Universidad Zaragoza

Trabajo Fin de Grado

Ingeniería Informática

Videojuego de lucha uno contra uno en red y con inteligencia artificial adaptativa

Online One-on-One Fighting Video Game with Adaptive Artificial Intelligence

Autor

Saúl Alarcón Cano

Director

Eduardo Mena Nieto

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Septiembre 2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. Saúl Alarcón Cano ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado (Título del Trabajo)

Título en español: Videojuego de lucha uno contra uno en red y con inteligencia artificial adaptativa
Título en inglés: Online One-on-One Fighting Video Game with Adaptive Artificial Intelligence

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 22 de Septiembre de 2020

Fdo: 

I. AGRADECIMIENTOS

En primer lugar agradecer a mi director, Eduardo Mena, por su ayuda y consejo a lo largo del desarrollo del proyecto, pero sobre todo por aceptar tutorar mi propia propuesta de proyecto.

Dar las gracias también a todos los profesores que me han dado clase hasta el día de hoy (tanto de estudios pre-universitarios, como universitarios), ya que gracias a sus enseñanzas y devoción he llegado hasta aquí.

Agradecer a mis amigos, en especial a Fo-kun (el que más me ha apoyado y ayudado sin lugar a dudas), que me ayudaron a realizar diversas pruebas de mi proyecto aun a costa de su tiempo libre, y también a Sergio y Sisqués, ya que participaron en el desarrollo del videojuego base para la asignatura.

Y finalmente, a mis padres por apoyarme constantemente y motivarme desde pequeño a esforzarme en los estudios, ya que sin ellos no sería quien soy hoy.

II. RESUMEN

Los videojuegos comenzaron poco después de que surgiese el concepto de Informática, intentando siempre cubrir esa necesidad de entretenimiento innata en las personas de formas que el mundo real no podía. Desde el nacimiento de los videojuegos, estos han ido evolucionando con la Informática de la mano, llegando a convertirse en un entretenimiento que está en el día a día de las personas, y siendo actualmente el sector del entretenimiento que más dinero genera (facturando en España desde 2008 más que el cine, el DVD y la música juntos). Los juegos en red están a la orden del día, y entre ellos cada vez cogen más fuerza los videojuegos enfocados al mundo competitivo (*e-sports*), punto a partir del cual surge este proyecto.

Este proyecto trata acerca de la creación de un juego de lucha en dos dimensiones, tomando como punto de partida las prácticas de la asignatura Videojuegos, convirtiendo un clon simplificado del juego *arcade* original Fatal Fury 2 (SNK 1992) en un videojuego que contiene características en red e incorpora una inteligencia artificial adaptativa. Para cumplir este objetivo, ha sido necesaria una arquitectura que permitiese a los jugadores enfrentarse entre sí desde cualquier ordenador con acceso a internet sin la necesidad de conocimiento alguno de Informática, y que les proporcionase diversos características que se consideran imprescindibles actualmente, como serían la socialización entre jugadores mediante 'amistades' y comunicaciones a través del propio juego.

Para proporcionar a los jugadores un reto mayor contra el que poder entrenar, se ha añadido posibilidad de jugar contra dos inteligencias artificiales adaptativas. En el caso de la primera, la inteligencia artificial va aprendiendo del estilo de juego del usuario y sus puntos débiles a medida que juegue partidas, proporcionándole así al jugador un medio para intentar mejorar sus flaquezas. Mientras que la segunda se centra en aprender de todos los jugadores y así suponer un reto a todos ellos, dándoles otra opción de entrenamiento más general.

Índice

Capítulo 1

Introducción

Poco después de la aparición de la Informática ya se pensó en aprovecharla de alguna forma para el entretenimiento personal, surgiendo la idea de los videojuegos. Estos fueron evolucionando junto con la Informática en sí, exprimiendo en muchos casos el hardware al límite de sus capacidades. Rápidamente los videojuegos triunfaron, extendiéndose cada vez más entre la gente y su vida diaria, llegando poco a poco al punto en el que se encuentra hoy, siendo los videojuegos algo que forma parte de la vida de las personas de una u otra manera. Parte del gran éxito que han alcanzado se debe a la inclusión de servicios en red, que hicieron que los videojuegos dejasen de ser un entretenimiento de ámbito local (jugar solo o con gente en persona), para pasar a ser un entretenimiento de ámbito global, permitiendo jugar y socializar con cualquier persona del mundo, desde la comodidad de tu hogar. Por esto, a día de hoy, se podría decir que los servicios en red forman una parte prácticamente indispensable de una gran parte de videojuegos, y sobretodo de aquellos que entran en el campo competitivo.

Con esto en mente, el propósito de este Trabajo de Final de Grado es mejorar un videojuego de lucha (realizado para la asignatura Videojuegos) para que proporcione dichos servicios en red que actualmente se consideran indispensables sobretodo en juegos multijugador. Algunos de estos servicios serían por ejemplo gestión de usuarios y amistades, y partidas en red. Estos y otros servicios son los que se proporcionarán al usuario de la forma más transparente posible y que no le exija conocimientos avanzados de Informática. Para conseguir esto, el ejecutable del juego deberá conectarse automáticamente con el servidor, sin que usuario se tenga que preocupar de nada más allá de crearse una cuenta, y sea el servidor el que se encargue de proporcionarle todos los servicios de forma segura.

Además de estos servicios básicos, se les proporcionará a los usuarios la posibilidad de entrenarse contra dos inteligencias artificiales adaptativas, cuyo objetivo será representar un desafío importante a los usuarios, que les permita mejorar sus habilidades en el juego. Concretamente una de las inteligencias artificiales se centrará

en aprender del propio jugador, buscando sus flaquezas, y haciendo que el jugador pueda mejorarlas, y la otra aprenderá de todos los jugadores, buscando suponer un reto más genérico para todos los jugadores. La segunda inteligencia artificial surge del hecho de que no se puede mejorar únicamente contra una que solo conozca nuestros patrones de ataque, se necesita enfrentarse a otras situaciones que fácilmente podrían no surgir con la primera.

1.1. Fatal Fury 2 (SNK 1992) y el clon de las prácticas

El juego original en el que se basa el videojuego de este proyecto es el Fatal Fury 2[1] (figura 1.1) lanzado en el año 1992 por SNK para la Neo Geo de arcade y diversas consolas de hogar como la Super Nintendo o la Sega Mega Drive. Este juego era la secuela del juego Fatal Fury: King of Fighters (SNK 1991), el cual era un juego de lucha en dos dimensiones, respecto del cual incluía nuevas mecánicas, como el uso de un total de cuatro botones, el salto rápido hacia atrás, el cambio de plano libre o el 'Movimiento de desesperación' (un ataque especial que solo se puede hacer cuando se tiene menos de un cuarto de vida).

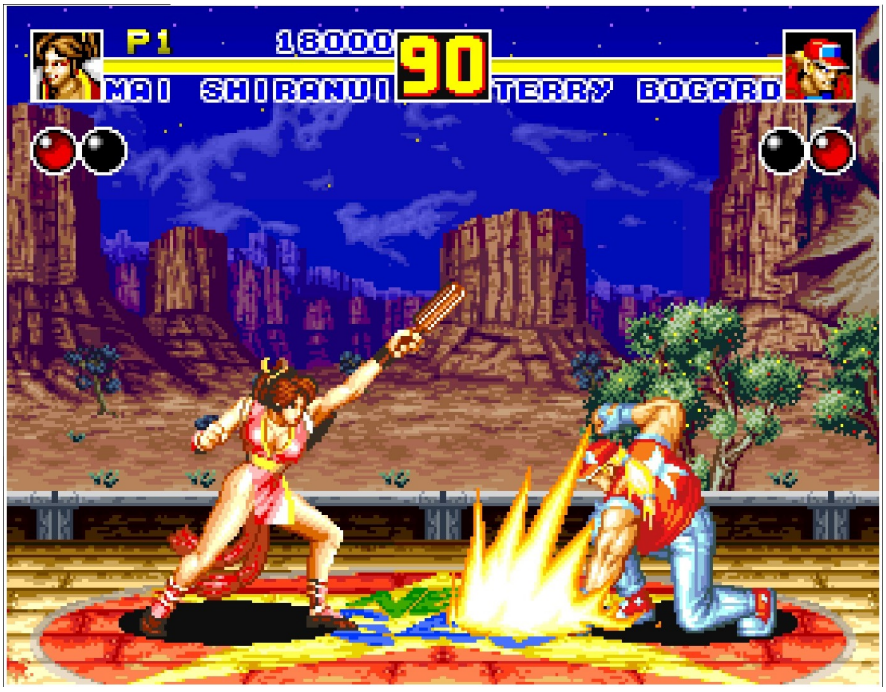


Figura 1.1: Fotograma de una pelea del juego original Fatal Fury 2 (SNK 1992).

El juego disponía de un total de ocho personajes jugables y otros 4 no jugables, y dos modos de juego, el de un solo jugador, en el que se enfrentaba a los ocho personajes

[1]Demostración del Fatal Fury 2 (SNK 1992): `https://www.youtube.com/watch?v=ZZmagM-y3w0`

jugables y posteriormente los cuatro no jugables a modo de jefes, y el modo jugador contra jugador que permitía a dos personas enfrentarse jugando ambos físicamente en la misma consola. La dificultad del modo de un jugador se seleccionaba en un menú de opciones, pudiendo elegir entre un total de ocho dificultades.

Para la asignatura Videojuegos el grupo *Chascarrillo Games*, del que formaba parte el autor de este proyecto, desarrolló un clon del Fatal Fury 2 (SNK 1992), con características reducidas por razones de tiempo. La mayor reducción fue en el número de personajes, pues de pasaron a tener únicamente tres personajes jugables. En cuanto a la pelea en sí se implementaron todas las mecánicas salvo dos, el cambio de plano y el 'Movimiento de desesperación', por lo que acabó siendo una pelea en dos dimensiones sin ninguna noción de profundidad (se la daba el cambio de plano), con cuatro tipos de ataques básicos (pata y puñetazo fuertes y débiles) y cuatro movimientos básicos (andar hacia delante y hacia atrás, saltar hacia arriba y agacharse). Combinando los cuatro tipos de ataques y los cuatro tipos de movimientos se daban lugar a muchas combinaciones posibles, correspondiéndose cada una a un tipo de movimiento, habiendo un total de 23 movimientos ejecutables por el jugador, más otras muchas animaciones que no son controlables por el jugador, como el retroceso por un golpe recibido.

En cuanto a los modos de juego, se implementaron un total de tres modos de juego, siendo el primero un modo de un jugador (con cuatro dificultades) que supone una secuencia de peleas contra los tres personajes jugables, otro modo de un jugador que permitía enfrentarse a un personaje con un nivel de dificultad elegibles, y un tercer modo para dos jugadores, que permitía enfrentarse dos personas, pero estando ambas físicamente juntas.

La IA implementada para el clon fue una IA muy básica basada en la definición de las probabilidades de hacer cada movimiento, las cuales se alteraban a lo largo de la partida en base a las configuraciones que se le estableciesen. Por ejemplo, un personaje podría tener un 0.5 de lanzar un puñetazo y un 0.5 de andar hacia atrás, pero si se le define un carácter agresivo al principio de la ronda y defensivo al final, empezaría la ronda con un 0.75 y un 0.25 (por ejemplo), y terminaría con un 0.25 u 0.75. Básicamente una IA muy simple, que no suponía un gran reto a los jugadores.

Por tanto, como se puede apreciar, había dos características a mejorar, la necesidad de estar físicamente juntas dos personas para poder jugar, y la IA que no suponía un reto, siendo esas dos características las que este proyecto abarcó. Señalar que para el proyecto en sí no se modifico nada de la lógica ni de los gráficos del juego en sí, manteniéndose los originales de las prácticas.

1.2. Motivación y objetivos

A la hora de elegir el Trabajo de Final de Grado fue necesario elegir entre realizar un trabajo de colaboración en una investigación, pero cuya elaboración resultase al autor mucho más dura y tediosa, o hacer uno que resultase más ameno de desarrollar y siguiese mostrando los conocimientos adquiridos mediante una propuesta propia. La elección fue la segunda opción, para lo cual se decidió que lo ideal sería hacer algo relacionado con los videojuegos. Sin embargo la idea seguía siendo muy genérica, por lo que se acudió a hablar con el profesor a cargo de la asignatura videojuegos, y el que ha sido el director del trabajo, Eduardo Mena. Su recomendación fue ampliar el videojuego realizado para las prácticas de la asignatura, ya que por la corta duración de la asignatura, se quedarían sin implementar muchas características, las cuales podrían ser interesantes. Tras conversarlo más, se concretó que el camino a seguir iba a ser el de incluirle al juego final servicios en red y una inteligencia artificial adaptativa. Una de las razones por la que se eligió esa elección, aparte del propio interés, fue que se consideró que abarcaba suficientes temas como para demostrar que se tenía conocimiento de un amplio abanico de conceptos que se consideran indispensables, y a su vez también incluía un aspecto (el aprendizaje automático por refuerzo) de la especialidad de computación que se había cursado, el cual como tal no se había practicado en las clases, implicando también cierta investigación propia.

Decidido que el camino iba a ser un trabajo dirigido por un profesor de la universidad (Eduardo Mena), con el objetivo básico de ampliar el videojuego de la asignatura mediante servicios en red y una inteligencia artificial adaptativa, el siguiente paso era establecer que hitos o requisitos debía cumplir el proyecto final. Las características establecidas como requisitos para el juego final fueron las siguientes:

- Proporcionar a los usuarios la posibilidad de crear cuentas que les representen como jugadores dentro de la comunidad del videojuego, y que guarden información de su interacción con esta, tanto con el juego en sí como con otros jugadores. Esto a su vez incluye características como la identificación del jugador o la recuperación de la cuenta.

- Incluir partidas en red que permitan jugar con otras personas sin la necesidad de estar ambas juntas en persona, procurando que ambos jugadores tengan la misma percepción de la partida. Proporcionando a su vez un sistema de emparejamiento automático.

- Permitir que los usuarios establezcan amistades entre sí, y que puedan interactuar, como sería el poder comunicarse mediante mensajes de texto a través

del propio juego, o poder invitarse a jugar una partida directamente.

- Dar al jugador la posibilidad de ver cierta información como serían número de partidas jugadas o los puntos clasificatorios, tanto propios como de sus amigos, a través de un perfil de usuario.

- Presentar al jugador un modo competitivo basado en una clasificación por puntos, que les permita conocer su posición en base a su habilidad respecto del resto de jugadores, incentivando la rivalidad entre ellos.

- Incluir un modo para enfrentarse a las inteligencias artificiales adaptativas, tanto la personal como la global.

Aparte de lo nombrado, también se concretó que habría que entrenar una inteligencia artificial por cuenta propia, con enfrentamientos contra las inteligencias artificiales originales, y analizar los resultados para demostrar que verdaderamente mejora.

1.3. Estructura del documento

El contenido de este documento se divide en un total de 5 capítulos siguiendo la siguiente estructura, además del presente capítulo de introducción:

- El capítulo 2 presenta el análisis hecho a partir de los hitos establecidos, y la arquitectura resultante de dicho análisis.

- El capítulo 3 explica el proceso de desarrollo de la inteligencia artificial adaptativa, desde la investigación de las posibilidades de diseño de la misma, hasta la explicación del algoritmo elegido y su entrenamiento y pruebas.

- El capítulo 4 desarrolla el proceso de implementación de los servicios en red, tanto en servidor como cliente, desde la elección de la base de datos, al funcionamiento de las partidas en red.

- El capítulo 5 está dedicado a mostrar conclusiones obtenidas a partir del desarrollo, mostrar brevemente la gestión del tiempo invertido, así como nombrar diversas posibles ampliaciones y mejoras, y expresar tanto técnica como personalmente una autoevaluación del trabajo realizado.

Al final se incluye una serie de anexos donde se incluye una información complementaría a la memoria principal.

Capítulo 2

Diseño de la arquitectura

Tras analizar los objetivos, y las características que estos requerían de forma implícita, o de forma indirecta, la conclusión fue que todo requería de una arquitectura distribuida [1] de modelo cliente-servidor [2]. Dicho modelo se adapta perfectamente a los requisitos que se exigen para poder cumplir los objetivos, ya que permite la centralización de la información (cuentas de usuario, las inteligencias artificiales, partidas jugadas, etc.) sin necesidad de que el usuario se tenga que preocupar de perder nada al eliminar el juego, en caso de que en un futuro quisiese volver a jugar. Otra de las mayores ventajas de este modelo, es que permite que se le puedan proporcionar al usuario todos los servicios nombrados, sin necesidad de que conozca ningún tipo de información de bajo nivel (IP pública por ejemplo) del resto de jugadores, ya que el servidor se encarga de hacer de intermediario entre los usuarios para la mayoría de interacciones.

Puesto que uno de los requisitos era que los servicios red se proporcionasen de forma segura, se decidió que los clientes establecerían con el servidor una conexión TCP [2] cifrada mediante certificados autofirmados [3]. Mientras que para las partidas en red se establecería una conexión UDP [2] directa entre los clientes, con el objetivo de reducir al mínimo los retrasos por temas de red, de forma que ambos usuarios tengan la mejor percepción posible de la partida. Si la comunicación entre clientes fuese TCP, o con el servidor como intermediario en vez de directa, se podrían dar muchos retardos que afectases a la experiencia del usuario. La arquitectura final del sistema se ve en la figura 2.1.

2.1. Arquitectura del cliente

El cliente debía poder proporcionar múltiples características propias de cualquier videojuego en red:

– Como es obvio, debía permitirle al usuario ver por pantalla el juego en sí, desde

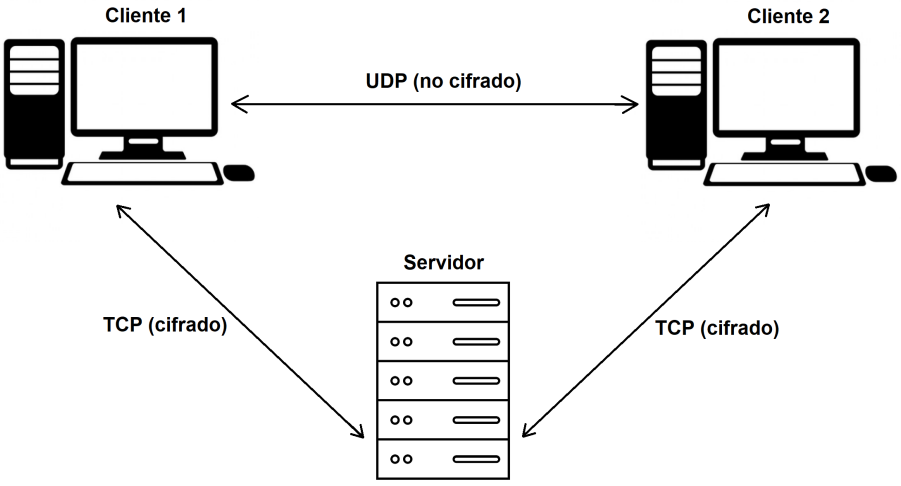


Figura 2.1: Arquitectura básica del sistema.

la interfaz de navegación hasta las peleas en sí, y permitirle interaccionar con dicha visualización.

– Se requerían servicios en red, para lo que se necesitaría poder comunicarse con el servidor y con el resto de clientes, pues las partidas en red se jugarían mediante comunicaciones directa entre ellos.

– Poder jugar tanto partidas en local como en red era una de las características clave, para lo que debía haber algún encargado de administrar toda la lógica que ello implica.

– Se debía poder jugar contra dos tipos de inteligencias artificiales, tanto una muy simple basada en probabilidades, como una adaptativa más compleja, siendo muy diferentes entre si, por lo que cada una se debía gestionar por separado.

– Como en todo *software*, era necesario algún administrador dentro del juego que se encargue de gestionar lo principal del juego y de unificar todos los procesos y fragmentos de este.

Por tanto, dentro del cliente (figura 2.2) encontraríamos un total de 7 componentes, siendo el principal el 'AdministradorDePantalla', pues es el encargado de iniciar el juego en sí, y lo que es más importante, de mostrar todo por pantalla al usuario para permitirle interactuar y visualizar el juego en general.

Un fragmento muy importante es el 'ComunicaciónUDPyTCP', pues es el encargado del establecimiento de todas las comunicaciones a través de la red, lo cual es un pilar imprescindible en el proyecto.

El administrador de pantalla crea el componente 'GestorDelJuego', el cual se encarga de la gestión y navegación de la interfaz, de la creación de las peleas y la comunicación con el servidor (para lo que depende de 'ComunicaciónUDPyTCP').

Para las partidas locales, el gestor del juego usa el componente 'GestorPeleasLocal' para que gestione todo lo relacionado con la pelea, tanto lo visual como la lógica de la pelea en sí. Este componente es capaz de permitir tanto peleas entre dos jugadores, como peleas contra la IA. Para este último caso, el gestor de la pelea depende de los componentes que controlan las IAs, tanto la simple ('IAbásica'), como la IA adaptativa ('IAadaptativa'), que se encargan de indicas las instrucciones a ejecutar por el personaje controlado por la máquina.

Para las partidas en red, el gestor del juego dependería del 'GestorPeleasEnRed', el cual gestiona no solo lo visual y la lógica de la pelea, sino también se ocupa de comunicar a los dos jugadores a través de la red, y asegurarse de que ambos tienen la misma percepción de la pelea (depende del componente 'ComunicaciónUDPyTCP').

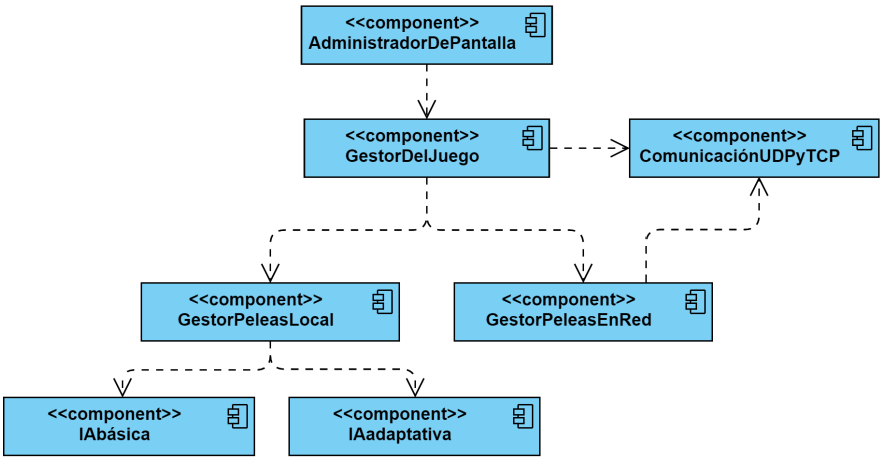


Figura 2.2: Arquitectura del cliente.

2.2. Arquitectura del servidor

El servidor, para que el sistema cumpliese los requisitos, debía cubrir ciertas necesidades:

- Toda la información, desde usuarios a registro de partidas, debía estar correctamente guardada y con un fácil acceso para la recuperación de la misma.
- El servidor debía poder ser capaz de proporcionar todos los servicios en red exigidos, y de administrar toda la concurrencia que pueda implicar el tener múltiples usuarios.

– Si se deseaba poder proveer servicios a los clientes, como es obvio, sería necesaria alguna estructura que permitiese la comunicación del servidor con los clientes.

– Cada comunicación con los clientes debería estar perfectamente controlada, de forma que no hubiese posibilidad de confundir usuarios dentro del propio servidor.

Por tanto, dentro del servidor (figura 2.3) encontraríamos un total de 6 componentes, siendo el más relevante el componente 'Principal', pues es el encargado de crear las estructuras de gestión del servidor ('GestorServicios'), de recibir todas las peticiones de conexión de los usuarios, de crear los procesos que se encargan de gestionar las comunicaciones con cada uno de ellos. Para el establecimiento de las comunicaciones con los clientes, depende de 'ComunicaciónTCP', componente que se encarga de la creación del socket principal del servidor, y de permitir comunicarse con los clientes a través de una conexión TCP cifrada mediante certificados autofirmados.

El componente 'GestorServicios' ya nombrado, se encarga de gestionar todo lo relacionado con los servicios en red como tal, es decir, de guardar los usuarios en la base de datos ('BASE DE DATOS SQL') o de emparejar dos usuarios para una partida en red, entre otros muchos servicios. Para proporcionarlos, este componente depende a su vez tanto de 'ComunicaciónTCP' para algunos servicios (principalmente notificaciones), como de 'ComunicaciónUDP' para el emparejamiento de usuarios para partidas en red, el cual se encarga únicamente de establecer una conexión temporal con los clientes antes de cada partida, para recabar cierta información necesaria.

Finalmente, el componente 'GestorComunicaciónUsuario', se encarga de recibir las peticiones del cliente, procesarlas, ejecutar lo que corresponda en 'GestorServicios', y notificarle al cliente el resultado. También se encarga de cerrar automáticamente todo lo relacionado con el cliente en caso de que este se desconectase. Obviamente depende de 'ComunicaciónTCP' para comunicarse con el usuario.

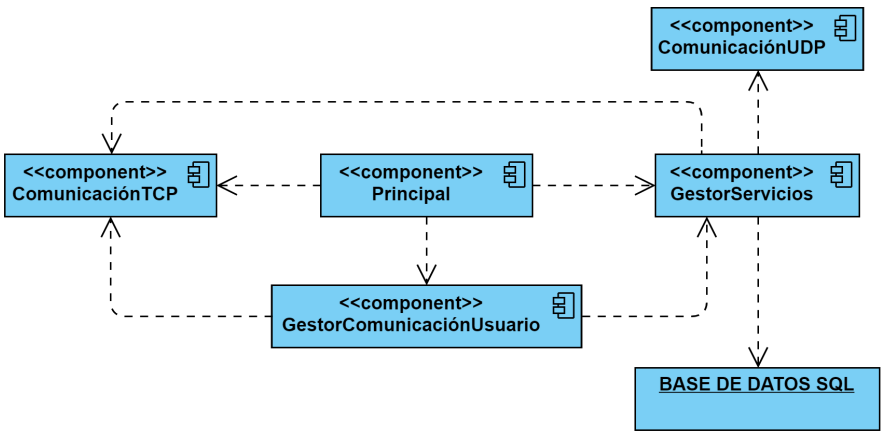


Figura 2.3: Arquitectura del servidor.

Capítulo 3

Inteligencia artificial adaptativa

Este capítulo tratará de explicar el proceso de desarrollo de una de las dos aspectos básicos del proyecto, la inteligencia artificial adaptativa, desde la investigación de opciones, hasta el entrenamiento y pruebas del método escogido. Para este caso en concreto, lo que hay que valorar ante todo es que sea un método que aprenda lo suficientemente rápido para que el usuario lo pueda llegar a percibir sin necesitar miles de partidas.

Antes de buscar soluciones, se definió el problema que se quiere resolver, es decir, concretar a qué pregunta la IA debe dar una respuesta, en este caso, la mayor pregunta a responder sería el cómo ganar la partida. Sin embargo también se debía plantear la pregunta, cuando gana, qué tan bien gana, o dicho de otra forma, con qué margen de diferencia gana. Si únicamente se valora si ha ganado o no, es muy probable que se den como buenas algunas decisiones erróneas tomadas durante la partida, y viceversa. Esto haría que el aprendizaje fuese mucho más lento, pues sería difícil distinguir si una acción en un estado es buena o no, pues depende del resultado de la partida y no en sí de la propia acción, haciendo que fuesen necesarias muchas más partidas para converger.

3.1. Documentación y análisis de opciones

El primer paso fue pensar en opciones para resolver dicho problema, basándose todas ellas en técnicas de aprendizaje automático [4]. En un primer acercamiento, se intentó replantear el problema desde diversos puntos de vista para intentar adaptarlos a algunos métodos de aprendizaje automáticos vistos en la asignatura de Aprendizaje automático. De estos planteamientos, los principales serían:

– Problema de regresión lineal[1] [4]: se puede plantear que dados un estado concreto de la partida y una acción, se desea estimar el beneficio que se podría obtener, es

[1] `https://es.wikipedia.org/wiki/Regresion_lineal`

decir, convertir la información de la partida junto con la acción en las variables en las variables de la función de regresión, a partir de las cuales se quiere estimar un beneficio (este beneficio sería calculado de forma automática en base a unas reglas preestablecidas, por ejemplo basado en el daño realizado). El problema de este planteamiento es que por la propia naturaleza del entorno, en el que el una misma acción no te va a llevar siempre al mismo estado resultado, es muy difícil que la regresión acabe generando un buen modelo. Sin embargo, el verdadero problema, sería que este método solo valoraría el beneficio inmediato, es decir, no miraría a futuro, lo cual en los videojuegos en general suele ser algo crucial a la hora de jugar.

- Problema de clasificación[2] [4]: una forma de ver el problema sería pensar que los estados los podemos clasificar en acciones, dicho de otra forma, a un conjunto de estados le corresponde la salida (categorización) de una acción concreta. Por ejemplo, en un estado concreto se ejecuta una acción, se analiza el resultado, y si es positivo, se añade al conjunto de datos de entrenamiento una fila con el estado como entrada y la acción como salida (esta clasificación se haría automáticamente bajo unos criterios preestablecidos), si el resultado fuera negativo, se descartaría puesto que no se puede saber a que otra clase (acción) pertenecería. De nuevo los problemas son que la clasificación difícilmente modelaría correctamente la incertidumbre, y que tampoco miraría a futuro puesto que solo tiene en cuenta la situación actual. Además, otro inconveniente sería el hecho de que se pierde información al descartar las acciones que no tuvieron un resultado positivo, haciendo mucho más largo el entrenamiento.

Tras descartar estas opciones, se decidió investigar acerca del aprendizaje por refuerzo[3] [4], concretamente el Aprendizaje por refuerzo profundo [4], el *Q-Learning* [5] y el actor-crítico[4] [6].

Los tres algoritmos pertenecen al aprendizaje por refuerzo, el cual se resumiría de la siguiente forma: un agente (la IA) se encuentra en un entorno o ambiente (una partida contra otro personaje) con el que puede interactuar a través de unas acciones definidas. Dichas acciones tienen un efecto sobre el entorno, y el estado resultante de la acción puede ser positivo o negativo, en base a lo cual se le da o se le quita una recompensa al agente. En todos los casos el objetivo del aprendizaje es que el agente maximice la recompensa total y están pensados para mejorar progresivamente.

[2] `https://es.wikipedia.org/wiki/Aprendizaje_automatico#Tecnicas_de_clasificacion`
[3] `https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo`
[4] `https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f`

Los tres algoritmos resuelven el problema de la IA en sí, proporcionando métodos de aprendizaje que aprenden de la experiencia, y a la hora de tomar decisiones valoran las recompensas futuras. Sin embargo, los algoritmos del actor-crítico y del Aprendizaje por refuerzo profundo están pensados para entornos en los que se pueda dejar a la IA entrenando y calculando sin restricciones de tiempo ni de consumo computacional, lo cual no encaja bien con el planteamiento del proyecto.

De entre los tres algoritmos, el menos costoso computacionalmente es el *Q-Learning*, por lo que se decidió usar este algoritmo.

3.2. Algoritmo Q-Learning

El algoritmo elegido para el proyecto fue el *Q-Learning*, debido a su ligereza computacional (puesto que su costo es tener una matriz de números reales en memoria y actualizarla tras cada acción mediante una fórmula), su capacidad de aprender a partir de la experiencia y su optimalidad basada en la actualización continua[5].

El algoritmo en sí, se resumiría en la actualización continua de una tabla de pares estado-acción, con las recompensas recibidas durante el entrenamiento por la ejecución de una acción en un estado concreto, siendo lo interesante del algoritmo la política de actualización de la tabla. La política de actualización destaca por su tendencia a converger hacia la solución óptima (gracias a la ecuación de Bellman[6]) y por incluir un factor de paciencia, que representa la valoración que da a las recompensas futuras frente a las presentes, es decir, toma decisiones mirando a futuro.
El algoritmo sería el siguiente:

Inicializar la tabla Q(s,a) aleatoriamente;
while *queden episodios* **do**
  do
    Elige una acción a ejecutar en base a la política ϵ-*greedy*;
      Con probabilidad ϵ elige acción aleatoria;
      y con probabilidad 1-ϵ elige acción $a = \text{máx}_{a \in A(s_t)}\, Q(s_t, a)$;
    Ejecutar a y esperar a recibir la recompensa r y llegar al estado s_{t+1}
    $Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[(r + \beta\, \text{máx}_{a2 \in A(s_{t+1})}\, Q(s_{t+1}, a2) - Q(s_t, a)]$;
    $s_t \leftarrow s_{t+1}$ Actualizar ϵ según se haya decidido;
  while s_t *no sea terminal*;
end

Algoritmo 1: Q-Learning

[5]Las IAs nunca dejan de aprender, pero si se quisiese guardar el estado de una IA, simplemente sería necesario guardar el fichero que contiene la tabla

[6]`https://es.wikipedia.org/wiki/Ecuacion_de_Bellman`

Señalar que este algoritmo necesitaba de un sistema de evaluación de recompensas, el cual indicaría cuán buena o mala es una acción en un estado basándose en el estado resultado, y de un sistema de representación de los estados, el cual constituiría la información del entorno que el agente percibiría. En el anexo A se muestra información más detallada del algoritmo, junto con algunos fundamentos necesarios para su completa conprensión.

Respecto de la implementación propia, destacar que con la intención de que las decisiones de la IA no fueran tan predecibles y a su vez fomentar ligeramente la exploración sin empeorar los resultados, en los estados que se hubiesen probado cuatro acciones o más, con un 0.8 de probabilidades se seleccionaría la acción con más recompensa estimada, y con un 0.2 se seleccionaría de forma aleatoria una de las 3 acciones subóptimas.

3.3. Sistema de estados y recompensas

Dos aspectos cruciales del algoritmo son el sistema de estados y el de recompensas, porque son los que marcan directamente cómo va a aprender. Por tanto hay que entender y analizar el entorno en el que se va a mover el agente, para después poder tomar decisiones acerca de ambos sistemas.

El entorno con el que va a interactuar el agente son partidas de un videojuego de lucha en dos dimensiones, con el objetivo de ganarlas, este deberá a su vez ganar 2 de las 4 rondas máximas de las que puede constar una pelea. Para ganar una ronda, deberá reducir la vida del enemigo a 0 siendo la suya propia 1 o más. Durante una partida, la información disponible es la siguiente: la vida de los dos personajes (el del jugador y el de la IA), la posición de ambos personajes en el escenario en dos dimensiones, la orientación en la que están mirando ambos personajes, el movimiento que está realizando cada uno de los personajes, el tiempo restante de la ronda en marcha, el número de ronda que se está jugando, y el número de rondas ganadas por cada uno de los participantes en la partida.

Con dicha información en bruto el número de estados se dispara (del orden de 10^{17}), de forma que resulta imposible en la práctica usar el algoritmo. Por tanto, fue necesario decidir que información era verdaderamente imprescindible, y con qué precisión. Para lo cual se realizó un análisis, que se basó en establecer unos razonamientos algo más humanos. Los jugadores no conocen la información exacta de cada variable, e incluso omiten algunas, y aun así son capaces de aprender a jugar. Por tanto, se hizo una simplificación de la información, quedándose con la verdaderamente importante. La representación final del estado para el agente, contendrá la siguiente información: la

vida de los dos personajes, simplificada en 3 valores (mucha, media, o poca vida), la distancia entre los personajes, simplificada en 3 valores (cerca, lejos, o distancia media), si el personaje del agente está saltando o no, y el estado del personaje del jugador. El proceso de simplificación se pude ver en el anexo B.

Con esta simplificación se reduce enormemente el número de estados, pero se mantiene la información suficiente para que el agente tenga una percepción 'humana' del entorno. Se podría decir que se abandona la idea imposible de una IA perfecta (pues para que fuese perfecta debería usar toda la información), en aras de un objetivo realista y alcanzable.

Establecido el sistema de estados, el siguiente paso era el sistema de recompensas, es decir, la evaluación del resultado de una acción desde un estado inicial, el cual valorará como de bueno o malo ha sido el resultado de dicha acción. Para la definición del sistema de evaluación se procuró valorar las cosas desde un aspecto humano, es decir, comportamientos como atacar al aire cuando el enemigo está muy lejos no tiene sentido, salvo en situaciones concretas con por ejemplo ataques especiales de largo alcance, o como cubrirse de un ataque estando muy lejos. También se valorará el cubrirse contra ataques incluso pese a recibir daño (se le recompensará con parte del daño recibido como recompensa), ya que si recibe el ataque bloqueando no hay retroceso por golpe recibido, lo cual es posible que puede beneficiar la siguiente acción. Sin embargo, lo que más se evaluará es si el movimiento realizado ha reducido la vida de alguno de los jugadores, siendo una buena decisión por ejemplo cuando baja la vida del enemigo sin reducirse la propia, o cuando baja más la vida del enemigo que la propia. El algoritmo completo, se puede ver en el anexo C.

3.4. Refuerzo con regresión lineal

Uno de los problemas del algoritmo era el hecho de que en los momentos de tomar una elección de explotación, si la IA no había estado nunca en ese estado, lo mejor que podía hacer era tomar una elección aleatoria. Este problema es más relevante cuando entrena contra una persona que cuando realiza un entrenamiento independiente, pues el número de partidas que va a jugar el usuario ni se acercará al de un entrenamiento independiente, y aun así, este querrá poder apreciar resultados. Por tanto lo ideal sería que en situaciones nuevas, pudiera tomar decisiones a partir de situaciones similares.

Para parle a la IA esa capacidad, se recurrió al uso de la regresión lineal para deducir la recompensa estimada de una acción en un estado, a partir de todos pares estado-acción visitados. La elección de la regresión lineal como algoritmo de apoyo para el *Q-Learning* se basa en que al fin y al cabo, lo que hace este es estimar un valor

de salida en base a unas variables de la partida, lo que encaja perfectamente con la definición de la propia regresión. Concretamente, el uso que se le dio a la regresión fue, que en caso de visitar un estado en el cuál no se había estado nunca, estimase la recompensa de cada acción para el estado a partir del modelo calculado (este se calcularía a partir de los estados visitados anteriormente y las acciones ejecutadas) y tomase la decisión en base a esas recompensas predichas. Es sencillo ver que una decisión basada en la información ya conocida (aunque las estimaciones contengan un margen de error grande), siempre va a ser mejor en promedio que tomar una elección aleatoria sin ningún tipo de base.

Previamente, se expusieron razones de porqué no se elegía dicho algoritmo para resolver el problema, por lo que puede resultar extraño que vuelva a surgir. Por ello, recalcar que la regresión no se está usando para resolver el problema en sí, sino que meramente se usa como apoyo para 'rellenar' la tabla de la función valor y darle la capacidad de tomar mejores decisiones, a medida que el entrenamiento avanza, en la explotación de estados no visitados. Señalar además, que en ningún momento se usa la regresión para actualizar la tabla de la función valor, únicamente se usa para elegir una acción. A continuación, se explica el porqué las desventajas que se nombraron no resultan problemáticas con el uso que se le da:

- En cuanto al coste computacional, decir que en el uso que se le da tiene un límite marcado por la propia tabla en base al número de estados y acciones, de forma que no se puede disparar como si lo usásemos para resolver el problema en sí. Además, como este coste viene acotado por el tamaño de la tabla, y se conoce que su coste temporal es pequeño, a la hora de su implementación como servicio, basta con que el juego cliente calcule la regresión en local antes de empezar la partida, sin necesidad de que el servidor realice estos cálculos.

- El hecho de que no modela bien la incertidumbre de este entorno sigue presente, pero al no analizar directamente el entorno, sino una tabla la cual ya contiene tratada en cierto modo esa incertidumbre, lo permite generar un modelo algo mejor, con resultados más que aceptables para el uso que se le da.

- El problema de que no evaluaría las recompensas futuras viene de nuevo solucionado por el hecho de que estima a partir de valores de la tabla, y no directamente del entorno. Esto se debe a que los valores de la tabla ya contienen el tratamiento de las recompensas futuras, es decir, esos valores ya tienen en cuenta tanto la recompensa inmediata como la futura, de forma que si la regresión aprende a partir de esos valores, de forma indirecta está valorando también las recompensas futuras.

Señalar que, concretamente, la regresión implementada sería una regresión lineal multivariable polinómica, con ajuste del exponente (un hiperparámetro) mediante k-fold cross-validation[7]. Para la regresión lineal en sí se hizo uso de la librería Weka [8], pero el *k-fold cross-validation* hubo que implementarlo.

A modo de ejemplo para mostrar cómo de buenas serían sus predicciones en el caso de una tabla muy entrenada, decir que en un entrenamiento con 13114 estados visitados tras 9000 partidas D.2, en las que se habrán revisitado muchos de ellos muchas veces, se conseguía un error absoluto medio de 68.8444 %. Incluso tras tantas partidas, el error relativo absoluto es muy alto, sin embargo, estos resultados distando mucho de ser perfectos, resultan más que suficientes para el uso que se le da (servir de guía para no tomar una mala decisión siempre que se encuentre con una situación desconocida).

3.5. Desarrollo e integración en el entorno

Previamente al desarrollo de la IA en sí, se desarrollaron unas estructuras, que contendrían la información de interés analítico acerca de todas las partidas jugadas por la IA, y se integraron en el entorno del videojuego, para posteriormente poder analizar el comportamiento y la evolución de la IA. Para analizar el comportamiento de la IA, se implementó una herramienta capaz de generar las gráficas a partir de las estadísticas, y poder ver su evolución. Decir que las gráficas muestran la tendencia a través de una recta de regresión.

Para empezar el desarrollo de la IA, el primer paso fue la definición de las estructuras de los estados, junto con el procedimiento que permite pasar de un estado a un único número que actúa como identificador del estado (debido a la simplificación de la información de los estados, muchos estados muy similares serán representados por un mismo identificador).

El siguiente paso fue crear la estructura del agente, que contuviese la tabla en sí, el sistema de actualización de la tabla y el método de elección de la acción en base al valor ϵ que se le asigne y a la probabilidad de elegir subóptima. Aparte de esas funcionalidades básicas, se le añadieron otras como registrar qué estados se han visitado, registrar todas las transiciones del entrenamiento, escribir la tabla, el registro de estados visitados y el de las transiciones, cargar la tabla y los estados visitados, y permitir actualizar la tabla a partir de un registro de entrenamiento.

Decir que el permitir actualizar una tabla a partir de un registro de entrenamiento, es especialmente relevante pues se implementase el jugar contra la IA global, no sería

[7] `https://es.wikipedia.org/wiki/Validacion_cruzada`
[8] `https://www.cs.waikato.ac.nz/ml/weka/`

extraño que más de una persona jugase a la vez contra la IA, generando cada uno su propia tabla actualizada (las cuales no se pueden juntar), por lo que la solución sería que a la tabla guardada en el servidor se le aplicasen todas las transiciones de cada partida de forma secuencial, consiguiendo generar una tabla unificada que ha aprendido de todas las partidas simultáneas

Y a continuación se introdujo el controlador de la IA para el agente, el cual era el encargado de permitirle interactuar con el medio. El agente durante toda la partida, extrae la información del entorno en un momento concreto y le pregunta al agente qué acción quiere ejecutar, ejecuta dicha acción, espera a que termine de ejecutarse la acción, y cuanto haya terminado notifica el resultado al agente, tras ello, repite. Decir que se presentaron diversos problemas de sincronización, como dar por terminadas acciones antes de que lo hubiesen hecho realmente, o como dar la recompensa de una acción a la acción siguiente, los cuales implicaron bastante tiempo de depuración.

Tras depurar todos los errores y problemáticas de la integración del algoritmo, y verificar que gestionaba bien las transiciones y actualizaba correctamente la tabla, se pasó a ampliar el algoritmo mediante la regresión lineal (multivariable polinómica con *k-fold cross-validation*) como apoyo. Obviamente, la regresión se integró en el agente, puesto que era el encargado de elegir las acciones a ejecutar. El modelo se generaría antes de la partida si no existiese uno, y normalmente se actualizaría el modelo tras cada partida.

3.6. Entrenamiento, pruebas y análisis de resultados

Como se esperaba no disponer de suficientes usuarios como para jugar miles de partidas, se desarrolló un modo de entrenamiento independiente para poder al menos demostrar que la IA de verdad mejoraba con el tiempo. Para este modo entrenamiento, la IA controlada por el agente se enfrentaría en peleas consecutivas contra las IAs desarrolladas para la asignatura Videojuegos y aprendería de ellas. Para el entrenamiento el ϵ se reducía progresivamente a lo largo de 3000 partidas, estableciéndose como ϵ mínimo 0.05. Este modo de esntrenamiento permitía entrenar tanto con el uso o no de la regresión, y tanto contra un mismo personaje y una misma IA fijos, o contra IAs y personajes aleatorios. En caso de que se usase la regresión, tras cada partida, se actualizaba el modelo de la regresión para que lo usase en la siguiente partida.

Aclarar que las IAs (desarrolladas para las prácticas de Videojuegos) contra las que se realizaban los entrenamientos distaban mucho de ser unas buenas IAs, por lo

que prácticamente se gana con movimientos aleatorios. Sin embargo, igualmente hay suficiente margen de mejora en el cómo gana, como para que se pueda apreciar una mejoría.
Se hicieron un total de cinco tipos de pruebas (todas las pruebas aparecen de forma significativamente más detallada en el anexo D):

– Un entrenamiento de 9000 partidas (en unas 170 horas) contra un mismo enemigo (Terry en nivel difícil) para mostrar las capacidades de aprendizaje, pese a que como se ve en la figura D.1a gana prácticamente siempre desde el comienzo (anexo D.1).

  En la figura 3.1 se puede apreciar claramente como todas las métricas de la IA mejoran considerablemente, destacando la vida media al final de la partida. También se percibe claramente, como la mejora viene marcada por el decrecimiento de ϵ. Señalar que ϵ se reduce de la partida 1 a la 3000.

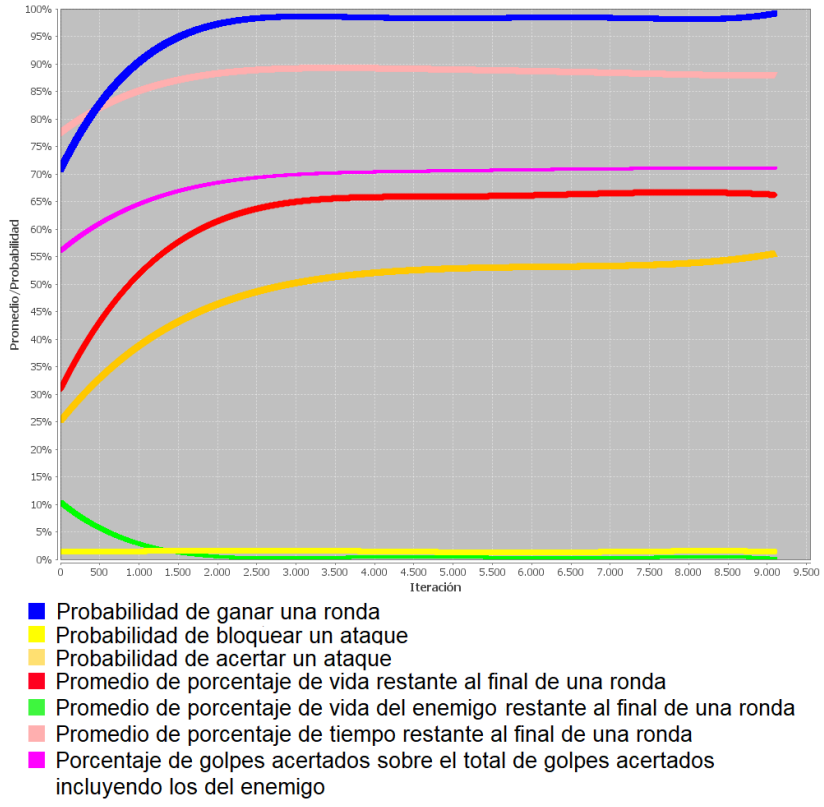
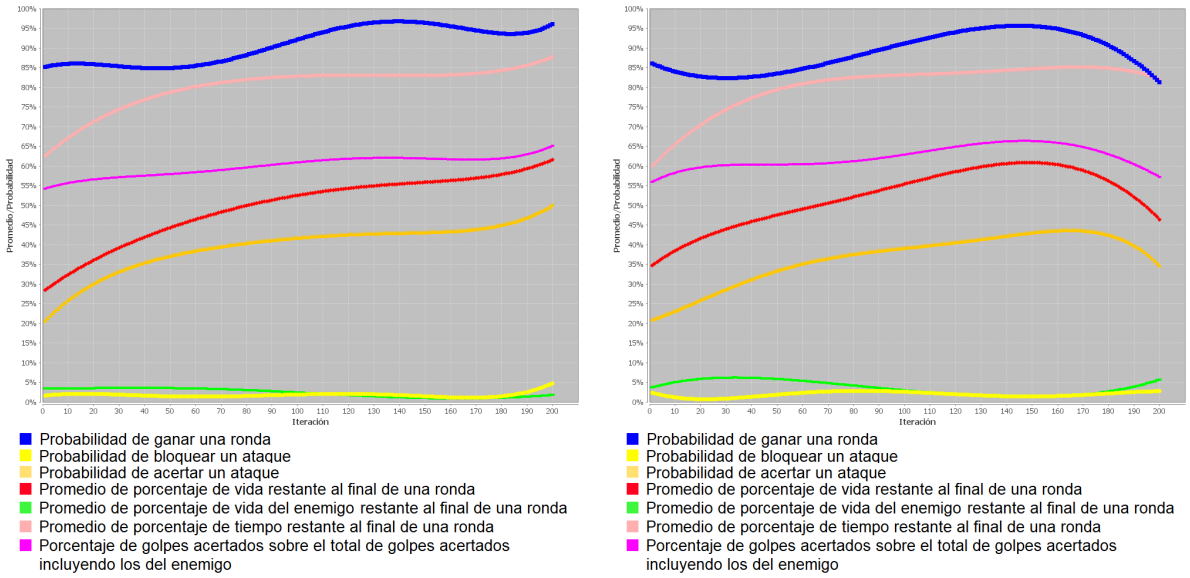


Figura 3.1: Métricas del entrenamiento de 9000 partidas contra Terry en nivel difícil, reduciéndose ϵ a lo largo de las primeras 3000.

– Una comparativa entre IAs entrenadas usando y sin usar regresión lineal para mostrar su efecto, tanto en una situación de un largo entrenamiento (5000 partidas), como en un caso de un corto entrenamiento (200 partidas) (anexo D.2).

En un largo entrenamiento en el que ϵ se reduce muy lentamente, la regresión lineal no marca apenas diferencia, pues cuando se empieza a explotar con alta frecuencia, ya se han visitado la mayoría de estados. Sin embargo, como se ve en la figura 3.2, en una situación en la que se reduce más rápidamente, la mejora es más constante obteniendo finalmente mejores resultados. Señalar que ϵ se reduce de la partida 1 a la 100.



(a) (b)

Figura 3.2: Métricas del entrenamiento con IA usando regresión (3.2a) y sin usar regresión (3.2b) en un entrenamiento de 200 partidas contra rivales aleatorios, reduciéndose ϵ a lo largo de las primeras 100.

– Un enfrentamiento entre dos IAs entrenadas, una contra todos los personajes y otra contra uno solo (Terry en difícil), para mostrar la evolución de ambas, tras haber terminado el entrenamiento base, contra un enemigo digno (anexo D.3).

  En la figura 3.3a se aprecia como desde el comienzo tiene ventaja la IA que entreno contra un único personaje (el que usan ambos) pues acumula un mayor número de victorias, pero en todo momento ambas mejoran y se adaptan (es lo que indican las oscilaciones), y la que entrenó contra todos los personajes no se queda atrás, suponiéndole siempre un reto y ganandole con cierta frecuencia.

– Una comparativa de IAs entrenadas, una contra todos los personajes y otra contra uno solo (Terry en difícil), contra todos los personajes (anexo D.4), que muestra como la primera las líneas se mantienen prácticamente rectas pues ha entrenado contra ellas, y la segunda muestra más oscilaciones pues solo conocía un personaje y se ha tenido que adaptar ligeramente.

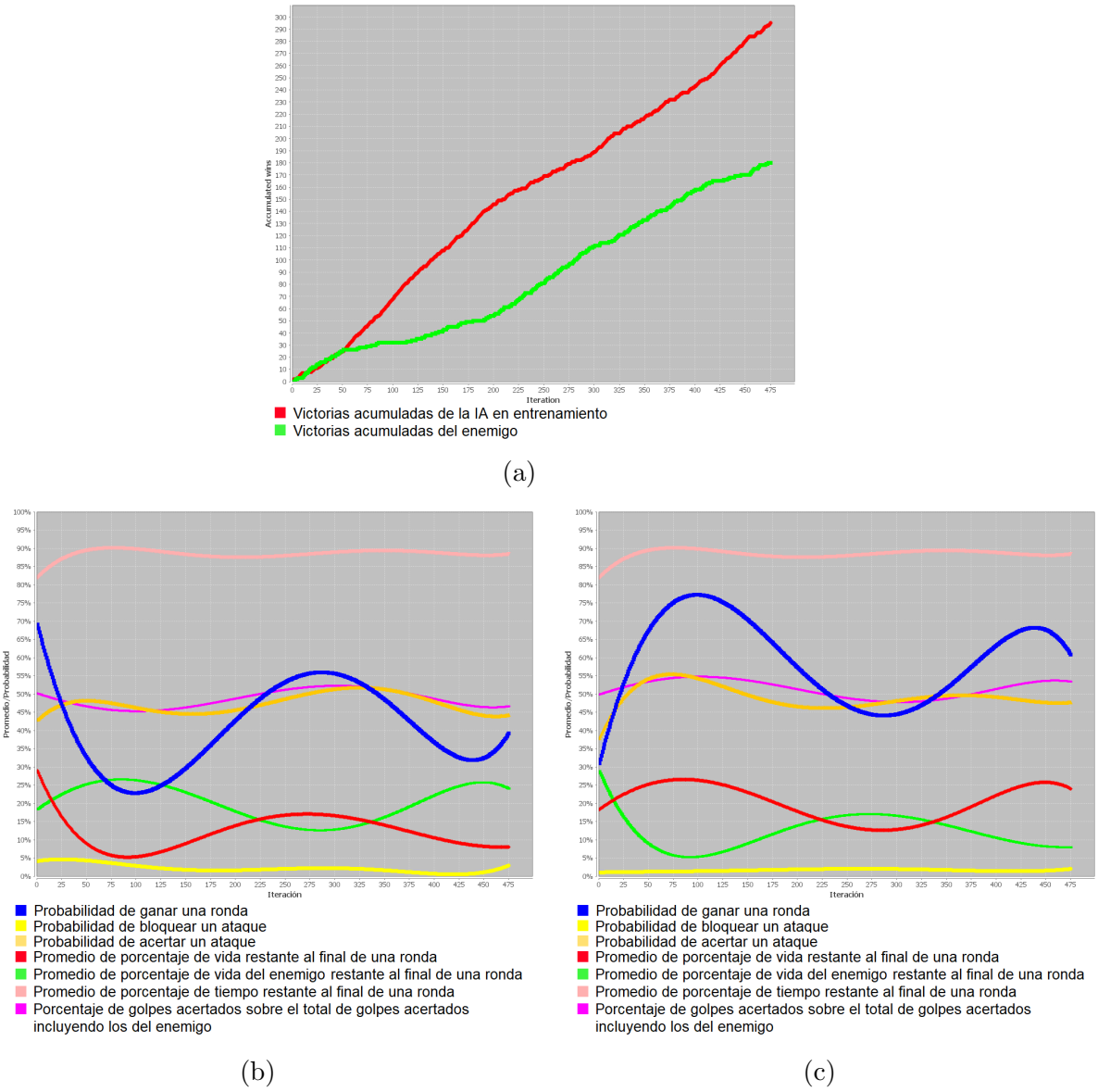


Figura 3.3: Victorias acumuladas (3.3a) y métricas (3.3b) de la IA entrenada contra enemigos aleatorios, y métricas (3.3c) de la IA entrenada contra Terry en difícil.

– Una prueba de la IA entrenada contra todos los personajes enfrentándose a seis humanos (anexo D.5), mostrando que pese a haber entrenado contra unas IAs muy simples, suponen un gran reto para los jugadores, consiguiendo entre todos apenas dos victorias entre decenas de partidas. En la figura 3.4 se puede apreciar la media de los resultados de 6 personas en sus 22 primeras partidas, viéndose, que pese a no ser tan buenos resultados como contra las IAs básicas, siguen siendo muy buenos resultados.

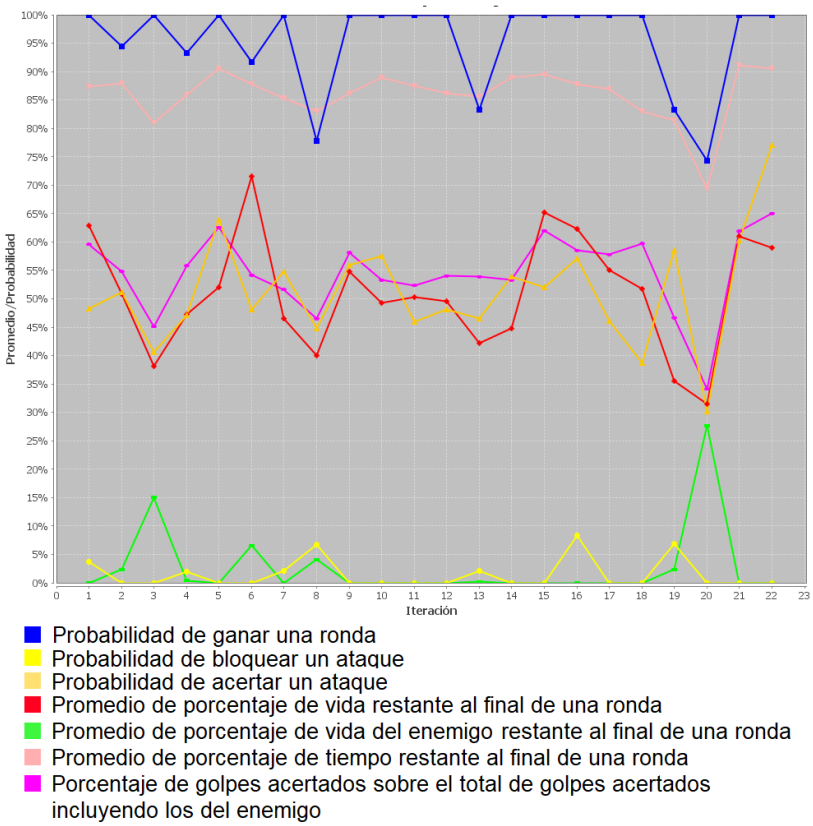


Figura 3.4: Resultados medios de 6 personas contra la IA entrenada contra todos los personajes en sus primeras 22 partidas.

Capítulo 4

Juego en línea

El segundo pilar de este proyecto eran los servicios en red, siendo algunos de estos la gestión de usuarios, la posibilidad de jugar partidas en red y relaciones de amistad, entre otros. Para esto, como ya se explicó al principio del documento, se tomó la decisión de hacer una arquitectura cliente-servidor centralizado, estableciendo entre los clientes y el servidor conexiones TCP [2] cifradas mediante certificados, y comunicaciones UDP [2] sin cifrar cuando se comuniquen dos usuarios durante una partida.

Nombrar que para todo el desarrollo y depuración se desplegaba el servidor localmente en el entorno de IntelliJ Idea abriendo los puertos correspondientes del *router*, y para el despliegue final se usaron los servicios de Google Cloud. Concretamente, el despliegue final, se usó una instancia de máquina virtual Debian de Gloogle Cloud, con la configuración mínima, a la cual se le añadieron al cortafuegos las dos reglas necesarias para abrir los puertos necesarios, y se instaló el gestor de bases de datos.

4.1. Base de datos

Uno de los cimientos de todo servidor es una base de datos en la que guardar toda la información de los usuarios e interacciones, tanto entre ellos como con el propio servidor. Para este proyecto se decidió crear una base de datos relacional [7] sobre el gestor PostgreSQL, la cual se basaba en el modelo entidad relación que se puede ver en la figura 4.1.

En cuanto a la implementación destacar que se hizo uso del mapeo objeto-relacional[1] [7], concretamente con el estándar JPA[2] [8] usando las bibliotecas de Hibernate[3]. En el anexo E se explica más detalladamente el proceso de creación de la base de datos.

[1] `https://es.wikipedia.org/wiki/Asignacion_objeto-relacional`
[2] `https://es.wikipedia.org/wiki/Java_Persistence_API`
[3] `https://hibernate.org/`

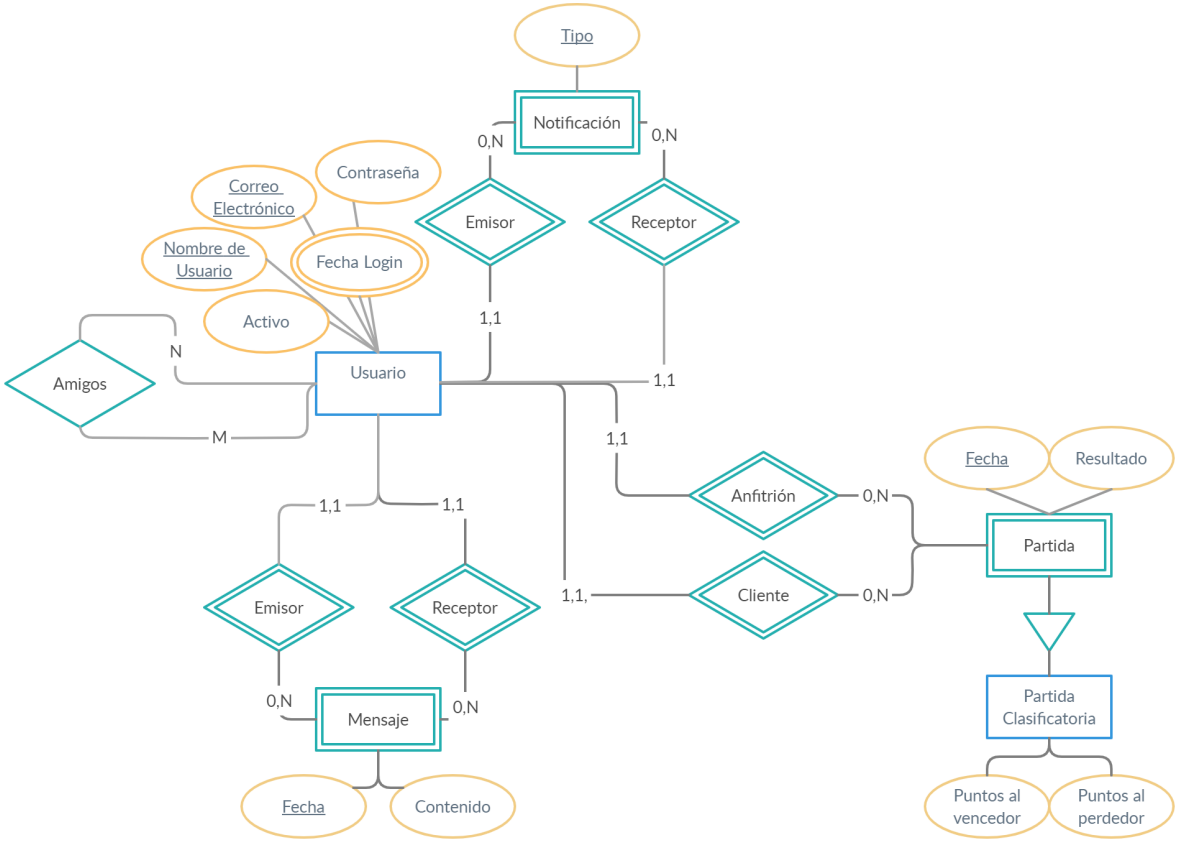


Figura 4.1: Modelo entidad relación de la base de datos.

4.2. Estructura de conexión

Establecida los cimientos del servidor con la creación de la base de datos, el siguiente paso fue crear una estructura de conexión que permitiese que los clientes se pudiesen comunicar con el servidor, y entre sí durante las partidas, la cual se basaba en el uso de las clases `Socket` [4], `ServerSocket` [5] y `DatagramSocket` [6] (en los anexos F.1 y F.2 se muestran unos ejemplos del uso de estas clases, en los que se basan la impletentación final). La decisión de hacer una estructura propia, se debió que a que no se encontró ninguna biblioteca para trabajar con UDP que tuviese cierto reconocimiento. En busca de la unicidad en la comunicación, independientemente de si fuese TCP o UDP, se decidió que lo ideal iba a ser una estructura propia adaptada a las necesidades del proyecto. Las necesidades a cubrir por dicha estructura, y que se introdujeron, eran simples:

– La comunicación entre cliente y servidor debía permitir enviar objetos complejos, y la comunicación entre clientes permitir mandar cadenas de texto.

[4]https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
[5]https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html
[6]https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html

- Todos los mensajes debían estar representados por un identificador numérico, por el cual se podrían recuperar cuando fueran recibidos. En cierto modo, se podría considerar que los identificadores establecerían un canal dentro de la comunicación.
- La comunicación TCP con el servidor debía estar cifrada.
- Los mensajes podían o no solicitar una confirmación de recepción.
- Debían proporcionar diversas funciones de utilidad, como sería mandar un mensaje y quedarse bloqueado esperando una respuesta, o como una función de recepción de mensajes que se quedase bloqueada hasta recibir un mensaje.

Aclarar que tanto para comunicación UDP como la TCP, la estructura crearía un hilo para recibir mensajes constantemente para que el resto de procesos puedan 'recibir' en base al identificador de forma no bloqueante, y a su vez respondería automáticamente a todos que requiriesen confirmación.

4.2.1. Comunicación UDP y problemas encontrados

Lo primero que se desarrolló de apartado de red, fue la partida entre dos jugadores, por lo que la primera parte de la estructura a desarrollar fue la comunicación UDP. Como ya se dijo, para la implementación de esta comunicación se hizo uso de la clase DatagramSocket, basándose en el ejemplo básico mostrado en el anexo F.1 (en el anexo G.1 se detalla alguna información acerca de la estructura). Sin embargo, pese a que aparentaba ser muy simple, surgieron muchos problemas inesperados.

Pese a que la comunicación UDP funcionaba correctamente en local y en principio la estructura aparentaba estar bien, al intentar usarla a través de la red, no funcionó. Durante horas (e incluso días) se intentó solucionar, incluso se creó un chat extremadamente simple que usaba la estructura, para depurar más fácilmente. Pero contra lo esperado, el chat solo incrementó la incertidumbre acerca del problema, pues el chat sí funcionaba, haciendo que se pensase que el problema estaba en otra parte, lo cual tampoco podía ser porque el juego en sí, ni siquiera recibía mensajes.

Finalmente se percibió un detalle, el chat funcionaba únicamente cuando ambos clientes habían mandado al menos un mensaje. El problema provenía por tanto, de que al establecer una comunicación UDP hay que enviar al menos un mensaje para iniciar el protocolo UDP en el puerto. En vistas de esto, para solucionar el problema simplemente se hizo que cuando se le indicase a la estructura el puerto destino, se enviase siempre un 'saludo' directamente al otro cliente para el cual se esperaría respuesta, iniciando así el puerto y confirmando el establecimiento de la comunicación por ambas partes.

Tras esto se consiguió hacer funcionar la estructura, consiguiendo poder jugar partidas en red. Sin embargo, surgió un nuevo problema, funcionaba con algunos usuarios, pero no todos. Tras días de intentos fallidos se descubrió acerca del protocolo `UPnP`[7] (es una opción del *router* que normalmente viene activa por defecto), el cual tiene como función solicitar abrir puertos de forma automática y autónoma de forma que el usuario no tenga que modificar la configuración de su *router* manualmente, y que se suele usar mucho para los videojuegos en red. Tenía sentido que fuese posible que en algunos casos no se pudiese abrir los puertos correctamente, por lo que se revisó que todos los usuarios de pruebas tenían dicha opción activa (todos la tenían activada), y se pasó a intentar usar dicho protocolo. Para hacer uso del protocolo se probaron diversas bibliotecas (Cling[8], WaifUPnP[9] y OhNet[10]), pero con ninguna se solucionó el problema, por lo que se concluyó que el problema no debía ser ese, y por tanto el protocolo UPnP no debía ser la solución al problema.

Tras otras tantas horas de depuración, se decidió hacer una prueba basada en el hecho de que la conexión TCP con el servidor sí funcionaba (tras muchas horas invertidas se hizo una pausa para avanzar con la conexión TCP), para la cual, se creo un servidor UDP con un puerto del *router* abierto para tráfico UDP (de forma que no era necesario mandar un mensaje para inicializarlo), y se intentó establecer una comunicación cliente-servidor con UDP. El resultado fue que el experimento funcionaba, se establecía la comunicación UDP correctamente.

Se depuró el servidor del experimento, hasta que finalmente se descubrió el problema. Pese a que en el cliente se iniciaba el *socket* en un puerto concreto, los paquetes que le llegaban al servidor indicaban en la cabecera que provenían de un puerto completamente distinto. Al ver esto, el problema estaba claro, los mensajes se mandaban al puerto en que en principio se debía estar escuchando, pero en verdad se estaba escuchando por otro, o mejor dicho el NAT, que estaba por medio y se comunicaba con el exterior en nombre del verdadero cliente, escuchaba (y enviaba) por ese otro puerto. El que solo funcionase a algunos cobró sentido rápidamente, pues era tan simple como que unos *router* tenían NAT y otros no (sin ser ellos conscientes de esto).

Una vez descubierto el problema, fue fácil encontrarle solución. Con un intermediario por medio entre los dos clientes (el servidor del experimento), con el que pudiesen iniciar una comunicación UDP siempre, y que este les notificase no solo la IP, sino también el puerto al que enviar del otro cliente, podrían establecer la comunicación

[7] `https://es.wikipedia.org/wiki/Universal_Plug_and_Play`
[8] `https://github.com/4thline/cling`
[9] `https://fdossena.com/?p=waifupnp/index.frag`
[10] `http://wiki.openhome.org/wiki/OhNet`

entre ambos pues ya conocerían el puerto del NAT al que enviar. Por tanto, para solucionar el problema, simplemente se creó en el servidor principal una estructura cuya única funcionalidad sería tener abierto constantemente un DatagramSocket esperando la recepción de algún paquete, únicamente para almacenar los pares IP/dirección - puerto. Posteriormente, el servidor podría notificar a los clientes el puerto de la dirección con la que desean comunicarse, y una vez informados ambos clientes, simplemente empezarían a enviar a la IP y puertos correspondientes. En la figura 4.2 se resume el protocolo seguido para el establecimiento de la primera conexión. Decir que al momento de escribir esto, investigando al respecto, se descubrió que esta técnica tenía nombre, y se conocía como Perforación de agujeros UDP [11] [9].

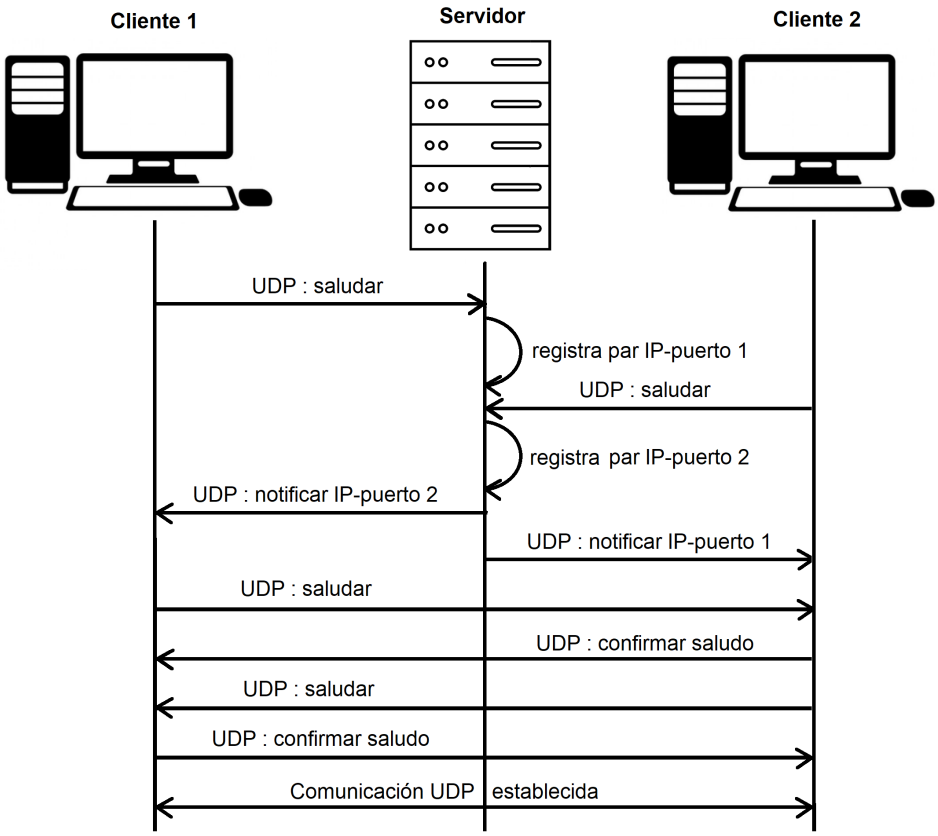


Figura 4.2: Diagrama del establecimiento de una comunicación UDP directa entre dos clientes.

Un último problema a destacar, pero el cuál sí que fue debido a la implementación hecha, fue que los mensajes de confirmación se almacenaban sobre la misma estructura que los mensajes recibidos en sí, es decir, un mensaje de confirmación podría reemplazar un mensaje pendiente de lectura en el mismo canal. Esto hacía que se perdiesen mensajes, provocando errores y/o bloqueos. Para solucionarlo, simplemente se separó el guardado de los mensajes de las confirmaciones en dos estructuras.

[11] `https://en.wikipedia.org/wiki/UDP_hole_punching`

4.2.2. Comunicación TCP y cifrado

Tras haber implementado la comunicación UDP, se pasó al desarrollo de la conexión TCP. Este desarrollo no presentó tantos problemas como la comunicación UDP, pues la comunicación cliente-servidor en sí funcionó prácticamente desde el primer intento. Para la comunicación TCP, como ya se dijo, se usaron las clases Socket y ServerSocket de Java. La primera es para la comunicación en sí, y la segunda es para que el servidor acepte peticiones de conexión y genere un objeto Socket para cada una. En el anexo F.2 se muestra un breve ejemplo de su uso, en el cual se basa la implementación final, con algunas modificaciones (en el anexo G.2 se explica la más importante). Decir que dichas estructuras permitían tanto el envío de cadenas como de objetos.

Tras esto, un aspecto importante a tratar de la conexión TCP era que la comunicación debía estar cifrada mediante certificados [3]. Para este fin se usaron las clases de java SSLServerSocketFactory[12] y SSLSocketFactory[13], que simplemente indicando un par de propiedades, permitía generar sockets cifrados. Para la creación de dichos sockets (tanto en el lado del cliente como del servidor) eran necesarias tres cosas, un par clave privada-clave pública [3], un certificado (en este caso autofirmado) que verifica que la clave pública de la entidad autorizada es la indicada en el certificado, y un almacén de certificados confiables, es decir, donde se guardan las claves públicas de entidades de confianza. Decir, que en este caso, el almacén de certificados de confianza de cada uno de los dos debería contener el certificado del otro (el del servidor contiene el del cliente y viceversa)

Para una primera versión, lo que se hizo fue usar Keytool[14] de Java, para generar dichos certificados (con algoritmo RSA) para el servidor y los clientes, siendo el de los clientes para todos igual (irían incluidos en el propio juego). Dicha herramienta también se usó para generar los almacenes de claves de confianza, tanto del cliente como del servidor. Una vez generados los certificados, cifrar la comunicación era tan simple como antes de usar SSLSocketFactory y SSLServerSocketFactory, asignar las propiedades de sistema *keyStore* (par de claves pública y privada), *keyStorePassword* (la contraseña para acceder a las claves), *trustStore* (el almacén de claves de confianza) y *trustStorePassword* (contraseña del almacén de claves de confianza). Después de asignar las propiedades, simplemente usar las clases nombradas para generar los *sockets*, y todo lo demás se mantenía igual.

En esta versión cualquier cliente podría leer los mensajes del resto de clientes, pero

[12] `https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLServerSocketFactory.html`

[13] `https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocketFactory.html`

[14] `https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html`

no los podrían leer aquellos que no tuvieran los certificados del juego. Sin embargo, esta versión no suponía una comunicación segura, pues para que se pueda considerar como tal, solo el propio cliente debería poder descifrar los mensajes que intercambia con el servidor.

Para hacer que la comunicación fuese verdaderamente segura, cada cliente debería poder tener sus propios certificados únicos, situación ante la cual surge un problema, y es que para que el servidor acepte una conexión, el certificado usado por el cliente en la conexión debe estar en el almacén de certificados de confianza, por lo que no se pueden usar sin más certificados generados dinámicamente por los clientes.

La solución era muy simple, si era la primera conexión del cliente con el servidor, el cliente se conectaría con los certificados 'por defecto' generados para la primera versión. Una vez conectado, el cliente generaría un certificado propio y único de forma automática (para esto se hizo uso de la API BouncyCastle[15]), y le enviaría dicho certificado al servidor (como el mensaje sería del cliente al servidor, el mensaje estaría cifrado con la clave pública del servidor, por lo que nadie podría descifrar el mensaje y extraer el certificado mandado). El servidor una vez recibido el certificado lo añadiría a su almacén de certificados de confianza, y ya estaría listo para aceptar conexiones con dicho nuevo certificado. Tras esto el servidor le enviaría al cliente una respuesta diciendo si todo ha transcurrido correctamente. Si hubiese funcionado correctamente, el cliente se desconectaría del servidor, actualizaría la opción de *keyStore* para usar el nuevo certificado, y volvería a establecer la comunicación con el servidor, esta vez ya con el nuevo certificado. En la figura 4.3 se resume el protocolo seguido para el establecimiento de la primera conexión.

4.3. Gestión del servidor y sistema de peticiones y notificaciones

Con toda la base del servidor creada el siguiente paso era establecer cómo iba a ser la interacción entre el cliente y el servidor. Primero, debía haber un proceso que se quedase constantemente en espera de nuevas conexiones y que lanzase hilos para manejar cada una de ellas, en este proyecto el encargado de esto es el programa principal, el cual simplemente hace lo dicho, crea el *socket* seguro y espera conexiones continuamente.

A su vez, el proceso principal tenía una instancia de una estructura que compartía con todos los hilos que lanzaba, es decir, dicha estructura era una zona de memoria y procesamiento compartido entre todos ellos, de forma que en este punto era donde se podrían dar problemas de concurrencia [10] como condiciones de carrera, y que hubo

[15] `https://www.bouncycastle.org/`

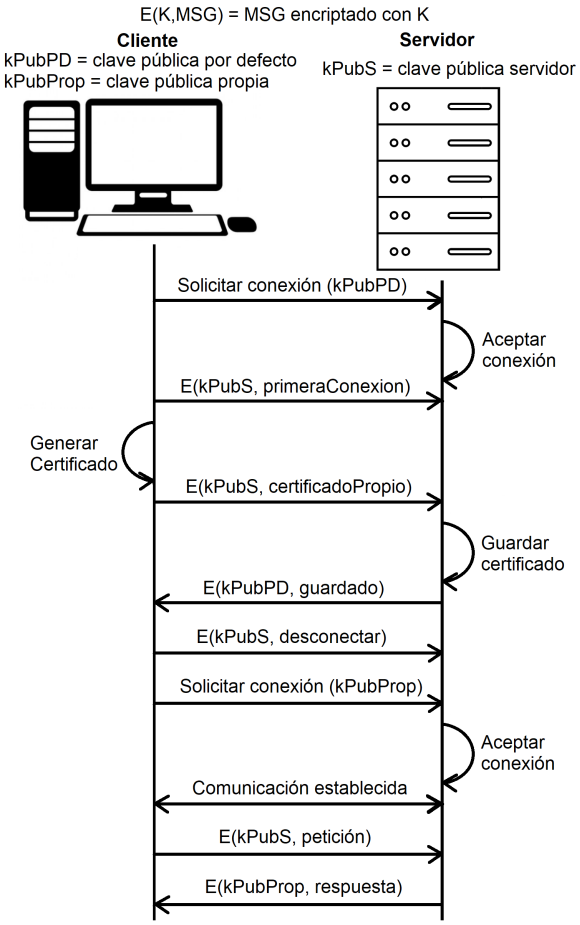


Figura 4.3: Diagrama del establecimiento de una comunicación TCP cliente-servidor segura en la primera conexión.

que gestionar (convirtiéndose en cierto modo en lo que se conoce como monitor[16]). Dicha estructura, el gestor del servidor, era la encargada de gestionar la interacción tanto con la base de datos (por ejemplo registrar un usuario), como la interacción entre usuarios (que uno mandase un mensaje y le llegase la notificación al otro usuario si estuviese conectado por ejemplo), como con el servidor en sí (iniciar sesión por ejemplo). Decir que en el anexo H se trata el tema de la concurrencia surgida de estas interacciones.

Además del gestor del servidor, se creó otra estructura intermediaria entre los hilos que se lanzaban para cada conexión y el gestor del servidor. Dicha estructura recibía una petición del cliente en forma de cadena (basado en el patrón identificadorPetición:parámetro1:parámetro2:..., siendo por tanto el carácter ':' reservado), y en base a la información que contuviese ejecutaba unas acciones u otras, y le respondía al cliente como correspondiese

Decir también, que dichos hilos lanzaban un hilo adicional cuya única función era,

[16] `https://es.wikipedia.org/wiki/Monitor_(concurrencia)`

periódicamente (cada 0.5 segundos), mandar un mensaje indicando que seguía activo, y comprobar si había recibido algún mensaje de vida del otro. Si pasasen más de 10 segundos sin respuesta se consideraría que el receptor no sería capaz de responder, situación en la que cortaría la comunicación directamente.

Por lo general, la comunicación se basaba siempre en el patrón petición-respuesta, y por tanto, una vez hecha la petición, el cliente esperaría la respuesta de esta antes de realizar otra petición. Sin embargo, hay algunas excepciones para las que no seguiría este comportamiento, casos en los que no se sabría cuándo llegaría el mensaje del servidor. Los casos en los que no se sigue el patrón, y por tanto son dignos de mención, son las notificaciones de amistad, los mensajes de chat, las notificaciones de reto, y la notificación de partida (cuando se está buscando). Para recibir todas estas notificaciones, en el cliente había un proceso encargado de recibir los mensajes con identificador de notificación, y de tratarlos como correspondiese.

Finalmente hacer mención que en el anexo I se habla acerca de la gestión de la IA en el servidor, la cual no se trata igual al resto de peticiones.

4.4. Partida en red

Para jugar una partida, el jugador debe seleccionar jugar partida, y esperar a que el servidor le mande la información del rival, pero como ya se explicó en la conexión UDP, para poder conectar dos clientes hay que seguir el protocolo descrito previamente.

El proceso completo comenzaría por solicitar buscar partida, y esperar a que otro jugador también busque partida. Una vez el servidor hubiese conseguido emparejar a dos jugadores, primeramente a cada uno de ellos les enviaría un mensaje notificándoles que iniciasen la comunicación UDP con él. Al comunicarse por UDP con el servidor, este registraría los puertos e IPs de cada uno de los dos clientes. Una vez ambos se hubiesen comunicado por UDP con el servidor, este les mandaría a ambos un mensaje indicándoles que les ha encontrado un rival y la información de éste (IP y puerto), además de indicarle si él sería anfitrión o cliente. Tras esto, ambos clientes se saludarían mutuamente mediante la comunicación UDP, y si se conectaron exitosamente se pasaría a la selección de personaje y mapa, en caso contrario se cancelaría el emparejamiento. Como se ha visto, el proceso era el protocolo de establecimiento de la comunicación UDP, pero con el paso previo de buscar partida y que el servidor notificase que inicie el protocolo.

Una vez en la selección de personaje, si el usuario ha sido notificado por el servidor para ser anfitrión, este podrá seleccionar personaje y mapa, en caso contrario, solo podría seleccionar personaje. Durante el tiempo que dura la selección cualquiera de

los dos jugadores podría salirse, notificándole al otro cliente que cancele la partida. Pasado el tiempo de selección, el anfitrión notifica al cliente su personaje y el mapa seleccionados, y después el cliente le notifica al anfitrión su personaje.

A continuación, se instanciarían los controladores de los personajes indicándole cual es el local y cual el remoto, puesto que según ese criterio actuarán diferente. En caso de ser local, se ejecutaría normalmente con el añadido de que continuamente estaría mandando las teclas reconocidas al otro cliente. Y si fuese remoto, las teclas reconocidas se extraerían de los mensajes recibidos del otro cliente, y se procesarían posteriormente como si se hubiesen introducido desde el teclado. Señalar que estos controladores, irían comprobando cuanto tiempo habría pasado desde el último mensaje de teclas, y si hubiesen pasado más de 5 segundos se consideraría que se habría perdido la conexión, y se finalizaría la partida.

Seguidamente se pasaría a la sincronización, proceso necesario para asegurar que ambos empezarían la partida a la vez y verían lo mismo. El proceso de sincronización era muy sencillo, si fuera anfitrión mandaría un mensaje de listo hasta recibir la confirmación y luego esperaría recibir un mensaje de listo del otro cliente, en caso de no ser anfitrión sería a la inversa. En este caso, el intercambio de mensajes sería mucho más rápido para reducir al mínimo el retardo entre lo que ve el anfitrión y el cliente.

Tras la sincronización, ambos clientes empezarían la partida en red, la cual no se gestionaba igual que una partida normal, pues había que asegurar que ambos jugadores veían lo mismo. Para esto se estableció que el anfitrión iba a 'guiar' la partida para ambos. Esto quiere decir que el anfitrión le indicaría siempre al cliente el estado de la partida y el cliente forzaría a que su partida estuviese en ese estado. Sin embargo, los controladores de los personajes controlarían estos tanto en el cliente como en el anfitrión, es decir, los movimientos que hiciesen los personajes y sus animaciones en sí, se calcularían en ambos lados, y en el lado del cliente únicamente se ajustarían las variables como la posición. De esta forma, el cliente no tendría tanto retardo en las animaciones en sí y tendría una mejor percepción de la partida.

Durante la partida en sí, en cualquier momento los jugadores podrían decidir rendirse y abandonar la partida, finalizando la partida de forma prematura y contando como derrota para el jugador que se rindiese. En caso de rendirse, se mandaría un mensaje al otro cliente notificándoselo para que terminase la partida.

Independientemente de como terminase la partida, el anfitrión mandaría mensajes al cliente notificándole el final de la partida por si no hubiese terminado correctamente por su cuenta. Tras esto, el anfitrión mandaría el resultado de la partida al servidor, indicándole si era clasificatoria o no. Si la partida fuese normal ambos clientes cerrarían

la comunicación UDP, pero si fuese clasificatoria, ambos esperarían un mensaje del servidor indicándoles cuantos puntos clasificatorios han ganado o perdido, y tras ello cerrarían la comunicación. Señalar que si el que hiciese de cliente, hubiese detectado la caída del anfitrión, sería este el que mandase el mensaje de registro de la partida al servidor.

Durante el desarrollo de la partida en red surgieron diversos problemas, entre ellos el más relevante lo que ya se comentó en la conexión UDP, sin embargo, hubo otros dos destacables. El primero de ellos fue que de vez en cuando la partida no iniciaba correctamente, quedándose los dos clientes bloqueados, y tras un tiempo excesivo, empezada la partida con personajes y mapas erróneos. El problema se debía a dos errores, siendo el primero que en un comienzo las confirmaciones se almacenaban en la misma estructura que los mensajes normales, y a veces los mensajes de confirmación reemplazaban un mensaje recibido, quedándose a la espera del mensaje borrado. Esto se solucionó guardando las confirmaciones y los mensajes por separado. El segundo error fue que había dos procesos leyendo mensajes con el mismo identificador, entonces, a veces, uno leía los mensajes del otro, haciendo que alguno de los dos perdiese un mensaje importante. Para solucionar esto, simplemente se hizo que los dos procesos leyesen mensajes con distinto identificador.

El segundo problema, fue que durante la partida el que actuaba como cliente veía a los personajes con muchos errores visuales. Se invirtieron bastantes horas en depuración, incluso rehaciendo las estructuras con otro enfoque. Sin embargo el problema no desaparecía, y las nuevas estructuras funcionaban incluso peor, por lo que se volvió a las iniciales. Finalmente, se descubrió que el problema se daba al actualizar el estado de la partida en base a lo que informase el anfitrión. Concretamente, se actualizaban las estructuras de los personajes y la partida, pero los gráficos que se mostraban por pantalla contenían información desactualizada. Esta inconsistencia entre los gráficos mostrados y la información en las estructuras producía todos esos problemas visuales. Para solucionar el problema, simplemente se corrigió la forma en la que se actualizaba el estado de la partida y cuándo se obtenían los gráficos, de forma que ambos fuesen coherentes entre sí.

Capítulo 5

Conclusiones

Este proyecto ha consistido en desarrollar un clon del videojuego de lucha en dos dimensiones Fatal Fury 2 (SNK 1992) que proporcionase diversos servicios en línea, siendo el más importante la posibilidad de jugar partidas en red, y que permitiese enfrentarse a dos IAs adaptativas que aprendiesen al jugar contra los usuarios. Este proyecto partía de las prácticas de la asignatura Videojuegos, y las ampliaba para proporcionar lo indicado, sin modificar nada de la lógica ni gráficos del juego base.

El algoritmo para la IA fue finalmente afrontado con aprendizaje por refuerzo, concretamente una solución basada en el *Q-Learning* [5] con regresión lineal como apoyo. Esta implementación cubría las necesidades de un aprendizaje significativamente rápido y efectivo, con un coste computacional muy bajo. El único problema es que si se buscase algo cercano a una IA perfecta, se necesitarían muchísimas partidas.

En cuanto a juego en red se decidió introducir la gestión de usuarios, la interacción entre usuarios mediante amistades y mensajes, entre otras, y la posibilidad de jugar partidas en red tanto competitivas como no competitivas, así como jugar partidas no competitivas con amigos. Para el desarrollo de estos servicios se creo una estructura cliente-servidor basada en el protocolo petición-respuesta, la cual se creó prácticamente desde cero, creando unas estructuras de comunicación propias (TCP cifrado y UDP), una base de datos relacional usando el estándar JPA, y una capa gestora del servidor que controlaba todas las acciones de los usuarios.

Se presentaron problemas con los que no se tenían experiencia y los cuales resultaron muy instructivos, como por ejemplo, establecer una comunicación UDP directa entre dos clientes a través de la red, los cuales se 'conocen' por primera vez.

Decir que se considera que se han alcanzado todos los objetivos planteados, pues se ha conseguido implementar una IA adaptativa que es capaz de aprender a un buen ritmo, y que se ha demostrado que verdaderamente aprende y mejora, y se han desarrollado todas las características en red prometidas, cumpliendo todos ellos con su función correctamente.

5.1. Cronograma

Aproximadamente, el desarrollo del proyecto implicó unas 310 horas aproximadamente, unas 90-100 la inteligencia artificial y unas 200-210 los servicios en red y el prototipo de interfaz, más unas 90-100 horas de documentación y desarrollo de la memoria del proyecto. Nombrar también que para los entrenamientos de las IAs de demostración, se les dejó entrenando un total de unas 270 horas (unas 170 la de las 9000 partidas, y unas 100 cuatro IAs, entrenadas simultáneamente de 5000 partidas), y para la realización de todas las pruebas se necesitaron entorno a unas 20 horas.

En la figura 5.1 se presenta un cronograma resumido del desarrollo del proyecto (la séptima semana no se hizo nada por estar de vacaciones).

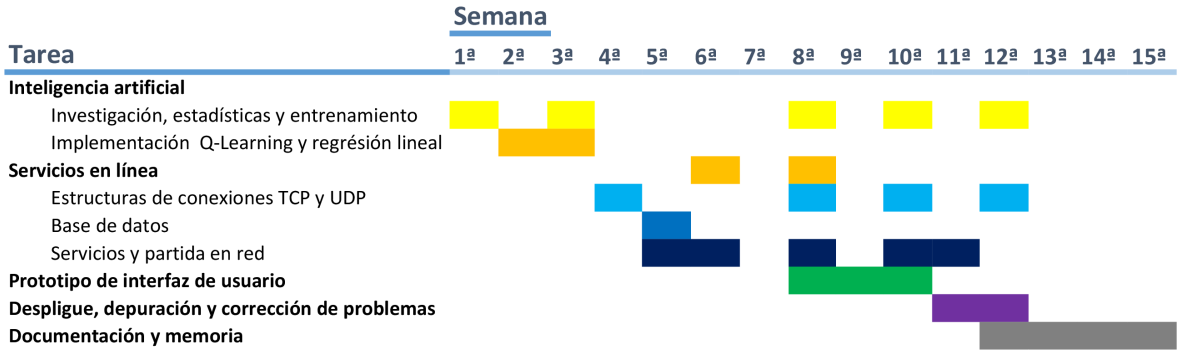


Figura 5.1: Cronograma del desarrollo del proyecto.

5.2. Posibles ampliaciones

A continuación se van a nombrar algunas mejoras que podrían resultar interesantes de desarrollar en un futuro.

- Respecto a lo que aporta el juego en sí a los usuarios, se podrían proporcionar otros modos de juego con mecánicas distintas, como por ejemplo peleas de más de dos jugadores, lo cual implicaría modificar toda la lógica de la pelea.

- En cuanto a la IA, se podría plantear desarrollar el algoritmo de aprendizaje por refuerzo profundo, o el del actor-crítico, con intención de generar una IA casi perfecta mediante un entrenamiento independiente no integrado en el servidor.

- En el servidor se podría introducir una mejora para ser capaz de gestionar un mayor número de usuarios, y proporcionar mayor disponibilidad y tolerancia a fallos, pues actualmente es un servidor centralizado. Una posible solución muy tradicional sería establecer una estructura primario-copia.

5.3. Opinión personal

Me gustaría decir que el proyecto en sí me ha resultado muy edificante pues me he tenido que enfrentar a diversos problemas en diversos campos, y aprendiendo de todos ellos.

El desarrollo de la IA hizo que me enfrentase por primera vez a un problema de mi especialidad sin guía alguna, pues todas las prácticas hechas para las asignaturas siempre indicaban la metodología a seguir (regresión, clasificación, etc.). El tener que documentar, valorar opciones, sus ventajas y desventajas, y la viabilidad en el proyecto, hizo que me sintiese algo más cercano a la sensación de estar en un puesto de trabajo en el que me han pedido que resuelva un problema, para el cual obviamente no tengo a un profesor diciéndome por donde ir. Me siento especialmente orgulloso de no solo haber intentado afrontar el problema con un algoritmo y ya, sino intentar ir un paso más allá e intentar adaptarlo y mejorarlo para el uso que se le iba a dar, mediante el uso de la regresión lineal.

Respecto al juego en línea, el crear el servidor y las comunicaciones prácticamente de cero, me ha hecho darme cuenta de todo lo que hay detrás de ello y los problemas que pueden presentar, que me han hecho aprender cosas que me resultarán útiles en un futuro. Entre las cosas aprendidas, he de destacar sobre todas las demás el establecimiento de la comunicación UDP entre dos clientes que no se conocen. En este caso, pese a que había una solución existente, la cual no conseguí encontrar en su momento, me siento orgulloso de poder decir que 'descubrí' la perforación de agujeros UDP [9] por mi cuenta. Es un protocolo muy simple y básico, pero que ni de lejos se vio en las clases, y que sin duda es un conocimiento que debería tener cualquier informático.

Brevemente, decir que me ha resultado muy instructivo, enfrentándome a problemas de diversos campos, y viendo mis propias capacidades para enfrentarme a ellos sin que alguien me diga exactamente qué hacer.

Bibliografía

[1] Tim Kindberg Georgle Coulouris, Jean Dollimore. *Distributed Systems: Concepts and design Fourth Edition.* Addison Wesley, 2005.

[2] Francisco de Asís López Fuentes. *Sistemas distribuidos.* 2015.

[3] Joshua Davies. *Implementing SSL/TLS using Cryptograpy and PKI.* Wiley Publishing, Inc., 2011.

[4] Ethem Alpaydin. *Introduction to machine learning Fourth Edition.* MIT Press, 2020.

[5] Peter Dayan Christopher J.C.H. Watkin. Technical note q-learning. *Kluwer Academic Publishers*, 1992.

[6] John N. Tsitsiklis Vijay R. Konda. Actor-critic algorithms. *Massachusetts Institute of Technology*, 1992.

[7] S. Sudarshan Abraham Silberschatz, Henry F. Korth. *Fundamentos de bases de datos Cuarta edición.* Mc Graw Hill, 2002.

[8] Gary Gregory Christian Bauer, Gavin King. *Java Persistence with Hibernate.* Manning Publications Co., 2006.

[9] R. Niederberger F. Petri E. Grünter, M. Meier. Dynamic configuration of firewalls using udp hole punching. *Forschungszentrum Jülich*, 2006.

[10] Fernando Sánchez Figueroa Alexis Quedada Arencibia José Tomás Palma Méndez, Mª Carmen Garrido Carrera. *Programación concurrente.* Thomson, 2011.

[11] Yunshu Du. Introduction to deep q-network. *Washington State University*, 2016.

[12] Mauricio Arango. Deep q-learning explained. *Oracle*, 2018.

[13] David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller Volodymyr Mnih, Koray Kavukcuoglu. Playing atari with deep reinforcement learning. *DeepMind Technologies*, 2013.

[14] Sangbin Moon Seongho Son Hyoil Lee Jinyun Chung Inseok Oh, Seungeun Rho. Creating pro-level ai for a real-time fighting game using deep reinforcement learning. *NCSOFT*, 2019.

[15] Raul F. Neto Matheus R. F. Mendonça, Heder S. Bernardino. Simulating human behavior in fighting games using reinforcement learning and artificial neural networks. *Universidad Federal de Juiz de Fora*, 2015.

[16] Luis Rincón. *Introducción a los procesos estocásticos*. Facultad de Ciencias UNAM, 2012.

Lista de Figuras

Anexo A

Algoritmo Q-Learning

Se citan los conceptos en los que se basa el algoritmo del *Q-Learning* [5]:

- El agente, basará sus acciones en los estados, siendo un estado la información del entorno que rodea al agente. que puede ser más o menos detallada y usarse toda o solo parte (en el caso del videojuego, esta información sería la posición de los personajes o la vida restantes de estos, por ejemplo). Dicho de otra forma, es la percepción del entorno que tiene el agente, en base a la cual debe tomar las decisiones, al igual que una persona analiza una situación, y se basa en experiencias pasadas para tomar una decisión.

- Los estados deben cumplir la llamada propiedad de Markov[1] [16], la cual dice que las recompensas que se pueden obtener a partir de él tomando distintas secuencias de acciones únicamente dependen de ese estado, y no de los anteriores, es decir, las futuras recompensas solo dependen del estado actual del entorno, independientemente de como se ha llegado a él.

- Una tarea de aprendizaje por refuerzo en la que para todos los estados del entorno se cumple la propiedad de Markov, se conoce como proceso de decisión de Markov[2] [16]. En caso de que el número de estados y acciones sea finito, se le conoce como proceso de desisión de Markov finito.

- Como se ha explicado antes, la política de distribución (π(s,a) siendo s el estado y a la acción) define la probabilidad de elegir una acción dado un estado, y la función valor siguiendo una política ($Q^{\pi}(s, a)$ siendo π la política, s el estado y a la acción) evalúa la recompensa estimada una acción para un estado concreto.

- El principio de optimalidad de Bellman[3] dice que, el valor de un estado concreto

[1] `https://es.wikipedia.org/wiki/Propiedad_de_Markov`
[2] `https://es.wikipedia.org/wiki/Proceso_de_Markov`
[3] `https://es.wikipedia.org/wiki/Ecuacion_de_Bellman`

depende de la mejor acción (la que más recompensa de) que se pueda tomar desde él, y que por tanto, una política óptima es aquella que partiendo de un estado y acción iniciales cualquiera, en la secuencia de estados-acciones que le sigue se selecciona siempre la o las acciones que maximicen el valor de cada uno de ellos. Esto da lugar a la ecuación de Bellman, la cual calcula el valor óptimo de un estado:

$$V^*(s_t) = \max_{a \in A(s_t)} \{Q^\pi(s_t, a) + \beta V^*(s_{t+1})\} \tag{A.1}$$

Donde 's_t' es el estado a evaluar, A(s_t) el conjunto de acciones ejecutables en el estado 's_t', 'a' una acción perteneciente a A(s_t), $Q^\pi(s, a)$ la recompensa de la acción 'a' para el estado 's_t' en base a la política π, 's_{t+1}' el estado al que se ha llegado desde 's_t' mediante la acción 'a' yβ el factor de paciencia (representa cuanto se valoran las recompensas futuras, siendo valor mínimo 0, es decir, solo se valora la recompensa inmediata, y valor máximo 1, se valora por igual la inmediata y las futuras). La traducción de la ecuación sería: el valor óptimo de un estado es igual al de la acción cuya suma de la recompensa inmediata y la futura sea la mayor de todas las acciones ejecutables en dicho estado.

– El aprendizaje por diferencias temporales[4] puede aprender directamente de la experiencia sin la necesidad de un modelo del entorno (como el aprendizaje basado en el método de Monte Carlo[5]), es decir, aprende a medida que toma decisiones (aunque en el caso típico de Monte Carlo, se aprende al final de cada 'episodio'), ya sean correctas o erróneas, y actualiza la función de valor de forma dinámica en base a la recompensa recibida y la que se estimaba que se iba a recibir (la cual se calcula a partir de experiencias pasadas). Por tanto, actualiza la función tras cada transición estado-acción sin esperar al resultado final del 'episodio' (sin esperar al resultado final de la partida). Un ejemplo básico de Monte Carlo (en el que se basa Q-Learning) para la actualización (al terminar el episodio) de la función valor de estado sería:

$$V(s) = V(s) + \alpha[R - V(s)] \tag{A.2}$$

Donde 's' sería el estado a actualizar, 'α' el factor de aprendizaje y 'R' la recompensa total recogida durante el episodio.

El *Q-Learning*, se basa en actualizar únicamente la función valor de estado, es decir, la función que evalúa el valor de un estado en base a las acciones que se pueden tomar

[4] `https://es.qwe.wiki/wiki/Temporal_difference_learning`
[5] `https://es.wikipedia.org/wiki/Metodo_de_Montecarlo`

desde él, o dicho de otra forma, se busca actualizar la estimación de las recompensas de las acciones para cada estado (punto de vista del *Q-Learning*), ya que como ya se ha explicado, el valor de un estado depende de las acciones futuras que se puedan tomar a partir de él. El que se pueda permitir únicamente actualizar la función valor, dejando fija la política de distribución, se debe a que en este caso la función de valor aprendida de aproxima directamente a la óptima, gracias a aplicar el principio de optimalidad de Bellman. Concretamente, el *Q-Learning* actualiza la función valor partiendo de la actualización más simple de Monte Carlo (ya se mostró antes), y reemplanzando en ese calculo la recompensa (R) por la ecuación de Bellman simplificada (en vez del valor del estado, que se basa en el máximo de todas las acciones, únicamente el cálculo de cada acción). La fórmula de la actualización de la función valor de las acciones en base al estado quedaría:

$$Q(s_t, a) = Q(s_t, a) + \alpha[(r + \beta \max_{a2 \in A(s_{t+1})} Q(s_{t+1}, a2) - Q(s_t, a)] \tag{A.3}$$

Donde 's_t' sería el estado desde el que se tomo una decisión, 'a' la acción elegida en el estado 's_t', Q('s_t',a) la estimación de la recompensa de la acción 'a' en el estado 's_t', 'α' el factor de aprendizaje, 's_{t+1}' el estado al que se llega desde 's_t' ejecutando 'a', A('s_{t+1}') el conjunto de acciones ejecutables desde el estado 's_{t+1}', 'β' el factor de paciencia y 'r' la recompensa recibida por la acción 'a' en el estado 's_t'. Por lo tanto, la traducción de esta fórmula sería: la nueva estimación de recompensa de la acción 'a' en el estado 's_t' es igual a la suma de la recompensa estimada original más, la diferencia (error de estimación) entre la recompensa óptima actual (recompensa recibida más recompensas futuras) y la estimación original, por el factor de aprendizaje.

Esta forma de actualizar la función tiende a ser óptima independientemente de la política de distribución, pero para que se cumpla y el algoritmo converja, todos los pares estado-acción, se deben actualizar de forma continua, es decir, para alcanzar la optimalidad, se debe seguir hasta el infinito (lo cual en el caso del videojuego sería así puesto que nunca dejarían de aprender, únicamente que probablemente irían a un ritmo más lento que el deseable). Por tanto, pese a que se pueda dejar la política de distribución fija, esta debe poder cumplir dicha característica de permitir actualizar todos los pares continuamente (aunque actualice unos con más frecuencia que otros).

Un primer pensamiento para esta política de distribución podría ser que basándose en experiencias anteriores, seleccionase siempre la que mejor resultado le hubiese dado. Sin embargo, esto haría que simplemente se explotasen mucho unos estados concretos, dejando de visitar muchos otros, y por ende no actualizándolos de forma continua como requiere el algoritmo. Esto nos indica que se ha de establecer una política que

establezca un equilibrio entre explotación de las acciones óptimas y exploración de nuevas acciones.

Una política muy simple, y muy generalizada para este algoritmo, es la política ϵ-*greedy*, la cual únicamente establece que a la hora de elegir una acción en un estado hay una probabilidad ϵ de elegir una acción aleatoria (exploración) y una probabilidad 1-ϵ de seleccionar una acción óptima basada en experiencias pasadas (explotación). Una forma de aplicar esta política (la cual es la que se eligió para la implementación) sería que a medida que van trascurriendo los 'episodios' del entrenamiento, el valor de ϵ se va reduciendo paulatinamente, haciendo que al comienzo explore mucho más de lo que explota, y hacia el final cuando ya tiene mucha experiencia, explotar más de lo que explora. Señalar además, que para que cumpla la condición de la actualización continua, el ϵ deberá tener siempre un valor mínimo (por ejemplo 0.05), que permita que, aunque con poca frecuencia, siga explorando otras posibles acciones no óptimas (que tal vez en un futuro acaben siendo óptimas).

El último punto a tratar sería el como se guardan y representan la función valor y las experiencias pasadas. En el caso del Q-Learning, este aspecto es muy simple, es una matriz de tamaño Nº de posibles estados * Nº de acciones totales, en la que en cada celda se guarda la estimación de recompensa de una acción en un estado concreto, estando dicha estimación basada a su vez en experiencias pasadas. Es decir, la función de valor se define a partir de la matriz (Q(s,a) = Q[s][a] siendo 's' el estado y 'a' la acción), la cual contiene la información de experiencias pasadas, de forma indirecta, gracias a la fórmula de actualización. A partir de todo lo explicado, el algoritmo resultante sería el siguiente:

Inicializar la tabla Q(s,a) aleatoriamente;
while *queden episodios* **do**
  do
    Elige una acción a ejecutar en base a la política ϵ-*greedy*; Con probabilidad ϵ elige acción aleatoria;
    y con probabilidad 1-ϵ elige acción $a = \text{máx}_{a \in A(s_t)} Q(s_t, a)$;
    Ejecutar a y esperar a recibir la recompensa r y llegar al estado s_{t+1}
    $Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[(r + \beta \, \text{máx}_{a2 \in A(s_{t+1})} Q(s_{t+1}, a2) - Q(s_t, a)]$;
    $s_t \leftarrow s_{t+1}$ Actualizar ϵ según se haya decidido;
  while s_t *no sea terminal*;
end

Algoritmo 2: Q-Learning

Conocido el algoritmo y sus ventajas es fácil entender por que se pudo elegir este algoritmo, sin embargo, como todo algoritmo tiene alguna desventaja, incluso aun habiéndolo elegido. En este caso los cuatro principales problemas son:

– Es necesario entender bien el problema para saber definir correctamente los

estados y el sistema de recompensas. Si los estados no contienen la información correcta, por ejemplo, la mayoría es información que no resulta útil, al agente le resultará muy difícil mejorar al no tener una buena percepción del entorno. A su vez, si el sistema de recompensas no es lo suficientemente bueno como para guiar al agente correctamente (por ejemplo, llegar a recompensar una mala acción), es posible que el agente no aprenda lo que se espera al basarse en valoraciones erróneas.

- Este algoritmo resulta idóneo para entornos sin demasiados estados ni acciones, ya que por tanto dichos pares serán visitados más frecuentemente. Por contra, si el número de estados y acciones es demasiado grande, resultará muy difícil que el agente visite todos o la mayoría de estados, y revisitará con mucha menos frecuencia, haciendo que el algoritmo prácticamente no pueda converger, salvo con un tiempo exacerbado. El número de estados crece o decrece en base a la información que se use del entorno, es decir, cuantas variables del entorno y como de detalladas son. Por esto último, se vuelve a destacar la necesidad de conocer bien el dominio del problema, para saber analizar que información es necesaria, y cómo de detallada, ya que es posible que una buena simplificación de esta información haga que un problema que parecía no iba a ser tratable por el algoritmo pase a serlo (como ocurre en el caso de este proyecto).

- Como no se puede saber con exactitud cuantos episodios se van a tardar en visitar todos o la mayoría de los estados, es difícil ajustar la política ϵ-*greedy* de forma que se mantenga el equilibrio entre exploración y explotación. En esta situación solo se pueden hacer suposiciones y estimar, ya que pese a aunque siempre se deje un pequeño margen de exploración, si no se han visitado y revisitado suficientes estados, se alejará más de la optimalidad.

- Para poder tomar una decisión mejor que aleatoria en las acciones de explotación el estado debe haber sido visitado al menos una vez, es decir, en caso de que fuese momento de explotar si no se ha visitado nunca solo se puede tomar una elección aleatoria. Sin embargo, lo ideal, sería que en esa situación pese a no haber estado nunca, tomase una decisión algo mejor que una decisión aleatoria (esto se tratará en el apartado 3.4).

Anexo B

Simplificación de los estados del entorno de la Inteligencia artificial

Durante una partida, la información disponible es la siguiente:

- La vida de los dos personajes (el del jugador y el de la IA), tendiendo ambos valores que van de 0 a 100, ambos incluidos.
- La posición de ambos personajes en el escenario en dos dimensiones, es decir, se conoce la posición en los ejes X e Y, yendo los valores de 0 a 1280 y de -100 a 720 respectivamente. Si se conoce su posición, se conoce también la distancia entre ellos, y también si están saltando o no (si están en mitad de un salto, las acciones ejecutables son distintas).
- La orientación en la que están mirando ambos personajes, la cual salvo en situaciones concretas, ambos personajes se estarán mirando mutuamente.
- El estado de cada uno de los personajes, entendiendo el estado como el movimiento que están realizando, no solo acciones, sino también aquellos que son estados consecuencia de las acciones (por ejemplo, el estado de retroceso tras un golpe recibido).
- El tiempo restante de la ronda en marcha, que toma valores de 0 a 90, el cual marca el final de la ronda en caso de que ninguna de las vidas llegue a 0.
- El número de ronda que se está jugando, con valores de 1 a 4.
- El número de rondas ganadas por cada uno de los participantes en la partida. Si alguno de los dos ha ganado 2 rondas, la partida habrá terminado.

Aparte de estos habría otros datos más profundos como el daño de cada ataque, la *hitbox* exacta de cada personaje y sus movimientos, etc., que como son información

que ni siquiera los propios jugadores conocen, y no son realmente de interés ya que quedan reflejados indirectamente por las experiencias (por ejemplo, cuando golpee con un movimiento recibirá una recompensa, que lleva de forma implícita el daño realizado), se omitirán. Sin embargo, aun eliminando esa información, con la indicada anteriormente en bruto el número de estados se dispara (del orden de 10^{17}), de forma que resulta imposible en la práctica usar el algoritmo con la información en bruto. Por tanto, fue necesario decidir que información era verdaderamente imprescindible, y con qué precisión.

El análisis realizado se basó en cierta forma, en establecer unos razonamientos algo más humanos, es decir, los jugadores no conocen la información exacta de cada variable y aun así aprender a jugar. Por tanto, el análisis realizado fue el siguiente:

- La diferencia de recompensa en base a la vida de los personajes va a ser ínfima por un punto más o menos de vida, incluso 10 puntos de vida por ejemplo no supondrán una gran diferencia, por lo que resulta obvio que no es necesaria una gran exactitud. Teniendo en cuenta que los propios jugadores no conocen la vida exacta, no tiene sentido que la IA se base en ella con plena exactitud, ya que, al fin y al cabo, la IA imita en cierto modo a las personas en su modo de aprender, y si los humanos no necesitan conocer el punto de vida exacto, la IA tampoco. A grandes rasgos, un usuario se fija en la vida pensando en si le queda mucha, media o poca vida, es decir, se pueden simplificar 100 valores de vida en 3 simplemente haciendo 3 intérvalos en los valores. Además, esta simplificación hace que haya la suficiente diferencia en la vida cuando cambia de un intérvalo a otro, como para que la recompensa recibida por una acción si que pueda verse modificada de forma significativa.

- Con la posición ocurre lo mismo que con la vida, los usuarios no conocen los valores exactos ni los necesitan. En cuanto al eje X, no les importa en que parte del mapa estén los personajes (porque es un mapa plano sin obstáculos), lo que les interesa de verdad es la distancia con el enemigo, para saber si un movimiento les puede dar o no por ejemplo. Aún así, de nuevo, no necesitan saber el valor exacto de la distancia, se guían por si el enemigo está cerca, lejos, o a una distancia media, lo cual suele ser más que suficiente, simplificando así 1280 valores en 3. Mientras trato, en el eje Y, en cierto modo solo es necesario saber si la propia IA está por encima de la altura que le permite ejecutar alguna acción en el aire, porque por debajo de esa altura, prácticamente se puede deducir en base al estado del personaje (por ejemplo salto vertical) y la distancia entre ellos, y por encima de dicha altura en el momento que se ejecute un movimiento se cambiará de

estado y se empezará a caer. A su vez, la altura del jugador se puede omitir, y hacerse una idea con el estado y la distancia. Por tanto, el eje Y se podría simplificar únicamente en si la IA está o no por encima de esa altura (alto o bajo), pasando de 820*820 combinaciones de valores a 2 valores.

- La orientación es una variable que ni los jugadores ni la IA pueden controlar, y casi siempre será la misma, ambos personajes mirándose el uno al otro. Siendo una variable fuera del control para los dos, y que realmente no aporta información útil, se puede omitir directamente.

- El estado de cada uno de los personajes es una variable que no se puede simplificar, puesto que cada estado tiene su propia información única (duraciones, daños, *hitbox*, *hurtbox* y *coverbox*, etc.). Sin embargo, la información del estado de la IA no es de interés más allá de si puede o no ejecutar una acción (en el momento que pueda, ejecutará una), lo cual va a venir siempre marcado por la propia fase del algoritmo de esperar el resultado de una acción, por lo que el estado de la IA se puede omitir completamente. Por tanto nos quedaríamos únicamente con el estado del jugador, pasando de unos 50*50 combinaciones de ambos valores, a únicamente 50.

- El tiempo restante de la ronda solo aporta a la pelea el saber si queda poco o no para terminar la ronda y ganar o perder en base a la vida restante, pero incluso eso tampoco resulta de interés pues la mayoría de veces la pelea no se acerca siquiera al tiempo de duración de la ronda, por lo que de por sí no tiene mucha relevancia. Omitiendo esta variable, se pierde ese factor de estrategia de posible victoria por tiempo, pero teniendo en cuenta que casi nunca se da el fin de la pelea por tiempo, y que siempre va a dar mayor recompensa el ganar por un golpe que por que se acabe el tiempo, hace que cobre sentido el omitir la variable, puesto que reduce significativamente el número de estados, y no se pierde apenas información de interés.

- El número de ronda que se está jugando la única información que aporta es si se está jugando o no la cuarta ronda, que tiene duración infinita hasta que uno de los dos o los dos mueran. Siendo que se ha omitido la variable del tiempo restante de la ronda, para la IA las 4 rondas serán iguales, por lo que no aporta ninguna información, y por ende se puede omitir.

- Siendo que las rondas no se afectan de ninguna forma entre sí (no se guarda la vida de una ronda para otra por ejemplo), da igual cuántas victorias tenga cada

uno, haciendo que el objetivo de la IA se simplifique al solo tener que pensar en ganar la ronda actual, y no tener que pensar en estrategias complejas (por ejemplo, sacrificar una ronda para tener ventaja en la siguiente). Y aunque el jugador llevase una victoria, y solo le faltase ganar una, el objetivo de la IA seguiría siendo el mismo, ganar la ronda actual. Puesto que esta variable no cambia el comportamiento de la IA de ninguna forma (siempre se concentrará en ganar la ronda actual, pues es la única forma de ganar la partida), las variables se pueden omitir.

Por tanto, la representación final del estado para el agente, contendría la siguiente información del entorno:

- La vida de los dos personajes, simplificada en 3 valores (mucha, media, o poca vida).
- La distancia entre los personajes, simplificada en 3 valores (cerca, lejos o distancia media), en base a si es una distancia a la que se puede acertar prácticamente cualquier ataque , si es una distancia a la que prácticamente ningún ataque salvo algún especial va a golpear, o si es una distancia a la que algunos ataques pueden golpear de vez en cuando.
- Si el personaje del agente está saltando o no.
- El estado del personaje del jugador.

Anexo C

Algoritmo del sistema de recompensas

entrada: Estado inicial s, Acción ejecutada a, Estado resultado s'
salida : Recompensa r
Recompensa r = 0;
if *s' es terminal de la pelea completa* **then**
  if *ha ganado la pelea la IA* **then**
    r = 200;
  else if *ha perdido la pelea la IA* **then**
    r = -200;
  else
    r = 0;
  end
else if *s' es terminal de una ronda que compone la pelea* **then**
  if *ha ganado la ronda la IA* **then**
    r = 100;
  else if *ha perdido la ronda la IA* **then**
    r = -100;
  else
    r = 0;
  end

Algoritmo 3: Sistema de recompensas parte 1

```
else
    /* Ambos siguen con la misma vida tras la acción                    */
    if s.vidaIA == s'.vidaIA and s.vidaJugador == s'.vidaJugador then
        /* El jugador en s estaba atacando y la IA se protege          */
        if s.jugadorAtacando and a == acción de protección then
            /* Se protege del ataque estando tanto en s como en s'
               lejos del enemigo (se cubre innecesariamente)            */
            if s.distanciaEntrePeronajes > 200
             and s'.distanciaEntrePeronajes > 200 then
                r = -1;
            /* En s o s' el ataque ha estado lo suficientemente
               cerca para suponer un riesgo                             */
            else
                r = 5;
            end
        /* a es un ataque y tanto en s como en s' está a una
           distancia que no puede dar al enemigo (ataca al aire)        */
        else if a == ataque and s.distanciaEntrePeronajes > 200
         and s'.distanciaEntrePeronajes > 200 then
            r = -1;
        /* Ha hecho un movimiento no defensivo ni ofensivo y no ha
           tenido una repercusión negativa                              */
        else
            r = 0;
        end
    /* A alguno de los dos, o a los dos, le ha bajado la vida tras
       la acción                                                        */
    else
        /* Se ha cubierto de un ataque y ha recibido daño, se
           recompensa con el daño recibido reducido para incentivar
           que se defienda                                              */
        if s.vidaIA ¡s'.vidaIA and a == acción de protección then
            /* Se da máximo 30 de recompensa. Se valora la
               proporción del daño recibido con la vida en s            */
            r = 30.0 * ((s.vidaIA - S'.vidaIA) / 2) / s.vidaIA;
        /* Hizo un movimiento no defensivo que tuvo una repercusión
           sobre las vidas                                              */
        else
            /* Se da entre 30 y -30 de recompensa. Se valora la
               proporción del daño recibido con la vida en s y la
               proporción del daño realizado con la vida en s del
               jugador                                                  */
            r = (30.0 * (s'.vidaIA - s.vidaIA)/ s.vidaIA + 30.0 * (s.vidaJugador
             - s'.vidaJugador) / s.vidaJugador);
        end
    end
end
return r;
```

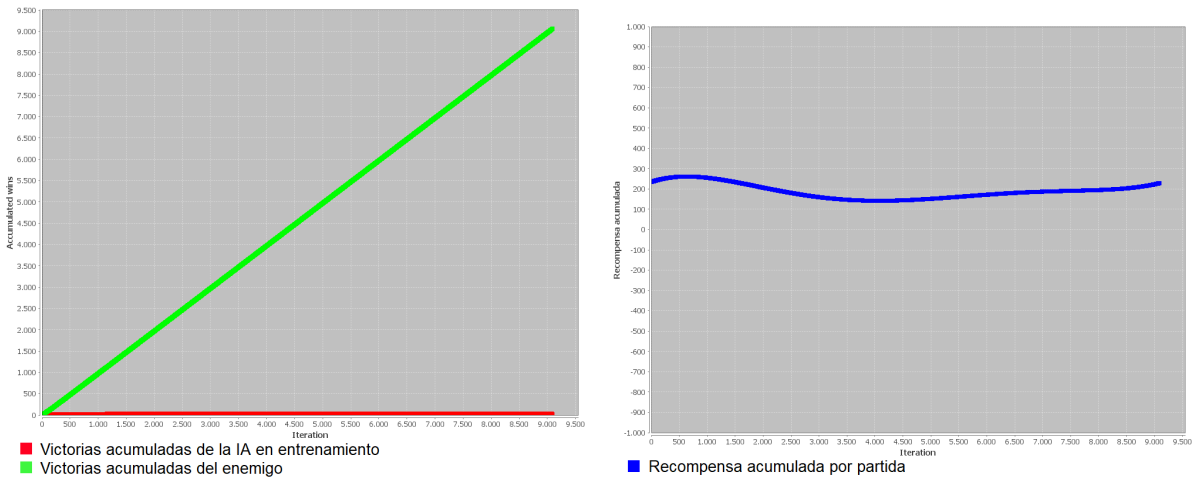
Algoritmo 4: Sistema de recompensas parte 2

Anexo D

Pruebas de la Inteligencia artificial

D.1. Entrenamiento contra un mismo enemigo

El primer entrenamiento fue cuando únicamente se había implementado el algoritmo de *Q-Learning* sin regresión, y previo a algunos cambios como lo de la probabilidad de elegir una acción subóptima, sin embargo, resulta de interés pues demuestra la capacidad del algoritmo en sí sin añadidos. Este entrenamiento fueron un total de poco más de 9000 partidas contra la IA de Terry en nivel difícil, jugadas a lo largo de unas 170 horas.



(a) (b)

Figura D.1: Victorias acumuladas (D.1a) y tendencia de la recompensa acumulada (D.1b) en el entrenamiento de 9000 partidas contra Terry en difícil, reduciéndose ϵ a lo largo de las primeras 3000.

En las gráficas mostradas en la figura D.1 se puede apreciar claramente lo que se comentó en un principio de que las IAs contra las que entrenaba no eran demasiado buenas, esto se ve reflejado en que la IA prácticamente no perdió ninguna partida desde el comienzo del entrenamiento y en que la tendencia de la recompensa acumulada

apenas varió, oscilando ligeramente entre loas 150 y los 250 (aproximadamente). Sin embargo, pese a que no se puede apreciar una mejora en estos dos aspectos, si que se puede ver una clara mejora en el cómo gana, como se ve en la figura D.2.

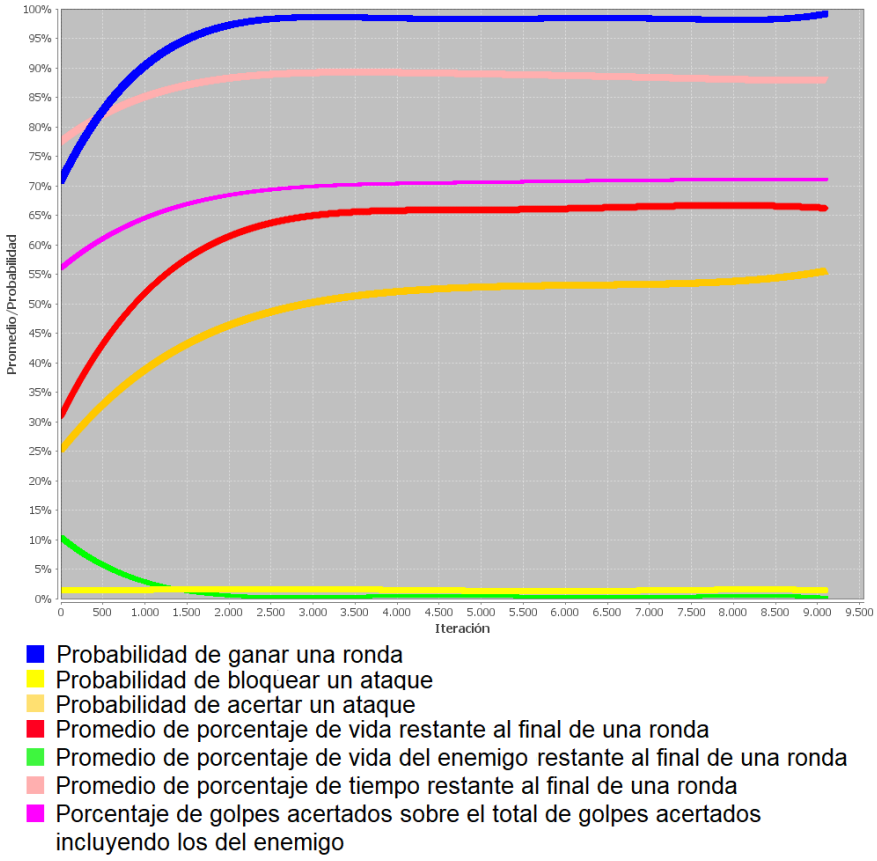


Figura D.2: Métricas del entrenamiento de 9000 partidas contra Terry en nivel difícil, reduciéndose ϵ a lo largo de las primeras 3000.

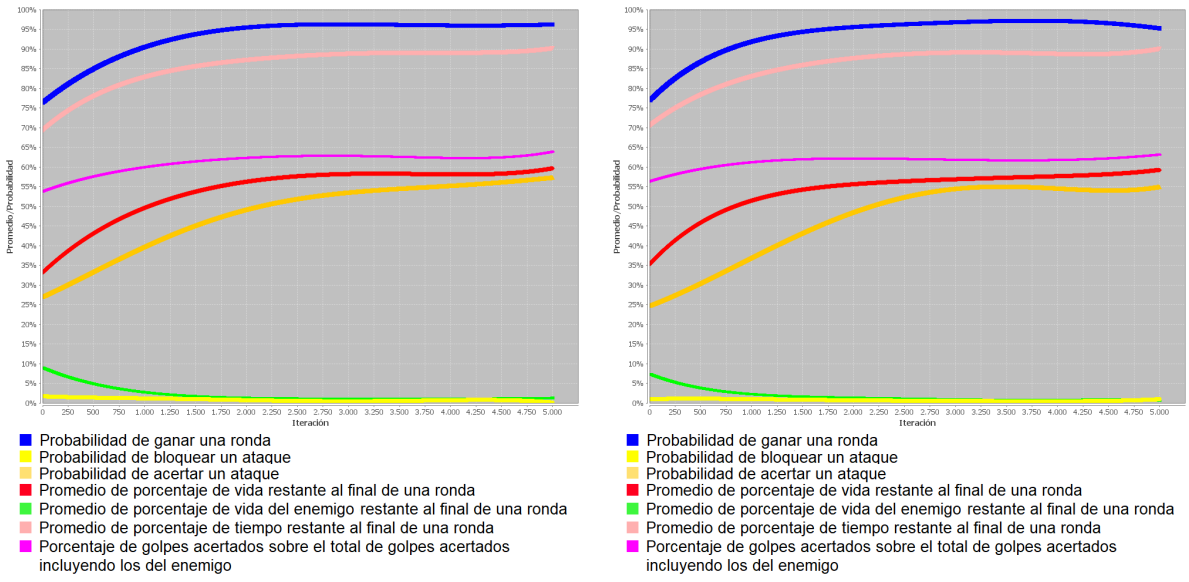
En la figura D.2 se ve como todas las métricas de la IA mejoran (salvo el bloqueo de ataques, porque suele dar mayor recompensa bloquear un ataque con otro ataque que con un movimiento defensivo, y la vida media del enemigo, la cual. obviamente, se busca que vaya decayendo como lo hace), estancándose entorno a la partida 2500-3000 punto en el cual el ϵ se fija a 0.05. Esto lo que demuestra, como era de esperar, es que la mejora en las estadísticas viene por la probabilidad con la que selecciona una acción óptima conocida, es decir, si en vez de hacer que se redujese a lo largo de 3000 partidas se hubiese hecho sobre 6000, la pendiente sería menos acentuada. A su vez, también es muy probable que tendiese a converger a unos mejores resultados, puesto que habría explorado más y cabe la posibilidad de que descubriese pares acción-estado que con 3000 no se descubrieron, y que podrían resultar mejores.

Sin embargo, lo que es innegable es que la mejora a lo largo del entrenamiento es obvia, especialmente en la vida media restante al final de cada ronda (ha aprendido a

esquivar más ataques), en la probabilidad de ganar una ronda (aunque ganase partidas desde el principio, perdía algunas rondas), y en la probabilidad de golpear a un enemigo con sus ataques (ataca menos al aire, lo cual se recompensaba negativamente, y ataca en mejores momentos). El resto de métricas también presentan mejoras, pero no tan marcadas como en los casos nombrados.

D.2. Entrenamientos con y sin apoyo de regresión lineal

Tras este entrenamiento, se incluyó el uso de la regresión lineal, y algún ligero cambio como lo de la decisión subóptima. A continuación se van a mostrar una comparativa entre los resultados obtenidos por dos IAs entrenadas (ambas durante 5000 partidas en aproximadamente 100 horas) contra los mismos rivales (todos los personajes con todas las IAs, siendo los combates aleatorios), usando usa la regresión lineal como apoyo y la otra no. Señalar que no se incluyen las gráficas de victorias acumuladas ni de recompensa acumulada, porque son prácticamente idénticas, ganando ambas prácticamente todas las partidas, y convergiendo ambas entorno a los 350 puntos de recompensa. En la figura D.3 me muestran las métricas de las IAs usando regresión y sin usarla.



(a) (b)

Figura D.3: Métricas del entrenamiento con IA usando regresión (D.3a) y sin usar regresión (D.3b) en un entrenamiento de 5000 partidas contra rivales aleatorios, reduciéndose ϵ a lo largo de las primeras 3000.

Como se puede apreciar en la figura D.3 ambos resultados son prácticamente

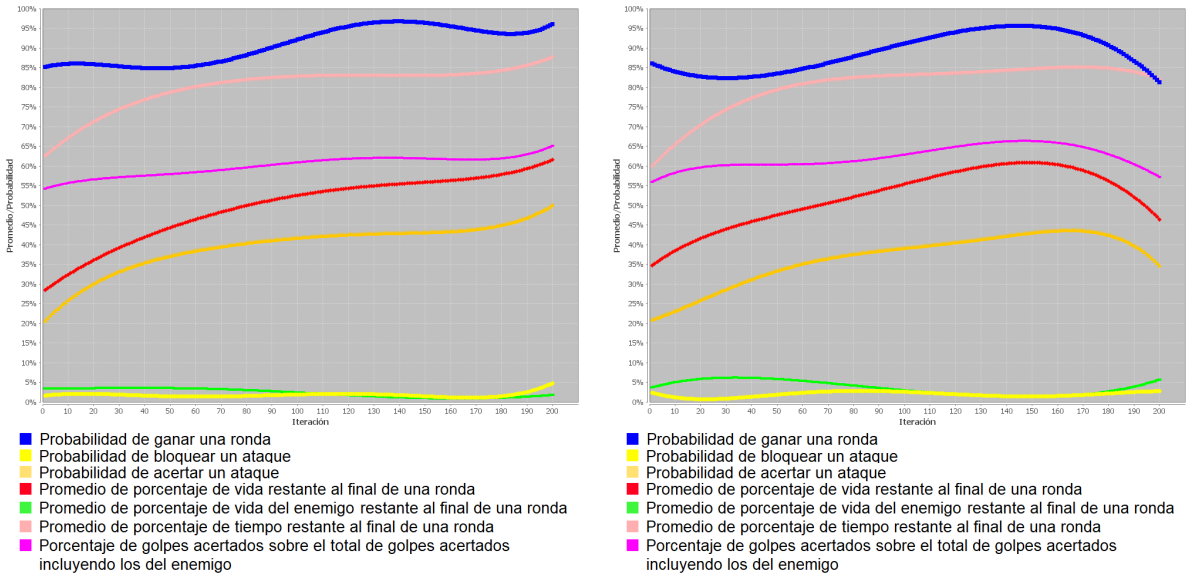
idénticos, siendo la mayor diferencia en la curva de la probabilidad acertar un golpe, que crecen de distinta forma, pero acabando en prácticamente el mismo valor. El que los resultados sean prácticamente idénticos viene marcado por dos razones:

- La primera es que las IAs contra las que entrena son tan simples, que no marca una verdadera diferencia el elegir una acción 'óptima' basada en la estimación de la regresión, frente a un movimiento aleatorio, en los casos de explotación. Ya se ha visto que desde el comienzo del entrenamiento, todas las IAs entrenadas ganan prácticamente todas las partidas desde el comienzo, y que la tendencia de las recompensas prácticamente varía, es decir, el margen de mejora desde un comienzo es reducido, es entendible que no puede haber una diferencia apreciable en este caso.

- La segunda razón, y la verdaderamente relevante, es básicamente el hecho de que el ϵ decrezca tan lentamente a lo largo de 3000 partidas. Como se ha dicho, tiene poco margen de mejora, y la explotación, situación en la que se podría hacer uso de la regresión, tarda mucho en alcanzar un punto en el que la explotación 'mejorada' se pudiera notar. A modo de ejemplo, en este caso hasta la partida 1500 no se explotan las decisiones pasadas con un 50 % de probabilidades, por lo que durante 1500 partidas ha tenido tiempo de visitar la gran mayoría de estados al menos una vez, sin haber tenido que tomar decisiones óptimas demasiado frecuentemente. Es decir, la regresión lineal solo tiene verdadero sentido cuando se quiere hacer que el ϵ se reduzca rápidamente, tanto, que difícilmente habrá visitado muchos estados, es decir, rápidamente se intentará explotar con una muy alta probabilidad, pero habrá muchos estados sin visitar. En esta situación es cuando verdaderamente sale a relucir el potencial del apoyo de la regresión lineal. Señalar que la situación descrita, es el caso de las IAs del servidor.

Por tanto, el que no haya apenas diferencias es debido a que no es una situación en la que se aproveche las capacidades de este algoritmo híbrido. Sin embargo, en el caso de las IAs del servidor, en las cuales el ϵ se reduce a lo largo de 50 partidas en el caso de la IA personal, y 100 en el de la global, es donde sacará a relucir sus verdaderas capacidades. En la figura D.4 se muestra una simulación de este caso (aunque sigue sin ser realista a causa de las IAs contra las que entrena), en la que durante las 100 primeras partidas se va reduciendo el ϵ, y de la 100 a la 200 sería ya la IA 'entrenada', es decir, con ϵ fijo a 0.05.

De nuevo, las diferencias no son demasiado apreciables por la simplicidad de las IAs a las que se enfrentan, pero al contrario que en el entrenamiento de 5000 partidas,



(a) (b)

Figura D.4: Métricas del entrenamiento con IA usando regresión (D.4a) y sin usar regresión (D.4b) en un entrenamiento de 200 partidas contra rivales aleatorios, reduciéndose ϵ a lo largo de las primeras 100.

la diferencias son algo más marcadas. En el caso de la IA entrenada por regresión (D.4a) se puede ver como todos los crecimientos siguen una tendencia más constante, e incluso pasando la partida número 100 sigue mostrando la misma tendencia, mientras que los crecimientos en el caso de la sin regresión (D.4b) son algo más inestables, incluso llegando a un punto en el que muestra una tendencia a decaer.

Las diferencias más claras entre ambos son la probabilidad de ganar las rondas, donde la con regresión muestra una mejor evolución (la sin regresión presenta bastante inestabilidad), y la vida media restante del enemigo, en la que la con regresión la mantiene desde el comienzo prácticamente a cero, mientras que la sin regresión tarda bastante en hacer que tienda también a 0, es decir, tiene más dificultades.

El que la IA sin regresión muestre esa caída al final probablemente se deba a que como casi siempre le tocaría explotar pares estado-acción ya visitados, debido al bajo número de partidas, quedarían muchos por visitar, casos en los que no sabría elegir una acción óptima, mientras que el de la regresión tendría una orientación. Sin embargo, la razón seguramente sea que debido a que en las situaciones de explotación de decisiones pasadas no tenía ningún tipo de guía, probablemente tomase malas decisiones, haciendo que cuando ya casi siempre tuviese que explotar, a lo mejor no conociese demasiadas buenas acciones. Mientras tanto, la versión con regresión en las fases de explotación de estados no visitados, cuanto menos tomaba decisiones algo guiadas, con lo que más tarde cuando siempre tocase explotar, tendría información más útil.

Señalar además, que como se puede apreciar en el caso de la con regresión muestra una tendencia a seguir mejorando, lo cual significa que siguen quedando estados sin visitar (como ocurre a lo largo de las partidas 100 a la 200), en los cuales tomará decisiones razonables basadas en el resto de pares estado-acción, haciendo que durante las explotaciones de esos estados en los que solo se podría escoger una acción aleatoria, 'explore' (puesto que al fin y al cabo no puede explotar ninguna acción al no haber estado nunca) acciones probablemente buenas.

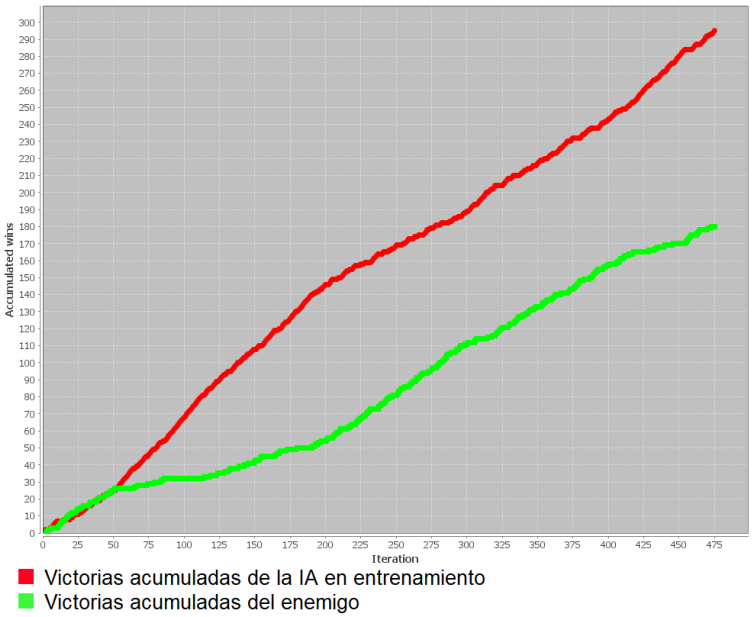
D.3. Enfrentamiento entre dos inteligencias artificiales entrenadas

A continuación se van a mostrar las gráficas (figura D.5) de los resultados del enfrentamiento entre una IA entrenada usando la regresión contra IAs y personajes aleatorios (genérica), y una IA entrenada usando regresión contra una única IA y un único personaje (el cual es el mismo personaje que el que usa el agente).

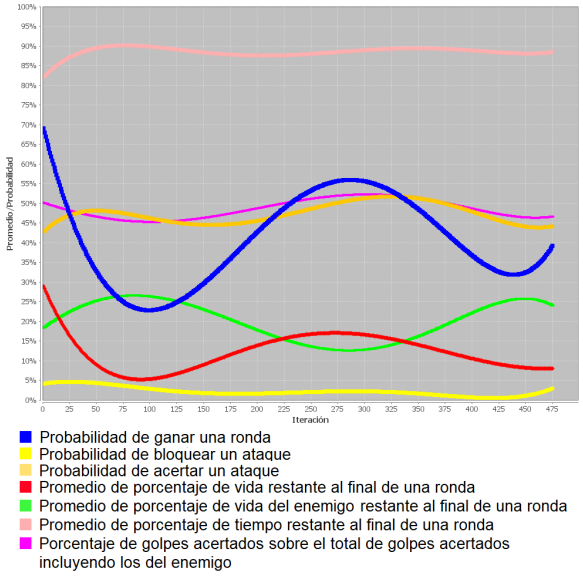
En la figura D.5 se puede apreciar en el número de victorias, que ambas IAs estaban más o menos parejas, al menos hasta la partida 50 (aproximadamente). A partir de ese punto la IA entrenada contra un mismo personaje (el del agente) y una misma IA empieza a ampliar la diferencia en el número de partidas ganadas. Que la dicha IA tome acabe dominando es completamente normal, puesto que esta únicamente ha entrenado contra ese personaje, es decir, tiene más conocimiento sobre ese personaje concreto, puesto que la otra, entrenando el mismo número de partidas, entrenó contra 3 personajes distintos, cada uno con sus características específicas.

El que al comienzo vayan bastante parejos, probablemente se deba a que la IA entrenada contra un único personaje, pese a haber entrenado mucho contra él, no se había enfrentado a estrategias o jugadas tan 'inteligentes' (la IA contra la que entrenó era muy simple). Aun así, en el momento que vio las jugadas de la otra IA y se adaptó ligeramente, fue marcando más y más diferencia con la IA más general. Sin embargo, lo verdaderamente interesante es que la IA más general, pese a no alcanzar el nivel de la otra, no se quedó completamente atrás, y seguía mejorando y ganando partidas con una frecuencia más que razonable.

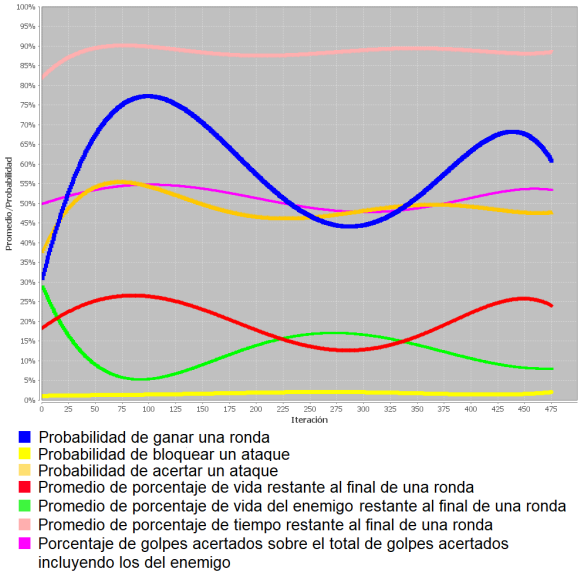
Como ya se ha dicho, lo interesante es el hecho de que pese a que la IA generalizada empezaba con una desventaja obvia por no tener el mismo conocimiento sobre el personaje, esta sigue evolucionándose y adaptándose lo suficientemente rápido como para seguir suponiéndole un reto a la otra IA. Obviamente a su vez la IA que solo se enfrentó a un personaje también sigue evolucionando y superándola poco a poco por más.



(a)



(b)



(c)

Figura D.5: Victorias acumuladas (D.5a) y métricas (D.5b) de la IA entrenada contra enemigos aleatorios, y métricas (D.5c) de la IA entrenada contra Terry en difícil.

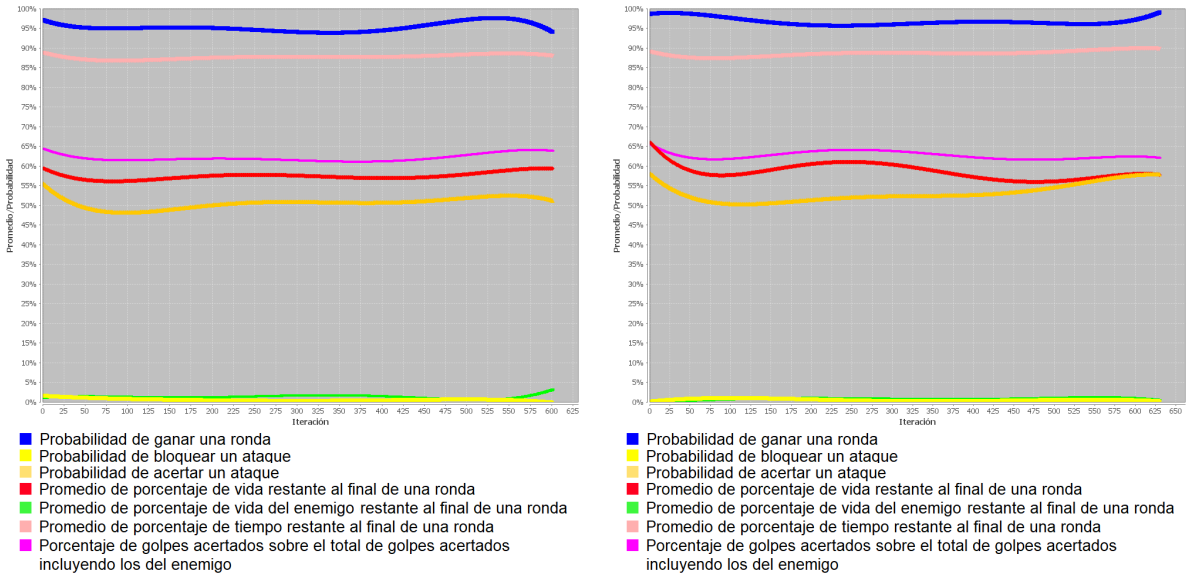
Simplemente con la gráfica de victorias ya se puede apreciar la adaptabilidad que presentan las dos, lo cual es una gran característica, especialmente por lo que se comentó en su momento, de que si por alguna razón los jugadores encontraban una deficiencia de la IA que explotar, esta debería ser capaz de corregirla pronto. Si se observan las gráficas de las métricas, se ve lo oscilantes que son en ambos casos, lo cual vuelve a mostrar esa adaptabilidad que presentan las IAs, ya que si fueran unas IAs muy estrictas pasado el período de entrenamiento, ambas deberían mantener siempre el

mismo comportamiento, y por tanto, las métricas deberían mantenerse mucho más estables (lo cual se ejemplificará en la siguiente comparativa).

Por tanto la conclusión de esta comparativa, es que enfrentándose por primera vez a un rival que les puede hacer frente, son capaces de aprender y adaptarse rápidamente, haciendo que sean mucho más flexibles y útiles para el uso que se le quiere dar.

D.4. Inteligencias artificiales entrenadas contra básicas

La siguiente comparativa (figura D.6), muestra los resultados de cada IA jugando unas 600 partidas contra enemigos aleatorios. El objetivo de esta comparativa, es mostrar como la IA que entrenó contra enemigos aleatorios mantiene las métricas prácticamente como una recta, mientras que las de la IA que se enfrentó solo a un personaje son ligeramente más oscilantes debido a la necesidad de adaptarse.



(a) (b)

Figura D.6: Métricas de la IA entrenada contra enemigos aleatorios (D.6a), y de (D.5c) de la IA entrenada contra Terry en difícil (D.6b).

La diferencia no es excesivamente marcada, debido de nuevo a la simplicidad de las IAs, y en este caso en concreto a que la IA que entrenó contra una única IA entrenó contra la más difícil (en los enfrentamientos aleatorios se enfrenta a todos los niveles de las IAs), con lo que pasar de entrenar con la más difícil a entrenar con esta y otras más fáciles no le perjudicó demasiado. Sin embargo, pese a ello se puede ver como la línea de la vida media restante de la IA (una de las métricas más importantes) es bastante más oscilante en el caso de la que entrenó únicamente contra un personaje

(figura D.6b), que en el caso de la general (figura D.6a), en la cual es poco menos que una línea recta. Lo más probable es que esa oscilación la produjese la aparición de los nuevos personajes, ya que como se ha dicho, el nivel de las IAs difícilmente le iba a suponer un problema.

Con esta comparativa (figura D.6) de nuevo se vuelve a mostrar la adaptabilidad de las IAs, en este caso al enfrentar la IA que solo conocía un personaje contra todos los personajes. Sin embargo, lo relevante es que en la gráfica de los resultados de la IA que ya había entrenado bajo las mismas circunstancias, todas las líneas son prácticamente rectas, corroborando lo que se dijo en la anterior comparación de que esas oscilaciones demostraban que evolucionaban y se adaptaban, y que sino fuese así serían líneas casi rectas, como en este caso.

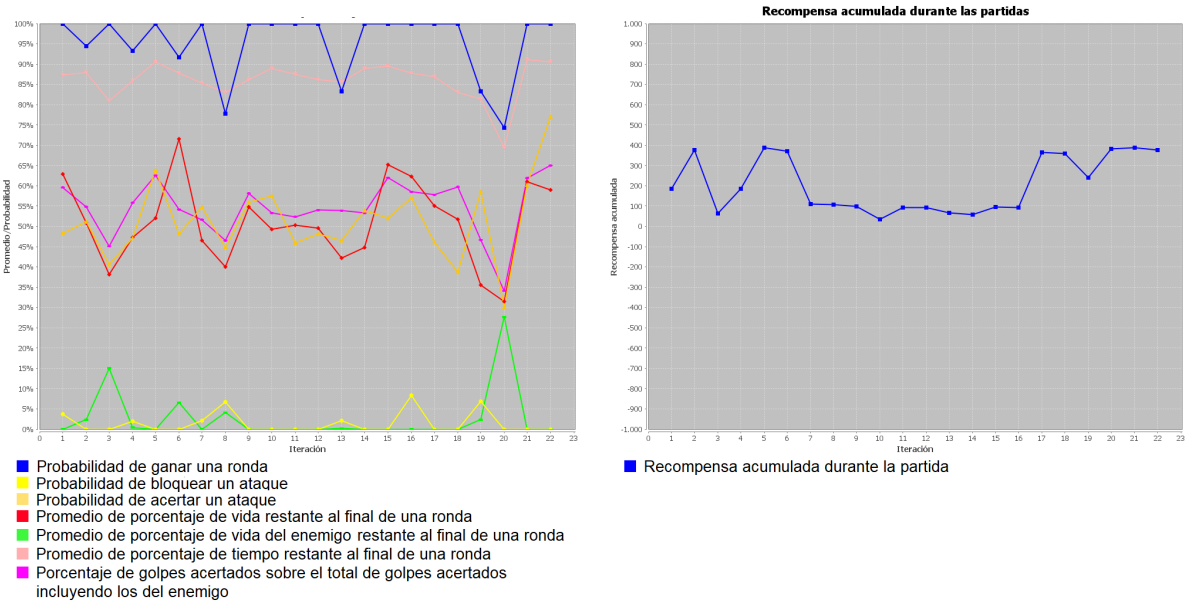
D.5. Humanos contra una Inteligencia artificial entrenada

La última prueba fue poner a prueba las capacidades de una IA entrenada contra IAs mediocres, concretamente la IA entrenada contra personajes aleatorios y usando regresión lineal, contra jugadores humanos. Pese a que los resultados contra las IAs de entrenamiento eran muy buenos, se esperaba que contra humanos no obtuviese tan buenos resultados pues son mucho más inteligentes que contra lo que había entrenado.

Para la prueba participaron un total de 6 usuarios, jugando cada uno de ellos un total de 22 partidas. Y el resultado del experimento fue que la IA se desenvolvía bastante mejor de lo esperado, pues pese a que obviamente no obtenía tan buenos resultados como contra las IAs de entrenamiento, no se alejaba demasiado ellos. Simplemente el hecho de que entre todos los jugadores consiguiesen únicamente dos victorias en todas las partidas jugadas, muestra que suponía un verdadero reto.

En la figura D.7a se muestran las métricas medias de los 6 jugadores en sus 22 partidas, y se aprecia claramente como en general todas las métricas alcanzan unos niveles más que satisfactorios, ganando siempre con una probabilidad media superior a 0.75, y con una vida media al final de cada partida de entre el 30 % y el 70 %. Respecto a la figura D.7b decir que se puede ver como en promedio siempre mantiene una recompensa positiva, lo cual era esperable teniendo en cuenta el resto de métricas. A modo ilustrativo, en la figura D.8 se muestra las estadísticas de un usuario.

Terminar diciendo, que las dos únicas victorias que consiguieron los usuarios se consiguieron abusando de un defecto en las cajas de colisiones de un personaje (Mai), por el cual la mayoría de ataques del personaje de la IA no le podían dar. Por tanto, las derrotas de la IA no fueron debidas a la propia IA en sí, sino a un mal diseño de



(a) (b)

Figura D.7: Resultados medios (D.7a) y recompensa acumulada media (D.7b) de 6 personas contra la IA entrenada contra todos los personajes en sus primeras 22 partidas.

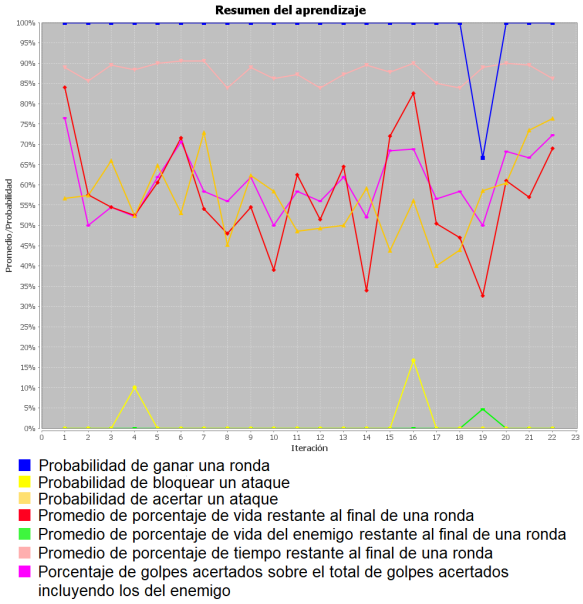


Figura D.8: Métricas de la IA entrenada contra enemigos aleatorios, contra un usuario aleatorio en sus primeras 22 partidas.

un personaje proveniente de las prácticas de Videojuegos.

Anexo E

Creación de la base de datos

Para la creación de la base de datos hay que tomar tres decisiones cruciales, la primera el modelado de los datos, el gestor de bases de datos, y el modelo conceptual de la base de datos.

- Para el caso de este proyecto se eligió el modelado relacional, principalmente por el dominio de dicho modelado frente a un modelo NoSQL indefinido dependiente de cada gestor, pues no hoy ningún tipo de estándar para estos.

- Para el gestor de bases de datos, se quería un gestor fiable y *open source*, por lo que se eligió PostgreSQL (desde la adquisición de MySQL por Oracle, en su versión gratuita es algo más restrictiva).

- Elegido el modelado de los datos y el gestor de bases de datos, faltaría el modelo conceptual, para el cual se usó el modelo entidad-relación extendido. El modelo conceptual resultado se ve en la figura E.1.

Tras estas tres decisiones, el siguiente paso era pasar a pensar en la implementación, para lo cual hay un paso intermedio, que traduce el modelo conceptual, a una estructura representable por el modelo de datos elegido (relacional en este caso). Este paso es el diseño lógico de la base de datos, el cual traduce del entidad relación a una colección de relaciones,que se corresponderían con las tablas que finalmente tendrá la base de datos (omitiendo las posibles optimizaciones).

En la figura E.1 se puede ver el modelo relacional resultante. Los únicos detalles destacables son el que las fechas de inicio de sesión (un atributo multievaludado) pasaron a tener su propia relación, que la relación de amigos pasó a tenre su propia tupla por ser una relación N a M, y que la especialización de la partida en partida clasificatoria pasó a ser un conjunto de atributos en la tupla de partida (actuando algunos de ellos como discriminadores).

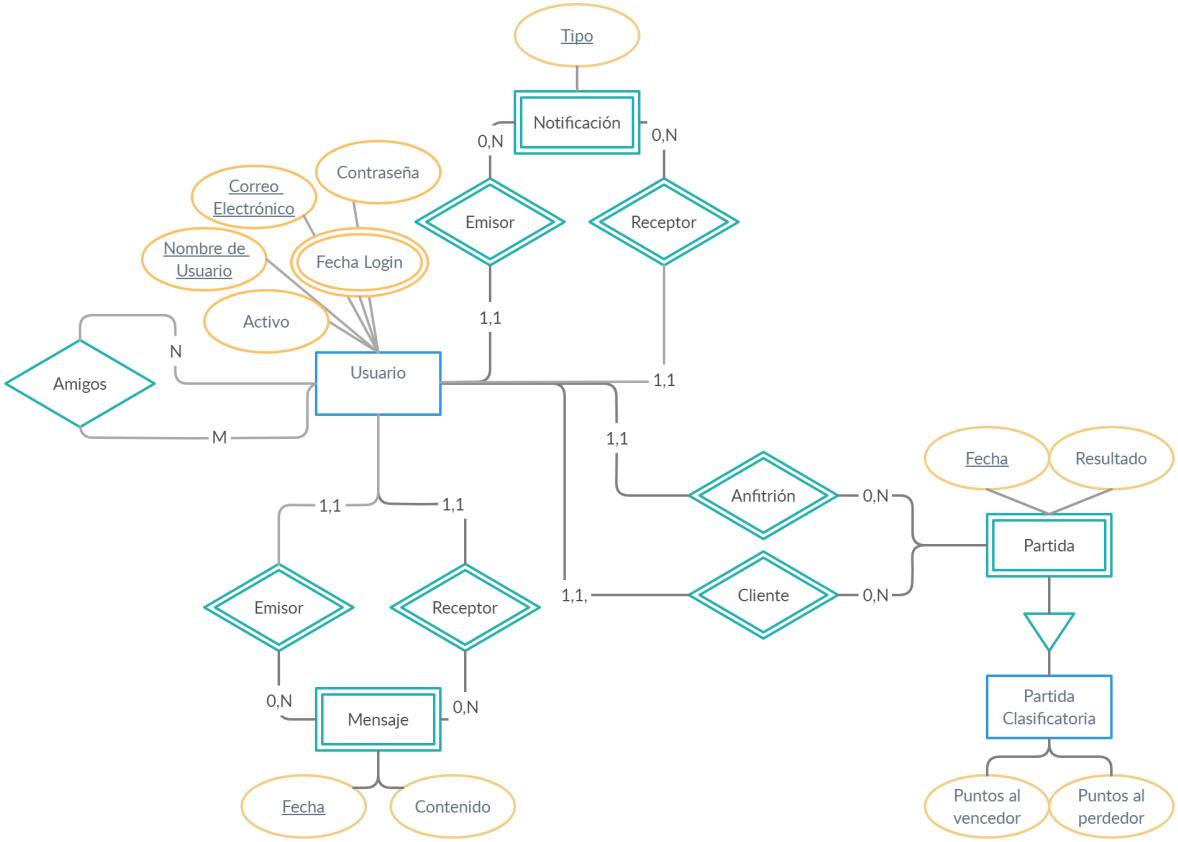


Figura E.1: Modelo entidad relación de la base de datos.

Dominios
tpTipoNotificacion = AMISTAD, MENSAJE
tpFechaExacta = Día:Mes:Año-Hora:Minutos:Segundos:Milisegundos
tpResultado = EMPATE, ANFITRIÓN, CLIENTE
Clave primaria Clave alternativa Clave extranjera
Relaciones
Usuario = (apodo:cadena(3..10), email:cadena(3..100), contraseña:cadena(3..100), verificado:booleano)
Login = (usuario:cadena(3..10), fecha:tpFechaExacta)
Amigos = (solicitante:cadena(3..10), solicitado:cadena(3..10))
Mensaje = (emisor:cadena(3..10), receptor:cadena(3..10), fecha:tpFechaExacta, contenido:cadena)
Notificación = (emisor:cadena(3..10), receptor:cadena(3..10), tipo:tpTipoNotificacion)
Partida = (anfitrión:cadena(3..10), cliente:cadena(3..10), fecha:tpFechaExacta, resultado:tpResultado,
clasificatoria:booleano, puntosAnfitrión:entero, puntosCliente:entero)

Figura E.2: Diseño lógico de la base de datos.

El último paso antes de la implementación, era optimizar el diseño lógico pensando en que consultas se van a hacer, y el coste de estas. La primera optimización era el meter las partidas normales y clasificatorias bajo una misma relación,

debido a que si estuvieran separadas se harían *JOINs* (una operación muy costosa computacionalmente) muy frecuentemente. La segunda fue introducir como clave primaria tanto en mensajes como en partidas un identificador numérico único, para simplificar el índice (ya que nunca que va a buscar por la clave completa) y porque dicho identificador a su vez establece un orden cronológico más simple y rápido de comparar que una fecha completa (solo interesa saber el orden de las partidas). Otra, fue separar las notificaciones en dos relaciones en vez de una con un atributo diferenciador, pues casi siempre se preguntaría por uno o por otro. La última fue incluir en el usuario cierta información, la cual se puede deducir a través del resto de relaciones (como el número de partidas jugadas), pero que al ser una información que se consultará frecuentemente, es mejor tener un acceso rápido, pese a tener redundancia, que tener que calcularlo siempre.

Diseñada la base de datos, el último paso es implementarla e integrarla en el servidor para que este pueda hacer uso de ella. Para la implementación, en vez de crear la base de datos a base de sentencias SQL, se decidió usar el mapeo objeto-relacional [7] para simplificar el acceso a la base de datos desde el servidor y el uso de estos en el mismo. Concretamente, se usó el estándar JPA [8] usando las bibliotecas de Hibernate puesto que se tenía experiencia con dichas tecnologías. Señalar que haciendo uso de esta tecnología, esta es capaz de crear las tablas correspondientes a los objetos. A su vez, todas las relaciones entre entidades/objetos, restricciones de los atributos e incluso la definición de como escribir en la base de datos la herencia de la partida normal y clasificatoria, todo se establece mediante etiquetas.

Respecto a la implementación de las relaciones en JPA, destacar que todas las relaciones son bidireccionales, haciendo que desde una clase siempre se pueda acceder a todos sus relacionados. Sin embargo, por defecto Hibernate al cargar un objeto, carga todos con los que está relacionado, haciendo que se pudiera cagar mucha información innecesaria, reduciendo el rendimiento innecesariamente, por lo que para evitar este problema, se marcaron todas las relaciones con *lazy fetch*, el cual no carga los relacionados hasta que se intenta acceder a ellos.

Finalmente, decir que para hacer más simple el guardado y la lectura de datos de la base de datos, se hizo una clase intermedia para abstraerlo (en vez de tener que declarar una transacción, ejecutarla y consolidar, simplemente llamar a una función guardar).

Anexo F

Ejemplos de comunicación

F.1. Ejemplo comunicación UDP

La base de la comunicación UDP mediante Datagramsocket, es declarar el *socket* indicándole el puerto a usar, para enviar crear un paquete al que se le indica la IP y puerto destino, y para recibir simplemente se recibe cualquier paquete que llegue al puerto designado. Un ejemplo muy simple del uso de esta clase para enviar un mensaje y luego recibir uno se puede ver a continuación.

```
//Instanciación del socket
int portReceive; //puerto en el que recibir
DatagramSocket socketUDP = new DatagramSocket(portReceive);

//Envío de un mensaje
byte[] bufSend = ("MENSAJE A ENVIAR").getBytes();
InetAddress destinationAddress; //dirección/ip a la que enviar
int portSend; //puerto al que enviar
DatagramPacket packet = new DatagramPacket(bufSend, bufSend.length,
                        destinationAddress, portSend);
socketUDP.send(packet);

//Recepción de un mensaje
byte[] bufReceive = new byte[65535];
DatagramPacket packetReceived = new DatagramPacket(bufReceive,
                                        bufReceive.length);
socketUDP.receive(packetReceived);
InetAddress sourceAddress = packet.getAddress();
int sourcePort = packet.getPort();
String message = new String(packet.getData(), 0, packet.getLength());
```

F.2. Ejemplo cliente-servidor TCP

Previamente a que se intentasen conectar los clientes (como es obvio), el servidor debía haber iniciado un ServerSocket para poder aceptar dichas conexiones. Por tanto, la forma más simple de gestionar las comunicaciones en el servidor era que al iniciarlo se abriese un ServerSocket que estuviese constantemente esperando por conexiones, y cuando recibiese una crear un hilo que a partir del Socket generado de la aceptación de la conexión se encargue de comunicarse con el cliente correspondiente. Para una primera versión, se hizo que solo se enviasen y recibiesen cadenas, tal y como se aprecia en el siguiente ejemplo básico.

```
/*EJEMPLO SERVIDOR*/
int portReceive;// puerto en el que recibir
//Instanciación del socket como ServerSocket
ServerSocket serverSocket = new ServerSocket(portReceive);
//Aceptación de una conexión
Socket newCon = serverSocket.accept();

//En un hilo aparte
DataInputStream  input = new DataInputStream(new BufferedInputStream(socket.get
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
//Recibir un mensaje
String msg = in.readUTF();
//Enviar un mensaje
out.writeUTF("Hola");

//Cerrar la conexión
in.close();
out.close();
socket.close();

/*EJEMPLO CLIENTE*/
int serverPort; //puerto al que enviar
InetAddress serverAddress; //ip/dirección a la que enviar
//Instanciación del socket
Socket socket = new Socket(address, port);
DataInputStream  input = new DataInputStream(new BufferedInputStream(socket.get
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

```
//Enviar un mensaje
out.writeUTF("Hola");
//Recibir un mensaje
String msg = in.readUTF();

//Cerrar la conexión
in.close();
out.close();
socket.close();
```

Anexo G

Detalles de la implementación de la comunicación

En este anexo se explicarán ciertos detalles de la implementación de la comunicación, que pueden resultar de interés en caso de querer comprender mejor cómo se llevaron a cabo .

G.1. Comunicación UDP

La estructura final de la comunicación UDP se basa en el ejemplo mostrado en el anexo F.1, pero usándolo de forma que cumpla las necesidades nombradas en el apartado 4.2. Para introducir la identificación numérica de los de los mensajes y marcarlos como que requieres confirmación o no, simplemente se hizo uso de un carácter reservado (';') en cual nunca iba a aparecer en los mensajes para el uso que se le iba a dar. Se hizo que todos los mensajes pasasen a ser una cadena con el siguiente patrón identificador;requiereConfirmación;mensajeOriginal, de forma que en una única cadena mensaje está toda la información necesaria.

Para hacer la función de recepción de mensajes en base a identificador no bloqueante, se creó un proceso que recibe mensajes constantemente, y almacena el mensaje en base al identificador (extraídos del mensaje completo con el patrón anteriormente nombrado). De esta forma, la función de recepción no bloqueante simplemente mirará en los mensajes almacenados si hay alguno con el identificador deseado y responderá acorde a ello. Señalar que el proceso nombrado, también se encargará de revisar si los mensajes recibidos requieren o no confirmación, y en caso de requerirla le enviará dicha confirmación.

Además de la función de envío básico y de recepción en base al identificador, destacar la creación de una función que envía un mismo mensaje hasta 30 veces cada cierto tiempo, esperando recibir una confirmación (esto se usaría para mensajes relevantes los cuales se necesita que se reciban, puesto que UDP no asegura la recepción

de los mensajes), y en caso de no recibir la confirmación notifica que no se ha logrado enviar correctamente (si se han perdido 30 mensajes casi seguro ha habido algún problema por el que el otro cliente no le puede responder).

G.2. Comunicación TCP

Una de las modificaciones de la comunicación TCP respecto del ejemplo en el anexo F.2, fue introducir el que la comunicación permitiese enviar objetos, pues se iba a necesitar para información más compleja que una simple cadena, como el perfil (número de partidas ganadas y perdidas de cada tipo e historial de partidas recientes). Esto era fácil, simplemente reemplazar DataInputStream y DataOutputStream, por ObjectInputStream y ObjectOutputStream, respectivamente, y usar sus funciones de lectura y escritura (se podrían tener abiertos los cuatro, pero a un mayor costo como es obvio, lo cual si se puede evitar es preferible, debido al número desconocido de jugadores). Sin embargo, el objeto recibido no se conoce de que clase es (es un Object de Java, la superclase de todos los objetos), por lo que a priori no se sabe si es una petición (se representarán como cadenas por simplicidad como se verá en el siguiente apartado), si un objeto recibido como respuesta a una petición o qué, solo quedaría intentar hacer conversiones a cadena y a otros objetos para inferir el tipo, lo cual tiene un coste computacional y puede dar lugar a errores fuera de control.

Para solucionar ese problema, se creó una estructura que haría de envoltorio a los verdaderos datos a enviar, y la cual contendría la información mínima necesaria para saber como tratarlos. Básicamente se creó una clase paquete (que implementa `Serializable` [1], pues era requisito para poderlo enviar), que contendría la siguiente información:

- El identificador del mensaje que indica el çanal.ªl que pertenece.
- Si requiere confirmación o no (aunque no sería necesario gracias a TCP).
- Si es un objeto o una cadena, de esta forma se diferenciarán claramente las peticiones de todo lo demás.
- La cadena a enviar, en caso de no enviarse un objeto.
- El objeto a enviar, en caso de que no se enviase una cadena.

Usando esta clase siempre durante la comunicación se sabía siempre que se ha de hacer la conversión a la clase paquete, y así obtener fácilmente la información de

[1] `https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html`

los datos originales. Decir que la diferenciación entre cadena y objeto se usaba para almacenarlos por separado, y así cuando se estuviese únicamente mensajes de texto con un identificador concreto, los objetos no interviniesen, y viceversa. También se separaron pues sería normal que una petición fuese acompañada de un objeto (por ejemplo al mandarle el cliente la petición de guardar la IA y este le mandase la tabla actualizada), haciendo que, si no se separasen, fácilmente uno sobrescribiera al otro en la estructura de almacenaje, perdiendo la información de uno de los dos.

Señalar que el objeto dentro de la estructura no era la superclase Object como se podría pensar, puesto que se exige que el objeto que se escribe en el flujo implemente Serializable, por lo que se creó una clase interfaz (SendableObject) que únicamente implementaba Serializable, y de la cual heredarían todos los objetos que se quisiesen enviar. De esta forma, todos los objetos que se enviasen podrían encajar en el atributo del paquete al heredar de dicha clase, y ya posteriormente en el receptor hacer la conversión cuando correspondiese.Señalar que en la estructura los objetos recibidos se almacenarían como si perteneciesen a dicha clase, y serían los propios procesos que solicitasen el objeto los que hicieran la conversión (esto se pudo hacer porque, como se verá, hasta que tanto cliente como servidor no han terminado de tratar una petición, no se manda otra).

Los objetos que se enviaban eran o simplificaciones de los objetos que representaban entidades de la base de datos (por ejemplo una partida contenía las relaciones con otros objetos, mientras que el objeto enviado únicamente contenía la información relevante de la partida en sí), o información mas procesada (como sería el perfil, con número de partidas e historial de partidas recientes). Independientemente de cual de los dos casos se tratase, había que hacer una conversión de un objeto a otro, los cuales en principio no tienen relación por lo que la conversión directa no es posible, por lo que se hizo una clase simple para hacer esta conversión. Decir que a todas las clases que fuesen enviadas a través del canal de comunicación, se les tuvo que asignar un código de serie UID, para asegurar que todas las máquinas supiesen que la clase enviada y la recibida eran la misma.

Anexo H

Concurrencia en el servidor

La estructura que actuaba como gestor del servidor, contenía la información de los usuarios conectados, con sesión iniciada, de los que estuviesen buscando partida, y de los que hubiesen invitado a otro usuario a jugar una partida. Toda esta información era memoria compartida y presentaban posibles problemas derivados de la concurrencia [10], por ejemplo, si muchos usuarios solicitasen buscar partida simultáneamente, la información de los usuarios en busca de partida sería muy sensible, pudiéndose dar situaciones como emparejar a una misma persona con más de un rival.

El conjunto de acciones que interaccionan con la base de datos, como registrar una partida, recuperar el perfil de un jugador o cargar la lista de mensajes de un usuario, no presentaban un problema en cuanto a aspectos de concurrencia, pues ya se encargaba el propio gestor de bases de datos (e Hibernate en la unidad de persistencia) de gestionar eso. Aunque en algún momento puntual se leyese una información sin actualizar (por ejemplo cargar un perfil justo cuando el jugador acaba de terminar una partida, y que esta no aparezca refleja), no supone ningún tipo de problema pues el usuario no lo percibe, y la siguiente vez que lo vuelva a mirar estará actualizado.

Con el conjunto de interacción entre usuarios tampoco había problema, pues pese a que durante el proceso de las notificaciones hay dos puntos de memoria compartida, la base de datos para el registro del mensaje, y la lista de usuarios conectados, estos no resultan de riesgo. En el primer caso, como ya se ha dicho, las escrituras y lecturas en el servidor no presentan problema. En el segundo caso, si bien es cierto que la lista de usuarios conectados es un punto en el que controlar la concurrencia durante la actualización, no supone un riesgo durante la lectura, pues en el peor de los casos se intentaría mandar un mensaje de notificación a un usuario desconectado, lo cual no supone un problema.

Por contra, el conjunto de acciones que interaccionaban con el servidor y modificaban las estructuras de memoria compartida, suponían un riesgo, por lo que todas dichas acciones debían ejecutarse en exclusión mutua, para lo cual se usaron

los métodos sincronizados[1] de Java (convirtiéndose en cierto modo en lo que se conoce como monitor[2]). Entre las acciones de riesgo cabe destacar el conectar usuarios y buscar emparejamiento de partida a los usuarios.

[1] `https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`
[2] `https://es.wikipedia.org/wiki/Monitor_(concurrencia)`

Anexo I

Gestión de la Inteligencia Artificial en el servidor

En este anexo se busca hacer mención de como se gestionaron las IAs del servidor, pues no era tan simple como la mayoría de peticiones, en las que el cliente pregunta por una información, y el servidor le devuelve dicha información.

Para el caso de la IA personal, el cliente primero le solicitaría al servidor la información de la IA en el estado actual, concretamente la tabla de valoración de pares estado-acción y la tabla de pares visitados. El servidor le mandaría la información como un objeto compuesto por dos matrices y un número real (el ϵ en base al número de partidas jugadas), y a partir de estos el cliente generaría el modelo de regresión lineal. Una vez generado el modelo, se iniciaría la partida y se esperaría a que el jugador terminase. Terminado el enfrentamiento, el cliente le mandaría las dos tablas actualizadas al servidor y este las guardaría directamente (sería el único actualizando esas tablas al ser la IA personal).

Para el caso de la IA global, sería exactamente igual hasta el punto en el que se termina la partida, a partir de ese punto el tratamiento cambia. Terminada la partida, el cliente le mandaría al servidor la lista de transiciones, y este, al recibirlas, entrenaría la IA globlal dentro del propio servidor con ellas, en exclusión mutua con cualquier otro entrenamiento de IA global. La razón de esto es que si por ejemplo dos usuarios intentasen actualizar la IA global a la vez, el resultado sería que el entrenamiento de una de las dos partidas de entrenamiento se habría perdido. De esta forma, se puede entrenar la IA de forma secuencial a través de las transiciones. El resultado no sería el mismo que el de haber jugado una partida y luego otra, pero desde luego sería mejor que el resultado de entrenar en base a solo una de las dos.

Señalar que al entrenar la IA personal también se mandaban las transiciones pese a que se guardaban las tablas directamente, porque dichas transiciones se podrían aprovechar para entrenar la IA global (sería el equivalente a haber jugado una partida

en la que siempre a tocado explorar, es decir, elegir una acción aleatoria). De nuevo, no produciría el mismo resultado que el que hubiera jugado una partida propia, pero sigue siendo información útil para entrenar las tablas.

Anexo J

Prototipo de interfaz de usuario

Con el servidor ya creado, y con la mayoría de los servicios implementados (pues aún después de empezar el prototipo se fueron introduciendo algunos más), se comenzó el desarrollo de una interfaz de usuario básica que permitiese acceder a todos los servicios, pues ningún usuario estaría satisfecho con un juego con el que tiene que interaccionar por terminal.

Para la creación de la interfaz, o mejor dicho prototipo de la interfaz pues era muy básica, se usó la librería gráfica de Java, Swing[1], la cual proporcionaba lo suficiente para poder proveer los servicios con una estética aceptable.

J.1. Diseño de la interfaz

Para el estilo se procuró mantener la esencia del juego base original, manteniendo el estilo de juego retro, pero introduciéndole la interacción por ratón (la cual no estaba en el base), pues por lo general los usuarios prefieren navegar con ratón que con teclado. En la figura J.1 se ve el menú principal, donde se puede apreciar cómo se mantiene el estilo retro, pero se introduce la interacción por ratón, en este caso principalmente a través de botones.

Se procuró hacer un diseño minimalista que no se percibiese sobrecargado, y permitiese al usuario reconocer fácilmente con que partes de la interfaz podría interaccionar y cuales no. Para que el usuario pudiese discernir rápidamente con qué podría interaccionar, todos esos componentes están bordeados con un fino marco, de forma que todos aquellos que no poseyesen dicho marco no permitirían interacción.

A su vez, para los componentes que permiten interacción se han usado siempre dos colores que se pueden apreciar en la figura J.1 en los botones centrales, y en los amigos de la derecha. El color oscuro se usaba para los botones con mayor relevancia o que abriesen alguna característica importante (los botones de partida, clasificación o añadir

[1] `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`

Figura J.1: Menú principal del modo en red.

amigo por ejemplo), o confirmaciones importantes (iniciar sesión, registrarse o cambiar contraseña), y el color más claro características menos relevantes como confirmaciones, selecciones o introducciones de texto. La única excepción a estos dos colores en cuanto a interacción se refiere, sería en el menú de interacción con los amigos que saldría al pulsar sobre ellos, con la única intención de indicar al usuario qué amigo habría seleccionado y hacer que el menú fuera coherente con la selección (figura J.2).

Figura J.2: Selección de un amigo y menú emergente.

Otra indicación de interacción sería la barra de desplazamiento vertical en un color llamativo, la cual aparece en la clasificación, la lista de amigos, el chat y el historial, en caso de que hubiese más entradas (ya sean jugadores, amigos, mensajes o partidas) de las que se pudieran mostrar en la sección de interfaz dedicada a dicha componente. En la figura J.3 se puede apreciar la barra de desplazamiento junto con unas flechas, indicando que se puede subir y bajar. Señalar que en el historial (figura J.3) para indicar quien ganó y quien perdió la partida, se escribían los nombres en verde y en rojo respectivamente, y no se hizo ningún añadido más, pues se consideraba que era una forma de expresión lo suficientemente estándar, como para que cualquier usuario

la entendiese.

Figura J.3: Historial de partidas en el perfil de un usuario.

Finalmente, decir que para que fuera más fácil distinguir qué era el juego base y qué lo nuevo, todo lo introducido en este proyecto está agrupado en una opción de menú, y a partir de ahí todo es nuevo. Concretamente al navegar por el menú básico, se da la opción de elegir entre modo local (juego base) y modo en red (todo lo nuevo).

Por tanto el resultado, fue una interfaz muy básica desarrollada con Swing, pero que procuró mantener una estética acorde a la del juego original. Manteniendo siempre una interfaz coherente e intuitiva que resultase cómoda y fácil de aprender.

J.2. Características del prototipo

A continuación, se presentará una lista de todos las características a las que el usuario tiene acceso por medio de la interfaz. Señalar que se omitirán interacciones básicas como cerrar sesión, retroceder, salir del juego, subir/bajar volumen o cancelar interacciones (por ejemplo cancelar el buscar partida), pues se dan por sobrentendidas.

- Registrarse, para lo cual se le pide un nombre de usuario, una contraseña la cual debe confirmar repitiéndola y un correo electrónico. Al darle al botón de confirmar registro, se le pedirá un código de verificación, el cual le habrá llegado al correo electrónico introducido. En caso de que en el formulario se hubiese introducido alguna información inválida, esto se le notificaría al usuario.

- Iniciar sesión, para lo cual se le pide un usuario y una contraseña, que el servidor verificará. En caso de que la información fuese errónea se le notificará. En caso de que se hubiese cerrado el juego durante el registro sin verificar la cuenta, al intentar iniciar sesión le volvería a pedir el código de verificación.

- Recuperar la cuenta en caso de haber olvidado la contraseña (se necesita el nombre de usuario), proceso el cual cambia automáticamente la contraseña por una aleatoria, y se la manda al correo electrónico del usuario.

- Acceder al perfil del propio usuario, donde se le mostrará la información más importante de este como jugador (nombre de usuario, puntos competitivos, partidas ganadas y perdidas e historial de partidas recientes).

- Cambiar la contraseña (se encuentra en el perfil), pidiendo la contraseña original para autentificar, y la nueva contraseña por la cual se desea cambiar. Al confirmar, le notificará si se ha actualizado la contraseña correctamente o si hubo algún problema.

- Enviar solicitud de amistad, para lo cual únicamente se pide el nombre de usuario a añadir.

- Aceptar o rechazar solicitudes de amistad recibidas. En caso de aceptar alguna, la lista de amigos se actualizaría para que apareciese el nuevo amigo.

- Borrar amigos, caso en el cual se le actualizaría la lista de amigos al amigo borrado y al propio usuario.

- Ver el perfil de un amigo, con la misma información que el del propio jugador, pero sin la opción de cambiar contraseña.

- Intercambiar mensajes con cualquier amigo a través de un chat el cual no ocupa la pantalla completa, sino que se abre como una pequeña ventana emergente. En caso de no tener abierto el chat con un amigo que le ha enviado un mensaje al usuario, en la lista de amigos saldría el nombre de dicho amigo en otro color, para indicar que se tienen mensajes pendientes.

- Invitar a cualquier amigo a jugar una partida y esperar su respuesta. En caso de que la aceptase empezaría la selección de personaje, y en caso contrarío se le notificaría que está ocupado. Mientras el usuario espera la respuesta también puede cancelar la invitación.

- Aceptar o rechazar las invitaciones a partidas de amigos, las cuales aparecen como una ventana emergente.

- Buscar, y jugar, partida normal o clasificatoria, solicitando al servidor un rival al que enfrentarse, y si el servidor le encontrase partida, la empezaría automáticamente. En cualquier momento el usuario puede cancelar la búsqueda. En el caso de que la partida fuese clasificatoria, se actualizarían los puntos clasificatorios en base al resultado de la partida.

– Jugar contra la IA personal y la global, actualizando las IAs en base a la partida jugada.

– Ver la clasificación de los jugadores en base a los puntos clasificatorios de cada uno.

J.3. Manual de usuario

Para llegar hasta el modo en red (se omite el modo local pues era parte de las prácticas), simplemente se ha de seleccionar jugar y a continuación seleccionar modo en red (figura J.4). A partir de este punto accede a todas las características desarrolladas para el proyecto. Señalar que para llegar al modo en red se ha de navegar por la interfaz con las flechas del teclado y con la tecla *enter*, y que si es la primera vez que se accede al modo en red, tardará un poco en pasar a la siguiente pantalla (el resto de veces irá mucho más rápido).

Figura J.4: Interfaz - Selección de modo de juego.

Una vez accedido al modo en red, en caso de no tener cuenta, se deberá seleccionar el registrarse (figura J.5) para crearse una, pues es necesaria para poner acceder a las características del modo en red. Al registrarse se le pedirá un nombre de usuario, una contraseña y un correo electrónico, y al confirmar el registro se le pedirá que mire su correo para introducir el código de verificación. En caso de que no se vea el correo en la bandeja de entrada, mirar en correo no deseado.

Con la cuenta creada, ya se podrá iniciar sesión (figura J.6) y acceder a todas las características. Decir que si el usuario olvidó su contraseña, para poder recuperar la cuenta simplemente ha de introducir su nombre en el inicio de sesión, y presionar el

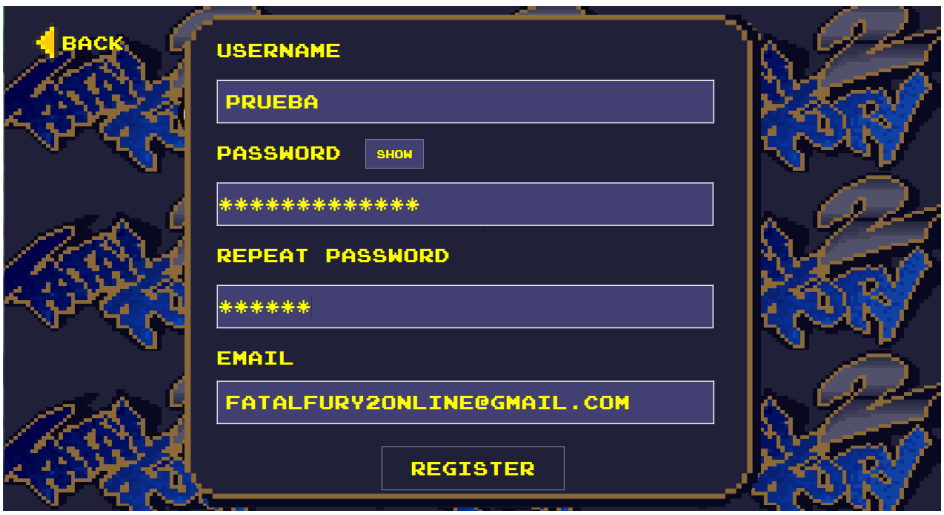


Figura J.5: Interfaz - Registro.

botón de recuperar cuenta. Tras presionar el botón se le mandará a su correo una nueva contraseña con la que podrá acceder, y posteriormente cambiar.

Figura J.6: Interfaz - Inicio de sesión.

Dentro del modo en red con la sesión iniciada, se le mostrará al usuario el menú principal (figura J.7) desde el que podrá seleccionar los modos de juego (partida normal, clasificatoria o contra una de las dos IAs), ver su perfil y la clasificación. También a su derecha tendrá la lista de amigos, con la posibilidad de añadir nuevos amigos con botón dedicado para ello.

Para interactuar con los amigos que tenga añadidos, simplemente deberá pulsar encima del nombre del usuario y le saldrá un menú desplegable (figura J.8) con todas las opciones (retar a una partida, mandar un mensaje, ver su perfil y borrarlo de amigos). Para mandar un mensaje a cualquier amigo, también puede pulsar dos veces encima del nombre del amigo, y se abrirá directamente el chat.

Figura J.7: Interfaz - Menú principal.

Figura J.8: Interfaz - Menú desplegable al pulsar sobre un amigo.

Al seleccionar partida normal o partida clasificatoria, o retar a un amigo, se pasará a una pantalla de espera (figura J.9), hasta que o el usuario cancele la búsqueda de partida o invitación, o hasta que se le encuentre un rival, o el amigo acepte la invitación.

Tras encontrar partida o que acepte un amigo invitado, se pasará a la selección de personaje y mapa (figura J.10). Si el servidor notificó al cliente del jugador que era anfitrión, este podrá seleccionar tanto personaje como mapa, mientras que sino se lo notificó, solo podrá seleccionar personaje. En la selección se puede elegir entre un total de 3 personajes, cada uno con sus peculiaridades únicas, y 3 mapas (que sólo afecta visualmente, no hay interacción con el escenario).

Tras la selección se pasará a la pelea (figura J.11) en sí, en donde el usuario deberá controlar a su personaje con el teclado según lo haya configurado (por defecto teclas: Q,W,E y R, y las flechas). En cualquier momento el usuario puede presionar el escape

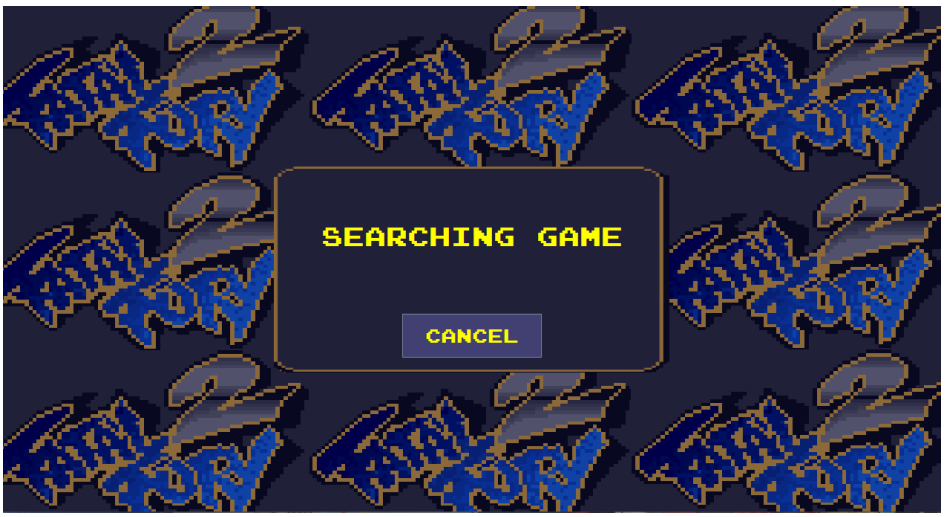


Figura J.9: Interfaz - Pantalla de espera de partida.

Figura J.10: Interfaz - Selección de personaje y mapa.

para abrir el menú de partida y abandonar la partida o salir del juego. Para ganar la partida deberá ganar al menos 2 rondas del máximo de 4 que se pueden jugar (la cuarta solo se juega en caso de que tras 3 rondas haya un empate).

Tras la partida se le mostrará una pantalla que indicará si ha ganado o perdido (figura J.12), y en caso de ser partida clasificatoria cuantos puntos ha ganado o perdido. Para volver al menú principal solo ha de pulsar el botón de confirmar.

En el menú principal, si selecciona la clasificación (figura J.13) verá una tabla con los jugadores con mayor cantidad de puntos, ordenados de mayor a menor cantidad de puntos.

Al seleccionar perfil en el menú principal, o al seleccionar ver el perfil (figura J.14) de un amigo, se mostrará una pantalla con la información básica del jugador. En caso de ser el perfil propio, aparecerá un botón para poder cambiar la contraseña, para lo

Figura J.11: Interfaz - Partida en curso.

Figura J.12: Interfaz - Pantalla de fin de la pelea.

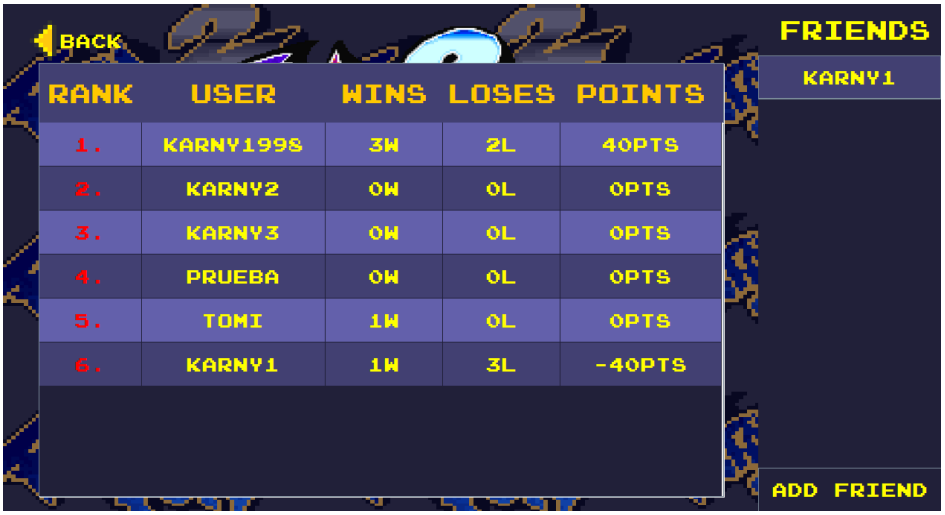


Figura J.13: Interfaz - Clasificación.

cual únicamente se requiere la contraseña actual y la nueva contraseña.

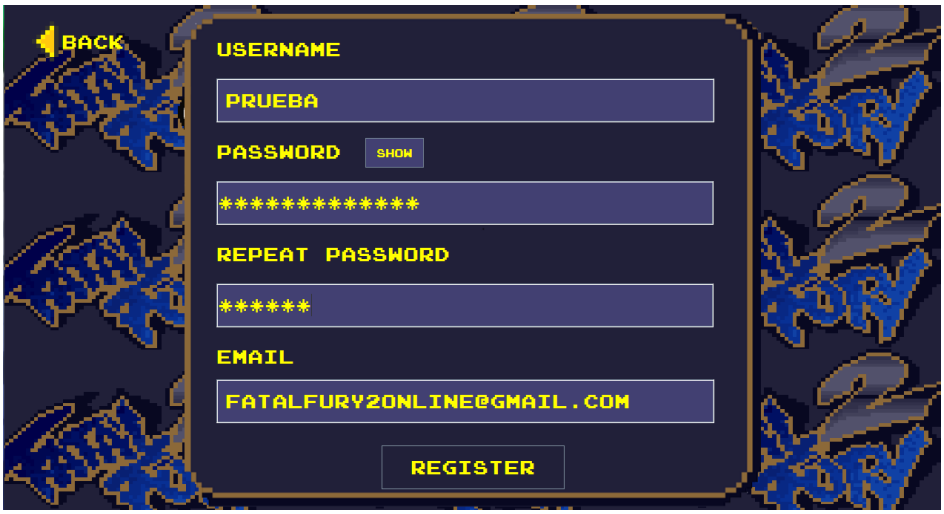
Figura J.14: Interfaz - Perfil.

Finalmente, para salir del juego, podrá presionar la X de la ventana en cualquier momento, o seleccionar salir en el menú principal. Para cerrar la sesión, simplemente en el menú principal pulsar volver atrás, mostrándole de nuevo la pantalla de inicio.

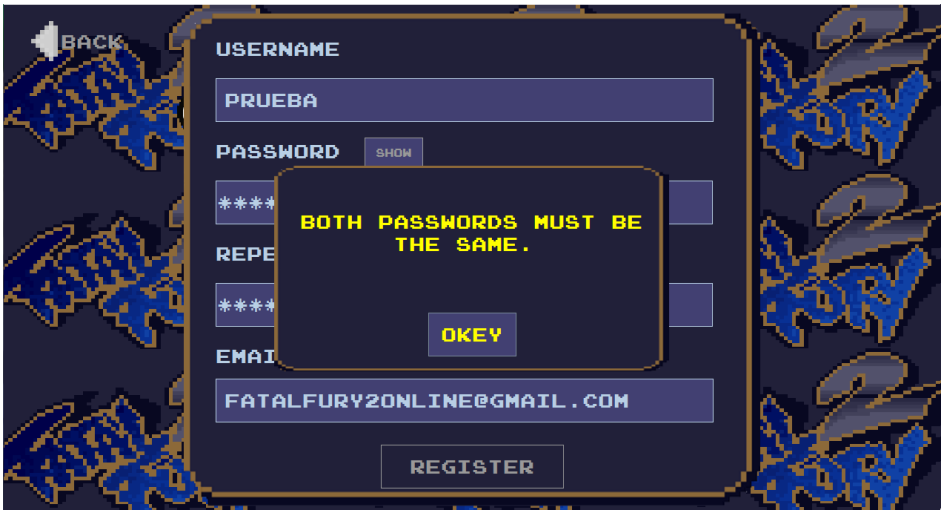
J.4. Navegación por la interfaz

(a) Selección modo local o modo en red.

(b) Pantalla de inicio al seleccionar modo en red.



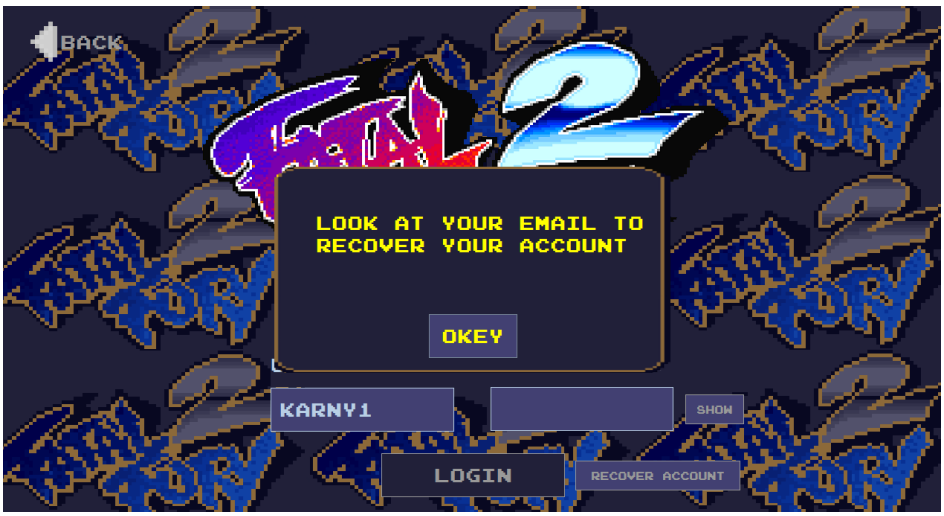
(c) Formulario de registro al seleccionar registrarse.



(a) Mensaje de error al confirmar el registro.

(b) Soliticud del código de verificación tras confirmar registro exitosamente.

(c) Formulario de inicio de sesión al seleccionar iniciar sesión en la patalla inicial.

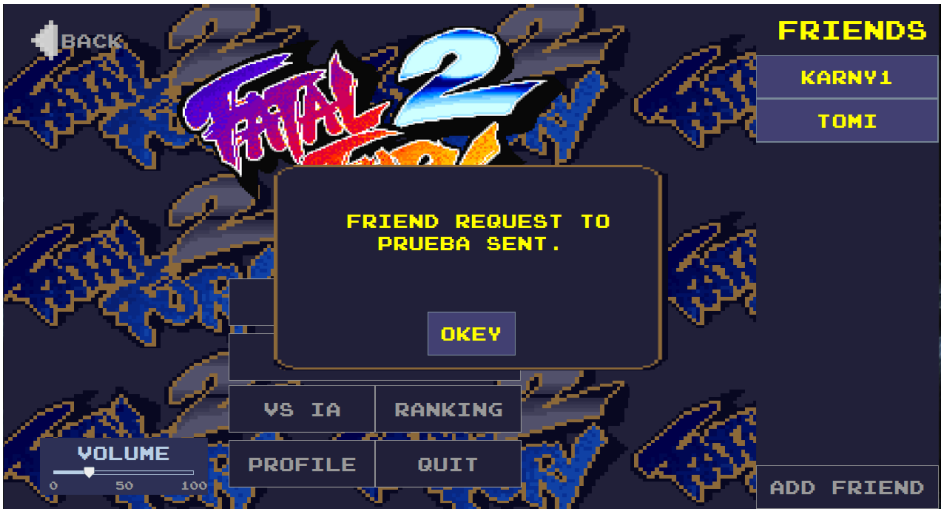


(a) Notificación al seleccionar recuperar cuenta (con un nombre de usuario introducido) en el inicio de sesión.

(b) Menú principal al iniciar sesión exitosamente.

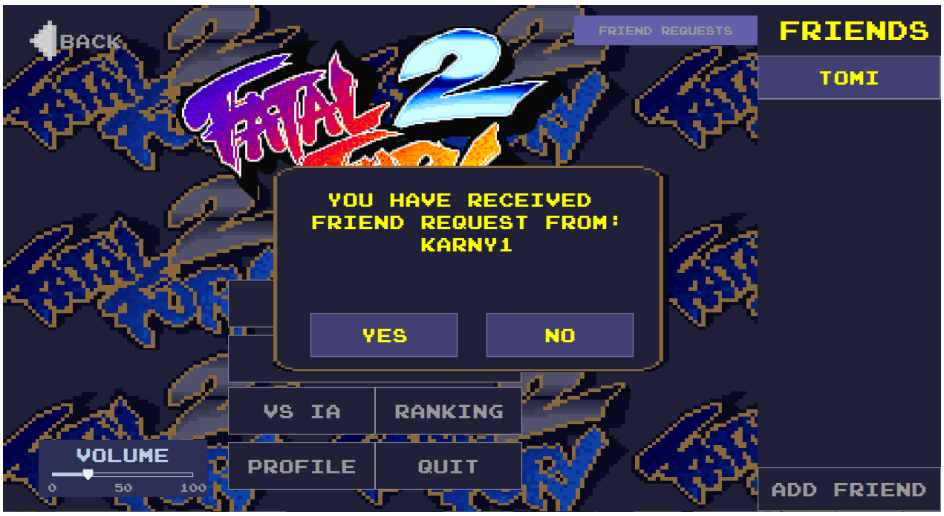
(c) Solicitud de nombre de usuario al seleccionar añadir amigo.

(a) Error al confirmar añadir amigo.



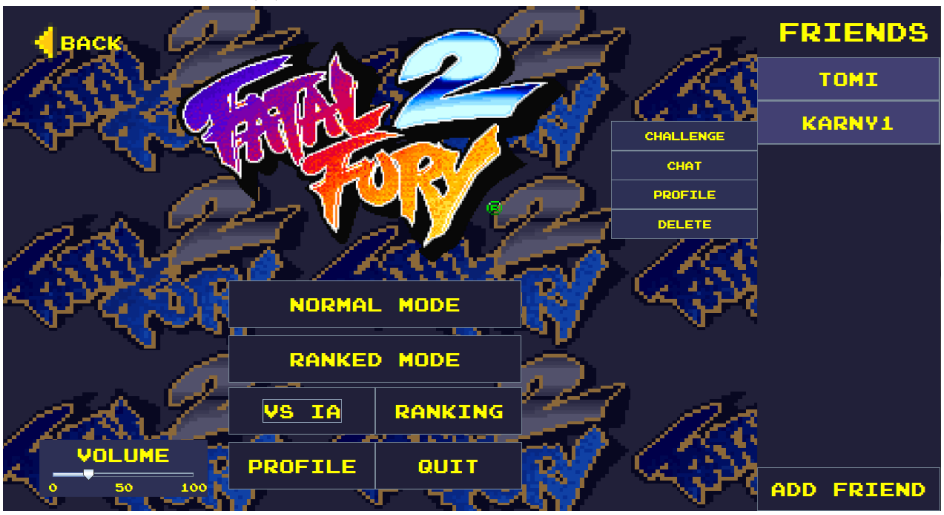
(b) Notificación de envío de solicitud de amistad correcto.

(c) Notificación de solicitud de amistad recibida (arriba a la derecha).

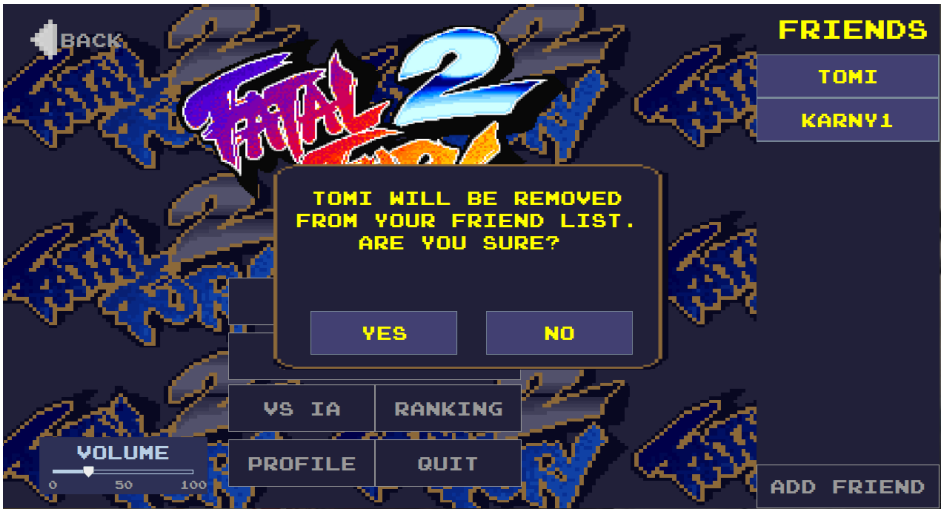


(a) Respondiendo una solicitud de amistad al seleccionar la notificación de solicitud.

(b) Solicitud de amistad aceptada.



(c) Menú emergente al seleccionar un usuario de la lista de amigos.



(a) Eliminando un amigo al seleccionar borrar del menú emergente de amigos.

(b) Amigo borrado al confirmar la eliminación de este.

(c) Chat con un amigo al seleccionar chat del menú emergente de amigos.

(a) Notificación de que el usuario no está conectado al seleccionar retar del menú emergente de amigos

(b) Perfil de un amigo al seleccionar perfil del menú emergente de amigos.

(c) Buscando partida normal al seleccionar modo normal en el menú principal.

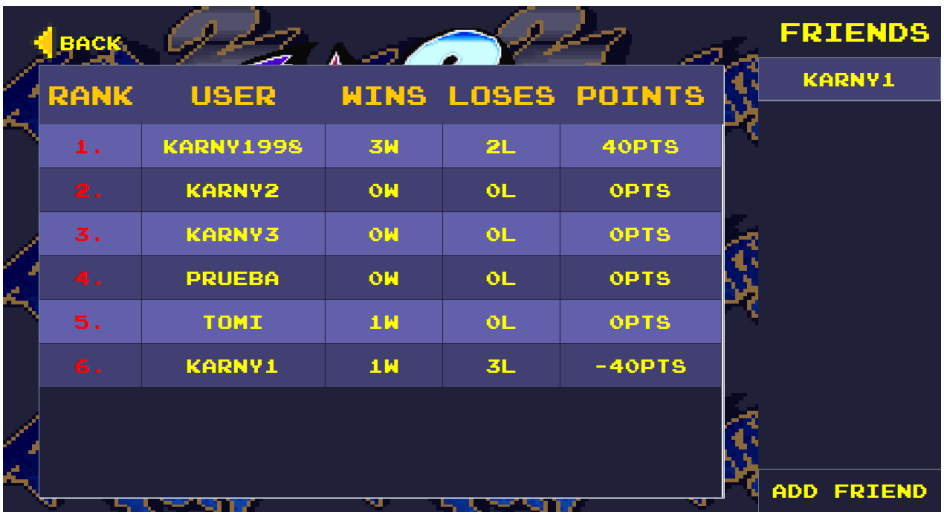
(a) Buscando partida clasificatoria al seleccionar modo clasificatorio en el menú principal.

(b) Selección de IA al seleccionar contra IA en el menú principal

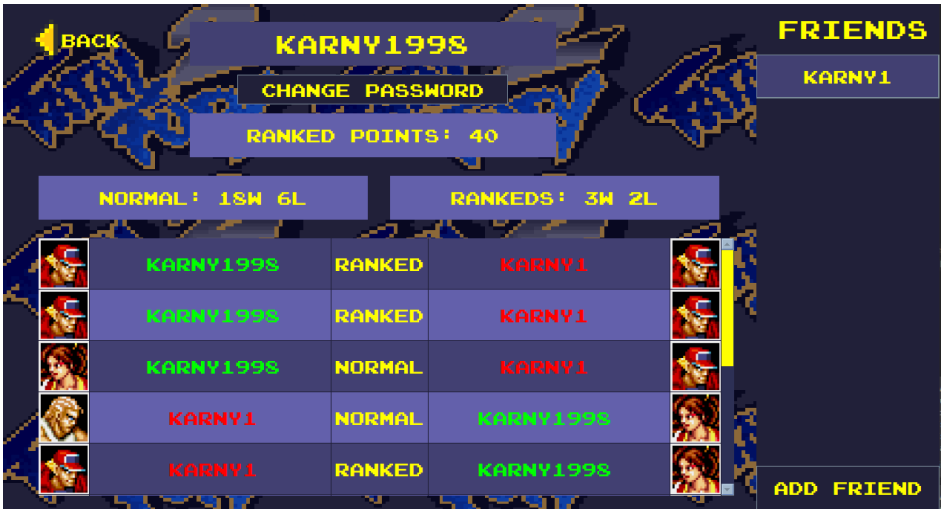
(c) Selección de personaje y mapa al seleccionar una IA, o iniciar una partida como anfitrión contra otro jugador (cambiaría PERSONAL IA por el nombre del jugador).

(a) Partida en red o contra IA.

(b) Final de partida contra IA o con otro jugador (saldría su nombre en vez de IA). Si fuera contra otro jugador y clasificatoria, en el centro saldrían los puntos ganados o perdidos.



(c) Tabla de clasificación al selecciona clasificación en el menú principal.

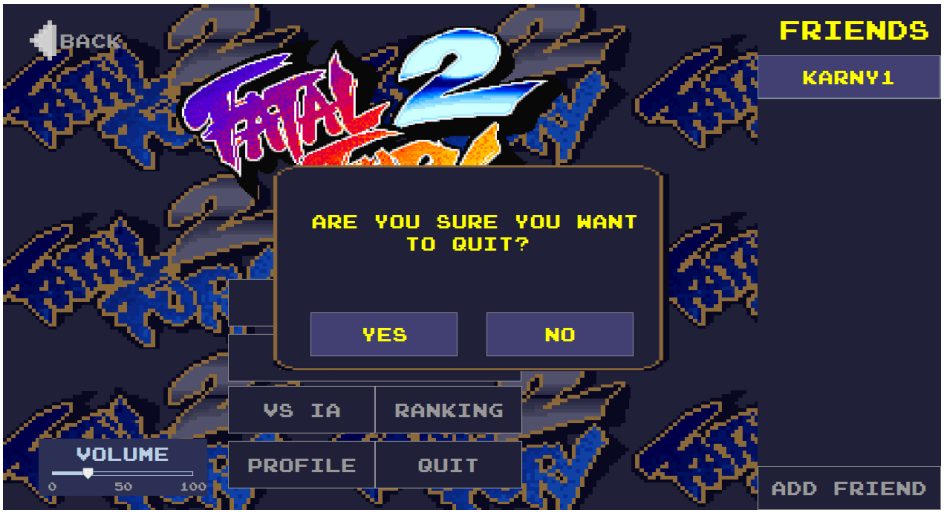


(a) Perfil propio al seleccionar perfil en el menú principal.

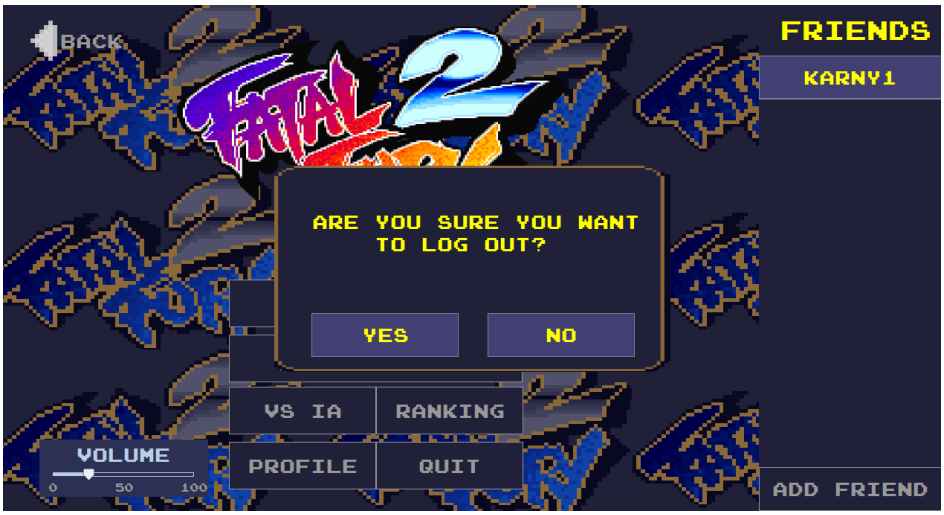
(b) Formulario de cambio de contraseña al seleccionar cambiar contraseña en el perfil propio.

(c) Formulario de registro al seleccionar registrarse .

(a) Selección modo local o modo en red.



(b) Pantalla de inicio al seleccionar modo en red.



(c) Formulario de registro al seleccionar registrarse .

Anexo K

Enlaces de interés

K.1. Código fuente

El código fuente está alojado en un repositorio de GitHub puesto en público con la licencia *GNU General Public License v3.0*. Decir que pese a que hay varios proyectos, los importantes son *Fatal-Fury-2-JPA-Server* (fuentes del servidor), *FatalFury2* (fuentes del juego) e *IA_Analyzer* (fuentes del generador de gráficas). El enlace al repositorio es el siguiente:

`https://github.com/karny1998/TFG-Fatal-Fury-2-Online`

K.2. Ejecutables

Se han generado dos ejecutables comprimidos, uno del juego (Fatal_Fury2_Game.zip) y otro del servidor (Fatal_Fury_2_Server.zip), en caso de que se desease probar. Si se quisiese correr su propio servidor, se debería modificar el *persistence.xml* de la carpeta interna META-INF del ejecutable. El ejecutable del juego incluido esta listo para conectar a un servidor lanzado en una máquina virtual de Google Cloud. El enlace a los ejecutables es el siguiente:

`https://mega.nz/folder/vN5hkSaL#hx2f6xebGun68goM5qXYgw`

K.3. Video de demostración

Para mostrar las funcionalidades del prototipo se grabó un vídeo mostrando las características principales del juego. Decir que durante la grabación se descubrió un pequeño error visual en el cambio de contraseña que se corrigió en el acto, de ahí que en el vídeo haya dos cortes en ese momento. El enlace al vídeo es el siguiente:

`https://www.youtube.com/watch?v=oP-fYyj1f4I`