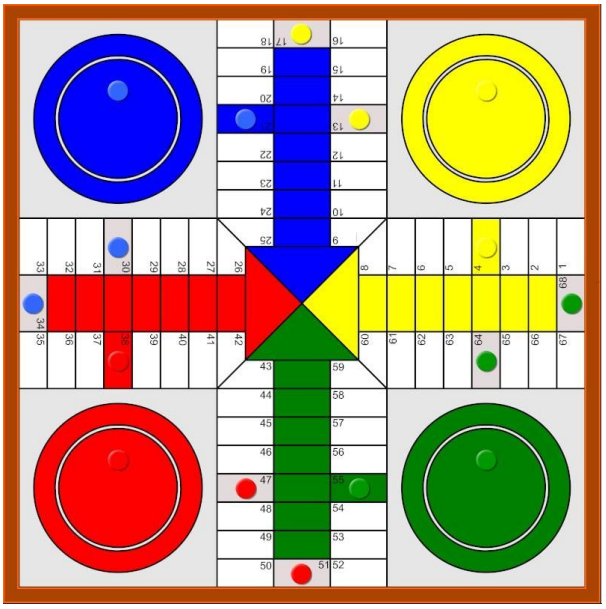
INTELIGENCIA ARTIFICIAL

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

Tutorial Práctica 3



Búsqueda con Adversario (Juegos)

El Parchís

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD DE GRANADA
Curso 2021-2022

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 Parchís 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.

Se recomienda, antes de empezar el tutorial, leer detenidamente el guión 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 5.3 del guión, 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 si es el jugador 1 y 1 si es el jugador 2.

```
class Player{
    protected:
    // Son heredados en la clase AIPlayer:
        Parchis *actual;
        int jugador;

    // ...

};
```

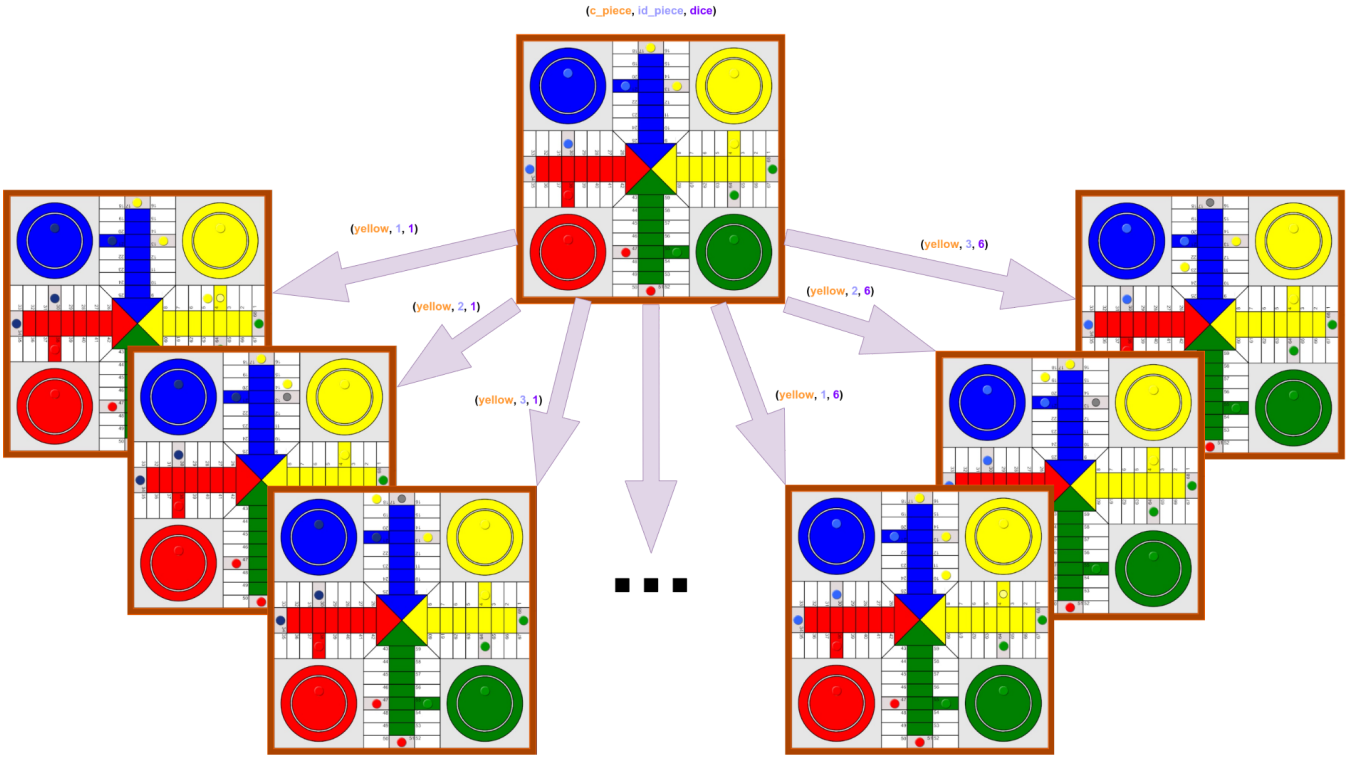
El método **think**, situado en la clase **AIPlayer**, es el encargado de, como en las prácticas anteriores, a partir de la información de la que dispone del juego el agente en el turno que le toque, determinar la acción a realizar. En este caso, la acción a realizar viene determinada por 3 parámetros:

- El **color de la ficha** que se mueve (en esta versión del Parchís **siempre será el color al que le toque mover en el turno actual**, que se podrá obtener llamando a la función **getCurrentColor()** de la clase Parchis). Este color se representa como un valor del enumerado **color** (ver Sección 5.1 del guión y 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 3. 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**, 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).

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 tendría en el tablero determinadas asignaciones de estos parámetros en el método think (la ficha amarilla de id 0 en este ejemplo es la que está en la casa y por tanto solo podría moverse al sacar un 5).



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. El color ya sabemos que será del color del turno actual y lo podremos asignar con **getCurrentColor()** directamente. Los dados disponibles los podemos obtener con **getAvailableDices(color)**. Nos devuelve un vector, 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(color, int)**. 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 color de ficha que se va a mover
    c_piece = actual->getCurrentColor();

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

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

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

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

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 van 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 thinkAleatorioMasInteligente(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:
            thinkAleatorioMasInteligente(c_piece, id_piece, dice);
            break;
        case 2:
            thinkFichaMasAdelantada(c_piece, id_piece, dice);
            break;
        case 3:
            thinkMejorOpcion(c_piece, id_piece, dice);
            break;
    }
}
```

En los siguientes apartados implementaremos comportamientos para las funciones **thinkAleatorioMasInteligente, thinkFichaMasAdelantada, 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:

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

3.1. Aleatorio pero con cabeza

Tras echar algunas partidas contra el jugador aleatorio propuesto podremos observar que es muy fácil ganarle. De hecho, demasiado fácil. Cuando empezamos a comernos sus fichas, podemos darnos cuenta de que la mayoría de ocasiones decide pasar turno a pesar de que podría haber elegido el dado número 5 para sacar una ficha. Pero como elige el dado completamente al azar, suele sacar un dado distinto con el que no tiene nada que mover.

Vamos a intentar mejorar este comportamiento. Para ello, en vez de elegir un dado completamente al azar, vamos a buscar primero para qué dados del jugador se puede mover al menos una ficha. Entonces, elegiremos aleatoriamente **solo entre los dados para los que se puede mover fichas**. Únicamente pasaremos turno cuando no pudiéramos mover ninguna ficha para ninguno de los dados disponibles.

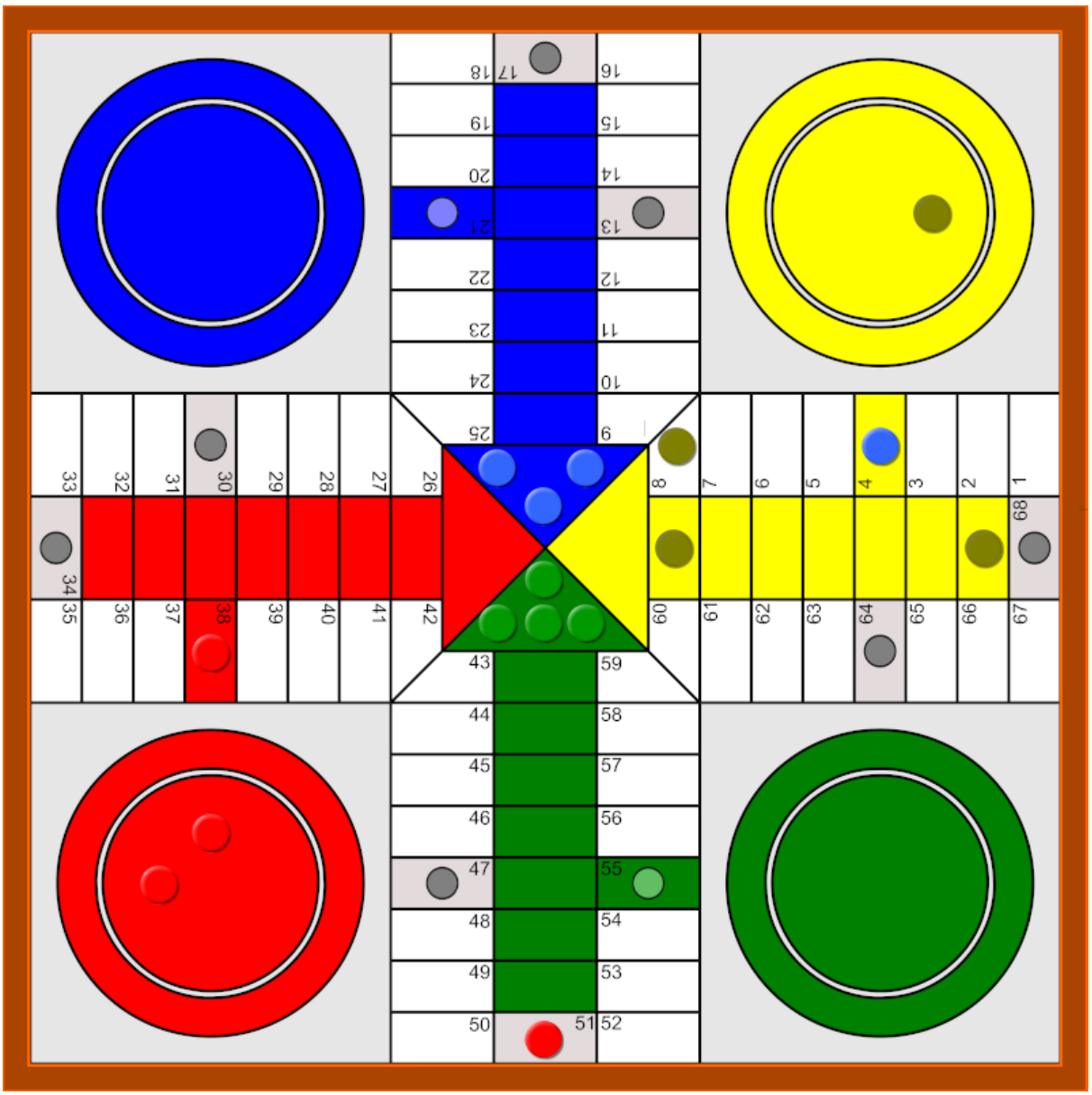
El comportamiento para la función **thinkAleatorioMasInteligente** queda como se muestra a continuación:

```
void AIPlayer::thinkAleatorioMasInteligente(color & c_piece, int & id_piece, int & dice) const{
    // El color de ficha que se va a mover.
    c_piece = actual->getCurrentColor();

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

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

    // En vez de elegir un dado al azar, miro primero cuáles tienen fichas que se puedan mover.
    vector<int> current_dices_que_pueden_mover_ficha;
    for (int i = 0; i < current_dices.size(); i++)
    {
        // Se obtiene el vector de fichas que se pueden mover para el dado elegido.
        current_pieces = actual->getAvailablePieces(c_piece, current_dices[i]);

        // Si se pueden mover fichas para el dado actual, lo añado al vector de dados que pueden mover fichas.
        if (current_pieces.size() > 0)
        {
            current_dices_que_pueden_mover_ficha.push_back(current_dices[i]);
        }
    }

    // Si no tengo ninún dado que pueda mover fichas, paso turno con un dado al azar (la macro SKIP_TURN me permite no mover).
    if(current_dices_que_pueden_mover_ficha.size() == 0){
        dice = current_dices[rand() % current_dices.size()];

        id_piece = SKIP_TURN;
    }

    // En caso contrario, elijo un dado de forma aleatoria de entre los que pueden mover ficha.
    else{
        dice = current_dices_que_pueden_mover_ficha[rand() % current_dices_que_pueden_mover_ficha.size()];

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

        // Muevo una ficha al azar de entre las que se pueden mover.
        id_piece = current_pieces[rand() % current_pieces.size()];
    }
}
```

Si tratamos ahora de enfrentar a nuestro nuevo jugador aleatorio “más inteligente” con el jugador aleatorio inicial vemos que, efectivamente, puede derrotarle fácilmente. Para ello ejecutamos el siguiente comando (recordemos que con el **id=0** lanzaremos el jugador aleatorio inicial y con el **id=1** nuestro nuevo jugador aleatorio):

```
./bin/Parchis --p1 AI 0 Random --p2 AI 1 "Random listo"
```

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



Efectivamente, nuestro nuevo jugador aleatorio (el jugador 2, azul y verde) ha ganado de forma bastante clara al jugador aleatorio original.

3.2. Eligiendo qué quiero mover

Hasta ahora, aunque hemos mejorado el comportamiento inicial del agente, los movimientos siguen siendo completamente aleatorios. En este apartado vamos 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.

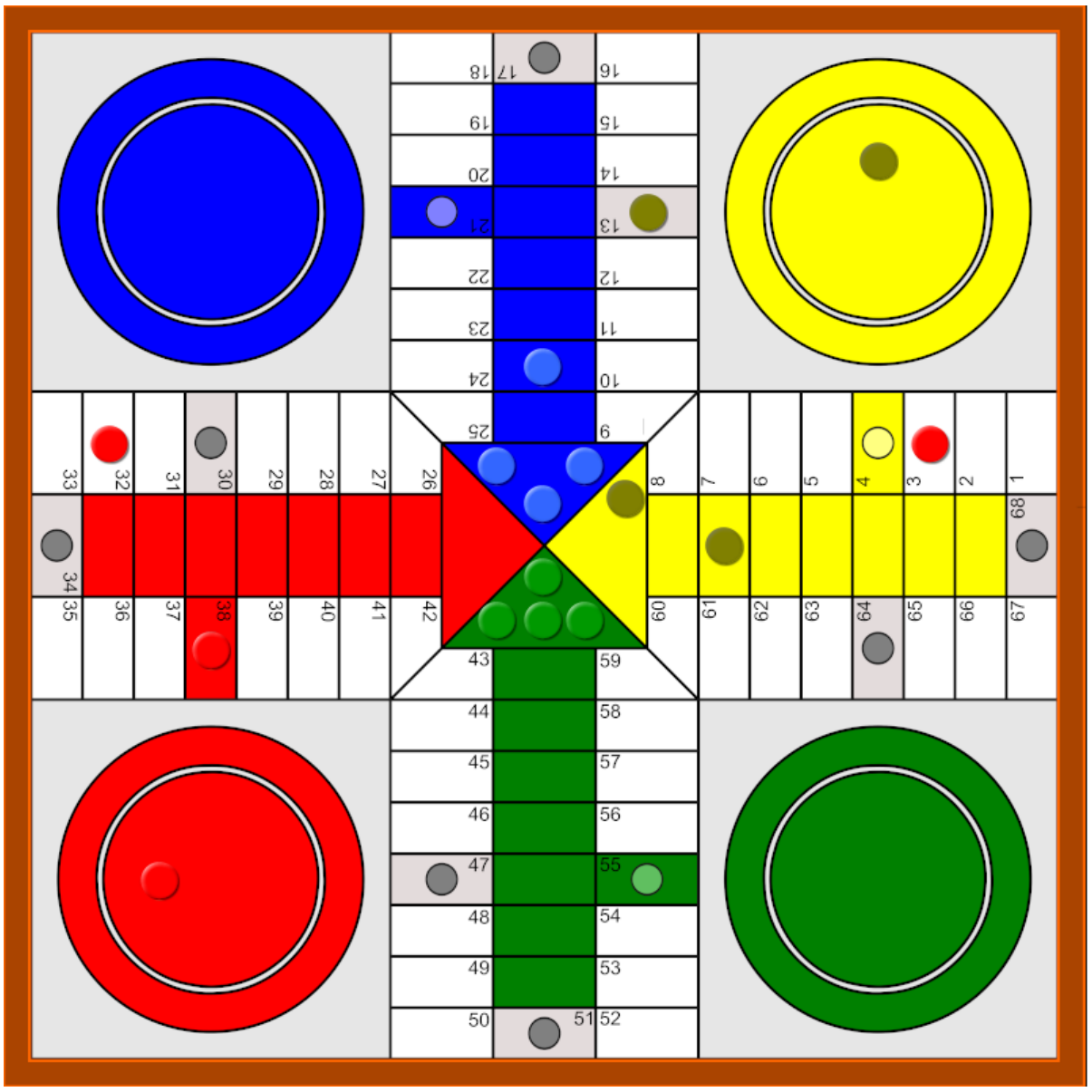
En el caso que estamos tratando ahora, como queremos mover siempre la ficha más adelantada, recorreremos todas las fichas de mi color y miraremos cuál tiene menor distancia a la meta. Finalmente seleccionaremos para mover la ficha con la menor distancia obtenida, o pasaremos turno si no hubiera ninguna 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 anterior.
    thinkAleatorioMasInteligente(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 la que esté más adelantada
    // (equivalentemente, la más cercana a la meta).

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

    int id_ficha_mas_adelantada = -1;
    int min_distancia_meta = 9999;
    for (int i = 0; i < current_pieces.size(); i++)
    {
        // distanceToGoal(color, id) devuelve la distancia a la meta de la ficha [id] del color que le indique.
        int distancia_meta = actual->distanceToGoal(c_piece, current_pieces[i]);
        if (distancia_meta < min_distancia_meta)
        {
            min_distancia_meta = distancia_meta;
            id_ficha_mas_adelantada = current_pieces[i];
        }
    }

    // Si no he encontrado ninguna ficha, paso turno.
    if (id_ficha_mas_adelantada == -1)
    {
        id_piece = SKIP_TURN;
    }
    // En caso contrario, moveré la ficha más adelantada.
    else
    {
        id_piece = id_ficha_mas_adelantada;
    }
}
```

De nuevo, probamos a enfrentar al nuevo jugador elaborado (de **id=2**) con el jugador aleatorio del apartado anterior (de **id=1**). Para ello, podemos usar el siguiente comando:

```
./bin/Parchis --p1 AI 1 "Random listo" --p2 AI 2 "Ya hace cosas"
```

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



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

3.3. Buscando entre los hijos: la función generateNextMove

A la hora de realizar la práctica será necesario implementar uno de los algoritmos de búsqueda pedidos. 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 de los métodos **generateNextMove** y **generateNextMoveDescending** para ello. Estos métodos actúan como una especie de iterador, de

forma que la primera vez que se llamen se pueda acceder al primer hijo del nodo que se esté desarrollando, y en las sucesivas llamadas se pueda acceder al resto de los hijos de forma ordenada. El orden que se sigue es ascendente en el valor de los dados para **generateNextMove** y descendente para **generateNextMoveDescending**. Dentro de cada valor de dado, las fichas se recorrerán siempre en orden según su id (de 0 a 3).

En este apartado vamos a continuar mejorando a nuestro agente y aprovecharemos para aprender a usar la función **generateNextMove**. 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 mantendremos el comportamiento anterior de mover la ficha más adelantada con un valor de dado elegido al azar.

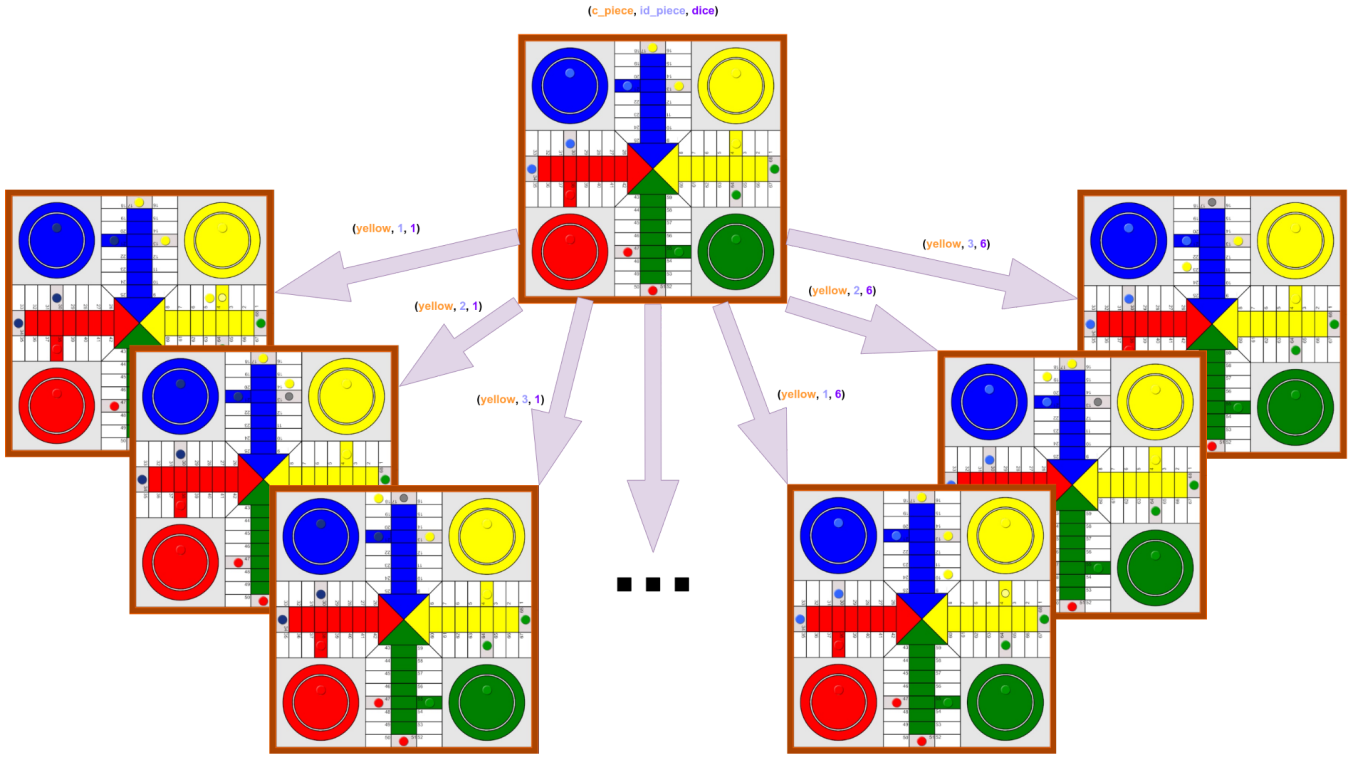
Para ello, es importante volver a destacar que la clase **Parchis** dispone de métodos para consultar casi cualquier cosa que se nos ocurra sobre el estado actual de la partida. En particular, para el caso planteado en este apartado disponemos de las funciones **isEatingMove(), isGoalMove(), gameOver()**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

- Recibe como argumentos (por referencia) un color de ficha **c_piece**, un id de ficha **id_piece** y un valor de dado **dice**. Como va a actuar como un iterador, el **c_piece**, **id_piece** y **dice** que tenemos que pasarle deben ser los de el último movimiento que hemos visitado. Cuando la función haya finalizado, **c_piece**, **id_piece** y **dice** se habrán actualizado a los valores del siguiente movimiento y la función nos devolverá el hijo (de la clase **Parchis)** asociado a dicho movimiento.
- Inicialmente, cuando todavía no he empezado a iterar, debo pasarle a la función los valores **c_piece=none**, **id_piece=-1** y **dice=-1**. Entonces, la función me devolverá el primer hijo y **c_piece, id_piece** y **dice** se actualizarán para ser el movimiento que nos lleva a ese primer hijo.
- Cuando ya no nos quedan hijos por recorrer, la función devuelve el nodo padre. De esta forma, y con ayuda también del **operador** == de la clase **Parchis**, podemos saber cuándo hemos terminado de recorrer los hijos.

Para ilustrar el funcionamiento, retomamos la figura que nos mostraba la ramificación para el tablero inicial de una partida:



Si llamamos **actual** al nodo padre (de la clase **Parchis**), y queremos recorrer todos los hijos de **actual**, podemos inicializar nuestras variables **c_piece=none**, **id_piece=-1** y **dice=-1**. Para acceder al primer hijo tenemos que hacer:

```
Parchis hijo = actual.generateNextMove(c_piece, id_piece, dice)
```

Tras esta llamada, tendremos que **c_piece=yellow**, **id_piece=1** y **dice=1**. Además, la variable **hijo** tendrá asociado el tablero más a la izquierda de la figura. Si volvemos a llamar a la función otra vez,

```
Parchis hijo = actual.generateNextMove(c_piece, id_piece, dice)
```

tendremos que los valores pasan a ser **c_piece=yellow**, **id_piece=2** y **dice=1**, y la variable **hijo** pasará a tener almacenado el segundo tablero. Si seguimos llamando a la función iremos pasando por cada uno de los tableros hijos y se seguirán actualizando **c_piece**, **id_piece** y **dice**. Finalmente, cuando hayamos generado el último hijo, que en este caso sería el tablero más a la derecha de la imagen, asociado al movimiento **c_piece=yellow**, **id_piece=3** y **dice=6**, si volvemos a llamar a **generateNextMove** el **hijo** que nos devolverá será el nodo **actual**. Entonces la comprobación **hijo == actual** nos permitirá asegurar que hemos terminado de recorrer todos los hijos.

Tras haber entendido cómo funciona **generateNextMove** 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 encuentro ningún hijo que cumpla esas condiciones, muevo con el comportamiento anterior.

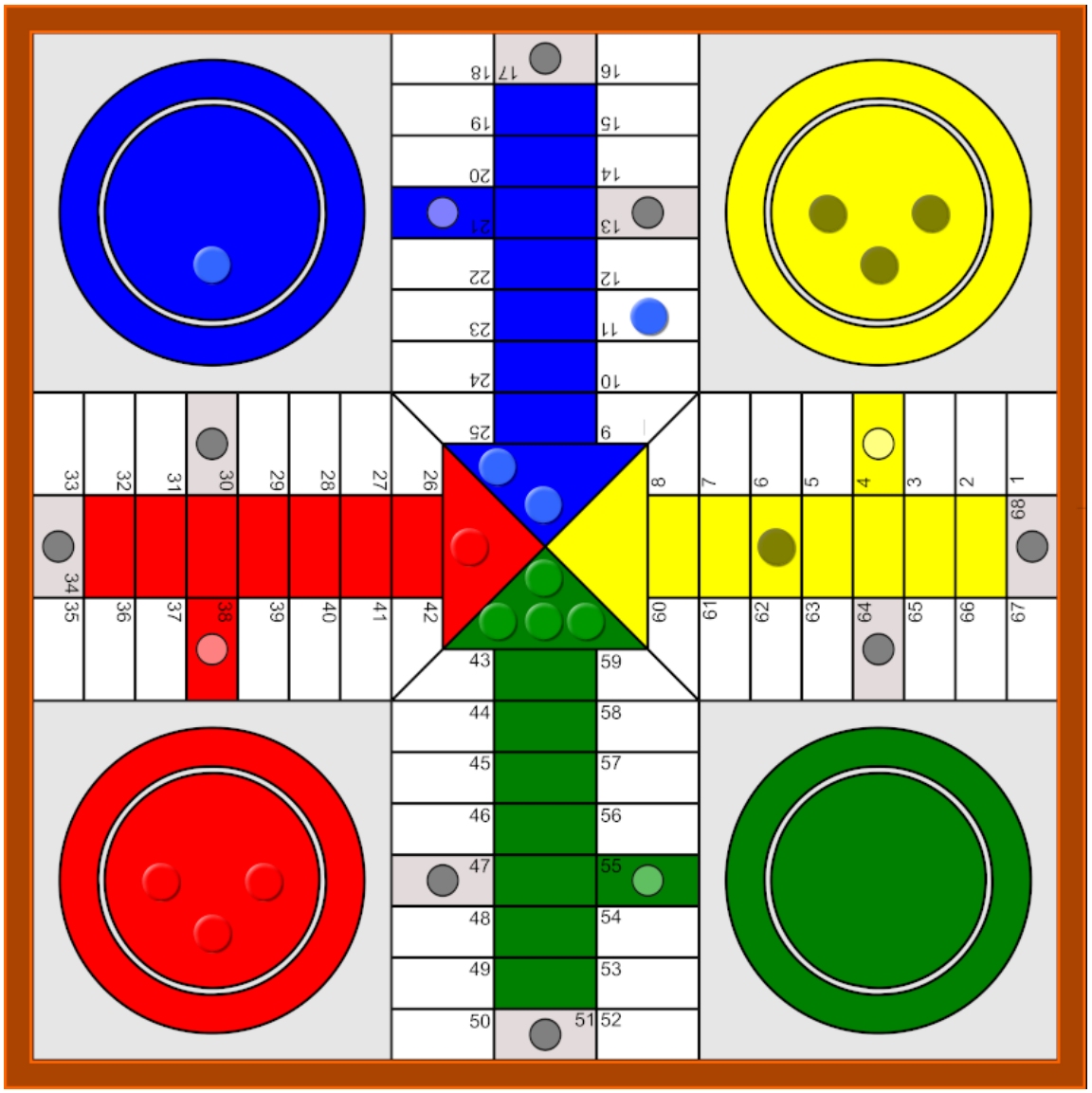
La implementación del comportamiento **thinkMejorOpcion** quedaría como se muestra a continuación:

```
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.

    // generateNextMove va iterando sobre cada hijo. Le paso la acción del último movimiento sobre
    // el que he iterado y me devolverá el siguiente. Inicialmente, cuando aún no he hecho ningún
    // movimiento, lo inicializo así.
    color last_c_piece = none; // El color de la última ficha que se movió.
    int last_id_piece = -1;    // El id de la última ficha que se movió.
    int last_dice = -1;        // El dado que se usó en el último movimiento.

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

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

    bool me_quedo_con_esta_accion = false;

    while(!(siguiente_hijo == *actual) && !me_quedo_con_esta_accion){
        if(siguiente_hijo.isEatingMove() or // Si con este movimiento como ficha, o
           siguiente_hijo.isGoalMove() or   // Si con este movimiento llego a la meta, o
           (siguiente_hijo.gameOver() and siguiente_hijo.getWinner() == this->jugador) // Si con este movimiento gano la partida.
        ){
            // Me quedo con la acción actual (se almacenó en last_c_piece, last_id_piece, last_dice al llamar a generateNextMove).
            me_quedo_con_esta_accion = true;
        }

        else{
            // Genero el siguiente hijo.
            siguiente_hijo = actual->generateNextMove(last_c_piece, last_id_piece, last_dice);
        }
    }

    // Si he encontrado una acción que me interesa, la guardo en las variables pasadas por referencia.
    if(me_quedo_con_esta_accion){
        c_piece = last_c_piece;
        id_piece = last_id_piece;
        dice = last_dice;
    }
    // Si no, muevo la ficha más adelantada como antes.
    else{
        thinkFichaMasAdelantada(c_piece, id_piece, dice);
    }
}
```

Una vez más, probamos a enfrentar al jugador que utiliza el **generateNextMove** (de **id=3**) con el jugador que mueve la ficha más adelantada del apartado anterior (de **id=2**). Para ello, podemos usar el siguiente comando:

./bin/Parchis --p1 AI 2 "Ya hace cosas" --p2 AI 3 "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 jugador 1.

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 método **generateNextMove** pero **en ningún momento se está implementando ninguno de los algoritmos de búsqueda** que se pide para la realización de esta práctica. Es tarea del alumnado implementar uno de dichos algoritmos e integrar la función que recorre los hijos de la forma adecuada dentro del algoritmo de búsqueda. Lo explicado aquí solo muestra cómo empezar a recorrer los hijos, cómo iterar y cómo parar, no se llega a hacer la búsqueda pedida.

4. ¿Y ahora qué?

En este tutorial hemos visto cómo usar la clase **AIPlayer** para diseñar comportamientos tanto aleatorios como ligeramente racionales para jugar al simulador del Parchís proporcionado. 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.

Para abordar la práctica, ahora será necesario implementar o bien el algoritmo Minimax o bien la Poda Alfa-Beta, tal como se indica en el guión. **El comportamiento racional del agente deberá ser consecuencia de alguno de estos algoritmos.** Un comportamiento sin búsqueda como los que se realizan en este tutorial no serán admitidos.

El algoritmo de búsqueda, cuando llegue al límite de profundidad establecido, deberá establecer una valoración para el nodo al que haya llegado. Esa valoración deberá medir cómo de prometedor es ese tablero de juego, y ahí es donde se pueden (y deben) empezar a considerar las distintas funciones de consulta de la clase **Parchis** que se han ido comentando a lo largo del tutorial.

De igual forma que en este tutorial hemos programado distintos comportamientos y luego los hemos enfrentado entre ellos, durante la práctica podremos diseñar distintas 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 5.4 del guión 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 la **sección 5 del guión**, en la que se detallan las principales clases y muchos de los métodos de consulta disponibles para la práctica. También, una vez leída esa sección, 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 guión 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 guión** luego):

- Puesto que, desde un punto de vista estratégico, en muchas ocasiones un **valor de dado mayor** puede suponer una **mayor ventaja** para el jugador que lo usa, es posible que explorando primero los dados de mayor valor se poden más nodos al aplicar la poda alfa-beta con una heurística apropiada. Por ello, a la hora de explorar los hijos, se proporcionan tanto **generateNextMove** como **generateNextMoveDescending**, para que el estudiante valore cuál puede resultar más eficiente durante la implementación. En el tutorial se ilustra con ejemplos cómo funcionan estos métodos.
- En segundo lugar, es importante destacar que en el juego propuesto **un turno se corresponde con un único movimiento de ficha, independientemente** de que ese movimiento sea repetido por el mismo jugador tras **sacar un 6**, o que sea un movimiento de **contarse 10 o 20** tras llegar a la meta o comer. En consecuencia, como sucesor de un nodo MÁX podríamos encontrarnos de nuevo otro nodo MÁX (lo mismo para los MÍN). Por tanto debemos tener en cuenta que la secuencia de nodos no va a ir alternándose necesariamente en cada nivel. Tras sacar un 6 como nodo MÁX bajaríamos a un nuevo nodo MÁX, con 1 más de profundidad. Igualmente, tras comer ficha bajaríamos a un nuevo nodo del mismo tipo, en el que únicamente tendremos que elegir de entre nuestras 4 posibles fichas con cuál contarnos 20. En cualquier caso, en todo momento podremos saber si somos un nodo MÁX o MIN, ya que conocemos el jugador que llama a la heurística y las funciones como **getCurrentPlayerId**, de la clase Parchis, nos indican a qué jugador le toca mover en cada turno. Recordemos que un nodo debería ser MÁX cuando el jugador que mueve es el que llamó al algoritmo de búsqueda.
- La **ramificación** del árbol de búsqueda **va a variar** de forma significativa **según la cantidad de dados** de los que dispongan los jugadores en cada turno. Por ello, es posible que el algoritmo de búsqueda sea **bastante lento inicialmente**, pero irá **aumentando su velocidad** conforme los jugadores vayan gastando dados, hasta que estos se renueven. También, conforme vaya

avanzando la partida irán quedando menos opciones por mover, por lo que también tenderá a pensar más rápido conforme pasen los turnos.

- Una vez más, recordamos que las **clases descritas en la sección 5** disponen de funciones que pueden ser de mucha utilidad para el desarrollo de heurísticas. Es **importante echar un vistazo a las cabeceras** de las clases mencionadas para descubrir todas las posibilidades que se ofrecen.