



UNIVERSIDAD
DE GRANADA

MÁSTER UNIVERSITARIO
EN CIENCIA DE DATOS E INGENIERÍA DE COMPUTADORES

Trabajo de Fin de Máster

**Desarrollo y Optimización de un Agente Inteligente
para Scripts of Tribute mediante Algoritmos
Evolutivos**

Autor

Francisco David Castejón Soto

Director

Dr. Pablo García Sánchez



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, 1 Julio de 2025

Prefacio

Para la realización de este documento, se ha utilizado la plantilla \LaTeX [17] específica para la ETSIIT.

Agradecimientos

A mi familia por el apoyo que me han brindado incondicionalmente durante estos años fuera de casa. Y a mi pareja, por mostrarme más caminos de los que nunca hubiera pensado que existieran para mí.

Índice general

Índice de figuras	xii
Índice de tablas	xiii
I Marco teórico	1
1. Introducción	3
2. Objetivos	5
2.1. Objetivos principales	5
2.2. Objetivos específicos	6
3. Antecedentes y estado del arte	7
3.1. IA en videojuegos comerciales	7
3.1.1. Funciones hash	8
3.1.2. Máquinas de estados finitos	9
3.1.3. Árboles de comportamiento	9
3.1.4. Planificación de acciones orientada a objetivos y la IA utilitaria	9
3.1.5. Aprendizaje por refuerzo y redes neuronales	10
3.2. El problema de la IA en los videojuegos comerciales	11
3.3. Metaheurísticas en la investigación científica	12
3.4. Algoritmos evolutivos en juegos de estrategia	13
3.4.1. Optimización de agentes de Hearthstone mediante un algoritmo evolutivo	13
3.4.2. La hegemonía de Monte Carlo	14
4. Plataforma de trabajo	17
4.1. El videojuego: Tales of Tribute	17
4.2. El entorno de simulación: Scripts of Tribute	19

ÍNDICE GENERAL

4.2.1. Interfaz gráfica	19
4.2.2. Arquitectura del entorno	21
4.3. La competición IEEE-CoG	22
5. Planificación del proyecto	25
5.1. Recursos utilizados	25
5.1.1. Recursos software	25
5.1.2. Recursos hardware	25
5.2. Ciclo de desarrollo del software	26
5.3. Distribución temporal y cronograma	26
II Metodología	29
6. Diseño del agente autónomo	31
6.1. Arquitectura de la toma de decisiones	31
6.2. Definición de los pesos	33
6.3. Lógicas específicas y heurísticas	34
7. Sistema de entrenamiento evolutivo	37
7.1. Arquitectura general del entrenador	37
7.2. Comunicación entrenador-simulador	39
7.3. El algoritmo evolutivo con Inspyred	39
7.4. El salón de la fama	41
7.5. Arquitectura del orden, la evaluación y la paralelización	43
7.5.1. Modo fijo: evaluación contra oponentes estáticos	44
7.5.2. Modo coevolución: competición interna	44
7.5.3. Modo híbrido: combinando estrategias	45
7.5.4. Análisis comparativo del coste computacional	45
III Experimentación y conclusiones	47
8. Diseño experimental	49
8.1. Configuración de los experimentos	49
8.1.1. Análisis de configuraciones híbridas	49
8.1.2. Análisis de los modos de entrenamiento y el salón de la fama	51
8.2. Métricas de evaluación	52
8.2.1. Bots de referencia	53
9. Resultados y discusión	55
9.1. Análisis de las configuraciones híbridas	55
9.2. Análisis de los modos de entrenamiento	55
9.3. Análisis del salón de la fama	55

ÍNDICE GENERAL

10. Conclusiones	57
10.1. Objetivos alcanzados	57
10.2. Líneas de trabajo futuro	57
Bibliografía	59

Índice de figuras

3.1. Taxonomía de la IA en videojuegos [33].	8
4.1. Tablero de juego de Tales of Tribute, con sus componentes principales numerados. [26]	18
4.2. Tablero de juego de Scripts of Tribute. [7]	20
4.3. Vista de elección de cartas en Scripts of Tribute. [7]	20
4.4. Vista de final de partida con historial de movimientos en Scripts of Tribute. [7]	21
4.5. Resultados de la competición IEEE-CoG 2024. [6]	23
5.1. Distribución temporal de las tareas del proyecto	27

Índice de tablas

6.1. Descripción de los pesos del genoma del agente evolutivo.	34
8.1. Configuraciones de entrenamiento en modo híbrido evaluadas. . .	50
8.2. Configuraciones para la comparativa de modos de entrenamiento y salón de la fama.	52

Parte I

Marco teórico

Introducción

La inteligencia artificial (IA) en videojuegos existe en una intersección entre la ciencia computacional y el arte del entretenimiento, dando vida a mundos virtuales y creando oponentes dignos de nuestras mejores estrategias. En este contexto, la IA se refiere únicamente a un conjunto de algoritmos y técnicas diseñadas para cumplir una función muy específica dentro del videojuego. Ejemplos de esto son la IA que “conduce” los coches en un videojuego de carreras, los Pokémon contra los que el jugador se enfrenta en una batalla o el simple movimiento de una línea de píxeles (a modo de pala contrincante) en el videojuego Pong. Esta acepción específica contrasta con la idea más extendida de IA, que posee un conjunto de connotaciones más amplias y generalistas como “la máquina que aprende” o “el estudio y construcción de sistemas que hacen «lo correcto», en función de su objetivo” [28]. Independientemente de la definición que se utilice, un factor clave en el desarrollo de un videojuego es la creación de una IA que resulte entretenida para el jugador. De la misma forma que en un libro o una película es necesaria una trama que plantea un desafío, como la lucha contra el Imperio Galáctico en *Star Wars*; o un compañero de aventuras que ayude al protagonista, como Sam en *El Señor de los Anillos*. En un videojuego es necesaria una IA que controle esos elementos que interactúan con el jugador, ya sean enemigos, aliados o incluso el propio entorno. Desde los propios inicios de la industria del videojuego, donde se construían máquinas específicas para ejecutar un juego concreto, como es el caso de Nim en 1948 [27], hasta los videojuegos modernos, donde se utilizan técnicas de aprendizaje automático complejas, como el aprendizaje por refuerzo [13], la IA ha sido un componente esencial para crear experiencias de juego divertidas y desafiantes.

La intención de este trabajo es la de crear una IA contra la que jugar en un videojuego de estrategia. En concreto, la que controla a un agente autónomo (o simplemente “bot”) en un simulador del videojuego de cartas *Tales of Tribute*¹.

¹Es importante la distinción entre “juego” y “videojuego” en este caso, ya que las cartas no son físicas y tangibles, sino que únicamente existen en el universo digital del videojuego.

CAPÍTULO 1. INTRODUCCIÓN

Para guiar las decisiones del bot, un algoritmo evolutivo se encarga de ajustar los pesos que se utilizan para calcular la puntuación de cada jugada posible, eligiendo vorazmente la jugada que maximiza dicha puntuación. De esta forma se ha conseguido crear un contrincante que juega de forma competitiva tanto contra el jugador humano como contra otros bots manejados por algoritmos de inteligencia artificial totalmente diferentes.

Capítulo 2

Objetivos

2.1. Objetivos principales

Los objetivos principales de este Trabajo de Fin de Máster representan dos metas entrelazadas. Por un lado, ahondar en la investigación y entendimiento de un área tan extensa como las metaheurísticas, y en concreto los algoritmos evolutivos. La optimización de soluciones a problemas complejos mediante técnicas que tratan de incluir conocimiento específico, unido a la generación iterativa de otras soluciones parciales, es un campo de estudio con un gran número de aplicaciones prácticas. Aunque el teorema de “No Free Lunch” nos advierte de que no existe una única técnica que sea la mejor para todos los problemas y desde todas las perspectivas [39], la experiencia empírica ha demostrado que los algoritmos evolutivos son una herramienta que se puede aplicar a la gran mayoría de problemas de optimización [35].

Es precisamente esa versatilidad la que ha propiciado su uso durante el desarrollo del segundo objetivo principal de este proyecto: la creación de un bot para un videojuego de cartas. Este tipo de videojuegos cuentan con un inmenso número de variables, pues no solo se deben de tener en cuenta las cartas existentes en las manos de cada jugador y en la pila de cartas, sino también las mecánicas² intrínsecas del juego, como la vida, los recursos de compra, la posibilidad de recuperar cartas usadas, o el uso de otro tipo de habilidades especiales. Todo esto hace que la creación de un bot que juegue a alto nivel a un videojuego de este estilo sea un reto interesante, y que el uso de técnicas de optimización evolutiva sea una herramienta adecuada para la optimización de su rendimiento.

²Gran parte de lo que define un juego es sus mecánicas, sus reglas, lo que los jugadores deben llevar a cabo para ganar. La labor del creador de videojuegos es conseguir un conjunto de reglas que estén equilibradas y permitan disfrutar a los jugadores [38].

2.2. Objetivos específicos

- OG1: Analizar en profundidad las mecánicas del juego “Tales of Tribute”, el entorno de desarrollo “Scripts of Tribute” y adquirir las competencias necesarias en el lenguaje de programación C#.
- OG2: Diseñar e implementar un agente inteligente en C# para “Scripts of Tribute”, cuya toma de decisiones se base en una evaluación heurística del estado del juego mediante una función de fitness ponderada, incorporando conocimiento específico del dominio.
- OG3: Desarrollar un programa de optimización en Python para ajustar los pesos de la función de fitness del agente, implementando y comparando dos estrategias principales: algoritmos coevolutivos y entrenamiento supervisado contra agentes de referencia de “Scripts of Tribute”.
- OG4: Desarrollar un conjunto de herramientas para la visualización y análisis de los datos generados durante el entrenamiento.
- OG5: Evaluar cuantitativamente el rendimiento del agente entrenado mediante las diferentes estrategias, utilizando métricas relevantes.
- OG6: Analizar comparativamente la efectividad y eficiencia de las estrategias de optimización implementadas, discutiendo sus ventajas y desventajas en el contexto específico de “Scripts of Tribute”.
- OG7: Investigar y contextualizar el enfoque desarrollado frente a otras técnicas predominantes en competiciones similares de IA en juegos, analizando las razones de su éxito.

Capítulo 3

Antecedentes y estado del arte

En este capítulo se encuentra una revisión de los antecedentes y el estado del arte de la inteligencia artificial en la industria del videojuego, así como de heurísticas empleadas en la investigación científica relacionada. Durante todo el capítulo se hace mención a la organización propuesta por Tommy Thompson en su artículo "AI 101: How AI is Actually Used in the Video Games Industry" [33]³. Para guiar el proceso, se ha incluido la figura 3.1, que él desarrolló para su artículo, en la que se muestra un resumen de los usos que se le está dando a la IA en la industria del videojuego actualmente, además de dos clasificaciones útiles para entender mejor el contexto.

3.1. IA en videojuegos comerciales

Como ya se mencionaba en la introducción, la IA puede tener varias acepciones en función del contexto en el que se utilice. En el caso de los videojuegos, se pueden encontrar dos vertientes principales: la IA simbólica y el aprendizaje automático. La IA simbólica, la más antigua de las dos, utiliza sistemas de razonamiento automático para modelar un problema y encontrar una solución dentro del espacio de búsqueda. Por otro lado, la IA basada en el aprendizaje automático utiliza métodos estadísticos que aprenden de datos existentes, extendiéndose hasta el aprendizaje profundo y, de forma reciente, a la IA generativa. Este último tipo de IA se está extendiendo rápidamente en la industria tecnológica, pero aún no se ha implementado de forma generalizada en el mundo del videojuego fuera de pequeñas demos técnicas o modificaciones a videojuegos preexistentes

³Aunque no es un artículo científico, Tommy cuenta con más de 10 años de experiencia en la industria del videojuego, trabaja como consultor de IA para videojuegos y ha impulsado la creación del mayor congreso de IA en videojuegos de Europa. En la "AI and Games Conference" se reúnen profesionales de la industria y académicos para discutir sobre el estado del arte de la IA en videojuegos. Por lo tanto, su artículo se considera una buena referencia para entender cómo se está utilizando la IA actualmente en la industria.

CAPÍTULO 3. ANTECEDENTES Y ESTADO DEL ARTE

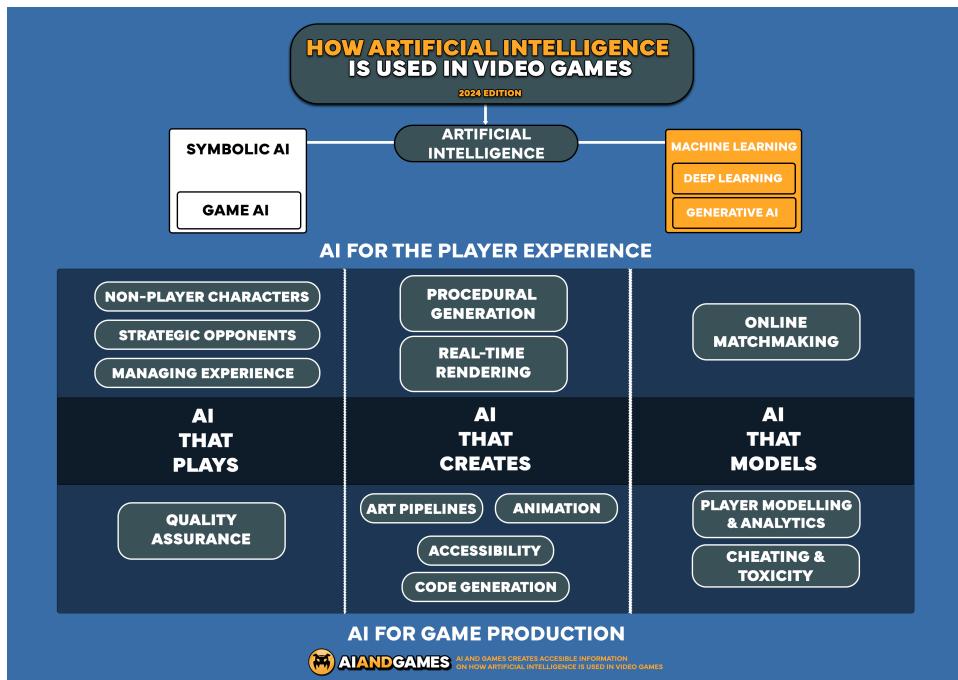


Figura 3.1: Taxonomía de la IA en videojuegos [33].

por parte de la comunidad [33], por lo que aun no se considera relevante para este proyecto.

En la figura 3.1 se aprecia una segunda clasificación, la cual divide a la IA según su función. En este caso, se pueden distinguir tres tipos: la “IA que juega” al videojuego de alguna forma, la “IA que crea” partes del videojuego y la “IA que modela” una propiedad o fenómeno del ecosistema de este. La primera categoría es la más relevante para este proyecto, pues se centra en la IA que controla a los personajes/enemigos o, en su forma más avanzada, un jugador virtual completo. Por esa razón, en este capítulo la mayoría de los ejemplos se centrarán en “IA que juega”, aunque también se mencionarán algunos ejemplos de las otras categorías para dar una visión más completa de tema a tratar.

3.1.1. Funciones hash

Una de las técnicas más sencillas y antiguas para controlar partes específicas de un videojuego son las funciones hash. Este tipo de funciones simplemente reciben un conjunto de parámetros de entrada y devuelven un valor o acción a realizar. Dada su simplicidad, generan comportamientos predecibles, pero requieren de muy pocos recursos para su procesamiento y son fáciles de implementar. Un ejemplo clásico de IA basada en funciones hash está en el videojuego *Space Invaders* [37] donde los invasores están controlados por este tipo de funciones.

3.1. IA EN VIDEOJUEGOS COMERCIALES

3.1.2. Máquinas de estados finitos

Las máquinas de estados finitos son un modelo de computación conceptual que describe un sistema que solo puede encontrarse en un estado a la vez, y que puede cambiar a uno de sus otros estados como respuesta a ciertos eventos. Este mecanismo se ha utilizado y sigue utilizándose para un gran número de aplicaciones dentro de los videojuegos. Por ejemplo, en el juego *Pacman*, la IA de los fantasmas se rige por el estado en el que se encuentran, como “cazando” o “enjaulados” [23]. Otro uso más centrado en la “IA que crea”, está en la gestión de las animaciones de los personajes, donde cada estado corresponde a una posición o movimiento específico de la malla⁴, que al mezclarse mediante interpolaciones generan animaciones nuevas. Al igual que las funciones hash, las máquinas de estados finitos requieren de pocos recursos para su procesamiento, pero su naturaleza determinista y la posibilidad de entrar en bucles sin salida si no están bien diseñadas, limitan su uso en los escenarios más exigentes.

3.1.3. Árboles de comportamiento

Como evolución a las máquinas de estados finitos, surgieron los árboles de comportamiento, permitiendo un mayor grado de complejidad y flexibilidad en la toma de decisiones de los personajes. Los árboles de comportamiento son grafos dirigidos acíclicos, con nodos hoja que representan acciones. Además, cuentan con varios tipos de control de flujo que determinan el orden de ejecución de las acciones [9]. Una de las características más útiles de los árboles de comportamiento son su modularidad, lo que permite reutilizar y combinar comportamientos específicos fácilmente. Un ejemplo de su correcto uso se expuso en la charla de la “Game Developers Conference” (GDC) de 2005, sobre la IA de *Halo 2*, donde se explica que la capacidad de reutilización de árboles y al ser más sencillos de entender para los diseñadores del juego, hizo que los programadores optaran por la implementación de árboles de comportamiento en lugar de las máquinas de estados finitos [19].

3.1.4. Planificación de acciones orientada a objetivos y la IA utilitaria

En contraste con los árboles de comportamiento, que generan una jerarquía de caminos, la planificación de acciones orientada a objetivos (GOAP, por sus siglas en inglés) es más parecida a la planificación de rutas, donde se busca el camino más óptimo para alcanzar un objetivo en función del estado del mundo. Cada uno de los pasos que llevan al objetivo tiene una serie de precondiciones que deben cumplirse, a los cuales se les puede asignar un coste. Es tal su similitud con la planificación de rutas, que en este tipo de IAs se suelen utilizar internamente

⁴Una “mesh”, o malla en español, es una estructura tridimensional formada por una colección de vértices, aristas y caras que definen la forma de un objeto 3D dentro de un videojuego [8].

CAPÍTULO 3. ANTECEDENTES Y ESTADO DEL ARTE

algoritmos de búsqueda como A*⁵ para encontrar el camino más óptimo. Una de las mejores implementaciones de este tipo de IA se encuentra en el título de 2005 *F.E.A.R.*, donde los enemigos y aliados utilizan GOAP para “planificar” su comportamiento una manera mucho más proactiva que sus predecesores del género [20].

El caso de la IA utilitaria tiene un enfoque muy parecido, pues trata de ayudar a encontrar la mejor sucesión de acciones para alcanzar un objetivo, o incluso decidir cuál es el más adecuado en cada momento. Por ejemplo, un personaje que utilice IA utilitaria puede utilizar factores como la distancia a un objeto, el número de recursos existentes o el riesgo potencial de una acción para obtener una puntuación final que le permita decidir qué acción tomar. Esta forma de tomar decisiones contrasta con técnicas como los árboles de comportamiento, donde las acciones se ejecutan en un orden predefinido a menos que se produzca un cambio en el estado del mundo [32].

3.1.5. Aprendizaje por refuerzo y redes neuronales

El aprendizaje por refuerzo (RL, por sus siglas en inglés) es un subcampo del aprendizaje automático que trata de mejorar el comportamiento de un agente a través del feedback que recibe de su entorno. Utilizando un sistema de recompensas y penalizaciones, el agente aprende a tomar decisiones que maximicen su recompensa total a lo largo del tiempo. Aunque el RL empezó con técnicas tabulares simples basadas en las cadenas de Markov, los algoritmos recientes incorporan redes neuronales en diferentes partes del proceso [15] de entrenamiento. Quizás el ejemplo que más se suele ver en la literatura de videojuegos es el uso de RL para controlar vehículos autónomos. Una implementación de esta tecnología en un videojuego comercial está en el reciente título *Star Wars: Outlaws*, donde las motos controladas por IA utilizan un sistema de RL para moverse por el mapa o perseguir al jugador [13]. Sus desarrolladores argumentan que les permitió tener un sistema de control que se adapta a las diferentes rutas del juego sin importar los cambios en el mapa que haya entre versiones de desarrollo.

Pero el uso de redes neuronales no se limita a la toma de decisiones. Su capacidad para realizar inferencias rápidas a partir de datos complejos las hace ideales para otras tareas, como la deformación de mallas. El objetivo en este caso es evitar el fenómeno conocido como el “valle inquietante” (*Uncanny Valley*), que se produce cuando un personaje es muy realista, pero no perfecto, generando una sensación de extrañeza en el espectador. Un personaje que sonríe, por ejemplo, pero cuyos músculos faciales alrededor de los ojos permanecen inmóviles, es un caso clásico de este efecto. Para intentar superar este problema, Epic Games optó por crear un sistema para Unreal Engine que utiliza modelos de redes neuronales que simulan el comportamiento de la musculatura y los tejidos blandos bajo la piel

⁵El algoritmo A* también se utiliza para la búsqueda de caminos tradicional dentro de los videojuegos, es decir, para saber la ruta que debe tomar un objeto por el mapa para llegar a un destino.

3.2. EL PROBLEMA DE LA IA EN LOS VIDEOJUEGOS COMERCIALES

al extenderse y contraerse [10]. Un ejemplo de uso de esta reciente tecnología se puede encontrar en la demo técnica que CD Projekt Red mostró en el Unreal Fest de 2025 [4]. En ella, se mostraba como se modelaba internamente la musculatura de un caballo para que sus animaciones de movimiento fueran mucho más vívidas.

3.2. El problema de la IA en los videojuegos comerciales

En la anterior sección se han hablado exclusivamente de técnicas de inteligencia artificial que se han empleado en videojuegos que han salido al mercado para el público general. Pero, salvo en funciones específicas, no se ha mencionando el uso de técnicas de inteligencia artificial “avanzadas” para el control de los personajes o contrincantes virtuales. Sin embargo, este tipo de IAs sí existen. Se han desarrollado con éxito bots capaces de jugar a alto nivel videojuegos como *StarCraft II* [36], *Dota 2* [25] o *Rocket League* [24], entre otros. Todos estos artículos científicos, aclamados por la comunidad debido a su complejidad, tienen algo en común: se han creado una vez el juego ya había sido lanzado y era estable.

A menudo, los videojuegos con componentes multijugador de éxito, acaban siendo actualizados y mejorados durante años. Este clima de constante cambio, el cual se ve acentuado aun más durante la etapa inicial de su desarrollo, hace que el uso de técnicas como el aprendizaje por refuerzo, los algoritmos evolutivos o las redes neuronales sean especialmente difíciles de implementar. El ejemplo de *Star Wars: Outlaws* es un caso muy reciente, con tecnología especialmente desarrollada para su motor gráfico y con un equipo detrás de cientos de personas con capacidad de delegar personal en los enfoques más vanguardistas del sector. Y aun así, sólo usaron el aprendizaje por refuerzo para un aspecto muy específico de su videojuego.

En los casos en los que se busca conseguir un jugador virtual, se podría incluso considerar que es un problema del tipo “el huevo o la gallina”, pues se necesita que el juego esté prácticamente listo para entrenar al bot, lo que permite crear un bot específico para esa versión, pero si el juego cambia, incluso con pequeños ajustes de balanceo de poder, entonces el rendimiento del bot puede verse afectado. Por eso todos esos artículos científicos se centran en juegos ya terminados o en una versión específica del videojuego. Además, en aquellos casos en los se ha conseguido crear un bot que maneje todos los aspectos del control, como lo fue para AlphaStar [36] o OpenAI Five [25], se han necesitado una gran cantidad de datos que solo se podrían haber obtenido gracias a la comunidad de jugadores de sus respectivos videojuegos y no antes de su lanzamiento. Sin embargo, la extrema dificultad de adaptación al cambio no significa que no se puedan crear “IAs que juegan” mediante aprendizaje automático para videojuegos aun en desarrollo. Utilizando el enfoque adecuado sí se pueden crear bots que se adapten a los cambios del videojuego, pero no se pueden utilizar para controlar

CAPÍTULO 3. ANTECEDENTES Y ESTADO DEL ARTE

todos los aspectos del mismo. Es decir, se deben utilizar para resolver problemas específicos o bien para el caso de videojuegos que no requieran de un escenario tan complejo como los mencionados anteriormente [3]. Un ejemplo de esto sería *Sophy*, la IA de conducción de *Gran Turismo 7* [40], que fue entrenada durante el desarrollo del videojuego para manejar los vehículos contrincantes del videojuego de carreras.

A día de hoy es posible que esa sea la forma más adecuada de utilizar este tipo de técnicas: o bien para pequeñas características del juego, o bien para entornos más reducidos. Uno de esos entornos reducidos son los juegos de cartas, en ese tipo de videojuegos, aunque el número de variables es alto, la cantidad de acciones posibles en cada turno es limitada, lo que permite entrenar a los bots de forma más eficiente. En las siguientes secciones se revisarán algunas de las investigaciones científicas que intentan aplicar este mismo enfoque a sus implementaciones.

3.3. Metaheurísticas en la investigación científica

Como se mencionó al principio de este capítulo, las “IAs que crean” son una forma de llamar a aquellas aplicaciones de la inteligencia artificial que se centran en la generación procedural de contenido, la cual trata de unir y entremezclar diferentes bloques de construcción para crear nuevos elementos dentro del videojuego. Por ejemplo, se podrían utilizar varias texturas de manchas, ruido, metales y óxido para crear una única textura que muestre un material deteriorado por el tiempo, pero que tenga un aspecto único cada vez que se utilice. Otro ejemplo sería el de colocar diferentes plantas, rocas y árboles sobre un terreno para crear un bosque, pero siguiendo una serie de heurísticas (es decir, IA simbólica) para que el resultado sea creíble. Ya en 2011, Togelius et al. [34] publicaron una revisión sobre los diferentes enfoques que se estaban utilizando para la generación procedural de contenido en videojuegos, y cómo las metaheurísticas se utilizaban para este fin. Uno de ellos era el sistema *Ludi*, que codificaba las reglas del juego como árboles de expresión y usaba programación genética para generar conjuntos de reglas balanceadas para el videojuego. En su artículo, los autores afirmaron que los sistemas existentes hasta entonces, los cuales usaban enfoques basados en la simulación de procesos naturales, debían mejorar la consistencia, la rapidez y la personalización para el usuario si se quisieran utilizar en videojuegos comerciales a gran escala.

Otro ejemplo de uso de metaheurísticas en la generación procedural de contenido es el trabajo de Geijtenbeek et al. [14] en 2013, donde se presenta un sistema de locomoción para criaturas bípedas que utiliza un algoritmo evolutivo para optimizar los pesos de las articulaciones necesarios para que la criatura se mueva adecuadamente. En su artículo, los autores muestran como los controladores eran capaces de adaptarse a diferentes terrenos con desnivel e incluso perturbaciones externas.

3.4. ALGORITMOS EVOLUTIVOS EN JUEGOS DE ESTRATEGIA

Más recientemente, en 2021, Snell et al. [31] presentaron un enfoque evolutivo para el balanceo de videojuegos de estrategia en tiempo real. En su estudio, utilizaron diferentes variables dentro del videojuego para conseguir variaciones en la dificultad de un nivel sin desviarse demasiado de la experiencia original del mismo. Concluyeron que su enfoque era capaz de generar niveles similares, pero que se sentían diferentes, lo que permitía al jugador disfrutar de una experiencia más variada sin ser injusta.

3.4. Algoritmos evolutivos en juegos de estrategia

El bot presentado en este proyecto utiliza como base el trabajo de Ematerasu et al. [5], quienes desarrollaron el motor *Scripts of Tribute* (SoT) para el videojuego de cartas *Tales of Tribute* de la saga *The Elder Scrolls*. Pero este no es el único ni el primer motor capaz de simular partidas de juegos de cartas. Ya en 2002, MagiSoft desarrolló el software *Magic Workstation* [21], que no solo se usó para jugar partidas online y que los jugadores pudieran enfrentarse entre ellos, sino que también les permitía construir sus propios mazos. *Magic Workstation* funcionaba con un gran número de juegos de cartas, como *Magic: The Gathering* o *Yu-Gi-Oh!*, pero en 2014 salió al mercado un videojuego que estaba pensado desde el principio para ser jugado en línea: *Hearthstone*. Este videojuego de cartas se convirtió rápidamente en uno de los más populares del mundo, y su comunidad de jugadores creció exponencialmente. Sin embargo, aunque ya era un entorno virtual de salida, no existía un software específico para crear y probar bots. En 2017, el equipo de *HearthSim* lanzó *Sabberstone* [18], un motor de simulación que cumplía dicha función. Este motor fue a su vez el que inspiró el trabajo que acabó sirviendo como base de conocimiento para el desarrollo de este TFM.

Paralelamente a los simuladores de juegos de cartas, también se han desarrollado motores para otros géneros, un ejemplo de ello es *Video Game Championships* (VGC), un simulador de combates *Pokémon* desarrollado por Reis et al. [30] en 2019. Este motor también se utiliza para simular partidas (o combates en este caso) y, aunque no es un videojuego de cartas, sí que utiliza un sistema de combate por turnos y contiene una gran cantidad de variables a tener en cuenta.

3.4.1. Optimización de agentes de *Hearthstone* mediante un algoritmo evolutivo

El título de esta sección también es el título (traducido del inglés) del trabajo que ha servido como punto de partida para el desarrollo del bot de este proyecto. Es un paper de García-Sánchez et al. [11], el director de este TFM. En su artículo, los autores utilizan un algoritmo evolutivo para optimizar el comportamiento de un bot en el motor de simulación de *Sabberstone* mencionado anteriormente.

CAPÍTULO 3. ANTECEDENTES Y ESTADO DEL ARTE

Uno de los agentes generados durante el desarrollo del paper quedó segundo en la competición de IA de *Sabberstone* de 2018, lo que demuestra la efectividad de su enfoque coevolutivo y conocimiento experto. Primero, utilizaron dicho conocimiento para definir la lista de pesos que el bot utilizaría para generar la evaluación de cada jugada. Este enfoque centrado en datos de conocimiento preexistentes se mantiene en el bot desarrollado en este proyecto, pero con la diferencia de que el autor de este TFM nunca había jugado a *Tales of Tribute* antes de empezar a desarrollarlo, por lo que no se tenía ningún conocimiento previo sobre el videojuego. De esta manera, una de las primeras diferencias entre ambos trabajos es que el bot de SoT se basa en un conocimiento previo más general sobre los juegos de cartas (principalmente conocer las reglas del videojuego), aportando heurísticas para casos muy concretos, mientras que el bot de García-Sánchez et al. utiliza los cimientos de un jugador con mucha experiencia en *Hearthstone*. Además, es necesario tener en cuenta que aunque ambos sean videojuegos de cartas, las reglas y mecánicas de juego son diferentes, lo que hace que el conocimiento experto no sea directamente aplicable entre ambos videojuegos. Por ejemplo, en *Hearthstone* cada jugador tiene un sistema de vida que se reduce a medida que recibe daño, mientras que en *Tales of Tribute* cada jugador tiene un sistema de puntos de prestigio, que van aumentando a lo largo de la partida.

Sin embargo, aunque el diseño y elección de los pesos sea diferente, la sección del algoritmo evolutivo es muy similar. En el caso del bot de *Sabberstone*, no se tenía acceso a bots preexistentes contra los que competir, por lo que los autores optaron por utilizar una estrategia de coevolución. Siguiendo este enfoque, una población de 10 bots se enfrentaba entre sí para decidir cuáles eran los mejores. Tras el torneo, los mejores bots se seleccionaban para crear una nueva generación de bots, que se enfrentaban entre sí de nuevo. Despues de un número específico de generaciones, el bot con mejor rendimiento se seleccionaba como el ganador. Este enfoque de coevolución se ha mantenido en el bot desarrollado en este proyecto, pero se ha expandido gracias a la posibilidad de utilizar bots preexistentes en *Scripts of Tribute*. Como se explica de forma más detallada en la sección 7.5, el proceso de evolución cuenta con tres modos de evaluación: el modo de coevolución, el modo fijo (contra bots de *Scripts of Tribute*) y el modo híbrido (una combinación de ambos). Además, se implementó un mecanismo de “salón de la fama” para darle una opción de preservación al mejor bot de cada generación si demuestra ser mejor que el elenco de campeones preexistentes. Todo ello expandiendo el nivel de paralelismo del algoritmo evolutivo, utilizando un orquestador de procesos propio para gestionar los diferentes bots y sus enfrentamientos.

3.4.2. La hegemonía de Monte Carlo

Scripts of Tribute también se utiliza para realizar competiciones entre bots, donde los participantes envían uno o varios agentes para competir entre sí. Por

3.4. ALGORITMOS EVOLUTIVOS EN JUEGOS DE ESTRATEGIA

el momento se han realizado dos competiciones, una en 2023 y otra en 2024. En ambas, el bot ganador fue desarrollado por Adam Ciężkowski y Artur Krzyżyński [2], quienes utilizaron un enfoque basado en Monte Carlo Tree Search (MCTS) junto con optimización de pesos mediante algoritmos evolutivos para la toma de decisiones de su bot. El algoritmo de optimización de búsqueda MCTS lleva años utilizándose en la creación de IA para videojuegos, especialmente en aquellos con un alto número de variables y acciones posibles. Ya en 2018, se utilizó para crear un bot capaz de jugar a *Hearthstone* [41], mientras que un año antes, AlphaZero comparaba diferentes búsquedas realizadas con MCTS para elegir la mejor jugada en partidas de ajedrez [29].

En el caso del bot de Adam y Artur, utilizan MCTS para explorar el espacio de búsqueda en aquellas situaciones en las que explorarlo completamente sería inviable. De esta manera, su bot genera un árbol parcial de acciones enfocándose en aquellas jugadas que tienen más probabilidades de ser las mejores según su función de evaluación. Es decir, para cada jugada, el bot simula varias partidas a partir de esa jugada de forma paralela, y en función del resultado de esas partidas, asigna un valor a la jugada. Dado que hay componentes de aleatoriedad en el videojuego, como el robo de cartas, cuantas más partidas simule, más preciso será el valor que le asigne a la jugada. Sin embargo, esto también significa que el tiempo de procesamiento del bot aumenta, por eso sus autores tuvieron que añadir un límite de tiempo para la generación del árbol. Además, para limitar el factor de ramificación del árbol, utilizaron diferentes heurísticas para descartar jugadas que no merecían la pena explorar, como el uso automático de aquellas cartas que siempre se deben jugar al principio del turno (lo que llamaron “movimientos instantáneos”). La función de evaluación de su bot se basa en un conjunto de pesos que se usan para calcular el valor final de cada rama del árbol. Esos pesos están optimizados mediante un algoritmo evolutivo, lo que indica un enfoque es muy similar al utilizado por García-Sánchez et al. y por este mismo proyecto. La diferencia está en el grado de complejidad de su implementación. Dado que su algoritmo ya necesitaba de una gran cantidad de código para implementar y optimizar MCTS, su función de evaluación tiene un número reducido de pesos (aproximadamente la mitad que en este trabajo), y no incorpora diferentes modos de ajuste de los pesos del algoritmo evolutivo, dejando de lado el enfoque coevolutivo y el híbrido. Además, en el trabajo tampoco se menciona el uso de un sistema de “salón de la fama”. Aun así, su bot ha demostrado ser muy efectivo en las competiciones de IA de *Scripts of Tribute*, ganando ambas ediciones hasta la fecha (más información en la sección 4.3) y incluso imponiéndose a otros bots que también usaban MCTS.

Con el contenido de este capítulo y en espacial de la sección 3.4 se concluye el objetivo general **OG7**, que consistía en investigar y contextualizar el enfoque desarrollado frente a otras técnicas predominantes en competiciones similares de IA.

Plataforma de trabajo

Una vez establecidos los objetivos del proyecto y habiendo explorado el contexto de la inteligencia artificial en la industria del videojuego, es fundamental detallar la plataforma sobre la cual se ha erigido el trabajo práctico. Este capítulo se dedica a describir en profundidad los dos pilares que constituyen el campo de pruebas para este proyecto: el videojuego de cartas “Tales of Tribute” y, de forma más importante, su motor de simulación de código abierto, “Scripts of Tribute”. Para poder comprender las decisiones que se han tomado durante el desarrollo del bot, es necesario conocer primero las reglas del entorno en el que opera. Por ello, se comenzará explicando las reglas y mecánicas del videojuego. Posteriormente, se analizará la arquitectura del entorno de simulación que no solo hace posible el enfrentamiento entre bots, sino que también ha servido de base para competiciones académicas internacionales.

4.1. El videojuego: Tales of Tribute

“Tales of Tribute” es un videojuego de cartas para dos jugadores del género de construcción de mazos, que existe dentro del popular videojuego *The Elder Scrolls Online*. En la figura 4.1 se puede ver una captura del tablero de juego, donde se han numerado los elementos principales para facilitar su identificación [26].

El objetivo principal del juego es conseguir puntos de prestigio⁶, el cual se puede consultar en su marcador (elemento 11 en la figura 4.1). Hay varias formas de ganar: un jugador consigue la victoria si alcanza 40 puntos de prestigio y mantiene la ventaja tras el turno de su oponente. Si no lo consigue, el juego continúa hasta que alguno de los dos jugadores alcance los 80 puntos. Existe, además, una condición de victoria alternativa que se consigue al ganarse el favor de todos los “patrones”, un tipo de mecánica especial que se detallará más

⁶El prestigio es la métrica principal de victoria en “Tales of Tribute”. Se podría considerar un análogo a los “puntos de vida” en otros juegos, pero en este caso, el objetivo es acumularlos en vez de reducir los del oponente.

CAPÍTULO 4. PLATAFORMA DE TRABAJO

adelante.



Figura 4.1: Tablero de juego de Tales of Tribute, con sus componentes principales numerados. [26]

Al comienzo de cada partida, ambos jugadores deben elegir con qué conjuntos de cartas quieren jugar. Para ello, se presentan siete mazos, cada uno asociado a un patrón (elemento 3), y los jugadores se turnan para seleccionar un total de cuatro. Estos cuatro mazos, junto con un mazo estándar asociado a la “Tesorería” que está presente en todas las partidas, se barajan conjuntamente para formar el mazo de robo de la Taberna (elemento 1). De este mazo se revelan cinco cartas en el área de la Taberna (elemento 2), que son las cartas que los jugadores podrán adquirir durante la partida. La esencia de la construcción de mazos reside en que ambos jugadores comienzan con un pequeño mazo de diez cartas básicas idénticas y, a lo largo de la partida, utilizan los recursos que estas generan para comprar cartas más poderosas de la Taberna, añadiéndolas a su propio ciclo de cartas. Además, el jugador que tiene el turno en segundo lugar obtiene una pequeña recompensa de 1 moneda debido a que cuenta con la desventaja de que su oponente ha podido jugar primero.

La estructura de un turno es la siguiente, el jugador activo roba cinco cartas de su mazo personal (elemento 5) para formar su mano (elemento 7). Jugar una carta no tiene coste y, al hacerlo, se activan sus efectos y se mueve a una pila de descarte temporal (elemento 8). Al final del turno, todas las cartas sin jugar aun en la mano y en la pila de descarte temporal se mueven a la pila de reposo (elemento 9). Cuando el mazo de robo personal se agota, la pila de reposo se baraja y se convierte en el nuevo mazo de robo, completando el ciclo. De esta manera, las cartas adquiridas se incorporan progresivamente al mazo del jugador, haciéndolo cada vez más potente.

4.2. EL ENTORNO DE SIMULACIÓN: SCRIPTS OF TRIBUTE

Las cartas se pueden clasificar en dos tipos principales: acciones y agentes. Las “cartas de acción” producen un efecto inmediato (como ganar monedas u obtener puntos de poder) y después se descartan. Por otro lado, las “cartas de agente” también tienen un efecto inmediato, pero tras jugarse, se colocan en el tablero en la zona de agentes activos (elemento 6). Estos agentes permanecen en juego y su habilidad puede ser activada una vez por turno, proporcionando una ventaja continua. Sin embargo, el oponente puede destruirlos de diferentes maneras, como utilizando puntos de poder (elemento 12), donde cada punto infinge un punto de daño al agente. Existen, además, las “cartas de contrato”, que son versiones de un solo uso de las acciones y agentes, siendo eliminadas del juego permanentemente tras su primer uso o al ser derrotadas. Este tipo de cartas de contrato son las que se han utilizado para crear heurísticas específicas en el bot, ya que sus efectos pueden ser muy poderosos.

Finalmente, además de jugar y comprar cartas, los jugadores disponen de otras acciones estratégicas. La más importante es la posibilidad de interactuar con uno de los cuatro patrones elegidos al principio de la partida. Cada patrón ofrece una habilidad única a cambio de un coste en monedas, poder u otros activos. Su estado puede cambiar de neutral a favorable o desfavorable según que jugador los active y la perspectiva desde donde se mire. Ganarse el favor de todos los patrones es, como se mencionó, una de las condiciones de victoria. Por último, una mecánica muy importante es que muchas cartas poseen efectos de “combo”, los cuales se activan si en el mismo turno se han jugado previamente otras cartas del mismo mazo o patrón, incentivando así la creación de mazos con sinergias.

4.2. El entorno de simulación: Scripts of Tribute

Para poder entrenar y evaluar un agente de inteligencia artificial de forma automática, es imprescindible contar con un entorno que permita la simulación de partidas sin intervención humana. Para cumplir esa función, se utilizó *Scripts of Tribute*, que es un motor de simulación de partidas de código abierto que reimplementa las reglas de una versión específica de “Tales of Tribute”. Desarrollado en C# sobre el framework .NET 8, su propósito principal es servir como un banco de pruebas para la investigación y competición de IA, permitiendo la creación de agentes autónomos y así como su enfrentamiento. El motor se divide en dos componentes principales: un cliente con interfaz gráfica para la visualización y depuración, y una aplicación de consola para la ejecución masiva de simulaciones, siendo este último el componente esencial para el entrenamiento evolutivo.

4.2.1. Interfaz gráfica

Aunque el entrenamiento de este proyecto se basa en la ejecución de simulaciones desde la consola de comandos, el ecosistema de SoT cuenta con un cliente

CAPÍTULO 4. PLATAFORMA DE TRABAJO

gráfico desarrollado en el motor Unity. Esta herramienta permite visualizar las partidas en tiempo real, tanto entre bots como entre jugador humano y bot. Como se puede observar en la figura 4.2, la interfaz replica el tablero de juego de “Tales of Tribute”.



Figura 4.2: Tablero de juego de Scripts of Tribute. [7]

El cliente gráfico también cuenta con pantallas específicas para las acciones especiales, en la figura 4.3 se muestra un ejemplo del panel de elección de descarte de cartas.



Figura 4.3: Vista de elección de cartas en Scripts of Tribute. [7]

4.2. EL ENTORNO DE SIMULACIÓN: SCRIPTS OF TRIBUTE

Finalmente, al concluir una partida, la interfaz presenta una pantalla de resumen como la de la figura 4.4, donde se muestra el ganador y un historial completo de los movimientos realizados por ambos jugadores. Esto también se puede realizar con la versión de consola, pero la interfaz gráfica ofrece una visualización más intuitiva y accesible para el usuario.

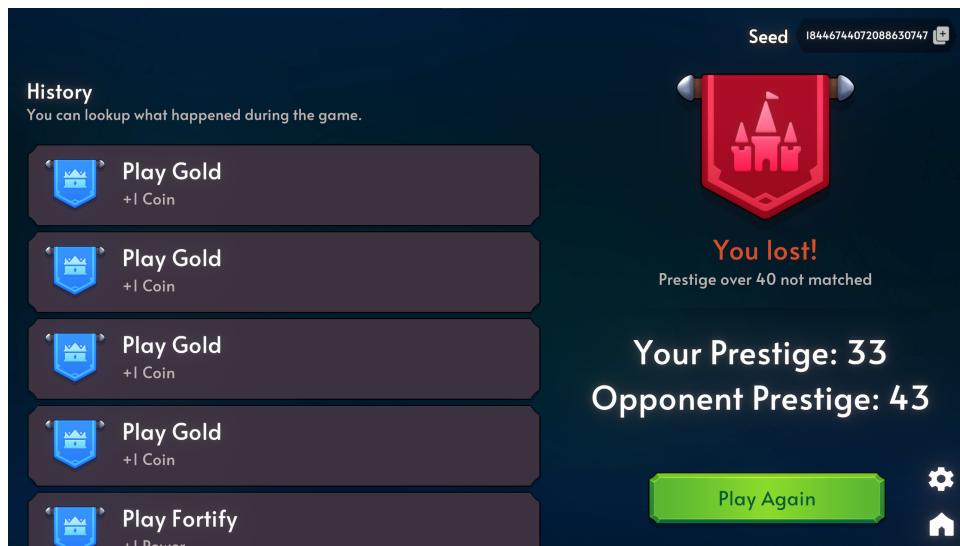


Figura 4.4: Vista de final de partida con historial de movimientos en Scripts of Tribute. [7]

4.2.2. Arquitectura del entorno

El núcleo del motor de simulación, *Scripts of Tribute-Core*, proporciona toda la lógica del juego y los puntos de entrada para el desarrollo de agentes. Para crear un nuevo bot, es necesario desarrollar una clase que herede de la clase abstracta "AI.cs" e implementar, como mínimo, dos métodos esenciales: "Select-Patron", que se encarga de la selección de patrones al inicio de la partida, y "Play", que se invoca repetidamente durante el turno del agente para que elija la siguiente acción a realizar. En cada llamada, el método "Play" recibe un objeto "GameState" que contiene una representación completa de toda la información visible en el juego, como las cartas en la mano, los agentes en el tablero o las cartas disponibles en la Taberna, permitiendo al bot tomar una decisión informada.

Si bien la interfaz gráfica es útil para la observación, la gran mayoría del trabajo de este proyecto se apoya en el "GameRunner", una aplicación de consola que forma parte del núcleo de SoT. Esta herramienta permite ejecutar un gran número de partidas entre dos bots especificados por línea de comandos, utilizando varios núcleos de la CPU si así se indica. Es precisamente este programa el que el script de entrenamiento invoca para cada una de las miles de evaluaciones que

componen el proceso evolutivo.

Actualizaciones recientes en el motor han añadido la posibilidad de interactuar con agentes escritos en lenguajes externos mediante gRPC⁷. Sin embargo, este proyecto inició su desarrollo antes de que esta funcionalidad estuviera disponible, por lo que el bot se tuvo que implementar en C# y no en Python. Esto generó un problema de comunicación entre el script de entrenamiento y el bot, ya que el motor no permite la inyección de los pesos del bot directamente desde la línea de comandos. Para resolverlo, script de ajuste de pesos (o “entrenador”) establece los pesos del bot mediante variables de entorno del intérprete de comandos utilizado (ya sea bash o Powershell), que se leen a través un método del bot antes de que comience la simulación. Esta dinámica permite que el algoritmo evolutivo en Python asigne pesos y evalúe el comportamiento de un agente que opera dentro del entorno C#.

Durante el continuo desarrollo del motor, Ematerasu et al. no solo se han añadido nuevas opciones técnicas, sino también nuevo contenido del juego. La edición de 2024 introdujo el mazo del patrón “Rey Hechicero Orgnum”, mientras que la de 2025 añadió el de “Santa Alessia”.

4.3. La competición IEEE-CoG

Más allá de ser un simple motor de simulación, *Scripts of Tribute* también se utiliza como plataforma para unas de las varias competiciones inteligencia artificial celebradas anualmente en el congreso del *IEEE Conference on Games* (CoG). El formato de la competición está diseñado para medir de forma robusta el rendimiento de los bots. Los agentes se enfrentan en un torneo de tipo *round-robin* (todos contra todos), donde el ganador se determina por el mayor porcentaje de victorias promedio tras un gran número de partidas espejo⁸. Las reglas imponen restricciones técnicas estrictas para los participantes, como un límite de tiempo de 10 segundos por turno y un uso de memoria que no debe exceder los 256 MB. Estas limitaciones obligan a los desarrolladores a buscar soluciones eficientes que equilibren la complejidad estratégica con el rendimiento computacional.

En la figura 4.5, se muestran los resultados de la edición de 2024, donde el bot ya mencionado en la sección 3.4.2 del capítulo anterior, gano por segundo a vez consecutiva. Los otros bots contaban con gran variedad de técnicas para su entrenamiento, desde MCTS hasta redes neuronales profundas e incluso aprendizaje por refuerzo.

⁷gRPC es un sistema de código abierto desarrollado por Google para gestionar llamadas a procedimientos remotos (RPC) de alto rendimiento. Permite que un programa ejecute una función en otro proceso, incluso en una máquina diferente, como si fuera una llamada local [16].

⁸En una “partida espejo”, dos agentes A y B juegan una contra el otro dos veces, intercambiando la posición de jugador inicial en cada partida para asegurar la equidad.

4.3. LA COMPETICIÓN IEEE-COG

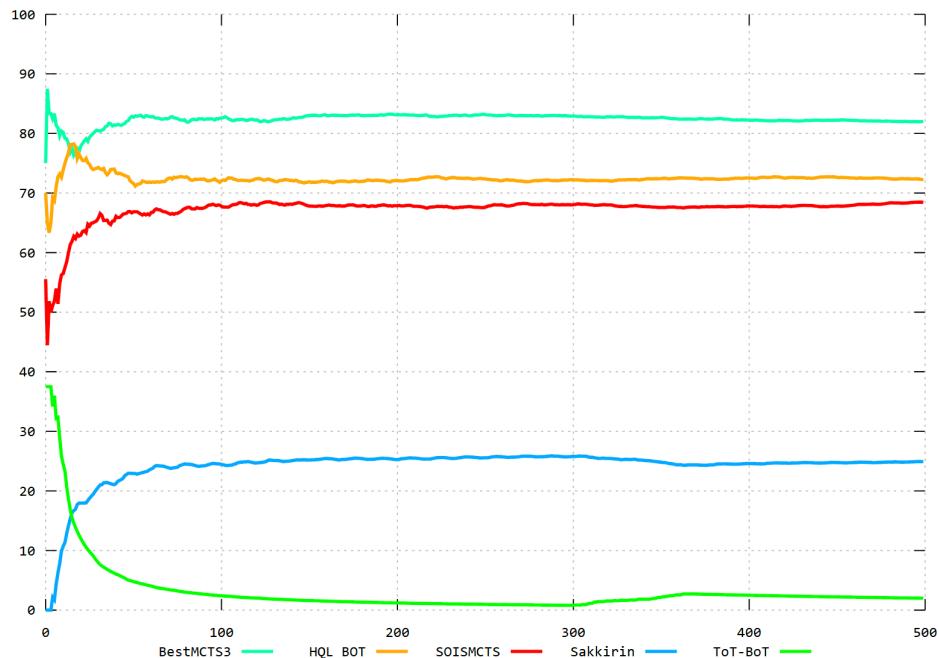


Figura 4.5: Resultados de la competición IEEE-CoG 2024. [6]

Este capítulo marca la conclusión del objetivo **OG1**, que consistía en analizar en profundidad las mecánicas del juego “Tales of Tribute”, el entorno de desarrollo “Scripts of Tribute” y adquirir las competencias necesarias en el lenguaje de programación C#.

Planificación del proyecto

En este capítulo se detallan los aspectos relacionados con los recursos que se han utilizado para el desarrollo del TFM, así como el ciclo de desarrollo del software que se ha seguido y la distribución temporal utilizada en cada tarea.

5.1. Recursos utilizados

5.1.1. Recursos software

El proyecto se puede dividir en tres partes: el desarrollo del bot, el ajuste de los pesos y la redacción del informe. Afortunadamente, existe una herramienta de código abierto y en constante mejora en la que se pueden realizar todas estas tareas: Visual Studio Code (VSCode). Este entorno de desarrollo es la navaja suiza que ha estado presente en todo el proceso, permitiendo la edición de código, la gestión de versiones, la redacción del informe en incluso la conexión con interfaz gráfica entre máquinas por SSH. Todo ello gracias a su robusto sistema de extensiones (y la gran comunidad que lo apoya), que permite añadir prácticamente cualquier funcionalidad que se necesite al editor. En el caso de la gestión de versiones, utilizando la integración con Git de VSCode, en todo momento se ha mantenido un repositorio actualizado al día en GitHub, lo que ha permitido la transferencia sencilla del progreso entre las dos máquinas utilizadas durante el desarrollo. Por último, destacar el uso de Zotero como software de gestión de referencias bibliográficas.

5.1.2. Recursos hardware

Se han contado con dos ordenadores de trabajo, el primero es el ordenador personal del autor, mientras que el segundo fue proporcionado por el director del proyecto para realizar los experimentos. Estas son sus características principales:

- Ordenador personal (Windows 11):

- Procesador: AMD Ryzen 7 5800X
- Memoria RAM: 32 GB DDR5
- Tarjeta gráfica: NVIDIA GeForce RTX 3070

■ **Ordenador de experimentación (Ubuntu 22.04):**

- Procesador: Intel i9-12900KF
- Memoria RAM: 126 GB DDR4
- Tarjeta gráfica: NVIDIA GeForce RTX 4060

5.2. Ciclo de desarrollo del software

Para el desarrollo de este TFM se ha seguido una metodología ágil a base de “sprints” regulares. Desde el 12 de noviembre de 2024, se mantuvieron reuniones de forma constante con el director del proyecto, Pablo García Sánchez, aproximadamente cada 3 semanas. En cada una, primero se procedía a repasar el progreso desde la última reunión, a menudo mostrando en tiempo real las nuevas características o los resultados obtenidos. Sobre esta base, el director proponía mejoras o correcciones a realizar para la próxima ocasión. Luego, entre ambos se decidían los siguientes pasos a seguir, mediante una comunicación abierta, lo que permitía ajustar el rumbo del proyecto según las necesidades y los resultados obtenidos.

Aunque las primeras fases del proyecto se realizaron enteramente en el PC personal del autor, en cuanto se necesitó ejecutar el entrenador, se comenzó a utilizar el ordenador proporcionado por el director para ese propósito. Esto permitió realizar experimentos de forma continuada y sencilla, mientras se seguía trabajando en el código del bot en el PC personal. Así se pudo mantener un flujo de trabajo fluido y eficiente, generando código funcional de forma regular y obteniendo resultados cada vez más desarrollados constantemente.

5.3. Distribución temporal y cronograma

Para visualizar la distribución temporal de las diferentes tareas realizadas durante el desarrollo de este TFM, se ha elaborado el siguiente diagrama de Gantt (Figura 5.1).

5.3. DISTRIBUCIÓN TEMPORAL Y CRONOGRAMA

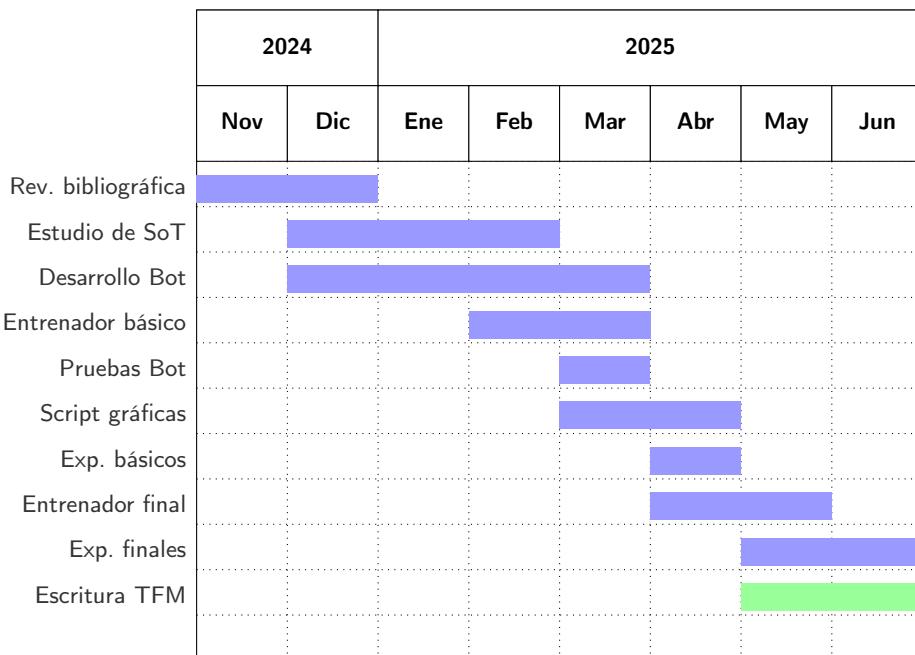


Figura 5.1: Distribución temporal de las tareas del proyecto

Como se puede observar en el diagrama, gran parte de las tareas se han desarrollado de forma secuencial, aunque algunos meses, como marzo, han tenido un gran solapamiento. Esto es debido a que los resultados preliminares que proporcionaban ciertas tareas afectaban directamente al desarrollo de otras. La escritura del TFM se concentró en el último mes y medio del proyecto, una vez que se contaba con resultados experimentales suficientes para documentar adecuadamente el trabajo realizado. Este es el contenido de cada una de las tareas:

- **Revisión bibliográfica:** se empezó revisando los trabajos ya descritos en la sección 3.4, además de aprender a jugar a Tales of Tribute y definir el alcance del proyecto.
- **Estudio de Scripts of Tribute:** en esta etapa se estudió el código fuente de Scripts of Tribute, y se empezó a trabajar con C# como lenguaje de programación, ya que este nunca había sido utilizado por el autor antes.
- **Desarrollo del bot:** se diseñó y desarrolló el bot a base de iteraciones de otros bots existentes, así como del trabajo de Pablo et al. [11].
- **Pruebas del bot:** primeras pruebas para comprobar que el funcionamiento del bot era correcto.
- **Entrenador básico:** en un principio, el entrenador solo tenía un modo de funcionamiento básico y estaba peor organizado, pero el ágil desarrollo de

CAPÍTULO 5. PLANIFICACIÓN DEL PROYECTO

esta versión permitió alcanzar resultados rápidamente a modo de prototipo.

- **Script de gráficas:** se creó un script para generar gráficas que visualizan los resultados de los experimentos realizados, así como la función del entrenador que crea los datasets necesarios para su funcionamiento.
- **Experimentos básicos:** experimentos iniciales para evaluar si el rendimiento del bot mejoraba con el ajuste de pesos.
- **Entrenador final:** desarrollo de la versión final del entrenador, incluyendo los diferentes modos de entrenamiento.
- **Experimentos finales:** experimentos pensados para la evaluación del rendimiento de los distintos modos de entrenamiento del entrenador final.
- **Escritura del TFM:** finalmente se redactó este informe, documentando todo el proceso realizado durante el desarrollo del proyecto.

Parte II

Metodología

Capítulo **6**

Diseño del agente autónomo

En este capítulo se desglosa el diseño y la implementación del agente autónomo. El enfoque principal se centra en un sistema de toma de decisiones que evalúa cada acción posible mediante una función de puntuación ponderada, cuyos pesos son optimizados por el algoritmo evolutivo descrito en el siguiente capítulo. A continuación, se detallará la arquitectura de este sistema, la definición de los pesos que conforman su “genoma” y las lógicas específicas y heurísticas que complementan su comportamiento.

6.1. Arquitectura de la toma de decisiones

El comportamiento del agente se basa en una estrategia de decisión voraz con un horizonte de planificación de un solo paso. Cuando el bot debe realizar una acción, el motor del juego le proporciona una lista de todos los movimientos legalmente posibles. Para cada uno de estos movimientos, el bot realiza una simulación interna para averiguar como cambiaría el estado de la partida al aplicarlo. A este nuevo estado se le asigna una puntuación numérica a través de una función de evaluación heurística. Finalmente, el bot selecciona y ejecuta de forma determinista el movimiento que conduce al estado con la puntuación más alta, repitiendo este ciclo hasta que la acción elegida sea la de finalizar el turno. Esto contrasta con el enfoque de MCTS, pues en dicho algoritmo no se simula solo un movimiento, sino la partida entera en múltiples posibles ramificaciones. Una de las ventajas del enfoque voraz es su rapidez, en ningún momento se llegaría ni al límite de tiempo de 10 segundos por turno, ni al de memoria de 256 MB, lo que permite al bot tomar decisiones incluso en situaciones de alta presión temporal. Además, aunque el enfoque MCTS podría ofrecer una mejor exploración de las posibles jugadas, también es cierto que cuanto más se aleja del estado actual de la partida, más incertidumbre hay sobre el valor de las jugadas debido a la alta estocasticidad del juego y menos valor tendría la evaluación de un estado futuro lejano.

CAPÍTULO 6. DISEÑO DEL AGENTE AUTÓNOMO

El punto más importante de este sistema es la función de evaluación, "Score-Move", que calcula la idoneidad de un estado del juego. Esta función es la que ejecuta la evaluación lineal ponderada, donde se calculan las diferencias entre el estado del juego antes y después de la jugada simulada a través de un conjunto de características. Cada una de estas características se multiplica por un peso específico, y la suma (o resta, en caso de gestionar datos relativos al contrincante) de todos estos componentes da como resultado la puntuación final de la acción.

La fórmula general que representa este cálculo se puede expresar de la siguiente manera:

$$\text{Puntuación}(m) = \sum_{i=1}^n w_i \cdot \Delta f_i(E_{t+1}, E_t)$$

Donde $\text{Puntuación}(m)$ es el valor asignado al movimiento m , n es el número total de características evaluadas, w_i es el peso asociado a la característica i , y Δf_i es la función que calcula la diferencia de valor de dicha característica entre el estado resultante E_{t+1} y el estado original E_t . Para garantizar que la comparación entre los diferentes movimientos sea justa y reproducible, todas las simulaciones internas se realizan utilizando una semilla pseudoaleatoria constante, lo que elimina la estocasticidad de ciertos eventos del juego, como el robo de cartas de la taberna. En el siguiente pseudocódigo se muestra el bucle de evaluación que sigue el bot para cada uno de los posibles movimientos:

Algorithm 1 Selección de la Mejor Acción en un Turno

```

1:   ▷ Lógica interna del agente para elegir una acción en el estado de juego
2:   actual  $S$ .
2: Entradas: Estado del juego  $S$ , Lista de acciones posibles  $A = \{a_1, \dots, a_n\}$ ,
   Pesos  $\vec{w}$ .
3:
4: mejorAcción  $\leftarrow$  AcciónAleatoria( $A$ )
5: mejorPuntuación  $\leftarrow -\infty$ 
6:
7: para cada acción  $a_i \in A$  hacer
8:   si  $a_i$  es “Pasar Turno” entonces
9:     puntuación  $\leftarrow -1$ 
10:  sino
11:     $S_{t+1} \leftarrow$  Simular( $S, a_i$ )
12:    puntuación  $\leftarrow$  Evaluar( $S_{t+1}, S, \vec{w}$ )           ▷ Usa la Ecuación 6.1.
13:  fin si
14:
15:  si puntuación  $>$  mejorPuntuación entonces
16:    mejorPuntuación  $\leftarrow$  puntuación
17:    mejorAcción  $\leftarrow a_i$ 
18:  fin si
19: fin para
20:
21: devolver mejorAcción
  
```

6.2. Definición de los pesos

El comportamiento estratégico del bot está enteramente definido por un conjunto de 20 pesos numéricos. Este vector de valores representa el “genoma” de un individuo dentro del algoritmo evolutivo. Cada peso está asociado a una característica específica del juego, permitiendo al proceso de optimización ajustar con precisión la forma en que el bot valora cada aspecto de la partida. La Tabla 6.1 detalla cada uno de estos pesos y su función dentro de la evaluación.

La transferencia de estos pesos desde el entrenador al bot es un componente clave del sistema. Para evitar la sobrecarga de reescribir ficheros constantemente, el mecanismo principal se basa en el uso de variables de entorno. El script de Python convierte el genoma de un individuo (un array de 20 números) en una cadena de texto separada por comas. Esta cadena se asigna a una variable de entorno antes de lanzar el proceso del “GameRunner”. Al iniciarse, el bot detecta estas variables, las lee, y utiliza sus valores para configurar la función de evaluación para esa partida en concreto.

CAPÍTULO 6. DISEÑO DEL AGENTE AUTÓNOMO

Tabla 6.1: Descripción de los pesos del genoma del agente evolutivo.

Categoría	Peso (Enum)	Descripción
Agentes y Combate	A_HEALTH_REDUCED	Penalización por el daño recibido por un agente propio, o bonificación por el daño infligido a un agente enemigo.
	A_KILLED	Penalización si un agente propio es destruido, o bonificación equivalente si se destruye un agente enemigo.
	A_OWN_AMOUNT	Pondera la importancia de tener un mayor número de agentes en el tablero.
	A_ENEMY_AMOUNT	Pondera la importancia de que el enemigo tenga un mayor número de agentes en el tablero.
Recursos y Cartas	CURSE_REMOVED	Bonificación por eliminar una carta de maldición del propio mazo.
	C_TIER_POOL	Valor asignado a las cartas del mazo según una clasificación de tier predefinida.
	C_TIER_TAVERN	Penalización por dejar cartas de tier alto disponibles para el oponente en la taberna.
	C_GOLD_COST	Valor asignado a las cartas en función de su coste en oro.
	C_OWN_COMBO	Bonificación por tener varias cartas del mismo patrón en el mazo, aumentando el potencial de combo.
	C_ENEMY_COMBO	Pondera el potencial de combo del mazo del oponente.
	COIN_AMOUNT	Pondera la diferencia de monedas generadas en un turno.
	POWER_AMOUNT	Pondera la diferencia de poder generado en un turno.
	PRESTIGE_AMOUNT	Pondera la diferencia de prestigio ganado o perdido.
Cartas Específicas	H_DRAFT	Peso que se utiliza como multiplicador defensivo para valorar la acción de quitarle una carta útil al oponente ("hate drafting").
	T_TITHE	Valor asignado a la carta "Tithe", que permite una activación de patrón adicional.
	T_BLACK_SACRAMENT	Valor asignado a "Black Sacrament", que permite destruir un agente enemigo.
	T_AMBUSH	Valor asignado a "Ambush", que permite destruir hasta dos agentes enemigos.
	T_BLACKMAIL	Valor asignado a "Blackmail", que proporciona poder o prestigio.
Patrones	T_IMPRISONMENT	Valor asignado a "Imprisonment", similar a "Blackmail" pero con mayor potencial.
	P_AMOUNT	Valor asignado a cualquier acción que cambie el favor de un patrón.

6.3. Lógicas específicas y heurísticas

Para aquellos casos especiales en los que el bot puede no ser capaz de discernir la importancia de una jugada, como es el caso de la aparición en la taberna de ciertas cartas importantes, se implementaron una serie de lógicas específicas y heurísticas codificadas manualmente pero con su propio peso asignado.

El ejemplo más representativo se encuentra en la función "CalculateTavernScore". Esta función no evalúa el valor de adquirir una carta de la taberna para uno mismo, sino que calcula el valor estratégico de eliminarla de la taberna para que el oponente no pueda adquirirla. Esta técnica, popularmente denominada como "hate drafting", se implementó tras ver a algunos jugadores explicar sus movimientos mientras jugaban. En los vídeos se observaba que a veces preferían quitarle una carta al oponente aunque no les fuera útil a ellos mismos, pues que el rival la utilizase sería peor que los recursos gastados en comprar la carta. Este cálculo es puramente heurístico y depende del estado actual de la partida. Por

6.3. LÓGICAS ESPECÍFICAS Y HEURÍSTICAS

ejemplo, para la carta “Tithe”, que permite una activación de patrón adicional, el bot la valora más positivamente de forma defensiva (es decir, quitarla para que el oponente no la compre) si el oponente está cerca de alcanzar la victoria por prestigio, ya que un turno con dos acciones de patrón podría ser decisivo para él. De forma similar, se evalúan otras cartas clave como “Ambush”, cuyo valor aumenta exponencialmente cuantos más agentes tenga el oponente en el tablero.

Además de esta lógica centrada en la taberna, existen otras heurísticas integradas en la evaluación. En “CalculateScoreAgents”, por ejemplo, se diferencia entre simplemente dañar a un agente y destruirlo por completo, asignando una penalización mucho mayor a esto último. Asimismo, en la función “ComputeCardPoolValue” se aplica una lógica especial para las cartas de “maldición”: tenerlas en el propio mazo es negativo, pero conseguir que el oponente adquiera una es positivo. Esta combinación de una función de evaluación general y optimizable con heurísticas específicas para casos concretos intenta unificar el conocimiento del jugador con la potencia bruta de la optimización evolutiva. El sistema evolutivo se encarga de ajustar el comportamiento general, mientras que el conocimiento codificado en las heurísticas garantiza que el bot actúe de forma decisiva en momentos clave de la partida.

El desarrollo del bot marca el cumplimiento del **OG2**, que establece el diseño e implementación un agente inteligente en C# para “Scripts of Tribute”, cuya toma de decisiones se base en una evaluación heurística del estado del juego mediante una función de fitness ponderada, incorporando conocimiento específico del dominio.

Sistema de entrenamiento evolutivo

Una vez definida la arquitectura del agente autónomo en el capítulo anterior, el siguiente paso lógico es el proceso de optimización de los pesos que rigen su comportamiento. La función de evaluación del bot, aunque incorpora heurísticas y conocimiento experto, depende fundamentalmente del vector de pesos para ponderar la importancia de cada faceta del juego. Encontrar un conjunto de pesos óptimo para esta función es un problema de búsqueda en un espacio de soluciones de alta dimensionalidad, una tarea para la cual los algoritmos evolutivos están especialmente indicados. Este capítulo se dedica a describir en detalle el “entrenador”: un sistema desarrollado en Python que orquesta un algoritmo evolutivo con el objetivo de encontrar el genoma (el conjunto de pesos) que maximice el rendimiento del bot. Se explorará su arquitectura general, los mecanismos de comunicación con el simulador, la implementación del algoritmo evolutivo utilizando la librería *inspyred*, y las distintas estrategias de evaluación de fitness que se han diseñado e implementado.

7.1. Arquitectura general del entrenador

El sistema de entrenamiento está formado por una serie de scripts modularizados y diseñados para ejecutar el proceso de optimización. Su objetivo final es determinar el conjunto de pesos más óptimo para la función de evaluación del agente descrito en el Capítulo 6.2. El sistema se apoya en la librería de código abierto *inspyred* [12], un paquete que proporciona los componentes fundamentales del proceso, como la gestión de la población, los operadores de selección y variación, y el bucle evolutivo principal. Sobre esta base, se ha construido una arquitectura a medida que permite una extracción de datos mucho más granular, así como los modos de evaluación fijo, coevolutivo e híbrido y un sistema de “salón de la fama” generacional.

CAPÍTULO 7. SISTEMA DE ENTRENAMIENTO EVOLUTIVO

El flujo de trabajo del entrenador sigue el paradigma de un algoritmo evolutivo, operando a través de un ciclo iterativo de mejora generacional. El proceso comienza con la generación de una población inicial de individuos, donde cada individuo representa un “genoma” completo, es decir, un vector de 20 pesos aleatorios de 0.0 a 1.0. A partir de ahí, se inicia el bucle principal. En cada generación, se lleva a cabo la fase de “evaluación”, donde cada individuo de la población es puesto a prueba. Para ello, el entrenador lanza una serie de partidas simuladas en el entorno de *Scripts of Tribute*, pasando los pesos del individuo al bot. La tasa de victorias obtenida en estas partidas se convierte en la puntuación de *fitness* de dicho individuo. Una vez que todos los individuos han sido evaluados, se aplica la fase de “selección”, donde los individuos con mejor *fitness* se eligen para convertirse en los “padres” de la siguiente generación. Posteriormente, en la fase de “variación”, estos padres se combinan mediante operadores de cruce y mutación para crear una nueva población de “hijos”. Finalmente, se escogen los individuos con mayor *fitness* (pues en este caso se ha empleado elitismo) y el ciclo vuelve a comenzar. Este proceso se repite hasta que se alcanza una condición de terminación, como un número máximo de evaluaciones, momento en el cual el mejor individuo que se haya encontrado a lo largo de toda la ejecución es proclamado como la solución más óptima encontrada. Como resumen de este proceso, en el algoritmo 2 se muestra el pseudocódigo del bucle evolutivo principal.

Algorithm 2 Proceso del Algoritmo Evolutivo

```
1:      > Orquestación del entrenamiento para optimizar los pesos del agente.
2: Entradas: Parámetros de configuración (tamaño de población  $\mu$ , máx. evaluaciones, etc.).
3:
4:  $P \leftarrow$  InicializarPoblaciónAleatoria( $\mu$ )
5:  $H \leftarrow \emptyset$                                 > Inicializar Salón de la Fama interno.
6:  $Evaluar(P, H)$ 
7:
8: mientras no se cumpla el criterio de parada hacer
9:    $Padres \leftarrow$  Seleccionar( $P$ )
10:   $Hijos \leftarrow$  Variar( $Padres$ )
11:   $Evaluar(Hijos, H)$ 
12:   $P \leftarrow$  Reemplazar( $P, Hijos$ )
13:   $H \leftarrow$  GestionarSalónDeLaFama( $P, H$ )          > Algoritmo 3.
14: fin mientras
15:
16: devolver MejorIndividuo( $P \cup H$ )
```

7.2. COMUNICACIÓN ENTRENADOR-SIMULADOR

7.2. Comunicación entrenador-simulador

La arquitectura elegida para esta comunicación se basa en la creación de subprocessos. Por cada evaluación que el entrenador necesita realizar, el script de Python utiliza la librería `subprocess` para lanzar una nueva instancia del ejecutable `GameRunner`. Obviamente no lanza todos los procesos de la generación al mismo tiempo, si no que limita su creación al número de núcleos marcados por un argumento de entrada del entrenador. Este enfoque permite aislar cada simulación en su propio proceso y entorno. Una vez que el `GameRunner` finaliza su ejecución, escribe en su salida estándar (`stdout`) un resumen de los resultados, como el número de victorias para cada jugador. El script de Python captura esta salida, la procesa para extraer los datos relevantes y así asigne una puntuación al individuo que estaba siendo evaluado.

Para la comunicación en sentido contrario se utiliza el método de variables de entorno ya descrito en el capítulo anterior. Más específicamente, antes de lanzar cada subprocesso del `GameRunner`, entrenador realiza los siguientes pasos:

1. Convierte el genoma del individuo (el vector de 20 pesos de tipo `float`) en una única cadena de texto separada por comas.
2. Utiliza el módulo `os` de Python para establecer esta cadena como el valor de una variable de entorno específica, como por ejemplo `EVO_BOT_P1_WEIGHTS`.
3. Lanza el subprocesso del `GameRunner`, el cual hereda una copia del entorno del proceso padre, incluyendo la variable recién establecida.

Al iniciarse la partida, el código del bot está programado para buscar y leer estas variables de entorno. Si las encuentra, procesa la cadena de texto para reconstruir el vector de pesos y se configura a sí mismo para la partida. Si no las encuentra, simplemente usa los pesos que lee desde un archivo de configuración que contiene los pesos del mejor individuo encontrado en el último experimento. Para garantizar la independencia de cada simulación, las variables de entorno se limpian y restauran a su estado original después de cada evaluación, evitando así que los pesos de un individuo se filtren accidentalmente en la evaluación del siguiente.

Como el sistema de gRPC no existía cuando se empezó a trabajar en el proyecto y se estimó que la funcionalidad brindada por la librería `inspyred` era indispensable para el desarrollo del algoritmo evolutivo, se optó por esta solución de comunicación entre procesos, la cual se ha intentado mantener lo más eficiente posible dentro de las limitaciones de su arquitectura.

7.3. El algoritmo evolutivo con `InsPyred`

El tipo de algoritmo evolutivo escogido para este trabajo sigue una Estrategia Evolutiva, concretamente la implementación canónica que proporciona la clase

CAPÍTULO 7. SISTEMA DE ENTRENAMIENTO EVOLUTIVO

`inspyred.ec.ES` [1]. Las estrategias evolutivas son especialmente adecuadas para problemas de optimización en espacios de búsqueda con valores reales, como es el caso de los 20 pesos que conforman el genoma del agente. La característica más notable de la implementación de `inspyred.ec.ES` es el uso de una mutación autoadaptativa, lo que significa que el genoma de cada individuo no solo contiene el vector de pesos de la solución, sino que también cuenta con un conjunto de “parámetros de estrategia” [1]. Estos parámetros, uno por cada peso, funcionan como tasas de mutación individuales que evolucionan junto con la solución. De esta forma, el algoritmo no depende de una tasa de mutación global y fija, sino que aprende por sí mismo cómo de grande debe ser el paso de mutación para cada peso, permitiendo dar grandes saltos exploratorios en las primeras generaciones y ajustes más finos en las últimas. La librería gestiona este proceso de forma transparente, envolviendo automáticamente las funciones de generación y evaluación para añadir y quitar estos parámetros de estrategia sin necesidad de modificar la lógica específica del problema [1].

La configuración del algoritmo utiliza algunos de los componentes que la clase `inspyred.ec.ES` ofrece por defecto y otros personalizados para adaptarse a las necesidades del proyecto. A continuación se detallan sus componentes clave:

- **Generador:** Se utiliza un método a medida, que crea un individuo con pesos aleatorios. La librería la envuelve con su propio generador para añadirle los parámetros de estrategia iniciales [1].
- **Evaluador:** Se emplean los orquestadores personalizados, que calculan la tasa de victorias. Internamente, `inspyred` se asegura de que solo los pesos de la solución, y no los parámetros de estrategia, se pasen a esta función [1].
- **Selector:** Se utiliza la selección por defecto de la clase, que simplemente selecciona a toda la población para que actúen como padres en la siguiente generación [1].
- **Variador:** El operador de variación es un método interno de la clase `ES` que implementa la mutación gaussiana autoadaptativa. Primero modifica los parámetros de estrategia de cada individuo y luego los utiliza como la desviación estándar para aplicar una mutación gaussiana a los pesos de la solución [1].
- **Reemplazador:** Se emplea la estrategia de reemplazo por defecto, conocida como “reemplazo $(\mu + \lambda)$ ”. En esta estrategia, la nueva generación se forma seleccionando a los mejores individuos del conjunto combinado de los padres de la generación actual y sus descendientes, asegurando que las mejores soluciones nunca se pierdan (como una forma de elitismo) [1].

7.4. EL SALÓN DE LA FAMA

7.4. El salón de la fama

Uno de los problemas inherentes a los algoritmos evolutivos es el conocido como “olvido catastrófico”. A medida que una población evoluciona durante muchas generaciones, su composición cambia. Puede especializarse en derrotar a sus contemporáneos inmediatos, pero al hacerlo, podría perder la capacidad de vencer a oponentes fuertes de generaciones muy anteriores, simplemente porque esos oponentes ya no existen en la población para ejercer presión selectiva. Para mitigar esta y otras patologías de la coevolución, una de las soluciones más extendidas es la implementación de un mecanismo de memoria a largo plazo [22]. El sistema de “salón de la fama” (*Hall of Fame*, HoF), implementado en este proyecto actúa como un archivo de campeones. Su función es preservar a los mejores individuos encontrados a lo largo de una misma carrera de entrenamiento⁹. Al forzar a los nuevos candidatos a competir contra estos campeones del pasado, se intenta que el conocimiento estratégico no se degrade y que cualquier “mejora” sea un progreso genuino y no una simple especialización contra oponentes débiles. Este mecanismo, además, introduce una capa de dinámica coevolutiva incluso en el modo de entrenamiento fijo, ya que la población no solo se mide contra un baremo estático, sino también contra un conjunto de sus propios ancestros más capaces, los cuales van cambiando a lo largo del tiempo.

El diseño de este salón de la fama se inspira en las estrategias analizadas por Nogueira et al. en su trabajo sobre algoritmos coevolutivos competitivos del 2013 [22]. En concreto, se ha implementado una variante de su estrategia HofCC-Quality. El objetivo de este enfoque es mantener en memoria únicamente a los individuos más capaces, eliminando periódicamente a los más débiles. En esta implementación, la “debilidad” de un campeón del salón de la fama se define por el número de derrotas que sufre contra la población de la generación actual. El sistema funciona bajo las siguientes reglas:

- El salón de la fama mantiene un tamaño máximo configurable.
- Cuando se encuentra un nuevo campeón (un individuo con el mayor *fitness* de su generación) y este no forma parte ya del salón de la fama, se considera su inclusión.
- Si el salón de la fama no está lleno, el nuevo campeón se añade directamente.
- Si el salón de la fama está lleno, el nuevo campeón debe “desafiar” al miembro más débil actualmente en el salón (aquel con más derrotas en la última generación). El nuevo campeón solo reemplazará al miembro

⁹Una “carrera” o “run” es simplemente un proceso de optimización evolutiva desde la primera generación hasta la última. Un experimento o entrenamiento está compuesto por varias carreras.

CAPÍTULO 7. SISTEMA DE ENTRENAMIENTO EVOLUTIVO

existente si gana este enfrentamiento directo con una tasa de victorias superior al 50%.

- Adicionalmente, para evitar el estancamiento, un mecanismo de “poda” se activa periódicamente cada cierto número de generaciones. Este proceso elimina un porcentaje configurable de los miembros más débiles del salón de la fama, forzando la renovación y asegurando que solo los contendientes más consistentemente fuertes permanezcan.

Esta lógica se puede resumir formalmente en el algoritmo 3.

Algorithm 3 Gestión del Salón de la Fama Interno

```

1:      ▷ Ejecutado al final de cada generación  $g$ , utilizando la población de
      supervivientes  $P$ .
2: Entradas: Población  $P$ , Salón de la Fama  $H$ , Generación actual  $g$ , Resultados detallados  $D$ .
3:
4: campeón  $\leftarrow$  MejorIndividuo( $P$ )
5: si  $\text{ID}(\text{campeón}) \notin \text{IDs}(H)$  entonces
6:   si  $|H| < \text{tamaño\_máximo}$  entonces
7:      $H \leftarrow H \cup \{\text{campeón}\}$ 
8:   sino
9:     si  $\text{ID}(\text{campeón}) \in \text{Claves}(D)$  entonces
10:     $m\_débil \leftarrow \text{MiembroMásDébil}(H, D)$ 
11:     $\text{tasa\_victorias} \leftarrow \text{Enfrentar}(\text{campeón}, m\_débil)$ 
12:    si  $\text{tasa\_victorias} > 0.5$  entonces
13:       $H \leftarrow (H \setminus \{m\_débil\}) \cup \{\text{campeón}\}$ 
14:    fin si
15:  fin si
16: fin si
17: fin si
18:
19:      ▷ Realizar la poda de forma periódica independientemente de si hubo un
      nuevo campeón.
20: si  $g \text{ (mód } \text{frecuencia\_poda)} = 0$  entonces
21:    $n\_poda \leftarrow \lfloor |H| \cdot \text{porcentaje\_poda} \rfloor$ 
22:   si  $n\_poda > 0$  entonces
23:      $H\_ordenado \leftarrow \text{OrdenarPorDerrotas}(H)$ 
24:      $H \leftarrow \text{EliminarPeores}(H\_ordenado, n\_poda)$ 
25:   fin si
26: fin si

```

En un principio se diseñó otro tipo salón de la fama, el cual abordaba el problema del olvido catastrófico de una forma diferente. En lugar de operar entre

7.5. ARQUITECTURA DEL ORDEN, LA EVALUACIÓN Y LA PARALELIZACIÓN

generaciones, operaba entre carreras. Este salón de la fama global almacenaba los campeones de cada carrera y los mantenía en memoria durante todo el ciclo de vida del entrenador para usarse en los siguientes entrenamientos. Sin embargo, este enfoque no era estadísticamente válido, ya que los campeones de una carrera no deberían afectar a los campeones de otra. Para generar resultados estadísticamente significativos, cada entrenamiento debe ser independiente y no influenciado por los resultados de otros. Por lo tanto, se optó por el salón de la fama interno detallado en esta sección.

7.5. Arquitectura del orden, la evaluación y la paralelización

La fase de evaluación es, sin duda, el componente más crítico y costoso computacionalmente de cualquier algoritmo evolutivo. Para que el proceso de entrenamiento sea viable en un tiempo razonable, es imprescindible ejecutar las miles de simulaciones de partidas de forma lo más paralela posible. Para ello, se ha diseñado una arquitectura de evaluación basada en el módulo `concurrent.futures` de Python, que gestiona un conjunto de procesos trabajadores (*workers*) para distribuir la carga de trabajo entre los múltiples núcleos de la CPU.

La comunicación del estado entre los procesos, como la gestión del salón de la fama interno, se gestiona utilizando objetos proxy proporcionados por la librería `multiprocessing.Manager`. Esta base tecnológica permite la implementación de diferentes “orquestradores” de evaluación, que son funciones de alto nivel encargadas de definir y distribuir las tareas específicas para cada una de las estrategias de fitness. La métrica de *fitness* utilizada en todas ellas es la tasa de victorias normalizada, calculada como un porcentaje entre 0 y 100 para asegurar que los resultados sean comparables entre generaciones. Este nivel de personalización y sistemas específicos para el proyecto es necesario no solo para la implementación de los modos de evaluación, sino también para la correcta integración de otras características necesarias en el sistema. Por ejemplo, en el capítulo 4 se describió que el propio juego cuenta con una mecánica para nivelar la desventaja inherente al orden de los jugadores. Esta mecánica hace que el jugador que tiene su turno en segundo lugar empiece su turno con 1 moneda más. Bien, incluso algo tan pequeño podría afectar a la tasa de victorias después de miles de partidas. Por eso, el sistema de evaluación se encarga de hacer que el bot juegue contra su oponente siendo primero un 50% de las partidas y segundo el otro 50%, y luego recoge acordemente el resultado de las partidas leyendo la salida estándar del `GameRunner`. Este tipo de características no podrían implementarse si se usase únicamente la librería `inspyred` sin una capa de orquestación que gestione la comunicación y la lógica de evaluación.

7.5.1. Modo fijo: evaluación contra oponentes estáticos

El modo fijo constituye el enfoque de evaluación más tradicional, donde el rendimiento de cada individuo se mide contra un conjunto estático de oponentes predefinidos. La lógica está encapsulada en la función orquestadora, que itera sobre la lista de candidatos a evaluar y, para cada uno, añade una tarea al conjunto de procesos. Cada tarea consiste en invocar a una función trabajadora que ejecuta una serie de partidas contra los bots de referencia de *Scripts of Tribute* (ej. *PatronFavorsBot*) y contra los campeones del salón de la fama interno. En este caso, el orquestador evalúa únicamente al conjunto de hijos. Los padres de la generación anterior no son re-evaluados, ya que su *fitness* contra un conjunto de oponentes estático no ha cambiado. Una vez finalizadas todas las simulaciones para un candidato, el trabajador devuelve su tasa de victorias, que el orquestador recoge para construir el vector de *fitness* de la generación.

Formalmente, el *fitness* para un individuo i en modo fijo se calcula como su tasa de victorias global contra el conjunto de bots estáticos B y el conjunto de campeones en el salón de la fama H , según la Ecuación 7.1.

$$Fitness_i = \frac{\sum_{b \in B} v(i, b) + \sum_{h \in H} v(i, h)}{(|B| \cdot g_B) + (|H| \cdot g_H)} \cdot 100 \quad (7.1)$$

Donde $v(i, j)$ representa el número de victorias del individuo i contra el oponente j . Los parámetros $|B|$ y $|H|$ son el número de bots estáticos y de miembros en el salón de la fama, respectivamente, mientras que g_B y g_H son el número de partidas jugadas contra cada tipo de oponente, tanto los fijos como los del salón de la fama.

7.5.2. Modo coevolución: competición interna

La evaluación por coevolución, por su parte, mide la calidad de los individuos en relación a sus propios compañeros, creando una especie de “carrera armamentística” interna. En cada generación, se forma un “conjunto de competición” P , que consiste en la unión de los padres de la generación anterior y los hijos recién creados. A continuación, se genera una lista exhaustiva de enfrentamientos, que incluye partidas de todos contra todos (*round-robin*) entre los miembros de P , así como enfrentamientos de cada miembro de P contra los campeones del salón de la fama H . Cada una de estas tareas se envía al conjunto de procesos para su ejecución en paralelo.

El cálculo del *fitness* en este modo, aunque también es una tasa de victorias, se basa en un conjunto de oponentes completamente dinámico. La Ecuación 7.2 formaliza este cálculo.

$$Fitness_i = \frac{\sum_{p \in P, p \neq i} v(i, p) + \sum_{h \in H} v(i, h)}{(|P| - 1) \cdot g_P + (|H| \cdot g_H)} \cdot 100 \quad (7.2)$$

Aquí, el sumatorio principal recorre a todos los demás individuos p en el conjunto de competición P , y $v(i, p)$ son las victorias de i contra ellos. El parámetro

7.5. ARQUITECTURA DEL ORDEN, LA EVALUACIÓN Y LA PARALELIZACIÓN

g_P es el número de partidas por enfrentamiento entre pares. El componente del salón de la fama es idéntico al del modo fijo. Al final del proceso, el orquestador agrega todos los resultados para calcular la tasa de victorias final de cada individuo.

7.5.3. Modo híbrido: combinando estrategias

El modo híbrido se diseñó para combinar la guía inicial de conocimiento de los bots preexistentes del modo fijo con el potencial de innovación de la coevolución. Esta modalidad permite dividir una única carrera evolutiva en múltiples segmentos, cada uno con su propia estrategia de evaluación. El comportamiento se controla a través del parámetro `-hybrid_schedule_str`, que define la secuencia y duración relativa de cada segmento (ej. 40% de fijo, 20% de coevolución y otro 40% de fijo). El bucle principal del entrenador es el responsable de interpretar este cronograma y de invocar dinámicamente al orquestador de evaluación correspondiente para cada fase de la evolución durante el número de generaciones adecuado.

7.5.4. Análisis comparativo del coste computacional

Como se ha descrito, los modos de evaluación fijo y de coevolución operan de formas muy distintas, lo que resulta en una diferencia sustancial en el coste computacional por cada generación. Para ilustrar este punto, a continuación se desglosa el número total de simulaciones de partidas requeridas por cada modo en una única generación. Es necesario aclarar que los valores totales dependen en gran medida de la configuración del experimento, como el número de partidas por enfrentamiento (tanto fijo como coevolutivo), el tamaño de la población o el número de miembros del salón de la fama. En este caso se han utilizado valores sencillos y representativos, pero que en algunos casos se ven incrementados para los experimentos finales.

- **Coste en Modo Fijo:** En este modo, solo se evalúa a los 10 individuos "hijos" de la nueva generación. Cada uno de ellos se enfrenta al conjunto de oponentes estáticos y a los miembros del salón de la fama.
 - *Partidas contra bots estáticos:* $10 \text{ individuos} \times 3 \text{ bots} \times 10 \text{ partidas/bot} = 300 \text{ partidas.}$
 - *Partidas contra el salón de la fama:* $10 \text{ individuos} \times 3 \text{ miembros del HoF} \times 3 \text{ partidas/miembro} = 90 \text{ partidas.}$
 - **Total por generación en modo fijo: 390 partidas.**
- **Coste en Modo Coevolución:** En este modo, el conjunto de la competición consta de 20 individuos (10 padres + 10 hijos), y todos ellos se enfrentan entre sí, además de contra el salón de la fama.

CAPÍTULO 7. SISTEMA DE ENTRENAMIENTO EVOLUTIVO

- *Partidas Round-Robin*: El número de enfrentamientos únicos en un grupo de 20 es $\frac{20 \times 19}{2} = 190$ emparejamientos.
- 190 emparejamientos \times 3 partidas/emparejamiento = **570** partidas.
- *Partidas contra el salón de la fama*: 20 individuos \times 3 miembros del HoF \times 3 partidas/miembro = **180** partidas.
- **Total por generación en modo coevolución: 750 partidas.**

Este pequeño cálculo demuestra que, a igualdad de tamaño de población, el modo de coevolución es aproximadamente el doble de costoso computacionalmente que el modo fijo. Esta diferencia fundamental justifica por qué en el diseño de los experimentos se ajustan parámetros como el tamaño de la población y el número máximo de evaluaciones para cada modo, con el fin de permitir que cada configuración se ejecute durante un tiempo total comparable y así poder realizar una comparación justa de su eficacia.

Este capítulo concluye el objetivo **OG3**, que consistía en desarrollar un programa de optimización en Python para ajustar los pesos de la función de fitness del agente, implementando y comparando dos estrategias principales: algoritmos coevolutivos y entrenamiento supervisado contra agentes de referencia de “Scripts of Tribute”.

Parte III

Experimentación y conclusiones

Capítulo 8

Diseño experimental

Esta tercera parte del TFM, aborda el diseño experimental, los resultados y las conclusiones del proyecto. Una vez descritos los componentes individuales, es decir, el agente autónomo y el sistema de entrenamiento evolutivo, este capítulo se centra en el diseño de los experimentos llevados a cabo para evaluar y comparar las distintas estrategias de optimización. El objetivo de la fase experimental no es solo encontrar el mejor conjunto de pesos posible, sino también entender cómo las diferentes configuraciones del algoritmo evolutivo afectan al proceso de aprendizaje y a la naturaleza de las soluciones encontradas. Para ello, se han diseñado una serie de experimentos que permiten analizar el impacto de factores como el modo de evaluación (fijo, coevolutivo o híbrido) y el uso de un salón de la fama.

8.1. Configuración de los experimentos

El diseño de los experimentos se ha centrado en responder a tres preguntas fundamentales: ¿qué combinación de estrategias en el modo híbrido produce los mejores agentes?, ¿qué modo de entrenamiento genera los mejores bots? y ¿cuál es el impacto del salón de la fama en el rendimiento del entrenamiento? Para abordar estas cuestiones, se crearon scripts de ejecución que automatizan el lanzamiento de varios conjuntos de entrenamientos, cada uno con una configuración específica. Cada experimento se conforma de una serie de entrenamientos, los cuales a su vez realizan 5 carreras diferentes cada uno. Los resultados de estas carreras se promedian para obtener una medida más estadísticamente robusta del rendimiento de cada entrenamiento.

8.1.1. Análisis de configuraciones híbridas

Para explorar en profundidad el modo híbrido, se diseñó un experimento dedicado exclusivamente a probar una veintena de configuraciones distintas de este

CAPÍTULO 8. DISEÑO EXPERIMENTAL

modo. Sus configuraciones varían tanto en el número de segmentos como en la proporción de evaluaciones dedicadas a cada uno de ellos, con el objetivo de encontrar un balance óptimo entre la explotación de conocimiento (evaluación contra oponentes fijos) y la exploración de nuevas estrategias (competición interna). Dado que en este experimento se busca comparar únicamente el efecto de la variación en el parámetro que controla el orden y proporción de los segmentos que conforman el modo híbrido, se mantuvieron constantes los demás parámetros del entrenador. Esta decisión permite aislar las diferencias de rendimiento únicamente al impacto de dicho parámetro, pero también tiene un efecto colateral: el tiempo de entrenamiento de cada configuración es diferente. Dado que el modo híbrido cuenta con diferentes proporciones de evaluaciones fijas y coevolutivas, el tiempo de entrenamiento puede variar significativamente entre cada configuración. Sin embargo, este aspecto se puede considerar como algo positivo, ya que también permite observar el rendimiento de cada configuración en función del tiempo de entrenamiento, pudiendo encontrar configuraciones que no solo sean más efectivas, sino que también sean más eficientes en términos temporales.

La Tabla 8.1 detalla cada una de las 20 configuraciones que se evaluaron. Para cada una, se ha calculado el número total de segmentos, el porcentaje de evaluaciones dedicadas a cada modo (fijo para la explotación y coevolución para la exploración), y una breve descripción de la estrategia que representa. Los parámetros comunes del experimento fueron: 5 carreras de entrenamiento por experimento, una población de 10 individuos, 500 evaluaciones máximas, 200 partidas por evaluación, 8 hilos de paralelismo, y la desactivación tanto del salón de la fama global como del interno para aislar el efecto de la planificación híbrida. Como se han asignado 500 evaluaciones a cada entrenamiento, y cada carrera consta de 10 rondas, esto implica que cada entrenamiento se ejecuta durante un total de 50 generaciones.

Tabla 8.1: Configuraciones de entrenamiento en modo híbrido evaluadas.

ID	Planificación (hybrid_schedule_str)	Seg.	% Fijo	% Coevo
H-1	fixed:0.4,coevolution:0.3,fixed:0.3	3	70%	30%
H-2	fixed:0.7,coevolution:0.3	2	70%	30%
H-3	fixed:0.4,coevolution:0.2,fixed:0.4	3	80%	20%
H-4	fixed:0.5,coevolution:0.5	2	50%	50%
H-5	coevolution:0.6,fixed:0.4	2	40%	60%
H-6	coevolution:0.4,fixed:0.2, coevolution:0.4	3	20%	80%
H-7	f:0.1,c:0.1,f:0.1,c:0.1,f:0.1,c:0.1, f:0.1,c:0.1,f:0.1,c:0.1	10	50%	50%
H-8	fixed:0.2,coevolution:0.2,fixed:0.2, coevolution:0.2,fixed:0.2	5	60%	40%
H-9	fixed:0.8,coevolution:0.2	2	80%	20%
H-10	coevolution:0.8,fixed:0.2	2	20%	80%

8.1. CONFIGURACIÓN DE LOS EXPERIMENTOS

– continuación de la Tabla 8.1–

ID	Planificación (hybrid_schedule_str)	Seg.	% Fijo	% Coevo
H-11	fixed:0.2,coevolution:0.5,fixed:0.3	3	50%	50%
H-12	fixed:0.3,coevolution:0.4,fixed:0.3	3	60%	40%
H-13	coevolution:0.3,fixed:0.4, coevolution:0.3	3	40%	60%
H-14	fixed:0.1,coevolution:0.8,fixed:0.1	3	20%	80%
H-15	coevolution:0.1,fixed:0.8, coevolution:0.1	3	80%	20%
H-16	fixed:0.1,coevolution:0.2,fixed:0.1, coevolution:0.6	4	20%	80%
H-17	coevolution:0.1,fixed:0.2, coevolution:0.1,fixed:0.6	4	80%	20%
H-18	fixed:0.9,coevolution:0.1	2	90%	10%
H-19	coevolution:0.9,fixed:0.1	2	10%	90%
H-20	fixed:0.33,coevolution:0.34,fixed:0.33	3	66%	34%

8.1.2. Análisis de los modos de entrenamiento y el salón de la fama

Para hallar el método que genera los bots con mayor porcentaje de victoria se creó un experimento que ejecuta los tres modos de entrenamiento principales (fijo, coevolutivo e híbrido) dos veces. La primera vez con el salón de la fama activado (con un tamaño de 3 individuos), y la segunda vez completamente desactivado. Por lo tanto, este segundo experimento trata de aportar información en dos frentes, tanto a nivel de modo de entrenamiento como a nivel de salón de la fama. El problema del tiempo de entrenamiento persiste en este experimento, pero acentuado aun más, pues los parámetros entre cada configuración pueden variar significativamente, no solo en el número de evaluaciones, sino también en el número de miembros de la población. Dado que este experimento no trata de comparar un único parámetro, sino configuraciones completamente diferentes, en este caso sí se optó por usar el tiempo como medida de unificación. Así, se ha tratado que las configuraciones se ejecuten durante un tiempo similar, permitiendo una comparación más justa entre las distintas estrategias de entrenamiento.

La Tabla 8.2 resume las 6 configuraciones específicas que se ejecutaron. Se puede observar cómo los parámetros de “Tamaño de Población” y “Evaluaciones Máximas” se ajustaron para cada modo. Para todos los experimentos de esta serie se mantuvieron constantes los siguientes parámetros base: 5 carreras de entrenamiento por experimento, 200 partidas por evaluación y 8 hilos de paralelismo para los procesos de Python.

CAPÍTULO 8. DISEÑO EXPERIMENTAL

Tabla 8.2: Configuraciones para la comparativa de modos de entrenamiento y salón de la fama.

ID	Modo	Población	Eval.	Salón F.	Plan híbrido
E-1	Fijo	50	1900	3	N/A
E-2	Híbrido	10	1000	3	f:0.4,c:0.2,f:0.4
E-3	Coevolución	10	480	3	N/A
E-4	Fijo	50	1900	0	N/A
E-5	Híbrido	10	1000	0	f:0.4,c:0.2,f:0.4
E-6	Coevolución	10	480	0	N/A

8.2. Métricas de evaluación

Para analizar los resultados de estos experimentos de una forma cuantitativa y visual, se desarrolló un script en Python que procesa los ficheros de datos en formato CSV generados por el entrenador. Este script genera un conjunto de gráficas estandarizadas que permiten evaluar y comparar el rendimiento de las diferentes configuraciones a través de varias métricas clave.

Una de las métricas principales es la evolución del *fitness* a lo largo de las generaciones. Se genera una gráfica de líneas que muestra el *fitness* máximo, medio y mínimo, promediado a través de todas las carreras de entrenamiento de un mismo experimento. Esta visualización pretende utilizarse para entender la velocidad de convergencia y si el algoritmo se estanca en óptimos locales.

Otra área de análisis se centra en la convergencia y distribución de los pesos del genoma. Para ello se utilizan dos tipos de gráficas. Primero, un mapa de calor que muestra la evolución del valor promedio de cada uno de los 20 pesos a lo largo de las generaciones. Segundo, un diagrama de barras que presenta el valor medio final de cada peso en la última generación. Estas gráficas permiten analizar si el algoritmo converge hacia soluciones “generalistas”, donde muchos pesos tienen valores moderados, o hacia soluciones “especialistas”, donde unos pocos pesos tienden a valores extremos (cercaos a 0 o 1), indicando que el bot ha aprendido a priorizar unas pocas heurísticas muy específicas. Para medir la consistencia y el rendimiento del campeón final, se utilizan dos enfoques. Por un lado, un diagrama de cajas para cada peso muestra la distribución de los valores finales de los campeones de todas las carreras. El script permite generar todas estas gráficas tanto de los campeones de cada generación como de la media de los pesos de los individuos de cada generación, lo que proporciona una visión desde varios ángulos del proceso de entrenamiento.

Finalmente, la métrica de rendimiento definitiva es el benchmark que se ejecuta al concluir cada entrenamiento. En esta fase, los campeones de cada carrera compiten entre sí para determinar al ganador absoluto, cuya tasa de victorias en este torneo final se considera la medida última del éxito de esa configuración de entrenamiento. Además, de entre los campeones de cada entrenamiento (que ya son los campeones de su conjunto de carreras), se realiza un segundo torneo final para determinar el campeón absoluto de ese experimento. Este torneo final se ejecuta contra un conjunto de bots fijos, que actúan como una referencia estable para evaluar el rendimiento del agente entrenado en un contexto parecido al de la competición real CoG.

8.2. MÉTRICAS DE EVALUACIÓN

Este conjunto de scripts para la generación de gráficas y benchmarking de agentes, cuyos resultados se utilizarán en el siguiente capítulo, demuestran el cumplimiento del objetivo específico **OG4** del proyecto, que se centra en el desarrollo de herramientas para la visualización y análisis de los datos generados durante el entrenamiento.

8.2.1. Bots de referencia

Durante toda esta memoria se ha hecho alusión a los agentes autónomos que se han usado en el modo fijo y en los torneos finales para evaluar el rendimiento de los agentes evolutivos. Estos bots de referencia vienen incluidos en *Scripts of Tribute-Core* y utilizan diferentes algoritmos y heurísticas para cumplir su función. A día de la redacción de este documento, existen 7 bots de referencia más el bot *Random*, que actúa como un agente completamente aleatorio. Lastimosamente, no todos funcionan sin errores. La razón de estos problemas de ejecución es la misma por la que no se han podido utilizar los bots ganadores de competiciones anteriores durante el entrenamiento evolutivo: el motor de simulación SoT se ha actualizado con nuevas funciones y cambios en la API, lo que ha provocado que algunos bots antiguos no sean compatibles con la versión actual del motor. Se contactó con los creadores de SoT para intentar que solucionaran estos problemas de incompatibilidad y, si bien están dispuestos a arreglar los bots en algún momento, la prioridad para ellos es el motor en sí. Por lo tanto, los agentes que sí se han podido utilizar son:

- **PatronFavorsBot**: bot simple que se centra en ganar el favor de los patrones a toda costa. En cada movimiento comprueba si puede ganar el favor de un patrón y, si no es así, realiza un movimiento aleatorio.
- **MaxAgentsBot**: este también es un bot simple, en este caso se centra en maximizar el número de agentes que tiene en el tablero. En cada movimiento, comprueba si puede comprar o colocar un agente, y si no es así, realiza un movimiento aleatorio. Además, para elegir entre los bots a comprar o colocar utiliza una heurística que prioriza las cartas de una mejor calidad.
- **MaxPrestigeBot**: similar a los bots anteriores, este bot se centra en maximizar su puntuación de prestigio. Para cada acción posible, evalúa cual es la que le otorga más puntos de prestigio y la ejecuta. Si no puede realizar ninguna acción que le otorgue puntos, realiza un movimiento aleatorio.
- **DecisionTreeBot**: este es el único bot que no se basa en una heurística simple, sino que utiliza un árbol de decisión diseñado por un experto. Es decir, es un bot con un esquema de funcionamiento completamente opuesto al *EvolutionaryBot* de este proyecto, ya que está diseñado para tomar decisiones basadas en un conjunto de reglas fijas y no evoluciona ni aprende de la experiencia. En lugar de evaluar y puntuar todos los movimientos posibles de forma unificada, el bot sigue una secuencia predefinida: primero intenta jugar las cartas de su mano, si no es posible, considera comprar cartas de la taberna, después activar un patrón, y así sucesivamente hasta encontrar una acción válida. La complejidad de este agente no reside en una función de evaluación ponderada, sino en las heurísticas específicas que gobiernan cada tipo de decisión. Por ejemplo, para decidir qué carta jugar, no elige simplemente la que da más recursos, sino que utiliza una lógica sofisticada que clasifica las cartas en mano según su potencial de combo. En ocasiones, “sacrifica” una carta de bajo potencial para preparar el terreno

CAPÍTULO 8. DISEÑO EXPERIMENTAL

y maximizar los efectos de combo de las cartas que jugará a continuación. De forma similar, para la compra de cartas, utiliza una función heurística que valora la sinergia (favoreciendo cartas de patrones que ya posee) y la calidad general de la carta. Durante la primera fase del proyecto se estudió este bot con detalle, pues su autor claramente tenía un conocimiento profundo del juego y de las estrategias óptimas (algo que no se puede decir del autor de esta memoria). El resultado de esos estudios fueron las heurísticas específicas que el *EvolutionaryBot* utiliza para evaluar la compra y uso de cartas específicas.

Capítulo **9**

Resultados y discusión

- 9.1. Análisis de las configuraciones híbridas**
- 9.2. Análisis de los modos de entrenamiento**
- 9.3. Análisis del salón de la fama**

Conclusiones

10.1. Objetivos alcanzados

10.2. Líneas de trabajo futuro

Bibliografía

- [1] Aaron Garret. Library Reference — inspyred 1.0.3 documentation, 2025. URL: <https://inspyred.readthedocs.io/en/latest/reference.html>.
- [2] Adam Ciężkowski and Artur Krzyżyński. Developing Card Playing Agent for Tales of Tribute AI Competition, 2023. URL: <https://jakubkowalski.tech/Supervising/Ciezkowski2023DevelopingCard.pdf>.
- [3] AI and Games. Why is It Difficult to Make Good AI for Games? | AI 101, January 2024. URL: <https://www.youtube.com/watch?v=qCkqpRnk1oU>.
- [4] CD Projekt Red. The Witcher 4 Unreal Engine 5 Tech Demo 4K | State of Unreal | Unreal Fest Orlando, June 2025. URL: <https://www.youtube.com/watch?v=aorRfK478RE>.
- [5] Ematerasu, Jakub Kowalski, and Radosław Miernik. ScriptsOfTribute · GitHub, 2022. URL: <https://github.com/ScriptsOfTribute>.
- [6] Ematerasu, Jakub Kowalski, and Radosław Miernik. ScriptsOfTribute-CompetitionsArchive, 2024. URL: <https://github.com/ScriptsOfTribute/ScriptsOfTribute-CompetitionsArchive>.
- [7] Ematerasu, Jakub Kowalski, and Radosław Miernik. ScriptsOfTribute-GUI-2.0, June 2025. original-date: 2025-04-14T08:38:44Z. URL: <https://github.com/ScriptsOfTribute/ScriptsOfTribute-GUI-2.0>.
- [8] Universidad Europea. ¿Qué es mesh en el desarrollo de videojuegos? | Blog CC, April 2025. Section: Blog. URL: <https://creativecampus.universidadeuropea.com/blog/meshes/>.
- [9] Epic Games. Behavior Tree in Unreal Engine - Overview | Unreal Engine 5.6 Documentation | Epic Developer Community, 2024. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---overview>.
- [10] Epic Games. ML Deformer Framework in Unreal Engine | Unreal Engine 5.6 Documentation | Epic Developer Community, 2025. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/ml-deformer-framework-in-unreal-engine>.

BIBLIOGRAFÍA

- [11] Pablo García-Sánchez, Alberto Tonda, Antonio J. Fernández-Leiva, and Carlos Cotta. Optimizing Hearthstone agents using an evolutionary algorithm. *Knowledge-Based Systems*, 188:105032, January 2020. URL: <https://www.sciencedirect.com/science/article/pii/S0950705119304356>, doi: [10.1016/j.knosys.2019.105032](https://doi.org/10.1016/j.knosys.2019.105032).
- [12] Aaron Garrett. aarongarrett/inspyred, 2012. original-date: 2012-03-08T04:44:48Z. URL: <https://github.com/aarongarrett/inspyred>.
- [13] Colin Gaudreau and Andreas Lasses. Game AI Summit: No Brakes! Machine Learning Vehicles in 'Star Wars Outlaws', 2025. URL: <https://gdcvault.com/play/1035556/Game-AI-Summit-No-Brakes>.
- [14] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Trans. Graph.*, 32(6):206:1–206:11, November 2013. doi: [10.1145/2508363.2508399](https://doi.org/10.1145/2508363.2508399).
- [15] Majid Ghasemi, Amir Hossein, and Dariush Ebrahimi. Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges, February 2025. URL: <https://arxiv.org/html/2411.18892v2#bib.bib84>.
- [16] Google. gRPC, 2016. URL: <https://grpc.io/>.
- [17] Juan Julián Merelo Guervós. JJ/plantilla-TFG-ETSIIT, September 2024. original-date: 2019-03-19T11:58:46Z. URL: <https://github.com/JJ/plantilla-TFG-ETSIIT>.
- [18] HearthSim. HearthSim/SabberStone, January 2017. original-date: 2017-01-17T08:58:37Z. URL: <https://github.com/HearthSim/SabberStone>.
- [19] Damian Isla. Managing Complexity in the Halo 2 AI System, 2005. URL: <https://www.gdcvault.com/play/1020270/Managing-Complexity-in-the-Halo>.
- [20] Orkin Jeff. GDC Vault - Three States and a Plan: The AI of F.E.A.R., 2006. URL: <https://gdcvault.com/play/1013282/Three-States-and-a-Plan>.
- [21] Magi-Soft Development. Magic Workstation, Cards Management and Decks Testing for CCG Players, Online Play, 2002. URL: <https://www.magicworkstation.com/>.
- [22] Mariela Nogueira, Carlos Cotta, and Antonio J. Fernández-Leiva. An Analysis of Hall-of-Fame Strategies in Competitive Coevolutionary Algorithms for Self-Learning in RTS Games. *ResearchGate*, January 2013. URL: https://www.researchgate.net/publication/290305105_An_Analysis_of_Hall-of-Fame_Strategies_in_Competitive_Coevolutionary_Algorithms_for_Self-Learning_in_RTS_Games, doi: [10.1007/978-3-642-44973-4_19](https://doi.org/10.1007/978-3-642-44973-4_19).
- [23] Mike. Game Programming Concepts–Finite State Machines, June 2016. URL: <https://gamefromscratch.com/game-programming-concepts-finite-state-machines/>.
- [24] Vasileios Moschopoulos, Pantelis Kyriakidis, Aristotelis Lazaridis, and Ioannis Vlahavas. Lucy-SKG: Learning to Play Rocket League Efficiently Using Deep Reinforcement Learning, May 2023. arXiv:2305.15801 [cs]. URL: <http://arxiv.org/abs/2305.15801>, doi: [10.48550/arXiv.2305.15801](https://doi.org/10.48550/arXiv.2305.15801).

BIBLIOGRAFÍA

- [25] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning, December 2019. arXiv:1912.06680 [cs, stat]. URL: <http://arxiv.org/abs/1912.06680>, doi:10.48550/arXiv.1912.06680.
- [26] Pixel. ESO Tales of Tribute Guide, June 2022. URL: <https://deltiasgaming.com/eso-tales-of-tribute-guide/>.
- [27] Raymond Redheffer. A Machine for Playing the Game Nim. *The American Mathematical Monthly*, 55(6):343–349, June 1948. Publisher: Taylor & Francis _eprint: <https://doi.org/10.1080/00029890.1948.11999249>. doi:10.1080/00029890.1948.11999249.
- [28] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Hoboken, April 2020.
- [29] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, December 2017. arXiv:1712.01815 [cs]. URL: <http://arxiv.org/abs/1712.01815>, doi:10.48550/arXiv.1712.01815.
- [30] Simão Reis. VGC AI Framework, October 2019. URL: <https://gitlab.com/DracoStriker/pokemon-vgc-engine>.
- [31] Jacob Snell, Martin Masek, and Chiou Lam. An evolutionary approach to balancing and disrupting real-time strategy games. *MODSIM2021, 24th International Congress on Modelling and Simulation*, January 2021. URL: <https://ro.ecu.edu.au/ecuworkspos2013/11752>, doi:10.36334/modsim.2021.M8.snell.
- [32] Tommy Thompson. AI 101: Introducing Utility AI, February 2024. URL: <https://www.aiandgames.com/p/ai-101-introducing-utility-ai>.
- [33] Tommy Thompson. How AI is Actually Used in the Video Games Industry, April 2025. URL: <https://www.aiandgames.com/p/how-ai-is-actually-used-in-the-video>.
- [34] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, September 2011. URL: <https://ieeexplore.ieee.org/document/5756645>, doi:10.1109/TCIAIG.2011.2148116.
- [35] Jose Torres-Jiménez and Juan Pavón. Applications of metaheuristics in real-life problems. *Progress in Artificial Intelligence*, 2(4):175–176, July 2014. doi:10.1007/s13748-014-0051-8.
- [36] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang,

BIBLIOGRAFÍA

- Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019. Publisher: Nature Publishing Group. URL: <https://www.nature.com/articles/s41586-019-1724-z>. doi:10.1038/s41586-019-1724-z.
- [37] Wikipedia. Artificial intelligence in video games, May 2025. Page Version ID: 1292216335. URL: https://en.wikipedia.org/w/index.php?title=Artificial_intelligence_in_video_games&oldid=1292216335.
- [38] Wikipedia. Diseño de juegos, May 2025. Page Version ID: 167396849. URL: https://es.wikipedia.org/w/index.php?title=Dise%C3%B1o_de_juegos&oldid=167396849.
- [39] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. URL: <https://ieeexplore.ieee.org/document/585893>, doi:10.1109/4235.585893.
- [40] Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Piyush Khandelwal, Varun Kompella, Hao-Chih Lin, Patrick MacAlpine, Declan Oller, Takuma Seno, Craig Sherstan, Michael D. Thomure, Houmehr Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Dürr, Peter Stone, Michael Spranger, and Hiroaki Kitano. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228, February 2022. Publisher: Nature Publishing Group. URL: <https://www.nature.com/articles/s41586-021-04357-7>, doi:10.1038/s41586-021-04357-7.
- [41] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms, August 2018. arXiv:1808.04794 [cs]. URL: <http://arxiv.org/abs/1808.04794>, doi:10.48550/arXiv.1808.04794.