

INTELIGENCIA ARTIFICIAL

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

Tutorial Práctica 3



Búsqueda con Adversario (Juegos)
El Parchís (BOOM!)

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



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 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 5.3 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 rojo) y 1 para el segundo (azul y verde).

```
class Player{
    protected:
        // ...

        // Son heredados en la clase AIPlayer
        Parchis *actual;

        int jugador;

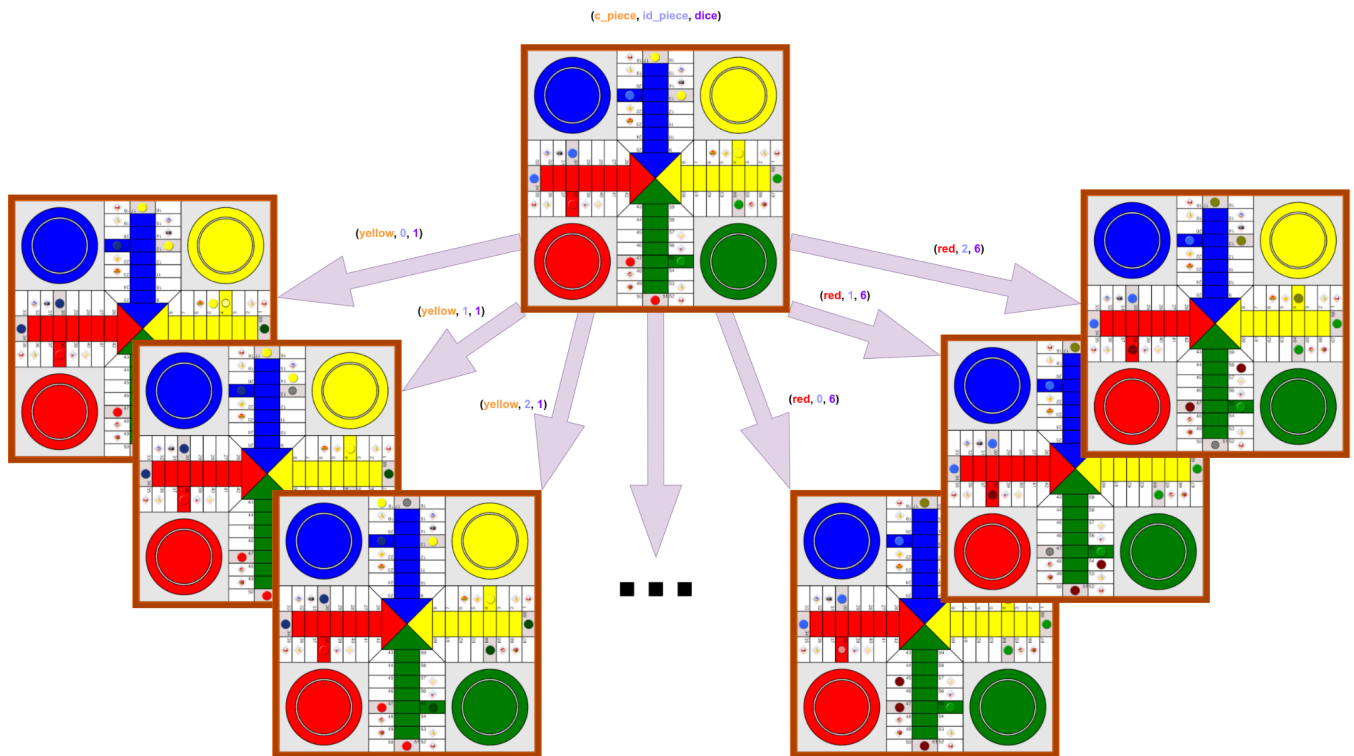
        // ...
};
```



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, 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 Sección 5.1 del guion 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 2. 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 **power**.

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 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
 */
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** 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:

```
./build/ParchisGame --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, es 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. También vemos que no hace uso de ninguno de los dados especiales que se haya podido ir encontrando cuando se ha movido por el tablero.

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 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;

    // Obtengo el vector de dados que puedo usar para el movimiento.
    // - actual->getAvailableNormalDices(player) -> solo dados disponibles en ese turno.
    // Importante : "getAvailableNormalDices" me da los dados que puedo usar en el turno actual.
    // Por ejemplo, si me tengo que contar 10 o 20 solo me saldrán los dados 10 y 20.
    // Puedo saber qué más dados tengo, aunque no los pueda usar en este turno, con:
    // - actual->getNormalDices(player) -> todos los dados
    current_dices = actual->getAvailableNormalDices(player);

    // 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(player, 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 ningún 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;

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

// 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(player, dice);

    // Muevo una ficha al azar de entre las que se pueden mover.
    int random_id = rand() % current_pieces.size();
    id_piece = get<1>(current_pieces[random_id]);
    c_piece = get<0>(current_pieces[random_id]);
}
```

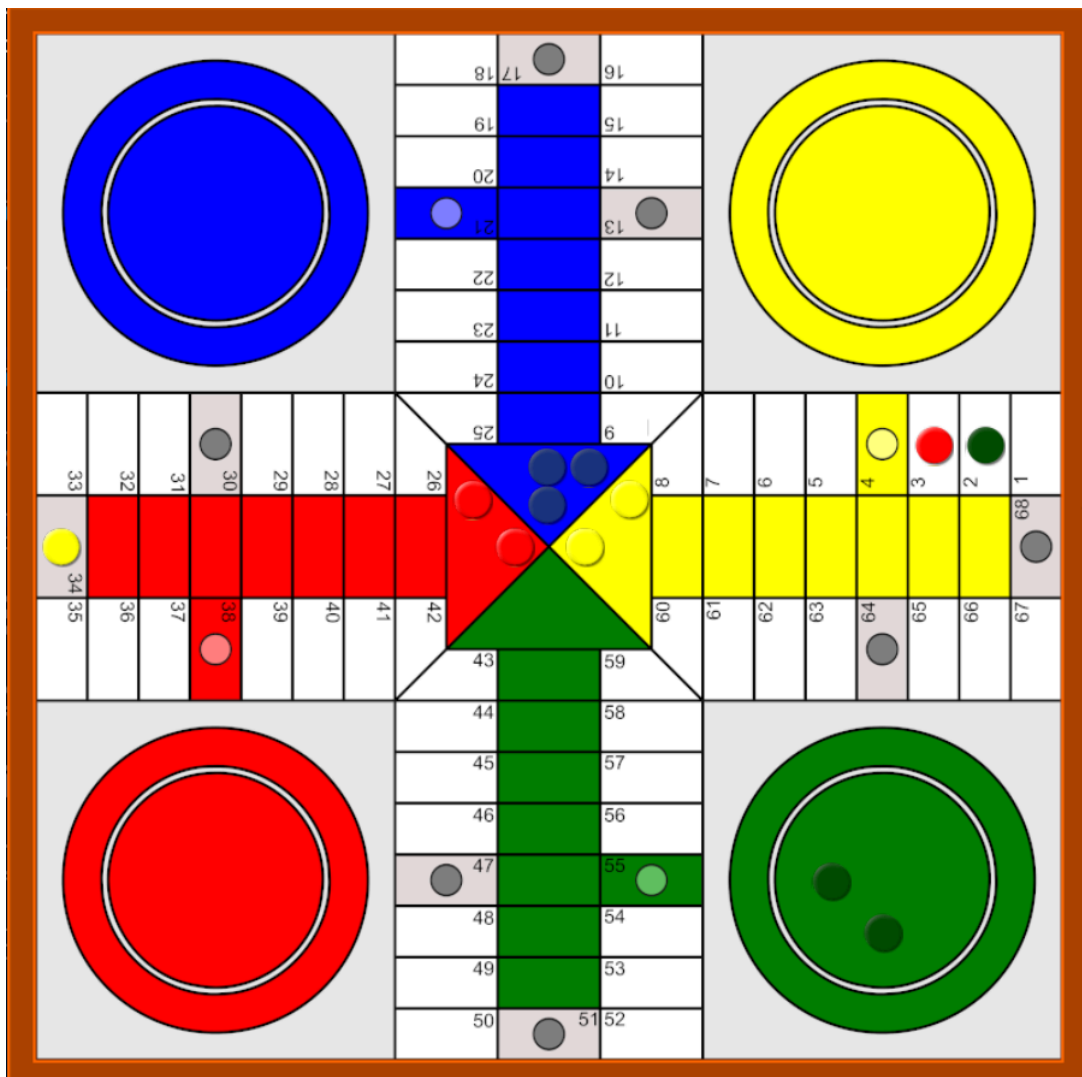
}

}

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):

```
./build/ParchisGame --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 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 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 (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;
    }
}

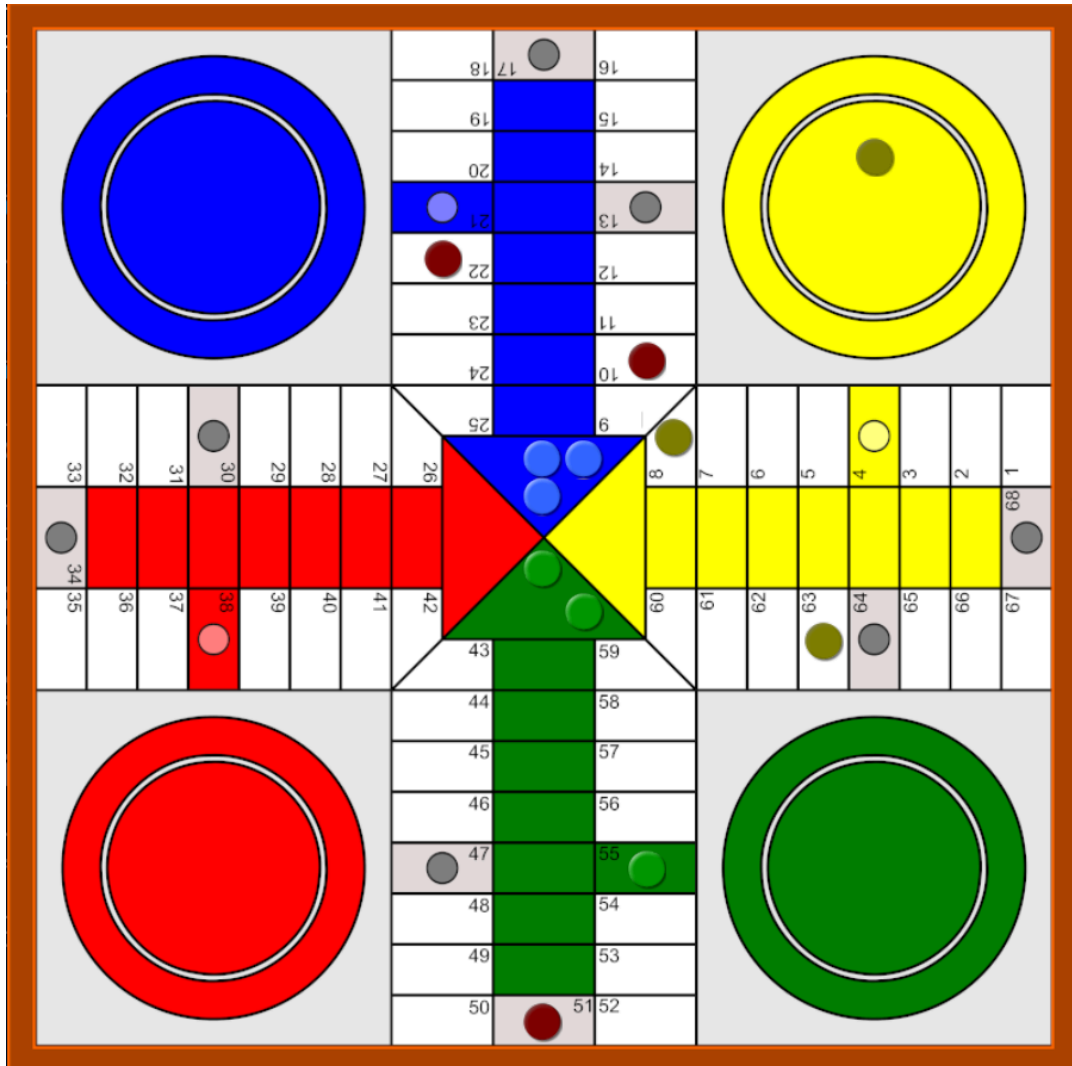
// Si no he encontrado ninguna ficha, paso turno.
if (id_ficha_mas_adelantada == -1)
{
    id_piece = SKIP_TURN;
    c_piece = actual->getCurrentColor(); // Le tengo que indicar mi color actual al pasar turno.
}

// En caso contrario, moveré la ficha más adelantada.
else
{
    id_piece = id_ficha_mas_adelantada;
    c_piece = col_ficha_mas_adelantada;
}
}
```

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:

```
./build/ParchisGame --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: el iterador de ParchisBros

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 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 2), visitando primero el primer color del jugador (amarillo/azul) y luego el segundo (rojo/verde).

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 que nos proporcione un mayor aumento de la energía.

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 **getPower(jugador)**, **isEatingMove()**, **isGoalMove()**, **gameOver()** y **getWinner()** que nos permiten obtener, respectivamente:

- El valor de la barra de energía asociada al jugador pasado como argumento.
- 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, me quedo con la acción que me suponga un mayor incremento de la energía.

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.


// Voy a moverme a la casilla siempre con la que gane más energía, salvo que me encuentre con
// algún movimiento muy interesante, como comer fichas o llegar a la meta.

int current_power = actual->getPowerBar(this->jugador).getPower();

int max_power = -101; // Máxima ganancia de energía.


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();
    }

    // En caso contrario, me voy quedando con el que me dé más energía.
    else{
        int new_power = siguiente_hijo.getPower(this->jugador);
        if(new_power - current_power > max_power){
            c_piece = it.getMovedColor();
            id_piece = it.getMovedPieceId();
            dice = it.getMovedDiceValue();
            max_power = new_power - current_power;
        }
    }
}
```



```

}

}

}

// 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).

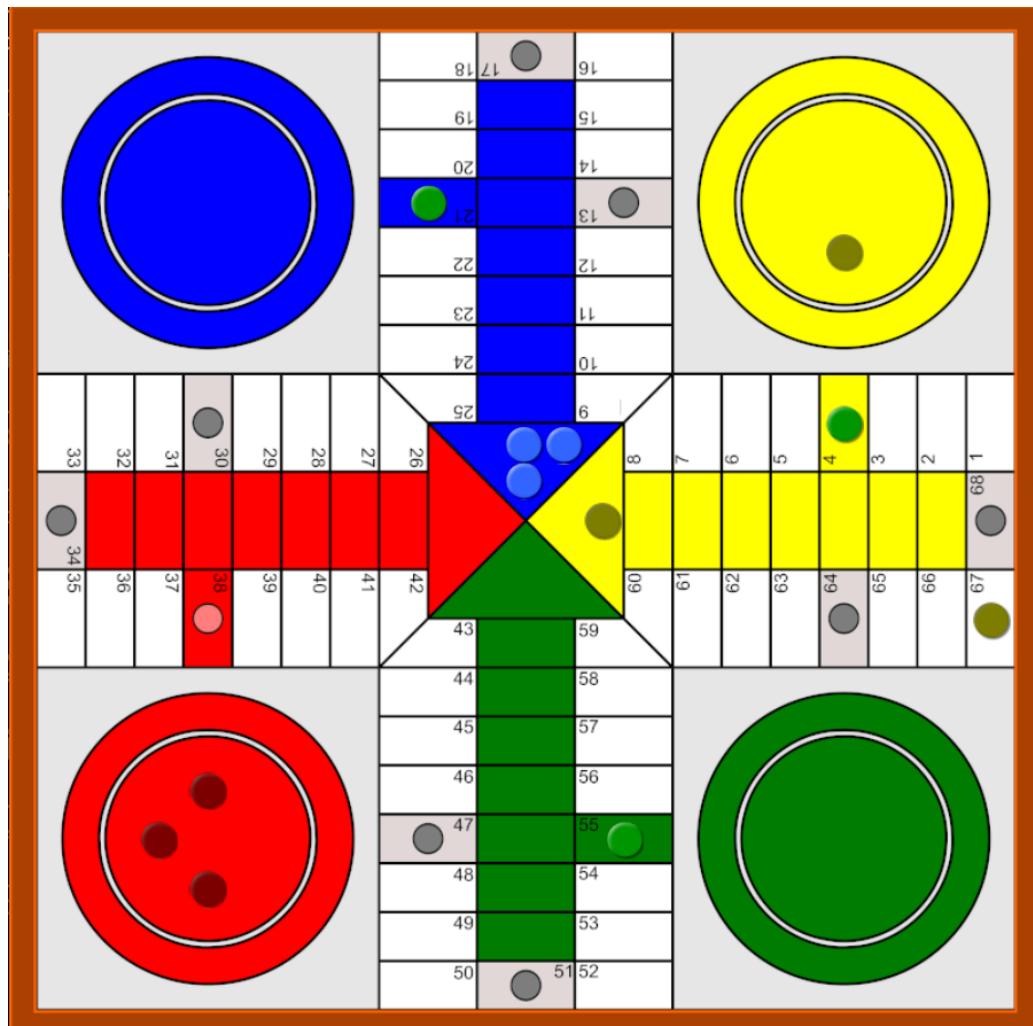
}

```

Una vez más, probamos a enfrentar al jugador que utiliza el iterador (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:

`./build/ParchisGame --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 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. 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.
- Este agente no aprovecha demasiado el poder del dado especial. El funcionamiento de este dado depende del valor de la energía en el momento de ser seleccionado, así que deberá elegirse con cabeza (y más sabiendo que a veces tiene consecuencias...).

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 guion. **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 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 la **sección 5 del guion**, 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 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 leer el guion** luego):

- En primer lugar, tanto si se opta por implementar Minimax como la Poda Alfa-Beta, va a ser necesario realizar una búsqueda en profundidad partiendo del estado actual del juego. En consecuencia, para cada nodo se hace necesario obtener todos los posibles tableros a los que se podría pasar, con cada uno de los posibles movimientos. Para ello, se ha implementado la clase **ParchisBros** que se encarga de la generación de los nodos hijos posibles a partir de un estado actual del parchís. Esta clase incluye un **iterador** que se usará para ir recorriendo los nodos hijos del nodo actual. Dado que desde un punto de vista estratégico, en muchas ocasiones un **valor de dado mayor** puede suponer una **mayor ventaja**, el operador++ del iterador se ha implementado recorriendo los hijos en orden descendente del dado.
- En segundo lugar, es importante destacar que en el juego propuesto **un turno se corresponde con un único movimiento de ficha, independientemente** de que ese movimiento sea repetido por el mismo jugador tras **sacar un 6**, que sea un movimiento de **contarse 10 o 20** tras llegar a la meta o comer, o que sea la acción que se ejecute mediante un dado especial. En consecuencia, como sucesor de un nodo MÁX podríamos encontrarnos de nuevo otro nodo MÁX (lo mismo para los MÍN). Por tanto debemos tener en cuenta que la secuencia de nodos no va a ir alternándose necesariamente en cada nivel. Tras sacar un 6 como nodo MÁX bajaríamos a un



nuevo nodo MÁX, con 1 más de profundidad. Igualmente, tras comer ficha bajaríamos a un nuevo nodo del mismo tipo, en el que únicamente tendríamos que elegir de entre nuestras posibles fichas con cuál contarnos 20. En cualquier caso, en todo momento podremos saber si somos un nodo MÁX o MIN, ya que conocemos el jugador que llama a la heurística y las funciones como `getCurrentPlayerId`, de la clase `Parchis`, nos indican a qué jugador le toca mover en cada turno. Recordemos que un nodo debería ser MÁX cuando el jugador que mueve es el que llamó al algoritmo de búsqueda.

- La **ramificación** del árbol de búsqueda **va a variar** de forma significativa **según la cantidad de dados** de los que dispongan los jugadores en cada turno. Por ello, es posible que el algoritmo de búsqueda sea **bastante lento inicialmente**, pero irá **aumentando su velocidad** conforme los jugadores vayan gastando dados, hasta que estos se renueven. También, conforme vaya avanzando la partida irán quedando menos opciones por mover, por lo que también tenderá a pensar más rápido conforme pasen los turnos.
- Las **clases descritas en la sección 5** 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.