yellow**, **blue**, **red**, **green** y **none**.

- El **id de la ficha** que se va a mover. Las fichas de cada color van numeradas del 0 al 1. Con cada uno de los números podremos acceder a cada una de ellas. **En ocasiones no habrá ninguna ficha** que se pueda mover. En estos casos, hay un valor especial de id, definido con la macro **SKIP_TURN**, que permitirá pasar turno sin mover ninguna ficha.
- El **valor del dado** con el que se mueven. Este valor, en general, será un **número entero entre 1 y 6** (sin el 3), aunque en determinados turnos podrá tomar también los valores **10 y 20** (cuando una ficha entra en la meta o cuando una ficha se come a otra). Para **el dado especial** usamos el valor predefinido ***yinyang.***

Nos referiremos a estos tres parámetros en el código, respectivamente, como **c_piece**, **id_piece**, y **dice**. El método **think** recibirá estos tres parámetros **por referencia**. Estos parámetros deberán actualizarse dentro del método al valor deseado del movimiento, para que una vez termine la ejecución del think, el movimiento elegido se lleve a cabo. La siguiente figura muestra un ejemplo de los efectos que tendrían en el tablero determinadas asignaciones de estos parámetros en el método think.



Volviendo al método think inicial, para conseguir que el agente realice un movimiento aleatorio primero comprobamos cuáles son los movimientos disponibles o válidos para nuestro jugador. En primer lugar, obtenemos nuestro número de jugador con **getCurrentPlayerId()**. Una vez conocido este valor, podemos obtener los dados disponibles de nuestro jugador con **getAvailableNormalDices(int)**. Nos devuelve un vector de enteros con los valores de los dados que podemos usar, y seleccionaremos un

elemento al azar, que será el valor del dado con el que nos moveremos. Una vez fijado el dado, miramos qué fichas podemos mover con ese valor del dado, con la función **getAvailablePieces(int, int)** (le pasamos nuestro número de jugador y el valor del dado). De nuevo, seleccionamos un id de ficha al azar, y si no pudiéramos mover ficha seleccionaríamos el id de ficha SKIP_TURN comentado anteriormente. Tras asignar las tres variables c_piece, id_piece y dice, a la salida del método think el juego realizará el movimiento aleatorio que haya salido:

```
void AIPlayer::think(color & c_piece, int & id_piece, int & dice) const{
   // El número de mi jugador actual.
   int player = actual->getCurrentPlayerId();

   // 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(player);
   // Elijo un dado de forma aleatoria.
   dice = current_dices[rand() % current_dices.size()];

   // Se obtiene el vector de fichas que se pueden mover para el dado elegido
   current_pieces = actual->getAvailablePieces(player, 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]);  // get<i>(tuple<...>) me devuelve el i-ésimo
       c_piece = get<0>(current_pieces[random_id]);   // elemento de la tupla
   }
   else
   {
       // Si no tengo fichas para el dado elegido, pasa turno (la macro SKIP_TURN me permite no mover).
```

```
        id_piece = SKIP_TURN;
        c_piece = actual->getCurrentColor(); // Le tengo que indicar mi color actual al pasar turno.
    }
}
```

3. Diseñando comportamientos más inteligentes

En este apartado vamos a ver cómo implementar algunos comportamientos más inteligentes para nuestro agente, con los que ir aprendiendo, además, cómo trabajar con el agente y con los distintos métodos de consulta con los que obtener la información del tablero.

En primer lugar, queremos volver a destacar que el software permite tener implementados simultáneamente a varios agentes, y es posible incluso enfrentar a dichos agentes entre sí. Para ello, la clase **AIPlayer** contiene una variable de instancia llamada **id**. Este número entero puede usarse para identificar a cada uno de los distintos comportamientos que se vayan elaborando, y a la hora de ejecutar el simulador, tanto desde la interfaz gráfica como desde la línea de comandos, es posible elegir con qué **id** de jugador queremos enfrentarnos.

Vamos a hacer uso de esta característica del software para ir probando distintos comportamientos y ver cómo va mejorando a nuestro agente. Para ello, en **AIPlayer.h** vamos a declarar varios métodos que iremos definiendo en las siguientes secciones:

```
        /**
         * @brief Función que se encarga de decidir el mejor movimiento posible a
         * partir del estado actual del tablero. Asigna a las variables pasadas por
         * referencia el valor de color de ficha, id de ficha y dado del mejor movimiento.
         *
         * @param c_piece Color de la ficha
         * @param id_piece Id de la ficha
         * @param dice Número de dado
         */
        virtual void think(color & c_piece,  int & id_piece, int & dice) const;
        void thinkAleatorio(color &c_piece, int &id_piece, int &dice) const;
        void thinkFichaMasAdelantada(color &c_piece, int &id_piece, int &dice) const;
        void thinkMejorOpcion(color &c_piece, int &id_piece, int &dice) const;
```

En la primera función declarada, **thinkAleatorio**, pondremos el código actual que realiza el movimiento aleatorio que hay ahora mismo dentro de **think**. Y ahora, dentro del método think, en función del **id** del jugador que se haya escogido, llamaremos a cada una de estas subfunciones, que asignarán a las variables *c_piece*, *id_piece* y *dice* movimientos diferentes, porque cada una elaborará un comportamiento diferente:

```
void AIPlayer::think(color & c_piece,  int & id_piece, int & dice) const{
    switch(id){
        case 0:
            thinkAleatorio(c_piece, id_piece, dice);
            break;
        case 1:
            thinkFichaMasAdelantada(c_piece, id_piece, dice);
            break;
        case 2:
            thinkMejorOpcion(c_piece, id_piece, dice);
            break;
    }
}
```

En los siguientes apartados implementaremos comportamientos para las funciones **thinkAleatorioMasInteligente, thinkFichaMasAdelantada** y **thinkMejorOpcion**. Pero, antes de continuar, revisa que todo funciona correctamente y que puedes compilar y seguir enfrentádote con el jugador aleatorio. Para ello, deja comentadas en el método anterior todas las llamadas a **thinkXXXXX**, salvo en el **case 0**. Las iremos descomentando conforme las implementemos en los siguientes apartados. Y comprueba que sigues pudiendo enfrentarte al jugador aleatorio inicial. Para ello, nos enfrentamos a la heurística siendo nosotros (por ejemplo) el Jugador 1. Podemos hacerlo a través de la interfaz, o a través de terminal con el comando:

./ParchisGame --p1 GUI 0 “Yo” --p2 AI 0 “Random”

3.1. Eligiendo qué quiero mover

En este apartado vamos a cambiar el comportamiento aleatorio inicial y a tratar de elegir la ficha que queremos mover de forma más inteligente. Lo que haremos será mover siempre la ficha que tengamos más adelantada, con el objetivo de llevarla cuanto antes a la meta. La elección del dado la seguiremos manteniendo aleatoria.

Las posiciones de las fichas y su distancia entre ellas o la distancia a la meta pueden resultar de gran interés a la hora de valorar determinadas circunstancias del juego. Para ello, la clase **Parchis** dispone de funciones como **distanceBoxtoBox** para medir la distancia entre dos casillas o **distanceToGoal** para medir la distancia a la meta. En la documentación se pueden encontrar más detalles sobre cómo usarlas.

En el caso que estamos tratando ahora, como queremos mover siempre la ficha más adelantada, recorreremos todas las fichas de mis colores y miraremos cuál tiene menor distancia a la meta. Finalmente seleccionaremos para mover la ficha con la menor distancia obtenida, o pasaremos turno si no hubiera ninguna ficha. La implementación del comportamiento **thinkFichaMasAdelantada** quedaría como se muestra a continuación:

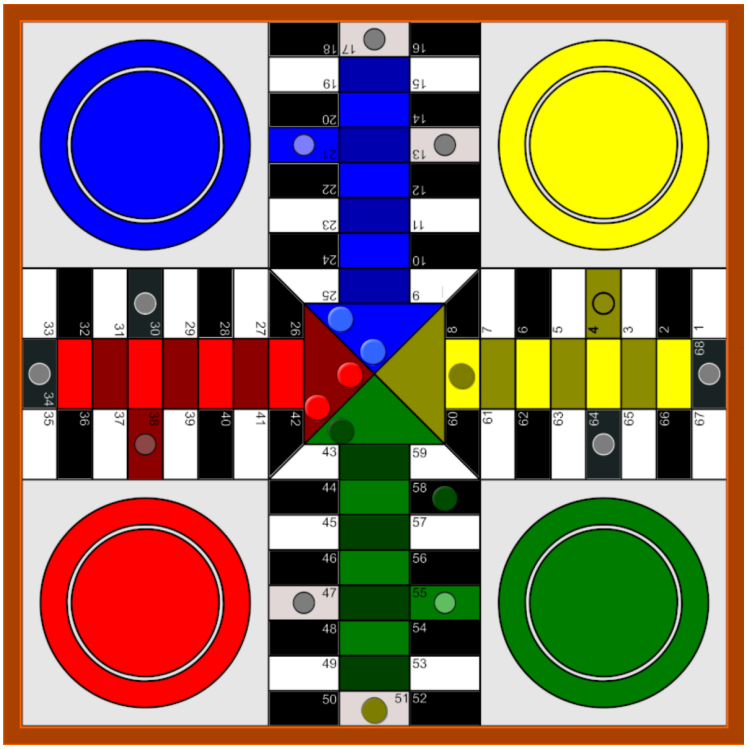
```
void AIPlayer::thinkFichaMasAdelantada(color &c_piece, int &id_piece, int &dice) const
{
   // Elijo el dado haciendo lo mismo que el jugador aleatorio.
   thinkAleatorio(c_piece, id_piece, dice);
   // Tras llamar a esta función, ya tengo en dice el número de dado que quiero usar.
   // Ahora, en vez de mover una ficha al azar, voy a mover (o a aplicar
   // el dado especial a) la que esté más adelantada
   // (equivalentemente, la más cercana a la meta).

   int player = actual->getCurrentPlayerId();
   vector<tuple<color, int>> current_pieces = actual->getAvailablePieces(player, dice);


   int id_ficha_mas_adelantada = -1;
   color col_ficha_mas_adelantada = none;
   int min_distancia_meta = 9999;
```

```
    for (int i = 0; i < current_pieces.size(); i++)
    {
        // distanceToGoal(color, id) devuelve la distancia a la meta de la ficha [id] del color que le indique.
        color col = get<0>(current_pieces[i]);
        int id = get<1>(current_pieces[i]);
        int distancia_meta = actual->distanceToGoal(col, id);
        if (distancia_meta < min_distancia_meta)
        {
            min_distancia_meta = distancia_meta;
            id_ficha_mas_adelantada = id;
            col_ficha_mas_adelantada = col;
        }
    }

    // Si no he encontrado ninguna ficha, paso turno.
    if (id_ficha_mas_adelantada == -1)
    {
        id_piece = SKIP_TURN;
        c_piece = actual->getCurrentColor(); // Le tengo que indicar mi color actual al pasar turno.
    }
    // En caso contrario, moveré la ficha más adelantada.
    else
    {
        id_piece = id_ficha_mas_adelantada;
        c_piece = col_ficha_mas_adelantada;
    }
}
```

Ahora, probamos a enfrentar al nuevo jugador elaborado (de **id=1**) (descomenta el case asociado al 1) con el jugador aleatorio del código original (de **id=0**). Para ello, podemos usar el siguiente comando:

./ParchisGame --p1 AI 0 "Random" --p2 AI 1 "Ya hace cosas"

El resultado que obtenemos al final de la partida es el siguiente:



Podemos comprobar que el jugador que mueve la ficha más adelantada (jugador 2) gana de forma contundente al jugador aleatorio.

3.2. Buscando entre los hijos: el iterador de ParchisBros

A la hora de realizar la práctica será necesario implementar diferentes versiones de algoritmos de búsqueda en árboles de juego. Para aplicar el algoritmo de búsqueda, será necesario a su vez poder generar el árbol del juego a partir de cualquier posible situación de partida. La clase **Parchis** dispone para ello del método **getChildren**. Este método genera una estructura iterable llamada **ParchisBros**, que contiene la información necesaria para poder recorrer todos los hijos (tableros generados a partir de cada posible movimiento) de un estado de juego concreto. El iterador **ParchisBros::Iterator** permite ir pasando uno a uno por estos hijos, y analizar la situación de cada hijo con los distintos métodos de consulta de la clase **Parchis**. Los hijos se recorren de mayor a menor valor de dado (*¿por qué puede ser interesante esto?*), y dentro de cada valor de dado las fichas se recorrerán siempre en orden según su id (del 0 al 1), visitando primero el primer color del jugador (amarillo/verde) y luego el segundo (rojo/azul).

En este apartado vamos a continuar mejorando a nuestro agente y aprovecharemos para aprender a usar el iterador de **ParchisBros**. Lo que haremos será mejorar el comportamiento del agente anterior de la siguiente forma: si detectamos que, para el siguiente turno, alguno de los movimientos que hagamos nos lleva a comernos una ficha, a colocar una de nuestras fichas en la meta o a ganar la partida, nos quedaremos con ese movimiento inmediatamente. En caso contrario elegiremos el movimiento llamando a la función de la sección anterior.

Para ello, es importante volver a destacar que la clase **Parchis** dispone de métodos para consultar casi cualquier cosa que se nos ocurra sobre el estado actual de la partida. En particular, para el caso planteado en este apartado disponemos de las funciones **isEatingMove(), isGoalMove(), gameOver() y getWinner()** que nos permiten obtener, respectivamente:

- Si en el último movimiento se ha comido alguna ficha.
- Si en el último movimiento alguna ficha ha entrado a la meta.
- Si la partida ha terminado.
- El ganador de la partida (0 para el J1 y 1 para el J2), en caso de que la partida haya terminado.

La estrategia que vamos a seguir es la siguiente. Con el iterador recorreremos todos los hijos del tablero actual y en cuanto nos encontremos con un hijo que cumpla alguna de las condiciones indicadas nos quedaremos con el correspondiente movimiento. Es importante primero conocer bien cómo funciona el iterador:

- En todo momento, un iterador de **ParchisBros** inicializado permitirá acceder al tablero hijo sobre el que se está iterando en ese momento con el operador de indirección (*). Además, el iterador posee también 3 métodos para obtener cuál es el movimiento (color + id de ficha + dado) que me ha llevado a obtener dicho tablero hijo: **getMovedColor()**, **getMovedPieceId()** y **getMovedDiceValue()**.

- Para obtener un nuevo iterador que comience a iterar sobre el primer hijo de la colección, puedo usar el método **begin()** de la estructura **ParchisBros** obtenida tras llamar a **getChildren().**

- Para saber cuándo he terminado de iterar, el método **end()** de **ParchisBros** junto con el operador de comparación del iterador me permitirán averiguar si he llegado al final.
- Para pasar de un tablero hijo al siguiente tablero hijo, simplemente tengo que llamar al operador ++ del iterador.

Tras haber entendido cómo funciona la iteración sobre los hijos de un estado de juego concreto, ya estamos en condiciones de programar el comportamiento propuesto en este apartado. Lo que tenemos que hacer simplemente es:

- Recorrer todos los hijos.
- Comprobar si en algún hijo se cumple alguna de las condiciones indicadas (comida, meta o victoria).
- Si se cumple, me quedo con la acción que me lleva a ese hijo.
- Si no se cumple en ningún hijo, asigno el movimiento con la función **thinkFichaMasAdelantada**.

La implementación del comportamiento **thinkMejorOpcion** quedaría como se muestra a continuación:

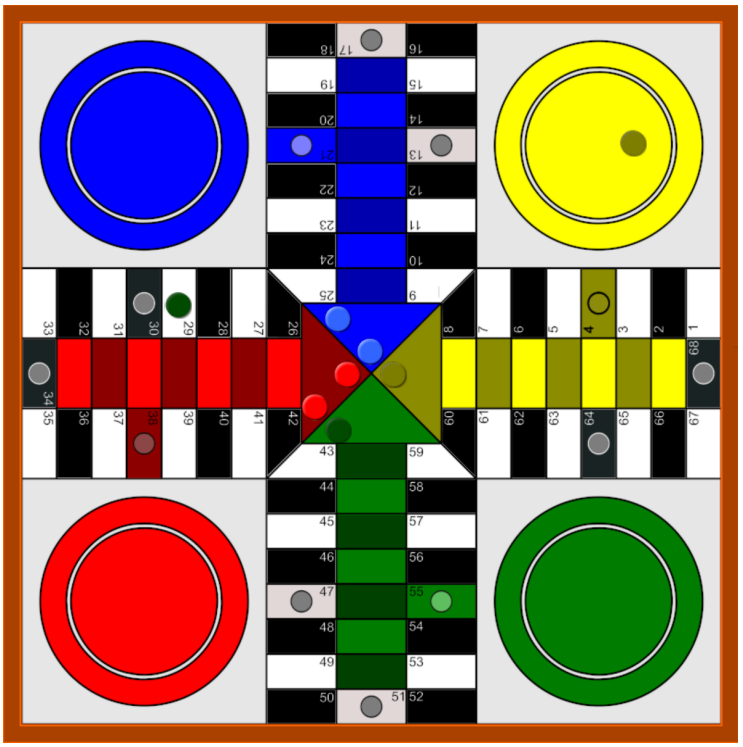
```
void AIPlayer::thinkMejorOpcion(color &c_piece, int &id_piece, int &dice) const
{
  // Vamos a mirar todos los posibles movimientos del jugador actual accediendo a los hijos del estado actual.

  // Cuando ya he recorrido todos los hijos, la función devuelve el estado actual. De esta forma puedo saber
  // cuándo paro de iterar.

  // Para ello, vamos a iterar sobre los hijos con la función de Parchis getChildren().
  // Esta función devuelve un objeto de la clase ParchisBros, que es una estructura iterable
  // sobre la que se pueden recorrer todos los hijos del estado sobre el que se llama.
  ParchisBros hijos = actual->getChildren();

  bool me_quedo_con_esta_accion = false;

  // La clase ParchisBros viene con un iterador muy útil y sencillo de usar.
  // Al hacer begin() accedemos al primer hijo de la rama,
  // y cada vez que hagamos ++it saltaremos al siguiente hijo.
  // Comparando con el iterador end() podemos consultar cuándo hemos terminado de visitar los hijos.

  for (ParchisBros::Iterator it = hijos.begin();
       it != hijos.end() and !me_quedo_con_esta_accion; ++it)
  {
     Parchis siguiente_hijo = *it;

     // Si en el movimiento elegido comiera ficha, llegara a la meta o ganara, me quedo con esa acción.
```

```
      // Termino el bucle en este caso.
      if (siguiente_hijo.isEatingMove() or
          siguiente_hijo.isGoalMove() or
          (siguiente_hijo.gameOver() and siguiente_hijo.getWinner() == this->jugador))
      {
         me_quedo_con_esta_accion = true;
         c_piece = it.getMovedColor();
         id_piece = it.getMovedPieceId();
         dice = it.getMovedDiceValue();
      }
   }

   // Si he encontrado una acción que me interesa, la guardo en las variables pasadas por referencia.
   // (Ya lo he hecho antes, cuando les he asignado los valores con el iterador).

   // Si no he encontrado ninguna acción que me interese, voy a mover la ficha más adelantada como en el caso
anterior.
   if(!me_quedo_con_esta_accion){
      thinkFichaMasAdelantada(c_piece, id_piece, dice);
   }
}
```

Una vez más, probamos a enfrentar al jugador que utiliza el iterador (de **id=2**) con el jugador que mueve la ficha más adelantada del apartado anterior (de **id=1**). Para ello, podemos usar el siguiente comando:

```
./ParchisGame --p1 AI 1 "Ya hace cosas" --p2 AI 2 "Va en serio"
```

El resultado que obtenemos al final de la partida es el siguiente:



Otra vez vemos que este nuevo comportamiento parece ser bastante superior a los anteriores. De nuevo, nuestro jugador nuevo (J2) gana claramente al J1.

IMPORTANTE!!!

- Este último agente si ve que se puede comer una ficha se la come sin pensárselo. ¡Muchas veces se come las de su otro color! Esto a veces le podría perjudicar (o no…)
- Este ejemplo pretende mostrar cómo se puede utilizar el iterador de **ParchisBros** pero **en ningún momento se está implementando ninguno de los algoritmos de búsqueda** que se pide para la realización de esta práctica. Veremos cómo aplicar este iterador para realizar la búsqueda en la siguiente sección.

3.3. La búsqueda de verdad: implementando el algoritmo Minimax.

Con lo que hemos visto en las secciones anteriores, ya estamos en condiciones de implementar nuestro primer algoritmo de búsqueda con adversario para abordar este juego. El **algoritmo Minimax** es una técnica de búsqueda diseñada para árboles de juego en los que no es viable explorar el árbol completo. Se apoya en funciones **heurísticas** que dan una valoración estática de cómo de prometedor es cierto estado de la partida. Con estas valoraciones, el algoritmo se encargará de encontrar la acción que pueda llevar al jugador al nodo más prometedor a una profundidad determinada y fijada previamente, asumiendo que el oponente siempre va a intentar realizar también la acción que más le pueda beneficiar.

Como hemos mencionado, en primer lugar, para poder poner en marcha el algoritmo Minimax, necesitaremos una heurística que nos permita valorar los nodos. En el código proporcionado, ya tenemos una implementada, denominada **ValoracionTest**. Vamos a analizarla. En primer lugar, veamos cómo se define la heurística. Esto es muy importante, ya que, **como se explica en el guion**, las nuevas heurísticas que decidamos desarrollar deben implementarse de la misma forma. Debemos crear una clase, con el nombre que queramos poner a nuestra heurística, que extienda a la clase **Heuristic**, una clase que ya existe en el proyecto y de la que no nos tenemos que preocupar mucho, lo único que debemos hacer es que nuestra clase herede de ella. Que no cunda el pánico con la herencia en C++. Lo único que tenemos que hacer que involucre a la herencia es esto:

```
// Fichero AIPlayer.h (este código ya está en el software)
// class <Nombre que quiero ponerle a mi clase> : public Heuristic
// (esta línea es lo único en lo que interviene la herencia, podemos dormir tranquilos)
class ValoracionTest: public Heuristic{
  protected:
     // Método que vamos a redefinir (debe ser protected)
     virtual float getHeuristic(const Parchis &estado, int jugador) const;
};
```

El método **getHeuristic** es lo que tendremos que definir para valorar los distintos estados del juego. Debe recibir los dos parámetros que se muestran arriba. Un estado de juego (de la clase **Parchis**) no modificable (solo podremos usar sus métodos de consulta **const**) y un entero indicando el jugador que invoca a la función (para saber si la situación del tablero es positiva o negativa para mí). Podemos ver cómo se redefine este **getHeuristic** para **ValoracionTest** y analizar qué hace exactamente:

```
// AIPlayer.cpp (este código ya está en el software)
float ValoracionTest::getHeuristic(const Parchis& estado, int jugador) const{
  // Heurística de prueba proporcionada para validar el funcionamiento del algoritmo de búsqueda.
  int ganador = estado.getWinner();
  int oponente = (jugador + 1) % 2;

  // Si hay un ganador, devuelvo más/menos infinito, según si he ganado yo o el oponente.
  if (ganador == jugador){
     return gana;
  }
  else if (ganador == oponente){
     return pierde;
  }
  else{
     // Colores que juega mi jugador y colores del oponente
     vector<color> my_colors = estado.getPlayerColors(jugador);
     vector<color> op_colors = estado.getPlayerColors(oponente);

     // Recorro todas las fichas de mi jugador
     int puntuacion_jugador = 0;
     // Recorro colores de mi jugador.
     for (int i = 0; i < my_colors.size(); i++){
        color c = my_colors[i];
        // Recorro las fichas de ese color.
        for (int j = 0; j < num_pieces; j++){
           // Valoro positivamente que la ficha esté en casilla segura o meta.
           if (estado.isSafePiece(c, j)){
              puntuacion_jugador++;
           }
           else if (estado.getBoard().getPiece(c, j).get_box().type == goal){
              puntuacion_jugador += 5;
           }
```

```
        }
    }

    // Recorro todas las fichas del oponente
    int puntuacion_oponente = 0;
    // Recorro colores del oponente.
    for (int i = 0; i < op_colors.size(); i++){
        color c = op_colors[i];
        // Recorro las fichas de ese color.
        for (int j = 0; j < num_pieces; j++){
            if (estado.isSafePiece(c, j)){
                // Valoro negativamente que la ficha esté en casilla segura o meta.
                puntuacion_oponente++;
            }
            else if (estado.getBoard().getPiece(c, j).get_box().type == goal){
                puntuacion_oponente += 5;
            }
        }
    }

    // Devuelvo la puntuación de mi jugador menos la puntuación del oponente.
    return puntuacion_jugador - puntuacion_oponente;
  }
}
```

Si vamos bloque a bloque analizando el código anterior, podemos ver que se va accediendo a diferentes funciones de la clase **Parchis** para hacer estas comprobaciones sobre el estado actual:

- Primero se comprueba si en ese estado alguien ha ganado o ha perdido la partida. En ese caso, se devuelve o el valor más grande posible (si mi jugador ha ganado) o el más pequeño posible (si mi jugador ha perdido).
- Después, extraigo cuáles son mis dos colores y cuáles son los del oponente.
- Para mi jugador, recorro todas mis fichas. Para recorrerlas, primero itero sobre mis colores, y después, dentro de mi color, itero sobre el número de ficha (0 y 1, al tener dos fichas por color). Para cada ficha, puntúo positivamente tanto que esté en una casilla segura como que haya llegado a la meta.

- Repito lo mismo con las fichas del oponente pero en negativo. Devuelvo la diferencia entre ambas puntuaciones como mi valoración final.

En resumen, ValoracionTest puntúa positivamente mis fichas en casillas seguras y en meta y penaliza las fichas del rival en esas mismas casillas. Como podéis intuir, tiene mucho margen de mejora. Por ejemplo, en ningún momento se llega a valorar si las fichas están más o menos cerca de la meta.

Una vez teniendo nuestra heurística implementada y entendiendo su funcionamiento, vamos por fin a implementar el algoritmo Minimax. Su funcionamiento es muy sencillo. Vamos recorriendo recursivamente el árbol de juego. Si estamos en un nodo hoja (la partida ha terminado) o hemos llegado al límite de profundidad, llamamos a nuestra heurística para que nos devuelva el valor de ese nodo. Después, en los nodos MAX, iteramos sobre todos los hijos y en cada hijo volvemos a llamar al Minimax con un nivel más de profundidad. Ese Minimax nos devolverá un valor (uno para cada hijo de los que iteramos). Al terminar de iterar, nos quedaremos con el máximo de los valores, que será lo que devolvamos. Por último, en los nodos MIN, iteramos de forma análoga a los nodos MAX, solo que en este caso nos quedaremos con el mínimo de los valores obtenidos.

Tendremos que devolver la acción a realizar, no solo el valor Minimax del árbol. Por ello, debemos pasar al algoritmo por referencia los tres valores que determinan el movimiento (los mismos tres valores que se pasan al método think). Y, cada vez que nos encontremos con un nodo con mejor valor minimax y **solo a profundidad 0** (porque queremos la acción inmediata y si actualizamos estos valores por referencia a otras profundidades pueden machacar la acción correcta que tuviéramos almacenada), debemos asignar a estos valores el valor del movimiento que nos llevaría a ese nodo (el que nos da el iterador, como vimos en la sección 3.2 del guion).

Basándonos en esto, podríamos proponer una cabecera para el Minimax de este estilo:

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

Donde:

- **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.
- **Heuristic *heuristic**: el puntero a la heurística que queramos utilizar para valorar los nodos. Por cómo estamos definiendo las heurísticas cuando queramos cambiar de heurística solo tendremos que cambiar el parámetro que pasamos en esta posición por la heurística que queramos.

Y el código finalmente, en base a la descripción anterior, quedaría como sigue:

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

{

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

  { // Nodo terminal o profundidad límite: llamo a la función heurística

    // IMPORTANTE: USAMOS EL MÉTODO evaluate AUNQUE HAYAMOS REDEFINIDO LA CLASE HEURISTIC

     return heuristic->evaluate(actual, jugador);

  }


  // Comparo mi jugador con el actual de la partida para saber en qué tipo de nodo estoy

  else if (actual.getCurrentPlayerId() == jugador)

  { // Nodo MAX

     float valor = menosinf; // Inicialización lo más pequeña posible para ir calculando el máximo


     // Obtengo los hijos del nodo actual y los recorro

     ParchisBros rama = actual.getChildren();

     for (ParchisBros::Iterator it = rama.begin(); it != rama.end(); ++it)

     {


        Parchis nuevo_hijo = *it;

        // Búsqueda en profundidad (llamada recursiva)

         float new_val = Minimax(nuevo_hijo, jugador, profundidad + 1, profundidad_max, c_piece, id_piece, dice,
heuristic);
```

```
        if (new_val > valor)
        {
           // Me voy quedando con el máximo
           valor = new_val;
           if (profundidad == 0)
           {
              // Almaceno el movimiento que me ha llevado al mejor valor (solo en la raíz)
              c_piece = it.getMovedColor();
              id_piece = it.getMovedPieceId();
              dice = it.getMovedDiceValue();
           }
        }
     }


     return valor;
  }


  else
  { // Nodo MIN
     float valor = masinf; // Inicialización lo más grande posible para ir calculando el mínimo


     // Obtengo los hijos del nodo actual y los recorro
     ParchisBros rama = actual.getChildren();
     for (ParchisBros::Iterator it = rama.begin(); it != rama.end(); ++it)
     {
        Parchis nuevo_hijo = *it;
        // Búsqueda en profundidad (llamada recursiva)
         float new_val = Minimax(nuevo_hijo, jugador, profundidad + 1, profundidad_max, c_piece, id_piece, dice,
heuristic);
        // Me voy quedando con el mínimo
        if (new_val < valor)
        {
           valor = new_val;
```

```
        }
      }

      return valor;
  }
}
```

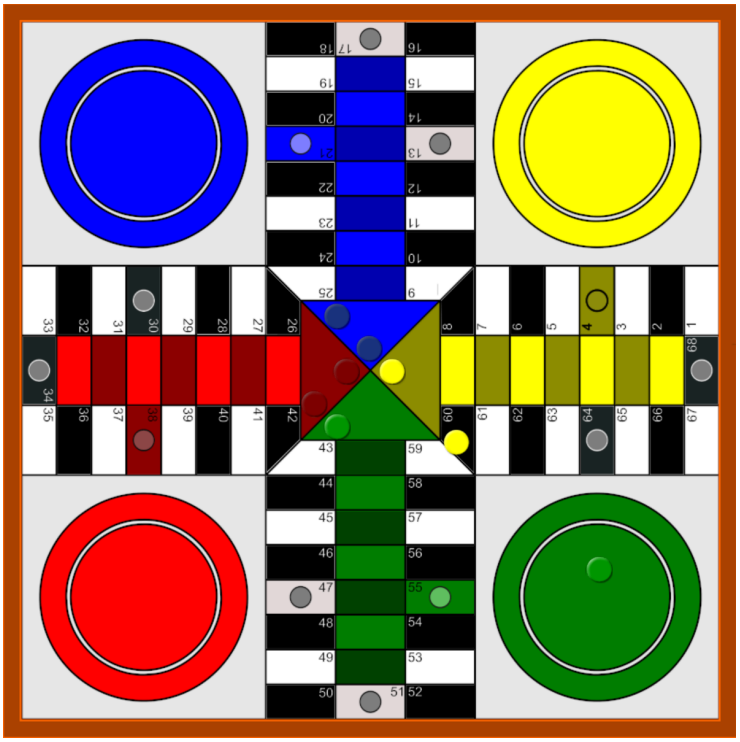
Ya solo tenemos que llamarlo desde nuestro método **think**. Podemos extender el switch anterior para ello (se recomienda usar `PROFUNDIDAD_MINIMAX=4`):

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

 // ...
  float valor; // Almacena el valor con el que se etiqueta el estado tras el proceso de busqueda.
  // Defino las heurísticas que quiera usar.
  ValoracionTest valoracionTest;

  switch(id){
    // ...
    case 3:
           valor = Minimax(*actual, jugador, 0, PROFUNDIDAD_MINIMAX, c_piece, id_piece, dice, &valoracionTest);
           break;
    }
   cout << "Valor MiniMax: " << valor << "  Accion: " << str(c_piece) << " " << id_piece << " " << dice
        << endl;
}
```

Una vez más, probamos a enfrentar al jugador que ya hace búsqueda (de **id=3**) con el jugador que utiliza el iterador (de **id=2**). Para ello, podemos usar el siguiente comando:

./ParchisGame --p1 AI 2 "Va en serio" --p2 AI 3 "Esto sí que es búsqueda"

El resultado que obtenemos al final de la partida es el siguiente:



Podemos comprobar que, por muy buena que fuera la idea que teníamos en los apartados anteriores, eran estrategias inmediatas, y una búsqueda con cierta profundidad va a acabar arrasando siempre a un jugador así de simple.

3.4. Controlando el límite de búsqueda.

Si intentamos ejecutar el mismo algoritmo del apartado anterior con `PROFUNDIDAD_MINIMAX=5`, además de que podemos desesperarnos de lo lento que es, seguramente llegaremos a una situación como esta (podemos verla en la terminal):

```
====================
Nodos generados: 4053947
Nodos evaluados: 3599130
Total de nodos: 7653077
Tiempo de generación+evaluación: 47.3351
Me parece que te pasaste de pensar... :(
====================
Tiempo de movimiento: 53.4948 segundos
++++++++++++++++++++++++
La partida ha terminado
Ha ganado el jugador 2 (???)
¡¡¡ENHORABUENA, b!!!
El jugador 1 ha explotado de tanto pensar.
++++++++++++++++++++++++
```

Como se explica en el guion, el juego permite un **máximo de nodos generados + nodos evaluados** en cada movimiento. Si nos pasamos, **perderemos la partida automáticamente**. ¿Cómo lo podemos solucionar? Muy sencillo. La clase encargada de gestionar la cuenta de nodos se llama **NodeCounter**. Podemos acceder a ella estáticamente sin necesidad de instanciarla, y tenemos un método denominado **isLimitReached()** que nos avisa de que hemos alcanzado el límite y debemos parar de ejecutar.

Entonces, podemos modificar el código anterior para que haga lo siguiente: ejecutamos el minimax de forma convencional, y añadimos una comprobación de si se alcanza el límite de nodos en el bucle. En caso de que se alcance, debemos devolver un valor apropiado:

- Si estamos en un nodo MÁX, podemos devolver el mejor alfa encontrado hasta el momento. Así, el jugador MÁX podrá tomar la mejor decisión con lo que le ha dado tiempo a descubrir. Además, a profundidad 0, debemos devolver la acción asociada.
- Si estamos en un nodo MÍN, la situación es algo más delicada. Aunque hay diferentes formas de hacerlo (podéis explorarlas), os proponemos tratar el nodo actual como nodo terminal y devolver su evaluación estática como estimación de beta.

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

{

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

  { // Nodo terminal o profundidad límite: llamo a la función heurística

    // IMPORTANTE: USAMOS EL MÉTODO evaluate AUNQUE HAYAMOS REDEFINIDO LA CLASE HEURISTIC

     return heuristic->evaluate(actual, jugador);

  }


  else if (actual.getCurrentPlayerId() == jugador)
```

```
{ // Nodo MAX
    float valor = menosinf;

    // Obtengo los hijos del nodo actual y los recorro
    ParchisBros rama = actual.getChildren();
    for (ParchisBros::Iterator it = rama.begin(); it != rama.end(); ++it)
    {

        Parchis nuevo_hijo = *it;

        // Verificar si hemos alcanzado el límite
        if (NodeCounter::isLimitReached())
        {
            cout << "Límite de nodos alcanzado, devolviendo el mejor nodo parcial" << endl;
            if(profundidad == 0)
            {
                c_piece = it.getMovedColor();
                id_piece = it.getMovedPieceId();
                dice = it.getMovedDiceValue();
            }
            return valor;
        }
        // Búsqueda en profundidad (llamada recursiva)
            float new_val = Minimax_Limitado(nuevo_hijo, jugador, profundidad + 1, profundidad_max, c_piece,
id_piece, dice, heuristic);
        if (new_val > valor)
        {
            // Me voy quedando con el máximo
            valor = new_val;
            if (profundidad == 0)
            {
                // Almaceno el movimiento que me ha llevado al mejor valor (solo en la raíz)
                c_piece = it.getMovedColor();
                id_piece = it.getMovedPieceId();
```

```
            dice = it.getMovedDiceValue();
         }
      }
   }

   return valor;
}

else
{ // Nodo MIN
   float valor = masinf;

   // Obtengo los hijos del nodo actual y los recorro
   ParchisBros rama = actual.getChildren();
   for (ParchisBros::Iterator it = rama.begin(); it != rama.end(); ++it)
   {
      Parchis nuevo_hijo = *it;

      // Verificar si hemos alcanzado el límite
      if (NodeCounter::isLimitReached())
      {
         cout << "Límite de nodos alcanzado, devolviendo el mejor nodo parcial" << endl;
         return heuristic->evaluate(actual, jugador);
      }
      // Búsqueda en profundidad (llamada recursiva)
          float new_val = Minimax_Limitado(nuevo_hijo, jugador, profundidad + 1, profundidad_max, c_piece,
id_piece, dice, heuristic);
      // Me voy quedando con el mínimo
      if (new_val < valor)
      {
         valor = new_val;
      }
   }
   return valor;
```

```
  }
}
```

Si probamos esta nueva implementación del Minimax veremos que ahora sí podemos continuar la partida sin problema:

```
====================
Nodos generados: 1855208
Nodos evaluados: 1644797
Total de nodos: 3500005
Tiempo de generación+evaluación: 18.8535
====================
Tiempo de movimiento: 20.5777 segundos
----------------
Turno: 2
Jugador actual: 2 (b)
```

3.5. Versión alternativa al Minimax almacenando los nodos en un vector

Puede ser necesario, a la hora de implementar algunas variantes de los algoritmos pedidos en la práctica, tener todos los nodos hijos generados previamente y procesarlos después. Para evitar en estos casos generar los hijos varias veces, la clase **Parchis** dispone de una función, **getChildrenList**, que nos permite obtener directamente el vector con los hijos. Estos hijos, en lugar de venir almacenados como elementos de la clase **ParchisBros**, vienen con estructura de **ParchisSis**, que es un encapsulamiento muy similar que nos permite acceder tanto al estado del tablero (también con un operador *), como al dado, color y número de ficha que llevaron a ese estado.

Pasar de la versión de **ParchisBros** a **ParchisSis** es casi inmediata. Solo tenemos que sustituir las partes de código que se muestran a continuación:

```
// Obtener los hijos del nodo actual
ParchisBros rama = actual.getChildren();

// ...
// Cabecera del bucle con el iterador
for (ParchisBros::Iterator it = rama.begin(); it != rama.end(); ++it)
{
    Parchis nuevo_hijo = *it; // Acceso al estado con el iterador
    // ...
```

```
    // Acceso al movimiento con el iterador
    c_piece = it.getMovedColor();
    id_piece = it.getMovedPieceId();
    dice = it.getMovedDiceValue();
    // ...
}
// ...
```

Por estas otras:

```
// Obtener los hijos del nodo actual
vector<ParchisSis> rama = actual.getChildrenList();

// ...
// Cabecera del bucle con el vector de hijos
for(int i = 0; i < rama.size(); i++)
{
    ParchisSis hijo_i = rama[i];  // Acceso al estado con el hijo i-ésimo del vector
    Parchis nuevo_hijo = *hijo_i; // Acceso al estado

    // ...
    // Acceso al movimiento del i-ésimo hijo.
    c_piece = hijo_i.getMovedColor();
    id_piece = hijo_i.getMovedPieceId();
    dice = hijo_i.getMovedDiceValue();
    // ...
}
// …
```

En general, usar las versión vectorial solo es recomendable cuando de verdad se vaya a realizar alguna operación que involucre a todos los nodos. Con la versión de iteradores es más fácil ajustarse a los límites y no explorar nodos innecesarios. No hay que tener miedo al iterador, apenas se diferencia de la **i** de los bucles for ;)

4. ¿Y ahora qué?

En este tutorial hemos visto cómo usar la clase **AIPlayer** para diseñar comportamientos tanto ligeramente racionales como para implementar algoritmos de búsqueda en juegos (minimax). También se ha mostrado cómo la clase **Parchis** dispone de herramientas para acceder a gran variedad de información sobre el estado actual de la partida, la cual puede ser consultada por el agente para tomar la decisión que considere más oportuna, es decir, para el **diseño de la heurística.**

Para abordar la práctica será necesario, partiendo del algoritmo minimax proporcionado, implementar tanto la poda alfa-beta como las sucesivas mejoras que se proponen en el guion. Para ello, se establece la **limitación** de un **número máximo de nodos evaluados + generados**, que deberá respetarse en todas las versiones. Para ello, se explica cómo devolver el mejor camino obtenido hasta el momento en la sección 3.4 de este tutorial.

De igual forma que en este tutorial hemos programado distintos comportamientos y luego los hemos enfrentado entre ellos, durante la práctica deberemos diseñar diferentes versiones de la poda alfa-beta y diferentes heurísticas y enfrentarlas entre ellas, procediendo de forma parecida a como hemos hecho aquí, aprovechando la variable **id** de la clase **AIPlayer**. En el software inicial se proporciona un posible punto de partida (la sección comentada en el método **think**) con el que empezar a desarrollar la práctica. Este punto de partida es solo una sugerencia y el estudiante lo puede modificar según considere más conveniente. Esta iniciación al proceso de búsqueda se describe con más detalle en la sección 6.3 del guion de la práctica.

Por último, hay que comentar que en este tutorial solo hemos visto una pequeña parte de toda la API de la que disponen la clase **Parchis** y sus prolongaciones (**Board, Dice, Box, …**) para consultar la información relativa a una determinada situación de la partida. Hay muchas más funciones que se pueden consultar relativas a muchos aspectos del juego: distancias, barreras, casillas seguras, turnos, posiciones en el tablero, rebotes, etc. Recomendamos leer con detenimiento **el anexo** de la práctica, en la que se detallan las principales clases y muchos de los métodos de consulta disponibles para la práctica. También recomendamos echar un vistazo a los métodos públicos en los ficheros **Parchis.h, Board.h, Dice.h** y **Attributes.h**, donde vienen definidas todas las funciones de consulta que podrían llegar a usarse para elaborar una heurística. En la heurística de prueba, **ValoracionTest**, se puede ver también cómo se usan otras funciones de consulta diferentes a las mostradas en el tutorial. Para concluir, si tienes alguna duda sobre cómo se podría sacar determinada información del juego puedes preguntarnos a los profesores. E incluso si crees que hay alguna funcionalidad que no está implementada y que debería existir podríamos considerar añadirla.

(**IMPORTANTE**: para que una función de consulta pueda usarse en el método **think** debe llevar el calificador **const** al final. Las funciones que no lo llevan modificarían el estado del juego, y por tanto no

se pueden usar ni en el proceso de búsqueda ni a la hora de elaborar la heurística; de hecho, el compilador no lo permitirá).

5. Consideraciones finales

Si has llegado hasta aquí siguiendo todos los pasos del tutorial pero el guion te ha parecido demasiado largo (lo sentimos) y no lo has mirado con tanto detalle, recordamos por aquí algunas consideraciones importantes a la hora de iniciar la práctica (pero recuerda **volver a leerte el guion** luego):

- 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 (como ya se hace en la implementación proporcionada del minimax), 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 al principio el algoritmo de búsqueda necesite explorar muchos más nodos, pero el número de nodos a explorar se reducirá 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 se reducirán los nodos generados conforme pasen los turnos.

- Las **clases descritas en el anexo** del guion 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.