

# AJEDREZ CHINO



3º Computación UCLM

Raúl Barba Rojas --- [Raul.Barba@alu.uclm.es](mailto:Raul.Barba@alu.uclm.es)

Jaime Camacho García --- [Jaime.Camacho1@alu.uclm.es](mailto:Jaime.Camacho1@alu.uclm.es)

Manuel Fernández del Campo --- [Manuel.Fernandez17@alu.uclm.es](mailto:Manuel.Fernandez17@alu.uclm.es)

Diego Guerrero del Pozo --- [Diego.Guerrero@alu.uclm.es](mailto:Diego.Guerrero@alu.uclm.es)

Ismael Pérez Nieves --- [Ismael.Perez3@alu.uclm.es](mailto:Ismael.Perez3@alu.uclm.es)

# ÍNDICE

## Contenido

<b>1. Introducción .....</b>	<b>4</b>
<b>2. Tarea 1: Espacio de estados .....</b>	<b>6</b>
<b>2.1. Representación del tablero y de las piezas .....</b>	<b>6</b>
<b>2.2. Representación humana del estado.....</b>	<b>6</b>
<b>2.3. Persistencia de los estados.....</b>	<b>9</b>
<b>3. Tarea 2: Acciones .....</b>	<b>10</b>
<b>3.1. Representación de los movimientos.....</b>	<b>10</b>
<b>3.2. Generar sucesores de un estado .....</b>	<b>11</b>
<b>3.2.1 Función sucesora .....</b>	<b>12</b>
<b>3.2.2 Representación de movimientos posibles .....</b>	<b>12</b>
<b>3.2.3 Generación de movimientos posibles de una pieza .....</b>	<b>12</b>
<b>3.3. Verificar si una jugada es correcta.....</b>	<b>16</b>
<b>4. Tarea 3: Entidades y lógica de juego.....</b>	<b>17</b>
<b>4.1. Cliente .....</b>	<b>17</b>
<b>4.2. Servidor de juego .....</b>	<b>17</b>
<b>4.3. Partida .....</b>	<b>18</b>
<b>5. Tarea 4 .....</b>	<b>19</b>
<b>5.1 Árbol de búsqueda de Montecarlo .....</b>	<b>19</b>
<b>5.2 Partidas de bot vs bot .....</b>	<b>20</b>
<b>5.2.1 Vs bot con algoritmo de Montecarlo.....</b>	<b>20</b>
<b>5.2.2 Vs bot aleatorio .....</b>	<b>20</b>
<b>5.3 Estrategias de juego.....</b>	<b>21</b>
<b>5.3.1 Descripción .....</b>	<b>21</b>
<b>5.3.2 Comparativas entre estrategias .....</b>	<b>22</b>
<b>6. Opinión personal.....</b>	<b>23</b>
<b>6.1 Raúl .....</b>	<b>23</b>
<b>6.2 Diego .....</b>	<b>23</b>
<b>6.3 Ismael.....</b>	<b>23</b>
<b>6.4 Jaime.....</b>	<b>23</b>
<b>6.5 Manuel.....</b>	<b>24</b>
<b>7. Anexo.....</b>	<b>25</b>

<b>7.1 Aspectos técnicos</b>	25
<b>7.1.1 Interfaces de usuario</b>	25
<b>7.1.2 Base de datos</b>	28
<b>7.1.3 Sockets y protocolo empleado</b>	29
<b>7.1.4 Dockers</b>	29
<b>7.1.5 Makefile y bash script</b>	30
<b>8. Referencias externas</b>	31
<b>8.1 De código</b>	31
<b>8.1.1 Pygame</b>	31
<b>8.1.2 Tkinter</b>	31
<b>8.2 De información</b>	31
<b>8.2.1 World Xiangqi Federation</b>	31
<b>8.2.2 Imágenes de las piezas</b>	31
<b>9. Manual de usuario</b>	32
<b>9.1 Instalación y ejecución de Docker</b>	32
<b>9.2 Ejecución de los archivos bash script e inicialización de la base de datos</b>	34
<b>9.3 Botones e interfaz de juego</b>	35
<b>9.4 Guía de uso</b>	36

# 1. Introducción

A modo de introducción sobre la práctica, primero tenemos que destacar las dos principales decisiones de diseño que afectan a la práctica.

- 1) Juego sobre el que se va a desarrollar la práctica.
- 2) Lenguaje de implementación de la práctica.

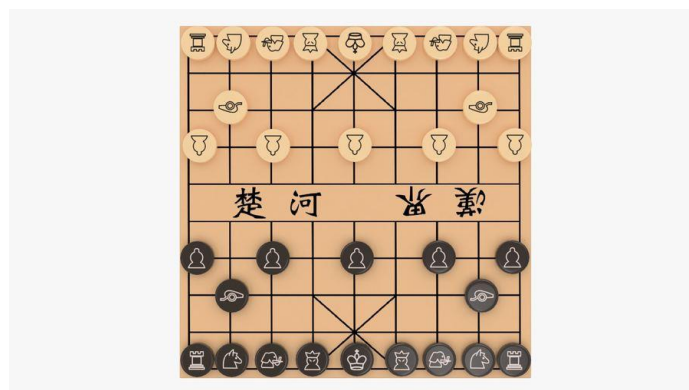
Sobre el juego, hemos decidido realizar la práctica sobre el juego conocido como "Ajedrez Chino", también conocido como Xiangqi. Un juego similar al ajedrez que tiene la diferencia de que consta de más tipos de fichas (en concreto tiene 7 tipos diferentes y no 6 como el ajedrez), y además muchas de las fichas tienen comportamientos completamente diferentes de lo habitual.

Sobre el lenguaje, hemos optado por desarrollar la práctica en Python, puesto que es un lenguaje conocido entre los miembros del grupo y disponíamos de una librería gráfica conocida para poder desarrollar una interfaz de usuario para el juego. Esa selección nos ha ahorrado cierto tiempo, ya que no hemos tenido que descubrir librerías similares para otros lenguajes (ni su funcionamiento).

Continuando con la introducción, también nos parece apropiado hacer una breve introducción al juego para que el lector pueda tener una visión más global de la práctica en sí misma.

Esta versión del ajedrez que tiene reglas muy similares al ajedrez tradicional. Disponemos de dos jugadores con 16 piezas cada uno, de color rojo o negro. Estas piezas son:

- 1 rey
- 2 consejeros (O advisor)
- 2 caballos
- 2 elefantes
- 2 torres (O rook)
- 2 cañones
- 5 peones



Al igual que en la versión occidental, la meta es forzar al otro jugador a no disponer de movimientos posibles que pudieran salvarle de que eliminemos a su rey, pero por simplicidad, en nuestra versión bastará solo con comerse al rey.

Cada una de las piezas puede moverse de diferente forma, lo que explicaremos luego en posteriores apartados. Respecto al tablero, a diferencia del ajedrez que conocemos, este posee cuatro diferencias notorias:

- La existencia de una zona llamada río en medio del tablero, lo que afectará a la forma de moverse del elefante o el peón.
- La zona de 3x3 donde comienzan el rey y los consejeros, llamada palacio, que prohibirá a estos dos tipos de piezas de moverse por otras zonas.
- Las piezas se desplazan sobre las líneas y no sobre las casillas.
- El tablero es de 10x9 y no de 8x8.

Además, el Xiangqi posee un par de reglas más para forzar a los jugadores a ser activos y no reactivos, pero que no hemos implementado para mayor comodidad. Otras reglas del ajedrez tradicional, como el enroque, no existen en esta variante, lo que simplifica todo.

## ***2. Tarea 1: Espacio de estados***

En este punto nos gustaría hablar principalmente de las 3 subtareas que engloban la tarea en sí misma, de modo que hemos tratado cada una de estas tareas por separado.

### ***2.1. Representación del tablero y de las piezas***

Primero tenemos que hablar de cómo vamos a representar internamente el tablero.

Inicialmente, pensamos una representación interna por medio de listas anidadas (2 niveles de profundidad), lo que nos permitiría un almacenamiento intuitivo de la información para manejarla, sin embargo, pensamos que esta opción no era la mejor de todas puesto que implicaría una mayor complejidad algorítmica (recorrer la matriz de posiciones) para poder elegir los movimientos (si bien es cierto que en esta tarea no los consideramos, es necesario desarrollar la práctica mirando hacia el futuro, y esto podría evitar una posterior optimización del código).

De modo que decidimos implementarlo a través de un diccionario de Python que actuará como una tabla hash, que contendrá 2 claves: 'w' y 'b' (posteriormente explicaremos esta notación). Y cada clave tiene como valor una lista con sus piezas con vida, ya que, en este juego, ninguna pieza muerta puede “resucitar” como si se puede en otros similares como el ajedrez normal.

Esta implementación nos asegura una complejidad lineal a la hora de determinar qué fichas podemos mover, lo que hará que el programa sea más eficiente.

Respecto a la representación de las piezas, puesto que todas ellas comparten ciertos atributos, hemos optado por crear una clase genérica, llamada 'Piece', de la cual heredan otras 7 clases, cada una para un tipo de pieza.

Dicha clase genérica posee un atributo para designar su color, que será 'b' para negro y 'w' para blanco (Aunque en la práctica utilizaremos el rojo para la representación), y otro atributo para la posición de la pieza, de la forma (fila, columna). Además, esta clase posee un método, `get_representation`, que devuelve un carácter representando la pieza, y que tendrá una implementación diferente para cada una de las clases hija.

Todos los valores mencionados anteriormente se almacenarán en constantes, del mismo modo que el número de piezas por tipo que existen al comienzo de una partida.

### ***2.2. Representación humana del estado***

Sobre la representación del estado, primero es necesario hablar sobre la notación estandarizada que seguiremos para representar los estados. Se trata de la representación llamada “Forsyth-Edwards Notation”, más conocida como FEN, en este caso, aplicada al ajedrez chino:

En general, una representación FEN contiene los siguientes campos (separados por un espacio en blanco) dispuestos como una cadena de caracteres:

<Piece Placement> <Side to move> <Castling ability> <En passant target square> <Halfmove clock>  
<Fullmove counter>

Como podemos ver, existen 6 campos y cada uno aporta información distinta y necesaria sobre el estado del juego:

1). **“Piece Placement”**: Es la parte principal de la representación y muestra en una cadena como es el estado del tablero. Para ello, separa las filas por medio del carácter “/”, y cada fila contiene una combinación de ciertas letras y números. Las letras que podemos encontrar son las siguientes:

- r: Rook (Torre)
- h: Horse (Caballo)
- e: Elephant (Elefante)
- a: Advisor (Oficial/Consejero)
- k: King (Rey/General)
- c: Cannon (Cañón)
- p: Pawn (Peón)

Si la letra es minúscula, entonces se trata de una ficha negra, mientras que, si es mayúscula, se trata de una ficha blanca/roja, dependiendo de qué representación visual se esté usando.

Por otro lado, también podemos encontrarnos números, en concreto, cada número refleja el número de espacios en blanco entre ficha y ficha. La siguiente cadena es un ejemplo de “Piece Placement” correcto en una representación FEN:

rheakaehr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RHEAKAEHR

2). **“Side to move”**: puede tomar solo 2 valores: ‘w’ y ‘b’, que representa quién mueve en dicho turno. ‘w’ representa white, por lo que sería el turno en el que mueven las piezas blancas/rojas, según la representación del juego. Por otro lado, ‘b’ representa black, por lo que sería el turno en que mueven las piezas negras.

3). **“Castling ability”**: este campo no se utiliza en este juego. No hay un estándar concreto sobre cómo representar que no se usa (algunos autores simplemente eliminan el campo, otros optan por marcarlo con el carácter ‘-’. Nosotros hemos optado por esa última representación, intentando mantener lo máximo posible la filosofía de esta representación.

4). **“En passant target square”**: lo mismo afirmado en el campo anterior aplica también a este campo. También será representado como ‘-’.

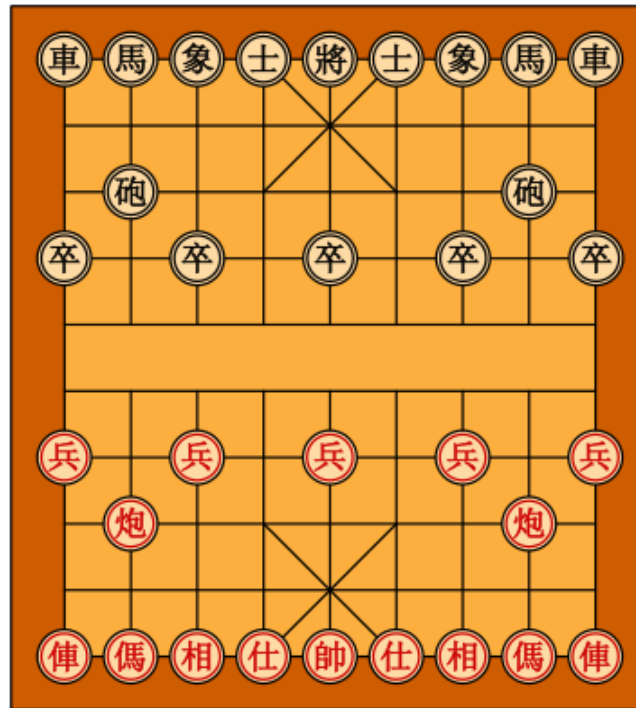
5). **“Halfmove clock”**: es un contador que implementa la regla de los “50-movimientos” (50-move-rule). Según dicha regla, si después de 50 movimientos de cada jugador (100 en total), ninguna ficha ha matado a otra (el número de fichas es el mismo) y ningún peón ha avanzado ninguna casilla hacia adelante, entonces la partida acaba en tablas. Cualquier movimiento hacia adelante de un peón o cualquier movimiento que elimine a una ficha reiniciará el contador.

6). **“Fullmove counter”**: es un contador de turnos del juego, que comienza en 1 y acaba en el turno en el que el juego se da por finalizado.

De esta forma, una representación FEN válida y completa sería la siguiente:

rheakaehr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RHEAKAEHR w - - 0 1

En concreto, esta representación muestra la situación inicial del juego, que se puede ver en la siguiente imagen:



Con respecto a la representación gráfica, todas las imágenes de las fichas que tenemos las hemos sacado de la página “[https://www.chessprogramming.org/Chinese\\_Chess#Pieces](https://www.chessprogramming.org/Chinese_Chess#Pieces)”. A partir de estas imágenes de 400x100, nos hemos quedado con las piezas occidentales ya que nos resulta más intuitivo de ver. El procesamiento de las imágenes las hemos realizado con GIMP 2.0, un programa de edición de imagen en el que recortamos y añadimos fondo transparente a las fichas.

Las fichas se quedaron con las dimensiones de 100x100, y en la página web de [Befunky](#) hemos ido reescalando las imágenes para que, a la hora de mostrarlas por pantalla, se viesen bien. Las resoluciones finales son:

- **Tablero:** 600 x 493 px.
- **Espacio entre casillas:** 50 px.
- **Fichas:** 40x40 px.

Este módulo coge como entrada la representación del estado en formato FEN para representarlo. Vamos leyendo línea a línea las piezas que hay en cada fila e ir las pintando en la pantalla. Esto lo hacemos con las funciones que tiene la librería de **pygame**, que nos deja cargar imágenes desde nuestros ordenadores para luego mostrarlas. Para un mejor acceso, hemos decidido usar diccionarios como la estructura de datos que guardaría las rutas de las imágenes.

Para saber la posición en la que hay que pintar las fichas, la fórmula es la siguiente:



- Para la posición X, hay que sumarle 23 píxeles debido al marco del tablero, restarle 20 que es la mitad del ancho de la ficha para que esté centrada y luego sumar el resultado de multiplicar la columna que le corresponde a esa pieza por 48,8 (hubiera sido 50 debido a la supuesta distancia entre casillas, pero en algún momento del procesamiento de las imágenes las casillas dejaron de ser perfectamente cuadradas).
- Para la posición Y, hay que sumarle 20 píxeles debido al marco del tablero, restarle 20 que es la mitad del alto de la ficha para que esté centrada y luego sumar el resultado de multiplicar la fila que le corresponde a esa pieza por 50.

También hemos implementado un mecanismo de reconocimiento de clicks que nos permite implementar todas las funcionalidades necesarias para el juego. Gracias a pygame, podemos detectar los eventos que consisten en un click izquierdo, que, tras comprobar que ha sido en la zona del tablero, obtenemos sus coordenadas.

Mediante una serie de operaciones matemáticas y ajustes, podemos calcular el centro de la posición donde el usuario hizo click, siendo la fila 0 y la columna 0 la esquina superior izquierda y 10, 9, respectivamente, para la esquina inferior derecha. Ahora podemos calcular un cuadrado de dimensiones 40x40 (Las mismas que las de la ficha), con punto medio en el centro mencionado anteriormente. Dicho cuadrado imaginario representará la zona de interacción de una posición válida de una ficha. Finalmente, se comprueba que el click del usuario entre dentro del cuadrado descrito antes.

## ***2.3. Persistencia de los estados***

En primera instancia hemos creado un directorio “data” por defecto en el cual se irán almacenando progresivamente cada uno de los archivos del programa que se vayan exportando (a tipo json), un archivo por cada uno de los movimientos que se realizan a lo largo de la partida. Para poder realizar esta operación de exportación, se ha definido un método que se encargará de recibir el archivo en un formato predefinido y exportarlo así a un archivo .json. Esto se realizará para todos y cada uno de los movimientos que pueda albergar una partida. Para finalizar se ha testeado con una serie de ejemplos para así poder comprobar la funcionalidad desarrollada de los métodos creados.

## 3. Tarea 2: Acciones

Para la tarea 2, el trabajo se descompuso, principalmente, en 4 diferentes tareas que nos permiten alcanzar el principal objetivo: poder hacer movimientos, saber si hemos ganado, perdido o empatado, y representar todos esos movimientos con alguna representación, que será especificada posteriormente.

### 3.1. Representación de los movimientos

Para la representación de los movimientos también hemos optado por utilizar una representación estandarizada, en este caso, es la representación de movimientos de la AXF (Asian Xiangqi Federation), y se basa en la idea de que un movimiento está representado por 4 valores diferentes separados por un espacio en blanco:

- El primero simboliza la pieza movida (K para rey, C para cañón, E para elefante...), siempre en mayúscula.
- El segundo, muestra la columna donde se encuentra la pieza, que no son equivalentes para ambos jugadores, es decir: la columna 7 para las negras sería la 3 para las rojas. Para las negras las columnas van de izquierda a derecha, y al revés para las rojas.
- El tercero indica el tipo de movimiento (+ para movimientos hacia delante, = para laterales, - si se mueve hacia atrás).
- El cuarto puede tener dos lecturas. Para el elefante, el caballo y el advisor implica la nueva columna a la que se mueven, puesto que estas piezas siempre implican un movimiento lateral, mientras que para las otras piezas puede indicar la nueva columna, pero en caso de que no sea un movimiento lateral, alude al número de filas que avanzan/retroceden.

Por ejemplo, el movimiento frontal de un peón en la columna 1 de las negras se anota como:

P 1 + 1

Y el movimiento lateral de un cañón de la columna 2 a la 5 de las rojas se anota como:

C 2 = 5

Además, el primer movimiento siempre corresponde a las rojas.

También existen ciertos casos especiales que se denominan casos tándem, que ocurren cuando tenemos dos piezas del mismo tipo en la misma columna. Cuando esto ocurre, el valor de la columna cambiará a + o - según si es la de delante o la de detrás, puesto que la representación anterior sería ambigua y no seríamos capaces de saber qué ficha realmente queremos mover. Para comprenderlo mejor: supongamos que tenemos dos cañones en la columna 4 y movemos el de delante a la columna 6, sería tal que:

C + = 6

Y si moviésemos el de detrás:

C - = 6

La explicación anterior es buena para las diferentes fichas, excepto para el rey que nunca podrá tener tándem (solo hay una ficha de este tipo en cada color), y para el peón, puesto que podría, potencialmente,

haber 5 peones en la misma columna. Los casos especiales de tandems para peones se definen de la siguiente manera:

**Tándem de 3 peones:** el peón delantero se denota como P +, el trasero como P -, mientras que el del medio conserva la notación usual. Si tuviésemos 3 peones en la columna 3 y moviésemos uno a la columna 4 sería:

P + = 4, para el delantero.

P 3 = 4, para el del medio.

P - = 4, para el de detrás

Algo a destacar es que el hecho de “estar delante” o “estar detrás” es algo que depende completamente del tipo de pieza. Para una ficha roja/blanca, estar delante será estar en filas con valor numérico más bajo, mientras que para las negras es justo al revés. Es fácil comprenderlo si vemos el tablero como una matriz que empieza en la posición (0,0) y termina en (9,8).

**Tándem de 5 peones:** el primer peón se denota como + +, y el último como - -. El segundo utilizaría P +, el cuarto P -, y el del medio la notación usual. Utilizando el ejemplo anterior:

+ + = 4, para el primero.

P + = 4, para el segundo.

P 3 = 4, para el del medio.

P - = 4, para el cuarto.

- - = 4, para el último.

**Tándem de 4 peones:** es igual que con 5 peones, pero sin utilizar la notación habitual para ninguno de ellos, solo + +, P +, P - y - -.

**Tándem de peones en dos columnas diferentes:** ahora, el primer valor de la notación indica la columna. Supongamos que tenemos 3 peones en la columna 3 y 2 en la columna 6. Moveríamos los de la columna 3 a la 2 o los de la 6 a la 7 para este ejemplo:

3 + = 2, para el primer peón de la columna 3.

3 3 = 2, para el peón del medio de la columna 3.

3 - = 2, para el último peón de la columna 3.

6 + = 7, para el peón delantero de la columna 6.

6 - = 7, para el peón trasero de la columna 6.

## ***3.2. Generar sucesores de un estado***

La representación de sucesores de un estado se realiza gracias a una función llamada “get\_possible\_movements()”, que está implementada de forma general en la clase Piece, y heredada por el resto de clases utilizadas por las piezas. Esta función, dado el tablero, retorna una lista de duplas (fila, columna) representando las posiciones hacia dónde la pieza que llama al método puede moverse.

Gracias a esto, podemos generar todos los estados sucesores a otro dado teniendo en cuenta que jugador tiene el turno, que piezas puede mover y hacia dónde.

### 3.2.1 Función sucesora

Esta función tiene de nombre “get\_sucesores”, está implementada en la clase “Board” y retorna un vector el cual contendrá los posibles sucesores de un tablero. Los sucesores obtenidos serán los del “side\_to\_move” del tablero en el cual se llame a la función, dichos sucesores se crean gracias a la función explicada en el apartado anterior “get\_possible\_movements()”, ya que con esta se obtendrán todos los movimientos posibles de cada pieza y por cada movimiento posible se genera un tablero, obteniendo así el vector dicho anteriormente.

### 3.2.2 Representación de movimientos posibles

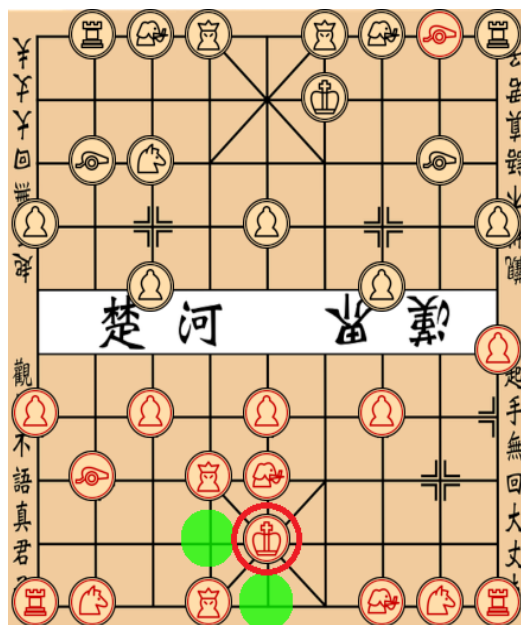
Para una mayor ayuda tanto al desarrollo del juego como a la propia partida, hemos implementado un mecanismo de detección de clicks que, dado el tablero, devuelve la pieza que se encuentra en la posición del clic, ayudándose del mecanismo de hitboxes implementado anteriormente.

Además, al elegir una pieza, se mostrarán las posiciones a donde esta puede moverse, que será útil para implementar la función sucesora.

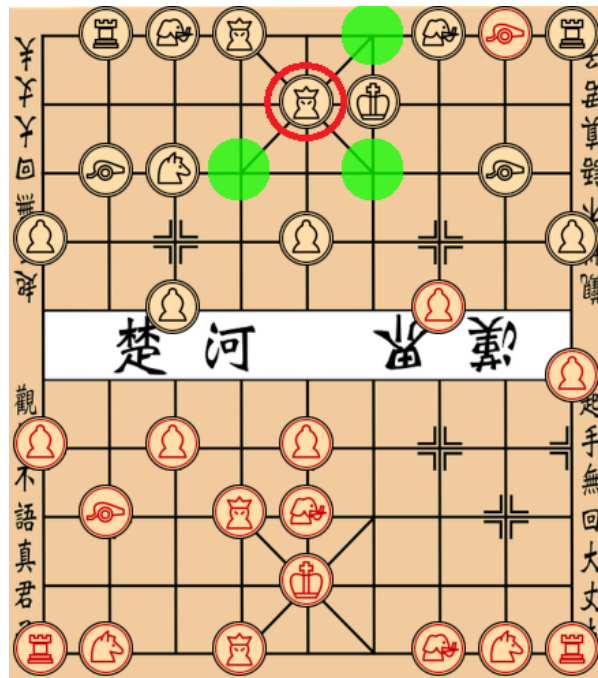
### 3.2.3 Generación de movimientos posibles de una pieza

Como hemos mencionado anteriormente, disponemos en la clase Piece de la función “get\_possible\_movements()”, que además tendrá una implementación distinta para cada tipo de pieza, que veremos en este apartado.

**Rey:** tiene un funcionamiento muy similar al rey del ajedrez tradicional con la excepción de que en el ajedrez chino el rey no puede moverse en diagonal. Hemos generado los 4 “posibles” movimientos que el rey tiene desde una posición dada. Se ha tenido en cuenta eliminar los movimientos en los que el rey pudiera salir del palacio ya que al igual que el consejero sólo puede moverse dentro de un número determinado de casillas denominadas en su conjunto palacio. También hemos eliminado movimientos en los que el rey quedase expuesto a ser capturado por otra pieza.

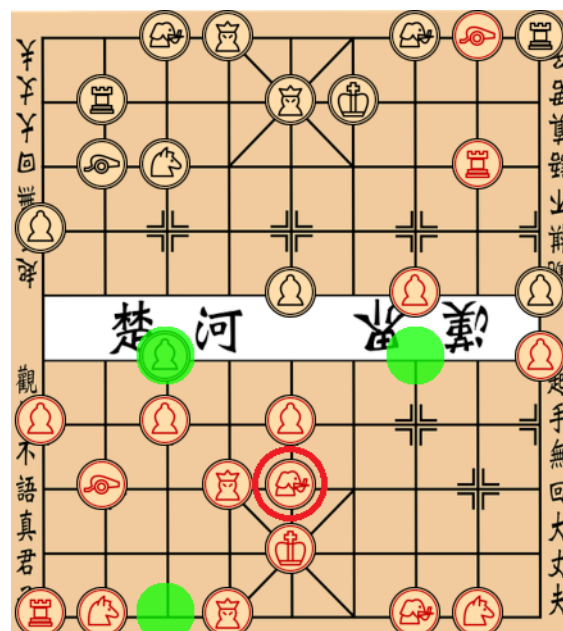


**Consejero:** La generación de los posibles movimientos del consejero, que es una pieza que no se encuentra en el ajedrez tradicional pero que podría compararse con el alfil, hemos seguido un procedimiento muy similar al del rey. El caso es que el consejero sólo puede moverse en diagonal y dentro del palacio.



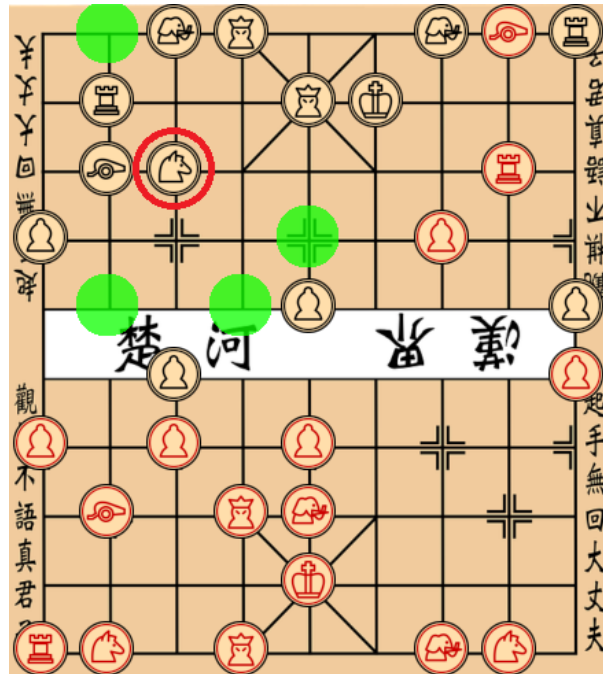
Tanto rey como consejero solo pueden mover de 1 casilla en 1 casilla.

**Elefante:** El elefante solo se puede mover dos casillas en diagonal en cualquier dirección desde su posición inicial, sin pasar nunca el río. Para esto simplemente hemos generado 4 movimientos (NW, NE, SW, SE), modificado el método de detectar si una ficha estaba en campo rival para que aceptase como entrada una posición y poder quitar los movimientos que acaban al otro lado del río y usar el método general para quitar los movimientos imposibles.

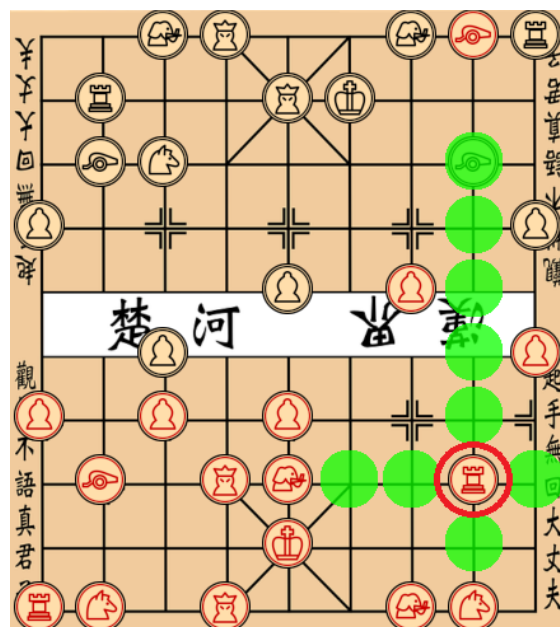


**Caballo:** igual que el caballo del ajedrez tradicional, puede moverse en forma de L, por lo que haremos será generar las posiciones correspondientes a las 8 casillas a las que teóricamente puede desplazarse. Tras esto, debemos tener en cuenta las posiciones a las que no podemos avanzar, ya que el punto donde pivotamos está ocupado.

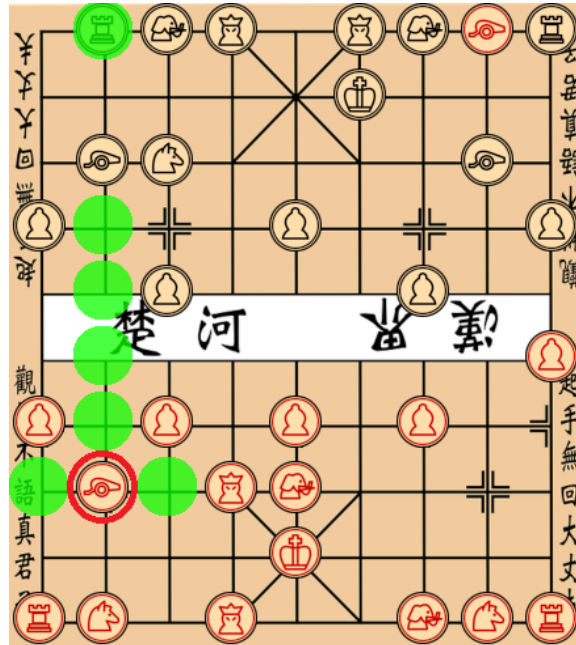
Finalmente, como en otros casos, eliminaremos las posiciones que se encuentren fuera del tablero, así como las que estén ocupadas por piezas del mismo equipo.



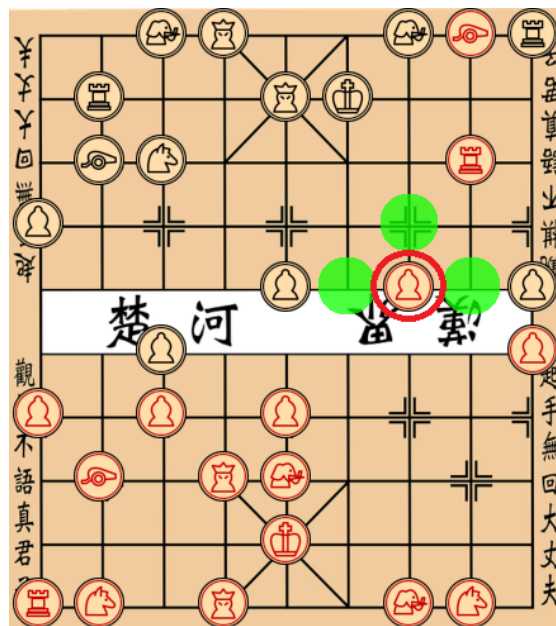
**Torre:** El chariot o torre se mueve como su homónimo del ajedrez tradicional, avanzado cuantas casillas sea posible tanto en vertical como en horizontal, pero no en diagonal. Simplemente hemos generado todos sus movimientos ajustando las piezas que tenía enfrente, debajo, a la izquierda o abajo, para limitar correctamente el movimiento.



**Cañón:** esta pieza es la más compleja, ya que su movimiento es similar al de la torre, pero sin poder retroceder; es decir, hay que calcular el número de posiciones que se pueden mover en cada dirección, así como la siguiente pieza detrás de estas, es decir: la pieza que teóricamente nos podríamos comer al saltar la pieza previa. Esto supondrá una complejidad de  $O(n)$ .



**Peón:** El peón se puede mover hacia delante en cualquier momento, y hacia la izquierda y a la derecha cuando cruza el río. Se añade siempre a la lista de posibles movimientos el movimiento hacia delante, con una condición se comprueba si la ficha está en campo rival y si es así se añaden los movimientos laterales y por último se utiliza el método general para quitar los movimientos imposibles.



### ***3.3. Verificar si una jugada es correcta***

“validarMovimiento” es el nombre del método encargado de verificar si un movimiento del rival ha sido válido o no, esto es gracias a la comparación del tablero mandado y el tablero recibido, ya que se comparan las piezas del rival en ambas situaciones. Esta comparación es realizada para saber cuál de las piezas ha cambiado su posición, una localizada la pieza en ambos tableros, se comprobará que la posición en el tablero recibido es una de las opciones de la pieza en el tablero enviado.



## 4. Tarea 3: Entidades y lógica de juego

En esta sección se implementaron las principales entidades que permiten simular el juego, por ello, dividiremos la sección en 5 partes diferentes:

- **Cliente**: es la entidad que, con ciertos datos asociados (como el nombre de usuario, contraseña y correo electrónico), jugará partidas del juego objeto de la práctica.
- **Servidor de juego**: es la entidad que permite gestionar usuarios y partidas.
- **Partida**: es la entidad que permite simular cada partida entre usuarios y/o agentes inteligentes.

### 4.1. Cliente

El cliente es usado como el intérprete del usuario, que luego comunicará sus acciones al servidor. Su funcionalidad se limita a pedirle al usuario información para luego pasársela al servidor en forma de mensaje, y es este el que se encarga de procesar la petición. Las funcionalidades las explicaremos en el siguiente punto, y las interfaces en sí las explicamos más adelante en el apartado 7.1.1.

### 4.2. Servidor de juego

Para implementar el servidor de juego, primero es necesario definir las funcionalidades que conforman la interfaz que proporciona y define el servidor de juego. Dichas funcionalidades son las siguientes:

Funcionalidad	Método asociado	Descripción
Crear partida	create_game	Es la funcionalidad que permite, bajo demanda de algún usuario, crear una partida. La partida será ejecutada en un hilo del servidor y será gestionada por la entidad Partida.
Obtener partidas	get_unprepared_games	Es la funcionalidad que permite, bajo demanda de algún usuario, obtener una lista de partidas que no se estén jugando todavía.
Añadir usuario a partida	add_player_to_game	Es la funcionalidad que permite, bajo demanda de un usuario, añadir un usuario a una partida que no se esté jugando todavía.
Añadir usuario	add_user	Es la funcionalidad que permite, bajo demanda de un usuario reconocido, añadir un nuevo usuario a la base de datos.

Eliminar usuario	remove_user	Es la funcionalidad que permite, bajo demanda de un usuario reconocido, eliminar un usuario de la base de datos.
Actualizar usuario	update_user	Es la funcionalidad que permite, bajo demanda de un usuario reconocido, actualizar un usuario de la base de datos.
Acceso a la aplicación	Login	Es la funcionalidad que permite, bajo demanda de un usuario no reconocido, acceder a la aplicación como usuario reconocido.

Para implementar dichas funcionalidades, recomendamos una lectura al Anexo I, concretamente la parte de arquitectura de sockets, puesto que el servidor de juego atiende peticiones bajo demanda de usuarios a través de sockets, que es la abstracción utilizada para la comunicación interproceso en nuestra implementación.

### 4.3. Partida

Esta entidad es responsable de gestionar la ejecución de la partida una vez que hay dos jugadores, sean reales o agentes inteligentes,

La entidad de la partida sigue el siguiente pseudocódigo:

```
def jugar_partida(jugador1, jugador2):
    partida_terminada = False
    mientras partida_terminada = False:
        mostrar_tablero(jugador1, jugador2, partida.tablero)
        movimiento = pedir_movimiento(partida.tablero, jugador1, jugador2)
        while not validar_movimiento(movimiento, partida.tablero):
            movimiento = enviar_mensaje_error_y_movimiento(tablero, j1, j2)
        si partida.es_final(partida.tablero):
            dar_ganador(jugador1, jugador2, partida.tablero)
```

Básicamente, cada la entidad partida ejecuta en un bucle hasta que se termine la partida diferentes acciones:

- Muestra el estado inicial.
- Pide un movimiento al jugador correspondiente, esto es, al jugador que tenga que mover con la disposición del tablero actual.
- Intenta validar el movimiento, si no es válido requerirá un movimiento válido hasta que reciba uno, cuando lo reciba, simplemente actualizará el tablero.
- Comprueba si la partida es final o no, en caso afirmativo manda el ganador a ambos jugadores, sino se vuelve al paso inicial del algoritmo.

## 5. Tarea 4

La cuarta tarea tuvo como principal objetivo la implementación del agente inteligente que nos permitirá establecer partidas de un humano contra un agente inteligente, o incluso partidas entre dos agentes inteligentes.

Para ello, fue necesario implementar un árbol de búsqueda de Montecarlo, que es lo que permite al agente inteligente escoger los movimientos, así como una extensión de la interfaz gráfica de usuario para poder jugar contra agente inteligentes, o espectral partidas entre agentes inteligentes.

### 5.1 Árbol de búsqueda de Montecarlo

En esta subsección se va a describir el algoritmo de Montecarlo utilizado, si bien no se va a profundizar en la función utilizada para calcular el valor UCT asociado a cada nodo, puesto que quedará definido en la sección 5.3.

Para la implementación del árbol de búsqueda de Montecarlo se decidió crear una clase nodo, que nos permitió implementar el árbol de búsqueda de Montecarlo.

Node
<code>board : Board explored_successors : dict parent : Node times_visited : int unexplored_successors win_record : list</code>
<code>MCTS() backpropagation(value) calculate_node_uct() estimate_simulation_iterations(node, repetitions) expand_node() is_terminal_node() random_selection() select_node() simulation()</code>

La clase nodo contiene ciertas variables a nivel de clase que permiten implementar una búsqueda de Montecarlo. Concretamente, tienen el siguiente significado:

- **board:** es el objeto tablero, que contiene el estado asociado al nodo.
- **explored successors:** es un diccionario cuyas claves son los movimientos ya explorados relacionados con el nodo actual, y cuyos valores son los nodos hijos a los que se llega tras realizar ese movimiento sobre el tablero del nodo actual.
- **parent:** es el nodo padre del nodo actual. Si se trata del nodo raíz, su valor será nulo, mientras que para cualquier otro nodo, su valor será otro nodo del árbol.
- **times visited:** es un entero que almacena la cantidad de veces que se ha visitado el nodo durante la ejecución del algoritmo que selecciona la mejor jugada posible. Inicialmente el valor es 0, y se va incrementando conforme este nodo es visitado a lo largo de la ejecución de la búsqueda de Montecarlo.
- **unexplored successors:** es una lista de sucesores sin expandir, contiene los movimientos que todavía no han sido utilizados para crear nodos hijos.

- **win\_record**: es un vector de enteros con 3 posiciones. La primera refleja el número de empates, la segunda el número de victorias del jugador blanco/rojo y la tercera refleja el número de victorias del jugador negro. Es actualizado en los nodos por los que la simulación pasó, mediante el método `backpropagation()::Node`.

Además, la clase nodo utiliza diferentes métodos para llevar a cabo la búsqueda de Montecarlo:

- **MCTS()**: es el método que lleva a cabo la búsqueda de Montecarlo hasta que se consumen los recursos establecidos, definidos como valores constantes del módulo Python asociado.
- **select\_node()**: es el método que implementa la primera etapa de la búsqueda de Montecarlo. Selecciona, de forma iterativa (para mayor eficiencia) un nodo que es o bien terminal, o bien un nodo que no tiene todos sus hijos expandidos. Para hacer la selección se utiliza el método auxiliar `calculate_node_uct()`
- **expand\_node()**: es el método que implementa la segunda etapa de la búsqueda de Montecarlo. Si el nodo a expandir es terminal, se retorna ese mismo nodo, si no es terminal entonces se retorna el nodo hijo expandido.
- **simulation()**: es el método que simula una partida aleatoria desde el nodo expandido hasta que se llegue a un nodo terminal, o bien se superen las 200 iteraciones. Dicho número fue fijado en base a un intervalo de confianza calculado con el método auxiliar `estimate_simulation`. Dicho método proporciona el intervalo de confianza con 95% de confiabilidad sobre el número de movimientos hasta que termina una partida.
- **backpropagation(value)**: es el método que implementa la cuarta etapa de la búsqueda de Montecarlo. Su principal objetivo es actualizar los valores de nodos visitados, así como las victorias de todos los nodos implicados en la simulación.

Mediante estos métodos, y algunos métodos auxiliares, se implementó la búsqueda de Montecarlo, que es lo que permite al agente inteligente determinar qué movimiento debe jugar en cada situación del tablero.

## ***5.2 Partidas de bot vs bot***

### ***5.2.1 Vs bot con algoritmo de Montecarlo***

Para dar la posibilidad de jugar partidas contra un **bot** que aplique el algoritmo previamente mencionado, ha sido necesario sustituir el tablero del segundo jugador por un bot; que en nuestro caso corresponde a un objeto de la clase Bot. Este bot simula el comportamiento de un usuario normal en la forma en la que su socket recibe y envía mensajes, pero, en lugar de poder elegir manualmente un movimiento, se crea un nodo para que, mediante el algoritmo de Montecarlo, genere un movimiento. Este paradigma se ha aplicado al poner dos bots enfrentándose entre ellos, tras modificar la clase que posee el tablero para poder visualizar correctamente las partidas entre bots.

### ***5.2.2 Vs bot aleatorio***

En este caso, ha sido necesario la creación e implementación de un bot, similar al anterior, pero al que llamaremos ahora **custom bot** o bot personalizado, que tendrá un atributo llamado `strategy`, que nos permitirá elegir entre hacer un movimiento aleatorio u otras estrategias, explicadas posteriormente.

Con ello, se ha automatizado un sistema de simulación de 100 partidas entre un bot aleatorio y uno que emplee la estrategia designada por nosotros; en un módulo Python llamado **automatic\_simulator**, módulo indispensable para el siguiente apartado.

### 5.3 Estrategias de juego

Esta sección pretende explicar las diferentes estrategias que puede utilizar nuestro agente inteligente, siguiendo el árbol de búsqueda de Montecarlo, pero cambiando parte de la función que se utiliza en el UCT, concretamente, la parte asociada a las victorias, para determinar la estrategia que mejor se adapta al ajedrez chino.

Estrategia	Número de partidas	Número de victorias	Número de derrotas	Porcentaje de victorias	Número promedio de turnos
$(2 \cdot \text{wins} - 4 \cdot \text{losses} + \text{ties})/n + \text{R.UCT}$	100	96	4	96%	52
$\exp(\text{wins})/\text{total} + \text{R.UCT}$	107	105	2	98'13%	40
$\text{wins} / n$	100	96	4	96%	41
$\exp(\text{wins})/\text{total}$	100	98	2	98%	40

**NOTA:** en el campo ESTRATEGIA solo se ha mostrado el sumando asociado a las victorias, y se escribió R.UCT indicando el resto de la fórmula asociada al UCT (el sumando que contiene la constante multiplicando a la raíz).

#### 5.3.1 Descripción

Las estrategias que se proponen son algunos ejemplos de modificaciones de la ecuación original de UCT, que resultó ser ineficiente para resolver este problema. De esta forma proponemos algunas posibles ecuaciones modificadas, junto con la tabla anterior para intentar demostrar cuál es mejor.

1. La primera estrategia de la tabla pretende ser una estrategia en la que al agente le dan igual los empates, puesto que solo busca victorias, de la forma más “segura” posible. Es por ello que recibe una penalización de  $\cdot(-4)$  por cada derrota. Así, el agente preferirá nodos que ganen a aquellos que empaten, y preferirá los que empaten a aquellos que pierdan. No deja de ser una modificación de la ecuación original, pero multiplicando por un factor de amplitud, para obtener el resultado.
2. La segunda estrategia de la tabla pretende ser una función exponencial, para que las victorias tengan mucho peso. De esta forma, el agente buscará elegir el nodo que tenga más victorias, teniendo en cuenta que hay un sumando asociado al UCT (el de la raíz), que podría hacerle explorar otros nodos.
3. La tercera estrategia es, simplemente, una estrategia greedy, que elimina el sumando característico de UCT (el de la raíz), para escoger siempre los nodos que tengan más victorias.

De esta forma, queremos comparar si realmente UCT nos está aportando algo, o si una estrategia greedy podría ser mejor.

4. La cuarta estrategia es una estrategia greedy basada en una exponencial, de forma que podamos comparar entre ambas estrategias greedy, y también con las estrategias derivadas de UCT.

Estas estrategias fueron implementadas en un módulo Python para mejorar la mantenibilidad del código. De esa forma, si en algún momento se quieren eliminar o añadir estrategias se puede hacer fácilmente. Y, además, la implementación del árbol de búsqueda de Montecarlo está programada de forma que pueda utilizar cualquiera de las funciones del módulo, para que se pueda probar cualquier estrategia sin modificar líneas de código.

### ***5.3.2 Comparativas entre estrategias***

Una vez hemos obtenido estos datos, podemos desarrollar un análisis básico para determinar cuál podría ser la mejor estrategia para utilizar. Resulta conveniente tomar como referencia los datos del porcentaje de victorias, y también del número promedio de turnos.

Con estos parámetros, podemos determinar que tanto la primera como la tercera estrategia son ligeramente peores, ambas bajando del 98% de victorias de sus competidoras; y teniendo un número promedio de turnos más elevado que los 40 de sus rivales.

Respecto a la diferencia del porcentaje de victorias entre las dos potenciales ganadoras, se debe al número de partidas jugadas, lo que no nos supone un problema, ya que la simulación de un mayor número de estas nos hace pensar que no serán muy dispares. Por tanto, a la hora de escoger alguna de ellas, elegiremos la segunda, puesto que es más cercana a la fórmula UCT tradicional, que tiene en cuenta al nodo padre.

Por último, resulta interesante destacar que la muestra no es excesivamente grande, por lo que alguna de estas estrategias puede resultar bastante mejor/peor a largo plazo, pero por ello también decidimos tener en cuenta el número medio de turnos por partida.

## ***6. Opinión personal***

### ***6.1 Raúl***

En mi opinión, esta práctica ha sido realmente interesante, porque ha combinado varios aspectos técnicos, como podría ser la utilización de sockets, la creación de interfaces gráficas... junto con otros aspectos puramente algorítmicos, como la búsqueda de Montecarlo. Esa combinación ha sido bastante interesante y positiva, al menos desde mi punto de vista.

Por otro lado, también me parece muy interesante la práctica, porque es realmente satisfactorio ver el resultado de la práctica en funcionamiento, tras el esfuerzo que requirió crear el juego en sí.

En definitiva, me pareció bastante interesante, sobre todo porque al coger un juego “difícil” o que no conocíamos, era como un reto, y como dije antes, es muy satisfactorio ver el resultado final, después de todo el trabajo en equipo necesario para desarrollar el juego.

### ***6.2 Diego***

En mi opinión, esta práctica ha resultado bastante interesante para mí, pues implica el desarrollo prácticamente total de un juego de tablero, además de tener un mayor conocimiento de cómo funcionan los agentes en este tipo de juegos. También me ha gustado la parte de comparar diferentes estrategias para ver cómo el bot funciona y reacciona a diferentes movimientos y tácticas.

Por otra parte, ha requerido de nosotros una mayor capacidad de trabajo en equipo y coordinación; así como del uso de herramientas colaborativas como GitHub, que, aunque ya había usado, he tenido que aprender más opciones aún para este trabajo.

### ***6.3 Ismael***

Este trabajo ha sido muy interesante y educativo. Habíamos hecho otros trabajos en los que teníamos que simular el funcionamiento de un equipo de desarrollo de software coordinándose con GitHub, pero nunca hemos tenido que desarrollar un proyecto de verdad. Realizando este trabajo he podido sentir cómo funciona el desarrollo de un proyecto de software, desde las primeras fases de documentación hasta fases más avanzadas en las que la división del trabajo y la coordinación son fundamentales.

Además, la implementación de un agente que toma decisiones de verdad ha resultado muy satisfactorio. Poder ver cómo todo tu trabajo previo se unifica al ponerle un agente que lo utilice y funcione correctamente ha sido una de las experiencias más instructivas y gratificantes de la carrera.

### ***6.4 Jaime***

Puedo decir de este proyecto, que, para mí, ha sido uno de los más interesantes y que más me ha gustado a lo largo de la carrera, por no decir el que más. Esto es el fruto, de un ambiente de trabajo muy bueno con mis compañeros y el docente, junto con la adquisición de nuevos conocimientos en lo referente a Python, interfaces y redes, además de un objetivo final muy llamativo, como es desarrollar un agente que juegue de forma autónoma contra una persona al ajedrez chino, sin olvidar otros objetivos como han sido, gestionar los usuarios mediante un servidor o la posibilidad de jugar online con otros usuarios.

Por lo tanto, podría decir que ha sido una experiencia muy buena en el desarrollo de un producto software desde cero, lo cual me ha gustado mucho y le ha sumado mucho a esta práctica.

## **6.5 Manuel**

En esta práctica, se han realizado además de profundizar en múltiples conceptos muy importantes y fundamentales en el desarrollo vital como “Programadores” que debemos tener. Todo englobado y perfectamente preparado de modo, que todos los participantes del grupo puedan disfrutar de todo lo impartido en los requerimientos de la práctica. Luego para finalizar detalles como un uso más serio de la plataforma GitHub, plataforma vista en otras asignaturas, pero sin entrar tanto en materia. Para finalizar, lo que en mi opinión ha supuesto un plus enorme, que ha sido todo el tema de la implementación de un agente inteligente, pudiendo apreciar su “habilidad” para desarrollar juego en base a “capacidades” programadas por nosotros mismos.



# 7. Anexo

## 7.1 Aspectos técnicos

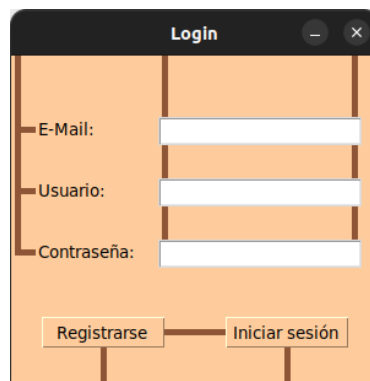
Aquí se hablará sobre las tecnologías y técnicas empleadas en esta práctica.

### 7.1.1 Interfaces de usuario

- **Login:** nada más abrir el cliente, lo primero que nos encontramos es la ventana de Login. En esta ventana se encuentran 3 campos en los que el usuario podrá introducir sus credenciales. Estos campos son: E-Mail, nombre de usuario y contraseña. Una vez estén todos los campos rellenos, el usuario podrá darle al botón de “Iniciar sesión”. Detrás de la interfaz, el cliente cogerá la información introducida en los campos y mandará con ellos un mensaje al servidor utilizando el protocolo que hemos implementado.

El servidor se encargará de buscar en la base de datos un usuario que tenga el E-Mail introducido y comprobará que las credenciales son correctas. En caso negativo o que no encuentre un usuario con ese E-Mail, la respuesta del servidor será un error que hará que el cliente muestre un mensaje diciendo al usuario que las credenciales son incorrectas. En caso positivo, la respuesta será de confirmación y el cliente pasará al menú principal.

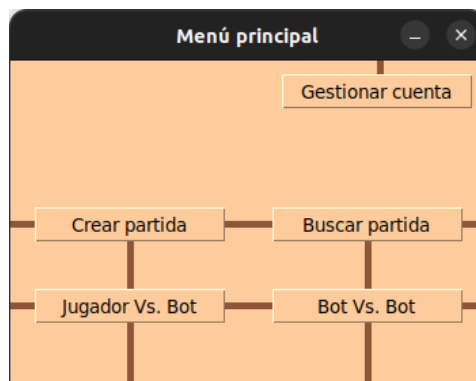
En caso de que el usuario no esté registrado, podrá pulsar en el botón “Registrarse” para ir al formulario de registro.

The image shows a screenshot of a web application window titled "Login". The window has a light orange background and a dark header bar with the title "Login" and standard window control buttons (minimize, maximize, close). On the left side, there are three labels: "E-Mail:", "Usuario:", and "Contraseña:". To the right of each label is a white input field. At the bottom of the window, there are two buttons: "Registrarse" on the left and "Iniciar sesión" on the right. The buttons are light orange with dark text.

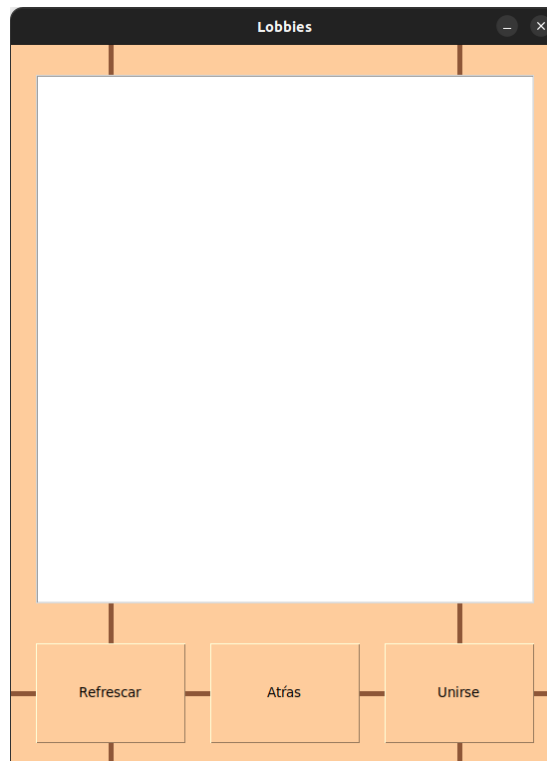
- **Forms:** en esta ventana, también aparecen 3 campos para introducir las credenciales del usuario. Una vez se hayan relleno, el usuario podrá darle al botón de “Crear nuevo usuario”. El cliente mandará un mensaje con las credenciales al servidor, y este se encargará de comprobar que no hay ningún otro usuario en la base de datos con el mismo nombre de usuario o E-Mail. Si la comprobación no encuentra nada, el usuario se añadirá a la base de datos y el cliente pasará al menú principal. Si el usuario ya existe, se le notificará con un mensaje de error al usuario.



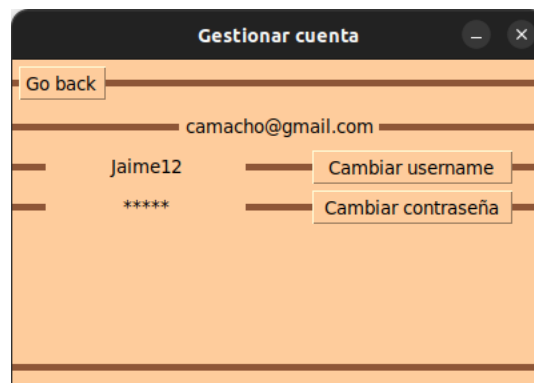
- **Main window:** esta ventana será la encargada de la navegación entre los diferentes modos de juego. Los modos online son “Crear partida” y “Buscar partida”. “Crear partida” mandará al servidor un mensaje de creación de sala utilizando el nombre del usuario para nombrar la sala y dejará al usuario con una ventana en negro hasta que alguien se una a esa partida. Para encontrar estas partidas creadas, el usuario pulsará el botón “Buscar partida” que le llevará a la ventana de “Lobby”. Los modos un jugador son “Jugador Vs. Bot” y “Bot Vs. Bot”. “Jugador Vs. Bot” mandará un mensaje de creación de partida y abrirá una ventana de juego en la que el jugador podrá jugar contra el agente que hemos programado. El botón “Bot Vs. Bot” también mandará un mensaje al servidor y abrirá otra ventana en la que el usuario podrá espectar una partida entre dos bots. En caso de que ocurra un error al crear la partida, se le notificará al usuario por un mensaje de error. Al acabar una partida, el cliente siempre volverá a esta ventana.



- **Lobby:** en esta ventana aparecerá una lista de todas las partidas abiertas en el servidor. El usuario podrá actualizar esta lista pulsando el botón “Refresh” o seleccionar una de ellas y después pulsar el botón “Join lobby”. Si el usuario pulsa este botón sin haber seleccionado una partida, se mostrará un mensaje de error. Si se ha hecho correctamente, el cliente mandará un mensaje al servidor solicitando unir el jugador a la partida. Si la respuesta es positiva, el cliente abrirá una ventana de juego y habilitará la ventana correspondiente al jugador que está al otro extremo iniciando la partida. Si la respuesta es negativa ya sea por un error de condición de carrera o un error en el servidor, se mostrará un mensaje de error.



- **Gestionar cuenta:** al abrir esta ventana podremos ver un botón en la parte superior izquierda, “Go back”, el cual nos sirve para volver a la ventana anterior, además también podemos ver labels con información en orden descendente correspondiente a nuestro usuario, como es el email, el nombre de usuario y la contraseña oculta por el símbolo “\*”. Junto al nombre de usuario podemos ver el botón “Cambiar username”, al igual que al lado de la contraseña se encuentra el botón de “Cambiar contraseña”. Una vez pulsemos cualquiera de estos dos últimos botones, en ambos casos aparecerán dos cajas para introducción de texto, donde habrá que introducir en ambos casos la contraseña actual por motivos de seguridad, como indican los labels a su izquierda. Además, debajo de estos dos cuadros, se mostrará otro más, donde habrá que introducir el nuevo usuario o la nueva contraseña, lo cual ha sido definido en la interacción con los botones, interacción la cual ha quedado reflejada en un label también, indicando la información que se va a cambiar. Finalmente tendremos un botón “Cambiar”, que también aparece cuando se interacciona con los otros botones y que sirve para que una vez rellenados todos los cuadros, realizar los cambios solicitados.



### 7.1.2 Base de datos

Nuestro servidor cuenta con una base de datos implementada en SQLite3. Esta base de datos la utilizaremos para guardar toda la información relacionada con los usuarios, como sus credenciales o su historial de partidas. El modelo relacional de esta base de datos es:

#### Relaciones:

JUGADOR(user, password, e-mail, elo, victorias, derrotas, empates)

PARTIDA(jugador1, jugador2, id\_partida, movimientos, fichas1, fichas2, resultado)

#### Foreign keys:

PARTIDA.jugador1, PARTIDA.jugador2 → JUGADOR.user

#### Restricciones:

El resultado de PARTIDA es un solo carácter. Si ganan las rojas, w. Si ganan las negras, b. Si es empate, d.

PARTIDA.id\_partida is UNIQUE.

JUGADOR.e-mail is UNIQUE

JUGADOR nos sirve para guardar la información relacionada con el jugador, como las credenciales para utilizarlas en la gestión del usuario, así como su recuento de partidas para mostrarle la puntuación online que tiene. Hemos decidido que tanto el nombre de usuario como el correo electrónico sean únicos, de forma que dos usuarios no pueden tener el mismo E-Mail y nombre de usuario.

PARTIDA guarda la información de cada partida guardada e información necesaria para luego calcular el elo del jugador. Para saber el resultado de la partida hemos decidido limitarlo a un solo carácter que especifique el ganador de la partida.

La base de datos es usada exclusivamente por el servidor. Siempre que el cliente necesite información de ella, manda una solicitud al servidor y este lo procesa y devuelve el resultado. Por ejemplo, para registrar a un usuario el cliente manda un mensaje con las credenciales nuevas y el servidor es el que se encarga de insertar la información a la base de datos.

### 7.1.3 Sockets y protocolo empleado

Para la comunicación interproceso en la práctica se ha hecho uso de tecnología basada en sockets, concretamente sockets UDP, puesto que el principal objetivo es la eficiencia en el intercambio de mensajes y no la fiabilidad que proporcionan otros protocolos como TCP.

Para implementar la arquitectura de sockets, fue necesario crear un protocolo de mensajes. El formato de cualquier mensaje es el siguiente:

CÓDIGO	CAMPO_1	-	CAMPO_2	-	CAMPO_N
--------	---------	---	---------	---	---------

Como se puede ver, cualquier mensaje tiene un primer campo relacionado con el código identificador del mensaje. Adicionalmente, los mensajes podrían tener más campos, en cuyo caso se añade el valor asociado al campo, utilizando el carácter “-” para separar diferentes campos.

Todos los códigos fueron definidos en un módulo Python separado, denominado protocol.py, para obtener una mayor modularidad en el programa y para facilitar la lectura del programa.

De esta forma, cualquier otro archivo de código fuente que necesite utilizar el protocolo de comunicaciones, tan solo incluye la dependencia del módulo anterior y podrá utilizar el protocolo sin problemas.

Tras la implementación del protocolo, ya pudimos implementar la arquitectura de sockets y establecer la aplicación del ajedrez chino como un videojuego distribuido. Para ello, los clientes utilizan sockets UDP, y conocen la dirección IP de la máquina que tiene el servidor de juego (se puede hacer de forma directa, o mediante docker containers). De esta forma, cualquier interacción entre el usuario y el servidor de juego se produce mediante sockets, haciendo uso del protocolo descrito anteriormente.

Para que la arquitectura funcione correctamente, es necesario dotar de un socket UDP a cada instancia del servidor de juego, así como a cada instancia de partida, puesto que la comunicación dentro de una partida no la dirige el servidor de juego, sino la partida (para reducir la carga de trabajo en el servidor de juego y acotar las responsabilidades).

### 7.1.4 Dockers

Hemos utilizado la tecnología docker para el despliegue del servidor de partida, mediante el cual los clientes/usuarios pueden acceder a una partida. Hemos partido de una imagen(en resumidas cuentas un paquete que contiene toda la configuración necesaria para que una aplicación se pueda ejecutar), a la cual le hemos añadido algunas configuraciones extra para el correcto y adecuado funcionamiento con respecto a nuestra práctica, para finalmente, lanzar/levantar el contenedor(es parecido a una MV pero sin la necesidad de un sistema operativo huésped particular, y mucho más liviano) que nos permitirá comenzar con el despliegue automático de la aplicación.

También hacemos uso de la herramienta Docker Compose, que es una herramienta que nos permite tener aplicaciones multicontenedor.

### ***7.1.5 Makefile y bash script***

El objetivo principal del archivo makefile es permitir una rápida ejecución del programa, tanto para lanzar el servidor, como para instalar las dependencias necesarias, que serían pygame, tkinter, docker y docker-compose. También hemos utilizado archivos bash para la ejecución de forma directa de los comandos necesarios para la ejecución del código y la instalación de las librerías necesarias.

## **8. Referencias externas**

En esta sección, se tratará sobre las referencias utilizadas para el desarrollo de las distintas partes de la práctica.

### **8.1 De código**

En este subapartado se hará mención de aquellas referencias que hayan afectado al código.

#### **8.1.1 Pygame**

Para el desarrollo e implementación de la jugabilidad en el tablero, hemos hecho uso de la librería pygame y su documentación oficial:

<https://www.pygame.org/docs>

#### **8.1.2 Tkinter**

Para el desarrollo de otras interfaces complementarias hemos empleado el módulo tkinter que nos permite crear interfaces gráficas en Python:

<https://docs.python.org/es/3/library/tkinter.html>

### **8.2 De información**

En este subapartado se hará mención de aquellas referencias utilizadas para la parte gráfica o el entendimiento de este juego.

#### **8.2.1 World Xiangqi Federation**

Para la estandarización del juego, de sus movimientos y sus representaciones, nos hemos atendido a las normas de la Federación Internacional de Xiangqi:

<http://wxf.ca/xq/computer/fen.pdf>

#### **8.2.2 Imágenes de las piezas**

Las imágenes empleadas para la representación del juego fueron obtenidas de la página:

[https://www.chessprogramming.org/Chinese\\_Chess#Pieces](https://www.chessprogramming.org/Chinese_Chess#Pieces)

Y fueron reescaladas con la herramienta BeFunky:

<https://www.befunky.com/es/>

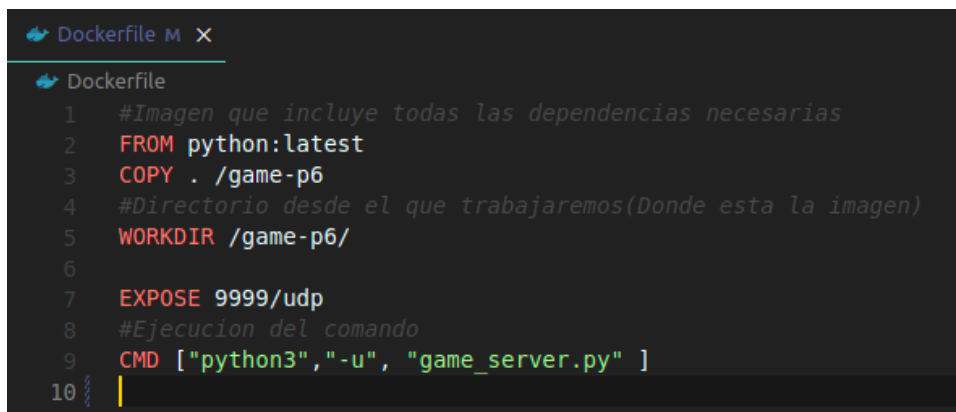
## 9. Manual de usuario

En este apartado se tratarán aspectos adicionales para el correcto uso del usuario del programa, así como entender su funcionamiento.

### 9.1 Instalación y ejecución de Docker

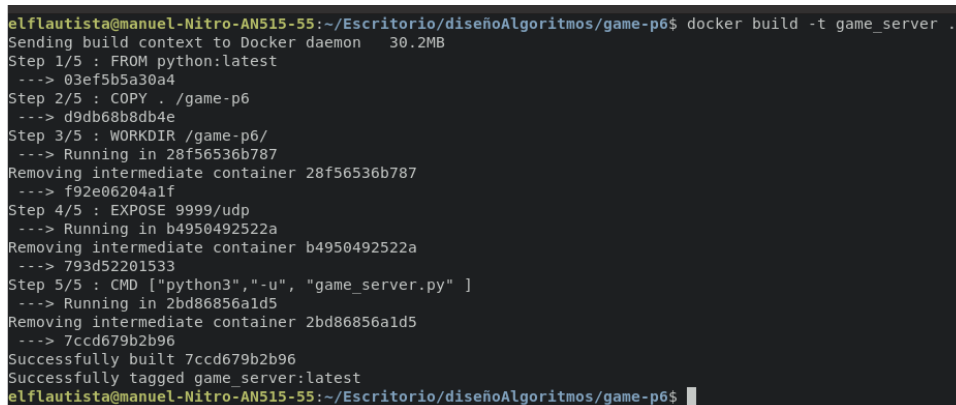
Para comenzar con el proceso de lanzamiento de Docker, es necesario la construcción de una imagen (que siempre comienza con la “base” o “fundamento” en otra imagen).

La construcción de la imagen se divide en 2 sencillos pasos: el primero, la construcción de un archivo que debe obligatoriamente denominarse Dockerfile (hay excepciones en las que no es necesario, pero trabajaremos en el caso de que si lo es).



```
Dockerfile M X
Dockerfile
1  #Imagen que incluye todas las dependencias necesarias
2  FROM python:latest
3  COPY . /game-p6
4  #Directorio desde el que trabajaremos(Donde esta la imagen)
5  WORKDIR /game-p6/
6
7  EXPOSE 9999/udp
8  #Ejecucion del comando
9  CMD ["python3","-u", "game_server.py" ]
10
```

Una vez tenemos el Dockerfile creado, y dentro del mismo repositorio local donde se encuentran los demás archivos del proyecto, se procede a “construir” la imagen. Para ello utilizaremos un comando de Docker, llamado, docker build. (El -t es para poder darle un “tag” o nombre a la imagen, esto se debe a que nos será bastante útil de cara al uso del Docker Compose.



```
elflautista@manuel-Nitro-AN515-55:~/Escritorio/diseñoAlgoritmos/game-p6$ docker build -t game_server .
Sending build context to Docker daemon  30.2MB
Step 1/5 : FROM python:latest
--> 03ef5b5a30a4
Step 2/5 : COPY . /game-p6
--> d9db68b8db4e
Step 3/5 : WORKDIR /game-p6/
--> Running in 28f56536b787
Removing intermediate container 28f56536b787
--> f92e06204a1f
Step 4/5 : EXPOSE 9999/udp
--> Running in b4950492522a
Removing intermediate container b4950492522a
--> 793d52201533
Step 5/5 : CMD ["python3","-u", "game_server.py" ]
--> Running in 2bd86856a1d5
Removing intermediate container 2bd86856a1d5
--> 7ccd679b2b96
Successfully built 7ccd679b2b96
Successfully tagged game_server:latest
elflautista@manuel-Nitro-AN515-55:~/Escritorio/diseñoAlgoritmos/game-p6$
```

Una vez tenemos construida la imagen, debemos con ello, lanzar/levantar el contenedor, que lo hace gracias a la construcción previa de una imagen. Todo contenedor funciona gracias a una imagen previamente construida. Para lanzar el contenedor utilizaremos el comando docker run:



```
elflautista@manuel-Nitro-AN515-55:~/Escritorio/diseñoAlgoritmos/game-p6$ docker run -d --rm -p 9999:9999 game_server
91d065bcff2d45db8c7f08ac7e629441f038ead29ec68cfd72be248080e64c33
elflautista@manuel-Nitro-AN515-55:~/Escritorio/diseñoAlgoritmos/game-p6$
```

Estos serían los pasos a seguir para levantar finalmente un contenedor. La parte final del run con algunas de las opciones por línea de comandos, lo hemos hecho de una manera un poco más sencilla, rápida y escueta que no con el docker run.

Hemos creado un Docker-compose, que es un archivo .yaml, que nos permite crear aplicaciones multiplataforma, pero también nos permite trabajar algunos aspectos de manera más rápida, y algunos otros que no son “posibles” desde la línea de comandos, como el establecimiento de una red, para que varios host distintos pudiesen acceder al contenedor.

```
🐘 docker-compose.yaml
1  version: "3"
2  services:
3    game_server:
4      network_mode: "host"
5      image: game_server:latest
6      ports:
7        - "9999:9999/udp"
```

La manera de ejecutarlo es muy sencilla, basta con introducir el comando **docker-compose up -d**. Al ejecutar el docker-compose, el docker run se ejecuta automáticamente sobre la imagen que se indica dentro del docker-compose.

Ahora iremos con la instalación de Docker. Como primer paso (sin tener en cuenta el típico `sudo apt update`), instalar algunos paquetes como requisito previo para permitir a apt usar paquetes a través de https.

```
$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

```
elflautista@manuel-Nitro-AN515-55:~$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

A continuación, añadimos la clave de GPG que es necesaria para el repositorio oficial de Docker en el sistema:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

```
elflautista@manuel-Nitro-AN515-55:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Agregamos el repositorio Docker a las fuentes de apt:

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

```
elflautista@manuel-Nitro-AN515-55:~$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

Ahora deberemos actualizar los paquetes con los paquetes de Docker recién agregados, y asegurarnos de que se está a punto de realizar la instalación desde el repositorio de Docker y no desde el de Ubuntu para finalmente se realiza la instalación de Docker:

```
$ apt-cache policy docker-ce
```

```
$ sudo apt install docker-ce
```

```
elFlautista@manuel-Nitro-ANS15-55:~$ apt-cache policy docker-ce
elFlautista@manuel-Nitro-ANS15-55:~$ sudo apt install docker-ce
```

Finalmente, un último paso para comprobar que la instalación ha sido realizada con éxito, es utilizar el comando `sudo systemctl status docker`, que debería dar una salida similar a la siguiente:

```
$ sudo systemctl status docker
```

```
elFlautista@manuel-Nitro-ANS15-55:~$ sudo systemctl status docker
[sudo] contraseña para elFlautista:
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2022-05-01 11:31:36 CEST; 2h 42min ago
   TriggeredBy: ● docker.socket
   Docs: https://docs.docker.com
   Main PID: 1636 (dockerd)
   Tasks: 18
   Memory: 109.9M
   CGroup: /system.slice/docker.service
           └─1636 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.687629538+02:00" level=warning msg="Your kernel does not support CPU realtime scheduler"
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.687647096+02:00" level=warning msg="Your kernel does not support cgroup blkio weight"
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.687653773+02:00" level=warning msg="Your kernel does not support cgroup blkio weight_device"
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.687961579+02:00" level=info msg="Loading containers: start."
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.808804285+02:00" level=info msg="Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option --bip
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.828964465+02:00" level=info msg="Loading containers: done."
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.838889776+02:00" level=info msg="Docker daemon commit=87a90dc graphdriver(s)=overlay2 version=20.10.14
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.859226090+02:00" level=info msg="Daemon has completed initialization"
may 01 11:31:36 manuel-Nitro-ANS15-55 systemd[1]: Started Docker Application Container Engine.
may 01 11:31:36 manuel-Nitro-ANS15-55 dockerd[1636]: time="2022-05-01T11:31:36.874820821+02:00" level=info msg="API listen on /run/docker.sock"
lines 1-21/21 (END)
```

## 9.2 Ejecución de los archivos bash script e inicialización de la base de datos

Hemos creado dos archivos bash script para la instalación de las librerías necesarias y la ejecución del cliente y del servidor.

El archivo `ajedrez_chino.sh` ejecuta los comandos necesarios para la instalación de las librerías relativas a la ejecución del cliente y la propia ejecución. Para lanzar el cliente sin problemas, el usuario podrá ejecutar cualquiera de los dos comandos:

```
$ bash ajedrez_chino.sh
```

```
$ ./ajedrez_chino.sh
```

El archivo `server.sh` realiza el mismo procedimiento, pero relacionado al server. Adicionalmente, debido a que hemos realizado la implementación del servidor por medio de dockers, también realiza las instalaciones relacionadas con la librería `docker.io`. El archivo realiza todo el procedimiento detallado en el apartado 9.1 de forma directa. Estos son los comandos para la ejecución del servidor de juego:

```
$ bash server.sh
```

```
$ ./server.sh
```

En caso de no poder ejecutar correctamente el servidor, principalmente debido a ciertas dependencias incluidas, instalar manualmente todas las dependencias y ejecutar el siguiente comando:

```
$ python3 game_server.py
```

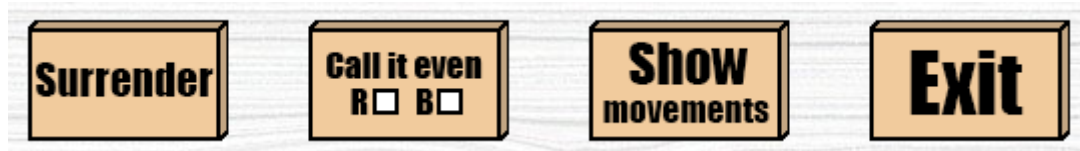
**NOTA:** El usuario debe de tener instalado de antemano python 3 para la ejecución de algunos comandos. En caso de que se quiera instalar las librerías necesarias, se deberán ejecutar los archivos bash con permisos de root por medio de comandos como sudo.

Para una correcta ejecución del programa, será necesario a veces inicializar la base de datos correctamente. Esto se realizará mediante la ejecución del módulo de python *generate\_db.py*, esto se puede hacer mediante el comando:

```
$ ./generate_db.py
```

### 9.3 Botones e interfaz de juego

Durante la partida, podremos observar 4 botones en la pantalla, siendo estos los siguientes:

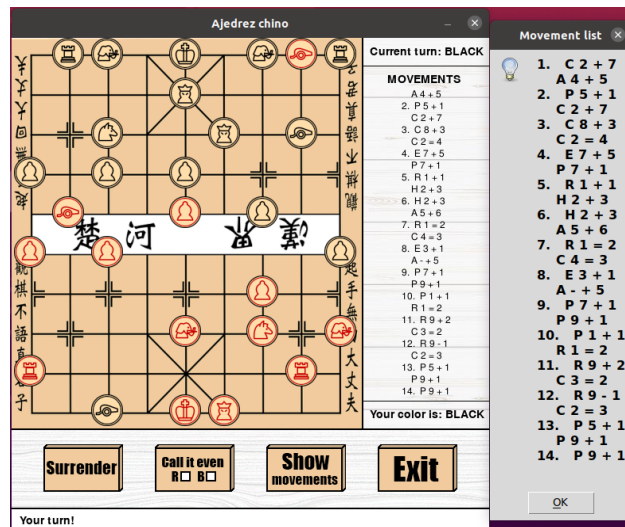


Estos botones sólo se podrán accionar durante el turno del jugador, y sus funciones son las siguientes:

- **Surrender:** nos permite rendirnos de la partida, dando la victoria al rival.
- **Call a tie:** nos permite pedir tablas. Al hacerlo, se marcará con una X la checkBox correspondiente a nuestro color. Si el otro jugador ya había pedido tablas, la partida quedará en empate.



- **Show movements:** muestra los movimientos jugados a lo largo de toda la partida en caso de que no puedan ser mostrados en la parte derecha de la pantalla.



- **Exit:** permite cerrar el tablero y volver a la página de lobbies siempre y cuando la partida haya finalizado.

## 9.4 Guía de uso

Una vez ejecutado el programa, accederemos a la primera ventana de este, donde deberemos iniciar sesión si ya disponemos de una cuenta, o registrarnos en caso contrario.

Tras esto, podemos ya elegir entre crear una partida, unirnos a una existente, jugar contra un bot o incluso espectral una partida entre bots.

Al acabar una partida, podemos cerrar la ventana para volver al menú anterior, sin necesidad de tener que iniciar sesión de nuevo.