UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Desarrollo del juego del Backgammon con integración de técnicas de aprendizaje por refuerzo

Development of the game of Backgammon integrating reinforcement learning techniques

Realizado por
Sergio Tineo Bravo

Tutorizado por
Lorenzo Mandow Andaluz

Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2019

Fecha defensa

El ____ de _____ de 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

Este documento representa el trabajo fin de grado sobre el diseño, desarrollo e implementación del juego Backgammon. Así como la integración de técnicas de aprendizaje por refuerzo en lenguaje Python.

El programa podrá ser ejecutado directamente desde linea de órdenes (command line) para lanzar los scripts que ejecutan los diferentes tipos de algoritmos y ver sus resultados. Así mismo también será posible jugar contra la maquina desde una interfaz gráfica más sencilla y amigable realizada con la interfaz de facto integrada en el lenguaje.

Python es un lenguaje multiparadigma e interpretado lo cual supone un nuevo planteamiento durante la implementación del programa con respecto a otros lenguajes como JAVA. Así mismo la aplicación de técnicas de aprendizaje por refuerzo a este juego con tantos estados posibles se hace muy interesante por la necesidad de comprender las características del lenguaje para una correcta implementación de estas técnicas. Por último tras la aplicación de estas al juego se analizarán los resultados.

Palabras clave: Backgammon, aprendizaje por refuerzo, Python, interpretado, multiparadigma, estados posibles, linea de órdenes, interfaz gráfica.

Abstract

This document presents the final degree project about design, development and implementation of the game of Backgammon. Also the integration of reinforcement learning techniques. The project has been developed using the Python 3 programming language.

The program can be executed directly from command line to launch the scripts that execute different types of algorithms and see their results. It is also possible to play against the machine from a simple GUI developed with the de facto interface integrated in this programming language.

Python is multi-paradigm and interpreted programming language. This implies a new approach during the implementation of the computer program, if we compare it with another programming language like JAVA. Moreover, the application of reinforcement learning techniques to this game with so many possible states, it is very interesting. Because it is necessary to understand the characteristics of the programming language for correct implementation of learning techniques. Finally, after applying these techniques to the game, results will be analyzed.

Keywords: Backgammon, reinforcement learning, Python, interpreted, multi-paradigm, possible states, command line, GUI

Índice

1

Introducción

A continuación veremos ciertos aspectos relevantes desde a que llevó hacer este trabajo fin de grado, el por qué, los objetivos que se esperan alcanzar, la metodología con la que se ha llevado a cabo, la definición de las palabras clave y las tecnologías utilizadas.

1.1 Motivación

La motivación de este trabajo es el desarrollo de una aplicación informática para el juego del Backgammon. Dicha aplicación incluye la posibilidad de jugar contra la máquina. Para ello se ha desarrollado igualmente un mecanismo de aprendizaje por refuerzo, de tal forma que es posible entrenar un jugador de Backgammon que aprende el juego jugando bien contra otro jugador especificado (por ejemplo, un jugador aleatorio).

Esta tarea supone la oportunidad de profundizar, por una parte, en los aspectos de la ingeniería del software estudiados en el Grado, y por otra, en los de inteligencia artificial, concretamente los de aprendizaje automático (machine learning). De ahí surge la necesidad de aprender y comparar algún algoritmo de los que existen, centrándonos en el estudio del más util para el juego que nos atañe, resulta de gran interés analizar en que se basa y cuál es su funcionamiento. Ser capaz aplicarlo a un juego un tanto desconocido por la mayoría de personas, el Backgammon, resulta satisfactorio ya que por una parte se está dando a conocer un juego tan desconocido

como es este como por otra estamos aplicando algunas técnicas que permiten dotar a la máquina de cierta inteligencia sabiendo elegir en cada situación cual es la mejor jugada que ella cree.

Por otro lado desarrollar este juego desde cero usando buenas metodologías de desarrollo software y ademas usando el lenguaje de programación Python, que está siendo solicitado por muchas empresas actualmente, es de gran interés por poder aplicar lo aprendido durante el grado y además aprender un nuevo lenguaje con sus similitudes y a la vez grandes diferencias con otros lenguajes orientados a objetos tales como Java.

Figura 1.1: Mapa

A modo de resumen la Figura 1.1 representa el mapa mental relacionado con las motivaciones que llevan hacer este trabajo, resaltando tanto lo que aporta como lo que me motiva.

1.2 Objetivos

Los objetivos principales del TFG serán aprender un nuevo lenguaje, afianzar las bases del desarrollo de aplicaciones software y estudiar y aplicar la capacidad de aprendizaje de los algoritmos de aprendizaje por refuerzo.

Se desarrollará en Python una aplicación para PC del juego para dos jugadores Backgammon, este tendrá una interfaz gráfica donde podrá observarse el estado actual de la partida y tendrá diversas modalidades de juego como Jugador Humano contra Jugador Humano o Jugador Humano contra Jugador Maquina. Así mismo tendrá ciertos detalles para la ayuda del jugador durante una partida.

Tras todo ello se estudiará y aplicará algún algoritmo de aprendizaje por refuerzo de forma que obtengamos un poderoso rival en el tablero. También se analizará el comportamiento del algoritmo y lo eficaz del mismo.

1.3 Estructura de la memoria

La descripción breve de la estructura de cada capítulo será la siguiente:

- Durante el presente capítulo, el primero, se describirán las características fundamentales de la memoria como la motivación, los objetivos o las tecnologías usadas entre otros.
- En el segundo capítulo se describe el juego de mesa Backgammon, antecedentes históricos, como se juega y las normas asociadas.
- En el tercer capítulo será donde veremos en que consiste el aprendizaje por refuerzo, el uso en la actualidad y la aplicación de estas técnicas al Backgammon, esto último influenciado por el ingeniero Gerald Tesauro [14].
- Durante el cuarto capítulo se introducirá el lenguaje Python, su perspectiva en la actualidad, sus principales usos y diferencias con otros lenguajes. Así como una introducción al lenguaje y a sus interfaces diseño de interfaces en este lenguaje.
- En el quinto realizaremos el analisis y diseño del programa con los requisitos, casos de uso y el primer diseño del modelo sobre el que nos basaremos para crear el programa y la interfaz.
- En el sexto capítulo se mostrará en detalle el desarrollo de la programación del juego, puntos importantes y detalles. De igual modo se detallará la elaboración de la interfaz gráfica y como se ha seguido un diseño modelo-vista-controlador.

- El séptimo capítulo se detallan las pruebas unitarias que se han realizado en la versión del programa sin aprendizaje por refuerzo. Así mismo se detallan pruebas reales sobre la interfaz.
- En el octavo capítulo incluiremos los nuevos requisitos, casos de uso y diseños necesarios para la implementación del aprendizaje por refuerzo.
- El noveno capítulo se aplicarán técnicas de aprendizaje por refuerzo al Backgammon y se realizarán pruebas unitarias vigilando el correcto funcionamiento.
- El décimo capítulo analizaremos los resultados de la aplicación del aprendizaje por refuerzo.
- El onceavo capítulo tendrá los objetivos alcanzados y futuros trabajos.

Tras la detalla explicación de todo el proceso de este trabajo fin de grado a continuación del capítulo once está la bibliografía y tras el los apéndices para ejecutar el programa.

1.4 Metodología

La memoria se desarrollará basándose en el conocido modelo de desarrollo incremental. Se planificará cada parte del proyecto en "bloques", desde el estudio necesario pasando por los requisitos hasta el desarrollo. De esta forma iremos obteniendo un producto cada vez mejor, más probado y más extenso.

El motivo principal por el que hemos decidido usar esta metodología es la facilidad de agregar funcionalidades, ya que es un desarrollo flexible, permite ir extendiendo al estudio de los algoritmos de aprendizaje con refuerzo.

Este modelo de desarrollo es bastante utilizado por su flexibilidad y ser una importante mejora respecto al modelo en cascada.

Conceptualmente esta forma de desarrollo consiste en planificar un proyecto más grande en bloques más pequeños, a los que se los denomina iteración, cada iteración debe satisfacer un requerimiento o un subconjunto no muy grande de requisitos del proyecto final, una iteración en si misma debe ser completa, es decir, pasar por los pasos de análisis, diseño, desarrollo y pruebas, y dar como resultado un producto usable. Claramente una iteración debe tener en cuenta la anterior y ampliar lo que se hubiera realizado a priori.

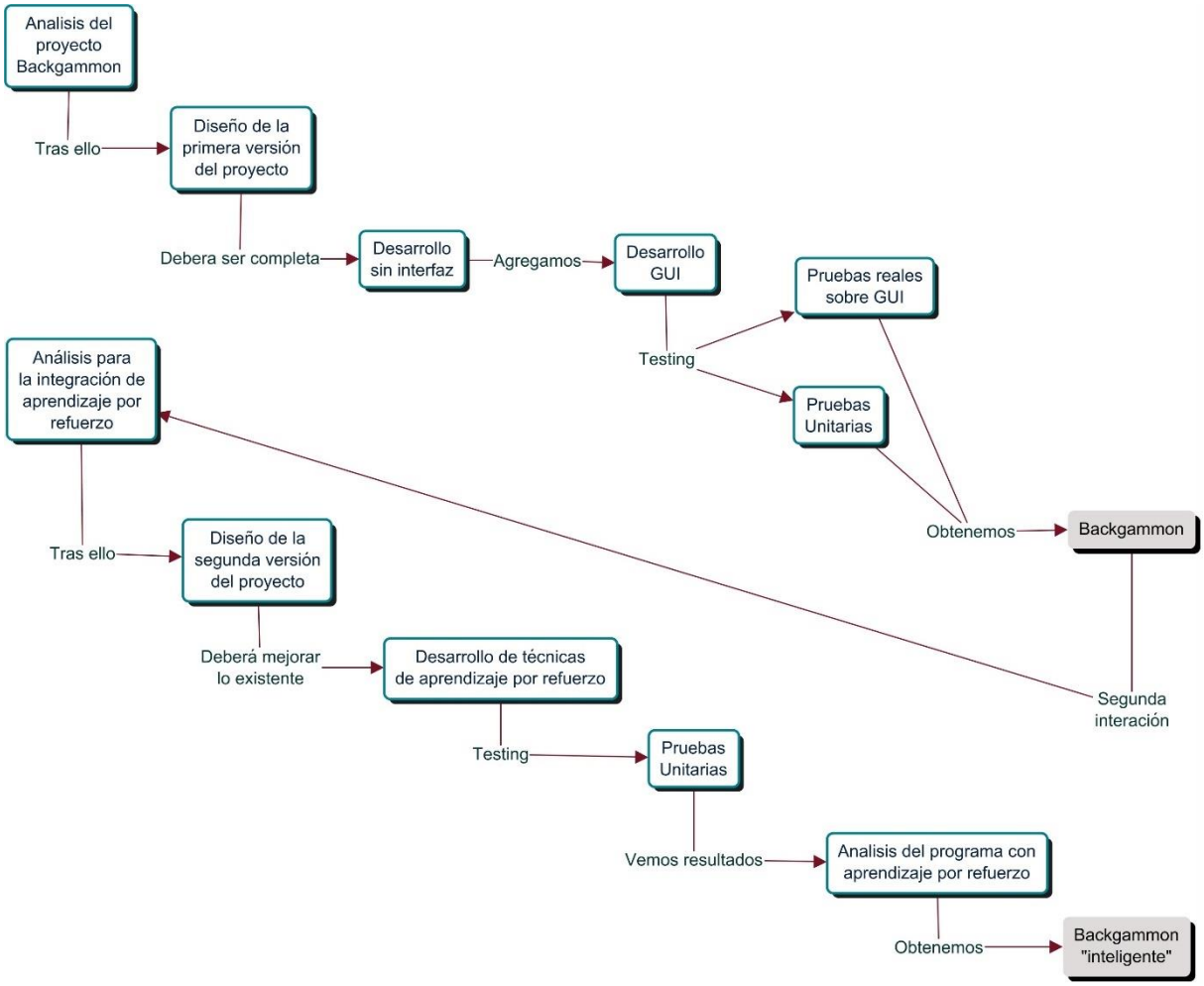

Figura 1.2: Mapa conceptual del desarrollo iterativo en mi trabajo

En el caso que nos atañe, como podemos ver en la figura 1.2, la primera iteración será todo lo correspondiente para el programa capaz de ejecutar el juego Backgammon

con y sin interfaz gráfica y la segunda iteración será agregar el aprendizaje por refuerzo, con todo lo que conlleva.

Gracias a esto se puede dividir mejor el trabajo, contando solo con el analisis, diseño, desarrollo y pruebas necesarios en cada momento.

1.5 Definición de conceptos

En este apartado vamos a definir las palabras claves que se pueden leer en las primeras páginas del proyecto. Más adelante se presentará una explicación más detallada de estos conceptos a lo largo de la memoria, sin embargo es se hace necesario definirlas brevemente porque se han usado y se usarán durante toda la memoria.

Backgammon, es un juego de mesa para dos jugadores en el cual cada jugador tiene dos dados y 15 fichas, generalmente de color blanco y negro, y el objetivo es quedarte sin fichas sacándolas del tablero. Se trata de un juego con azar y estrategia.

Aprendizaje por refuerzo, es un área del aprendizaje automático (machine learning) que se puede aplicar a agentes software, estos no son más que una representación programada de una "entidad" cuyo único propósito es decidir una acción, los cuales en cada momento observaran su entorno, el tablero en el Backgammon, e intentarán decidir cual es la mejor jugada en cada situación maximizando recompensa o acercándose a la victoria.

Python, lenguaje de programación multiplataforma el cual tiene como filosofía ser legible y sencillo de entender.

Interpretado, referido a un lenguaje de programación, es un lenguaje para el cual no existe compilación previa y generalmente se ejecuta directamente linea a linea tras previamente haberlas traducido a código máquina.

Multiparadigma, referido a un lenguaje de programación, es un lenguaje capaz de soportar diferentes paradigmas de programación como puedan ser el orientado a objetos, el imperativo, etc.

Estados posibles, si entendemos como estado la situación concreta en la que se encuentra el juego en un momento dado, los estados posibles son las situaciones posibles que se pueden dar de forma inmediata a esa situación concreta del juego.

Línea de órdenes, es una forma que tiene el usuario de dar órdenes a un programa informático mediante linea de texto simple en un terminal, ya sea desde el cmd de Windows o el terminal integrado Python.

Interfaz gráfica, o interfaz gráfica de usuario (GUI), es una vista más amigable y sencilla que la terminal para la mayoría de usuarios ya que suele representarse con una ventana, botones de acción, entradas y texto en su interior, que permiten ver el estado actual y dar órdenes al programa informático.

1.6 Herramientas utilizadas

En este apartado explicaremos las herramientas utilizadas en el trabajo tanto para el desarrollo del programa como para el desarrollo de la memoria. Es cierto que se podría haber incluido Python como "tecnologías usadas" y una definición más extensa, ese punto lo trataremos más adelante (véase 4. Python), y en este nos centraremos solo en las herramientas. Entre ellas haremos más hincapié en las necesarias para realizar el programa.

1.6.1 PyCharm Community v2019.2.1 [27]

Es un IDE, entorno de desarrollo integrado, para Python de los más completos y usables que hay para este lenguaje. Forma parte de la *suite* de herramientas ofrecidas por JetBrains. Concretamente se usó la versión PyCharm Community 2019.2.1 que es gratuita y completa.

Este IDE tiene un editor de código específico para Python lo cual tiene grandes ventajas como que convierte las tabulaciones en cuatro espacios, o busca de forma eficaz entre los módulos y paquetes del proyecto.

Figura 1.6: Logo de PyCharm

También permite ejecutar desde consola, lanzar ventanas, depurar a tiempo real y usar puntos de ruptura, atajos de teclado para comentar bloques, contraer bloques, dar índices y ayudar a dar formato. Además de buscar en tiempo real posibles problemas en el código o problemas de sangrado mientras se escribe.

En este IDE se ha desarrollado todo el proyecto desde cero, en una primera instancia sin control de versiones, y tras la primera versión ejecutable se le añadió el control de versiones.

1.6.2 GitHub [24]

Es una de las plataformas más conocidas para el alojamiento, modificación e intercambio de código del mundo. Sirve para alojar proyectos y llevar un control de versiones Git. Es un software que se encarga de controlar los cambios realizados en un proyecto y aplicarlos o generar una rama nueva.

Figura 1.3: Logo de GitHub

Se ha usado el control de versiones Git del IDE PyCharm, un IDE del que hablamos en el apartado 1.6.3, ya que es sencillo y gracias a usarlo se ha podido ir guardando los cambios del proyecto y poder tener una copia actualizada y privada en

red, poder tener el proyecto alojado de forma privada es una funcionalidad gratuita desde hace poco en GitHub.

1.6.3 Anaconda [25]

Anaconda es una distribución de Python específica para el mundo científico y al mismo tiempo una de las mejores formas de empezar a programar en Python ya que se trata de una *suite* completa que integra el intérprete, numerosos paquetes, un IDE y otros elementos interesantes para el desarrollo.

Figura 1.4: Logo de Anaconda

Se ha utilizado principalmente de la *suite* el Jupyter Notebook 6.0.0 para comenzar a programar y algo el IDE Spyder 3.3.3 para ejecutar scripts, aunque al poco se cambió a otro IDE más ágil y confortable en el que se desarrolló todo el proyecto, por lo que sobre este IDE no hablaré aunque sí del entorno Jupyter.

1.6.3.1 Jupyter Notebook v6.0.0 [26]

Figura 1.5: Logo de Jupyter Notebook

Es un entorno de trabajo interactivo basado en la web, es decir, se ejecuta desde el navegador aunque localmente. Permite ejecutar codigo Python de manera dinámica y adicionalmente combinar esto con un editor de texto que permite títulos, textos e

imágenes. Su uso ha sido específicamente educativo y para ejecutar pruebas simples y sencillas rápidamente.

1.6.4 Notepad++ v7.7 [28]

Es un editor de codigo, aunque también puede ser usado como editor de texto, con soporte a muchos lenguajes de programación e incluye opciones avanzadas muy útiles para programar.

Figura 1.7: Logo de Notepad++

Cada vez que se ha necesitado una herramienta rápida para visualizar codigo, editar múltiples líneas de texto simultáneamente o cualquier tarea simple y rápida o avanzada y de cierta complejidad se ha recurrido al uso de esta util y flexible programa.

1.6.5 MagicDraw 19.0 [29]

MagicDraw de No Magic es una herramienta utilizada para el diseño de proyecto entre otras muchas utilidades, estamos hablando de una herramienta de ingeniería software muy completa y que es compatible con el estándar UML 2.3.

Figura 1.8: Logo de MagicDraw

Se ha usado esta potente herramienta para generar los diagramas de casos de uso, de secuencia, de actividad, de clase y los que sean necesarios para el adecuado analisis y diseño del programa.

1.6.6 GIMP 2.8.22 [30]

Es un editor de imágenes que permite el retoque fotográfico y la elaboración de dibujos y montajes. Soporta extensiones, macros y además es compatible con la mayoría de formatos fotográficos. Es la mejor alternativa gratuita al Adobe Photoshop.

Figura 1.9: Logo de GIMP

Tanto para el icono de juego como para editar cualquier imagen necesaria para la memoria, se ha recurrido a este potente software.

Ahora vamos a describir brevemente las herramientas usadas para la elaboración de la memoria.

1.6.7 FreeMind y CmapTools [31][32]

FreeMind sirve para realizar mapas mentales. Es una herramienta que puede resultar de gran utilidad para, entre otros usos, ayudar a plasmar de alguna manera alguna idea.

CmapTools permite construir, navegar, compartir y criticar modelos de conocimiento representados como mapas conceptuales.

Ambas serán, y han sido usadas, cuando se requiera plasmar de forma más intuitiva alguna idea o concepto.

1.6.8 Office Word 365 [12]

Uno de los componentes más importantes de la suite ofimática de Microsoft. En el se redactará toda la memoria y se le dará formato.

2

Backgammon

En este capítulo vamos a detallar ampliamente todos los aspectos referentes al juego de mesa Backgammon. Por regla general si se tratara de un juego conocido por la inmensa mayoría como el parchís o el ajedrez, tal vez sería suficiente con un breve repaso, sin embargo resulta que este juego no es tan conocido como cabría esperar en España.

2.1 Historia

Se tiene conocimiento de que existen juegos de características similares al Backgammon desde hace más de 5000 años, pero demos más detalles [8].

Durante el año 1922 el arqueólogo Leonard Woolley llevó a cabo excavaciones en campos sumerios, su labor allí tardó nada más y nada menos que 12 años pero entre otros objetos se hizo con un tablero parecido al de hoy día.

Lógicamente este fue el desencadenante de especulaciones acerca del origen de este juego, datándolo inicialmente hace en torno a 5000 años en el Juego real de Ur, que fue encontrado en unas excavaciones cerca de la ciudad que da nombre al juego, actualmente Irak. Es más, en unas excavaciones posteriores cerca de allí, en la actual Irán, se haría un descubrimiento similar solo que datando su origen 150 años antes. Y la profanación de tumbas continuó, aquellos años la arqueología era muy importante, lucrativa y probablemente con menos leyes o leyes menos restrictivas al respecto. En una de las excavaciones más famosas del mundo, la de la tumba de Tutankamon, se

encontró un juego conocido como Senet que guarda cierta relación tanto con el Juego real de Ur, como con el Backgammon de hoy día.

Unos cuantos años después, Platón en su obra República, se habló de Kubeia, además de Petteia, un juego relacionado con el uso de dados, que se cree que acabó derivando en el Grammai, que se puede observar en la Figura 2.1. También debemos mencionar que en la obra Fedón elaborada por Platón se dice que los griegos se divertían con juegos de herencia Egipcia, aquí tenemos un primer nexo con los juegos de las excavaciones. De igual manera que hicieron los romanos adueñándose de los dioses griegos, también hicieron lo propio con los juegos, donde Grammi tras unos años y algunas modificaciones acabó conociéndose como Tabula.

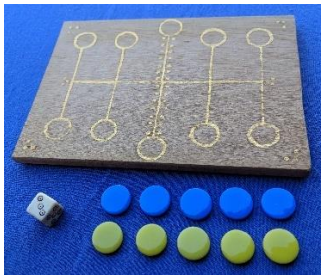

Figura 2.1: Grammai. [7]

Por su parte, antes del IX se jugaba a Nard en Mesopotamia, un juego donde las fichas se disponían de manera concreta en el tablero y donde el sentido de movimiento de los jugadores es opuesto.

Así llegamos al Backgammon, bueno, ya mismo. Sobre el siglo XI llegó a Francia tanto Tabula como Nard los cuales fueron los precursores del juego actual. Desde Francia Alfonso X en su obra Libros de juegos, dio a conocer varios juegos de Tabla que en los siglos posteriores se consolidarían por Europa. Fue sobre el 1650 aproximadamente cuando se pronunciara el termino Backgammon, donde *back* se refiere a volver y *gammon* a juego adquiriendo este nombre por el hecho que las fichas pueden volver al tablero aunque sean eliminadas.

Y por el siglo XVIII Edmon Hoyle publicó un manual especificando las reglas tal y como son hoy día.

2.2 Perspectiva actual

Este juego también recibe otros nombres como Tablas reales, Chaquete o Chanchullo.

Actualmente se realiza un campeonato mundial de Backgammon al año (Backgammon World Championship) [9] en Monte Carlo con premios de miles de euros. De hecho, en Monte Carlo lleva celebrándose desde hace más de 40 años y ha habido ganadores de todas partes del mundo a lo largo de la historia de este campeonato, muchos de Europa aunque ningun español.

Cualquier persona que quiera jugar al Backgammon podrá hacerlo tanto online como en muchas webs de compra generalista, encontrando juegos desde 10€ hasta 300€. Por lo que si se quiere se puede tener acceso a el y no es difícil de encontrar.

2.3 Como se juega

En este juego se enfrentan dos jugadores en un tablero y cuentan con dos dados cada uno [10][11].

2.3.1 Disposición del tablero

Es un juego donde el tablero se configura en 24 "picos" o casillas tal como se muestra en la figura 2.2. En la que se pueden apreciar dos picos y se enfrentan dos jugadores con 15 fichas de color diferente cada uno. El tablero está dividido en 4 cuadrantes de 6 picos cada uno, los picos del 1 al 6 y del 19 al 24 conforman el cuadrante interno de cada jugador, tanto en la documentación del programa como en la memoria en algunas ocasiones haremos referencia a estos cuadrantes como Inicio de X o Meta de X, donde X será el color de la ficha.

Figura 2.2: Picos del Backgammon

Mientras que en muchos juegos se hablan de casillas, en el Backgammon generalmente nos referiremos como picos al lugar donde se pondrán las fichas, en la figura 2.2 se pueden ver dos picos de colores diferentes. Los tableros se suelen numerar de muchas formas, a veces de forma simétrica a cada jugador y otras comenzando por un cuadrante interno y acabando por el otro, de esta última forma es como hemos planteado la numeración del tablero. Se aprecia la numeración elegida en la figura 2.3.

Figura 2.3: Tablero Backgammon numerado

Generalmente jugaremos con dos fichas que nos son familiares por las Damas, las fichas blancas y las fichas negras, así pues tal como se ha indicado en la figura 2.3

tendremos el cuadrante interno de las blancas entre el 1 y el 6 que será el Inicio de las blancas, las blancas serán el jugador 1, y la Meta de las blancas estará entre 19 y 24.

Las fichas se ponen en el tablero siguiendo una forma concreta, con la enumeración establecida van tal como se indica en la tabla 2.1.

Color de la ficha	Posición en el tablero	Cantidad
Blancas	1	2
	12	5
	17	3
	19	5
Negras	24	2
	13	5
	8	3
	6	5

Tabla 2.1: Esquema de colocación de fichas

En la figura 2.4 se puede ver una representación visual del detalle de las posiciones iniciales de las fichas en el tablero.

Figura 2.4: Tablero Backgammon con las fichas en las posiciones iniciales

2.3.2 Movimiento y objetivos

El objetivo principal del juego es sacar las fichas del tablero, para ello todas deberán estar en el cuadrante de Meta que le corresponda y además las fichas solo podrán moverse en un sentido, siguiendo con el esquema de color del apartado 2.3.1, las blancas se moverían en el sentido contrario a las agujas del reloj, donde el eje de las agujas del reloj estaría a la derecha en la mitad del tablero. Y el movimiento de las negras sería el opuesto pero manteniendo el eje de las agujas.

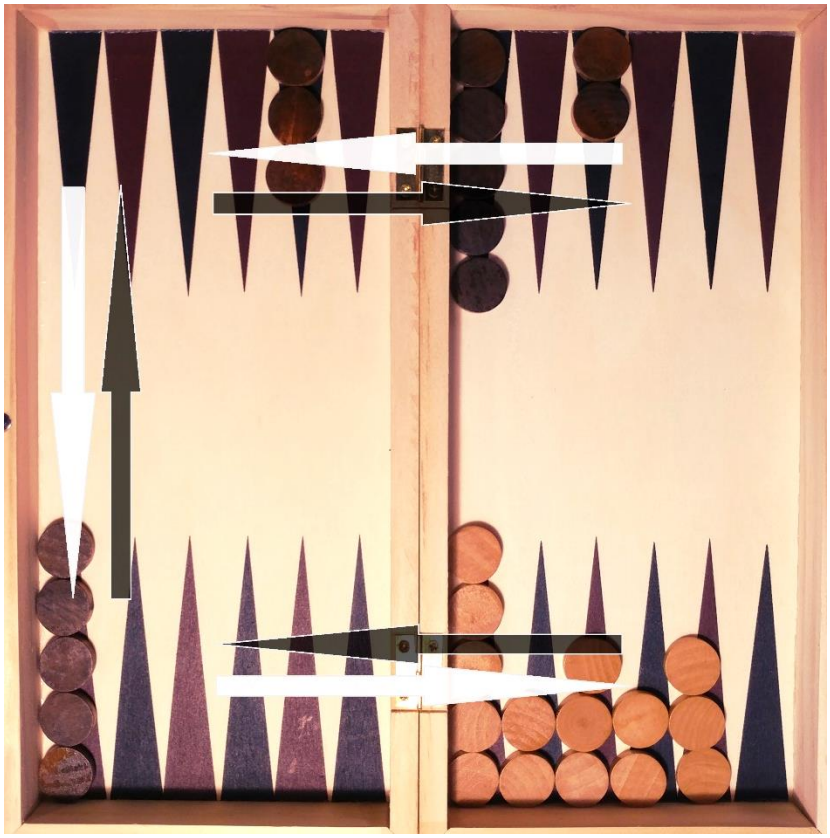

Figura 2.5: Movimiento de las fichas en el tablero

En la figura 2.5 se aprecia lo anteriormente descrito, las fichas blancas comenzaron arriba a la derecha y han ido avanzando hasta abajo a la derecha, es decir, hacia su meta.

Se puede sacar ficha del tablero, finalizando su recorrido y acercando la victoria, si al lanzar dado da como resultado un numero más del numero de pasos necesarios para llegar a la última casilla. Por ejemplo si en la figura 2.5 se tiraran los dados, jugara

el jugador de las blancas, y saliera un 2 y un 3 una posible jugada sería mover una ficha del pico 23 y sacarla, y otra ficha del pico 22 y sacarla.

2.4 Normas de juego

El juego tiene una serie de reglas importantes aplicables al movimiento, condición de victoria, capturar y liberar fichas.

2.4.1 Movimiento

Los jugadores tendrán dos dados, generalmente diferenciados, para movimiento de sus fichas. Supongamos que tendrán un dado A y un dado B y se lanzan, entonces pueden suceder una serie de posibilidades:

1. Avanzar un ficha las posiciones que indique el dado A y avanzar la misma ficha u otra ficha las posiciones que indique el dado B.
2. Avanzar un ficha las posiciones que indique el dado B y avanzar la misma ficha u otra ficha las posiciones que indique el dado A.
3. En el caso de que no puedan realizarse dos movimientos, es decir, ni con A y B ni con B y A entonces se deberá usar el valor del dado mayor para realizar un solo movimiento.
4. En el caso que aún no se hayan podido realizar movimientos entonces se moverá con el otro dado, el que no es mayor.

En el caso que el valor de los dados A y B sean iguales, entonces el jugador podrá hacer hasta cuatro movimientos en cualquiera de sus combinaciones. Podrá mover una ficha dada u otra suya 4, 3, 2, 1 veces. Siempre agotando el número máximo de movimientos.

Una ficha puede moverse a un destino donde no haya ninguna ficha, donde haya cualquier número de fichas de su mismo color o donde haya una sola ficha enemiga, en

este último caso la ficha enemiga se captura y se pone en la barra (fuera de juego temporalmente).

Es muy importante aclarar, aunque ya se interpretaba de las normas numeradas, que cuando se vaya a mover una sola ficha esta no llega al destino directamente, si por ejemplo el dado A sale 6 y el B sale 4 la ficha no se moverá directamente 10 casillas sino que deberá primero avanzar 6 casillas y después 4 casillas. Por lo que se darán situaciones en las que el movimiento no será posible.

En el caso de mover fichas solo será posible pasar el turno si no se puede mover ninguna ficha propia del tablero.

2.4.2 Liberar fichas

Si el enemigo ha capturado fichas entonces se tienen fichas en la barra, es decir, fichas que no estan en el tablero pero que tampoco han acabado, estan fuera de juego temporalmente. La barra en el caso de las figuras 2.3, 2.4 ó 2.5, que son el mismo tipo de tablero, es la parte central que recorre el tablero de arriba hacia abajo situado entre las posiciones 6 y 7, y las posiciones 18 y 19.

Al principio de cada turno el jugador tras tirar los dados deberá volver a poner en el tablero las fichas que tenga en la barra, estas deberán ponerse en el cuadrante de Inicio de su color, de hecho hasta que no se hayan puesto todas las fichas de la barra en el campo de juego no podrá mover fichas. Si se da la situación en la que el jugador no puede poner ficha de la barra al tablero se verá obligado a pasar el turno hasta que pueda poner todas las fichas de la barra de nuevo al juego.

En algunas versiones del juego la liberación de fichas es un tanto diferente, hay casos en los que si no se pueden poner se sigue jugando y otros en los que las fichas deben ponerse en la casilla que marque el valor de los dados. Por razones de simplicidad hemos visto más facil y a la vez entretenido dar libertad al jugador de donde quiera poner su ficha en el tablero siempre que sea en el cuadrante que le corresponda.

2.4.3 Ganar

Para que el jugador se acerque a la victoria deberá ir sacando las fichas del tablero por el cuadrante de Meta. Para que esto pueda suceder es necesario que las 15 fichas del jugador estén en el cuadrante de Meta tras lo cual deberá salir un valor exactamente con un movimiento más al límite del tablero.

Si tenemos una ficha blanca en el pico 24 necesitará un solo movimiento de una unidad para poder salir del tablero. El caso opuesto sería que una ficha negra estuviera en el pico 1 por lo que de igual manera necesitaría igualmente un movimiento de una unidad. Si en estos dos casos salieran dados con valores superiores a uno, la ficha no podrá, en el pico mencionado, moverse.

De forma que el jugador ganador será el primero que saque, o libere, lo cual es lo mismo, todas las fichas del tablero.

Es de interés mencionar que pueden haber tres tipos de victoria, la primera, que es la común, es que ambos jugadores saquen fichas pero uno las saque todas. Después esta la victoria Gammon que se da cuando el jugador contrario no ha sacado ninguna ficha y la victoria aplastante, Backgammon, se da cuando no solo no ha sacado ninguna ficha sino que además aun tiene fichas en la tabla o en su cuadrante Inicial. El tipo de victoria cobra gran interés cuando se realizan apuestas.

2.4.4 Apuestas

Junto a todo ello en este juego pueden hacerse apuestas, generalmente suelen ser para los torneos de Backgammon, generalmente con un dado específico para este cometido, aunque por motivos de simplicidad no hemos agregado todo el conglomerado de normas de apuestas al trabajo, es interesante mencionarlo.

Inicialmente se apuesta un punto que se lo lleva el primer jugador que saque todas las fichas del tablero. En cualquier momento del juego el contrincante puede aumentar la apuesta y el jugador puede retirarse y perder o aceptar y continuar la

partida. Si un jugador dobla ya no podrá volver a hacerlo hasta que el otro jugador doble, en el caso de que quiera hacerlo.

Así que en el caso de las apuestas se pueden ganar de tres maneras, sacando todas las fichas antes que el contrincante, ganar si el contrincante abandona o doblar la apuesta con el dado de apuestas y que el contrincante abandone.

En funcion del tipo de victoria se pueden otorgar más puntos, si la victoria es Gammon se duplican los puntos que gana el jugador y si es Backgammon se triplican.

2.5 Consejos para ganar

Es un juego con un gran nivel de azar debido a los dos dados, pero de igual manera tiene un gran componente de estrategia. Vamos a dar algunos consejos sencillos:

- ✓ Evitar dejar fichas en solitario ya que el contrincante podría capturarla.
- ✓ Si se sigue una estrategia de juego donde se necesita dejar fichas en solitario es una buena idea dejarlas a más de 7 posiciones del adversario.
- ✓ Tener fichas repartidas por el tablero durante varias fases del juego es una buena estrategia.
- ✓ Tener fichas en el tablero interno siempre es buena idea ya que se complica la salida al rival o se acerca a tener fichas en la Meta.
- ✓ Capturar cuando suponga algún tipo de ventaja, en otro caso no es recomendado.

2.6 Pequeño estudio sobre Backgammon

Este juego de mesa en España por regla general o bien ni se conoce o bien solo conocen el nombre pero no como jugar, es decir, solo conocen parcialmente las reglas o ninguna.

Es por ello que realicé una pequeña encuesta para saber si la gente conocía el juego y su rango de edad. Las preguntas realizadas eran dinámicas, es decir, en funcion

de la respuesta acababa el cuestionario o seguí preguntando, tal como se ve en la figura 2.6.

Figura 2.6: Preguntas y respuestas de la encuesta

Los resultados fueron los esperados, aunque cierto es que podría haber un sesgo por la edad ya que la mayoría de respuestas eran de gente joven, de 18-30, y además pertenecientes a la comunidad universitaria ya que es ahí de donde se realizaron más encuestas. En la figura 2.7 se observan los resultados más exactamente.

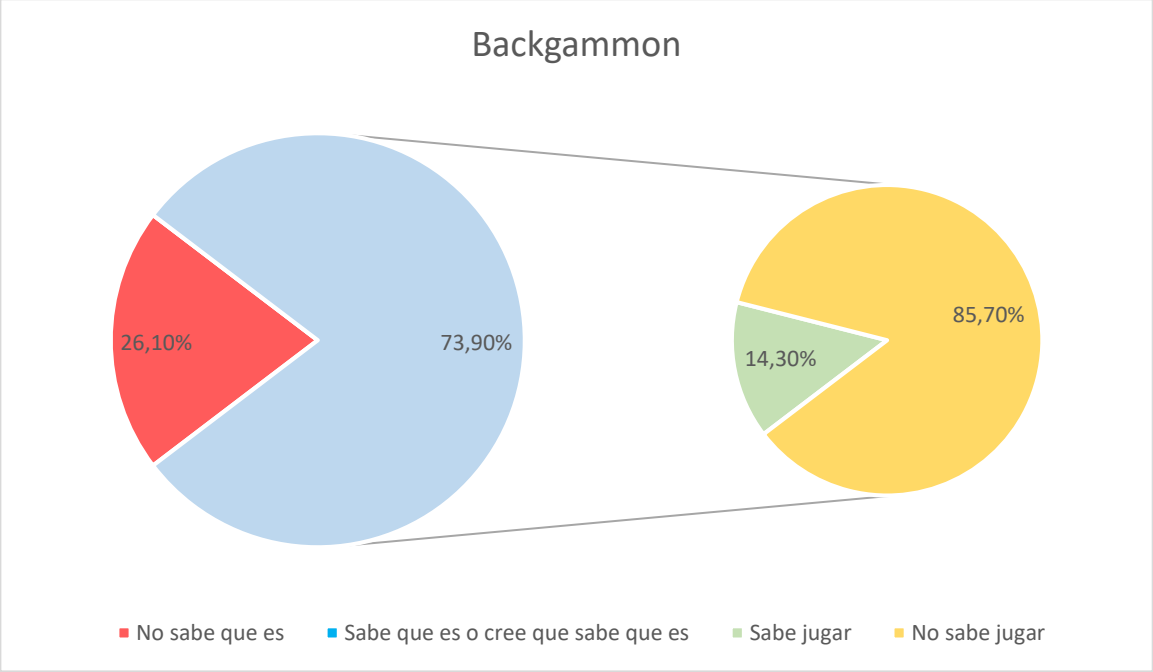

Figura 2.7: Estadísticas de conocimiento de Backgammon

De los datos extraemos que la mayoría sabe que es o al menos han odio hablar del juego por lo que conocen al menos la palabra Backgammon. Sin embargo la mayoría de encuestados no sabían jugar al juego de mesa, o bien solo conocían algunas de las reglas o bien ninguna. Sobre la edad de los encuestados los resultados se observan en la figura 2.8 y efectivamente la mayoría de encuestados era gente joven.

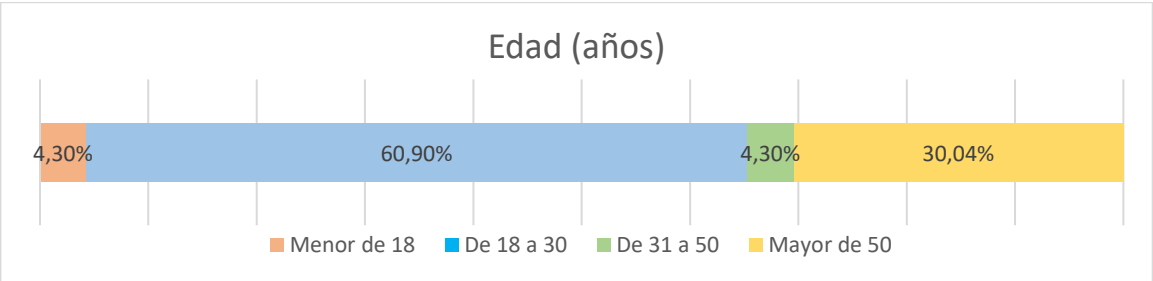

Figura 2.8: Estadísticas de edad de la encuesta

3

Aprendizaje por refuerzo

En este tercer capítulo se explica qué es el aprendizaje automático, detalla qué es el aprendizaje por refuerzo, se describe cual es el problema a resolver en estos así como la aplicación de estas técnicas al juego del Backgammon.

3.1 Aprendizaje automático

Nace en el camino de conseguir IA, Inteligencia artificial. El aprendizaje automático *(machine learning)* es uno de los campos a los que se dedica la ciencia de la computación, más concretamente es un tipo o subcampo de la inteligencia artificial que otorga la capacidad a los computadores de aprender, lo que se pretende es que un programa dado cambie la forma de actuar cuando recibe un o unos nuevos estados.

Con aprendizaje nos referimos cuando en base a experiencias se consigue alterar el comportamiento inicial por uno más adecuado en función del estado en el que se encuentre. Esto se suele conseguir mediante la aplicación de heurísticos o de algoritmos.

Sin embargo, la carga computacional suele ser bastante alta debido a la forma de aprendizaje ya que requiere de numerosos cálculos previos antes de una decisión, ya sea por el hecho de aprender previamente, de observar, buscar, comparar y/o calcular el estado más óptimo, es decir, el más adecuado para el fin del programa, lleva bastante

tiempo del procesador y a veces también de memoria. A pesar de esto, debido la elevada complejidad de ciertos problemas, que de intentar encontrar la mejor decisión buscando en todos los estados posibles no sería viable en tiempo, el aprendizaje automático intenta darles una posible solución adecuada.

Podemos destacar tres tipos de aprendizaje principalmente, y además otros tres derivados o similares a los anteriores. Comenzando por los principales:

- **Aprendizaje supervisado**, en este tipo de aprendizaje se emplea un conjunto conocido de entradas y salidas para con esta información hacer predicciones y a partir de estas ofrecer una respuesta.
- **Aprendizaje no supervisado,** en este tipo de aprendizaje no se dispone de datos necesarios para el entrenamiento, solo se conocen los datos de entrada pero no las salidas, por lo que el planteamiento es estudiar la estructura intrínseca de los datos.
- **Aprendizaje por refuerzo,** este aprendizaje se basa en un agente que realiza una acción y observa su alrededor actualizando su estado y obteniendo una recompensa por ello. Sobre este tipo hablaremos en el siguiente apartado más en profundidad.

Y los otros tres tipos restantes son el **aprendizaje semisupervisado**, que es una combinación del supervisado y no supervisado, la **transducción** que es una forma de aprendizaje supervisado y el **aprendizaje multitarea** que aprende múltiples tareas haciendo uso de una representación compartida para todas ellas.

Como se suele deducir las aplicaciones del aprendizaje automático son muchísimas. Recientemente las más usuales son el reconocimiento de imágenes, reconocimiento de voz, predicciones en muchos ámbitos, desde precios de inmuebles, de la luz, el petróleo; juegos, desde estrategias para ganar al jugador humano como simplemente saber dónde moverse y evitar obstáculos, también se usa en coches

autónomos y por supuesto en motores de recomendación, desde la mismísima Netflix hasta Spotify, toda nuestra actividad en esas plataformas es recogida y monitorizada para ofrecernos lo más recomendado a nuestros gustos.

Incluso en un futuro no muy lejano el aprendizaje automático podrá mejorar nuestra calidad de vida con mejoras en diagnósticos médicos [18].

3.2 Aprendizaje por refuerzo

El aprendizaje por refuerzo [1][6][21] está relacionado con la psicología conductista. Esta hace hincapié en describir las leyes generales que rigen la conducta voluntaria, para ello hacen uso del condicionamiento operante [22].

El **condicionamiento operante** se basa en una serie de condicionantes de conducta, ante una serie de respuestas existirán varias casuísticas. Si se recompensa una respuesta esta tendrá una alta probabilidad de repetirse. A esto Burrhus Frederic Skinner, padre del condicionamiento operante, lo llamó refuerzo positivo o recompensa. Así pues ante una respuesta que no sea reforzada probablemente deje de repetirse y las respuestas no deseadas por consecuencias no deseables serán castigadas. Skinner probó este condicionamiento experimentando con ratas.

Esto aplicado a una máquina se basa usar este principio de condicionamiento de forma que exista un aprendizaje, recompensando los estados que propician una victoria y castigando los que llevan a la derrota. A diferencia del experimento de Skinner [22], o de la existencia de la aplicación de este concepto a en la vida diaria como por ejemplo premiar a quien, en un principio, más a estudiado con una nota más alta, una máquina permite entrenarse cientos, miles o millones de veces haciendo que acabe adoptando la que maximice la recompensa acumulada.

Una de las características más importantes y que hacen interesante el aprendizaje por refuerzo es que permite correlacionar las respuestas de un momento

dado con las consecuencias que tendrá a largo plazo, ya que toda respuesta se evalúa y tiene un peso a largo plazo. [6].

Para definir un problema que deba ser resuelto con aprendizaje por refuerzo generalmente necesitamos de un agente software que pueda ser entrenado y ante un estado inicial realice una acción, esta acción cambiara el ambiente, es decir, el agente tendrá un nuevo estado y a cambio una recompensa. Esto se aprecia muy bien en la figura 3.1. El flujo sería nuevo estado -> acción -> recompensa, y vuelta a empezar [21].

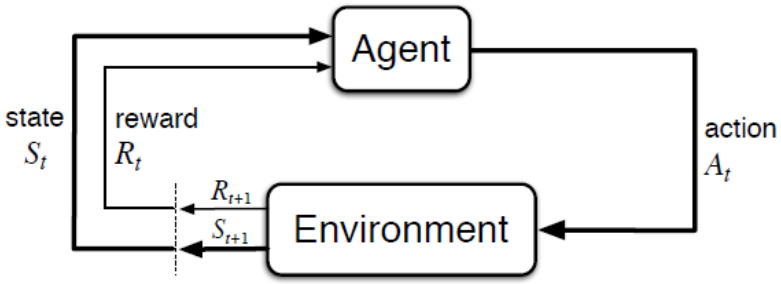

Figura 3.1: Interacción entre agente y ambiente [5]

A parte de la recompensa que tenga el agente tras una acción, también existirá una recompensa acumulada que consiste en haber ganado o no anteriormente.

Partiendo de las premisas mencionadas, para definir el problema que debe ser resuelto, es decir, ganar contra otro jugador en el juego Backgammon podríamos definirlo de la siguiente manera. El agente, llamémoslo JugadorEntrenable, con un estado, es decir, turno, cantidad de fichas, fichas fuera de juego y fichas que han acabado, dentro del ambiente, un tablero con sus 24 picos, deberá realizar una acción, es decir, mover la ficha adecuada la cantidad de veces que indiquen los dados pero maximizando el refuerzo positivo o recompensa. La experiencia del agente se divide en este caso en partidas, cada una denominada episodio, al finalizar el cual se vuelve al estado inicial. La única recompensa que recibe el agente es al final del juego, según su resultado.

3.3 Programación dinámica

Programación significa optimizar un programa, si hablamos de matemáticas, y dinámica es que tiene algun componente de decisión secuencial, es decir, que la solución a un problema viene dada por una secuencia de decisiones individuales. En nuestro caso una secuencia de movimientos del juego hasta terminar la partida. Por lo que el resultado sería que la programación dinámica es un método de optimización para problemas secuenciales [23].

La programación dinámica pretende resolver problemas dividiendo el problema en subproblemas más pequeños. Debe presentar la propiedad de subestructura óptima lo que significa que cada problema debe poder dividirse en subproblemas más pequeños resolubles de forma óptima y cuando vuelvan a unirse la solución seguirá siendo optima. Hasta aquí los procesos de decisión de Markov (MDP) [1] cumplen las propiedades de subestructura óptima pero en principio no para la propiedad de subproblemas superpuesto [23], se produce cuando hay subproblemas similares o que compartan propiedades. Para esto se puede agregar una función de valor, normalmente denominada ecuación de Bellman o función de valor de Bellman, la cual calcula los valores para una determinada acción.

Junto a la programación dinámica en nuestro caso, esta se combinará con el método del aprendizaje por diferencias temporales TD(0) [1], que explicaremos a continuación, y que es una técnica alternativa a la programación dinámica clásica cuando se desconocen las probabilidades de transición en el entorno. Se basa en aprender mediante interacción con el entorno.

3.4 Aprendizaje por diferencias temporales

Sutton [1] denominó a la asignación de crédito temporal como aprendizaje por diferencias temporales, o simplemente TD (Temporal Difference) [16]. La idea

fundamental de los métodos TD es que el aprendizaje esta basado en la diferencia entre predicciones temporalmente sucesivas.

Este método de aprendizaje posee las ventajas de la programación dinámica, como es la posibilidad de usar las actualizaciones de valor en el momento de exploración de los posibles movimientos, en el caso que nos atañe, y determinar sobre qué estado se debería actualizar la política.

En TD se suele usar como medida del error la diferencia entre predicciones sucesivas en el tiempo. Gracias a lo cual aprende por el hecho de que existen cambios entre estas predicciones sucesivas. Podemos ver la representación matemática del método del subgradiente estocástico TD(0) en una función de estado representada en la figura 3.2.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Figura 3.2: TD(0). [16]

Donde α será la tasa de aprendizaje, es decir, el porcentaje de aprendizaje por cada buena o mala acción, R_{t+1} es la recompensa por el siguiente estado, $V(S_{t+1})$ la evaluación del siguiente estado y la evaluación del estado actual es $V(S_t)$.

En el caso del Backgammon para poder calcular la evaluación de un estado dado será necesario poder codificar el tablero de una forma entendible y relevante para la máquina de forma que distinga entre diferentes estados y a cada uno de esos estados le corresponda un peso o importancia para de esa forma conocer si es positivo o no.

3.4.1 Aprendizaje por diferencias temporales y redes neuronales [14]

El aprendizaje por diferencias temporales con redes neuronales multicapa planteado por Gerald Tesauro [14] es diferente al de este trabajo pero se hace muy interesante hablar de su estudio. Este en cierta medida comparte aspectos con el nuestro por el hecho de usar las diferencias temporales, ya que él no usa exactamente

una función lineal como hemos usado, sino que usa algo basado en redes neuronales multicapa llamado TD(lambda) aplicado a su particular TD-Gammon.

El TD(lambda) que usó Tesauro es una técnica que bajo ciertas condiciones puede acelerar la convergencia del método TD.

Tesauro concluyó en sus investigaciones que intentar diseñar un programa de Backgammon que buscase entre todas las posibles jugadas del tablero no era algo viable debido a la enorme profundidad que tendría, estimó que sobre 10^{20} estados posibles, añadido a esto debido a la aleatoriedad de los dados tampoco era factible usar el método de fuerza bruta, usado con buenos resultados en ajedrez y damas, por las ramificaciones resultantes. Por todo ello un programa de Backgammon deberá confiar en algun heurístico posicional.

En cuanto al Backgammon, o al uso de TD en juegos de dos jugadores, los estados deben codificarse de forma que la máquina entienda la situación dada (el numero de fichas blancas y negras en sus posiciones, fichas capturadas, etc.). Si además se le añadiera la probabilidad de ser golpeado o la fuerza de un bloqueo podría mejorarse el rendimiento general.

La integración de TD(lambda) en un juego como el Backgammon dio sus frutos, y es que en su primera versión funcional perdía contra los mejores jugadores del mundo con un promedio de 0.2 o 0.25. Pero conforme se perfeccionó en su segunda versión ya solo perdía ante auténticos maestros por centésimas. Lo cual quiere decir que el aprendizaje por refuerzo basado en diferencias temporales con redes neuronales multicapa resulta en algo efectivo y abarcable. Y dado que este programa generalmente aprendía de si mismo y no de otros jugadores sus estrategias solían ser algo diferentes a las de los profesionales pero efectivas.

Debido al comportamiento aleatorio de los dados, ya que hablamos de un juego no determinista, la sucesión de los estados tiene un comportamiento estocástico, sin

embargo esto produce una mayor exploración que si de un juego determinista, donde los movimientos son fijos para cada ficha, se tratase.

Es curioso que en un estudio minucioso de las primeras fases del aprendizaje de TD(lambda) haciendo uso de funciones no lineales, la red neuronal en un principio y sin demasiado entrenamiento extrajo ciertos conceptos elementales, como mover fichas, golpear al oponente o no dejar fichas expuestas, que son posibles de expresar con una función lineal, dejando la parte no lineal para los sucesivos entrenamientos.

En conclusión, el aprendizaje por diferencia temporal es una técnica prometedora para el aprendizaje con recompensas retardadas.

La aplicación de este tipo de conceptos de aprendizaje podría ser llevado a otros juegos de dos jugadores con componente estocástico, control de robots e incluso en estrategias de negociaciones financieras.

3.5 Aplicación de aprendizaje por refuerzo en juegos

Llegados a este punto ya hemos introducido numerosos conceptos importantes como las diferencias temporales, la recompensa, etc. y también se ha planteado superficialmente su aplicación en el juego Backgammon.

Para el caso del Backgammon, como con el de otros juegos de mesa cuyo comportamiento no es determinista por los dados y haciendo uso del aprendizaje por diferencias temporales, no se usa la habitual funcion de valor-acción ni de valor-estado, al menos no en el sentido habitual. Ya que este tipo de funciones evalúan el valor del estado en los que el agente tiene la opción de seleccionar una acción, sin embargo la implementación de TD en juegos, y la que implementamos en Backgammon, evalúa las posiciones después de haber realizado el movimiento. A estas funciones ejecutadas con posterioridad las llamaremos funciones de valor-estado-posterior, o *afterstate value functions*. Estas funciones son muy útiles cuando tenemos el conocimiento de una parte inicial del estado pero no podemos conocer estado futuros, ya que en el caso del

Backgammon sin saber qué dados se tienen no se puede saber qué estado de los miles que se dan en el juego debería ser evaluado, es decir, sabemos cómo está el tablero y qué movimientos de nuestras fichas dados los valores de los dados podemos hacer pero no el movimiento que realizará nuestro oponente [20].

Es por esto que este tipo de funciones son ideales ya que la evaluación de los estados se produce tras que el enemigo haya jugado contra nosotros. Además estas funciones son más eficientes ya que no se evalúan jugadas cuyos estados resultantes son iguales, sino tras la jugada observa y aprende.

En resumen, en el caso del Backgammon diseñaremos una funcion de evaluación siguiendo el esquema de aprendizaje por diferencias temporales y dicha función será de valor-estado-posterior, o afterstate value function, con la que tras la jugada el jugador enemigo, que llamaremos en adelante min, evaluaremos el estado de juego y actualizaremos en el caso de que sea final, en otro caso el jugador inteligente, que llamaremos en adelante MAX, realizara una jugada y aprenderá. Y vuelta a comenzar hasta que la partida finalice.

4

Python

En el presente capítulo daremos un repaso a todo el lenguaje de programación Python desde que es, para lo que se usa hoy en día, como se debe escribir en Python ya que existen unas normas, las PEP, de las que también hablaremos, entre otros puntos. El trabajo está realizado usando la versión 3.7 del interprete. Damos por hecho que quien lea este capitulo tiene ciertos conocimientos de programación o al menos está familiarizado con ciertos conceptos de lenguajes.

4.1 ¿Qué es?

Python es un lenguaje de programación **multiparadigma** porque permite programar de forma imperativa, orientada a objetos e incluso funcional.

Además de todo ello tenemos que Python es un lenguaje de **tipado dinámico**, es decir, una misma variable puede ser de diferente tipo en diferente momento, e incluso durante la vida de cualquier variable no se hace necesario en ningun momento especificar tipo. Deja en manos del programador usar correctamente los métodos que puedan tener asociados una variable que en un momento dado sea un objeto pero que después no lo sea. Esto es algo común entre los lenguajes interpretados.

A pesar de ello es posible definir un tipo explicito para una variable, un parámetro o el valor de retorno de una función.

Python, como hemos mencionado, es un lenguaje **interpretado**, por lo que la mayoría de sus instrucciones, de líneas de código, se ejecutan directamente pasándose a código máquina sin una compilación previa. Más específicamente, el intérprete es quien ejecuta el programa directamente, el cual pasa cada sentencia del programa en un conjunto de una o más de una subrutina ya compiladas a código máquina pudiendo ya ser ejecutado. En la figura 4.1 podemos apreciar que no hay necesidad de previa compilación, todo se hace al vuelo. Una diferencia importante si lo comparamos con la figura 4.2 que requiere de una previa compilación para poder ser ejecutado.

Figura 4.1: Esquema simplificado de lenguaje interpretado

Figura 4.2: Esquema simplificado de lenguaje compilado

Aunque el hecho de que sea interpretado da algunos beneficios también puede ocasionar ciertos errores y desventajas. Una de las desventajas más importante es que cuando se interpreta un programa en Python suele ser más lento que si se estuviera ejecutando el programa ya previamente compilado, aunque en ciertas ocasiones el tiempo global de interpretar puede ser menor que el de compilar y después ejecutar el programa como sería en otros lenguajes. Algunos de los errores más importantes también de este hecho, y del tipado dinámico, es que se puede intentar acceder a una variable u objeto que en cierto momento su tipo fue cambiado dando lugar a errores. Básicamente en este tipo de lenguajes, los interpretado, dado que no en todas las situaciones se compilan todas las líneas del codigo, pueden existir errores en el código, que hasta que la línea del error no se ejecute el error no aparecerá. Es por ello que el hecho de pasar pruebas se hace especialmente importante en este tipo de lenguajes, debemos asegurarnos que al menos todas las líneas de nuestro código sean correctas.

Además Python posee la capacidad de **conteo de referencias**, es decir, es capaz de contabilizar las veces que un recurso está siendo referido. De esta forma cuando nunca se vuelve a referenciar ese recurso, este podrá ser liberado de la memoria. Aunque tiene algunas desventajas, como que si las referencias forman un ciclo, los recursos a los que se referencian nunca serán liberados, aunque a partir de la versión 2.0 del lenguaje se agregó un sistema de recolección de basura solucionando este problema inical.

4.2 Orígenes y futuro

Python nació en los Países Bajos de la mano de Guido van Rossum a finales de los años ochenta en el CWI, centro para las matemáticas y la informatica, como heredero del lenguaje ABC desarrollado en ese mismo centro años antes. Aunque esta primera versión no es exactamente la que conocemos hoy en día.

El nombre curiosamente proviene de los Monty Python ya que Rossum sentía gran afición por estos humoristas.

A mediados de los noventa Python ya alcanzó una primera versión, esta versión incluía herencia, manejo de excepciones, funciones, tipos str, list, dict, funciones lambda, map, filter, entre otras características.

Rossum continuó su trabajo con Python en el CNRI, Corporación de Iniciativas Nacionales de Investigación, en Virginia, donde lanzó más versiones del lenguaje. Ya sobre el año dos mil el equipo de desarrolladores de Python se pasaron a BeOpen, tras un tiempo lanzaron la versión 2.0 de Python trayendo novedades como las listas por comprensión basadas en el lenguaje funcional Haskell. Tras esta versión los desarrolladores se unieron a Digital Creations. Durante años posteriores se sucedieron una serie de acuerdos referidos a la licencia de uso Python, la cual llegó a buen puerto puesto que la hicieron compatible bajo GNU GPL, es decir, Licencia Publica General de GNU.

Y llegamos a la versión integrada en nuestro Backgammon, Python 3, que agregó excepciones al comparar tipos diferentes, la impresión por consola es una funcion, se permiten caracteres tales como “ñ” o la “ú” ya que ahora las cadenas estan codificadas en UTF-8, las variables de los bucles son locales, entre otros cambios.

De hecho recientemente se comunicó que en 2020 se le cerraba el soporte a Python 2.7 y de hecho según calcula JetBrains el 87% de los nuevos desarrollos son en Python 3.X, es decir, nueve de cada diez desarrolladores han usado Python 3 durante 2019.

Y su uso más extendido, al menos hasta 2019, como se puede ver en la figura 4.3 es para el análisis de datos y el desarrollo web, dos de las tareas más relevantes del panorama actual.

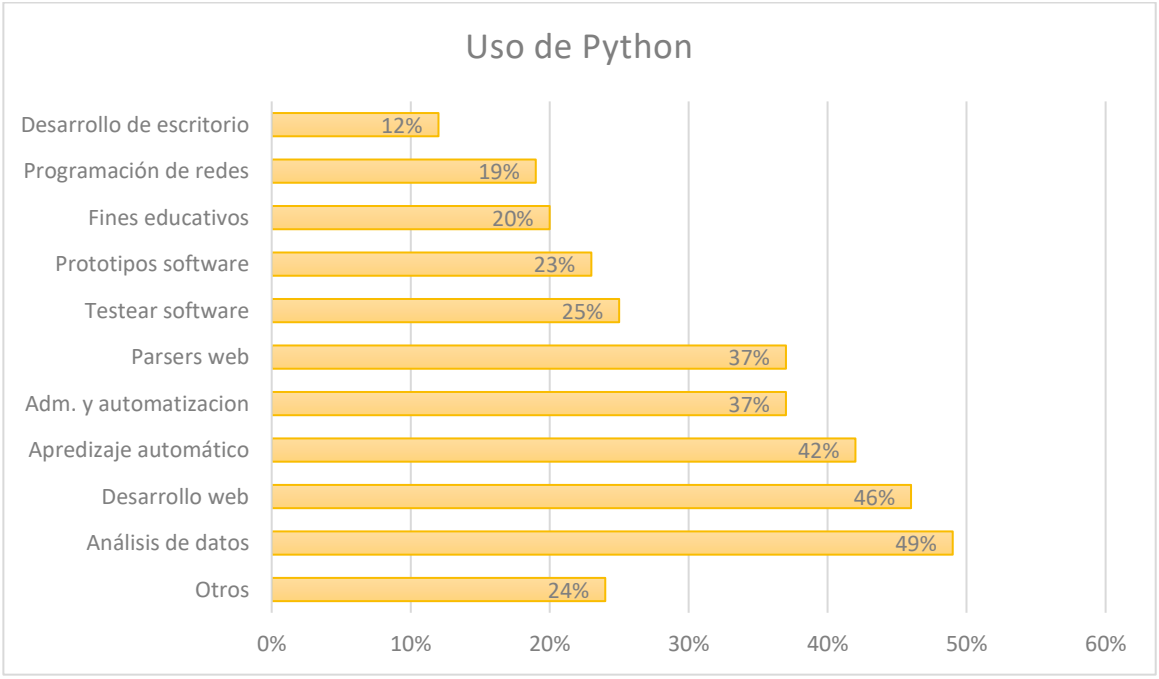

Figura 4.3: Uso de Python en la actualidad. [13]

En el aprendizaje automático Python ha ido adquiriendo una gran relevancia ya que como hemos visto es un lenguaje sencillo, en principio, flexible pero robusto a la vez que escalable, y un punto muy importante es que es gratuito, esto sumado a la

gran cantidad de bibliotecas para tal fin que existen, ha hecho que para gran parte del sector dedicado a estos fines sea su lenguaje predilecto.

4.3 PEP

La Propuestas de Mejora de Python, o *Python Enhancement Proposals*[2], son una serie de documentos relacionados al diseño que ofrecen al programador información de diferentes tipos, desde una guía de estilo a la hora de escribir en Python, formas de seguir estándares de la comunidad, bugs corregidos, módulos obsoletos aun presentes entre otros muchos.

Uno de los más conocidos y usados es **PEP8**, la cual es una propuesta de estilo para programar con el lenguaje Python. Alguna de las más importantes y que se siguen a lo largo de este trabajo son:

- ✓ Usar 4 espacios en vez de tabular, y nunca mezclar tabuladores y espacios.
- ✓ Dos líneas en blanco antes y después de las clases, métodos dentro de clases con una linea en blanco y funciones fuera de clases separados por dos líneas en blanco.
- ✓ Espacio en blanco después de coma o dos puntos pero nunca espacios innecesarios.
- ✓ Importar módulos de Python primero y módulos propios después.
- ✓ Constantes en mayúsculas.
- ✓ Métodos protegidos con guion bajo al principio.
- ✓ Usar variables protegidas con doble guion bajo delante.

Hace cierto tiempo, Tim Peters, un entusiasta de este lenguaje con cierta relevancia en el mundo Python plasmó unos principios de diseño, que de hecho están recogidos en PEP20 que dice así.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Peters, T. (2004). The Zen of Python. *Python Enhancement Proposals* (PEP20).

Recuperado de https://www.python.org/dev/peps/pep-0020/

De hecho el propio Rossum integró estos propósitos en el propio lenguaje, ya que se pueden consultar directamente en el intérprete escribiendo:

```
>>> import this
```

Intentando seguir un poco por el camino de estilo que planteó Peters, hemos seguido varios puntos clave, como, explícito mejor que implícito, simple mejor que complejo, **disperso mejor que denso**, de hecho este junto a la documentación asociada a cada clase y método ha sucedido que un método que podría quedarse en 15 líneas pero acabe estando bastante más disperso pero más claro y documentado.

4.4 Características del lenguaje y librerías

En este apartado hablaremos de ciertas características sobre como programar con Python [2][3][4] enfocándonos en elementos usados en el trabajo. De igual manera hablaremos sobre las librerías que hemos necesitado usar, dando algunos detalles, incluida la necesaria para la interfaz gráfica.

Por ejemplo, si quisiéramos hacer un pequeño programa que sumase todos los numeros enteros hasta 1000, podríamos hacer como en la figura 4.4.

```
numero = 0
for i in range(1001):
    if i % 2 == 0:
        numero += i
print(numero)
```

Figura 4.4: Ejemplo Python de sumar 1000 números y mostrar el resultado

Sin embargo Python pone a nuestra disposición características interesantes, como `sum(lista)` que suma todo el contenido de una lista, o `range(min, max, suc)` que permite crear dinámicamente una lista inmutable de números enteros entre min y max en sucesión aritmética (suc). Este suele ser también ideal para hacer bucles "for" cuando se dan otras casuísticas, como en el codigo de la figura 4.4. Sin embargo en el caso de la figura 4.4 podríamos dejarlo en una sola línea tal como se ve en la figura 4.5.

```
print(sum(range(0, 1001, 2)))
```

Figura 4.5: Ejemplo Python de sumar 1000 números y mostrar el resultado en solo una línea

Otra característica que podemos apreciar es la ausencia de paréntesis en bucles y sentencias condicionales, y es que en Python los paréntesis si no son obligatorios porque se trate de los argumentos de un método o funcion, es mejor no usarlos.

El array como tal no suele ser la estructura de datos más usada generalmente en Python, por regla general es preferible usar listas y cuando son pocos datos una

tupla. De forma básica las listas (`list`) en Python estan definidas como una pila, es decir, siguen una estructura de almacenamiento *LIFO*, ultimo en entrar primero en salir, con los métodos `append(elemento)`, que agrega un nuevo elemento al final, y `pop()` que retorna y elimina el último elemento. Aunque también puede ser usado como un array dinámico ya que permite acceso y asignación usando lista[posición] como si de un array se tratase. Otra utilidad muy util de las listas es que simplemente usando la variable de lista como sentencia condicional retorna verdadero si hay elementos y falso si no hay.

```
In [3]:
lista = [1, 2, 3]
if lista:
    lista.append(4)
    lista.append(5)
    lista[3] = 100
    ultimo_elemento = lista.pop()

print(lista, ultimo_elemento)
```

```
[1, 2, 3, 100] 5
```

Figura 4.6: Ejemplo Python de operaciones con lista con resultado de ejecución

En la figura 4.6 podemos observar que se crea una lista de tres elementos [1, 2, 3] inicialmente, aunque la declaración entre corchetes pueda parecerse a un array en otros lenguajes, en Python 3 se crea una lista. Tras ello se comprueba que la lista no esté vacía y si tiene elementos, que claramente en este caso si los tendrá, se agrega el número 4 y el 5 a la lista, después se accede a la posicion 4 y se cambia el valor por 100, esto ilustra que es una estructura de datos usable como array, como ArrayList del lenguaje JAVA y como pila, tres ello se retira y almacena el último elemento, es decir, se quita de la lista y se almacena el 5. Aquí se debe mencionar un detalle importante, aunque la variable `ultimo_elemento` está definida dentro del bloque de la sentencia condicional, esta es accesible también desde fuera de ese bloque de código, desde Python 3 esto no sucede en el uso de bucles pero sí en sentencias condicionales. La última línea simplemente imprime por pantalla en la consola el estado de la lista y el elemento

extraído. Por lo que podemos decir que `list` es una herramienta muy potente en lo que su uso como estructuras de datos se refiere.

Otra característica de Python es la compresión de listas, un rasgo heredado de la programación funcional en Haskell y que gracias a ella, entre otros usos, podemos crear en una sola línea una lista con elementos, como se puede ver en la figura 4.7, e incluso hacer listas de listas con condiciones específicas de agregación de elementos.

```
In [5]:
lista = [i for i in range(2,5)]
print(lista)

[2, 3, 4]
```

Figura 4.7: Ejemplo Python compresión de listas

Los ejemplos previos se pueden ejecutar tal cual, es decir, a diferencia de otros lenguajes no es necesario un método o procedimiento main, sino que directamente si ejecutas un fichero desde la consola del interprete de Python con las figuras 4.6 o 4.7 podrás ver el resultado directamente, de hecho esos ejemplos fueron ejecutados desde Jupyter y de ahí que pueda observarse el resultado, por otra parte las figuras 4.4 y 4.5 se ejecutarían de igual forma pero no veríamos los resultados. El concepto que pretendo dejar claro es que aquí estamos usando el lenguaje como si se tratara de uno imperativo, sentencias ejecutadas una detrás de otra, también podrían llamarse a procedimientos o funciones.

```
def suma_dos(numero1, numero2):
    return numero1 + numero2
```

Figura 4.8: Ejemplo Python definición de función

En la figura 4.8 podemos apreciar la definición de una función con numero1 y numero2 como entrada y devuelve la suma de ambos numeros.

Por otra parte, la creación de objetos se realiza de manera sencilla pero presenta ciertos cambios con respecto al lenguaje orientado a objetos de referencia JAVA.

```
In [15]: class Padre:

             def __init__(self, i: int):
                 self.variable = "Hola soy variable de instancia {}".format(i)


         class MiClase(Padre):

             def __init__(self, i: int):
                 super().__init__(i)

             def metodo(self):
                 print(self.variable, " -> ", self.__class__.__name__)


         clase1 = MiClase(5)
         clase1.metodo()

         Hola soy variable de instancia 5  ->  MiClase
```

Figura 4.9: Ejemplo Python de clases y herencia

Como podemos ver en la figura 4.9 se define una clase Padre y una clase que hereda de Padre llamada Miclase, en Python los constructores se crean como se puede ver en la figura 4.11.

```
def __init__(self, parametro1, parametro2=None)
```

Figura 4.10: Constructor

Donde todo lo que hay posterior a self son los argumentos de entrada que deberán pasarle cuando se cree un objeto. En Python no puede haber más de un constructor para cada clase, si se necesitan varios pasar diferentes cantidades de argumentos se pueden definir valores por defecto simplemente igualando el parámetro opcional a un valor por defecto.

En cuanto a la herencia Python permite herencia múltiple, en este caso MiClase hereda solo de Padre, y este a su vez de object ya que todas las clases heredan de object, porque en la definición de la clase MiClase se le ha especificado, suele seguir una estructura así como se puede observar en la figura 4.11.

```
class Clase(Superclase1, Superclase2)
```

Figura 4.11: Clase y herencia

Otra característica importante es que para referenciar a los métodos de la clase o a variables de clase o de instancia es necesario usar self, este es obligatorio tenerlo como parámetro en la definición de constructor, de los métodos o de la redefinición de métodos por defecto [15].

4.4.1 Tkinter, copy y random

Estas son las herramientas presentes por defecto en Python 3 que han sido de vital importancia durante la elaboración de este trabajo.

Tkinter es un *binding*, es decir, una adaptación de una biblioteca ya presente en muchos sistemas, en este caso es un adaptación de la biblioteca gráfica Tcl/Tk. Con todo ello Tkinter es el paquete para crear interfaces graficas de usuario de facto en Python. A pesar de esto se hace relativamente complejo crear programas muy grandes o con demasiada información usando este paquete porque todo el diseño debe programase, aunque con paciencia se podría.

Comenzar a usar Tkinter es sencillo, solo es necesario importar el paquete correspondiente, crear un objeto raíz y lanzarlo como se aprecia en la figura 4.12 y el resultado de la ejecución en la figura 4.13.

```
from tkinter import *

raiz = Tk()
Label(raiz, text="Ejemplo Python\nSergio Tineo").pack()
raiz.mainloop()
```

Figura 4.12: Ejemplo Python Tkinter

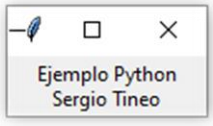

Figura 4.13:Ventana realizada con Tkinter

En la figura 4.12 además de crear y lanzar la ventana tiene una etiqueta que se posicionó en la ventana usando la gestión automática de inserción de elementos pack, este sigue una invocación tal como se ve en la figura 4.14.

```
objeto_de_ventana.pack()
```

Figura 4.14: Gestor de geometría pack

copy es un módulo integrado por defecto en Python que sirve para copiar objetos. Generalmente cuando se usan listas y se necesitan tener dos listas, una original y otra con cambios por ejemplo, si se hiciese una asignación tal cual (lista2 = lista1) sería incorrecto ya que si se modificase la primera lista se modificaría la segunda. Para solventar esto existe el módulo mencionado y además las propias listas integran un método `lista.copy()` que realiza una "copia superficial" del contenido de la lista, es decir, que copia la lista pero no el contenido si son objetos, de forma que si alteramos una de los listas no se alteraría la otra pero si alteramos uno de los objetos y ambas listas tienen la referencia a ese objeto, ese objeto aparecería cambiado en ambas listas. Para esto último el módulo copy integra el método deepcopy [19] que realiza una copia tanto del elemento pasado como argumento como de las posibles referencias a otro objetos que tenga ese elemento. Para usarlo solo hay que importar el módulo y llamar al método como se ve en la figura 4.15.

```
import copy
lista_copiada = copy.deepcopy(lista_original)
```

Figura 4.15: Deepcopy

random es el módulo integrado en Python de generación de números pseudoaleatorios que permite generar números enteros, comprendidos entre dos rangos, flotantes e incluso la opción de barajar listas. Por lo que su utilidad es bastante grande en un juego como el Backgammon donde se necesitan dos numeros aleatorios por jugada, entre otros usos.

5

Análisis y Diseño Backgammon

En este quinto capítulo se detalla el análisis previo a la implementación del juego Backgammon y el diseño del programa, correspondiendo a la **primera iteración** de la metodología de desarrollo, es decir, antes de la incorporación del aprendizaje por refuerzo. Dado que el analisis y diseño de la aplicación final está dividido en dos partes hemos visto más adecuado juntar en un mismo capítulo la etapa de análisis y la etapa de diseño.

5.1 Análisis

En este apartado se detallará lo que pretendemos que realice el programa una vez este haya sido finalizado, al menos lo correspondiente a esta primera iteración. Así mismo dada la dimensión de la aplicación y además su división en dos iterables, se deberían generar dos **Documento General de Requisitos**, el primero lo vamos a integrar a continuación eliminando elementos innecesarios con el fin de no extender más de lo necesario la memoria ya que se repetiría demasiado los mismos detalles.

5.1.1 Introducción

5.1.1.1 Objetivos

Estos fueron detallados en el capítulo 1 apartado 2.

5.1.1.2 Alcance

El alcance del presente trabajo abarcan todos los usuarios que quieran jugar al Backgammon así como ver competir un sistema sin aprendizaje.

5.1.1.3 Definiciones

Jugador, es el usuario que jugará contra la máquina.

Jugada, es el conjunto de acciones de lanzar los dados y realizar los movimientos necesarios o pasar turno para que sea el turno del siguiente jugador.

Sistema sin aprendizaje, es un sistema que actúa dentro de las normas del juego pero de forma aleatoria.

5.1.1.4 Resumen

Se va a realizar un programa con el que se pueda llevar a cabo diferentes acciones usando como entorno el juego de mesa Backgammon, estas acciones será ver el comportamiento de un jugador no controlable por el usuario humano, esto es una partida automatizada, en la que se podrá ver cada una de las jugadas y el resultado. Otro uso del programa será que el jugador mediante una interfaz gráfica de usuario pueda jugar según las normas explicadas en el capítulo 2.

5.1.2 Participantes

5.1.2.1 Detalle de los participantes

Nombre	Descripción
Jugador	Jugará al Backgammon o podrá lanzar y ver una partida automatizada del sistema sin aprendizaje.

Tabla 5.1: Detalle de participantes

5.1.2.2 Perfil del participante

Representante	Jugador
Descripción	Podrá usar el programa en su totalidad
Tipo	Persona que quiera jugar al Backgammon o ver como juega el sistema.
Criterio de éxito	Que sepa lanzar un script Python

Tabla 5.2: Perfil de participante

5.1.2.3 Alternativas y competencia

Actualmente existen numerosos juegos de Backgammon tanto online como en forma de aplicación, la mayoría sin técnicas de aprendizaje.

5.1.3 Visión general del producto

5.1.3.1 Entorno

El entorno normal de funcionamiento será cualquier ordenador o computadora capaz de ejecutar el interprete de Python. Más detalles en el apéndice.

5.1.3.2 Características

Beneficios para el cliente	**Descripción**
Jugar a Backgammon	Poder ver y realizar jugadas en el juego Backgammon
Ver juego automatizado	Se podrá ver la maquina jugando contra si misma al Backgammon
Jugadas validas	Se comprobará que las jugadas de cualquier jugador siempre sean válidas

Tabla 5.3: Características del entorno

5.1.4 Requisitos funcionales

RF1. El jugador podrá realizar movimientos o pasar turno en el juego Backgammon y ver su jugada.

RF2. El jugador podrá tirar los dados en el juego y observar cuales el número de movimiento posibles.

RF2. Es sistema podrá jugar contra si mismo y mostrar al jugador el estado de la partida si el jugador así lo quiere y observarlo.

Gracias lo exacto de lo que se desea desarrollar, podemos afirmar que se cumplen los requisitos de consistencia y compleción gracias a que es un sistema bien acotado.

5.1.5 Requisitos no funcionales

RNF1. *Operacional.* Cualquier acción que realice el jugador o el sistema deberá ser validada para el cumplimiento de las normas de juego en todo momento.

RNF2. *De interfaz.* El sistema debe tolerar entradas de datos incorrectos.

RNF3. *De documentación.* Se facilitará la refactorización o mejora de código documentando cada parte del programa.

RNF4. *De mantenibilidad.* Se podrá ver todo el flujo de ejecución para cada tarea si así lo desea el desarrollador.

Junto a cada requisito no funcional se puede observar la categoría correspondiente que tiene, siguiendo las categorías del documento [17].

Se hace necesario matizar que hay ciertos puntos como las directivas del proyecto, los costes, las licencias, etc. que no se han tratado en este DGR integrado en el presente capítulo para evitar una extensión innecesaria. Es por ello que la intencionalidad de este apartado de análisis se ha cumplido pero no puede decirse que se haya integrado un Documento General de Requisitos al completo.

5.2 Concepto preliminar de la interfaz de usuario

Este apartado trata sobre el diseño de la interfaz gráfica de usuario de usando lo que se conoce como *mockups* o maqueta.

En la figura 5.1 se muestra el estado inicial de la aplicación con GUI. En este primer estado la partida aun no ha empezado, la única acción que se podría realizar es pulsar en empezar para empezar partida.

En la figura 5.2 se muestra el estado habitual del juego, pudiendo lanzar dados, poner la posición de la ficha que se quiere mover (origen) y realizar el movimiento. Pudiendo en todo momento ver el estado actual de la partida (TABLERO). En la parte inferior se mostrará información relevante de la partida (INFORMACIÓN).

Las figura 5.3 y 5.4 son ventanas emergentes (PopUps) de información y advertencia, estas se deberán lanzar cuando el usuario realice una acción no permitida importante.

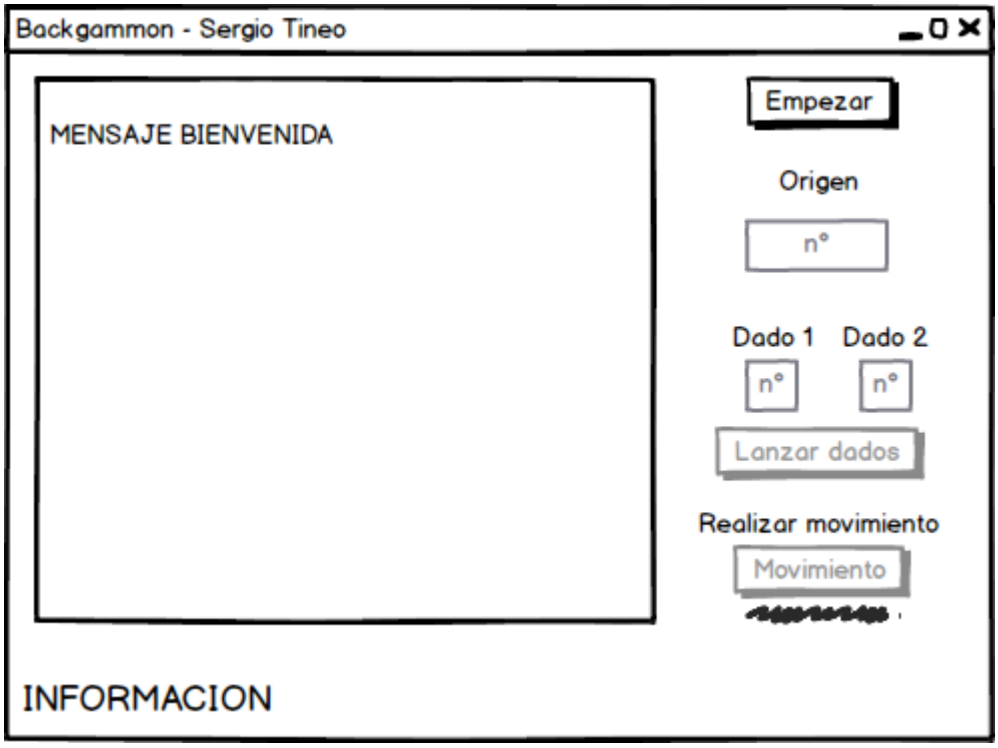

Figura 5.1: Ventana de inicio

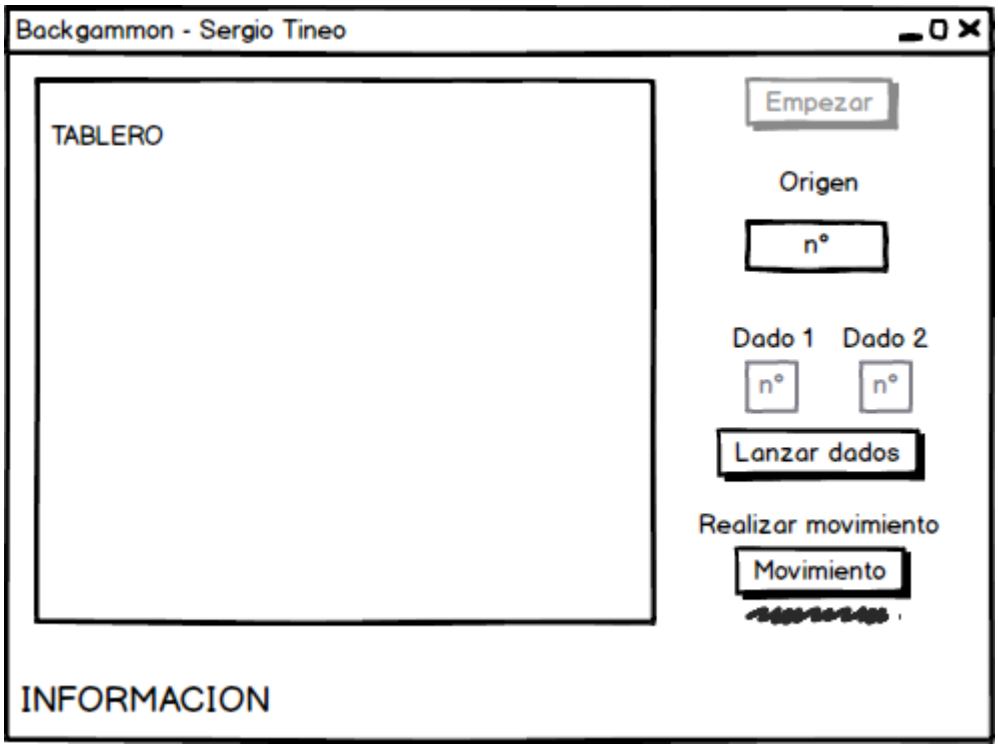

Figura 5.2: Ventana de juego

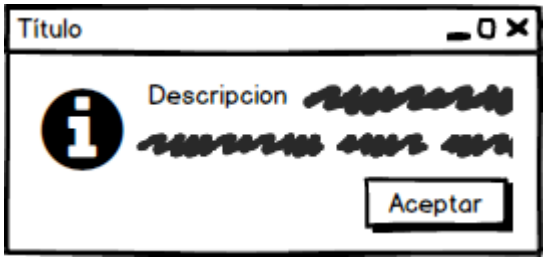

Figura 5.3: PopUp de información

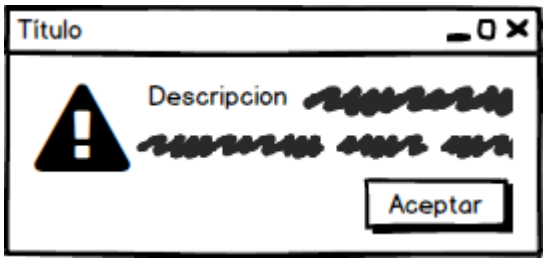

Figura 5.3: PopUp de advertencia

5.3 Diseño

En esta sección trataremos todo lo referido al diseño de la aplicación, desde casos de uso hasta el diagrama de clases inicial, modelo preliminar, para la aplicación.

5.3.1 Casos de uso

Solo tenemos un jugador con las características que se han detallado en el apartado de análisis, este jugador a grandes rasgos solo podrá hacer dos acciones.

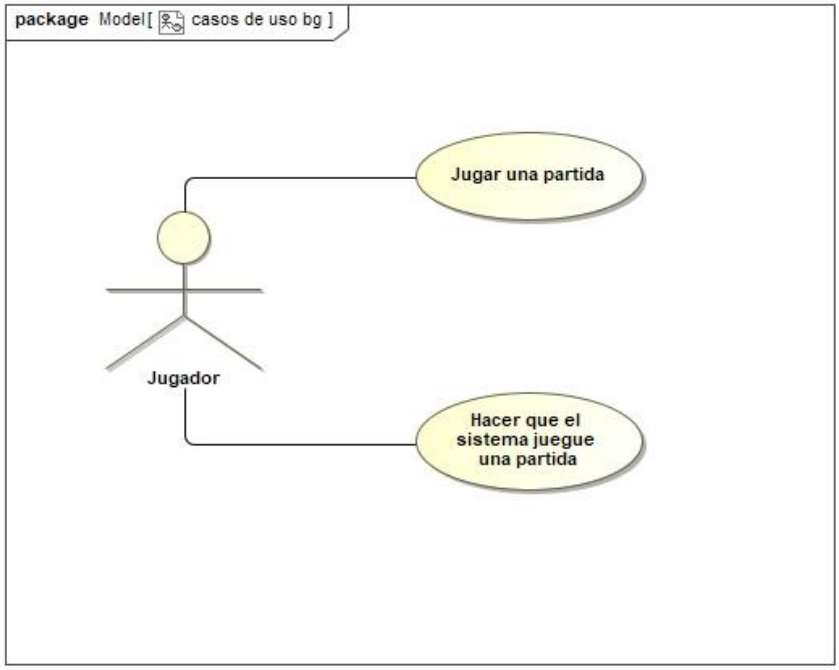

Figura 5.5: Casos de uso

En la figura 5.5 el jugador podrá realizar dos acciones determinadas en esta iteración del trabajo, o bien jugar al Backgammon o bien hacer que el sistema juegue contra si mismo y verlo o no. En ninguno de los dos casos existirá aun un aprendizaje por parte del sistema.

Entrando más en detalle sobre las acciones que implican jugar una partida, en la figura 5.6 se puede ver más específicamente. Este caso de uso parte de la acción anteriormente descrita como “Jugar una partida”.

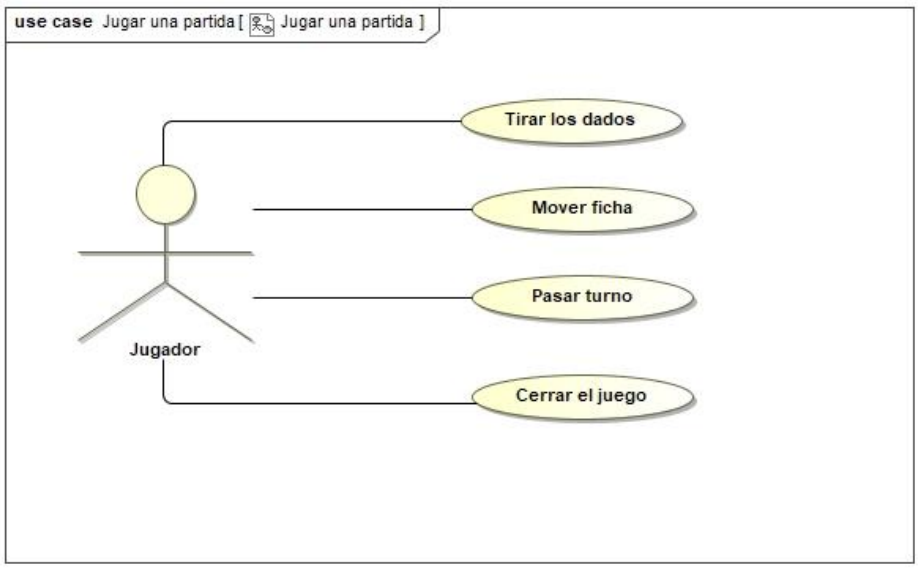

Figura 5.6: Casos de uso de jugar partida

El jugador puede tirar los dados, mover ficha, pasar turno o cerrar el juego. Estas acciones pueden ser tratadas de forma independiente ya que aunque para mover ficha es necesario haber tirado los dados, el jugador puede igualmente intentar mover y ya el juego decidirá el resultado de su acción.

El siguiente caso de uso pertenece a la acción descrita en la figura 5.5 "Hacer que el sistema juegue una partida", se puede ver en detalle en la figura 5.7.

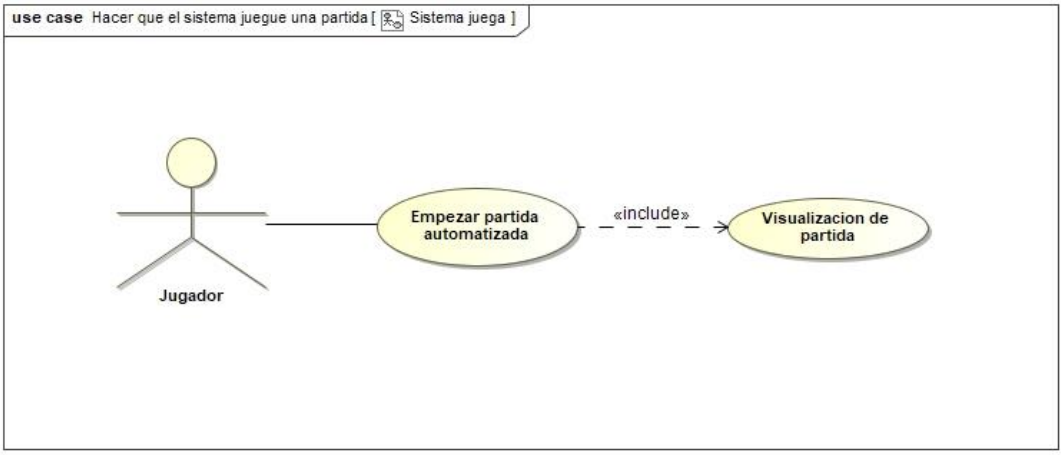

Figura 5.7: Caso de uso sistema juega

El jugador podrá empezar una partida automatizada, lo cual implicará ver o no ver la sucesión de jugadas, es decir, la partida.

Asociado al caso de uso de la figura 5.6 podemos ver en general el funcionamiento de cómo se juega una partida contra la máquina de forma simplificada tal como se muestra en la figura 5.8.

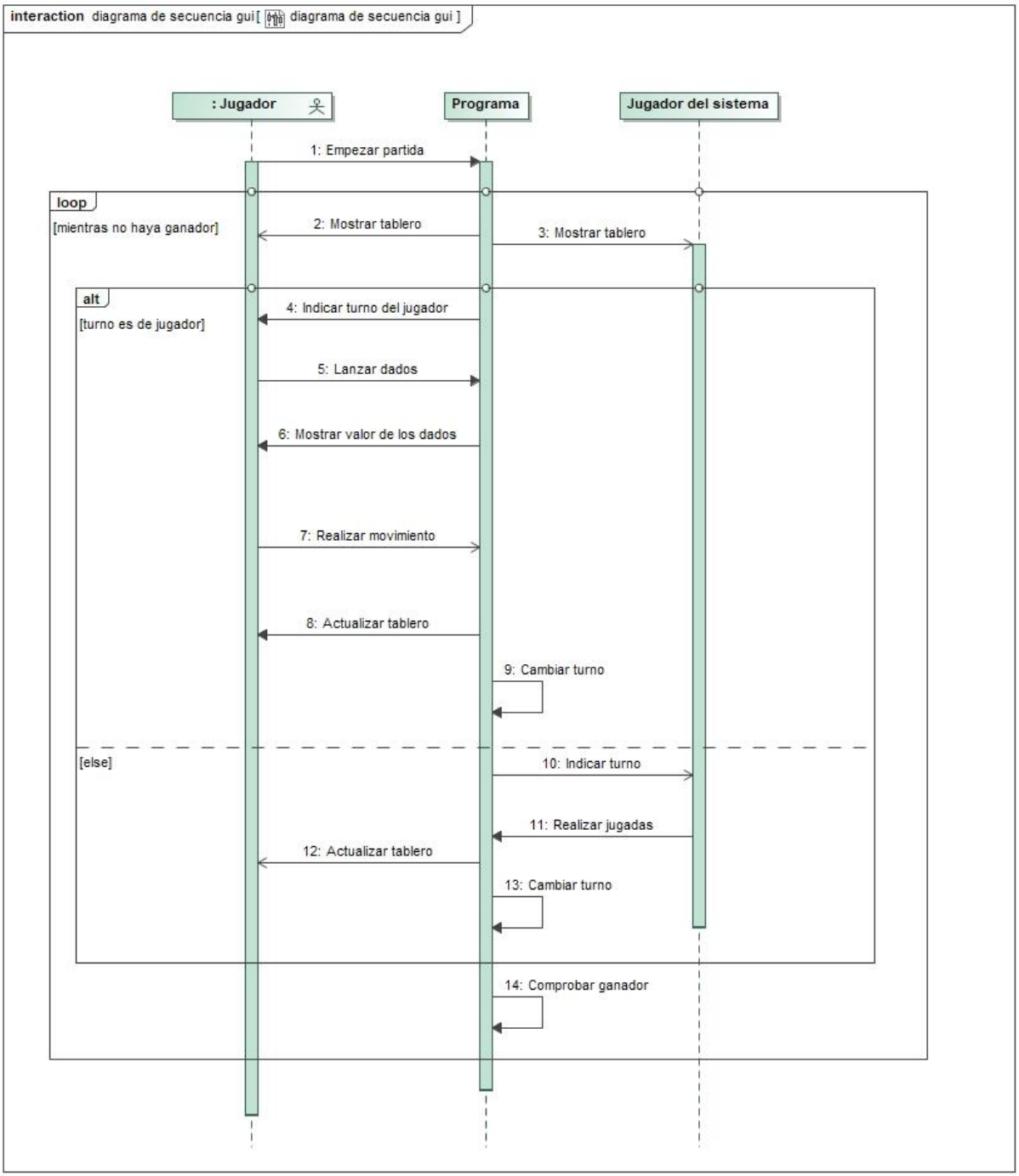

Figura 5.8: Diagrama de Backgammon con GUI

El jugador empezará una partida y mientras sea su turno tirará los dados y realizara los movimientos que quiera. Finalmente se cambiara el turno, y en el caso del jugador del sistema este simplemente realizará un movimiento y se mostrará el nuevo estado del tablero al jugador.

El diagrama de secuencia asociado al caso de uso de la figura 5.7 se puede ver en la figura 5.10.

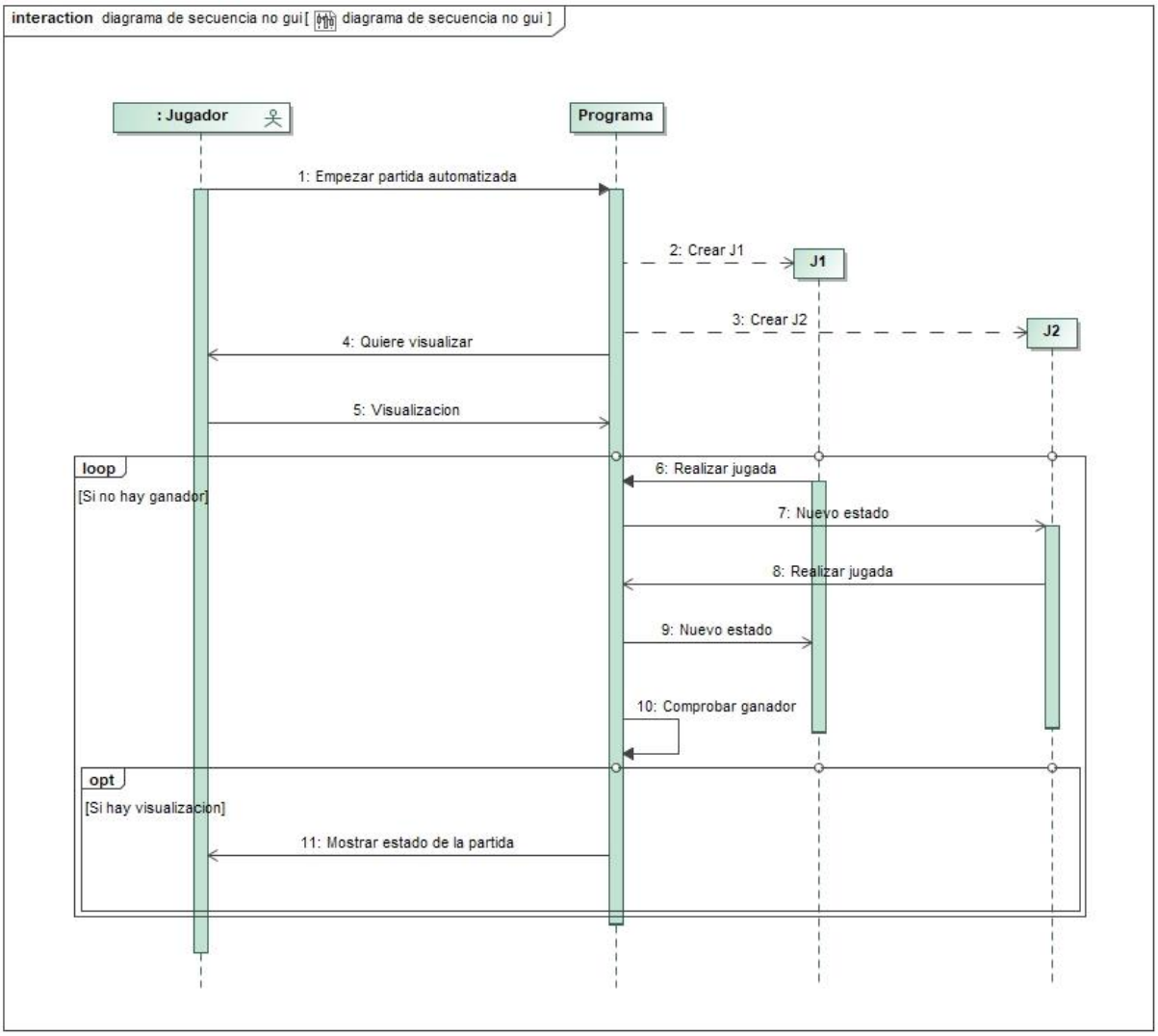

Figura 5.10: Diagrama de jugada automatizada

El jugador comenzará la partida automatizada, el programa creará un jugador para el turno 1 (J1) y otro para el 2 (J2) y comenzaran a jugar. Y si el jugador quería visualizar las jugadas se mostrarán.

5.3.2 Modelo

Es este apartado presentamos un modelo base para diseñar el juego, con el diagrama de clases de la figura 5.11 se cubren los aspectos más importantes del diseño del programa con las relaciones fundamentales y realizado de forma modularizada.

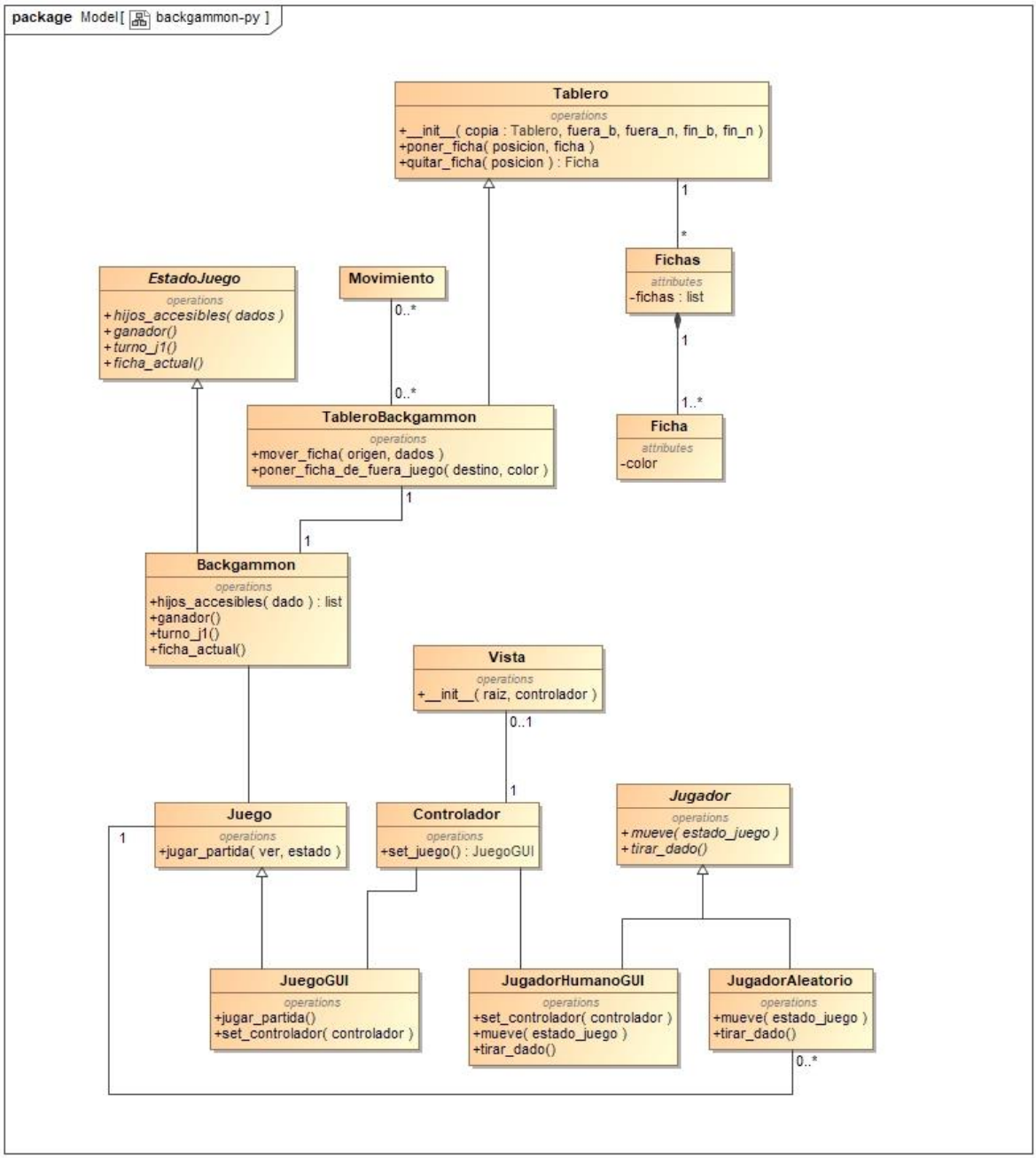

Figura 5.11: Diagrama de clases

Vamos a dar una vista preliminar del diseño del programa. La Ficha es el elemento básico, esta tiene un color y partir de el le correspondería un jugador, el primero blancas y el segundo negras, Fichas está compuesta obligatoriamente por

fichas, es decir, si Fichas no contiene alguna Ficha este no podría existir, de ahí la relación de composición estricta, por su parte Tablero es el encargado de almacenar las Fichas, las fichas fuera de juego y las que han acabado, así como poner y quitar Fichas. TableroBackgammon hereda de Tablero ya que añade la característica necesaria de movimiento de fichas siguiendo las reglas del juego. Backgammon es la descripción detallada del momento en el que se encuentra la partida, con el tablero, turno del jugador, y debe ser capaz de saber si hay ganador y de generar las posibles jugadas a partir de unos dados. Jugador es una clase abstracta de la cual hereda JugadorAleatorio que será el jugador contra el que se jugará y el que podremos ver jugar contra si mismo en la partida automatizada. Juego permite comenzar dado un estado del Backgammon inicial una partida y ver el resultado. Y ya para finalizar Vista, Controlador, JuegoGUI y JugadorHumanoGUI son los encargados de hacer que podamos jugar con interfaz gráfica de usuario.

5.3.3 Diagrama de despliegue

Será bastante sencillo desplegar, o iniciar, el programa ya que de entrada los requisitos son muy bajos. Solo es necesario un sistema operativo (OS), el intérprete de Python y el proyecto, en la figura 5.12 se puede observar el diagrama que lo representa.

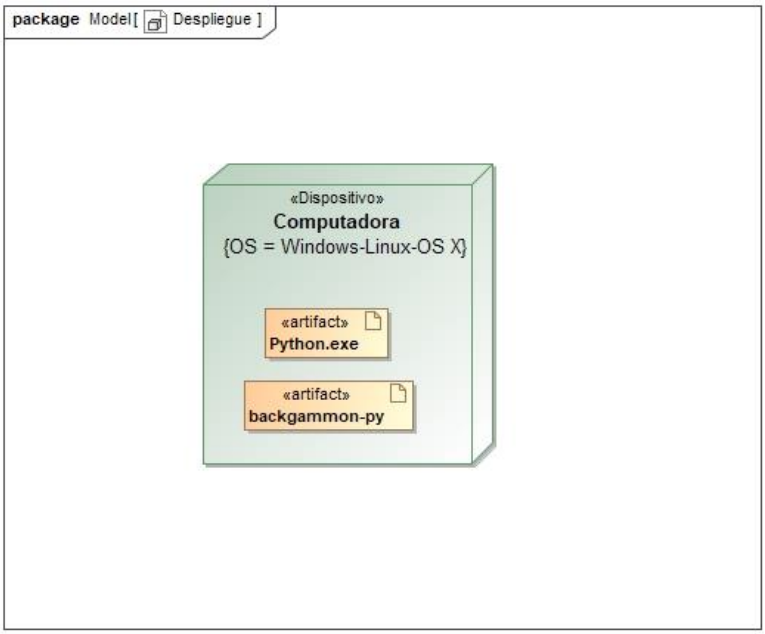

Figura 5.12: Componentes necesarios para usar el programa

6

Implementación Backgammon

En el presente capítulo se darán detalles sobre la implementación del juego Backgammon. Dividiremos el capítulo en función del paquete que estemos tratando, una sección por paquete, y la documentación presente en los ficheros del codigo se quitará en este documento. Para la presente implementación del programa se ha usado como base algunos recursos de la asignatura Inteligencia Artificial para Juegos [6].

6.1 Recursos

Primero de todo definamos este paquete que contiene funciones accesibles desde todo el contexto de la aplicación.

6.1.1 Dados

```
def generar_dados():
    tirada1 = randint(__MIN_VAL_DADO, __MAX_VAL_DADO)
    tirada2 = randint(__MIN_VAL_DADO, __MAX_VAL_DADO)
    if tirada1 == tirada2:
        return (tirada1, tirada2, tirada1, tirada2)
    else:
        return (tirada1, tirada2)
```

Figura 6.1: Función dado

La figura 6.1 muestra la función que devuelve una tupla de dos números. Los cuales están comprendidos entre dos aleatorios enteros especificados por las constantes de mínimo y máximo con valores de 1 y 6 respectivamente (valores del dado).

```
    def num_aleatorio(lim: int):
        return randint(0, lim)

    def num_aleatorio_min_max(min: int, max: int):
        return randint(min, max)

    def num_aleatorio_flotante():
        return random()
```

Figura 6.2: Funciones con retorno de números enteros

La primera función de la figura 6.2 retorna valores aleatorios entre 0 y el límite indicado como argumento de la función, ambos números incluidos. La segunda es parecida a la primera pero con un mínimo variable y la tercera simplemente retorna un valor aleatorio flotante menor a 1 y mayor o igual a 0. Estas funciones de generación de números aleatorios han sido puestas en este módulo para hacer de intermediario entre todas las clases y los recursos ofrecidos por Python.

6.1.2 Impresión

Este conjunto de funciones hacen de intermediario entre la impresión por defecto en consola de órdenes y los datos de entrada que deseen ser mostrados.

```
def print_dados(dados: tuple, msg: str = ""):
    res = "---> DADOS: "
    for i in dados:
        res += str(i) + " "
    res += msg
    print(res)
```

Figura 6.3: Función de visualización de dados

La figura 6.3 muestra una impresión perfectamente visible para el usuario de la tupla generada tras una tirada de dos dados y un mensaje.

```
def print_error(mensaje: str):
    print("--- {} ---".format(mensaje.upper()))
```

Figura 6.4: Función de visualización de errores

La figura 6.4 define la función para mostrar errores y que aparezcan más destacados.

```
def dar_bienvenida(mensaje: str):
    res = "-------------------------------------\n" + \
          "-------BIENVENIDO A BACKGAMMON-------\n" + \
          "-------------------------------------\n" + mensaje.upper()
    print(res)
```

Figura 6.5: Función de bienvenida

La función 6.5 sirve para dar la bienvenida y un mensaje de acompañamiento puesto en mayúsculas. Debo mencionar las funciones `print_(msg)`, `print_tablero(msg)`, `print_cod(msg)`, `print_analisis(msg)` que simplemente llaman a imprimir por consola. El hecho de haberlo dividido es para ver mejor el flujo del programa pudiendo mostrar u ocultar cuando lo necesite la impresión de ciertos datos.

6.2 Espacio de juego

En este paquete, figura 6.6, se define la base sobre la que se sustentará la implementación del Backgammon. Destacamos atributos y métodos más relevantes.

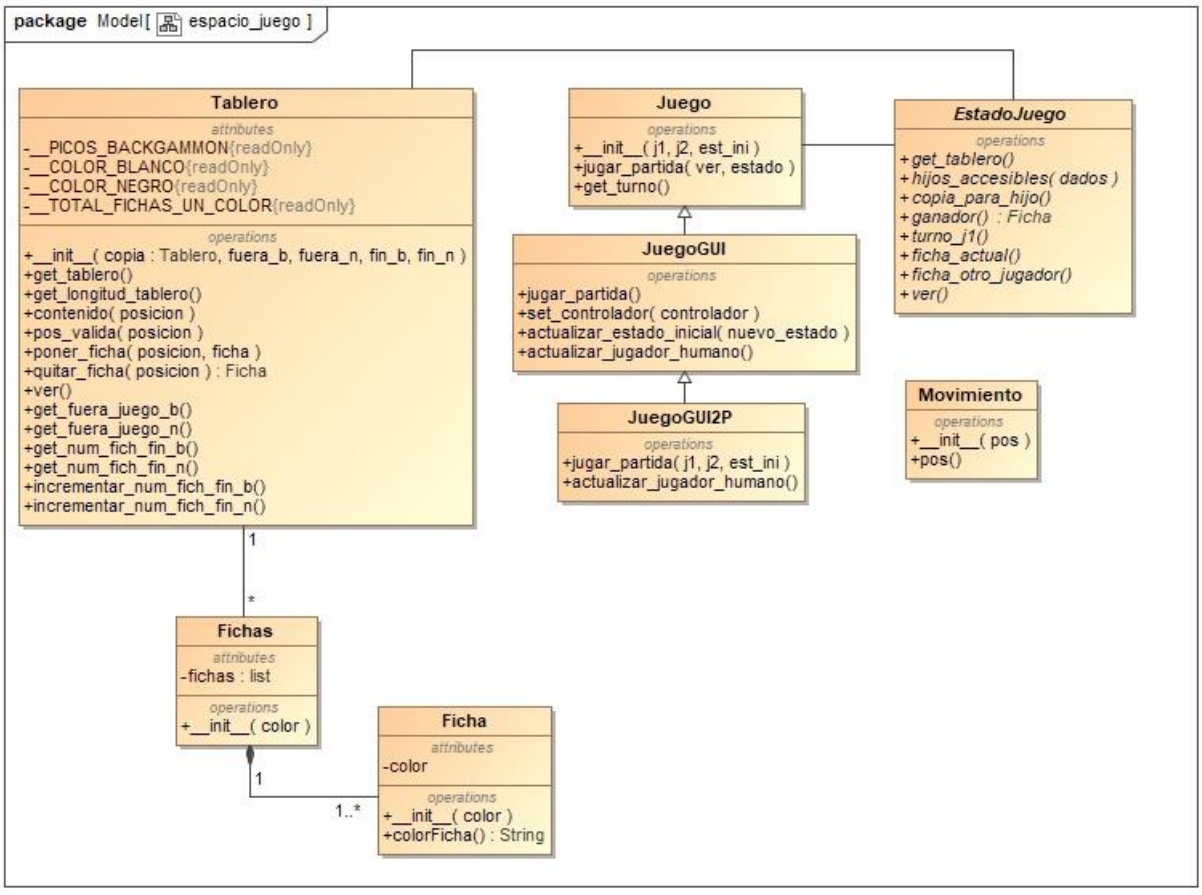

Figura 6.6: Diagrama de clases espacio_juego

Es necesario mencionar que solo aparecen las relaciones existentes entre los módulos de este paquete, no así todas las relaciones que existen entre los objetos del proyecto.

6.2.1 Ficha

El constructor, en la figura 6.7, de la ficha no permite asignar color si lo que se le pasa como argumento no es una cadena de caracteres de un solo elemento, en caso de hacerlo lanza un error. También con la función color ficha se recupera el color de la ficha.

```
def __init__(self, color):
    try:
        self.color = str(color)
        if len(color) > 1:
            print_("Pieza invalida")
            self.color = None
            del(self)
    except:
        print_error("ERROR: El argumento pasado no es un String")

def colorFicha(self):
    return self.color
```

Figura 6.7: Clase Ficha

De igual manera se han redefinido los métodos de comparación e impresión para poder comparar fichas en base al color en el caso del primero. Y ver el color de una ficha cuando se le solicite su impresión o conversión a cadena de caracteres en el caso del segundo.

6.2.2 Fichas

Como se observa en la figura 6.8, Fichas puede almacenar una o más fichas, para que esto pueda suceder fichas implementa una lista que es tratada como si fuera una pila. En esta pila siempre tiene que haber fichas, nunca puede estar vacía, si estuviera vacía la referencia de la pila sería eliminada. Por su parte también es posible obtener el color de la ficha y la cantidad de fichas qué contiene este objeto. De igual manera se puede incrementar la cantidad de fichas agregando una nueva ficha a la pila o decremento dar el número de fichas. Si se decrementa hay que vigilar que cuando no

haya ficha se elimine la referencia a la pila. En el caso de alimentar fichas además se devuelve la ficha que ha sido quitada de la pila ya que ésta será necesaria usarla probablemente en el tablero.

```
class Fichas():
    def __init__(self, ficha: Ficha):
        self.fichas = list()
        self.fichas.append(ficha)

    def get_ficha(self):
        return self.fichas[0] if self.fichas is not None else None

    def get_color(self):
        return self.fichas[0].colorFicha() if self.fichas is not None else ""

    def get_cantidad(self):
        return len(self.fichas) if self.fichas is not None else 0

    def incrementar_fichas(self, ficha: Ficha):
        self.fichas.append(ficha)

    def decrementar_fichas(self):
        if len(self.fichas) > 1:
            return self.fichas.pop()
        elif len(self.fichas) == 1:
            aux = self.fichas.pop()
            self.fichas = None
            return aux
        else:
            return None

    def __eq__(self, other):
        return isinstance(other, Fichas) and \
               self.get_ficha() == other.get_ficha() and \
               self.get_cantidad() == self.get_cantidad()
```

Figura 6.8: Clase Fichas

Es por ello que estamos ante una relación de agregación estricta ya que fichas no puede existir si no tiene ficha.

Es cierto que se podría haber implementado dos arrays uno para un tipo de fichas y otro para otro tipo de fichas sin embargo de esta manera creemos que se explota más la orientación a objetos.

6.2.3 Tablero

La clase Tablero es la base del juego ya que este contiene todas las fichas, y permite el movimiento.

```
class Tablero():
    __PICOS_BACKGAMMON: int = 24
    __COLOR_BLANCO = "B"
    __COLOR_NEGRO = "N"
    __TOTAL_FICHAS_UN_COLOR: int = 15

    def __init__(self, copia: list = None, fuera_b: list = None, fuera_n:
                 list = None, fin_b: int = 0, fin_n: int = 0):
        if copia is None:
            self.tablero = [None for i in range(0, self.__PICOS_BACKGAMMON)]
            self.inicializar()
            self.tab_fuera_juego_blancas = list()
            self.tab_fuera_juego_negras = list()
        else:
            try:
                self.tablero = deepcopy(copia)
                self.tab_fuera_juego_blancas = deepcopy(fuera_b)
                self.tab_fuera_juego_negras = deepcopy(fuera_n)
            except:
                print_error("ERROR - El tablero no fue copiado, algun argumento
                            invalido")

        self.fichas_blancas_acabado = fin_b
        self.fichas_negras_acabado = fin_n

    def get_tablero(self) -> list:
        return self.tablero

    def get_longitud_tablero(self) -> int:
        return len(self.tablero)

    def contenido(self, posicion: int) -> Fichas:
        return self.tablero[posicion]

    def pos_valida(self, posicion) -> bool:
        return 0 <= posicion < self.__PICOS_BACKGAMMON

    def poner_ficha(self, posicion: int, ficha: Ficha):
        if self.tablero[posicion] is None:
            self.tablero[posicion] = Fichas(ficha)
        elif self.tablero[posicion].get_color() == ficha.colorFicha():
            self.tablero[posicion].incrementar_fichas(ficha)

    def quitar_ficha(self, posicion: int) -> Ficha:
        if self.tablero[posicion] is None:
            return None
        else:
            aux = self.tablero[posicion].decrementar_fichas()

        if self.tablero[posicion].get_cantidad() == 0:
            self.tablero[posicion] = None

        return aux

    def ver(self):
        print_tablero(self)
```

Figura 6.9: Case Tablero métodos importantes

En la figura 6.9 se aprecian los métodos más importantes de la clase tablero, este contiene información de las fichas, por una parte las fichas que están jugando, es decir, las fichas que se encuentran actualmente disponibles para que los jugadores las puedan mover, también almacena las fichas que están fuera de juego de ambos jugadores y por último el número total de fichas que han finalizado de cada jugador.

Su constructor si no recibe ningun argumento crea la lista que representa el tablero, las dos listas de fichas fuera de juego vacías. Si recibe argumentos, copia el tablero que se le pase, copia las dos listas con las fichas fuera de juego. Sea como fuere asigna los valores a las variables encargadas de contar las fichas que han finalizado. Debemos hacer especial hincapié en la necesidad de usar deepcopy [19] en vez de copy, ya que al trabajar con listas de objetos es necesario una copia profunda y no una copia superficial para evitar las referencias a objetos, necesitamos copias.

Es necesario destacar los métodos encargados de poner ficha y quitar ficha. Cuando se pone ficha es necesario vigilar que o bien no haya ninguna ficha en la casilla o bien las fichas que hay son del color del jugador en otro caso no puede producirse movimiento es decir no puede ponerse ninguna ficha, por otra parte si tenemos que quitar ficha ésta se encarga de vigilar que no se quite ficha si no existe ya que podría dar fallo y en el caso de que haya ficha poder quitarla. La comprobación del color de esta ficha se encargará la clase que veremos más adelante que hereda de tablero. De igual modo cuando no haya ficha en una posición nada será necesario eliminar la referencia en el array de la ficha en cuestión a quitar. Si ha quitado las ficha satisfactoriamente la devuelve.

Aunque no aparecen en la figura anterior por su extrema sencillez existen métodos que recogen y devuelven el número de fichas que han finalizado para un jugador determinado. Que devuelven una lista de fichas que están fuera de juego para un jugador determinado. Y que incrementan el número de fichas que han finalizado.

También se ha redefinido la implementación de la conversión a cadena de caracteres. Para en cada momento poder ver adecuadamente la situación dada del tablero, en principio solos las fichas dentro el tablero.

Cómo se ha implementado esta redefinición se puede ver en la figura 6.10

```
def __str__(self):
        cad = ""
        iniB = "Inicio {}".format(self.__COLOR_BLANCO)
        iniN = "Inicio {}".format(self.__COLOR_NEGRO)
        mitad = int(self.__PICOS_BACKGAMMON / 2)
        parte_superior = self.tablero[0:mitad]
        parte_superior.reverse()
        parte_inferior = self.tablero[mitad:self.__PICOS_BACKGAMMON]

        cad += self._str_guiones(mitad * 3 - len(iniB) - 1) + \
               iniB + "\n"
        cad += self._str_linea_numeros(mitad, 0) + "\n"

        for fs in parte_superior:
            if fs is None:
                cad += "-- "
            else:
                cad += str(fs) + " "

        cad += "\n\n"

        for fs in parte_inferior:
            if fs is None:
                cad += "-- "
            else:
                cad += str(fs) + " "

        cad += "\n" + \
               self._str_linea_numeros(mitad + 1,
                                       self.__PICOS_BACKGAMMON + 1) + "\n"
        cad += self._str_guiones(mitad * 3 -
                                 len(iniN) - 1) + iniN + "\n"
        return cad
```

Figura 6.10: Redefinición de método __str__

La representación cuando se ve, es muy intuitiva, aunque algo compleja a la hora de realizar ya que hay que invertir parte de la lista, la mitad, y la otra lista así como calcular la posición la cantidad y el color de las fichas del tablero para esto hicimos unos métodos privados sólo accesibles desde esta clase que simplemente general lo que nosotros necesitamos de cadena de caracteres.

Siguiendo la numeración propuesta durante el capítulo dos de Backgammon, la casilla número 1 está arriba a la derecha y la casilla número 24 está abajo a la derecha siendo la casilla número 1 el comienzo de las fichas blancas en la casilla número 24 el comienzo de las fichas negras, como se puede ver en la figura 6.11.

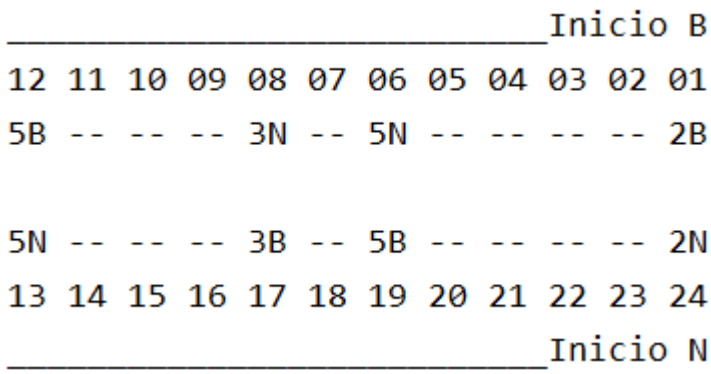

Figura 6.11: Visualización del tablero

Esta es la representación del tablero, donde xB o xN son Fichas, siendo "x" el número de ficha y B o N el color. Los números bajo "..._Inicio B" y arriba de "..._Inicio N" que empiezan en 01 y acaban en 24 son las posiciones, los "picos" del tablero de juego, y "- -" representa la ausencia de Fichas.

6.2.4 Movimiento

Esta clase se usa cuando ha sucedido algun movimiento, igualmente también se usa cuando el movimiento no ha sido válido y en este caso es None.

Se decidió usar Movimiento como un objeto que realmente es volátil ya que prácticamente en cada interacción se crea y se destruye para poder revisar cuál fue el último con movimiento punto y aparte dada su extrema sencillez y con lo mencionado a cubierto todos los aspectos que cubre este objeto.

6.2.5 EstadoJuego

Es una clase abstracta que conecta diferentes partes del juego y que la clase que lo implemente deberá describir las características de los métodos. De tal manera que sirva como molde y no falte ningún detalle necesario para que el juego pueda progresar,

es decir, que el juego pueda empezar, se pueda jugar en él y pueda acabar. Es por todo ello qué tiene métodos que deberán implementarse y poder usarse para obtener el tablero, ver cuáles son los hijos accesibles, poder hacer una copia de un hijo, saber quién es el ganador y devolver su ficha, saber cuál es el turno del jugador uno, saber cuál es la ficha del jugador actual y saber cuál es la ficha del otro jugador. Además de poder ver el estado del juego.

6.2.6 Juego

Es la clase base sobre la que se puede jugar una partida esta recibe al constructor dos jugadores y el estado inicial del juego. Ya que el estado inicial es necesario para saber a quién le toca jugar y puedan jugar sobre un tablero.

```
class Juego():
    def __init__(self, j1: Jugador, j2: Jugador, est_ini: EstadoJuego):
        self.j1 = j1
        self.j2 = j2
        self.est_ini = est_ini

    def jugar_partida(self, ver: bool, estado: EstadoJuego = None):
        res = 2

        if estado is None:
            return self.jugar_partida(ver, self.est_ini)
        else:
            if ver:
                print_tablero(str(estado))

            if estado.ganador() is not None:
                res = 1 if estado.ganador() == estado.FICHA_J1 else -1
            else:
                if estado.turno_j1():
                    res = self.jugar_partida(ver, self.j1.mueve(estado))
                else:
                    res = self.jugar_partida(ver, self.j2.mueve(estado))

        return res

    def get_turno(self):
        return self.est_ini.turno_j1()
```

Figura 6.12: Clase Juego

En la figura 6.12 se puede observar la implementación de la clase Juego, el objeto que permite a dos jugadores jugar una partida.

El método que sirve para jugar partida necesita de entrada un valor booleano para saber si el usuario desea ver la representación del tablero conforme los jugadores juegan y Por otra parte el estado del juego en ese momento. Si no hay ningún estado de juego entonces comienza la partida con el estado inicial. Si ya existía un estado eso significa qué es una partida intermedia o final, en este caso 1º si es necesario imprimr por consola el estado y después comprueba si hay ganador. Si hay ganador pone la variable de retorno con el valor que corresponda, en otro caso de forma recursiva se llama a sí mismo haciendo qué uno de los jugadores mueva dependiendo de si es su turno o es del otro.

Adelantándonos un poco, la llamada del jugador J1 o J2 a mueve a partir del estado, modifica ese estado tras la jugada del jugador de forma que al hacerlo de recursivamente van moviendo cada uno de los dos jugadores, hasta que uno de los dos gana. Ese valor se retorna, y vuelve hacia atrás, hasta llegar a esta llamada inicial retornando **1** si gano el primer jugador o **-1** si gano el 2º jugador.

6.2.7 JuegoGUI

Esta clase hereda de Juegos y reimplementa sus métodos para ser capaz de poder jugar con una interfaz gráfica de un jugador contra la máquina.

Dicho de otro modo hace de enlace entre una persona que sea jugador y otro jugador que sea del sistema como se puede ver en la figura 6.13.

En este caso al jugar partida a diferencia del otro aquí se revisa si el jugador que está actualmente jugando es el humano o no para en función de eso que mueva el jugador del sistema, actualice el tablero y muestre un mensaje de que el jugador enemigo realiza un movimiento. Aquí ya se empieza a entrever parte del modelo del patrón de diseño MVC como ya que éste sería la parte de modelo para la interconexión entre el controlador y los jugadores.

```
class JuegoGUI(Juego):
    def __init__(self, j1: Jugador, j2: Jugador, est_ini: EstadoJuego):
        super().__init__(j1, j2, est_ini)

    def jugar_partida(self, ver: bool = False, estado: EstadoJuego = None):
        if estado is not None:
            self.est_ini = estado

        self.actualizar_jugador_humano()

        if not isinstance(self.j1, JugadorHumanoGUI) and
                        self.est_ini.turno_j1():
            self.actualizar_estado_inicial(self.j1.mueve(self.est_ini))
            self.controlador.actualizar_tablero(str(self.est_ini))
            self.controlador.set_info("El jugador enemigo realizo
                                       movimiento")
        elif not isinstance(self.j2, JugadorHumanoGUI) and not
                        self.est_ini.turno_j1():
            self.actualizar_estado_inicial(self.j2.mueve(self.est_ini))
            self.controlador.actualizar_tablero(str(self.est_ini))
            self.controlador.set_info("El jugador enemigo realizo
                                       movimiento")
        self.actualizar_jugador_humano()

    def set_controlador(self, controlador: ctrl_vista_basica):
        self.controlador = controlador

    def actualizar_estado_inicial(self, nuevo_estado: EstadoJuego):
        self.est_ini = nuevo_estado

    def actualizar_jugador_humano(self):
        if isinstance(self.j1, JugadorHumanoGUI) and self.est_ini.turno_j1():
            self.j1.set_estado(self.est_ini)
            hay_fuera_juego = self.j1.hay_fichas_fuera_de_juego_aun()
            self.controlador.set_info("Hay fichas fuera de juego que debes
                                poner antes") if hay_fuera_juego else None
        elif isinstance(self.j2, JugadorHumanoGUI) and not
                                                self.est_ini.turno_j1():
            self.j2.set_estado(self.est_ini)
            hay_fuera_juego = self.j2.hay_fichas_fuera_de_juego_aun()
            self.controlador.set_info("Hay fichas fuera de juego que debes
                                poner antes") if hay_fuera_juego else None
        if self.est_ini.ganador():
            self.controlador.finalizar_juego(\
                                self.est_ini.ganador().colorFicha())
```

Figura 6.13: Clase JuegoGUI

Además incorpora algunas funcionalidades nuevas, como se puede ver en la figura 6.13, la primera y principal es poder conectar el Controlador al Juego, y la segunda es poder revisar el estado del juego En busca de fichas que el enemigo haya comida mostrar el mensaje y de igual modo comprobar si hubo ganador.

6.2.8 JuegoGUI2P

Esta clase hereda de JuegoGUI reimplementando el método de jugar partida y de actualizar el jugador humano.

En este caso no hace falta hacer distinción entre un jugador y otro jugador ya que ambos son exactamente iguales, es decir, son jugadores humanos. Es por ello que se han tenido que realizar modificaciones en los métodos citados Y que se ven en detalle en la figura 6.15.

```
class JuegoGUI2P(JuegoGUI):
    def __init__(self, j1: Jugador, j2: Jugador, est_ini: EstadoJuego):
        super().__init__(j1, j2, est_ini)

    def jugar_partida(self, ver: bool = False, estado: EstadoJuego = None):
        if estado is not None:
            self.est_ini = estado

        self.actualizar_jugador_humano()

        self.controlador.actualizar_tablero(str(self.est_ini))

    def actualizar_jugador_humano(self):
        if self.est_ini.turno_j1():
            self.j1.set_estado(self.est_ini)
            hay_fuera_juego = self.j1.hay_fichas_fuera_de_juego_aun()
            self.controlador.set_info("Hay fichas fuera de juego que debes
                                poner antes") if hay_fuera_juego else None
        elif not self.est_ini.turno_j1():
            self.j2.set_estado(self.est_ini)
            hay_fuera_juego = self.j2.hay_fichas_fuera_de_juego_aun()
            self.controlador.set_info("Hay fichas fuera de juego que debes
                                poner antes") if hay_fuera_juego else None
       if self.est_ini.ganador():
                  self.controlador.finalizar_juego(
                                self.est_ini.ganador().colorFicha())
```

Figura 6.14: Clase JuegoGUI2P

Como se ve en la figura 6.14 para jugar la partida debemos de actualizar el jugador humano, que se actualizará en función del turno, y actualizar el tablero.

Para actualizar el jugador humano, probamos 1º el turno del jugador que corresponde, actualizamos su estado, y comprobamos si hay fichas fuera de juego. Mandándole un mensaje de que hay si las hubiera.

6.3 Backgammon

En este paquete se implementa todo lo referido al Backgammon, hasta ahora lo que tenemos es un tablero sobre el que podemos poner fichas, dicho tablero puede almacenar fichas que estén fuera de juegos y puede llevar la contabilización de las fichas que han salido del juegos pero no establece las normas para el movimiento de fichas.

En la figura 6.15 se puede observar el diagrama de clases de este paquete, nuevamente omitiendo las relaciones existentes con elementos fuera del paquete.

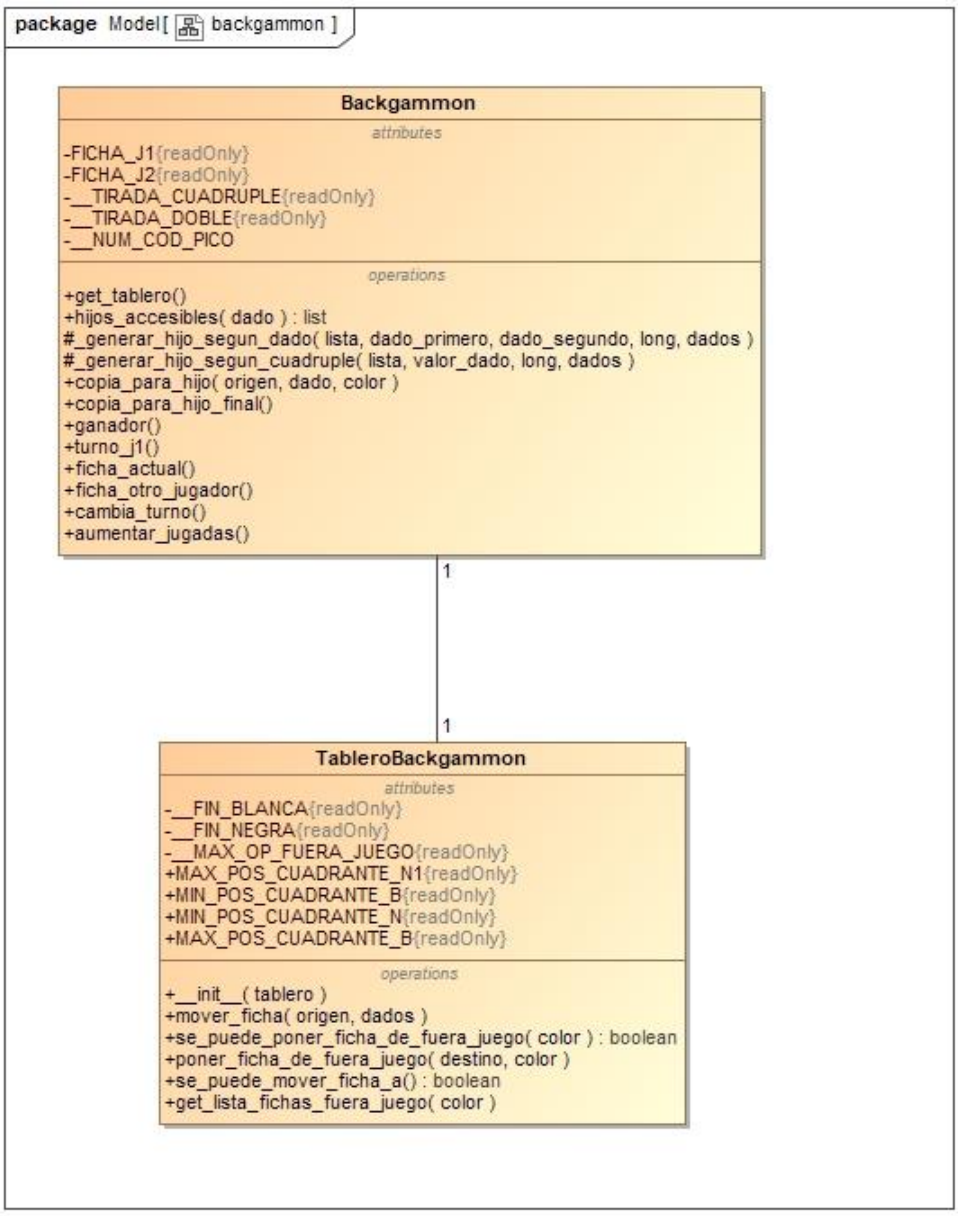

Figura 6.15: Diagrama de clases Backgammon

6.3.1 TableroBackgammon

Esta clase establece todas las normas de movimiento del Backgammon. Gracias a ella se pueden mover fichas, se puede preguntar si es posible mover una ficha de un determinado color, se puede poner una ficha de fuera de juego y preguntar si es posible poner una ficha de fuera de juego.

De igual manera se establecen constantes de posición del cuadrante inicial del jugador blanco y del jugador negro, así como el número máximo de fichas que se pueden capturar y las posiciones correspondientes al fin del tablero.

```
class TableroBackgammon(Tablero):
    __FIN_BLANCA = 24
    __FIN_NEGRA = -1
    __MAX_OP_FUERA_JUEGO = 1
    MIN_POS_CUADRANTE_B = 0
    MAX_POS_CUADRANTE_B = 5
    MIN_POS_CUADRANTE_N = 18
    MAX_POS_CUADRANTE_N = 23

    def __init__(self, tab: Tablero = None):
        if tab is not None:
            super().__init__(tab.get_tablero(), tab.get_fuera_juego_b(),
                             tab.get_fuera_juego_n(),
                             tab.get_num_fich_fin_b(),
                             tab.get_num_fich_fin_n())
        else:
            super().__init__()
```

Figura 6.16: Clase TableroBackgammon (1)

Como se puede apreciar en la figura 6.16, esta clase TableroBackgammon llama a la clase padre, el Tablero, para crearse e inicialmente se crean unas constantes.

En la figura 6.17 se puede observar la jerarquía de reglas para el movimiento de una ficha del Backgammon. Más detalladamente hace lo siguiente:

- En el caso que el movimiento resulte llegar a -1 una ficha negra o a 24 una ficha blanca se elimina del tablero (acerca la victoria) siempre que todas sus fichas están en la "meta", es decir, en el cuadrante superior (negras) o inferior (blancas) izquierdo.

- Si en la casilla de destino hay una sola ficha del color contrario esta pasa a estar fuera de juego hasta que el siguiente jugador la ponga en su inicio.
- Si hay dos fichas del color contrario no es posible mover a esa posicion la ficha.
- Si en la casilla destino no hay ninguna ficha o las que hay son de su color se mueve sin problemas.

```
def mover_ficha(self, origen, dado, color):
  try:
    if self.pos_valida(origen) \
            and self.tablero[origen] is not None \
            and self.tablero[origen].get_color() == color:

        if color == self.get_color_blanco():
            destino = origen + dado
            if self.pos_valida(destino):
                if self.tablero[destino] is None or
                                  self.tablero[destino].get_color() == color:
                    self.poner_ficha(destino, self.quitar_ficha(origen))
                    return Movimiento(destino)

                elif self.tablero[destino].get_cantidad() ==
                                                    self.__MAX_OP_FUERA_JUEGO:
                    self.tab_fuera_juego_negras.append(
                                                    self.quitar_ficha(destino))
                    self.poner_ficha(destino, self.quitar_ficha(origen))
                    return Movimiento(destino)
                else:
                    return None
            elif destino == self.__FIN_BLANCA and
                        self._comprobar_fichas_en_meta(color):
                self.quitar_ficha(origen)
                self._ficha_ha_acabado(color)
                return Movimiento(destino)

            else:
                return None
```

…

IGUAL PARA LAS OTRAS FICHAS

…

```
        else:
            raise ValueError("Color de pieza invalido")
    else:
        return None

except:
    print_error("ERROR GRAVE: Durante la evaluacion del tablero "
                "para mover ficha ha sucedido un fallo")
```

Figura 6.17: Clase TableroBackgammon (2)

Además si en algún momento mientras evalúa el tablero porque se desea mover y se produce alguna excepción, o incluso que el color de la ficha no sea válido, eleva o imprime dicho error.

Aquí cuando se ha producido un movimiento válido el valor de retorno es un Movimiento, el cual almacena el destino. Y en otro caso es que no fue posible.

```
def se_puede_poner_ficha_de_fuera_juego(self, color):
    if color == self.get_color_blanco() and self.tab_fuera_juego_blancas:
        for i in range(self.MIN_POS_CUADRANTE_B, self.MAX_POS_CUADRANTE_B):
            if self.contenido(i) is None or self.contenido(i).get_color() ==
                 color or self.contenido(i).get_cantidad() <=
                                            self.__MAX_OP_FUERA_JUEGO:
                return True
    elif color == self.get_color_negro() and self.tab_fuera_juego_negras:
        for i in range(self.MIN_POS_CUADRANTE_N, self.MAX_POS_CUADRANTE_N):
            if self.contenido(i) is None or self.contenido(i).get_color() ==
                 color or self.contenido(i).get_cantidad() <=
                                            self.__MAX_OP_FUERA_JUEGO:
                return True
    else:
        return False
```

Figura 6.18: Clase TableroBackgammon (3)

En la figura 6.18 se muestra el método que comprueba si es posible poner fichas de fuera de juego en el tablero. Retorna True si al menos se puede poner una (o más) y False en otro caso. Para ello se comprueba el color de la ficha para después verificar que haya fichas de ese color fuera del tablero, y recorremos el tablero entre los valores de su cuadrante de inicio para ver si podemos poner una ficha.

Si se desea poner una ficha en el tablero existe un método, `poner_ficha_de_fuera_juego(destino, color)`, que comprueba lo mismo de la figura 6.19 pero en vez de devolver verdadero o falso pone la ficha y retorna el movimiento.

Da igual modo en el Tablero mostrábamos las fichas que habían en el tablero redefiniendo el método para mostrar un objeto como cadena de caracteres. En TableroBackgammon hacemos lo mismo agregando las listas de fichas que están fuera de juego o han finalizado.

También hay un método que retorna un valor de verdadero si hay una ficha que puede poner desde una posición de origen hasta una posición final calculada a partir del lado ésta se puede apreciar en la figura 6.19.

```
def se_puede_mover_ficha_a(self, origen, dado, color) -> bool:
    try:
        if self.pos_valida(origen) \
                and self.tablero[origen] is not None \
                and self.tablero[origen].get_color() == color:
            if color == self.get_color_blanco():
                destino = origen + dado
                if self.pos_valida(destino):
                    if self.tablero[destino] is None or
                         self.tablero[destino].get_color() == color:
                        return True
                    elif self.tablero[destino].get_cantidad() ==
                         self.__MAX_OP_FUERA_JUEGO:
                        return True
                    else:
                        return False
                elif destino == self.__FIN_BLANCA and
                         self._comprobar_fichas_en_meta(color):
                    return True
                else:
                    return False
...
CONTINUA IGUAL PARA LAS OTRAS FICHAS
...
            else:
                raise ValueError("Color de pieza invalido")
        else:
            return None
    except:
        print_error("ERROR GRAVE: Durante la evaluacion del tablero "
                    "para comprobar si es posible mover ficha ha sucedido un
                     fallo")
```

Figura 6.19: Clase TableroBackgammon (4)

Este método primero comprueba el color de la ficha para saber si el movimiento hacia delante o hacia detrás. Tras esto calcula cuál sería su posible destino y retorna verdadero o falso en función de si pudiera moverse. Nuevamente también se eleva una excepción en el caso de color de ficha no válido. Y si no es posible moverla se retorna un valor de None qué significaría que no se ha podido averiguar si el movimiento era posible, esto en la práctica esto no puede pasar.

Por último en la figura 6.20 Puede observarse un método simple para conseguir retornar las fichas de fuera de juego de un jugador de un determinado color.

```
def get_lista_fichas_fuera_juego(self, color):
    return self.get_fuera_juego_b() if color == self.get_color_blanco() else
                                                    self.get_fuera_juego_n()
```

Figura 6.20: Clase TableroBackgammon (5)

6.3.2 Backgammon

Esta clase, que también da nombre al paquete, es de las más importantes y la más compleja de esta primera iteración ya que en ella se deben mantener todos los datos en cada momento del juego. Como puedan ser el tablero con todas las fichas, las que hayan finalizado y las que estén fuera de juego. Así como guardar el último movimiento, el turno actual, algo fundamental, y el número de jugadas.

```
class Backgammon(EstadoJuego):
    FICHA_J1 = None
    FICHA_J2 = None
    __TIRADA_DOBLE = 2
    __TIRADA_CUADRUPLE = 4
    __NUM_COD_PICO = 6

    def __init__(self, turno=True, tablero: TableroBackgammon = None,
                         movimiento_ultimo: Movimiento = None, jugadas=0):
        self.turno = turno
        if tablero is None:
            self.tablero = TableroBackgammon()
        else:
            self.tablero = TableroBackgammon(tablero)
        self.FICHA_J1 = Ficha(self.tablero.get_color_blanco())
        self.FICHA_J2 = Ficha(self.tablero.get_color_negro())
        self.movimiento_ultimo = movimiento_ultimo
        self.jugadas = jugadas
```

Figura 6.21: Clase Backgammon (1)

Cuando se crea un objeto Backgammon, figura 6.21, por defecto el turno es verdadero, es decir, es el turno del primer jugador. Si el tablero es vacío entonces lo crea, en otro caso es que le han dado el tablero de entrada y realiza una copia gracias a la llamada a TableroBackgammon. Sea como fuere crea una ficha de jugador uno y dos (J1 y J2) de referencia, asigna cuál es el movimiento y el numero de jugadas, es decir, cuantos movimientos (dobles o cuádruples) ha habido. Es por ello que almacena

los valores constantes de cual es la tirada doble y la tirada cuádruple. De igual forma tiene en una constante cual es el número de picos de un cuadrante.

```
def hijos_accesibles(self, dados) -> list:
    lista = list()
    long = self.tablero.get_longitud_tablero()
    if len(dados) == self.__TIRADA_DOBLE:
        self._generar_hijo_segun_dado(lista, 0, 1, long, dados)
        if not lista:
            self._generar_hijo_segun_dado(lista, 1, 0, long, dados)
        if not lista:
            pos_dado_mayor = 0 if dados[0] > dados[1] else 1

            self._generar_hijo_segun_dado(lista, pos_dado_mayor, -1, long,
                                                                    dados)

            if not lista:
                pos_dado_menor = 1 if pos_dado_mayor == 0 else 0

                self._generar_hijo_segun_dado(lista, pos_dado_menor, -1,
                                                          long, dados)
...

CONTINUA LA TIRADA CUADRUPLE

...

def _generar_hijo_segun_dado(self, lista: list, dado_primero: int,
dado_segundo: int, long: int, dados: tuple):
    if dado_segundo >= 0:
        for i in range(0, long):
            if self._pos_licita(i, dados[dado_primero]):
                aux = self.copia_para_hijo(i, dados[dado_primero],
                                           self.ficha_actual().colorFicha())
                for j in range(0, long):
                    if self._pos_licita(j, dados[dado_segundo]):
                        aux2 = aux.copia_para_hijo_final(j,
                                 dados[dado_segundo],
                                 self.ficha_actual().colorFicha())
                        if not self._hijo_existente(lista, aux2):
                            lista.append(aux2)
    elif dado_segundo == -1:
        # Se hace una jugada
        for i in range(0, long):
            if self._pos_licita(i, dados[dado_primero]):
                aux = self.copia_para_hijo_final(i, dados[dado_primero],
                                           self.ficha_actual().colorFicha())
                if not self._hijo_existente(lista, aux):
                    lista.append(aux)
```

Figura 6.22: Clase Backgammon (2)

En la figura 6.22 el primer método, el de hijos accesibles, retorna una lista de estados que sean accesibles desde el estado actual a partir del valor de los dados que es una tupla. Esto quiere decir que a partir de un turno de un jugador, con un tipo de

ficha, en un momento dado en el tablero, hijos accesibles retornaría todos los posibles movimientos que podría hacer para esa situación dada.

Para hacer esto deben cumplir las normas, así que primero prueba todas las combinaciones posibles con donde primero mueve una ficha con el primer dado y después mueve otra con el segundo, que pueden ser las mismas o no. Si no hay ningún tipo de movimiento posible entonces prueba a mover con el segundo dado y después con el primero. Si sigue sin haber posibles movimientos, entonces prueba con el dado mayor, y si sigue sin haber posible jugada prueba con el lado menor. De esta manera nos aseguramos que se cumplan las normas para el jugador del sistema.

Por su parte el segundo método que es privado funciona de la como se describe a continuación. Se le proporciona una lista de entrada, él dado que debe jugar 1º y 2º, y la tupla de los dados. Adicionalmente el parámetro del dado segundo (dado que debe jugar en segundo lugar) se usa para indicar si la tirada es doble o cuádruple. Si es cuádruple vale -1 ya que el valor de todos los dados en las jugadas cuádruples es el mismo.

Aunque funciona de forma similar, vemos necesario describir el funcionamiento del método `def _generar_hijo_segun_dado_cuadruple(self, lista: list, valor_dado, long: int, num_jugadas=None)`. A este método se le proporciona una lista de entrada, el valor del dado, la longitud del tablero y el número de jugadas que se deberían encadenar. El funcionamiento sería el siguiente, primero se crea una lista moviendo una ficha cada vez, después a sus nuevos estados intentaremos mover las fichas otras veces, y esto se repite hasta 4 veces. Si en alguno de los casos no se han generado movimientos posibles se descarta esa lista y se usaría la anterior, la que sí ha tenido movimientos. El planteamiento ha sido así para intentar evitar buscar 4 jugadas encadenadas de golpe. Se va progresando en el árbol de estados posible y eliminando aquellos que no pueden continuar.

```
def copia_para_hijo(self, origen: int, dado: int, color: str):
    estado_nuevo = Backgammon(self.turno, TableroBackgammon(self.tablero),
                                  None, self.jugadas)
    estado_nuevo.movimiento_ultimo = estado_nuevo.tablero.mover_ficha(origen,
                                         dado, color)
    return estado_nuevo

def copia_para_hijo_final(self, origen: int, dado: int, color: str):
    estado_nuevo = Backgammon((not self.turno),
                                  TableroBackgammon(self.tablero), None,
                                  self.jugadas + 1)
    estado_nuevo.movimiento_ultimo = estado_nuevo.tablero.mover_ficha(origen,
                                         dado, color)
    return estado_nuevo

def ganador(self) -> Ficha:
    if self.tablero.get_fichas_finalizadas_B() ==
        self.tablero.get_total_fichas_inicial_por_jugador():
        return self.FICHA_J1
    elif self.tablero.get_fichas_finalizadas_N() ==
        self.tablero.get_total_fichas_inicial_por_jugador():
        return self.FICHA_J2
```

Figura 6.23: Clase Backgammon (3)

En la figura 6.23, el primer método sirve para crear una copia del estado actual, una copia del tablero y realizar el movimiento en esa copia con los datos de entrada. Para retornar el nuevo estado que es una copia con un movimiento. Por su parte el segundo método es igual al primero pero cambia el turno e incrementa el número de jugadas ya que ésta sería la copia final de una serie de movimientos. Dicho de otro modo, tras la ejecución del segundo método, le tocará al siguiente jugador porque el anterior ya habrá realizado todos los movimientos que pudiera.

El tercer método de la misma figura comprueba si existe ganador. Para hacer esto simplemente comprueba que el conjunto de todas las fichas que han finalizado, que es un número que contiene el tablero, es igual al número total de fichas iniciales por cada jugador.

Nuevamente se reimplementa la forma de mostrar el estado actual del tablero como cadena de caracteres. Este método en realidad es bastante sencillo, ya que parte de la impresión el tablero y de las listas, lo cual está implementado en la clase TableroBackgammon. En la figura 6.24 se puede ver con más detalle la implementación.

```
def __str__(self):
    res = "----------BACKGAMMON----------\n"
    res += "Jugada numero: {}\n".format(self.jugadas)
    res += "Ultimo movimiento: {}\n".format(str(self.movimiento_ultimo.pos()
                                        + 1) if self.movimiento_ultimo is
                                                not None else "--")
    res += "TURNO JUGADOR: {}\n".format(self.ficha_actual().colorFicha())
    res += str(self.tablero)

    return res
```

Figura 6.24: Clase Backgammon (4)

De esta manera podemos ver a simple vista todos los detalles del juego, desde la cantidad de jugadas, el último movimiento realizado, el turno del jugador actual y la representación completa del TableroBackgammon. Este representa el tablero de juego con todas las posiciones, la cantidad de fichas de cada jugador fuera de juego y las que han finalizado. El resultado se puede ver en la figura 6.25.

```
----------BACKGAMMON----------
Jugada numero: 0
Ultimo movimiento: --
TURNO JUGADOR: B
_______________________________Inicio B
12 11 10 09 08 07 06 05 04 03 02 01
5B -- -- -- 3N -- 5N -- -- -- -- 2B

5N -- -- -- 3B -- 5B -- -- -- -- 2N
13 14 15 16 17 18 19 20 21 22 23 24
_______________________________Inicio N
Fichas fuera de juego:      0B y 0N
Fichas que han finalizado:  0B y 0N
```

Figura 6.25: Representación completa de un estado

En esta figura, la 6.25, ya muestra toda la información de un estado determinado.

6.4 Jugadores

El paquete jugadores contiene todos y cada uno de los posibles jugadores que pueden jugar al Backgammon en esta primer iteración. Por una parte tenemos la clase abstracta Jugador que contiene los métodos que deben implementar los jugadores para jugar al Backgammon. Y por otra los jugadores que, valga la redundancia, juegan.

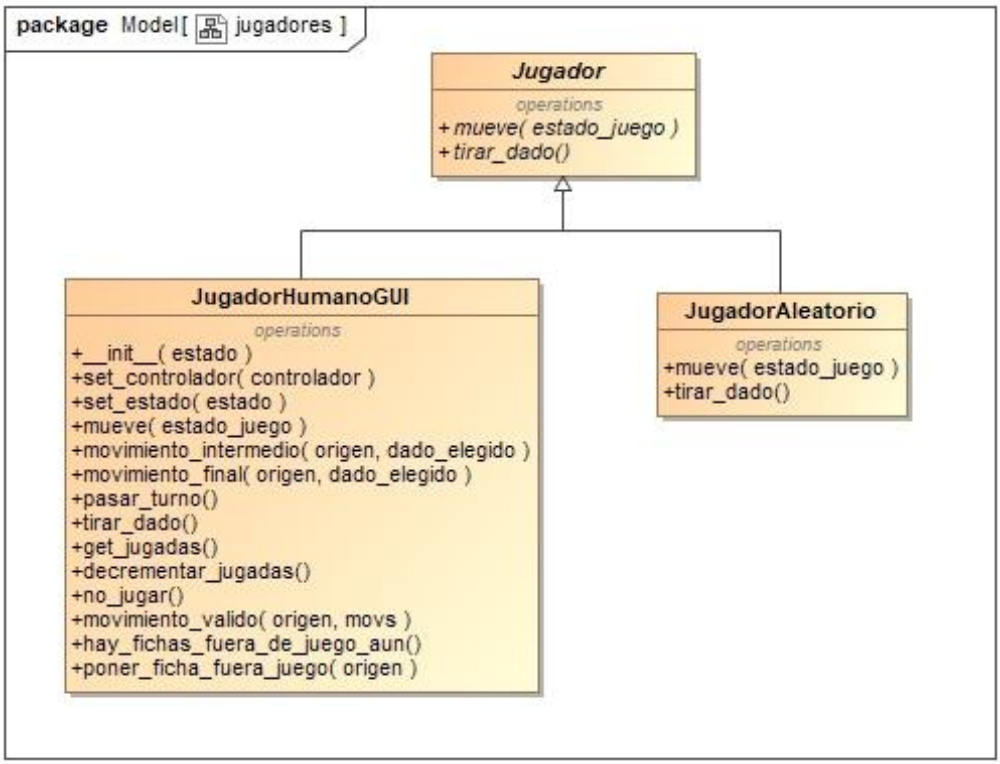

Figura 6.26: Diagrama de clases jugadores

En la figura 6.26 se observa con mayor detalle el diagrama de clases con los jugadores y las relaciones establecidas solo entre ellos.

6.4.1 Jugador

Esta es una clase abstracta que contiene dos métodos que serán usados por todos los jugadores del sistema para que pueda funcionar el programa correctamente. El caso de jugador humano con interfaz gráfica de usuario es un poco diferente, ya que al estar conectado con un controlador y trabajar a tiempo real, no se puede usar la recursividad que se usa con los jugadores del sistema para que jueguen partidas.

6.4.2 JugadorAleatorio

Este jugador no tiene constructor así que comencemos mostrando la definición del método mueve.

```
class JugadorAleatorio(Jugador):
  def mueve(self, estado_j: Backgammon) -> Backgammon:
    dados = self.tirar_dado()
    color = estado_j.ficha_actual().colorFicha()

    print_dados(dados, estado_j.ficha_actual().colorFicha())

    min = estado_j.get_tablero().MIN_POS_CUADRANTE_B if color ==
                         estado_j.get_tablero().get_color_blanco() \
                        else estado_j.get_tablero().MIN_POS_CUADRANTE_N

    max = estado_j.get_tablero().MAX_POS_CUADRANTE_B if color ==
                         estado_j.get_tablero().get_color_blanco() \
                        else estado_j.get_tablero().MAX_POS_CUADRANTE_N

    while estado_j.get_tablero().se_puede_poner_ficha_de_fuera_juego(color):
        pos = num_aleatorio_min_max(min, max)
        estado_j.get_tablero().poner_ficha_de_fuera_juego(pos, color)

    estados = list()
    if not estado_j.get_tablero().get_lista_fichas_fuera_juego(color):
        estados = estado_j.hijos_accesibles(dados)
    if estados:
        sel = num_aleatorio(len(estados) - 1)
        return estados[sel]
    else:
        print_("El jugador {} no puede moverse, se cambia
                 turno\n\n\n".format(color))
        estado_j.cambia_turno()
        return estado_j
```

Figura 6.27: Clase JugadorAleatorio

Como se puede ver en la figura 6.27, el funcionamiento del jugador aleatorio es simple. Recibe un estado de juego como entrada del método mueve, y para esa entrada busca y elige una posible jugada aleatoriamente. Para hacer esto primero revisa que no haya fichas fuera de juego, si las hay las tiene que poner, y la pondrá de forma aleatoria. Si no quedan fichas fuera de juego entonces generará todos los hijos a los que tiene acceso desde su estado, es decir, todos los posibles estados futuros inmediatos. De ellos seleccionará de forma aleatoria uno de ellos y lo retornará. Si por algún casual no pudiera moverse, o bien porque se da una situación de bloqueo y no puede poner fichas

que están fuera de juego, o bien porque la tirada de los dados no le ha dado un número con el que pueda jugar, entonces pasa turno.

El método `def tirar_dado(self)` simplemente retorna el resultado de llamar a `generar_dados()` del paquete recursos. Siempre será una tupla de 2 o 4 valores.

6.4.3 JugadorHumanoGUI

Esta clase corresponde al jugador con interfaz gráfica, hereda de Jugador, pero debido a la ausencia de recursión en esta clase mueve no se re implementa. Esta clase se ejecuta en tiempo real y las acciones del jugador desembocarán en un nuevo estado. Las acciones de otro jugador solo cambian el estado del tablero pero deberá esperarse a que el jugador humano vuelva a realizar alguna jugada.

```
def __init__(self, estado: Backgammon = None):
    self.dados = tuple
    self.dados_tirados = False
    self.jugadas_restantes = 0
    self.estado = estado

def set_controlador(self, controlador: ctrl_vista_basica):
    self.controlador = controlador

def set_estado(self, estado):
    self.estado = estado
```

Figura 6.28: Clase JugadorHumanoGUI (1)

A diferencia del jugador aleatorio, como se puede ver en la figura 6.28, en este jugador necesitamos saber si los dados se han tirado, las jugadas restantes que le quedan y un estado inicial del tablero. Así mismo cómo se sigue el patrón de diseño MVC, debe asignársele un controlador, como se ve en el primer método. Y cuando otro jugador juegue es necesario darle el nuevo estado, para ello se usa el segundo método.

Como se puede ver en la figura 6.29 existen movimientos intermedios que se dan cuando al jugador aún le quedan turnos. El movimiento final se usa en el último movimiento qué puede realizar el jugador, después de el se cambia el turno. En el caso del segundo método se retorna el nuevo estado al que ha llevado el jugador humano.

```
def movimiento_intermedio(self, origen: int, dado_elegido: int):
    self.estado = self.estado.copia_para_hijo(origen-1,
                                              self.dados[dado_elegido],
                                    self.estado.ficha_actual().colorFicha())

def movimiento_final(self, origen: int, dado_elegido: int):
    self.estado = self.estado.copia_para_hijo_final(origen-1,
                                              self.dados[dado_elegido],
                                    self.estado.ficha_actual().colorFicha())
    return self.estado

def pasar_turno(self):
    self.estado.cambia_turno()
    return self.estado

def tirar_dado(self):
    self.dados = generar_dados()
    self.dados_tirados = True
    self.jugadas_restantes = len(self.dados)
    return self.dados
```

Figura 6.29: Clase JugadorHumanoGUI (2)

En la figura 6.29 se define también los métodos para pasar turno, que simplemente le indica al estado que pase turno, y retorna ese nuevo estado, y otro método para tirar dados. Aunque en este caso tirar dados es un poco diferente, ya que genera los dados, le dice a la variable que los datos han sido tirados, e inicializa las jugadas restantes a la longitud de los dados, es decir, a si le han dado dos dados o cuatro dados, según el tipo de jugada.

```
def get_jugadas(self) -> int:
    return self.jugadas_restantes

def decrementar_jugadas(self):
    if self.jugadas_restantes > 0:
        self.jugadas_restantes -= 1

def no_jugar(self):
    self.jugadas_restantes = 0

def movimiento_valido(self, origen: int, movs: int):
    return self.estado.get_tablero().se_puede_mover_ficha_a(origen-1, movs,
                                    self.estado.ficha_actual().colorFicha())
```

Figura 6.30: Clase JugadorHumanoGUI (3)

En la figura 6.30 en el primer método se retornan las jugadas que aún le quedan al jugador, el segundo método decrementa las jugadas en uno en el caso de que haya más de cero, el tercer método hace que las jugadas pasen a 0, esto junto a pasar turno

se puede usar para no jugar. El último método de la figura le pregunta al tablero si, para una posición de origen y un movimiento concreto, la ficha se puede mover.

Por su parte los métodos `def hay_fichas_fuera_de_juego_aun(self)` y `poner_ficha_fuera_juego(self, origen: int)` sirven para decir si hay fichas fuera de juego que deba poner el jugador y poner una ficha de fuera de juego.

6.5 Controlador y Vista

Llegados hasta aquí, se han dado los detalles de implementación del Jugador y del Juego, y que sus hijos destinados a GUI necesitan de un controlador. En el patron de diseño MVC el Controlador hace de intermediario entre la lógica de negocio, en nuestro caso los jugadores y el juego, y una vista.

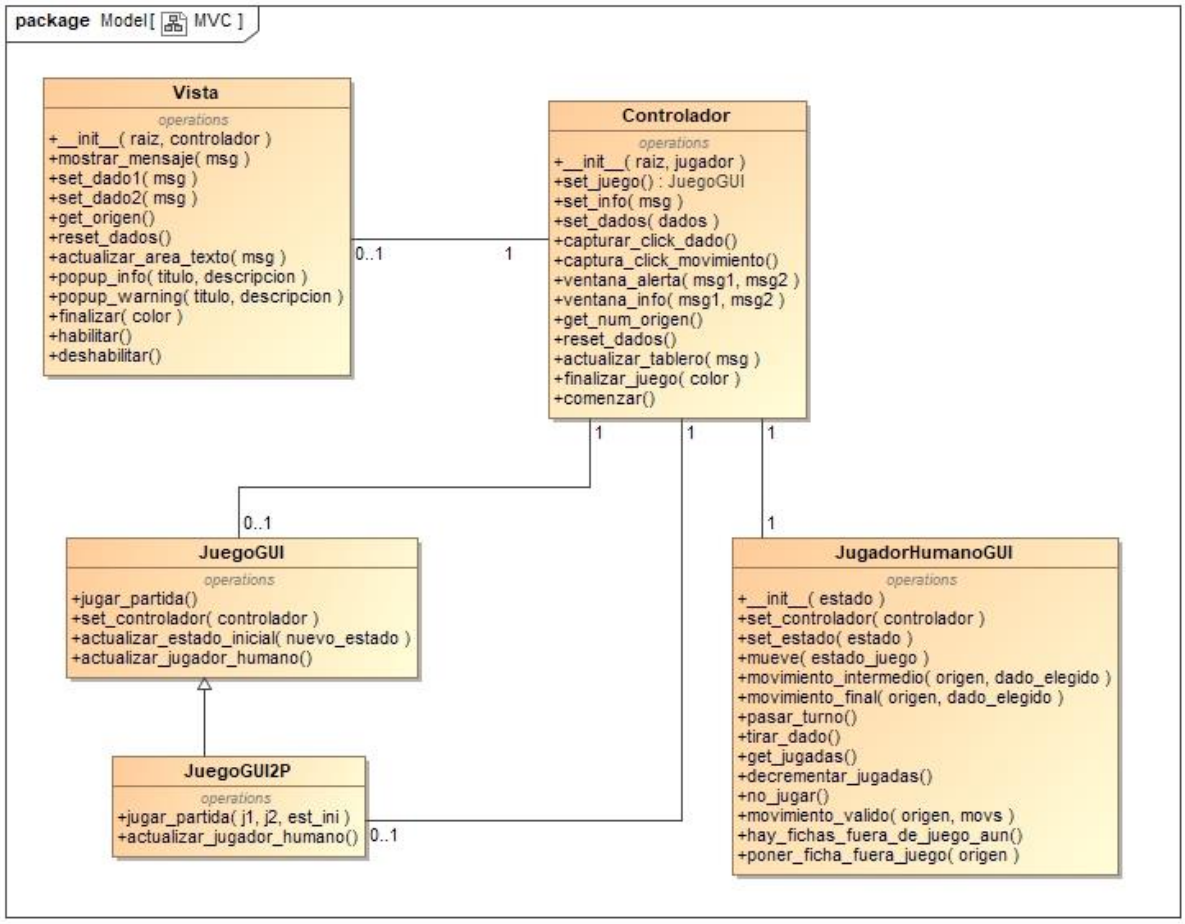

Figura 6.31: Diagrama de clases Modelo-Vista-Controlador

En esta ocasión se representa en el diagrama de clases de la figura 6.31 la relación entre Juego&Jugador-con-Controlador&Vista.

6.5.1 Controlador

Controlador tiene su paquete que se llama de la misma forma. Este es un intermediario como hemos comentado entre la vista, el juego y el jugador. El constructor, como se puede ver en la figura 6.32, de esta clase almacena la vista el jugador y la raíz. Además también la variable de posición que se desea mover.

```
def __init__(self, raiz, jugador: JugadorHumanoGUI):
    self.vista = VistaBasica(raiz, self)
    self.jugador = jugador
    self.numero_origen = None
    self.raiz = raiz

def set_juego(self, juego: Juego):
    self.juego = juego

def set_info(self, msg: str):
    self.vista.mostrar_mensaje(msg)

def set_dados(self, dados):
    self.vista.set_dado1(dados[0])
    self.vista.set_dado2(dados[1])

    if len(dados) == 2:
        self.set_info("Jugada comun. Tienes 2 movimientos.")
    elif len(dados) == 4:
        self.set_info("Jugada cuadruple. Tienes 4 movimientos.")

def capturar_click_dado(self):
    if not self.jugador.dados_tirados:
        self.set_dados(self.jugador.tirar_dado())
    else:
        self.set_info("Ya tiraste los dados")
```

Figura 6.32: Clase Controlador (1)

El primer método de la figura 6.32 asigna el juego a una variable para que el controlador pueda comunicarse entre la vista, el juego y el Jugador. El segundo método envía la cadena de texto que se desea mostrar en la parte inferior de la vista, es un mensaje informativo, el tercer método lo que hace es enviar a la vista el valor de los dados previamente generados, dependiendo de si hay dos o cuatro se muestra el tipo de jugada. El último método es lo que se conoce como un listener, este acciona los mecanismos para que se tiren los dados y se muestran. Si se tiraron no se tirarán.

```
def capturar_click_movimiento(self):
    origen = self.vista.get_origen()
    if origen.isdecimal():
        origen = int(origen)
        hay_fuera_juego = self.jugador.hay_fichas_fuera_de_juego_aun()
        self.set_info("Hay fichas fuera de juego que debes poner antes") if
                                                    hay_fuera_juego else None
        if hay_fuera_juego and ((1 <= origen <= 6 and self.juego.get_turno())
                 or (19 <= origen <= 24 and not self.juego.get_turno())):
            mov_de_fuera = self.jugador.poner_ficha_fuera_juego(origen)
            if mov_de_fuera is None:
                self.set_info("No se puede poner la ficha de fuera de juego
                              en la casilla {}".format(origen))
        elif not hay_fuera_juego and 1 <= origen <= 24:
            if self.jugador.dados_tirados:
                num_jugadas = self.jugador.get_jugadas()
                if num_jugadas > 0:
                    self.numero_origen = int(self.vista.get_origen())
                    self.set_info("Vas a intentar mover la ficha {}
                                  ".format(self.numero_origen))
                    dado_actual = len(self.jugador.dados) - num_jugadas
                    if self.jugador.movimiento_valido(origen,
                                  self.jugador.dados[dado_actual]):
                        if num_jugadas == 1:
                            self.juego.jugar_partida(False,
                                  self.jugador.movimiento_final(origen,
                                                        dado_actual))
                            self.jugador.decrementar_jugadas()
                            self.reset_dados()
                            self.numero_origen = None
                        else:
                            self.jugador.movimiento_intermedio(origen,
                                                        dado_actual)
                            self.jugador.decrementar_jugadas()

                        self.actualizar_tablero(str(self.jugador.estado))
                    else:
                        self.set_info("Jugada no valida")
                else:
                    self.set_info("Turno de la maquina")
                    sleep(3)
            else:
                self.ventana_info("Debes lanzar los dados", "Antes de mover
                                  tienes que lanzar los dados")
        else:
            if self.juego.get_turno():
                self.ventana_info("Numero fuera de rango",
                                   "Debes usar valores desde 1 hasta
                                  {}".format(6 if hay_fuera_juego else 24))
            else:
                self.ventana_info("Numero fuera de rango",
                                   "Debes usar valores desde {} hasta
                                  24".format(19 if hay_fuera_juego else 1))
```

Figura 6.33: Clase Controlador (2.1)

```
elif origen == "-":
    self.reset_dados()
    self.juego.jugar_partida(False, self.jugador.pasar_turno())
    self.jugador.no_jugar()
    self.numero_origen = None
else:
    self.numero_origen = None
    self.ventana_alerta("Valor de origen no valido", "Introduce un valor
                        numerico en el origen")
```

Figura 34: Clase Controlador (2.2)

La captura del evento de click del botón de movimiento de la vista, que se puede ver en la figura 6.33 y 6.34, es el más importante de todos ya que debe controlar que la entrada de la vista, llamado origen, sea un numero valido en función de la jugada que se deba realizar. Existen varias casuísticas.

- Primero comprueba que sea un valor decimal, en el caso de que se desee pasar turno comprueba que si no es un numero sea un guion (-) ya que en ese caso pasara el turno al enemigo y en cualquier otro caso mostrara una ventana de advertencia.
- Si es un numero después comprueba si existen fichas fuera de juego, esta siempre deberán ser jugadas continuar. Si existen fichas fuera de juego el valor introducido deberá ser entre 1 y 6 o 19 y 24, dependiendo del color, en otro caso muestra una ventana de información. Si existieran fichas fuera de juego que se deban jugar se jugaran. Comprobaran si la posición a la que van a ir es válida, si no lo fuera se muestra en el texto inferior de información.
- En el caso que no haya fichas fuera de juego el numero deberá estar dentro del tablero, posiciones de 1 a 24, si el número es mayor o menor a ese rango se muestra una ventana de información
- Tras ello comprueba que se hayan tirado los dados y en caso de que no se comunica al usuario.

- Por ultimo va realizando jugadas hasta que no queden tiradas, en ese caso le pasa el turno al jugador contrario.

```
def ventana_alerta(self, msg1: str, msg2: str):
    self.vista.popup_warning(msg1, msg2)

def ventana_info(self, msg1: str, msg2: str):
    self.vista.popup_info(msg1, msg2)

def get_num_origen(self):
    return self.numero_origen

def reset_dados(self):
    self.vista.reset_dados()
    self.jugador.dados_tirados = False

def actualizar_tablero(self, msg: str):
    self.vista.actualizar_area_texto(msg)

def finalizar_juego(self, color: str):
    self.vista.finalizar(color)
    del self.juego
    self.raiz.destroy()

def comenzar(self):
    self.vista.habilitar()
    self.juego.jugar_partida()
    self.vista.actualizar_area_texto(str(self.jugador.estado))
```

Figura 6.35: Clase Controlador (3)

El primer y el segundo método, de la figura 6.35, le dicen a la vista que lance ventanas de alerta y de información respectivamente. El tercer método retorna el número de origen que corresponde al dato que se haya introducido, sí se ha introducido, en la entrada de la ventana. El método reset dados llama a resetear datos de la vista y le dice al jugador que los dados los tiene disponibles, esto sucederá cuando acabe su turno. El cuarto método es actualizar el tablero, simplemente se le deberá mandar como argumento de entrada la representación del tablero, y este se dibujará en la vista.

Comenzar y finalizar el juego lo que hacen respectivamente si se pulsa el botón de empezar de la vista se habilitan los botones y el juego empieza a funcionar. También se actualiza el área de texto con el estado inicial o la jugada si jugó primero el enemigo. Finalizar el juego por su parte dice quien ha sido el jugador qué ha ganado, elimina el juego y destruye la raíz, es decir, acaba el programa.

6.5.2 VistaBásica

Representa la vista sencilla pero usable de nuestro juego Backgammon, en la figura 6.36 se comienza a detallar su implementación, concretamente ahí se construye la vista del programa y se enlaza con el constructor.

Detallando la vista cómo de construye, siguiendo de arriba abajo el código, la vista se construye de la siguiente manera. Primero se establece la comunicación con el Controlador, para cada vez que se haga en el clic en cualquier botón, sepa que hay que hacer, asi mismo muestre detalles en la vista.

Primero se le pone título a la ventana y se establece un frame principal sobre el que se pondrán objetos de la vista. A continuación se crea un "panel de texto" donde mostrar el tablero, esto es un área de texto grande.

Se crea un subpanel de comandos el cual estará situado a la derecha del área de texto. En el subpanel derecho, ponemos de arriba abajo, primero el botón de Empezar, después la un título, la posición del origen de la ficha (input). Se crea un subpanel para la información de los dados, este es un grip de dos filas y dos columnas en los cuales están los títulos de los dos dados y dos entradas de texto deshabilitadas que no se pueden alterar pero muestran el valor de los dados. Abajo se pone el botón de los dados para que cuando se pulse se llama controlador y se muestren los dados.

A continuación creamos el botón de Acción, este tiene un titulo arriba y también se da una opción un subtítulo bajo el botón, en principio mostrando información. Tras ello comienza la ubicación de cada uno de los componentes haciendo llamadas a pack, al menos a los que antes no se situaron.

Por último, el último componente de la vista es la línea de información de la parte inferior. E inicialmente se deshabilitan los botones hasta que se pulse el botón de empezar partida.

```
class VistaBasica():
  def __init__(self, raiz: Tk, controlador):
    self.controlador = controlador
    self.raiz = raiz
    self.raiz.title("Backgammon - Sergio Tineo")
    self.raiz.resizable(0, 0)
    try:
        WORKDIR = Path(__file__).parent.parent
        raiz.iconbitmap(WORKDIR/"logo"/"icono_bg_py_st.ico")
    except:
       pass
    frame_principal = Frame(self.raiz, bd=10)
    frame_principal.pack()
    self.txt_tablero = Text(frame_principal)
    self.txt_tablero.config(bd=0, width=38, height=14, font=("Consolas", 14))
    self.txt_tablero.insert(END, "-------------------------------------\n" +
                            "-------BIENVENIDO A BACKGAMMON-------\n" +
                            "-------------------------------------")
    self.txt_tablero.grid(row=0, column=0)
    frame_comandos = Frame(frame_principal, padx=15)
    frame_comandos.grid(row=0, column=1)
    self.boton_empezar = Button(frame_comandos, text="Empezar", width=12,
                                command=self.controlador.comenzar)
    self.boton_empezar.pack(side="top")
    Label(frame_comandos).pack(side="top")  # Espacio
    label_origen=Label(frame_comandos,text="Origen",
                       font=("TkDefaultFont",11))
    label_origen.pack(side="top")
    self.origen = Entry(frame_comandos)
    self.origen.config(justify="center", width=17)
    self.origen.pack(side="top")
    Label(frame_comandos).pack(side="top")  # Espacio
    frame_dados = Frame(frame_comandos)
    label_d1 = Label(frame_dados, text="Dado 1", font=("TkDefaultFont",11))
    label_d2 = Label(frame_dados, text="Dado 2", font=("TkDefaultFont",11))
    label_d1.grid(row=0, column=0, padx=1)
    label_d2.grid(row=0, column=1, padx=1)
    self.valor_d1 = Entry(frame_dados, width=8, justify="center",
                       state="readonly")
    self.valor_d2 = Entry(frame_dados, width=8, justify="center",
                       state="readonly")
    self.valor_d1.grid(row=1, column=0, padx=2)
    self.valor_d2.grid(row=1, column=1, padx=2)
    self.boton_dados = Button(frame_dados, text="Lanzar dados", width=12,
                              command=self.controlador.capturar_click_dado)
    self.boton_dados.grid(row=2, column=0, pady=5, columnspan=2)
    frame_dados.pack(side="top", pady=5)
    Label(frame_comandos).pack(side="top")  # Espacio
    label_accion = Label(frame_comandos, text="Realizar movimiento",
                                              font=("TkDefaultFont",11))
    self.boton_accion = Button(frame_comandos, text="Movimiento", width=12,
                       command=self.controlador.capturar_click_movimiento)
    self.label_resto_acciones = Label(frame_comandos, text="Con '-' cambias
                       el\nturno sin jugar", font=("TkDefaultFont",8))
    label_accion.pack(side="top", pady=2)
    self.boton_accion.pack(side="top", pady=2)
    self.label_resto_acciones.pack(side="top", pady=2)
    self.label_info = Label(self.raiz, text="Información", justify="left",
```

Figura 6.36: Clase VistaBasica

Al principio en la raíz se define un icono, la clausula try-except simplemente se usa para que en el caso de que no encuentre no falle la ejecución, simplemente se muestra la vista sin icono.

En la Vista los métodos que hay son simplemente para mostrar información al usuario. Toda la lógica la tiene el Controlador así que veo innecesario repetir cosas que en el propio controlador se explican cómo se hace, para que se hacen y cuál es el resultado.

Pero se pueden mencionar ciertos detalles de algunas funciones de Tkinter en Python. Por ejemplo, para poder alterar los dos entradas de texto correspondiente a la de los dados, que están deshabilitadas, es necesario habilitarla, poner el valor y deshabilitarla.

Para actualizar el área de texto, es un poco diferente a como nos podríamos imaginar, nos tenemos que situar al principio del área de texto y borrar del principio al final con palabras reservadas. Entonces después puedes insertar el texto, que aparecerá desde la primera posición, ya que se ha eliminado lo que tenía.

El resultado de cómo es la vista la tenemos, en las figuras 6.39 se muestra el estado inicial de la venta y 6.40 el estado después de haber empezado. Las figuras 6.37 muestra un mensaje de información y las 6.38 uno de advertencia.

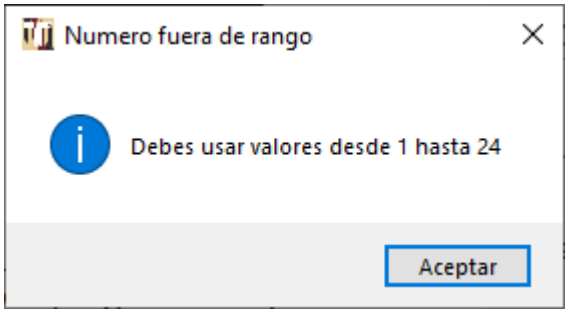

Figura 6.37: PopUp informacion

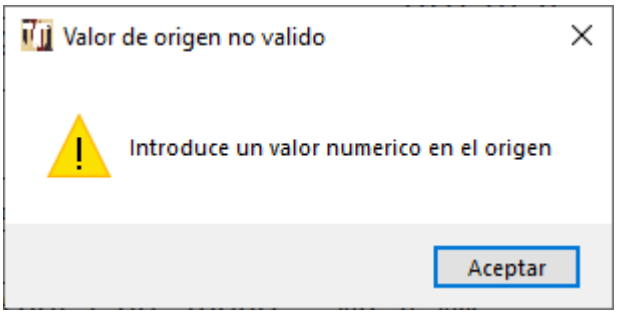

Figura 6.38: PopUp advertencia

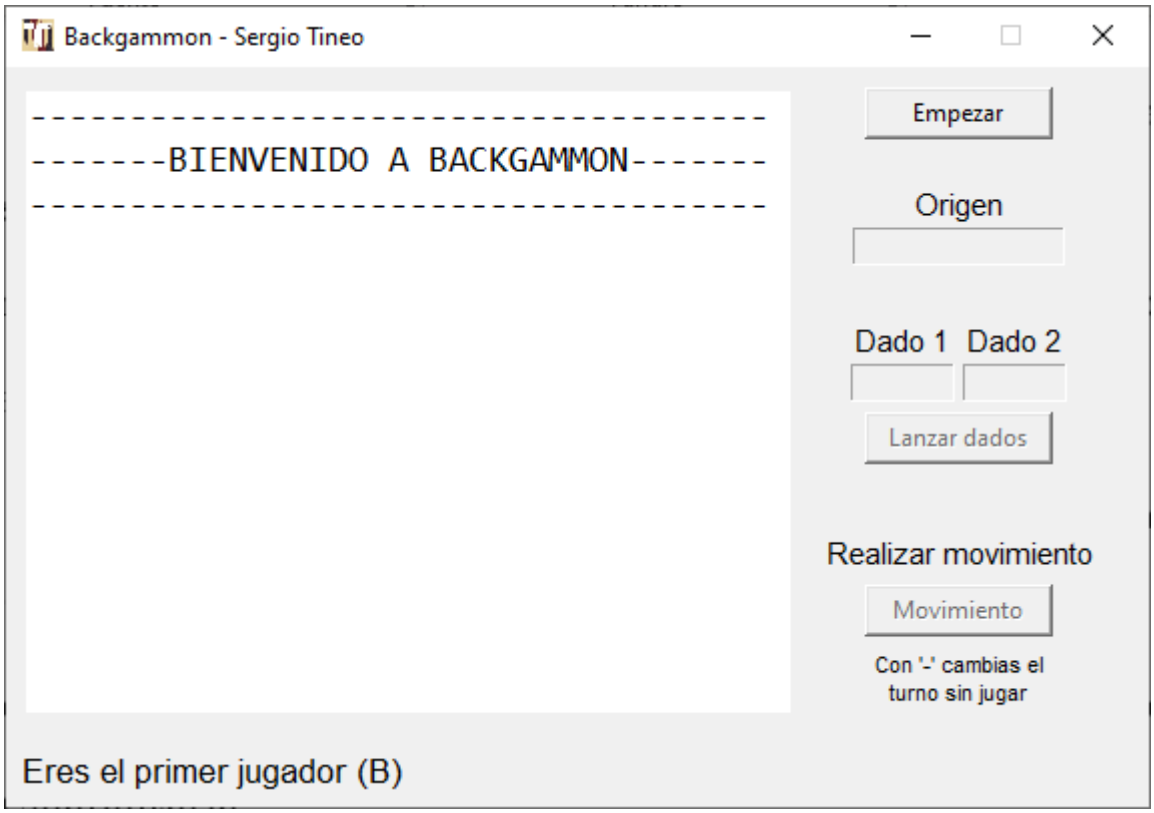

Figura 6.39: Ventana de inicio

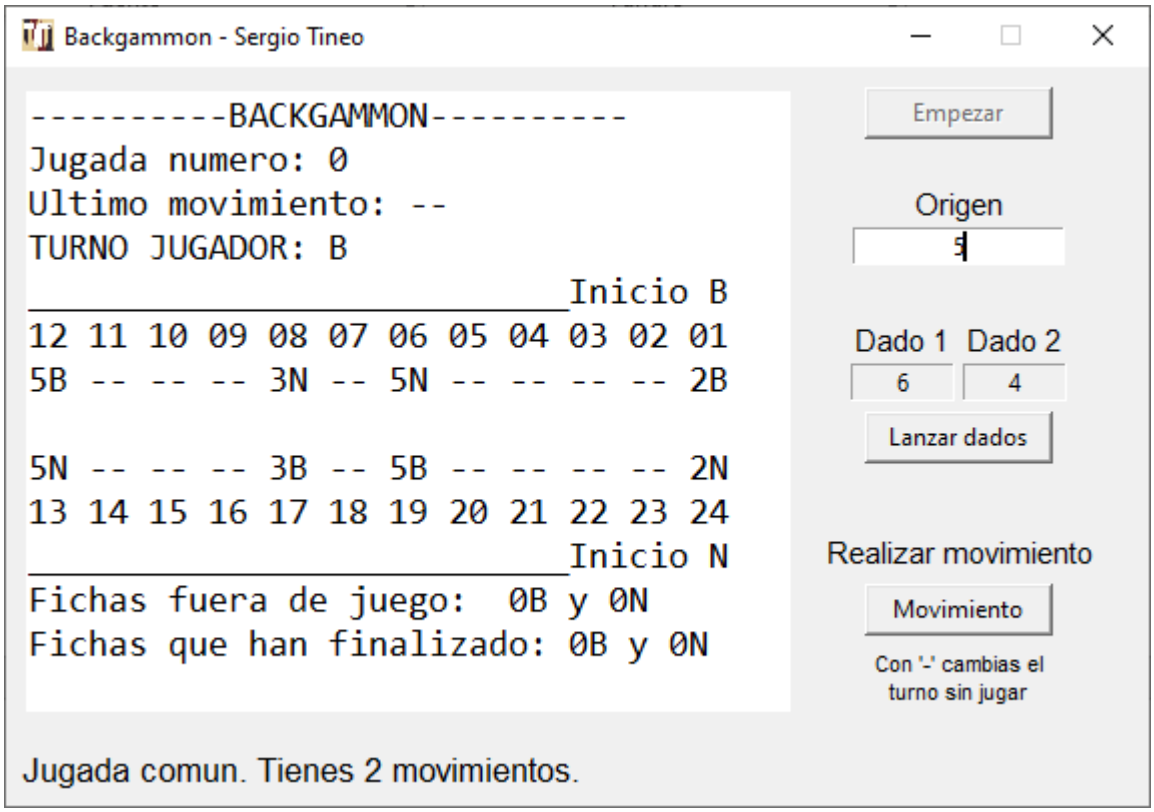

Figura 6.40: Ventana tras empezar y tirar dados

Detallando la vista de la GUI, figura 6.40, en el área de texto se ven todos los datos que contiene el estado en el turno del jugador al que le toca jugar.

Viendo la figura y describiéndola de arriba abajo, se describe así, la primera línea es el número de jugadas, recordemos que una jugada se da cuando se ha pasado turno y hubo movimientos. Último movimiento corresponde a el destino de la última ficha que ha sido desplazada. Turno jugador corresponde al turno del jugador actual (B o N). Inicio B representa el inicio de las blancas, qué es arriba a la derecha, los números de 12 al 1 son las posiciones de la parte superior del tablero. Bajo estos existe un número y una letra (xB o xN) que corresponde a la cantidad de fichas en esa casilla (x) y el color de la ficha. La línea siguiente se describe igual. Y la siguiente a esa que va del número del 13 al 24 corresponden a las otras casillas del tablero. Así como Inicio N es el inicio de las fichas negras.

Bajo esto se pueden ver las fichas que están fuera de juego, la cantidad y el color, y tras esto las fichas que han finalizado usando el mismo formato.

7

Pruebas

En el presente capítulo se detallaran las pruebas que se han realizado, aunque no se pondrá el código, ya que este está en el propio proyecto y extendería innecesariamente el trabajo. Se explicará detalladamente en qué consiste cada una de las pruebas.

7.1 Test unitarios

Las funciones necesarias del paquete recursos, las clases y los métodos del presente trabajo han pasado tests unitarios para verificar su comportamiento, y que conforme el programa fue creciendo no sucedieran errores. En total se han pasado 40 pruebas a esta iteración. Independientes de las pruebas de la interfaz.

7.1.1 Tests Espacio Juego

Se han realizado pruebas para la clase Ficha que está detecte correctamente cuando dos fichas son iguales es decir tienen tiempo de color o son diferentes. También se ha probado a intentar introducirle más letras a la Ficha o valores incorrectos. Los resultados han sido o bien un valor nulo cuando se solicita el color o bien elevar una excepción.

Por su parte, sobre clase Fichas se ha comprobado la comparación entre fichas iguales y fichas diferentes. Que se incrementen correctamente la cantidad de Ficha en

Fichas y que se decremento correctamente. También que cuando se decremente y se quite la última ficha, la referencia a la pila de fichas deba ser eliminada.

Las pruebas de la clase Movimiento han sido, que tras su creación se retorne el valor correctamente y que el resultado de para el objeto a cadena de caracteres también sea correcto.

Las pruebas sobre la clase Tablero han consistido en comprobar el tamaño correcto del tablero, que el contenido que retorne sea un valor válido. Se comprueban posiciones válidas y posiciones inválidas con el método de posición válida. Se comprueba que cuando se crea un tablero este inicialmente tiene las fichas donde deberían de estar. También se prueba a poner una pieza en un hueco vacío, a poner dos piezas del mismo color, a intentar poner una pieza de distinto color, y a que no le corresponda ser puesta. Quitar una ficha cuando hay dos, quitar una ficha cuando hay una e intentar quitar una ficha cuando no hay, y que no suceda ningún error, ni se modifique el tablero.

También se prueba que las clase abstracta funcione bien, concretamente la clase EstadoJuego, y no pueda ser instanciada.

Las pruebas sobre los juegos que interactúan con el controlador se han realizado junto con las pruebas de verificación de funcionamiento de la GUI.

7.1.2 Test Jugadores

Se prueba que las clase abstracta funcione bien, concretamente la clase Jugador, y no pueda ser instanciada.

A que sea aprobado el JugadorAleatorio realice acciones y esta acción tenga sentido, es decir, el jugador ponga las fichas que estén fuera de juego, que mueva ficha cuando pueda y que cambie de turno cuando no pueda jugar. También se ha probado a tirar los dados y que funcione.

Las pruebas sobre los jugadores que interactúan con el controlador se han realizado junto con las pruebas de verificación de funcionamiento de la GUI.

7.1.3 Backgammon

En la clase TableroBackgammon, se ha probado a mover ficha y que se mueva adecuadamente, es decir, que se pueda mover a una casilla vacía, a una casilla con una ficha de su mismo color o a una casilla donde haya solo un enemigo y este enemigo sea capturado. También se comprueba que las fichas finalicen correctamente y que se puedan poner fichas de fuera de juego en el tablero, en el caso de que haya que ponerlas. Se comprueba si se puede mover una ficha a un destino pero sin moverla. Y se prueba que si se pide la lista de fichas de fuera de juegos se devuelva la correcta.

En la clase Backgammon se prueba que las fichas de referencia de J1 y J2 sean correctas. Se comprueba que se retorne adecuadamente la lista de hijos accesibles en función de las tiradas de dados. Se verifica las referencias de copia para hijo, de forma que no salte el contenido de uno si usamos otro. Se revisa que la condición de ganar partida funciona adecuadamente. Y se comprueba que los métodos de cambiar turno y aumentar jugador funcionen adecuadamente.

7.2 Test interfaz gráfica de usuario

Estos test han consistido en ir probando reiteradamente el programa de forma que nunca se llegue a un estado de bloqueo. Además se vigilan que las entradas de datos sean tolerante a cualquier tipo de dato. En el caso de los botones se revisa que no suceda nada o suceda cuando deba de suceder. También se revisa que las ventanas de información y advertencia se lancen cuando deban de lanzarse. Y que el usuario siempre pueda estar informado de lo que sucede en el tablero mediante el mensaje de la parte inferior de la ventana. En la figura 7.1 se revisa que inicialmente solo se pueda interactuar con el botón de empezar, en este caso se puede modificar la ventana de texto pero esto no causa ningún efecto. La figura 7.2 prueba el inicio correcto, por lo que efectivamente se inicia bien aunque se escriba algo incorrecto en el area de texto que representa el estado actual.

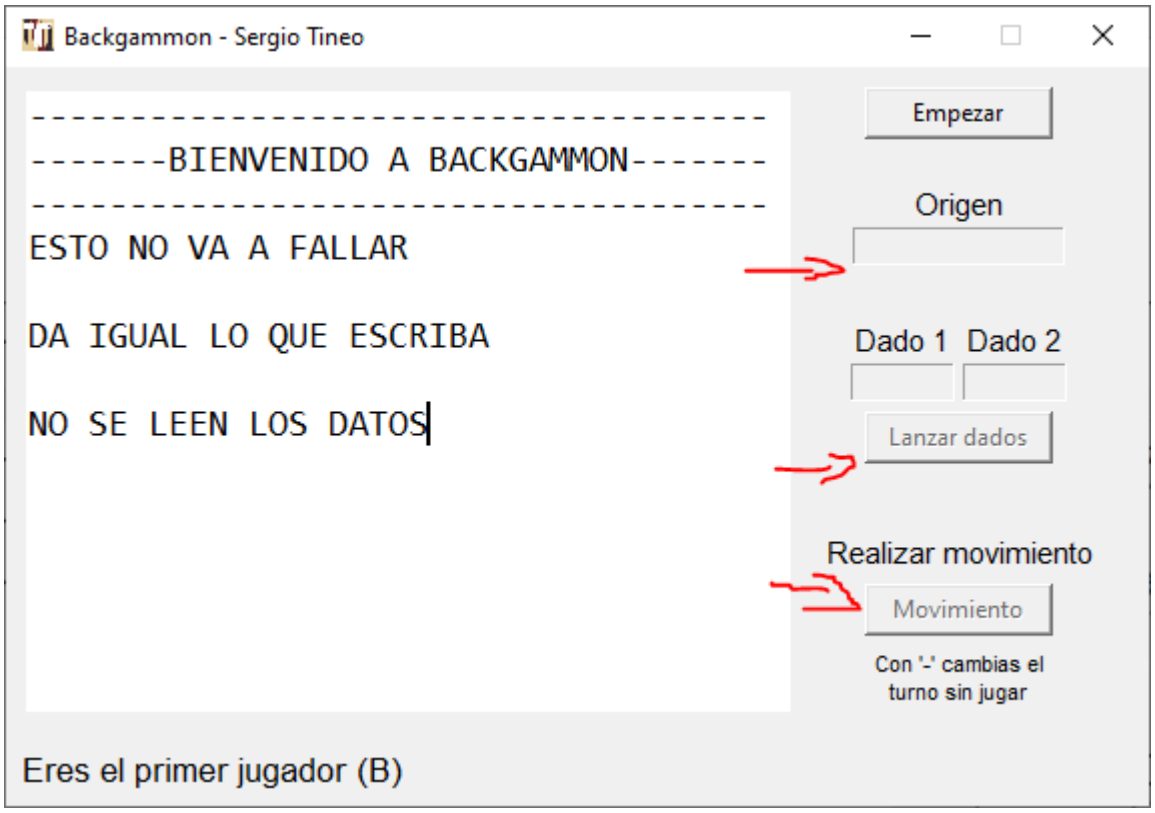

Figura 7.1: Prueba de inhabilitación inicialmente de todas las entradas y botones menos Empezar

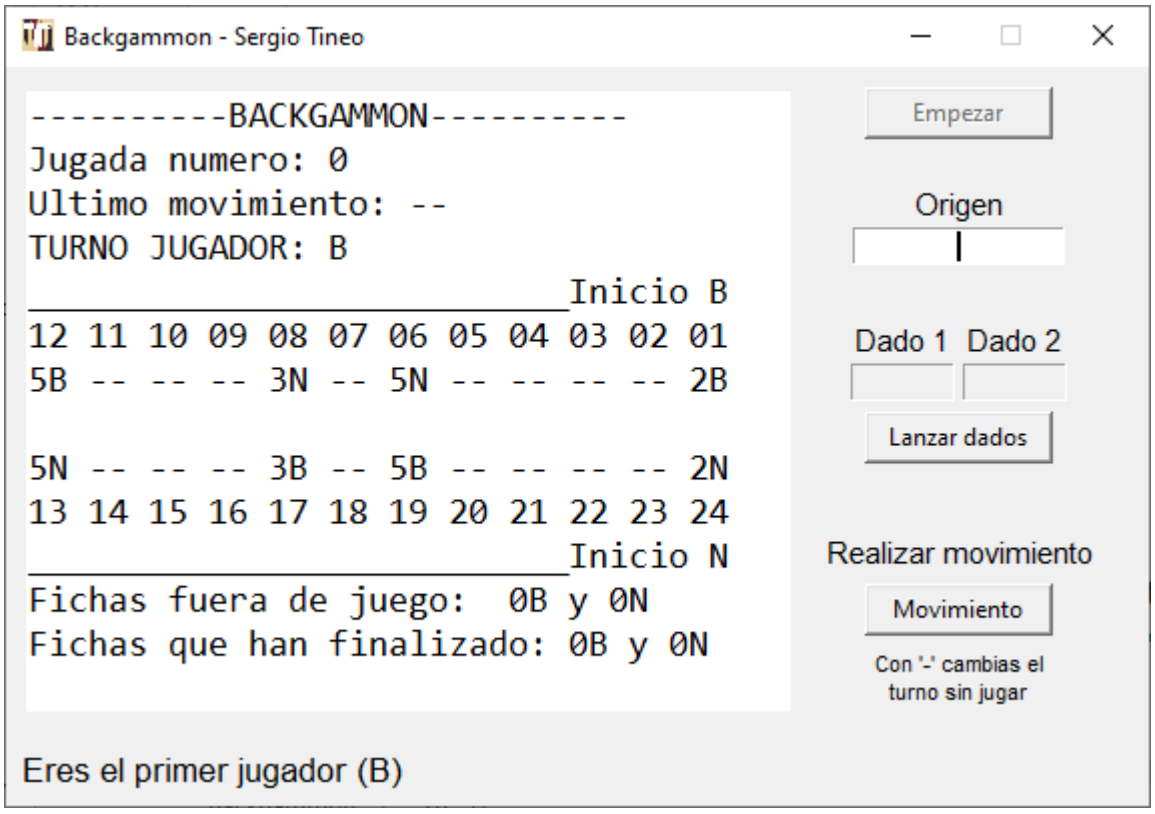

Figura 7.2: Prueba inicio correcto

La figura 7.3 muestra un movimiento valido, la 7.4 intentar volver a tirar los dados y la 7.5 intentar poner una ficha en un lugar no válido.

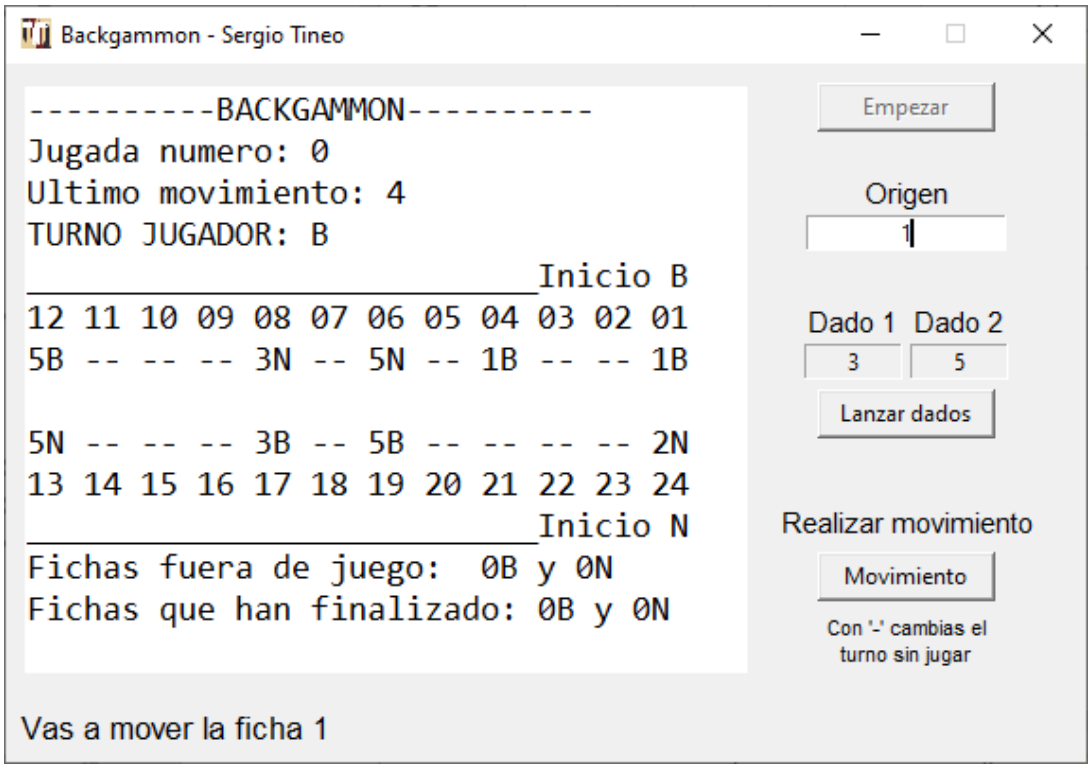

Figura 7.3: Prueba movimiento válido

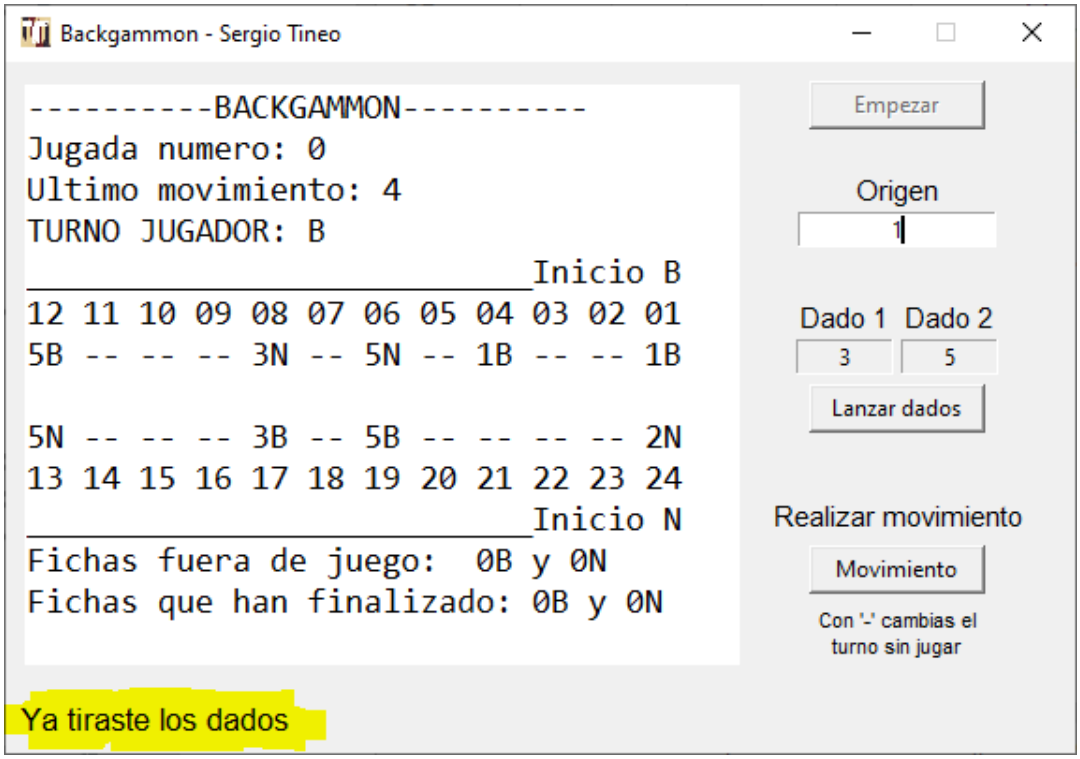

Figura 7.4: Prueba intento de lanzar dados dos veces

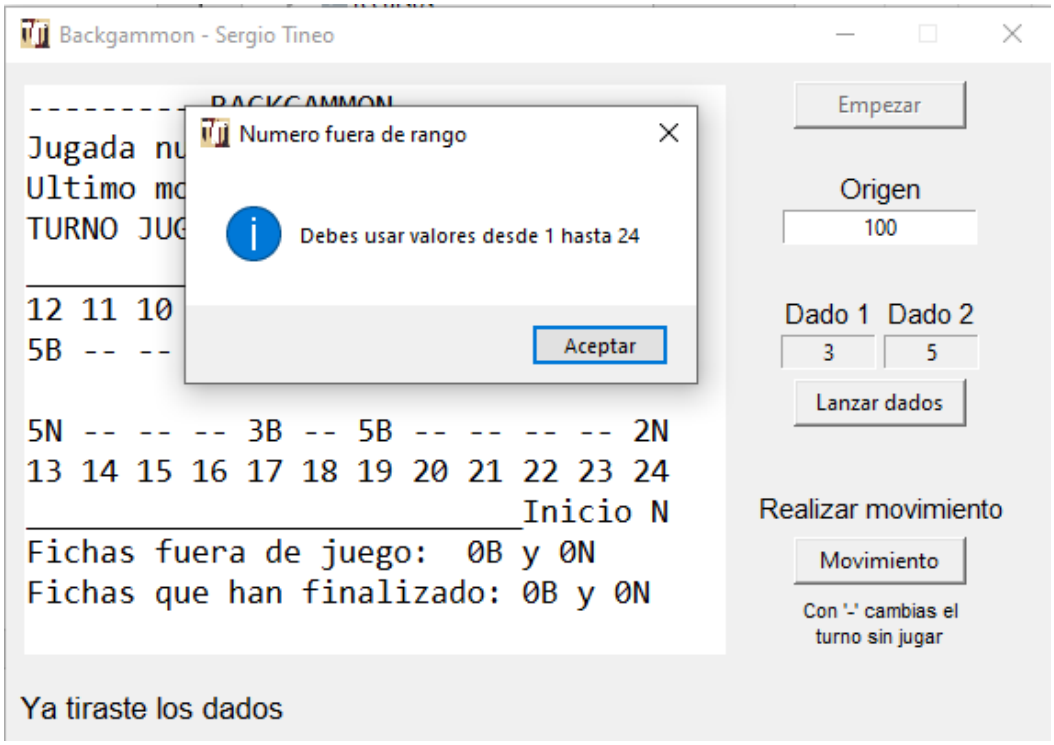

Figura 7.5: Prueba valor de entrada fuera de rango

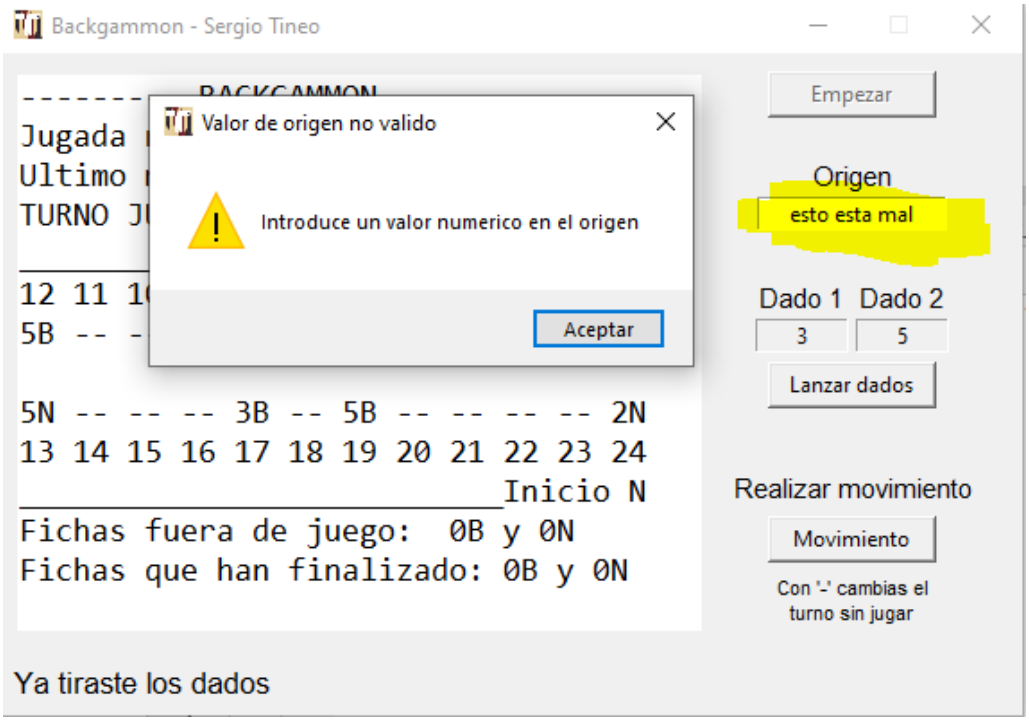

Figura 7.6: Prueba valor de entrada no es numero

También se verifica que cuando haya un ganador, este se notifique y se cierre el juego, evidencia en figura 7.7.

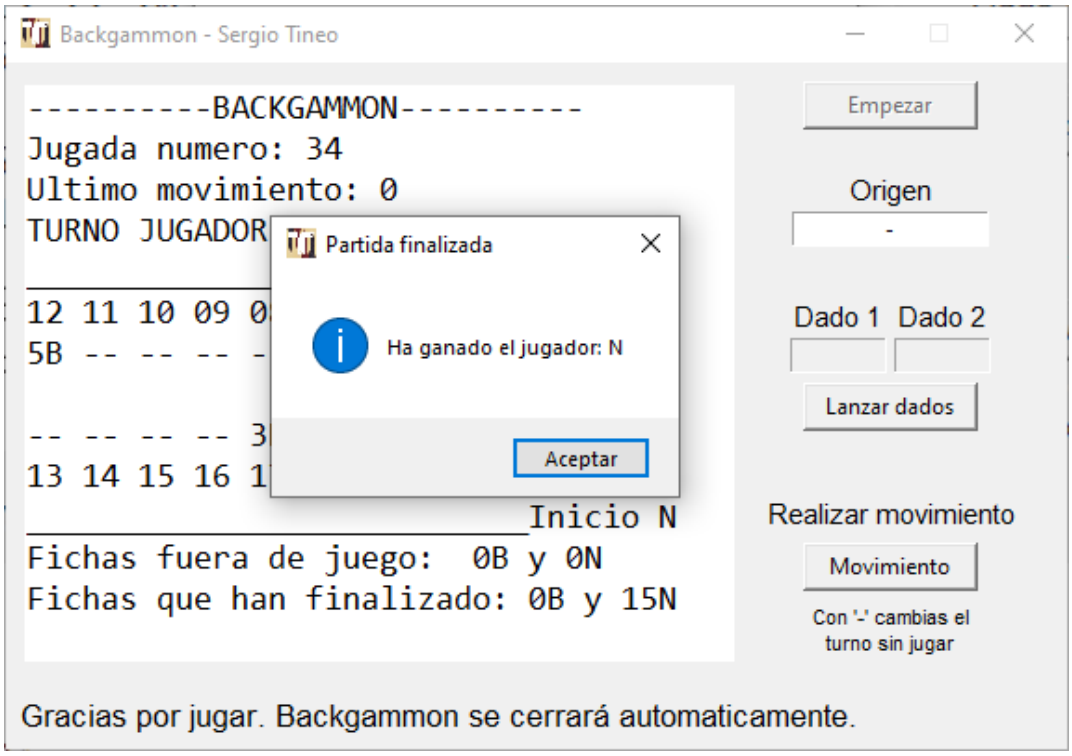

Figura 7.7: Prueba ganador

8

Análisis y Diseño Aprendizaje por refuerzo

Es en este octavo capítulo cuando se detalla el análisis previo a la implementación del aprendizaje por refuerzo al juego de Backgammon, que ya ha sido desarrollado en la primera iteración. También realizaremos su diseño centrándonos en los cambios con respecto a la primera iteración. El análisis y diseño que se verán en este capítulo serán los correspondientes a la segunda iteración, el **segundo iterable**, como se suele decir en la metodología de desarrollo que estamos llevando en el trabajo. Dado que el análisis y el diseño de la aplicación final esta vez dividida en dos partes nuevamente hemos decidido juntar en este mismo capítulo la etapa de análisis y la etapa de diseño.

8.1 Análisis

En este apartado se detallará qué pretendemos que realice el programa una vez este haya sido finalizado, al menos lo correspondiente a esta segunda iteración. Así mismo dada la dimensión de la aplicación y además su división en dos iterables, se deberían generar dos **Documento General de Requisitos**, el segundo lo vamos a

integrar a continuación eliminando elementos innecesarios con el fin de no extender más de lo necesario la memoria ya que se repetiría demasiado los mismos detalles.

8.1.1 Introducción

8.1.1.1 Objetivos

Estos fueron detallados en el capítulo 1 apartado 2.

8.1.1.2 Alcance

El alcance del presente trabajo abarca todos los usuarios que quieran jugar al Backgammon así como ver competir un sistema con aprendizaje y además poder extraer datos del aprendizaje automatizado.

8.1.1.3 Definiciones

Jugador, es el usuario que jugará contra la máquina.

Jugada, es el conjunto de acciones de lanzar los dados y realizar los movimientos necesarios o pasar turno para que sea el turno del siguiente jugador.

Sistema con aprendizaje, es un sistema que actúa dentro de las normas del juego habiendo aprendido contra un jugador aleatorio antes a jugar.

8.1.1.4 Resumen

Se va a realizar un programa con el que se puedan llevar a cabo diferentes acciones usando como entorno el juego de mesa Backgammon desarrollado en la primera iteración. Estas acciones serán hacer aprender al sistema con aprendizaje y poder verlo entrenar. Otro uso del programa será que el jugador mediante una interfaz gráfica de usuario pueda jugar contra el sistema con aprendizaje. Y por último ser capaces de generar ficheros con los datos de haber ido aprendiendo, y que estos puedan ser ejecutados por MatLab u otro programa de investigación capaz de hacer gráficas mediante un vector de números, básicamente para poder ver en un gráfico de líneas la progresión de aprendizaje.

8.1.2 Participantes

8.1.2.1 Detalle de los participantes

Nombre	Descripción
Jugador	Jugará al Backgammon o podrá lanzar y ver una partida automatizada del sistema con aprendizaje en el cual el sistema aprenderá y podrá crear estadísticas.

Tabla 8.1: Detalle de participantes

8.1.2.2 Perfil del participante

Representante	Jugador
Descripción	Podrá usar el programa en su totalidad
Tipo	Persona que quiera jugar al Backgammon, ver como juega el sistema o generar estadísticas de aprendizaje.
Criterio de éxito	Que sepa lanzar un script Python

Tabla 8.2: Perfil de participante

8.1.2.3 Alternativas y competencia

Actualmente existen numerosos juegos de Backgammon tanto online como en forma de aplicación, la mayoría sin técnicas de aprendizaje.

8.1.3 Visión general del producto

8.1.3.1 Entorno

El entorno normal de funcionamiento será cualquier ordenador o computadora capaz de ejecutar el intérprete de Python. Más detalles en el apéndice.

8.1.3.2 Características

Beneficios para el cliente	Descripción
Jugar a Backgammon	Poder ver y realizar jugadas en el juego Backgammon
Generar estadísticas de aprendizaje	Se podrán generar estadísticas de aprendizaje basadas en la experiencia del sistema con aprendizaje. Evaluando así que es mejor que un sistema sin aprendizaje
Jugadas validas	Se comprobará que las jugadas de cualquier jugador siempre sean válidas

Tabla 5.3: Características del entorno

8.1.4 Requisitos funcionales

Los requisitos de la primera iteración (véase capítulo 5) se presuponen ya para esta iteración ya que el jugador deberá poder hacer lo mismo que antes y además tendrá nuevas funcionalidades. A continuación se exponen únicamente los nuevos requisitos.

RF1. El sistema con aprendizaje podrá aprender jugando contra un sistema sin aprendizaje.

RF2. El jugador podrá generar estadísticas de aprendizaje.

8.1.5 Requisitos no funcionales

Igualmente los requisitos no funcionales de la primera iteración son fundamentales, por ello también estarán presentes en el programa. Pero ahora nos centraremos en los nuevos.

RNF1. *Operacional.* Se podrá ver la ejecución del avance del aprendizaje durante en tiempo real si se desea.

RNF2. *De portabilidad.* Se generarán ficheros de texto con estadísticas que puedan ser abiertos con cualquier programa y ejecutadas en algún programa de investigación como MatLab.

Junto a cada requisito no funcional se puede observar la categoría correspondiente que tiene, siguiendo las categorías del documento [17].

Se hace necesario matizar que hay ciertos puntos como las directivas del proyecto, los costes, las licencias, etc. que nuevamente no se han tratado en el DGR integrado en el presente capítulo para evitar una extensión innecesaria. Cierto es que para ejecutar cómodamente el script de estadísticas que genera el programa se hace interesante tener licencia de MatLab, pero no es necesario ya que lo que genera es un vector de números para ejes X e Y.

Por lo tanto la intencionalidad de este apartado de análisis se ha cumplido pero no puede decirse que se haya integrado un Documento General de Requisitos al completo.

8.2 Concepto preliminar de la interfaz de aprendizaje

La interfaz gráfica de aprendizaje es la única ventana nueva, las de la primera iteración siguen presentes e iguales. En la figura 8.1 se puede apreciar el *mockup* de la ventana entrenamientos previos a una partida que enfrente un jugador humano con un sistema con aprendizaje.

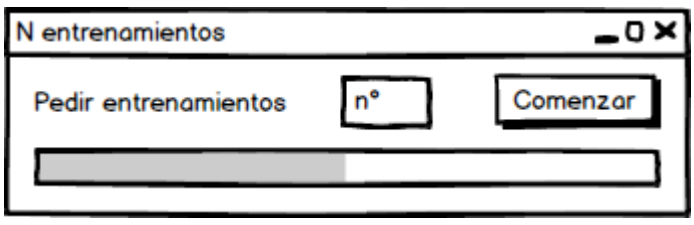

Figura 8.1: PopUp de entrenamientos

8.3 Diseño

Ahora trataremos todo lo referido al diseño del aprendizaje por refuerzo, desde casos de uso hasta el diagrama inicial de clases, qué esta vez será agregado al ya existente más extenso.

8.3.1 Casos de uso

Seguimos teniendo solo un participante, el jugador, con las características que se detalladas. Este jugador podrá realizar cinco acciones, dos de las cuales pertenecen a la primera iteración.

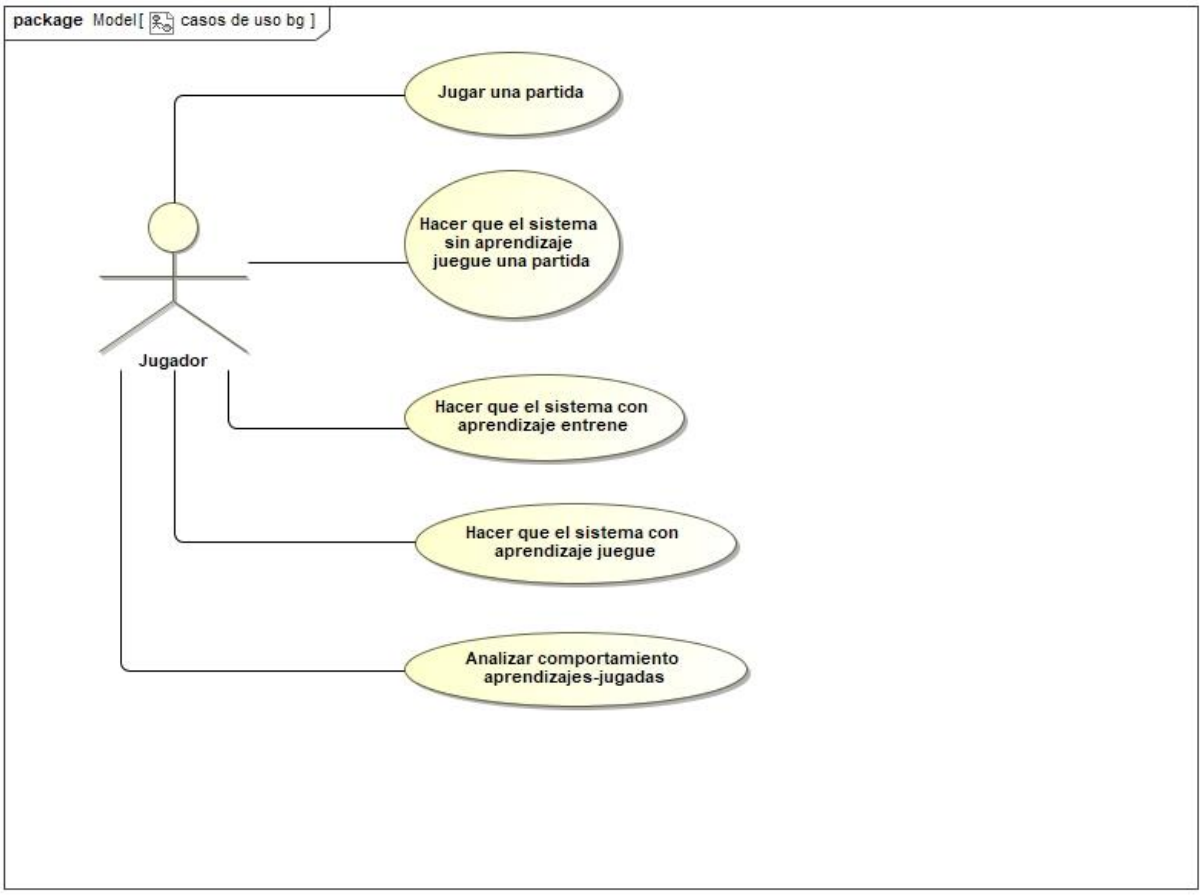

Figura 8.2: Casos de uso

En la figura 8.2 el jugador puede hacer las mismas dos jugadas que podría hacer antes, como son jugar una partida o hacer que el sistema sin aprendizaje juegue una partida. Las siguientes tres acciones son las que vamos a detallar a continuación.

La primera nueva acción de hacer que el sistema con aprendizaje entrene, se detalla en la figura 8.3. El jugador puede hacer que el sistema con aprendizaje entrene y decidir si desea ver la progresión del entrenamiento.

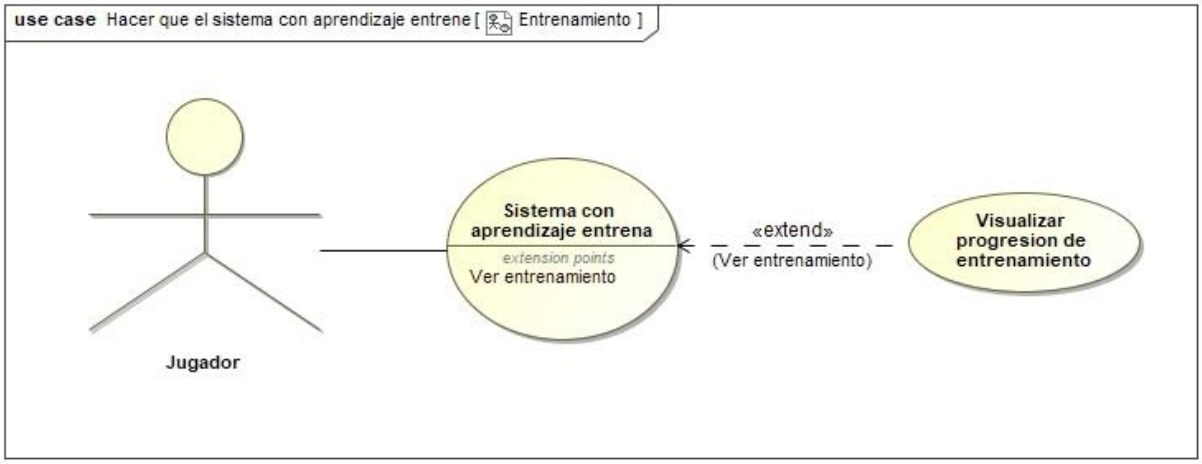

Figura 8.3: Casos de uso de entrenar sistema con aprendizaje

La siguiente acción nueva se puede ver en la figura 8.4, esta consiste en que el sistema con aprendizaje pueda jugar, así como ver cómo va jugando.

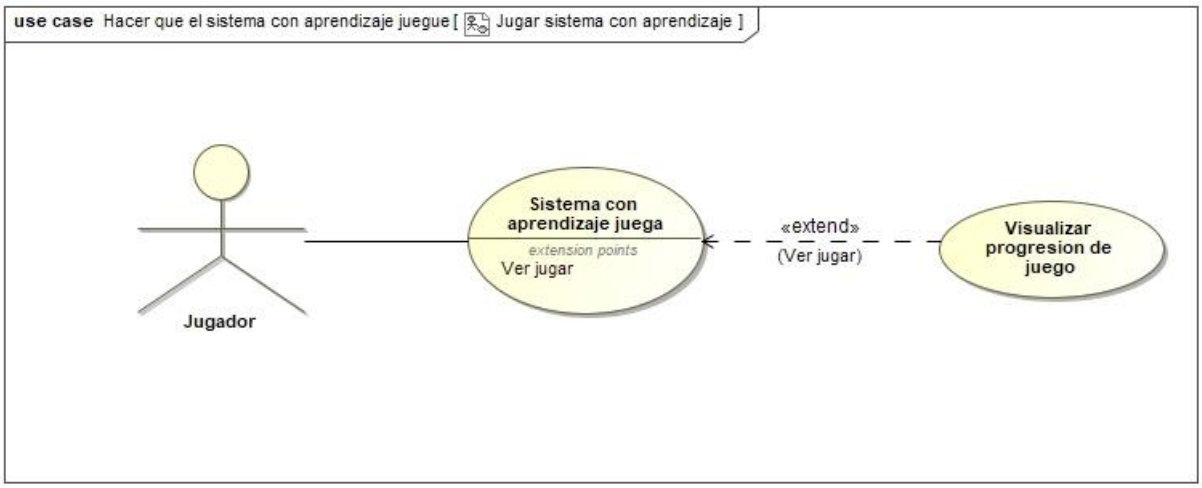

Figura 8.4: Casos de uso de jugar sistema con aprendizaje

La última nueva acción que el jugador puede llevar a cabo consiste en hacer que el sistema con aprendizaje entrene y juegue, lo cual podrá hacer que se generen una serie de estadísticas de aprendizaje y de victorias y derrotas. Se puede ver en detalle este caso de uso en la figura 8.5.

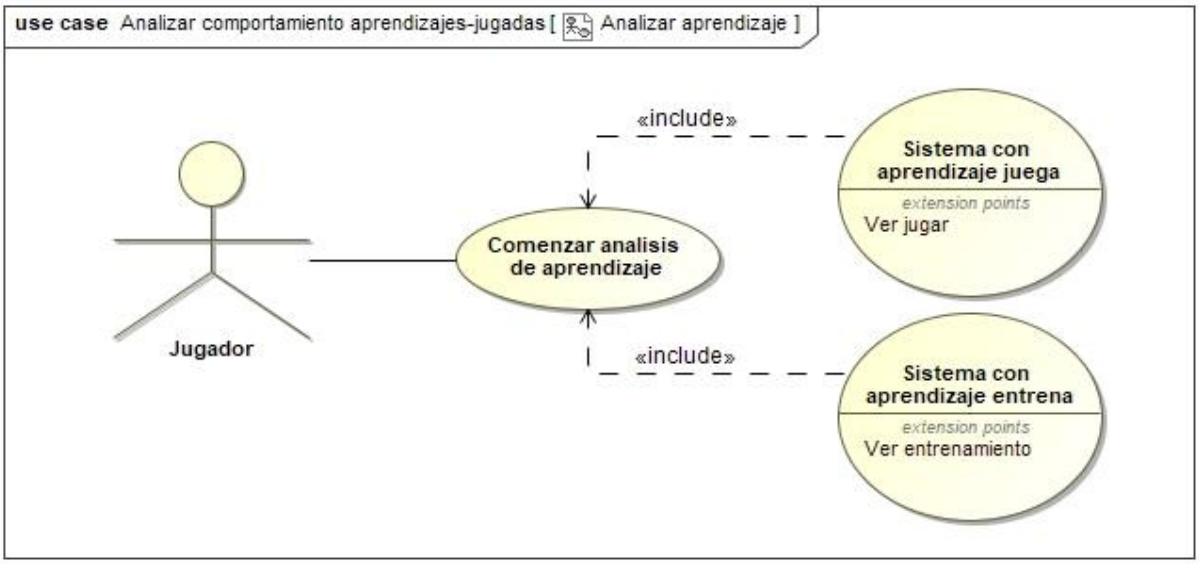

Figura 8.5: Casos de uso de analizar aprendizaje

Centrándonos en estos nuevos casos de uso, asociado al caso de uso de la figura 8.3 y 8.4 podemos ver en general el funcionamiento de como entrena y/o aprende en la figura 8.6 y 8.7, son el mismo diagrama de secuencia pero está dividido en dos partes para una mejor visualización.

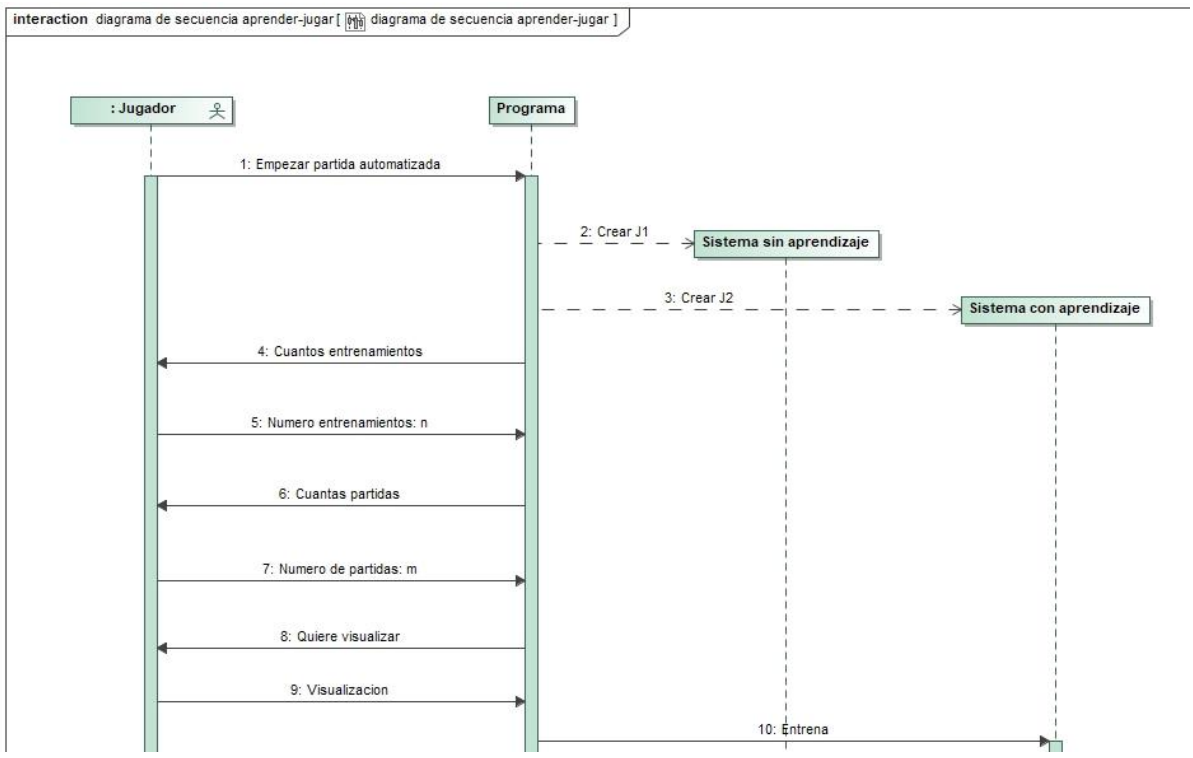

Figura 8.6: Diagrama de secuencia de aprender-jugar (1)

En esta primera parte del diagrama, figura 8.6, se puede ver como el jugador empieza una partida. Tras esto el programa crea los jugadores con y sin aprendizaje, el programa le pregunta al jugador cuantos entrenamientos y partidas quiere. Por último, le pregunta si quiere visualizar o no. A todas estas preguntas el jugador deberá responder con unos parámetros. Justo después de esto le ordena al sistema con aprendizaje que empiece a entrenar.

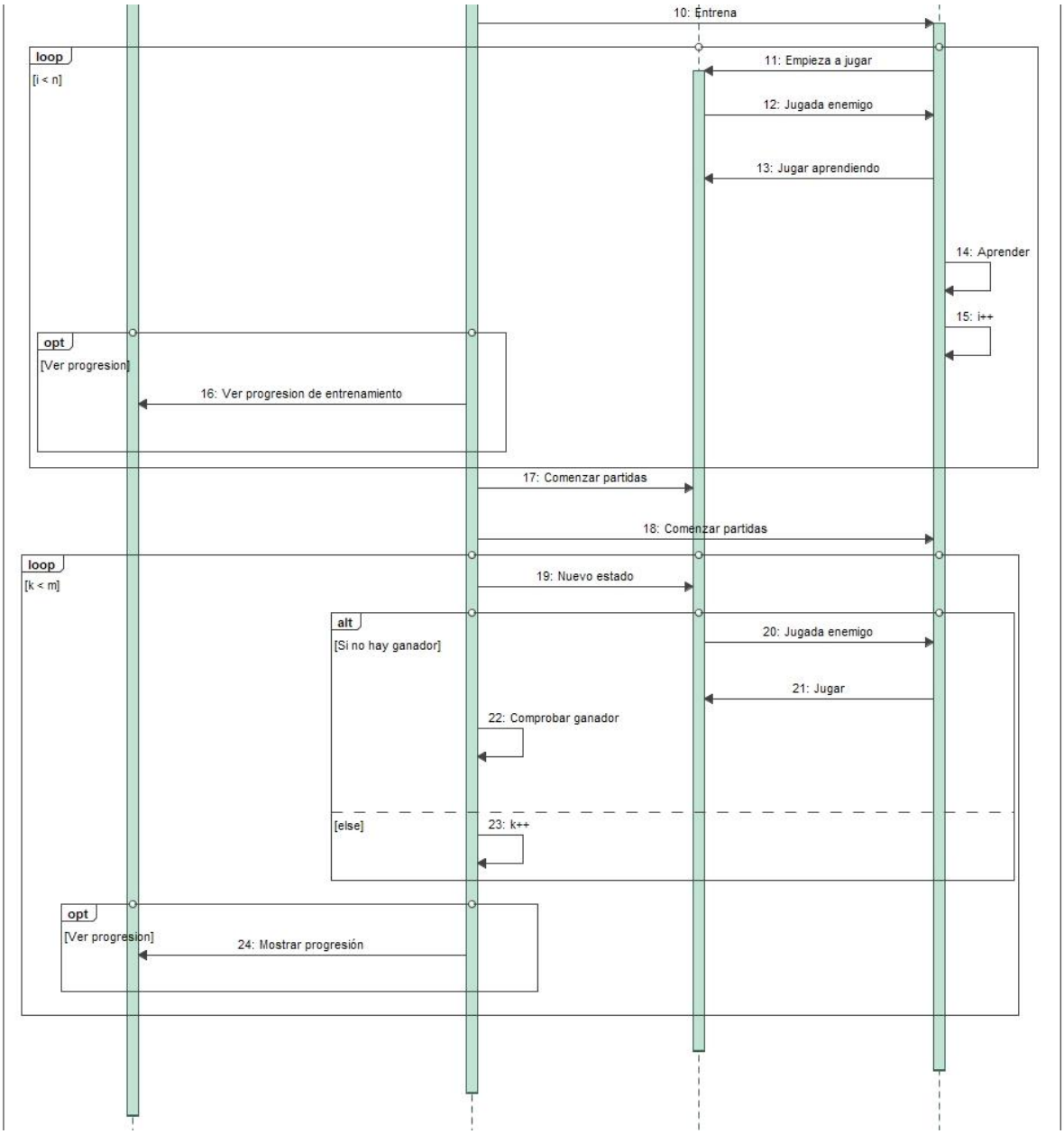

Figura 8.7: Diagrama de secuencia de aprender-jugar (2)

Para que sea más intuitivo el sistema sin aprendizaje lo llamaremos jugador aleatorio, a veces también se menciona como enemigo, y el sistema con aprendizaje lo llamaremos jugador con aprendizaje.

En La figura 8.7 el programa le ordena al jugador con aprendizaje que empiece a jugar, tras esto el jugador aleatorio hace una jugada y el jugador con aprendizaje realiza la suya y aprende. Se incrementa el número de jugadas y se muestra la progresión, si así lo desea el jugador, una vez se han entrenado todas las veces que se le han indicado entonces comienzan las partidas.

Al jugador aleatorio, como es el primero, se le tiene que pasar un estado nuevo en cada iteración del bucle. Tras esto el jugador aleatorio hace una jugada y el jugador con refuerzo realiza su jugada, en consecuencia se comprueba si hay ganador para incrementar la variable k de numero de partidas. Esto se repite las veces indicadas con la sentencia de control del bucle y se muestra por pantalla la progresión si el jugador lo quería.

El caso de uso de analizar aprendizaje (véase figura 8.5) se describe su funcionamiento en detalle en las figuras 8.8 y 8.9.

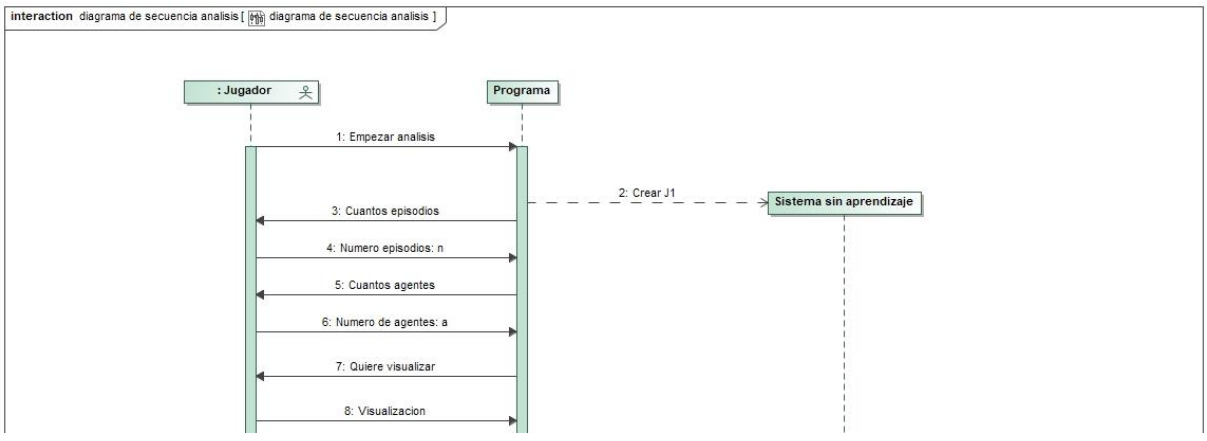

Figura 8.8: Diagrama de secuencia análisis de aprendizaje (1)

En la figura 8.8 el jugador comienza el análisis entonces el programa crea el sistema sin aprendizaje y el sistema con aprendizaje, es decir, el jugador aleatorio y el jugador con aprendizaje.

Entonces el programa le pregunta el jugador cuantos episodios quiere, un episodio es una secuencia aprendizaje primero y jugar después, acto seguido le pregunta a cuantos agentes va a querer. El número mínimo de agentes es 1, qué significa que no se haría ninguna media, solo se usaría un simple agente para generar los datos. Los resultados del análisis si se escogen varios agentes se hace una media de sus resultados.

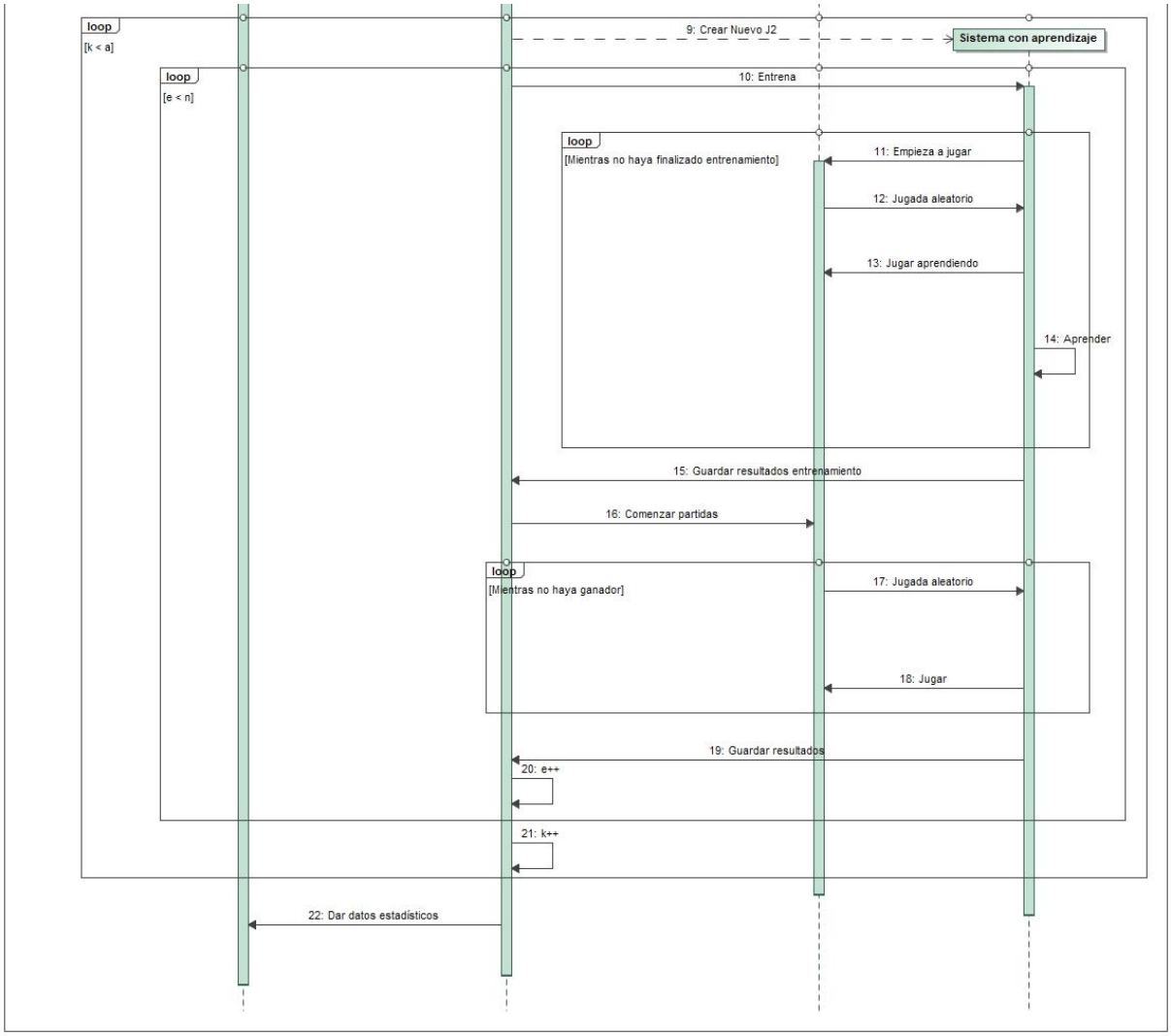

Figura 8.9: Diagrama de secuencia análisis de aprendizaje (2)

La figura 8.9 es la segunda parte del diagrama de la figura 8.8, en la 8.9 se ve un bucle externo que se ejecutará en función de la cantidad de agentes que se desee. El programa hará una media de ellos y lo devolverá como se puede ver al final en dar datos estadísticos. Por otra parte el primer bucle indica el número de episodios. Detalladamente un episodio es primer entrenar y guardar los datos de entrenamiento,

y comenzar una partida y guardar los resultados de la partida. De esta forma se puede saber cuál es la evaluación del jugador tras haber aprendido y las veces que ha ganado o perdido.

8.3.2 Modelo

En este apartado presentamos el modelo para la implementación de técnicas de aprendizaje por refuerzo. Así mismo también estará la implementación del primer iterable en el diagrama de clases.

La figura 8.10, que es el diagrama cubre los aspectos más importantes de esta segunda iteración. En ésta se ven nuevas clases como puedan ser el jugador entrenable o el jugador evaluador, qué son los nuevos jugadores. El controlador de PopUp y la vista de ese controlador, esta vista será anterior a la vista de jugar, en ella se decidirán cuantos entrenamientos tendrá el jugador entrenable. La clase que permite aprender contra aleatorio, es decir, que entrenen al jugador entrenable jugando contra el jugador aleatorio. La clase de análisis de aprendizaje genera las estadísticas gracias a que puede contener varias partidas de aprendizaje contra aleatorio.

Una de las clases más destacadas es el Evaluador, este permite la evaluación del estado de juego en cada momento. Usar en la técnica de aprendizaje mediante diferencias temporales que se explicó en el capítulo 3. Y a partir de la evaluación ser capaz de otorgar una recompensa.

Se ha agregado un nuevo método llamado codifica a la clase Backgammon, este método codifica el estado actual en una lista necesario para poder evaluar los estados.

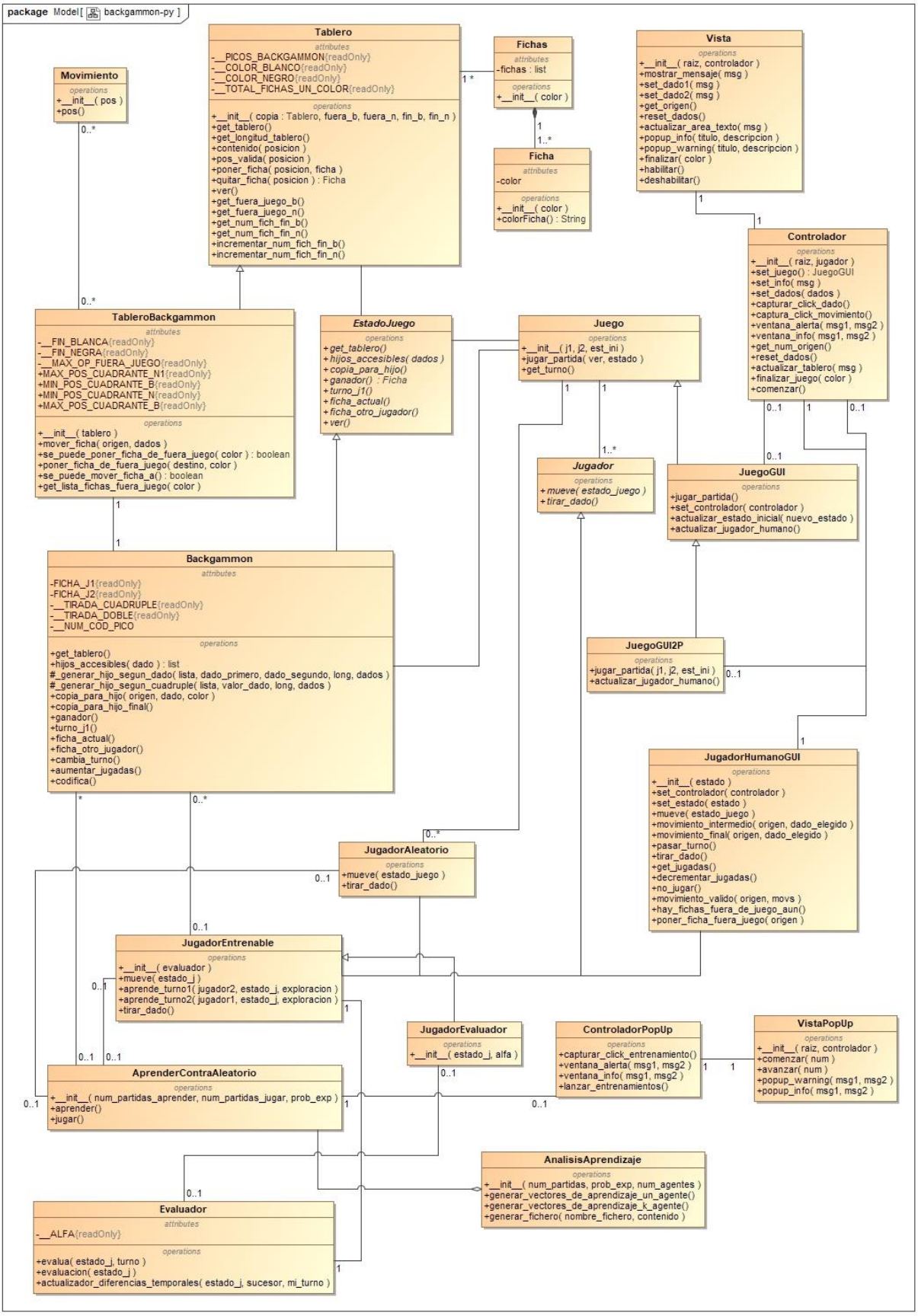

Figura 8.10: Diagrama de clases Backgammon-PY

9

Implementación Aprendizaje por refuerzo

En este capítulo se darán los detalles sobre la implementación de las técnicas de aprendizaje por refuerzo, más concretamente, sobre la implementación de la técnica de aprendizaje por diferencias temporales. Así mismo se verá en detalle los jugadores que entrenan, las partidas que se pueden generar con aprendizaje y jugada (episodio), y finalmente se podrá ver en detalle como se ha realizado el análisis de los datos de aprendizaje. Nuevamente vamos a dividir los apartados según los paquetes del proyecto que hayan sido modificados.

9.1 Backgammon

En el paquete de Backgammon sólo ha sufrido cambios el encargado de mantener el estado de juego, es decir, la clase Backgammon. En esta se ha agregado un método codifica y métodos privados para modularizar la lógica del negocio. También se ha agregado un método para poder mostrar la codificación por pantalla. Y por último hay un método en el que se genera una lista con los posibles lugares donde se puede poner

una ficha que se encuentre fuera de juego. En la figura 9.1 se puede observar el diagrama del paquete Backgammon.

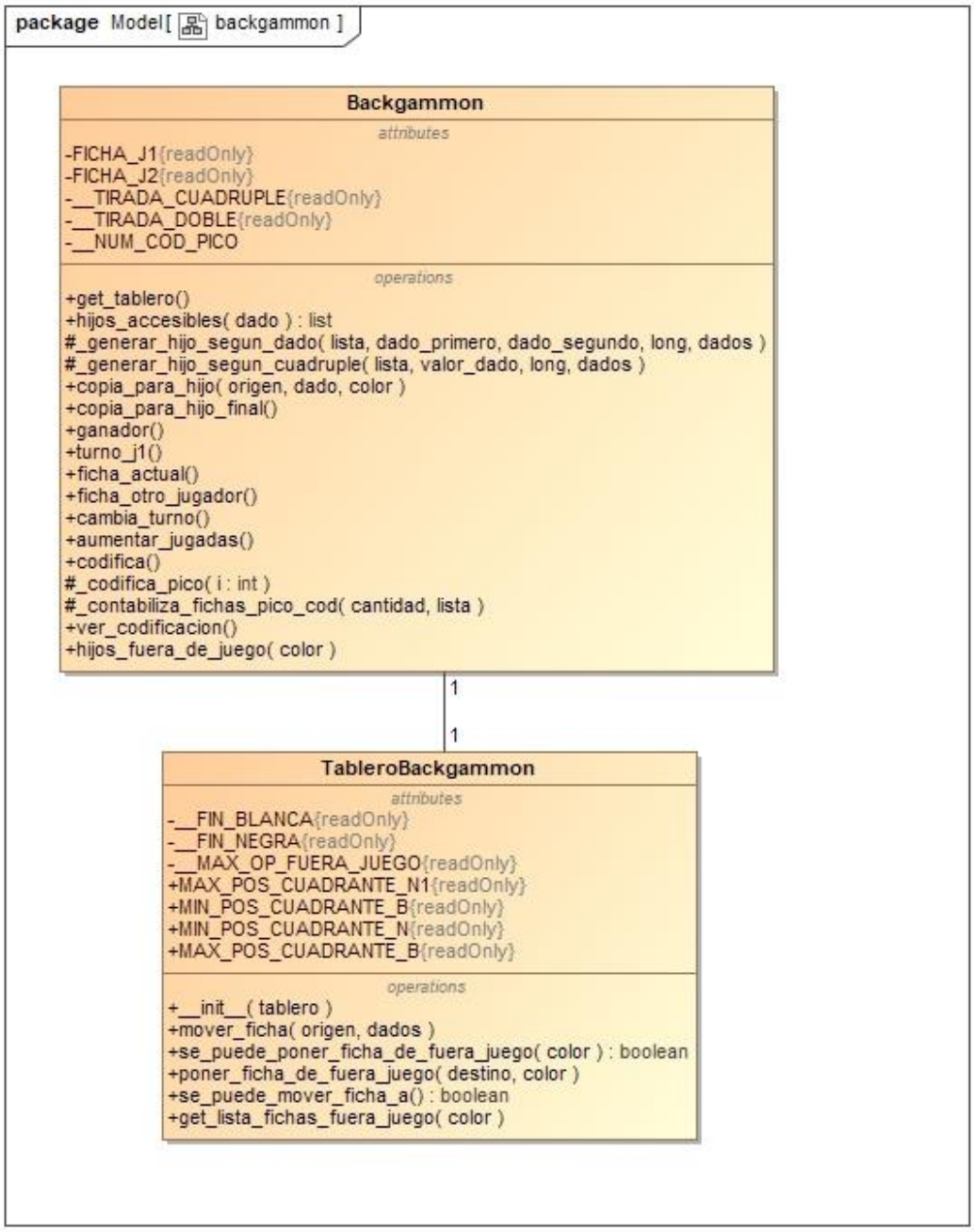

Figura 9.1: Diagrama de clases Backgammon

9.1.1 Backgammon

La clase Backgammon sigue funcionando de la misma manera que lo hacía antes, es decir, mantiene toda la información sobre le estado de la partida, solo que ahora además es capaz de codificar el tablero para poder ser usado en una función de evaluación. En la figura 9.2 se ve el funcionamiento del método codifica.

```
def codifica(self) -> list:
    cod = list()
    cod.append(1)
    cod.append(len(self.get_tablero().get_fuera_juego_b()))
    cod.append(len(self.get_tablero().get_fuera_juego_n()))
    cod.append(self.get_tablero().get_num_fich_fin_b())
    cod.append(self.get_tablero().get_num_fich_fin_n())
    for i in range(0,self.get_tablero().get_longitud_tablero()):
        cod += self._codifica_pico(i)
    return cod
```

Figura 9.2: Método codifica

El método codifica crea una lista que interpreta la situación del estado actual siguiendo las características que se describen a continuación:

- A la posición 0 se le asigna un término independiente a 1.
- A la posición 1 se le asigna el número de fichas golpeadas del primer jugador.
- A la posición 2 se le asigna se el número de fichas golpeadas del segundo jugador.
- A la posición 3 se le asigna se el número de fichas liberadas, es decir, que han finalizado, del segundo jugador.
- A la posición 4 se le asigna se el número de fichas liberadas del segundo jugador.
- El resto de posiciones de la lista toma de base las 24 posiciones del tablero y rellena cada posición con seis valores de la siguiente forma.
    - Un 1 si en la casilla hay una ficha del primer jugador sino 0.
    - Un 1 si en la casilla hay dos fichas del primer jugador sino 0.
    - Si en la casilla hay más de dos fichas del primer jugador, por ejemplo N fichas, el número que se pondrá será N-2.

    Y ahora lo equivalente pero sobre el segundo jugador.

    - Un 1 si en la casilla hay una ficha del segundo jugador sino 0.
    - Un 1 si en la casilla hay dos fichas del segundo jugador sino 0.

- Si en la casilla hay más de dos fichas del segundo jugador, por ejemplo N fichas, el número que se pondrá será N-2.

Para conseguir realizar la codificación modularizamos el problema de codificar en dos subproblemas, en la figura 9.3 se pueden observar los métodos para poder realizarla.

```
def _codifica_pico(self, i: int) -> list:
    res = list()
    fichas = self.get_tablero().contenido(i)
    if fichas is None:
        res = [0 for i in range(0, self.__NUM_COD_PICO)]
    else:
        cantidad = fichas.get_cantidad()
        if fichas.get_color() == self.FICHA_J1.colorFicha():
            self._contabiliza_fichas_pico_cod(cantidad, res)
            res += [0, 0, 0]
        elif fichas.get_color() == self.FICHA_J2.colorFicha():
            res += [0, 0, 0]
            self._contabiliza_fichas_pico_cod(cantidad, res)

    return res

def _contabiliza_fichas_pico_cod(self, cantidad: int, lista: list):
    if cantidad == 1:
        lista.append(1)
    else:
        lista.append(0)
    if cantidad == 2:
        lista.append(1)
    else:
        lista.append(0)
    if cantidad > 2:
        lista.append(cantidad - 2)
    else:
        lista.append(0)
```

Figura 9.3: Método protegido codifica pico y contabiliza fichas pico

El método codifica_pico retorna la codificación de un pico concreto qué se le haya especificado como argumento y que sigue las reglas descritas.

El método contabiliza_fichas_pico_cod es un selector que permite a partir de una cantidad de fichas y una lista agregar valores correspondientes a la codificación de un pico a la lista. Como la lista se pasa por referencia cualquier valor que se agregue se mantiene.

El método `def ver_codificacion(self)` permite ver la codificación en un formato que sea más entendible para el usuario. Por ejemplo, si se codifica el estado inicial del tablero y se llama a ese método el resultado sería el que se ve en la figura 9.4.

```
[1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0]-> Pico 1
[0, 0, 0, 0, 0, 0]-> Pico 2
[0, 0, 0, 0, 0, 0]-> Pico 3
[0, 0, 0, 0, 0, 0]-> Pico 4
[0, 0, 0, 0, 0, 0]-> Pico 5
[0, 0, 0, 0, 0, 3]-> Pico 6
[0, 0, 0, 0, 0, 0]-> Pico 7
[0, 0, 0, 0, 0, 1]-> Pico 8
[0, 0, 0, 0, 0, 0]-> Pico 9
[0, 0, 0, 0, 0, 0]-> Pico 10
[0, 0, 0, 0, 0, 0]-> Pico 11
[0, 0, 3, 0, 0, 0]-> Pico 12
[0, 0, 0, 0, 0, 3]-> Pico 13
[0, 0, 0, 0, 0, 0]-> Pico 14
[0, 0, 0, 0, 0, 0]-> Pico 15
[0, 0, 0, 0, 0, 0]-> Pico 16
[0, 0, 1, 0, 0, 0]-> Pico 17
[0, 0, 0, 0, 0, 0]-> Pico 18
[0, 0, 3, 0, 0, 0]-> Pico 19
[0, 0, 0, 0, 0, 0]-> Pico 20
[0, 0, 0, 0, 0, 0]-> Pico 21
[0, 0, 0, 0, 0, 0]-> Pico 22
[0, 0, 0, 0, 0, 0]-> Pico 23
[0, 0, 0, 0, 1, 0]-> Pico 24
```

Figura 9.4: Codificación del estado inicial

En la figura 9.5 podemos ver un método que genera una lista con todos los posibles movimientos correspondientes a colocar una ficha que esté fuera de juego en el tablero. Pero no la combinación de estos, es decir, solo colocar una de las fichas.

Funciona de la siguiente manera, se halla la posición mínima y máxima del cuadrante interno del jugador que corresponda, se crea una lista vacía, se recorre el tablero de la oposición en mínima y máxima, de las cuales se van creando copias del tablero con la nueva ficha puesta. Si fue posible poner la ficha, la agrega a la lista de hijos fuera de juego. Esa lista con las copias es la que se retorna.

```
def hijos_desde_fuera_juego(self, color):
    min = self.get_tablero().MIN_POS_CUADRANTE_B if color ==
         self.get_tablero().get_color_blanco() \
        else self.get_tablero().MIN_POS_CUADRANTE_N
    max = self.get_tablero().MAX_POS_CUADRANTE_B if color ==
         self.get_tablero().get_color_blanco() \
        else self.get_tablero().MAX_POS_CUADRANTE_N
    lista_est = list()
    for i in range(min, max):
        estado_nuevo = Backgammon(self.turno,
                                  TableroBackgammon(self.tablero), None,
                                  self.jugadas)
        mov = estado_nuevo.get_tablero().poner_ficha_de_fuera_juego(i, color)
        if mov is not None:
            lista_est.append(estado_nuevo)
    return lista_est
```

Figura 9.5: Método hijos fuera de juego

9.2 Jugadores

En este paquete hemos definido dos nuevos jugadores y una clase para realizar la evaluación. En la figura 9.6 se puede ver el diagrama de clases del paquete.

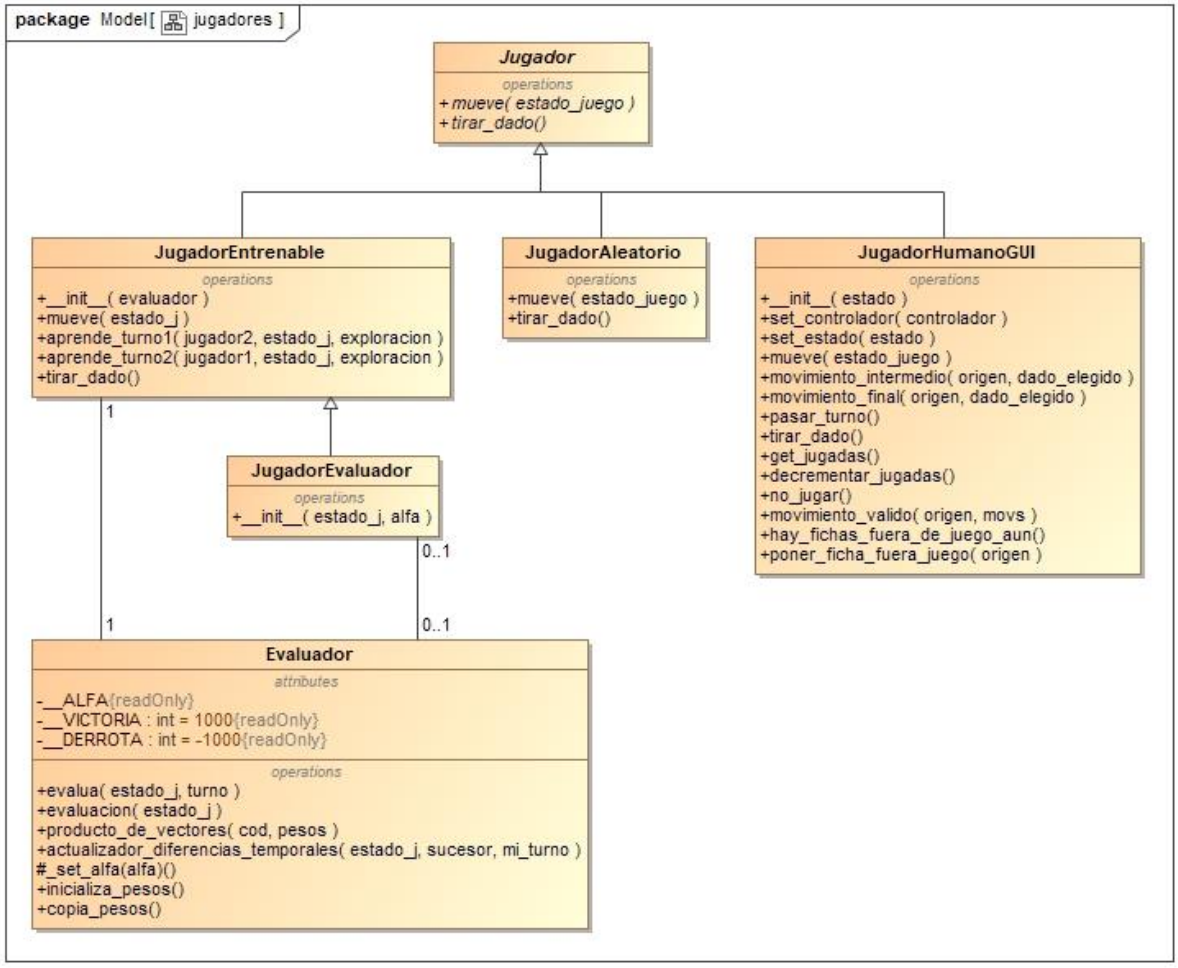

Figura 9.6: Diagrama de clases jugadores

Comencemos primero con el Evaluador debido a su importancia, ya que en el se aplica la técnica de aprendizaje por diferencias temporales. Siendo los jugadores nuevos quienes las usan.

9.2.1 Evaluador

El Evaluador es una clase que tiene implementado el método evaluar, entre otros. Este sirve para evaluar un estado determinado y retornar o bien la victoria, o bien la derrota, o bien un heurístico en funcion del estado actual, es decir, una llamada al método de evaluación. En la figura 9.7 podemos ver la clase Evaluador.

```
class Evaluador():
    __VICTORIA: int = 1000
    __DERROTA: int = -1000
    __ALFA = 0.1  # Tasa de aprendizaje

    def __init__(self, num_pesos, alfa=__ALFA):
        self.long = num_pesos
        self.pesos = list()
        self.inicializa_pesos()
        self.alfa = alfa
        print_("Constructor del evaluador con alfa {} y {}
                pesos".format(self.alfa, self.long))

    def evalua(self, estado_j: Backgammon, mi_turno: bool):
        ficha_ganadora = estado_j.ganador()
        es_mi_turno = mi_turno == estado_j.turno_j1()
        if ficha_ganadora == estado_j.ficha_actual() and es_mi_turno \
                or ficha_ganadora == estado_j.ficha_otro_jugador() and not
                                                        es_mi_turno:
            return self.__VICTORIA
        elif ficha_ganadora == estado_j.ficha_actual() and not es_mi_turno \
                or ficha_ganadora == estado_j.ficha_otro_jugador() and
                                                        es_mi_turno:
            return self.__DERROTA
        else:
            return self.evaluacion(estado_j)

    def evaluacion(self, estado_j: Backgammon):
        return self.producto_de_vectores(estado_j.codifica(), self.pesos)

    def producto_de_vectores(self, cod: list, pesos: list):
        valor = 0
        for i in range(0, len(pesos)):
            valor = valor + pesos[i] * cod[i]
        return valor
```

Figura 9.7: Clase Evaluador (1)

Para ser creado el Evaluador necesita saber la cantidad de pesos que habrá, realmente en el Backgammon es constante, pero hemos intentado generalizar un poco con el fin de poder probar que funcionase inicialmente con menos pesos. Necesita también un valor de Alfa, que por defecto será el valor de la variable privada y constante ALFA. Tras esto crea una lista e inicializa los pesos.

El método evalua coge un estado del tablero y el turno del jugador. Con ello retorna o bien la recompensa si hubo victoria, el castigo si hubo derrota o si el juego no ha acabado llama al método evaluación para obtener el valor de la función lineal para el estado en el que se encuentra.

Por su parte el método de evaluación solo necesita el estado de la partida actual, lo que hace es codificar el estado y multiplicarlo por lista de pesos usando el producto escalar, ya que ambos son vectores, tanto la codificación como la lista de pesos y obtener un valor de evaluación. Este valor solo debería oscilar, teniendo en cuenta las recompensas y castigos, entre 1000 y -1000.

```
def actualizador_diferencias_temporales(self, estado_j: Backgammon,
                                        sucesor:Backgammon, mi_turno: bool):
    cod = estado_j.codifica()
    delta = self.alfa * (self.evalua(sucesor, mi_turno) -
                                                    self.evaluacion(estado_j))
    for i in range(0, self.long):
        self.pesos[i] = self.pesos[i] + delta * cod[i]
```

Figura 9.8: Clase Evaluador (2)

En la figura 9.8 se puede ver en qué consiste el método de diferencias temporales (véase capítulo 3), este como parámetros de entrada necesita el estado actual del juego, que se actualizara, el sucesor alcanzado a partir del estado que se va actualizar y el turno del jugador que va aprender, siendo verdadero si es el primer jugador y falso si es el segundo jugador.

Este método actualiza los pesos, es decir, en funcion de la evaluación actual y el resultado de evaluar el estado sucesor, aprende. Ello se consigue usando el método del subgradiente estocástico TD(0). El sucesor podría ser un estado final, en cuyo caso

necesitamos conocer su recompensa final, es decir, si hay victoria o derrota. Si la partida no ha acabado se llama a evaluación.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Figura 9.9: TD(0). [16]

Con el fin de relacionar conceptos, concretamente con la fórmula de la figura 9.9 descrita en el capítulo 3, la formula anterior se representa en el programa con el método `actualizador_diferencias_temporales(self, estado_j: Backgammon, sucesor: Backgammon, mi_turno: bool)`, el cual podemos describir como sigue. Tenemos que la recompensa esperada del estado futuro R_{t+1} va a ser el resultado de `evalua(self, estado_j: Backgammon, mi_turno: bool)` si hubiera victoria o derrota, en ese caso se recompensa en gran medida. Si no se ha ganado o perdido, ese mismo método retorna el valor de evaluación, del método `evaluacion(self, estado_j: Backgammon)`, con el estado al que se quiere ir $\gamma V(S_{t+1})$ con $\gamma = 1$. Por último se resta el valor de la evaluación del estado actual $-V(S_t)$. A ese resultado se le multiplica la constante del coeficiente de aprendizaje, es decir, la rapidez con la que aprende. Es más interesantes tener valores bajos, como $\alpha = 0.001$, para evitar que los valores nunca converjan. A valor de α más bajo se necesitan más entrenamientos pero el aprendizaje se hace más estable. Por último el resultado de todas esas operaciones $\alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ se suman al valor de la evaluación actual $V(S_t)$ llamada pesos en la figura 9.8, y se actualiza el valor dichos pesos en consecuencia.

Lo que se pretende hacer es evaluando el estado actual y el estado al que se quiere ir, que sea siempre mejor a donde va. Bueno, aquí no se decide, solo se evalúa, será el propio jugador con aprendizaje quien elija la mejor evaluación, la más favorable para él y por tanto peor para el enemigo.

9.2.2 JugadorEntrenable

Esta clase hereda de jugador, este jugador elige la jugada que cree mejor en cada momento, es decir, la que tenga mayor valor su en función de la evaluación. En la figura 9.10 se puede ver como se crea el jugador entrenable y decide el movimiento.

```
class JugadorEntrenable(Jugador):
    def __init__(self, evaluador: Evaluador):
        self.evaluador = evaluador

    def mueve(self, estado_j: Backgammon) -> Backgammon:
        print_("Mueve inteligente, mov
                {}".format(estado_j.movimiento_ultimo))

        dados = self.tirar_dado()
        print_dados(dados)
        turno = estado_j.turno_j1()
        optimo = None
        valor_optimo = float("-inf")
        while estado_j.get_tablero().se_puede_poner_ficha_de_fuera_juego(
                                    estado_j.ficha_actual().colorFicha()):
            optimo_temp = None
            valor_optimo_temp = float("-inf")
            estados_intermedios = estado_j.hijos_desde_fuera_juego(
                                    estado_j.ficha_actual().colorFicha())
            shuffle(estados_intermedios)
            for est in estados_intermedios:
                nuevo_valor = self.evaluador.evalua(est, turno)

                if nuevo_valor > valor_optimo_temp or optimo_temp is None:
                    valor_optimo_temp = nuevo_valor
                    optimo_temp = est
            estado_j = optimo_temp
        estados = list()

        if not estado_j.get_tablero().get_lista_fichas_fuera_juego(
                                    estado_j.ficha_actual().colorFicha()):
            estados = estado_j.hijos_accesibles(dados)

        if estados:
            shuffle(estados)
            for est in estados:
                nuevo_valor = self.evaluador.evalua(est, turno)
                if nuevo_valor > valor_optimo or optimo is None:
                    valor_optimo = nuevo_valor
                    optimo = est
            return optimo
        else:
            print_("El jugador {} no puede moverse, se cambia
                 turno\n\n\n".format(estado_j.ficha_actual().colorFicha()))
            estado_j.cambia_turno()
            return estado_j
```

Figura 9.10: Clase JugadorEntrenable (1)

Como se ve en la figura 9.10, el constructor solo requiere de un evaluador, uno de los componentes más importantes y en los que se basa el aprendizaje de este jugador.

Por su parte mueve, elige de entre todos los hijos posibles, para un estado de juego dado, el hijo que reciba una mayor evaluación. Además la lista se baraja para intentar evitar un comportamiento determinista.

Este método primero va intentando poner de la manera más adecuada las fichas que estuvieran fuera de juego, es decir, poner las que le den una mayor evaluación. Tras esto tira los dados y comienza a evaluar y buscar la mejor jugada de entre todas las posibles. Si no pudiera mover, igual que con otros tipos de jugadores, se pasa turno.

Por su parte el primer método de la figura 9.11, sirve como primer paso para entrenar al jugador mediante una partida con el jugador2 a partir del estado de juego pasado como parámetro. Donde jugador2 es el jugador que juega en el primer turno, estado_j es el estado inicial del juego y exploración es la probabilidad de exploración. En cualquiera de los casos la partida, o el aprendizaje, continua en aprende_turno2 con el turno que corresponda.

El segundo método de la figura 9.11 entrena al jugador mediante una partida con el jugador1 a partir del estado de juego. Recibe las mismas entradas que aprender en turno1. Se ha dividido de esta manera para evitar duplicar código.

Primero hace que mueva el rival, ya que el jugador con aprendizaje juega en el turno después del rival, tiene una probabilidad de exploración para evitar los mínimos locales que lo que hace es realizar una jugada aleatoriamente, solo cuando no hay exploración el jugador con aprendizaje mueve, evaluando previamente y escogiendo la mejor jugada, y acto seguido llama al evaluador por diferencias temporales para actualizar los pesos tras su jugada.

```
def aprende_turno1(self, jugador2: Jugador, estado_j: Backgammon,
                         exploracion: float):
    nuevo_estado = None
 if num_aleatorio_flotante() < exploracion:
        estados = estado_j.hijos_accesibles(self.tirar_dado())
        if estados and \
            not estado_j.get_tablero().get_lista_fichas_fuera_juego(
                 estado_j.ficha_actual().colorFicha()):
            nuevo_estado = estados[num_aleatorio(len(estados) - 1)]
        else:
            nuevo_estado = estado_j.cambia_turno()
    else:
        nuevo_estado = self.mueve(estado_j)

    self.aprende_turno2(jugador2, nuevo_estado, exploracion, True)


def aprende_turno2(self, jugador1: Jugador, estado_j: Backgammon,
                         exploracion: float, mi_turno=False):
    nuevo_estado = jugador1.mueve(estado_j)
    if nuevo_estado.ganador() is not None:
        self.evaluador.actualizador_diferencias_temporales(
                                 estado_j, nuevo_estado, mi_turno)
    else:
        nuevo_estado2 = None
        explora = num_aleatorio_flotante() < exploracion
        if explora:
            estados = nuevo_estado.hijos_accesibles(self.tirar_dado())
            if estados and not
                 nuevo_estado.get_tablero().get_lista_fichas_fuera_juego(
                    nuevo_estado.ficha_actual().colorFicha()):
                num = num_aleatorio(len(estados) - 1)
                nuevo_estado2 = estados[num]
            else:
                nuevo_estado.cambia_turno()
                nuevo_estado2 = nuevo_estado
        else:
            nuevo_estado2 = self.mueve(nuevo_estado)
        if not explora:
            self.evaluador.actualizador_diferencias_temporales(estado_j,
                                                 nuevo_estado2, mi_turno)

        if nuevo_estado2.ganador() is None:
            self.aprende_turno2(jugador1, nuevo_estado2, exploracion,
                                                                 mi_turno)
```

Figura 9.11: Clase JugadorEntrenable (2)

9.2.3 JugadorEvaluador

Cómo se puede ver en la figura 9.12 este jugador simplemente es una "lanzadera" el cual a partir de un estado inicial y un posible valor de alfa, no obligatorio, crea un evaluador con una codificación inicial del estado. Y pudiendo pasarle un valor concreto de Alfa. Este jugador hereda de JugadorEntrenable.

```
class JugadorEvaluador(JugadorEntrenable):
    def __init__(self, estado_j: Backgammon, alfa: float = None):
        if alfa is None:
            super().__init__(Evaluador(len(estado_j.codifica())))
        else:
            super().__init__(Evaluador(len(estado_j.codifica()), alfa))
```

Figura 9.12: Clase JugadorEvaluador

9.3 Análisis

Este paquete contiene todo lo necesario para realizar aprendizajes de manera automatizada y también generar estadísticas. Su diagrama de clases se aprecia en la figura 9.13.

Figura 9.13: Diagrama de clases paquete análisis

9.3.1 AprenderContraAleatorio

Esta clase sirve para realizar un entrenamiento del jugador entrenable contra el jugador aleatorio un numero de veces determinado en el constructor. Esta clase se puede ver en la figura 9.14.

```
class AprenderContraAleatorio:
    ALFA = 0.001

    def __init__(self, num_partidas_aprender, num_partidas_jugar,
                 probabilidad_exploracion):
        self.num_partidas_aprender = num_partidas_aprender
        self.num_partidas_jugar = num_partidas_jugar
        self.estado = Backgammon()
        self.jugador_aleatorio = JugadorAleatorio()
        self.jugador_entrenable = JugadorEvaluador(self.estado, self.ALFA)
        self.probabilidad_exploracion = probabilidad_exploracion
        self.partidas_ganadas_j1 = 0
        self.partidas_ganadas_j2 = 0

    def aprender(self, ver=True):
        if self.num_partidas_aprender > 0:

            for i in range(self.num_partidas_aprender):
                self.estado = Backgammon()
                print_muestra_avance(i,
                 self.jugador_entrenable.evaluador.evaluacion(self.estado),
                                     "entrenamiento Nº:V(inicial)", ver)

self.jugador_entrenable.aprende_turno2(self.jugador_aleatorio, self.estado,
                                       self.probabilidad_exploracion)

    def jugar(self, ver=True):
        for i in range(self.num_partidas_jugar):
            print_muestra_avance(num=i, mensaje="partida Nº", observar=ver)
            self.estado = Backgammon()
            juego = Juego(self.jugador_aleatorio, self.jugador_entrenable,
                          self.estado)
            res = juego.jugar_partida(False)
            if res == 1:
                self.partidas_ganadas_j1 += 1
            elif res == -1:
                self.partidas_ganadas_j2 += 1

        return self.partidas_ganadas_j1, self.partidas_ganadas_j2

    def get_evaluacion_inicial(self):
        return self.jugador_entrenable.evaluador.evaluacion(self.estado)

    def set_num_partidas_aprender(self, num: int):
        self.num_partidas_aprender = num

    def set_num_partidas_jugar(self, num: int):
        self.num_partidas_jugar = num

    def set_jugador_entren(self, jugador_nuevo):
        self.jugador_entrenable = jugador_nuevo
```

Figura 9.14: Clase aprender contra aleatorio

El constructor sirve para definir el número de partidas destinadas a aprender y a jugar, de las que se jueguen se extraerá una estadística a tiempo real.

El método aprender ejecuta el número de aprendizajes previamente definido en el constructor y muestra el avance por consola.

El método jugar Ejecuta un numero de partidas previamente definido en el constructor y retorna el conjunto de partidas ganadas y perdidas en una tupla. De esta manera se pueden sacar estadísticas con proporción de victorias y derrotas.

9.3.2 AnalisisAprendizaje

Esta clase se puede observar en la figura 9.15, esta permite generar hasta cuatro scripts para Matlab con el resultado del analisis de poner el jugador entrenable a jugar un numero de episodios determinado.

El método de generar vectores de aprendizaje de un agente genera dos ficheros los cuales contienen dos vectores (x,y) para poder ser representados gráficamente correspondientes uno a la evaluación del estado inicial del tablero (y) para cada aprendizaje de cada episodio y otro correspondiente al número de partidas (x). Esto sirve para uno de los gráficos de linea, el de evaluación, y el otro son los vectores para el porcentaje de victorias (y) y otro correspondiente al porcentaje de derrotas(y2). En cuyo grafico de líneas se hará uso también del número de partidas (x).

Gracias a lo cual se podrá ver un gráfico de líneas en Matlab para ver el crecimiento y convergencia de la evaluación asi como un gráfico de % de victorias y derrotas.

Por su parte el método de generar vectores de k agentes, que se puede ver en la figura 9.16, realiza lo mismo que el anterior y además para cada episodio realiza la media de los k agentes.

El método de construir y generar fichero, de la figura 9.17, el primer método, el de construir pasa a cadena de caracteres los datos de los vectores, llama al comandos matlab según corresponda y genera el fichero llamando al método correspondiente.

```
class AnalisisAprendizaje:
    NOM_FICH_VAL_EPISODIOS = "ev_ep.m"
    NOM_FICH_VAL_EP_MED = "ev_ep_k_agentes.m"
    NOM_FICH_VIC_DERR = "victorias_derrotas.m"
    NOM_FICH_VIC_DERR_MED = "victorias_derrotas_k_agentes.m"
    VAR_X = "X"
    VAR_Y = "Y"
    VAR_Y2 = "Y2"
    COMANDO_MATLAB = "plot({},{});".format(VAR_X, VAR_Y)
    ESCRITURA = "w"

    def __init__(self, num_partidas, prob_exp, num_agentes=1):
        self.num_partidas = num_partidas
        self.prob_exp = prob_exp
        self.num_agentes = num_agentes
        self.partida_de_referencia = AprenderContraAleatorio(1, 1,
                                                    self.prob_exp)
        self.lista_de_partidas = [AprenderContraAleatorio(1, 1,
                                                       self.prob_exp) for i in
                                                    range(self.num_agentes)]
        self.rep_vector_ep =
                self._inicializar_vector_episodios(self.num_partidas+1)

    def generar_vectores_de_aprendizaje_un_agente(self):
        rep_vector_eval_ref = "{} = ".format(self.VAR_Y)
        vector_eval_ref = list()
        rep_vector_vict = "{} = ".format(self.VAR_Y)
        vector_vict = list()
        rep_vector_der = "{} = ".format(self.VAR_Y2)
        vector_der = list()

        for i in range(self.num_partidas):
            evaluacion = self.partida_de_referencia.get_evaluacion_inicial()
            vic_der = self.partida_de_referencia.jugar(False)
            self.partida_de_referencia.aprender(False)
            vector_eval_ref.append(evaluacion)

            vector_vict.append(vic_der[1] * 100 / (i + 1))
            vector_der.append((i + 1 - vic_der[1]) * 100 / (i + 1))

        rep_vector_eval_ref += str(vector_eval_ref)
        rep_vector_vict += str(vector_vict)
        rep_vector_der += str(vector_der)

        cf1 = self.construir_contenido_fichero(self.rep_vector_ep,
                                                    rep_vector_eval_ref)
        cf2 = self.construir_contenido_fichero(self.rep_vector_ep,
                                        rep_vector_vict, rep_vector_der)

        self.generar_fichero(self.NOM_FICH_VAL_EPISODIOS, cf1)
        self.generar_fichero(self.NOM_FICH_VIC_DERR, cf2)
```

Figura 9.15: Clase AnalisisAprendizaje

```
def generar_vectores_aprendizaje_k_agentes(self):
    rep_vector_eval_ref = "{} = ".format(self.VAR_Y)
    vector_eval_ref = list()
    rep_vector_vict = "{} = ".format(self.VAR_Y)
    vector_vict = list()
    rep_vector_der = "{} = ".format(self.VAR_Y2)
    vector_der = list()

    for i in range(self.num_partidas):
        evaluacion = 0;
        vic_der_acumulado = (0, 0)
        for k in range(self.num_agentes):
            partida = self.lista_de_partidas[k]
            evaluacion += partida.get_evaluacion_inicial()
            vic_der = partida.jugar(False)
            vic_der_acumulado = (vic_der_acumulado[0] + vic_der[0],
                                 vic_der_acumulado[1] + vic_der[1])
            partida.aprender(False)

        # Media de evaluaciones
        evaluacion = evaluacion/self.num_agentes
        vic_der_acumulado = (vic_der_acumulado[0]/self.num_agentes,
                                 vic_der_acumulado[1]/self.num_agentes)

        vector_eval_ref.append(evaluacion)
        vector_vict.append(vic_der_acumulado[1] * 100 / (i + 1))
        vector_der.append(vic_der_acumulado[0] * 100 / (i + 1))

    rep_vector_eval_ref += str(vector_eval_ref)
    rep_vector_vict += str(vector_vict)
    rep_vector_der += str(vector_der)

    cf1 = self.construir_contenido_fichero(self.rep_vector_ep,
                                           rep_vector_eval_ref)
    cf2 = self.construir_contenido_fichero(self.rep_vector_ep,
                                           rep_vector_vict, rep_vector_der)

    self.generar_fichero(self.NOM_FICH_VAL_EP_MED, cf1)
    self.generar_fichero(self.NOM_FICH_VIC_DERR_MED, cf2)

def _inicializar_vector_episodios(self, num):
    rep_vector_ep = "{} = ".format(self.VAR_X)
    vect = [i for i in range(1, num)]
    rep_vector_ep += str(vect)
    return rep_vector_ep
```

Figura 9.16: Clase AnálisisAprendizaje

Por su parte en la figura 9.16 el método de inicializar vector de episodios genera una lista con valores de 1 al número de episodios.

```
def construir_contenido_fichero(self, x: str, y: str, y2: str = None):
    """Construye el contenido del fichero y lo retorna"""
    res = x + ";\n"
    res += y + ";\n"
    if y2 is not None:
        res += y2 + ";\n"
        res += self.comandos_matlab_dos_vectores()
    else:
        res += self.comandos_matlab_un_vector()

    return res

def generar_fichero(self, nombre_fichero, contenido: str):
    try:
        fichero = io.open(nombre_fichero, self.ESCRITURA)
        fichero.write(contenido)
        fichero.close()
    except:
        print_error("ERROR - En la generación del fichero")

def comandos_matlab_dos_vectores(self):
    res = "plot({},{},\'b\',{},{},\'r\',\'linestyle\',\'-.\'
);\n".format(self.VAR_X, self.VAR_Y, self.VAR_X, self.VAR_Y2)
    res += "ylim([0 100]);\n"
    res += "title(\'Victorias (azul) y derrotas (rojo)\');\n"
    res += "xlabel(\'Numero de episodios\');\n"
    res += "ylabel(\'Porcentaje %\');"
    return res

def comandos_matlab_un_vector(self):
    res = "plot({},{});\n".format(self.VAR_X, self.VAR_Y,)
    res += "title(\'Crecimiento del aprendizaje\');\n"
    res += "xlabel(\'Numero de episodios\');\n"
    res += "ylabel(\'Evaluacion inicial del tablero\');"
    return res
```

Figura 9.17: Clase AnalisisAprendizaje

9.4 Controlador y Vista

Dado que el nuevo controlador solo se va a encargar de entrenar al jugador entrenable, no existe conexión directa entre los dos pares controlador-vista.

En la figura 9.18 se puede ver el diagrama de clases del controlador y la vista nuevos e independientes a los anteriores.

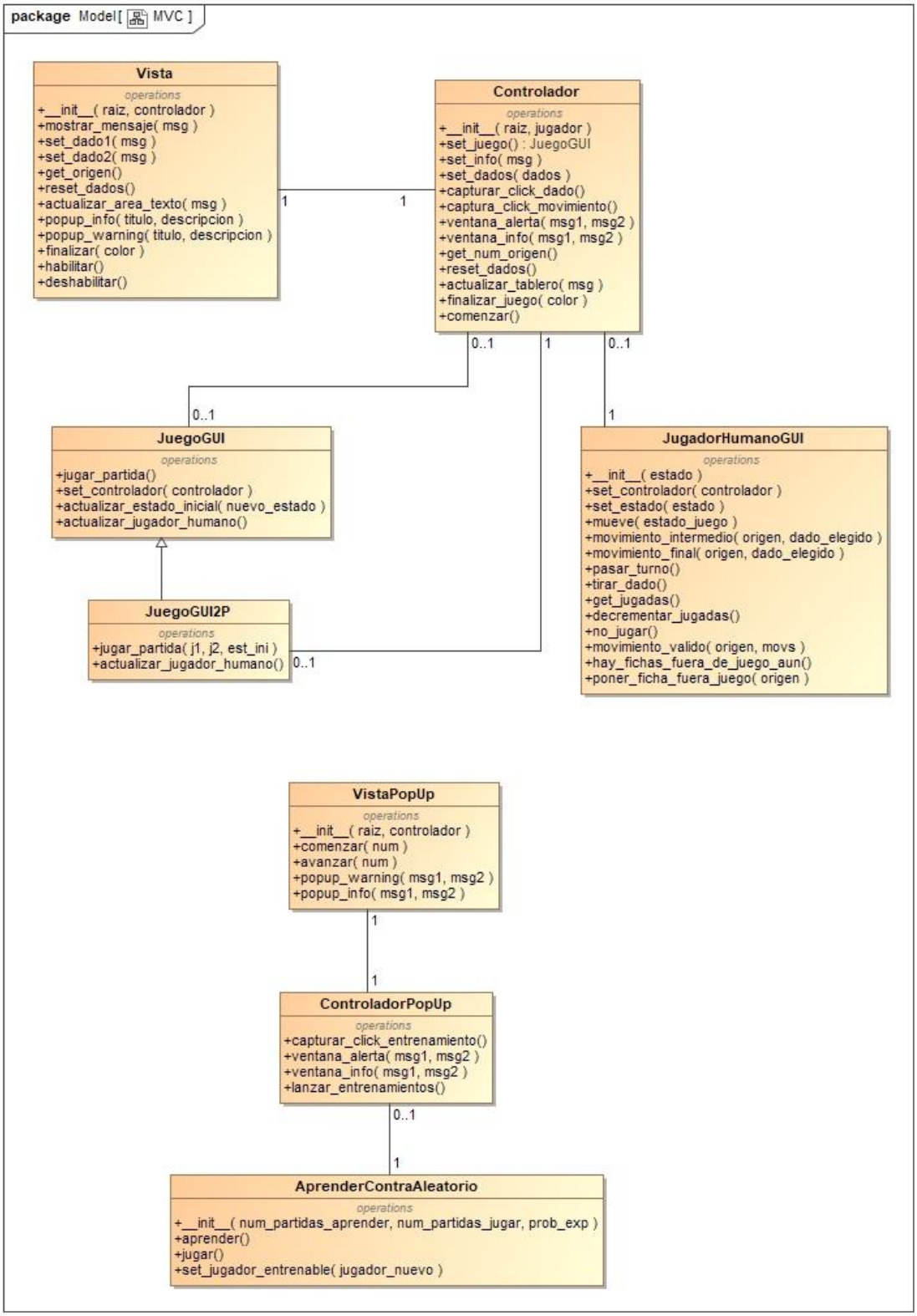

Figura 9.18: Diagrama de clases MVC

9.4.1 ControladorPopUp y VistaPopUp

Este controlador hace de intermediario entre la vista y la clase de aprendizaje (MVC). Esto sirve para aprender contra aleatorio, gracias a ello tras un número

determinado de partidas qué el jugador pasa por la entrada de la vista, el controlador se lo dirá a la clase de aprender y el jugador con aprendizaje comenzará a aprender. Mientras se actualiza una barra de progreso. Dado que no difiere la forma de programación de lo visto en el primer iterable, voy a centrarme en lo que fue la parte nueva que más compleja, es decir, la barra de progreso. Así como el hilo de vida de esta vista, ya que debe cerrarse cuando se acaba el aprendizaje.

```
def lanzar_entrenamientos(self):
    self.ventana_alerta("Paciencia", "La duracion del proceso de
                        entrenamiento puede ser largo")
    self.vista.comenzar(self.get_entrenamientos())

    for i in range(self.get_entrenamientos()):
         self.aprendizaje.aprender(False)
         self.vista.avanzar(1)
         self.raiz.update_idletasks()

    self.vista.avanzar(self.get_entrenamientos()-0.01)

    self.ventana_info("Fin de los entrenamientos", "Comieza la partida en 5
                        segundos")
    self.raiz.update_idletasks()
    sleep(5)

    self.raiz.destroy()
```

Figura 9.19: Método de lanzar entrenamientos de la clase ControladorPopUp

El método de la figura 9.19 lanza una ventana de advertencia inicial, ya que este proceso tarda bastante e incluso puede parecer que el programa se cuelga, pero no, está cargando. De hecho inicialmente debido a lo fuerte del proceso que se ejecutaba en primer plano, como es el hecho de llamar a aprender, la barra de progreso no se actualizaba. Afortunadamente Tkinter soluciona esto pudiendo exigir que se muestren las tareas que estubieran en espera y se dediquen al apartado gráfico, esto se hace gracias a la llamada `update_idletasks()`.

Por su parte la clase VistaPopUp es una pequeña ventana con un texto, una entrada controlada, un botón y una barra de progreso. No tiene más.

Tras el aprendizaje el controlador lanza un mensaje de finalización y cinco segundos después se cierra la ventana y se abre la del juego. En la figura 9.20 se puede apreciar la nueva ventana.

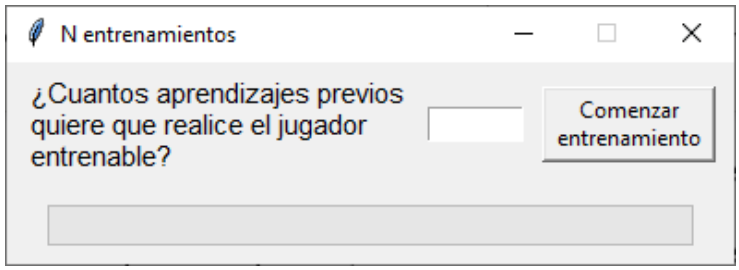

Figura 9.20: Ventana PopUp para dar numero de aprendizajes

9.5 Pruebas

En este subcapítulo se detallan algunas de las pruebas que se han realizado específicamente para el análisis y el aprendizaje. Aunque es cierto que la mejor prueba del funcionamiento adecuado, dado que ya teníamos un sistema de juego sólido y probado, es analizar las estadísticas que se comprueban en el capítulo siguiente.

9.5.1 Test unitarios

En esta ocasión se ha probado qué la codificación de la clase Backgammon que codifica un estado funcione adecuadamente, codifique bien y genere correctamente la lista de valores.

Se han llevado pruebas en el paquete de análisis destinadas a comprobar que no existan posibles estados de bloqueo y no salten errores.

9.5.2 Test nueva parte de la interfaz gráfica

Centrándonos únicamente en el funcionamiento del nuevo controlador y la nueva vista se ha probado repetidamente que los valores inadecuados de entrada lancen ventana de advertencia (figura 9.21), que al empezar una partida con un valor valido se indique al jugador que este proceso puede tardar (figura 9.22), que la barra de

progreso avance (figura 9.23) y que tras la ejecución de los aprendizajes se pueda jugar (figura 9.24).

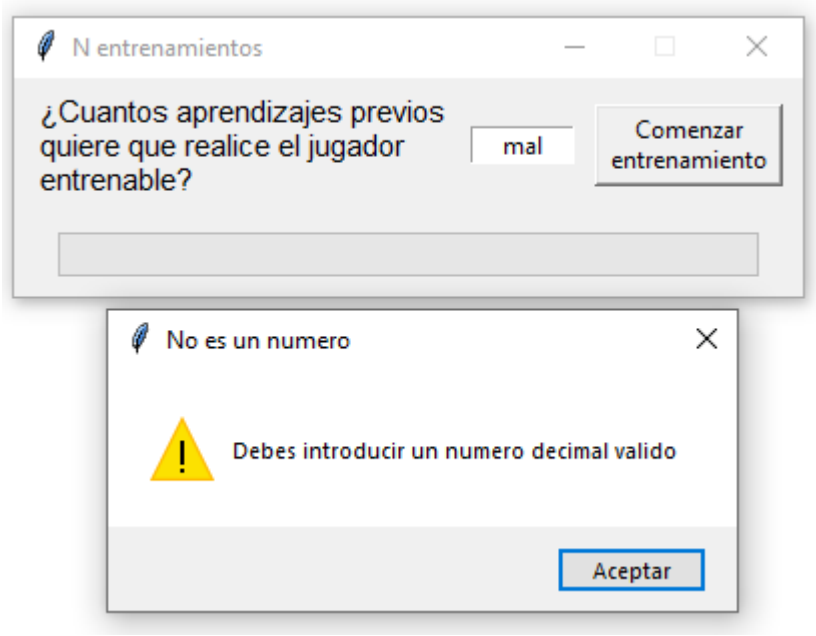

Figura 9.21: Ventana de advertencia dato mal introducido

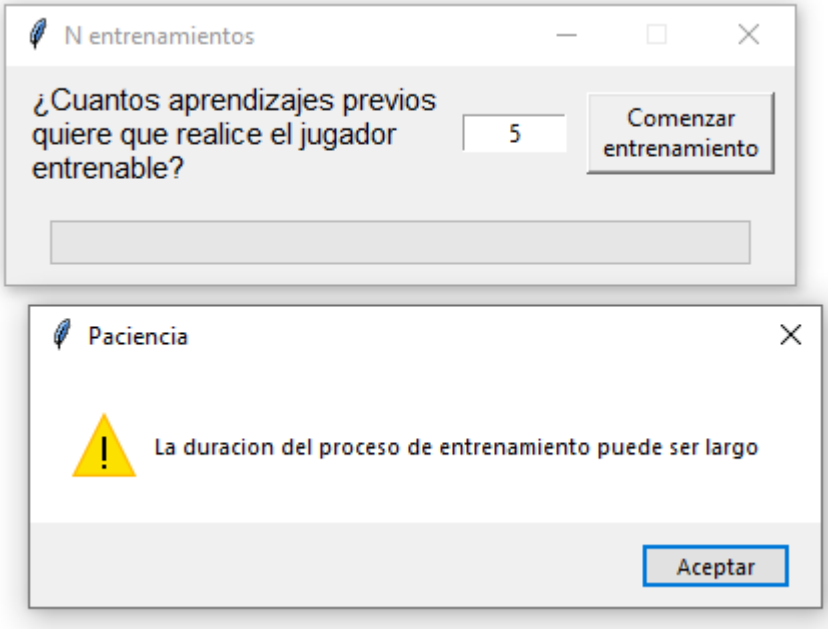

Figura 9.22: Advierte al usuario que el proceso puede tardar

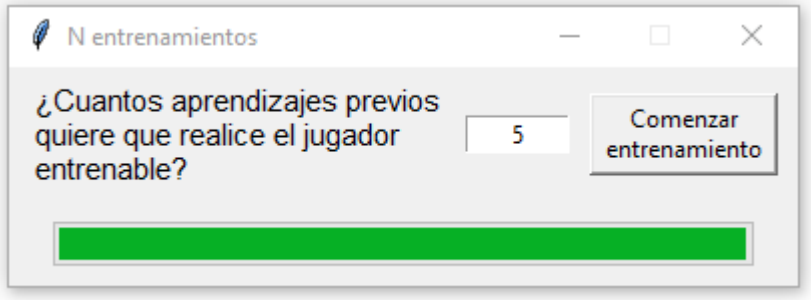

Figura 9.23: Proceso de aprendizaje avanza

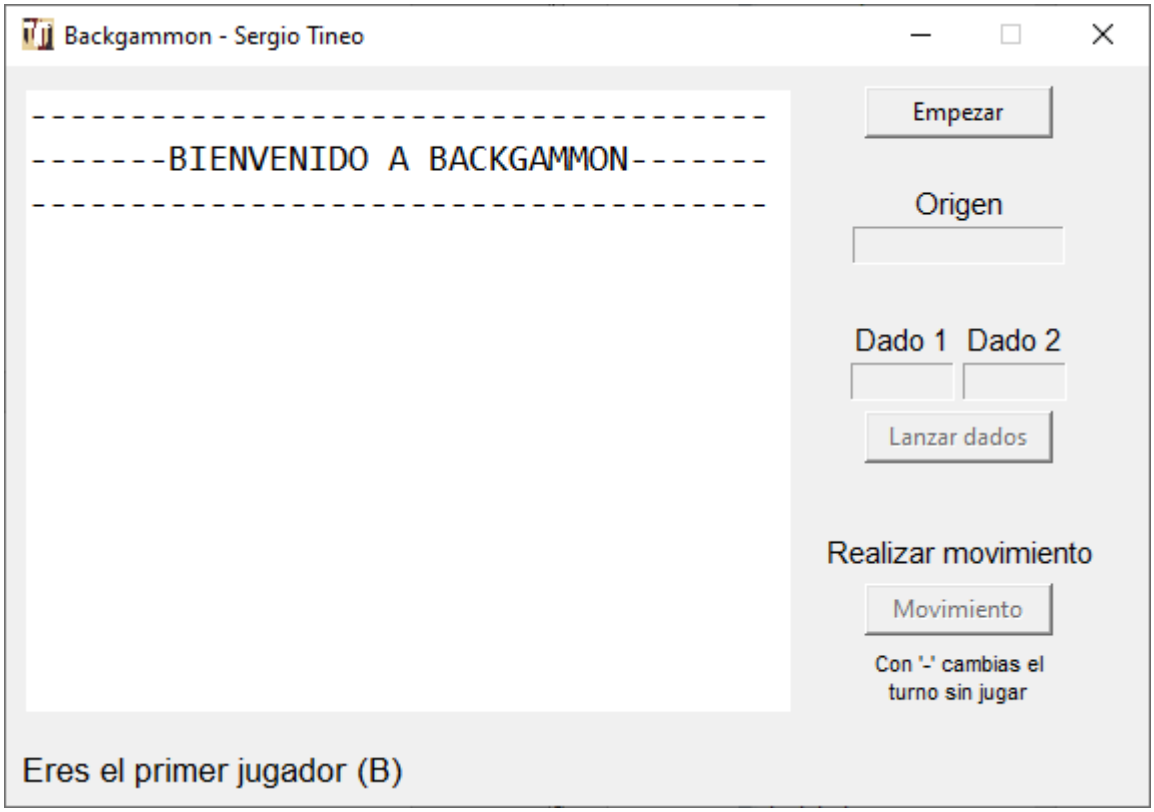

Figura 9.24: Se puede jugar como siempre

10

Análisis de resultados

En este capítulo se extraen e interpretan los resultado obtenido de haber ejecutado varias veces el AnlisisAprendizaje.

10.1 Convergencia de Evaluación inicial para un jugador

Una de las formas de ver que el aprendizaje es efectivo es, por ejemplo, evaluar el estado inicial del tablero y que se estabilice en un valor positivo, es decir, que comience a converger la evaluación. Ya que significa que el jugador entrenable está aprendiendo. En los próximos casos para un agente se han realizado 150 episodios. Recordemos que un episodio consiste en primero un entrenamiento y después jugar una partida.

En un aprendizaje realizado por un solo agente podemos observar como a lo largo de los episodios la evaluación del estado inicial va creciendo, inicialmente es cero y conforme comienza a encontrar las mejores posiciones en el tablero va mejorando su evaluación del estado, se puede observar en la gráfica en la figura 10.1 (figura 10.2, fichero: ev_ep.m). Sin embargo no siempre converge para un numero de episodios dado, ya que la figura 10.1 crece pero no llegue claramente a estabilizarse, sin embargo la figura 10.3 sí se aprecia más claramente.

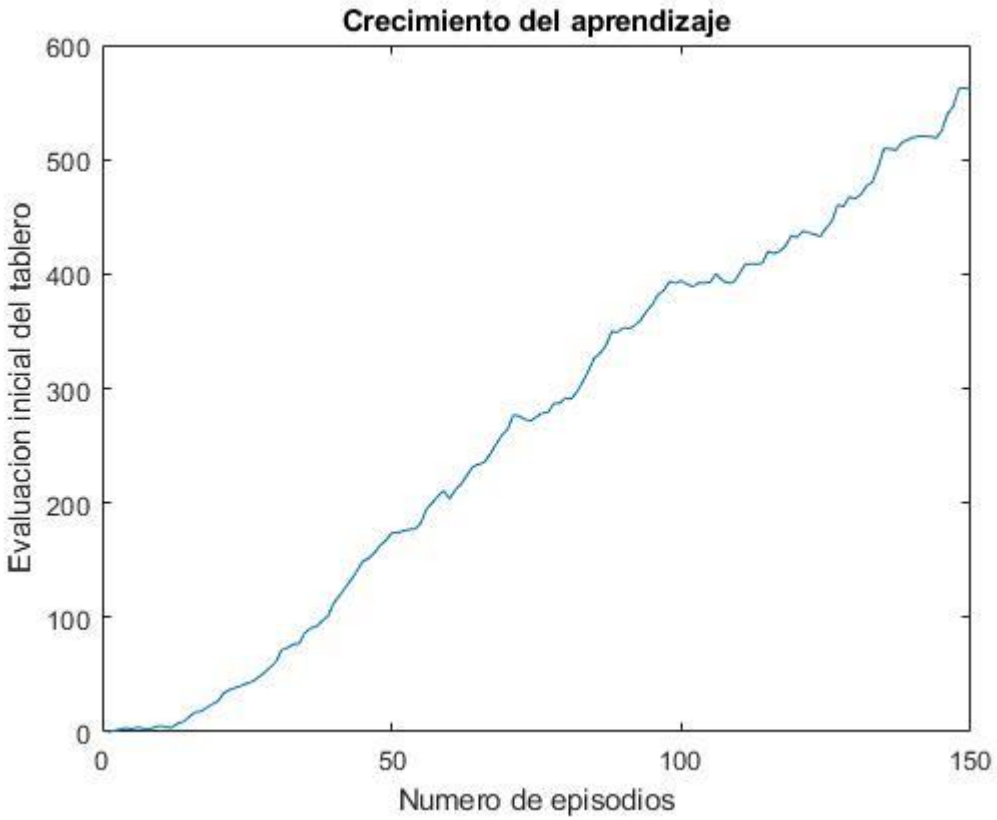

Figura 10.1: Curva de aprendizaje de un solo jugador (1)

```
X = [1, 2, …, 149, 150];
Y = [0, 1.0, 2.098658673271468, …, 562.3070073819393, 561.9277175147654];
plot(X,Y);
title('Crecimiento del aprendizaje');
xlabel('Numero de episodios');
ylabel('Evaluacion inicial del tablero');
```

Figura 10.3: Fichero ev_ep.m de evaluación inicial de un agente referifa a la gráfica 10.2

En la figura 10.3 se han eliminado 146 de X y 145 valores de Y para que pueda entrar en el documento. Esa figura ilustra el fichero de evaluaciones generado tras el análisis.

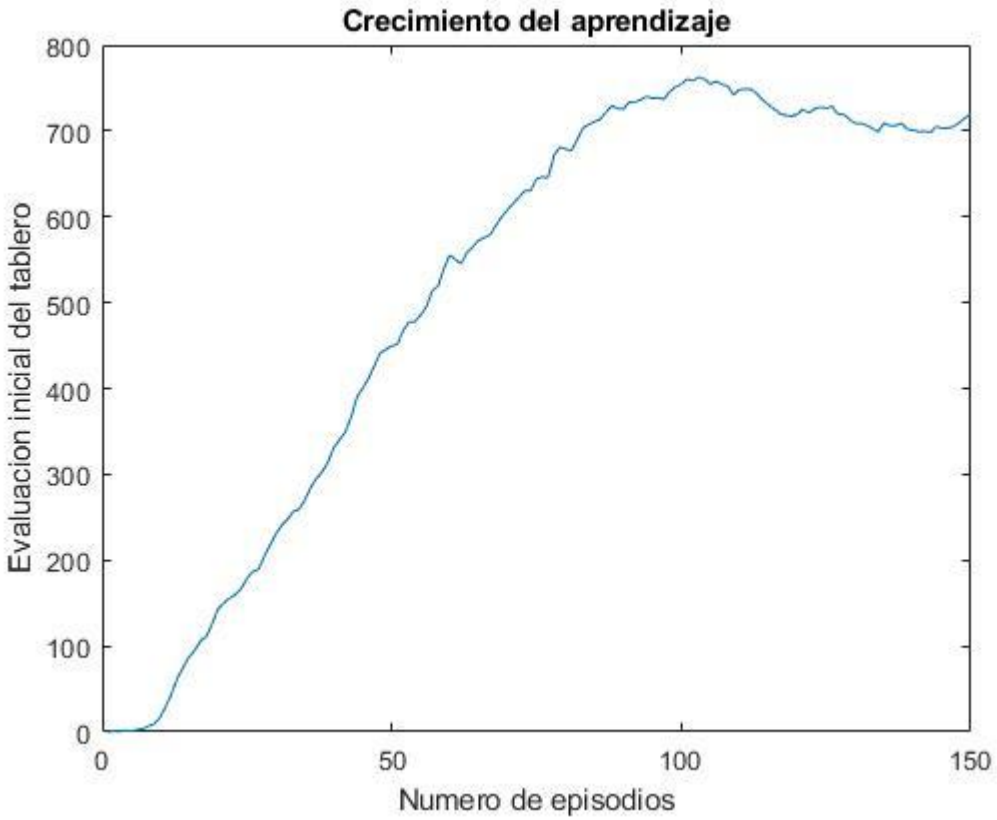

Figura 10.3: : Curva de aprendizaje de un solo jugador (2)

En la figura 10.3 se puede ver como los valores comienzan a fluctuar en torno a 700. Estas diferencias se deben a que no siempre se van a dar las mismas situaciones para realizar el aprendizaje, en partidas diferentes la mayoría de sucesión de estados serán diferentes.

10.2 Porcentaje de victorias/derrotas para un jugador

En cuanto al ratio de victorias y derrotas, inicialmente la cantidad de veces que comienza perdiendo o ganando es bastante aleatorio debido al comportamiento estocástico de los dados y a que aún no sabe cuáles son los mejores lugares donde debe situar sus fichas, sin embargo, conforme aprende comienza a saber cuáles son sus jugadas más favorables y por ello empieza a ganar un mayor número de veces. En la figura 10.4, una gráfica es reflejo de la otra por el hecho de que no se ha contemplado situación de empate, es decir, todo lo que no gane un jugador, gana el otro. La linea

punteada en azul corresponde a las victorias y la punteada en rojo a las derrotas. (figura 10.5, fichero: victorias_derrotas.m).

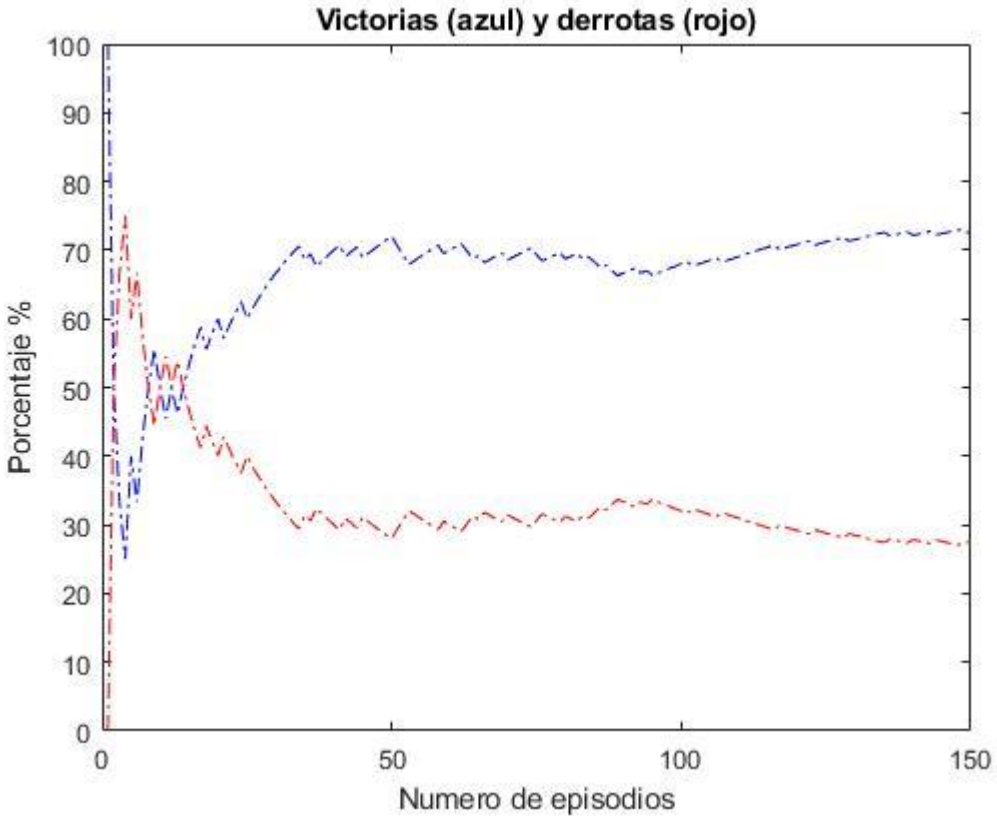

Figura 10.4: Porcentaje de victorias y derrotas para un solo jugador

Igualmente en la figura 10.5 se ha eliminado una cantidad de valores para que se pueda apreciar. Destacar que Y es la cantidad de victorias e Y2 es la cantidad de derrotas para el jugador con aprendizaje.

```
X = [1, 2, …, 149, 150];
Y = [100.0, 50.0, … 72.48322147651007, 72.66666666666667];
Y2 = [0.0, 50.0, …, 27.516778523489933, 27.333333333333332];
plot(X,Y,'b',X,Y2,'r','linestyle','-.' );
ylim([0 100]);
title('Victorias (azul) y derrotas (rojo)');
xlabel('Numero de episodios');
ylabel('Porcentaje %');
```

Figura 10.5: Fichero victorias_derrotas.m de victorias y derrotas de un agente

En la vista de los presentes datos podemos afirmar que efectivamente el jugador está aprendiendo. Ya que jugando aleatoriamente no se puede ganar la mayoría de veces. Igualmente debido a los dados también es imposible ganar siempre.

10.3 Convergencia de Evaluación inicial para k jugadores

Para intentar evitar reducir el hecho de que un jugador entrenable, llegue antes o más tarde que otro jugador entrenable, a una evaluación inicial del tablero que converja, es decir, para evitar el "factor suerte" de los dados para el rival se hace muy interesante generar estadísticas de la media de k agentes.

Con 10 agentes y 150 episodios, grafica de la figura 10.6, ya podemos observar que sí se suele estabilizar el valor de evaluación inicial.

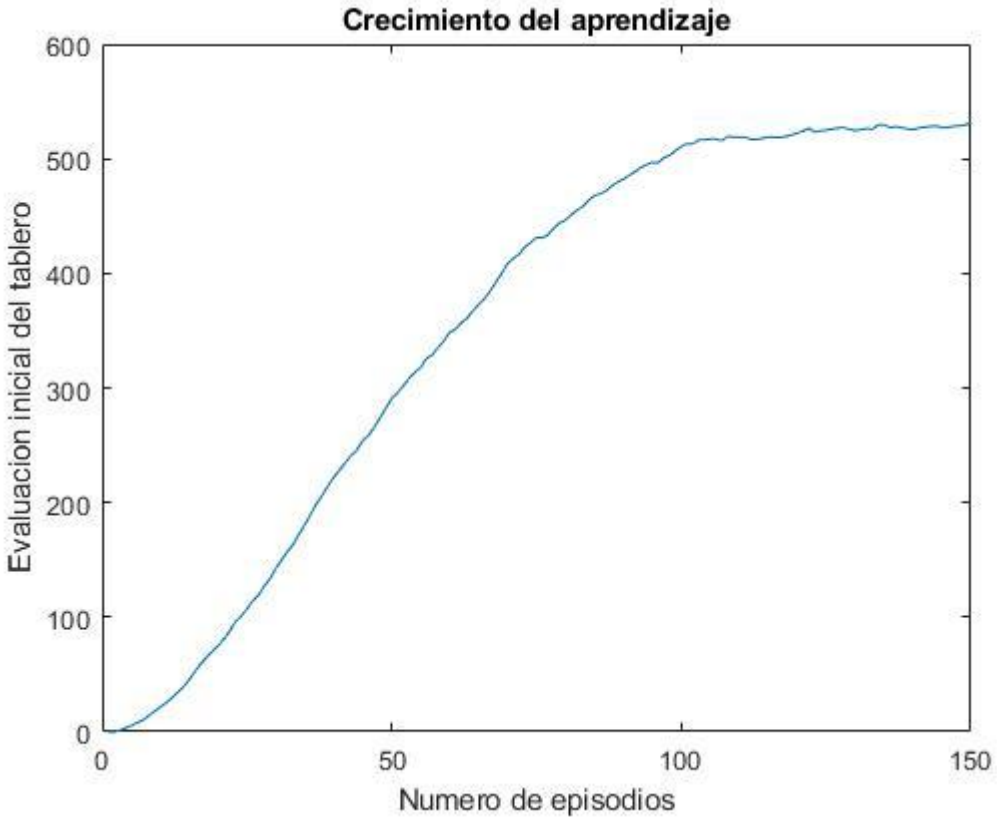

Figura 10.6: Curva de aprendizaje de diez jugadores

Aquí podemos ver una progresión mucho más lineal del crecimiento. El fichero generado tiene otro nombre (ev_ep_k_agentes.m) pero sigue la misma estructura que el de la figura 10.2.

10.4 Porcentaje de victorias/derrotas para k jugadores

En la figura 10.7 podemos observar como es mucho más suave la progresión del porcentaje de victorias/derrotas con la media de 10 agentes, comenzando inicialmente en torno al 50%, algo lógico, y acabando muy favorablemente para el jugador que aprende. Llegando a estar en torno al 80% de media las victorias con solo 150 aprendizajes.

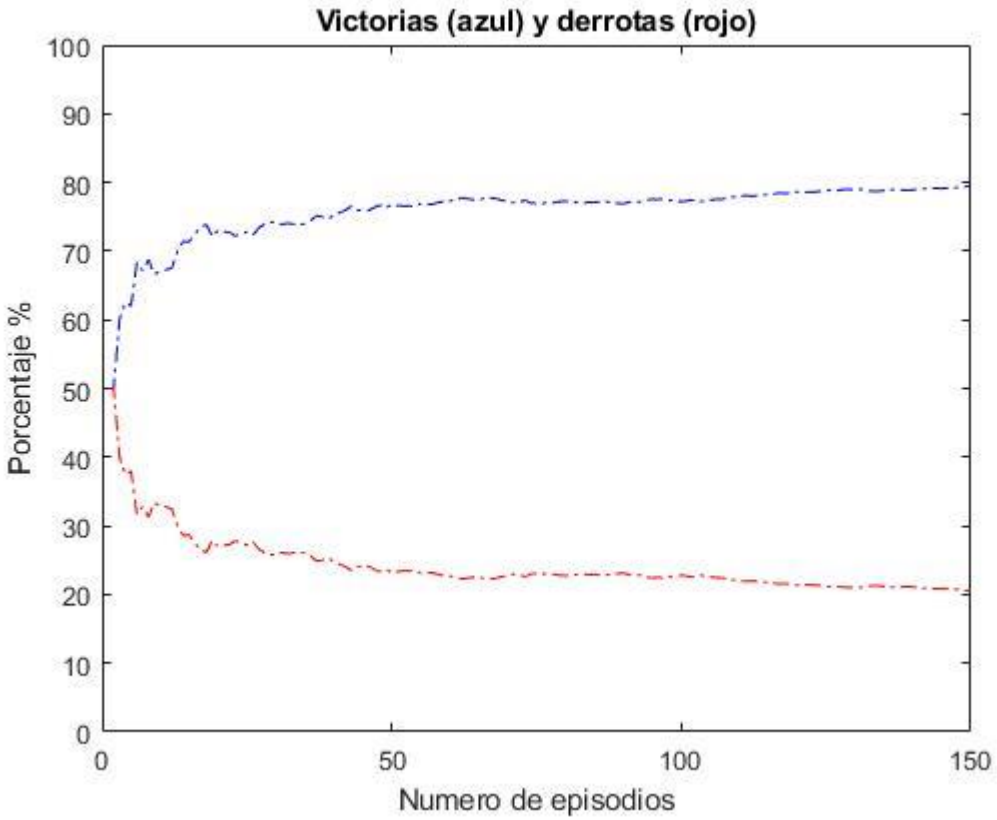

Figura 10.7: Porcentaje de victorias y derrotas para diez jugadores

El fichero generado en este caso sigue teniendo la misma estructura que el de la figura 10.5 pero tiene un nombre diferente (victorias_derrotas_k_agentes.m).

10.5 Conclusiones del análisis de los datos

Podemos concluir que nuestro jugador aprende, es imposible que en 150 partidas se de el caso de ganar el 80% de forma estable, incluso con diferentes agentes con sus diferentes situaciones. Así mismo con una mayor cantidad de partidas podríamos hablar tranquilamente de un 90-95% de victorias.

De hecho una de las pruebas intermedias que hicimos para ver la evolución favorable del aprendizaje, mostró que ganaba más del 93% de las ocasiones con 300 aprendizajes completos previos y 1000 partidas. Evidencias en la figura 10.8. Para ejecutar esto se tubo realizar de una forma diferente (véase apéndice).

```
RESULTADOS:

Con 300 aprendizajes previos y 1000 partidas

Los resultados son 68 victoria para J1 y 932 victorias para J2

O lo que es lo mismo:

El jugador con inteligencia por refuerzo gana un 93.2% y pierde un
6.8% de las veces
```

Figura 10.8: Resultados de 300 aprendizajes y 1000 partidas para un mismo jugador

Ahora se entiende por qué se necesita de granjas de servidores para conseguir hacer que "jugadores inteligentes" que aprendan. En el equipo donde se han realizado las pruebas se contaba con un procesador i7 de sexta generación pero debido al reparto de tareas de Windows no se solía superar el 20% de ocupación del procesador y esto hacía que el cálculo de los datos se alargase. Por su parte una buena noticia es que jamás llegó a superar los 100Mb de memoria dinámica.

11

Objetivos alcanzados y trabajos futuros

En este capítulo se describirán los objetivos que se han alcanzado, podemos adelantar que se han cumplido todos los del anteproyecto y alguno adicional. Y también hablaremos sobre los trabajos futuros.

11.1 Objetivos alcanzados

Podemos afirmar que se han cumplido cada uno de los objetivos que se plantearon en el anteproyecto en el capítulo 1 del presente trabajo. Enumerando los objetivos alcanzados, tenemos:

- ✓ Aprender un nuevo lenguaje (Python).
- ✓ Afianzar las bases del desarrollo software partiendo de análisis, pasando por implementación y llegando a pruebas.
- ✓ Estudiar y aplicar aprendizaje por refuerzo al desarrollo.
- ✓ Programa para un jugador del juego Backgammon sin y con jugador con aprendizaje por refuerzo.
- ✓ Programa para ver competir la máquina contra si misma.

- ✓ Programa para dos jugadores del Backgammon con interfaz gráfica de usuario.
- ✓ Analisis de los datos tras la aplicación de algun algoritmo de aprendizaje por refuerzo. El algoritmo usado fue finalmente el de aprendizaje por diferencias temporales.

Adicionalmente también hemos alcanzado un nuevo objetivo que no estaba inicialmente, pero se planteó durante la fase de analisis de la segunda iteración del trabajo, y se implementó. Esta nueva capacidad es la de generar un archivo que Matlab es capaz de abrir y ejecutar mostrando gráficos de líneas con los datos extraídos del aprendizaje.

Por su parte durante la elaboración del trabajo se han sucedido algunos problemas que afortunadamente se solventaron. La mayoría de estos fallos son propios de este lenguaje, como que si una clase y un paquete que se importe se llaman igual no da error en tiempo de escritura en el IDE pero en la ejecución sí. Otro fallo es que como el lenguaje no exige el tipado, en ocasiones este hecho puede conducir a error por el simple hecho de escribir mal la llamada a un método aunque ni si quiera salte el error inicialmente.

11.2 Trabajos futuros

El programa que se presenta en este trabajo es una base sobre la que se puede trabajar de forma sencilla gracias a la documentación que presenta cada una de las clases y funciones del programa.

Una de las ideas más interesantes de cara al jugador sería utilizar algun módulo gráfico de Python, como pueda ser Pygame. Este es un módulo sirve para crear juegos 2D con *sprites*, algo ideal para el juego del Backgammon.

Otra posible ampliación, de cara a continuar con las técnicas de aprendizaje, sería seguir el modelo de redes neuronales multicapa planteado por Gerald Tesauro [14] e implementar nuestra versión de TD-Gammon en Python.

Bibliografía

[1] Sutton, R. S. & Barto, A. G. (Ed. 2º). (2018). *Reinforcement Learning: An Introduction*. London: The MIT Press.

[2] Python-dev. (2019). Python Enhancement Proposals (PEPs). *Python Developer's Guide*. Recuperado de https://www.python.org/dev/peps/

[3] Documentacion de Python (Python Documentation). (s. f.). Recuperado de https://docs.python.org/3/contents.html

[4] Python tutorials. (s. f.). *W3schools.* Recuperado de https://www.w3schools.com/python/

[5] Sutton, R. S. & Barto, A. G. (2018). The Agent–Environment Interface. (Ed. 2º), *Reinforcement Learning: An Introduction*. (pp. 48). London: The MIT Press.

[6] Mandow Andaluz, L. (2017, 2018) Documentación de signatura Inteligencia Artificial para Juegos. Malaga: Universidad de Malaga.

[7] Blasco, R. (2018). Backgammon: una historia muy antigua. *Cajón desastre*. Recuperado de https://www.anthropologies.es/backgammon-una-historia-muy-antigua/

[8] Ancient game boards. (2018). Recuperado de https://ladyheatherhall.com/2018/02/16/ancient-game-boards/

[9] Campeonato mundial de Backgammon. Sitio web https://www.bwcmc.com/index.php

[10] Backgammon Rules. (s. f.). Recuperado de https://bkgm.com/rules.html

[11] Reglas del juego Backgammon. (s. f.). Recuperado de http://www.gammon-expert.es/reglas-juego-backgammon.htm

[12] Office 365. Sitio web: https://www.office.com/

[13] JetBrains. Python 2019 - Infografía del estado del exosistema del desarrollador en 2019. (2019). *El estado del ecosistema del desarrollador 2019*. Recuperado de https://www.jetbrains.com/es-es/lp/devecosystem-2019/python/

[14] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communucations of the ACM,* 38(3).

[15] Recursos Python. (2018). Clases: métodos mágicos y propiedades. *Guias y manuales*. Recuperado de https://recursospython.com/guias-y-manuales/clases-metodos-magicos-y-propiedades/

[16] Sutton, R. S. & Barto, A. G. (2018). TD Prediction. (Ed. 2º), *Reinforcement Learning: An Introduction.* (pp. 119-123). London: The MIT Press.

[17] IEEE Computer Society. IEEE-Std 830-1993. (1993*). IEEE Recommended Practice for Software Requirements Specifications*. Institute of Electrical and Electronics Engineers.

[18] Kononenko, I. (2001). Machine learning for medical diagnosis: history, state of the art and perspective. Artificial Intelligence in medicine, 23(1), 89-109.

[19] Benito. A. (2018). Copias superficiales y profundas. Recuperado de https://github.com/grialusal/materiales-informatica-

i/blob/master/labs/lab5/lab5.ipynb
[20] Sutton, R. S. & Barto, A. G. (2018). Games, Afterstates, and Other Special Cases. (Ed. 2º), *Reinforcement Learning: An Introduction.* (pp. 136-137). London: The MIT Press.
[21] Merino, M. (2019). Conceptos de inteligencia artificial: qué es el aprendizaje por refuerzo. *Inteliegencia Artificial*. Recuperado de https://www.xataka.com/inteligencia-artificial/conceptos-inteligencia-artificial-que-aprendizaje-refuerzo
[22] Garcia Moreno, A. (2013). Condicionamiento Operante, explicacion. Recurso audiovisual recuperado de https://www.youtube.com/watch?v=-tWf4VTN5r0
[23] Silva, M. (2019). Aprendizaje por Refuerzo: Planificando con Programación Dinámica. Recuperado de https://medium.com/aprendizaje-por-refuerzo-introducción-al-mundo-del/aprendizaje-por-refuerzo-planificando-con-programación-dinámica-200ebd2af48f
[24] GitHub. Sitio web: https://github.com/
[25] Anaconda. Sitio web: https://www.anaconda.com/
[26] Jupyter Notebook. Sitio web: https://jupyter.org/
[27] Pycharm. Sitio web: https://www.jetbrains.com/pycharm/
[28] Notepad++. Sitio web: https://notepad-plus-plus.org/
[29] MagicDraw. Sitio web: https://www.nomagic.com/products/magicdraw
[30] GIMP. Sitio web: https://www.gimp.org/
[31] Free Mind. Sitio web: http://freemind.sourceforge.net/wiki/index.php/Main_Page
[32] CmapTools. Sitio web: https://cmap.ihmc.us/
[33] Interprete de Python. Sitio web: https://www.python.org/downloads/

Apéndice A

Manual de Instalación

En este apéndice se explica los materiales necesarios para poder hacer funcionar la aplicación de manera sencilla y adecuada.

A.1 Interprete de Python

Para hacer funcionar el programa necesitamos tener el intérprete de Python instalado. Para hacerlo simplemente es necesario descargarlo de su página oficial [33] e instalarlo como cualquier otro programa.

Una vez lo tenemos instalado existen dos posibilidades, una de ellas es si solo queremos ejecutar algunos de los archivos *.exe proporcionados que lanzan cada una de las partes de la aplicación, no es necesario continuar con este manual. Acabaría aquí.

A.2 PyCharm

Opcionalmente, pero a la vez lo **recomendado**, es instalar PyCharm descargándolo desde su web oficial [27] ya que desde el se tendrá acceso a todos los ficheros fuentes y estarán debidamente indexados por el IDE.

Instalar PyCharm al igual que con el intérprete es muy sencillo, simplemente "siguiente" y "siguiente" hasta acabar. La parte compleja viene a la hora de enlazar PyCharm con el intérprete.

Una vez abrimos PyCharm por primera vez si no nos encuentra el intérprete debemos indicárselo de forma manual. Para eso nos vamos a la pestaña File y buscamos

la opción Setting, una vez ahí navegamos en la barra de menú lateral siguiendo la siguiente jerarquía mostrada en la figura AF.1.

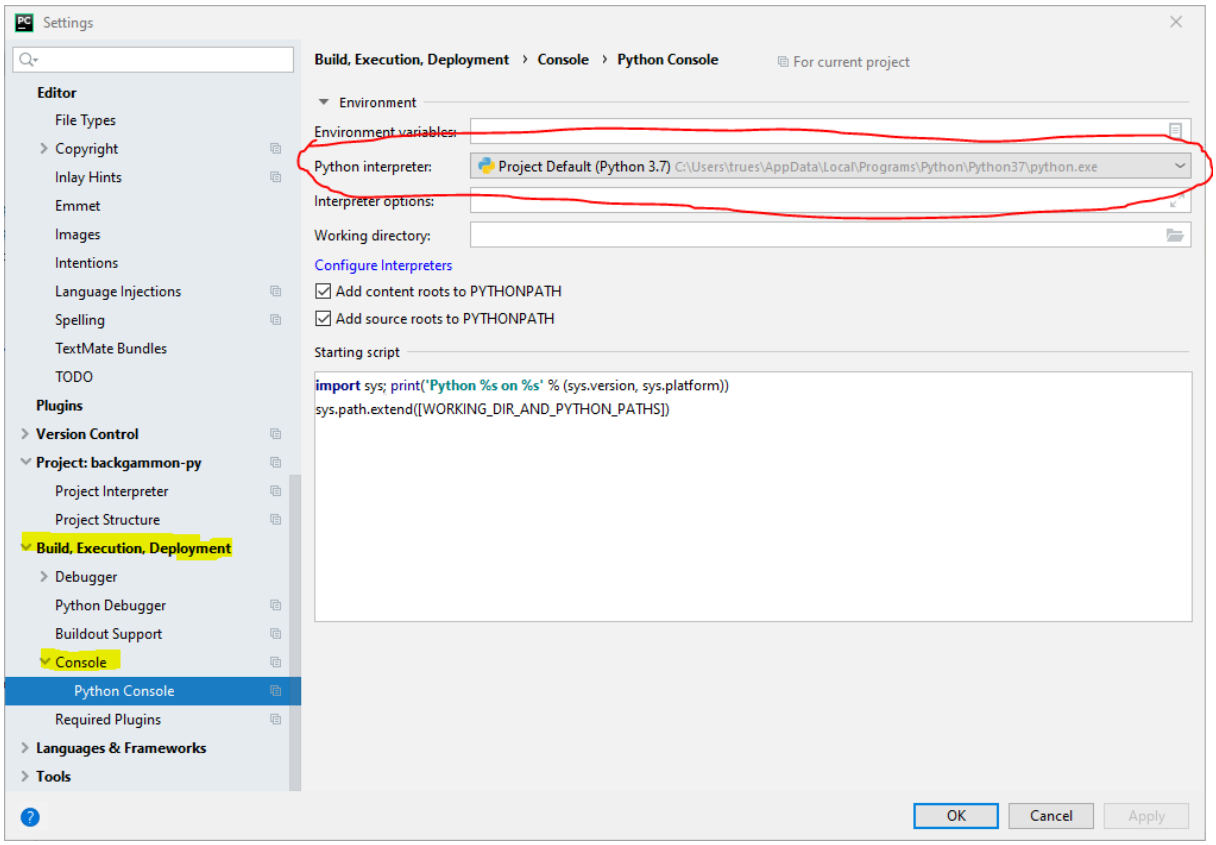

Figura AF.1: PyCharm configuración

La ruta deberá ser la ruta del exe de vuestro intérprete, variara dependiendo de donde lo instaléis. Pero generalmente estará en la carpeta de vuestro usuario, en AppData\Local\Programs\Python como se observa en la figura AF.1.

Si seguís teniendo problemas deberéis decirle que interprete usar exactamente para vuestro proyecto en concreto, figura AF.2.

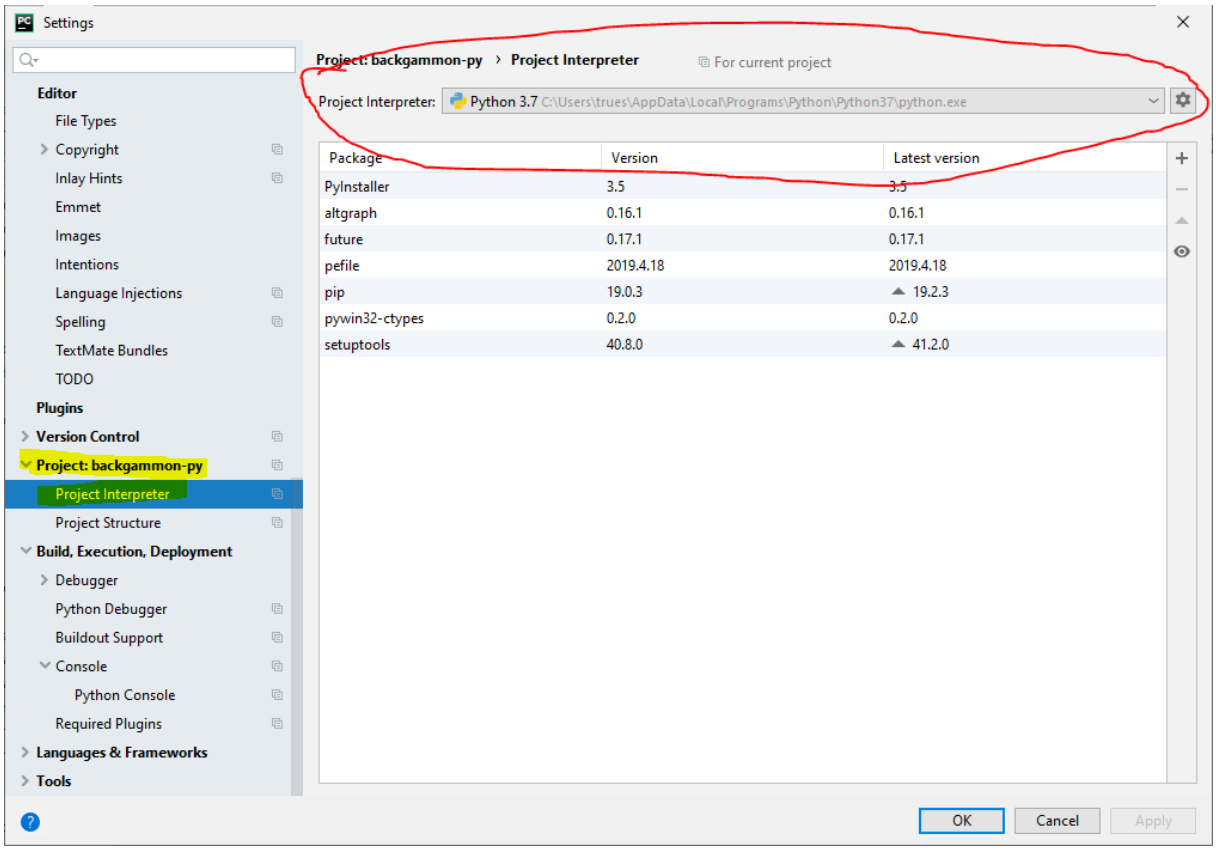

Figura AF.2: PyCharm configuración (2)

Una vez hecho esto ya estaría todo listo. Ya podéis disfrutar de este maravilloso programa y ver como funciona.

Apéndice B

Manual de para ejecutar los Scripts

En este apéndice se detalla la forma de ejecutar cada uno de los scripts y que ejecuta cada uno. Entendemos por jugador aleatorio al jugador sin aprendizaje.

B.1 Ejecutar juego con jugador humano vs jugador aleatorio

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_gui_pers_vs_alea/**ejec_gui_pers_vs_alea.exe**

Este lanzará en programa para jugar contra el jugador aleatorio haciendo uso de la interfaz gráfica de usuario.

Si usamos PyCharm deberemos lanzar el script:

bin/**ejec_gui_pers_vs_alea.py**

B.2 Ejecutar juego humano vs humano

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_gui_2p/**ejec_gui_2p.exe**

Este lanzará en programa para jugar contra otro jugador humano haciendo uso de la interfaz gráfica de usuario. Ambos jugaran en la misma ventana.

Si usamos PyCharm deberemos lanzar el script: bin/**ejec_gui_2p.py**

B.3 Ejecutar juego humano vs jugador con aprendizaje

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_gui_pers_vs_entr/**ejec_gui_pers_vs_entr.exe**

Este lanzará en programa para jugar contra el jugador aleatorio haciendo uso de la interfaz gráfica de usuario.

Si usamos PyCharm deberemos lanzar el script:

bin/**ejec_gui_pers_vs_entr.py**

B.4 Ejecutar juego jugador aleatorio vs jugador aleatorio

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_no_gui_j_aleatorio/**ejec_no_gui_j_aleatorio.exe**

Este lanzará en programa para ver por consola de órdenes jugar a dos jugadores aleatorios.

Si usamos PyCharm deberemos lanzar el script:

bin/**ejec_no_gui_j_aleatorio.py**

B.5 Ejecutar aprendizaje jugador aleatorio vs jugador con aprendizaje

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_aprendizaje/**ejec_aprendizaje.exe**

Este lanzará en programa para ver por consola de órdenes jugar a dos jugadores aleatorios.

Si usamos PyCharm deberemos lanzar el script:

bin/**ejec_aprendizaje.py**

B.6 Ejecutar conjunto de aprendizaje con generación de datos estadísticos

Deberemos buscar el archivo dentro del proyecto en la carpeta de ejecutables dentro de bin. Concretamente ejecutar:

bin/dist/ejec_aprendizaje/**ejec_analisis.exe**

Este lanzará en programa para ver por consola de órdenes jugar a dos jugadores aleatorios.

Si usamos PyCharm deberemos lanzar el script:

bin/**ejec_analisis.py**

Y recordad, la mejor forma de aprender a jugar es practicando. Hasta aquí la memoria del fantástico backgammon-py.