INTELIGENCIA ARTIFICIAL

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

Práctica 3

Búsqueda con Adversario (Juegos)

El ParCheckers

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

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

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

1.1. Struct Box

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

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

```
//Struct para definir las casillas: número de casilla, tipo y color.
struct Box
{
   //Número de casilla:
   //{1, 2, ..., 68} para casillas normales (normal)
   //{1, 2, ..., 7} para casillas del pasillo a la meta (final_queue)
   //0 en caso contrario
   int num;
   //Tipo de la casilla
   box_type type;
   //Color de la casilla
   color col;

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

   /**
    * @brief Constructor por defecto de un objeto Box
    *
    */
```

```
    inline Box(){};
};
```

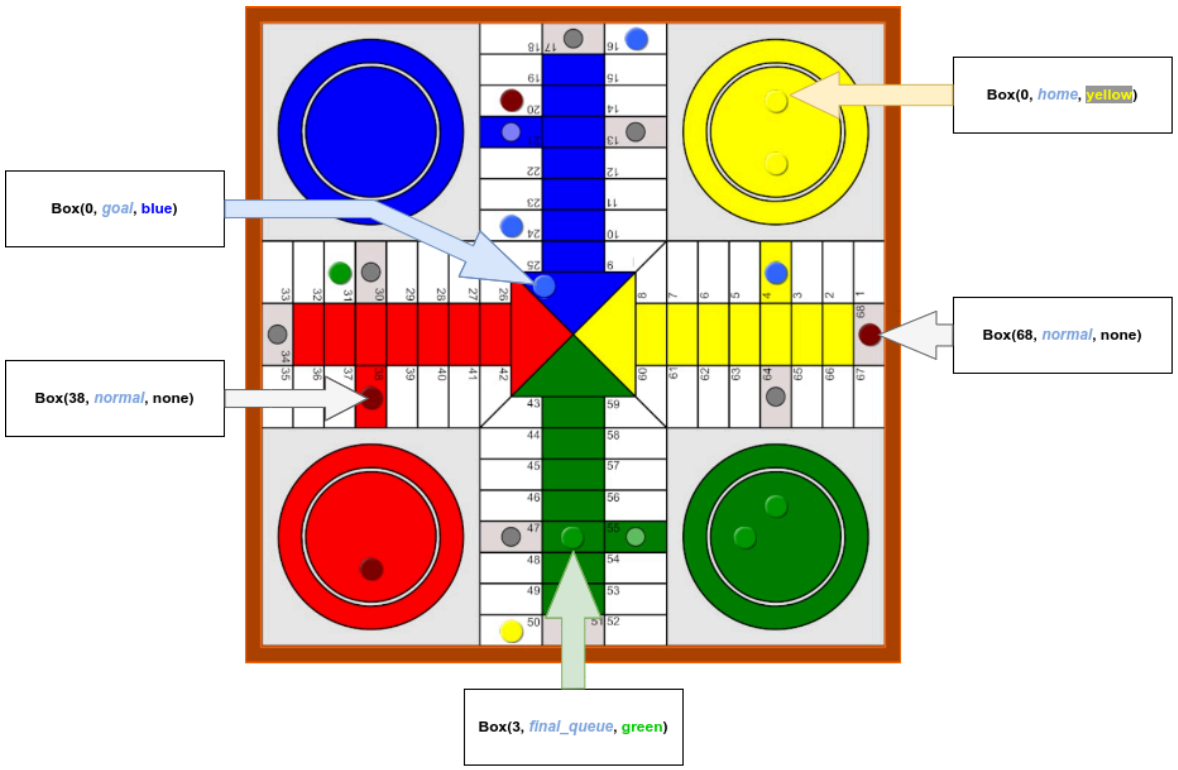
En el mismo fichero definimos los enumerados de posibles colores y tipos de casillas. Con respecto a los colores incluimos los 4 colores del parchís y el color vacío (para las casillas blancas), y el respectivo paso a string a partir de un objeto de tipo ***color.*** Finalmente, definimos en esta clase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

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

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

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

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



1.2. Clase Board

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

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

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

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

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

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

1.3. Clase Dice

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

```
class Dice{
   private:
       /**
        * @brief Dados para cada jugador. Los dados se agrupan por capas.
        *
        * - Capa 1: Dados clásicos del 1-6 (quitando el 3) + dado especial.
        * - Capa 2: Dados forzados (mover 10 o 20)
        *
        */
       map <color, vector<vector <int>> > dice;
```

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

2. Representación del juego (Clase Parchis)

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

Los atributos que definen esta clase son:

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

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

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

        //Turno actual
        int turn;

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

5. Variables auxiliares para controlar las acciones de los jugadores y los movimientos especiales.
6. Variables para almacenar las casillas especiales para cada color.

Métodos destacables de la clase Parchis

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

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

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


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

/**
* @brief Si la partida ha terminado, devuelve el color del jugador ganador.
*
* @return color
*/
color getColorWinner() const;
```

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

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

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

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

/**
* @brief Función que devuelve la distancia a la meta de la ficha identificada
* por id_piece del jugador identificado por player.
*
* La distancia se entiende como el número de casillas que hay que avanzar hasta
* la meta.
*
* @param player
* @param id_piece
* @return int
*/
int distanceToGoal(color player, int id_piece) const;
```

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

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

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

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

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


/**
* @brief Función que devuelve el vector de colores de las barreras (en caso de haberlas) del
* camino entre b1 y b2.
*
* Esto es, se va recorriendo todas las casillas que habría que recorrer para ir de b1 y b2,
* y siempre que se encuentran dos fichas de un mismo color en una misma casilla se añade ese
```

```
* color al vector que se devuelve.
*
* Por ejemplo: si en la casilla 2 hay una barrera amarilla y en la 4 una azul, el
anywalls(1,6)
* devuelve {yellow, blue}
*
* @param b1
* @param b2
* @return const vector<color>
*/
const vector<color> anyWall(const Box & b1, const Box & b2) const;
```

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

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


/**
* @brief Función que devuelve el atributo board.
*
* @param player
* @return const vector<int>&
*/
const Board & getBoard () const;
/**
* @brief Función que devuelve todas las fichas de player que pueden
* hacer un movimiento según el valor del dado dice_number.
```

```
*
* Por ejemplo, si dice_number = 2, las fichas que se encuentran en home
* no aparecerán como disponibles.
*
* También se gestionan las barreras y otros casos particulares.
*
* @param player
* @param dice_number
* @return const vector<int>&
*/
const vector<tuple<color,int>> getAvailablePieces (color player, int dice_number) const;
```

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

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

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

```
*/
bool isLegalMove(const Piece & piece, int dice_number) const;

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

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

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

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

```
}
```