



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Aplicación de técnicas de inteligencia artificial en juegos de estrategia

Grado en Ingeniería Mecánica

ALUMNO: Francisco Carrasco García-Almenta

DIRECTORES: Milagros Fernández Gavilanes
Norberto Fernández García

CURSO ACADÉMICO: 2024-2025

Universida_{de}Vigo



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Aplicación de técnicas de inteligencia artificial en juegos de estrategia

Grado en Ingeniería Mecánica
Intensificación en Tecnología Naval
Cuerpo General

UniversidadeVigo

RESUMEN

Este Trabajo de Fin de Grado explora la aplicación de técnicas de inteligencia artificial en juegos de estrategia, con un enfoque en el desarrollo de una IA para el juego Hundir la Flota. Se parte de una implementación básica del juego y se introducen progresivamente mejoras en la estrategia de la IA, desde enfoques aleatorios hasta técnicas avanzadas de optimización.

Inicialmente, la IA juega de forma completamente aleatoria. Posteriormente, se incorporan estrategias heurísticas, como la búsqueda dirigida tras un impacto y patrones de disparo optimizados. Además, se estudian estrategias de colocación de barcos, incluyendo distribuciones optimizadas para aumentar la dificultad de detección.

Para mejorar aún más el rendimiento de la IA, se implementan modelos basados en probabilidad y memoria, permitiendo que el agente recuerde información clave entre disparos y partidas. Finalmente, se introduce aprendizaje por refuerzo, entrenando un agente capaz de mejorar su desempeño a medida que juega más partidas, optimizando su toma de decisiones de manera autónoma.

Los resultados se evalúan mediante simulaciones, comparando la efectividad de cada estrategia y analizando el impacto del aprendizaje automático en el rendimiento del juego. Este estudio demuestra el potencial de la inteligencia artificial en la optimización de estrategias dentro de juegos de estrategia.

PALABRAS CLAVE

Inteligencia artificial, estrategia, táctica y hundir la flota.

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer todo el apoyo de mi familia y amigos, no sólo durante la realización del trabajo, sino durante los 5 años de formación en la Escuela Naval Militar. Donde gracias a ellos he podido encontrar la motivación necesaria para poder cumplir mis objetivos.

Ayudándome así a cumplir un sueño y el primer hito importante de mi carrera en la Armada.

Además, quiero agradecer a mis tutores, Milagros y Norberto, por su paciencia y consejo durante estos meses, y sobre todo por el tiempo y atención que me han dedicado durante el desarrollo del trabajo y gracias al cual he podido.

Por último, agradecer a la Armada, y a toda la gente que la conforma, todo el tiempo y recursos invertidos en mi formación y por inculcarme unos valores que serán un pilar fundamental durante el resto de mi vida.

CONTENIDO

Contenido	1
Índice de Figuras	4
Índice de Tablas.....	6
1 Introducción y objetivos	7
1.1 Contexto y motivación	7
1.2 Objetivos	8
1.3 Estructura de la memoria	9
2 Estado del arte	10
2.1 La inteligencia artificial	10
2.1.1 Definición	10
2.1.2 Contexto histórico.....	10
2.2 Aprendizaje automático	11
2.3 Aprendizaje profundo	13
2.4 Aprendizaje por refuerzo	14
2.4.1 Fundamentos del aprendizaje por refuerzo	14
2.4.2 Fundamentos del aprendizaje por refuerzo profundo	15
2.4.3 Principales algoritmos de DRL.....	16
2.4.4 Aplicaciones	16
2.5 Juegos de estrategia.....	16
2.5.1 AlphaGo.....	17
2.5.1.1 Fundamentos técnicos.....	18
2.5.1.2 Impacto de AlphaGo en la IA y más allá.....	18
2.5.2 AlphaStar	19
2.5.2.1 Principales diferencias entre AlphaGo y AlphaStar	19
2.5.2.2 Fundamentos técnicos.....	20
2.5.2.3 Impacto de AlphaStar en la IA y más allá	20
3 Desarrollo del TFG.....	22
3.1 Implementación del juego de Hundir la Flota.....	23
3.1.1 Inicialización del juego.....	24
3.1.2 Lógica de turnos y mecánica de disparos	27
3.2 Estrategias básicas de la IA.....	29
3.2.1 Estrategia aleatoria: primeros pasos	30
3.2.2 Implementación de estrategias heurísticas.....	30
3.2.3 Optimización de disparos en función de impactos	35

3.3 Mejoras avanzadas en la IA	38
3.3.1 Estrategias probabilísticas para encontrar barcos	38
3.3.2 Memoria entre partidas para optimización de disparos	40
3.4 Implementación del aprendizaje por refuerzo (RL)	43
3.4.1 Implementación del entorno de aprendizaje	44
3.4.2 Definición del agente	45
3.4.3 Entrenamiento del agente	46
4 Resultados	50
4.1 Explicación general	50
4.2 Colocación de barcos con disposiciones predefinidas	50
4.2.1 Código de colocación de barcos con disposiciones predefinidas	51
4.2.2 Impacto en la simulación de partidas	52
4.3 Registro del número de turnos por partida en las simulaciones	52
4.3.1 Modificaciones en el código	53
4.4 Resultados obtenidos	53
4.4.1 Tasa de victorias	53
4.4.2 Análisis de turnos por partida	56
4.4.3 Evaluación del rendimiento de las estrategias	63
4.5 Resultados del agente con RL	71
4.5.1 Puntuación del agente a lo largo de los episodios	71
4.5.2 Comparación con otras estrategias	72
5 Conclusiones y líneas futuras	74
5.1 Líneas futuras	74
6 Bibliografía	76
Anexo I: Implicaciones Ambientales	79
Anexo II: Reflexiones Éticas y Sociales	80
Anexo III: Código de la estrategia aleatoria	81
A3.1 Aleatorio vs Aleatorio	81
A3.2 Aleatorio vs Humano	82
Anexo IV: Código estrategia de caza	86
A4.1 Estrategia de caza vs Aleatorio	86
A4.2 Estrategia de caza vs Humano	88
Anexo V: Código de patrón de búsqueda	92
A5.1 Patrón de búsqueda vs Aleatorio	92
A5.2 Patrón de búsqueda vs Humano	94

Anexo VI: Código de búsqueda por cuadrantes	98
A6.1 Búsqueda por cuadrantes vs Aleatorio.....	98
A6.2 Búsqueda por cuadrantes vs Humano	100
Anexo VII: Código de búsqueda en zigzag.....	104
A7.1 Búsqueda en zigzag vs Aleatorio	104
A7.2 Búsqueda en zigzag vs Humano	106
Anexo VIII: Código de colocación espaciada	110
A8.1 Colocación espaciada vs Aleatorio	110
A8.2 Colocación espaciada vs Humano.....	112
Anexo IX: Código de colocación en los bordes	116
A9.1 Colocación en los bordes vs Aleatorio.....	116
A9.2 Colocación en los bordes vs Humano	118
Anexo X: Código de optimización de búsqueda tras impacto	122
A10.1 Optimización de búsqueda tras impacto vs Aleatorio.....	122
A10.2 Optimización de búsqueda tras impacto vs Humano	124
Anexo XI: Código de memoria de barcos restantes	128
A11.1 Memoria de barcos restantes vs Aleatorio	128
A11.2 Memoria de barcos restantes vs Humano	130
Anexo XII: Código de estrategia probabilística	134
A12.1 Estrategia probabilística vs Aleatorio	134
A12.2 Estrategia probabilística vs Humano.....	137
Anexo XIII: Código de memoria entre partidas	142
A13.1 Memoria entre partidas vs Aleatorio.....	142
A13.2 Memoria entre partidas vs Humano	145
Anexo XIV: Código de RL	150
A14.1 Código de entrenamiento de RL	150

ÍNDICE DE FIGURAS

Figura 2-1 Conferencia de Dartmouth [11].	11
Figura 2-2 Esquema básico de aprendizaje supervisado [15].	12
Figura 2-3 Esquema básico de aprendizaje no supervisado [15].	12
Figura 2-4 Esquema básico del funcionamiento de las redes neuronales [16].	13
Figura 2-5 Esquema básico de aprendizaje por refuerzo [15].	14
Figura 2-6 Juego de Go [31].	17
Figura 2-7 Partida de <i>Go</i> entre <i>AlphaGo</i> y el campeón del mundo [32].	17
Figura 2-8 AlphaStar en partida 1v1 [40].	19
Figura 3-1 Metodología seguida en el proceso de desarrollo de la IA.	22
Figura 3-2 Código de inicialización del tablero.	24
Figura 3-3 Código de impresión del tablero.	25
Figura 3-4 Código de definición de los barcos.	25
Figura 3-5 Código para la colocación de los barcos manualmente.	26
Figura 3-6 Código de la colocación aleatoria de los barcos.	27
Figura 3-7 Código que define el disparo aleatorio.	28
Figura 3-8 Código de comprobación de barcos en el tablero.	28
Figura 3-9 Código que inicia la simulación de las partidas.	29
Figura 3-10 Código de disparo con estrategia de caza.	31
Figura 3-11 Código de generación de cuadrantes.	32
Figura 3-12 Código generador del patrón zigzag.	32
Figura 3-13 Código de generación de patrón de búsqueda.	33
Figura 3-14 Código de disparo con patrón de búsqueda.	33
Figura 3-15 Código de colocación de barcos con separación.	34
Figura 3-16 Código de colocación de barcos en los bordes.	35
Figura 3-17 Código de mejora de la estrategia de caza.	36
Figura 3-18 Código de identificación de los barcos restantes.	37
Figura 3-19 Código de comparación de tamaño entre barcos restantes y hueco en el tablero.	37
Figura 3-20 Código de creación del mapa de probabilidad.	39
Figura 3-21 Código de disparo con probabilidad.	40
Figura 3-22 Código que guarda la posición de barcos entre partidas.	41
Figura 3-23 Código para actualizar el registro de la memoria de la colocación de los barcos.	41
Figura 3-24 Código de cálculo de probabilidad contando con el factor de la memoria.	42
Figura 3-25 Código para simular la partida y actualizar la memoria.	43
Figura 3-26 Código de creación del ambiente para el agente RL.	44
Figura 3-27 Código de la función que lleva la puntuación del agente.	45

Figura 3-28 Código para resetear el tablero.	45
Figura 3-29 Código de la red neuronal del agente RL.	46
Figura 3-30 Código que ajusta los valores de Q.....	47
Figura 3-31 Código que extrae experiencias aleatorias.....	48
Figura 3-32 Código que calcula la predicción del valor Q.....	48
Figura 3-33 Código que aplica la ecuación de Bellman.....	48
Figura 3-34 Código que ajusta los valores Q.	48
Figura 3-35 Código de actualización de <i>épsilon</i>	49
Figura 3-36 Código de simulación de episodios.	49
Figura 3-37 Código que guarda cada 50 episodios.	49
Figura 4-1 Código de colocación preestablecidas.....	51
Figura 4-2 Disposiciones preestablecidas	51
Figura 4-3 Código que verifica que no hay error colocando los barcos.....	51
Figura 4-4 Modificación en la función para simular partida.....	52
Figura 4-5 Código para crear el histograma de los turnos en los que finaliza la partida.	53
Figura 4-6 Histograma de estrategia aleatoria.....	57
Figura 4-7 Histograma de estrategia de caza.....	58
Figura 4-8 Histograma de patrón de búsqueda.....	58
Figura 4-9 Histograma de búsqueda por cuadrante.....	59
Figura 4-10 Histograma de búsqueda en zigzag.....	59
Figura 4-11 Histograma de colocación espaciada.	60
Figura 4-12 Histograma de colocación en los bordes.	60
Figura 4-13 Histograma de optimización de búsqueda tras impacto.	61
Figura 4-14 Histograma de memoria de barcos restantes.	61
Figura 4-15 Histograma de estrategia probabilística.....	62
Figura 4-16 Histograma de memoria entre partidas.	62

ÍNDICE DE TABLAS

Tabla 2-1 Algoritmos más usados en DRL	16
Tabla 2-2 Diferencias entre AlphaGo y AlphaStar	20
Tabla 4-1 Tabla de resultados de las estrategias contra el oponente aleatorio.	54
Tabla 4-2 Tabla de resultados de las estrategias contra el oponente humano.	54
Tabla 4-3 Resultados de turnos por partida.	57
Tabla 4-4 Evolución del agente durante la simulación.	71

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Contexto y motivación

Los videojuegos han evolucionado de manera constante desde sus inicios, pasando de sistemas con reglas predefinidas a entornos complejos con gráficos realistas e interacciones avanzadas. A medida que la industria ha crecido, también lo ha hecho la necesidad de ofrecer experiencias más inmersivas y desafiantes, impulsando el desarrollo de tecnologías innovadoras.

Uno de los primeros hitos en la inteligencia artificial (IA) aplicada a juegos ocurrió en 1951 con el desarrollo de Nimrod, una computadora diseñada para jugar al juego de Nim, presentada en el Festival de Gran Bretaña [1]. Posteriormente, en 1956, Arthur Samuel creó un programa de damas capaz de mejorar su desempeño con la práctica, sentando las bases del aprendizaje automático [2]. Sin embargo, el verdadero salto en la IA de juegos llegó con el ajedrez: en 1997, *Deep Blue*, desarrollado por IBM, logró vencer al campeón mundial Garry Kasparov, demostrando el potencial de los algoritmos avanzados en juegos de estrategia [3].

Un avance aún más significativo ocurrió en 2016 con *AlphaGo*, desarrollado por DeepMind [4], que marcó un antes y un después en la aplicación de la IA en los juegos. A diferencia de *Deep Blue*, basado en fuerza bruta y evaluaciones preprogramadas, *AlphaGo* utilizó aprendizaje profundo y aprendizaje por refuerzo, lo que le permitió desarrollar estrategias superiores y vencer a los mejores jugadores humanos de Go, un juego reconocido por su enorme espacio de decisiones y profundidad estratégica.

El éxito de *AlphaGo* puso de manifiesto el enorme potencial de la IA y, más concretamente, el *deep learning* [5]. Estas tecnologías han permitido no solo crear adversarios más inteligentes en los videojuegos, sino también diseñar sistemas adaptativos que pueden aprender y evolucionar en tiempo real. Algoritmos como el aprendizaje por refuerzo y redes neuronales profundas han sido clave para lograr sistemas capaces de tomar decisiones complejas y ofrecer interacciones dinámicas, llevando los videojuegos a un nivel completamente nuevo.

Más allá del entretenimiento, las tecnologías empleadas en los videojuegos tienen un enorme potencial para ser adaptadas a otros campos, como el de la defensa. En este contexto, los simuladores desempeñan un papel crucial para el entrenamiento de personal militar, la planificación de estrategias y la evaluación de escenarios complejos. Esta importancia se ve reflejada en la Escuela Naval Militar, donde existen dos tipos de simuladores: el Simulador Táctico (SIMTAC) y el Simulador de Navegación (SIMNAV). Incorporar aprendizaje automático en estos simuladores podría permitir que los sistemas no solo repliquen escenarios predefinidos, sino que también aprendan y se adapten en tiempo real a las acciones de los usuarios, creando entornos más realistas y efectivos.

El Ministerio de Defensa de España, consciente de los avances tecnológicos globales y los retos del contexto actual, ha establecido como una prioridad estratégica el fortalecimiento de las capacidades tecnológicas a través de la inversión en investigación, desarrollo e innovación (I+D+i). Según la Estrategia de Tecnología e Innovación para la Defensa (ETID), esta política busca dotar a las Fuerzas Armadas de herramientas avanzadas que les permitan enfrentar los desafíos contemporáneos con eficacia y rapidez [6].

Entre las líneas de I+D+i destacadas, la simulación ocupa un lugar central. Los simuladores son herramientas esenciales para el entrenamiento de personal militar, la planificación de estrategias y la evaluación de escenarios operativos complejos. Estas plataformas permiten recrear entornos realistas y desafiantes donde los usuarios pueden entrenar y probar tácticas sin los riesgos y costos asociados a ejercicios reales. Incorporar tecnologías como la IA y el aprendizaje automático en estos simuladores eleva su potencial, al permitir que los sistemas se adapten dinámicamente a las acciones de los usuarios, emulando situaciones cambiantes y realistas.

La motivación detrás de este trabajo radica en explorar cómo estas tecnologías, que han revolucionado el entretenimiento, pueden aplicarse a otros ámbitos, como el de la defensa. Los simuladores son una herramienta esencial para el entrenamiento, la planificación y la evaluación de estrategias en contextos complejos. Incorporar IA y técnicas de aprendizaje profundo en estos sistemas podría transformar su capacidad de adaptarse en tiempo real a las acciones de los usuarios, creando entornos más efectivos, realistas y personalizados. Este enfoque no solo tiene el potencial de mejorar el rendimiento en simulaciones militares, sino también de abrir nuevas posibilidades en otras áreas donde los sistemas adaptativos sean cruciales.

1.2 Objetivos

Este trabajo aborda el desarrollo del juego *Hundir la Flota*, explorando su progresión desde la utilización de estrategias básicas hasta métodos avanzados de optimización y aprendizaje automático. La implementación sigue una evolución progresiva, incorporando mejoras estratégicas en la IA antes de llegar a modelos de autoaprendizaje.

El objetivo principal es diseñar y optimizar las estrategias para que mejoren su desempeño mediante la aplicación de distintas técnicas de IA, evaluando su rendimiento en escenarios de juego progresivamente más desafiantes. Para ello, se han implementado estrategias heurísticas, optimización basada en memoria y modelos probabilísticos, antes de explorar el uso de aprendizaje por refuerzo para lograr una IA más adaptativa y eficiente.

En cuanto a los objetivos específicos del presente trabajo, son los siguientes:

1. **Desarrollar la base del juego Hundir la Flota**, implementando sus reglas y mecánicas para servir como entorno de prueba en el diseño y evaluación de estrategias de IA.
2. **Desarrollar y evaluar estrategias heurísticas y de optimización**, incluyendo patrones de disparo inteligentes, estrategias de colocación de barcos y métodos de búsqueda más eficientes.
3. **Incorporar técnicas avanzadas como el uso de memoria y probabilidad**, permitiendo que la IA recuerde información relevante entre disparos e identifique patrones estratégicos para mejorar su desempeño.
4. **Analizar el impacto de cada estrategia en la eficiencia y su tasa de victorias**, comparando sus resultados contra diferentes tipos de oponentes, desde jugadores aleatorios hasta modelos con patrones predefinidos.
5. **Implementar un algoritmo de aprendizaje por refuerzo**, utilizando un enfoque basado en la experiencia previa para que la IA pueda mejorar de forma autónoma tras múltiples partidas.

6. **Evaluar la efectividad del aprendizaje automático en la jugabilidad**, comparando el rendimiento de la IA entrenada con modelos tradicionales basados en reglas estáticas o estrategias heurísticas.

Este trabajo no solo tiene como objetivo explorar la aplicabilidad de distintas estrategias de IA en un entorno de juego, sino también analizar el impacto progresivo de cada mejora implementada en la IA. Se pretende desarrollar enfoques que vayan desde estrategias heurísticas y probabilísticas hasta técnicas basadas en aprendizaje automático. Además, los resultados obtenidos servirán como base para futuras investigaciones en la optimización de agentes inteligentes en juegos de estrategia y simulaciones más complejas.

1.3 Estructura de la memoria

Tras haber definido el contexto y los objetivos de este trabajo, se presenta la estructura de la memoria con el objetivo de organizar de manera clara la exposición de los desarrollos realizados, las estrategias implementadas y los resultados obtenidos. La memoria se ha estructurado en cinco capítulos, seguidos de la bibliografía y anexos, que proporcionan información adicional para complementar la comprensión del trabajo. La organización es la siguiente:

- **Capítulo 1: Introducción y objetivos:** en este capítulo se presenta el contexto del trabajo, explicando la importancia de la IA en los juegos de estrategia y su aplicabilidad a otros campos. También se detallan los objetivos del proyecto y se justifica la elección del juego *Hundir la Flota* como entorno de estudio.
- **Capítulo 2: Estado del arte:** se realiza una revisión teórica de los conceptos fundamentales en los que se basa este trabajo. Se introducen los diferentes enfoques de IA aplicados a juegos, incluyendo el aprendizaje automático, el aprendizaje por refuerzo y sus aplicaciones en sistemas estratégicos. Se analizan referentes destacados como *AlphaGo* y *AlphaStar*, resaltando los avances en IA y su impacto en la optimización de estrategias.
- **Capítulo 3: Desarrollo del TFG:** este capítulo constituye el núcleo del trabajo, describiendo la implementación del juego *Hundir la Flota*. Se inicia con la programación del juego base, estableciendo las reglas y mecánicas fundamentales. A partir de ahí, se detallan las distintas estrategias de la IA, desde enfoques heurísticos hasta técnicas más avanzadas de optimización y toma de decisiones probabilísticas. Finalmente, se describe la integración del aprendizaje por refuerzo, permitiendo que la IA mejore su desempeño de manera autónoma.
- **Capítulo 4: Resultados y validación:** en esta sección se presentan los resultados obtenidos a partir de la ejecución de múltiples simulaciones, evaluando la eficacia de cada estrategia implementada. Se analizan métricas clave como la tasa de victorias y el número de turnos necesarios para completar una partida, comparando el rendimiento de las diferentes versiones de la IA. Además, se incluyen gráficos y tablas que permiten visualizar y contrastar los datos obtenidos.
- **Capítulo 5: Conclusiones y líneas futuras:** en este capítulo se resumen las conclusiones del trabajo, destacando los logros alcanzados y las mejoras introducidas en la IA del juego. Se discuten las limitaciones del enfoque actual y se proponen posibles mejoras para trabajos futuros, incluyendo optimizaciones en el entrenamiento mediante aprendizaje por refuerzo y su aplicación a juegos de mayor complejidad.

Finalmente, la memoria se complementa con la bibliografía, donde se citan todas las fuentes utilizadas, y los anexos, que contienen información adicional relevante para la comprensión del desarrollo del proyecto.

2 ESTADO DEL ARTE

El presente capítulo tratará de estudiar en qué consiste la IA y el desarrollo de esta tecnología, la cual ha revolucionado diversos ámbitos, entre ellos los videojuegos. Para ello, se presentará un análisis de las diferentes técnicas de *Machine Learning*, abordando los diferentes tipos de aprendizaje, como el aprendizaje supervisado, no supervisado y por refuerzo. Además, se expondrán los algoritmos más destacados de estas técnicas, haciendo énfasis en los avances recientes relacionados con el aprendizaje profundo (DL, *Deep Learning*) y su impacto en la industria de los videojuegos y simuladores. Como son los casos de *AlphaGo* y *StarCraft* que han demostrado la capacidad de las máquinas para aprender y superar a jugadores humanos en entornos altamente complejos.

2.1 La inteligencia artificial

2.1.1 Definición

La IA es una disciplina de la informática que se centra en el diseño y desarrollo de sistemas capaces de realizar tareas que requieren inteligencia humana, como el aprendizaje, la razón, la percepción y la toma de decisiones. Esta área multidisciplinaria combina conceptos de matemáticas, estadística, computación y neurociencia para construir máquinas que puedan simular comportamientos inteligentes [7].

De acuerdo con John McCarthy, considerado uno de los padres de la IA, esta disciplina puede definirse como "*la ciencia e ingeniería de crear máquinas inteligentes*" [8]. En esencia, la IA busca dotar a las máquinas de la capacidad de pensar y actuar de manera racional, utilizando algoritmos y modelos matemáticos para analizar datos, identificar patrones y tomar decisiones basadas en esa información.

2.1.2 Contexto histórico

La historia de la IA puede rastrearse hasta mediados del siglo XX, cuando matemáticos, lógicos y científicos comenzaron a conceptualizar la posibilidad de máquinas pensantes. Uno de los primeros hitos importantes fue el trabajo de Alan Turing en la década de 1930 y 1940. En su artículo de 1950 titulado "*Computing Machinery and Intelligence*", Turing planteó la famosa pregunta: "*¿Pueden las máquinas pensar?*" [9] y propuso el conocido "Test de Turing" como una forma de determinar si una máquina exhibía comportamiento inteligente indistinguible del humano.

En 1956, el campo de la IA fue formalmente definido durante la conferencia de Dartmouth, organizada por John McCarthy, Marvin Minsky, Nathaniel Rochester y Claude Shannon. Este evento marcó el comienzo oficial de la IA como disciplina académica y condujo a una oleada de investigación en el diseño de algoritmos capaces de simular el razonamiento humano [10].



Figura 2-1 Conferencia de Dartmouth [11].

Durante las décadas de 1950 y 1960, los investigadores centraron sus esfuerzos en sistemas basados en reglas y lógica simbólica. Estas primeras aproximaciones a la IA se conocían como "IA simbólica" o "IA basada en reglas" [12], ya que consistían en codificar manualmente conocimientos humanos en forma de reglas lógicas. En este sentido, los sistemas simbólicos lograron algunos éxitos tempranos en áreas como la resolución de problemas matemáticos y los juegos. Por ejemplo, en 1956, el programa *Logic Theorist* de Allen Newell y Herbert Simon fue capaz de demostrar teoremas matemáticos. Más adelante, en 1962, IBM desarrolló un programa que jugaba al ajedrez [13], considerado un gran avance en la simulación del pensamiento humano.

Sin embargo, estos sistemas tenían limitaciones significativas. Dependían en gran medida de conocimientos predefinidos y carecían de la capacidad de adaptarse a nuevas situaciones. A medida que los problemas que intentaban resolver se volvían más complejos, las reglas requeridas crecían exponencialmente, lo que hizo evidente la necesidad de enfoques más avanzados.

El entusiasmo inicial por la IA llevó a expectativas poco realistas sobre lo que la tecnología podía lograr en un corto período de tiempo. Esto resultó en lo que se denominó el "invierno de la IA" [12], es decir, una década de estancamiento en la financiación y la investigación debido a los resultados decepcionantes.

Durante este tiempo, los sistemas existentes demostraron ser incapaces de resolver problemas del mundo real de manera eficiente. La IA basada en reglas carecía de la flexibilidad necesaria para manejar entornos complejos y cambiantes. A pesar de ello, algunos avances importantes se realizaron en esta época, incluyendo el desarrollo de los primeros sistemas expertos, que utilizaron bases de conocimiento para realizar tareas específicas, como diagnósticos médicos (por ejemplo, el sistema MYCIN) [12].

2.2 Aprendizaje automático

La década de 1980 marcó un renacimiento en la investigación de la IA con la llegada del aprendizaje automático (ML, *Machine Learning*) [14] y el resurgimiento del interés en las redes neuronales artificiales. El enfoque del ML se centró en permitir que las máquinas aprendieran de los datos en lugar de depender exclusivamente de reglas programadas manualmente.

Este avance permitió el desarrollo de tres enfoques fundamentales en el aprendizaje automático: el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo. En el aprendizaje supervisado, los modelos se entrenan con datos etiquetados, es decir, cada entrada tiene una salida conocida que el algoritmo debe aprender a predecir (como se muestra en la Figura 2-2). Este enfoque es ampliamente utilizado en tareas como clasificación, donde el objetivo es asignar una categoría a una entrada (por ejemplo, detección de spam en correos electrónicos), y regresión, que busca predecir valores continuos (como el precio de una vivienda en función de sus características). Para lograrlo, se emplean

algoritmos como la regresión lineal, máquinas de soporte vectorial (SVM) y redes neuronales artificiales (ANN) [15].

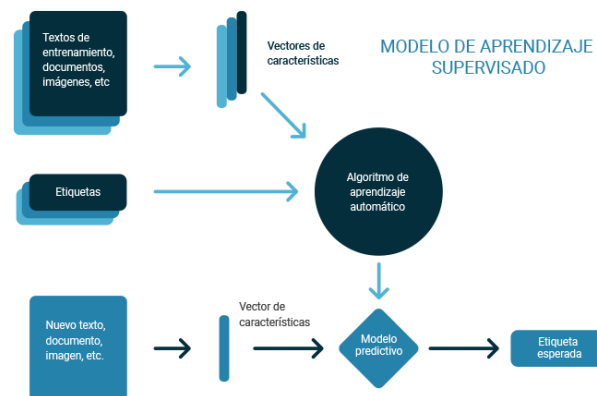


Figura 2-2 Esquema básico de aprendizaje supervisado [15].

En contraste, el aprendizaje no supervisado se aplica cuando los datos no tienen etiquetas y el algoritmo debe identificar patrones o estructuras ocultas en la información (véase la Figura 2-3). Este enfoque es especialmente útil en *clustering*, donde los datos se agrupan según similitudes (como la segmentación de clientes en marketing), y en reducción de dimensionalidad, optimizando la representación de datos complejos (por ejemplo, mediante el análisis de componentes principales, PCA). Algunos de los algoritmos más utilizados en *clustering* incluyen K-Means y modelos de mezcla de Gaussianas, mientras que para la reducción de dimensionalidad se emplean métodos como PCA y autoencoders. Mientras que el aprendizaje supervisado permite hacer predicciones precisas basadas en datos históricos, el aprendizaje no supervisado se centra en descubrir relaciones y estructuras sin intervención humana, lo que ha ampliado significativamente las aplicaciones de la IA en distintos campos.

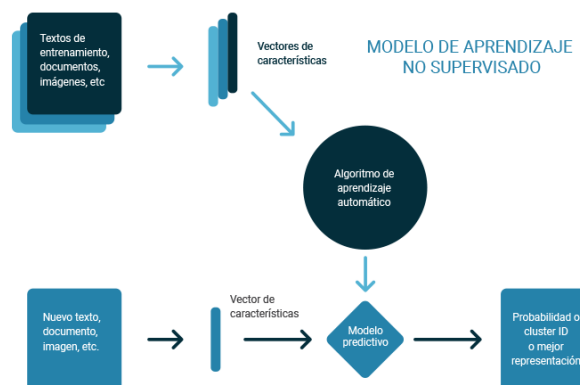


Figura 2-3 Esquema básico de aprendizaje no supervisado [15].

Además de los avances en el aprendizaje supervisado y no supervisado, la década de 1980 también marcó el resurgimiento del interés en las redes neuronales artificiales, un modelo inspirado en la estructura del cerebro humano que permite a las máquinas procesar información de manera distribuida y aprender patrones complejos. Una red neuronal está compuesta por neuronas artificiales organizadas en capas (como se puede ver en la Figura 2-4): la capa de entrada, que recibe los datos; las capas ocultas, que procesan la información a través de conexiones ponderadas y funciones de activación; y la capa de salida, que genera la predicción final. Inicialmente, las redes neuronales enfrentaron limitaciones debido a la falta de métodos eficientes para ajustar sus pesos, lo que llevó a su declive durante el anteriormente mencionado, "invierno de la IA". Sin embargo, en la década de 1980, el desarrollo del algoritmo de retro

propagación del error permitió entrenar redes profundas ajustando sus parámetros mediante gradientes descendentes, lo que revitalizó la investigación en este campo.

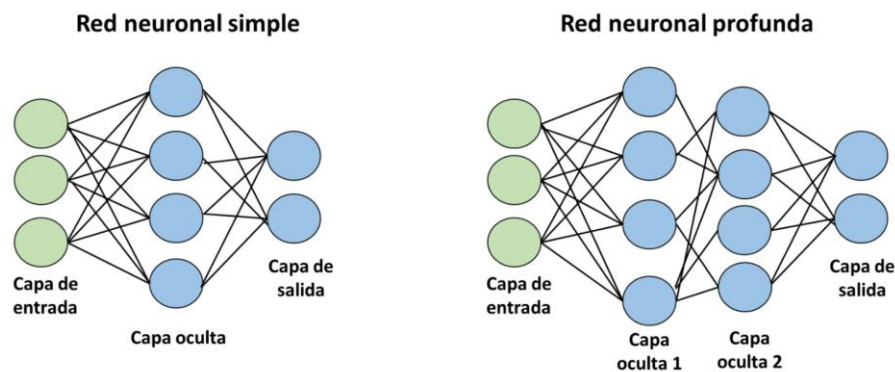


Figura 2-4 Esquema básico del funcionamiento de las redes neuronales [16].

2.3 Aprendizaje profundo

Con el tiempo, las técnicas de aprendizaje automático evolucionaron hacia el DL, un enfoque basado en redes neuronales profundas (DNNs) con múltiples capas (véase la Figura 2-4). Estas arquitecturas han demostrado un rendimiento sobresaliente en tareas como el reconocimiento de imágenes, el procesamiento del lenguaje natural y los juegos de estrategia [17]. Modelos avanzados, como las redes neuronales convolucionales (CNNs), especializadas en el análisis de imágenes, y las redes neuronales recurrentes (RNNs), diseñadas para procesar secuencias de datos como texto y audio, han superado enfoques tradicionales en numerosos campos [18]. Esto ha consolidado el DL como una de las herramientas más poderosas dentro del aprendizaje automático.

El concepto de redes neuronales ha evolucionado desde sus primeras formulaciones en los años 40 con McCulloch y Pitts y el perceptrón de Rosenblatt en los años 50. Como se mencionó en la sección de aprendizaje automático, el verdadero auge del DL ocurrió cuando la retro propagación del error permitió entrenar redes neuronales multicapa de manera eficiente en los años 80. Sin embargo, el avance más significativo se dio en la última década gracias a mejoras en hardware, acceso a grandes volúmenes de datos y optimización de algoritmos [18].

Este resurgimiento se debe principalmente a tres factores clave:

1. Mayor poder computacional: la evolución de las GPUs y el desarrollo de hardware especializado permitieron entrenar modelos con millones de parámetros en tiempos razonables [17].
2. Disponibilidad de grandes volúmenes de datos: la proliferación de internet, redes sociales y el almacenamiento masivo facilitaron el acceso a conjuntos de datos extensos y diversos, esenciales para entrenar modelos profundos [18].
3. Avances en algoritmos y optimización: técnicas como la normalización por lotes (*batch normalization*) y optimizadores avanzados como Adam mejoraron la estabilidad y eficiencia del entrenamiento [18].

Estos avances han convertido al DL en una tecnología clave para múltiples aplicaciones. En el contexto de este trabajo, su impacto se ha visto reflejado especialmente en el ámbito de los videojuegos y simulaciones, donde permite el desarrollo de personajes no jugables (NPCs, Non-Playable Characters) con comportamiento inteligente y la generación de entornos dinámicos. La combinación del DL con aprendizaje por refuerzo ha llevado al desarrollo de sistemas avanzados como AlphaGo y agentes en StarCraft II, que exploraremos en apartados posteriores [5].

El impacto del DL en los videojuegos ha sido significativo. Gracias a estas técnicas, se han desarrollado agentes que pueden aprender de manera autónoma, mejorando su rendimiento en tiempo real. Modelos como *AlphaGo* han demostrado el potencial del aprendizaje profundo en juegos de estrategia, mientras que, en el ámbito militar, las simulaciones han integrado IA para mejorar el realismo y la adaptabilidad de los escenarios [5].

Si bien el DL ha logrado avances espectaculares en el reconocimiento de patrones y procesamiento de datos, su combinación con el aprendizaje por refuerzo (RL, *Reinforcement Learning*) ha abierto nuevas posibilidades en la toma de decisiones autónomas. Este enfoque, conocido como aprendizaje por refuerzo profundo (DRL, *Deep Reinforcement Learning*), ha permitido que los agentes aprendan estrategias óptimas en entornos complejos y dinámicos [19].

En el siguiente apartado, exploraremos en detalle el RL, sus fundamentos y cómo se integra con el DL para desarrollar sistemas de IA más avanzados y adaptativos.

2.4 Aprendizaje por refuerzo

El RL es un paradigma de aprendizaje automático basado en la interacción de un agente con un entorno. A diferencia del aprendizaje supervisado, en el que un modelo se entrena con ejemplos etiquetados, en el RL un agente aprende mediante prueba y error, recibiendo recompensas o penalizaciones en función de sus acciones [19].

El RL se inspira en la psicología del comportamiento, particularmente en el concepto de condicionamiento operante. La idea central es que un agente debe aprender una política óptima (*policy*), que le permita seleccionar las acciones que maximicen su recompensa acumulada a lo largo del tiempo (como se observa en la Figura 2-1) [14].



Figura 2-5 Esquema básico de aprendizaje por refuerzo [15]

2.4.1 Fundamentos del aprendizaje por refuerzo

El RL se basa en varios conceptos clave:

- I. **Agente y entorno:** el agente es el sistema que aprende, mientras que el entorno es el espacio en el que el agente opera. El agente observa un estado (*state*, s) del entorno, toma una acción (*action*, a) y recibe una recompensa (*reward*, r). El objetivo del agente es maximizar la suma de recompensas que obtiene a lo largo del tiempo [5].
- II. **Política:** la política (*Policy*, π) es la estrategia que sigue el agente para seleccionar sus acciones. Puede ser determinista ($\pi(s) \rightarrow a$) o estocástica ($\pi(a|s) = P(a|s)$). La política define el comportamiento del agente y es el objetivo principal del aprendizaje por refuerzo [19].

- III. **Función de valor y función de acción-valor:** la función de valor (V , *Value Function*,) estima la recompensa esperada que un agente puede obtener desde un estado dado siguiendo una política. Por otro lado, la función de acción-valor (Q , *Q-Function*) estima la recompensa esperada para un estado específico cuando se toma una acción determinada [19].
- IV. **Proceso de Decisión de Markov:** el aprendizaje por refuerzo se formula matemáticamente a través de un Proceso de Decisión de Markov (*MDP*, *Markov Decision Process*), que se define por una tupla (S, A, P, R, γ) [14], donde:
 - S : conjunto de estados posibles.
 - A : conjunto de acciones posibles.
 - $P(s'|s, a)$: función de transición de estados.
 - $R(s, a)$: función de recompensa.
 - γ : factor de descuento, que determina la importancia de las recompensas futuras.
- V. **Exploración vs. explotación:** uno de los desafíos en RL es la exploración (*exploration*) frente a la explotación (*exploitation*). El agente debe equilibrar la exploración de nuevas estrategias con la explotación de aquellas que ya han demostrado ser efectivas [14]. Una de las técnicas más utilizadas para gestionar este equilibrio es la **política ϵ -greedy**, en la que el agente elige la mejor acción conocida con una alta probabilidad, pero mantiene una pequeña probabilidad ϵ de seleccionar una acción aleatoria, favoreciendo así el aprendizaje continuo y evitando caer en soluciones subóptimas [14].
- VI. **Algoritmos de aprendizaje por refuerzo:** existen diferentes enfoques para resolver problemas de RL:
 - **Q-Learning:** algoritmo basado en la actualización de la tabla Q mediante la ecuación de Bellman [5].
 - **Métodos basados en políticas:** optimización directa de la política sin necesidad de una función de valor.
 - **Deep Reinforcement Learning (DRL):** integración de DNNs con RL para manejar entornos complejos [19].

En este sentido, el aprendizaje por refuerzo profundo (DRL, *Deep Reinforcement Learning*,) es una combinación del RL clásico con DNNs, lo que permite a los agentes aprender en entornos de alta dimensionalidad y sin necesidad de una representación explícita del estado del entorno [19]. A diferencia de los métodos tradicionales de RL, que dependen de tablas de valores como *Q-Learning*, el DRL utiliza redes neuronales para aproximar funciones de valor o directamente la política de acción, permitiendo un escalamiento eficiente en problemas complejos [14].

2.4.2 Fundamentos del aprendizaje por refuerzo profundo

El DRL se basa en los siguientes conceptos clave:

1. Uso de DNNs: en lugar de almacenar valores en tablas, las DNNs permiten modelar funciones complejas, lo que facilita la generalización en entornos con estados continuos o de alta dimensión [18].
2. Exploración y explotación: al igual que en RL tradicional, el DRL debe equilibrar la exploración de nuevas estrategias con la explotación de las ya aprendidas [20].
3. Optimización por gradientes: se utilizan técnicas como descenso de gradiente estocástico (SGD) y algoritmos avanzados como Adam para ajustar los parámetros de las redes neuronales [21].
4. Memoria de experiencia (*experience replay*): introducida en Deep Q-Networks (DQN), esta técnica almacena experiencias pasadas y las reutiliza para entrenar de manera más eficiente y estable [22].

2.4.3 Principales algoritmos de DRL

Los algoritmos más utilizados en DRL se dividen en tres categorías mostradas a continuación en Tabla 2-1:

Métodos basados en valores (<i>Value-Based Methods</i>)	Estos algoritmos utilizan una red neuronal para estimar la función de valor $Q(s, a)$ y seleccionar la acción que maximiza la recompensa esperada. • Deep Q-Networks (DQN): introducido por DeepMind, mejora el Q-Learning clásico mediante el uso de redes neuronales convolucionales (CNNs) para procesar imágenes de entornos como Atari [22]. • DQN Mejorado: se han introducido variantes como Double DQN, Dueling DQN y <i>Prioritized Experience Replay</i> para mejorar la estabilidad del entrenamiento [23].
Métodos basados en políticas (<i>Policy-Based Methods</i>)	En lugar de estimar una función de valor, estos métodos aprenden directamente una política óptima $\pi(a s)$. • <i>Policy Gradient</i> (PG): entrena una política estocástica utilizando gradientes de la recompensa esperada [24]. • <i>Proximal Policy Optimization</i> (PPO): introducido por OpenAI, mejora la estabilidad del aprendizaje con restricciones en las actualizaciones de la política [25].
Métodos Actor-Critic	Estos algoritmos combinan métodos basados en valores y políticas para mejorar la estabilidad del entrenamiento. • <i>Advantage Actor-Critic</i> (A2C): divide el modelo en dos partes: un actor, que aprende la política óptima, y un crítico, que evalúa la acción tomada [26]. • <i>Asynchronous Actor-Critic</i> (A3C): variante distribuida que permite entrenar múltiples agentes en paralelo [26].

Tabla 2-1 Algoritmos más usados en DRL

2.4.4 Aplicaciones

El DRL ha demostrado ser efectivo en múltiples dominios:

- Videojuegos: *AlphaGo* y *StarCraft II* han mostrado el poder del DRL para aprender estrategias óptimas superando a jugadores humanos [27].
- Robótica: permite que robots aprendan tareas complejas sin programación explícita, como manipulación de objetos y navegación autónoma [28].
- Finanzas: se usa en la optimización de carteras y estrategias de trading automatizadas [29].
- Salud: aplicaciones en diagnóstico médico asistido por IA y optimización de tratamientos personalizados [30].

2.5 Juegos de estrategia

Los juegos de estrategia han sido un campo de prueba clave para la IA debido a la complejidad de sus reglas y la necesidad de planificación a largo plazo. A diferencia de los juegos de azar o acción, estos requieren que los jugadores tomen decisiones basadas en información incompleta y anticipen los movimientos del oponente. La IA ha avanzado significativamente en este ámbito, logrando desarrollar sistemas capaces de superar a jugadores humanos de élite en juegos como el ajedrez, *Go* y *StarCraft II*.

Dos de los hitos más importantes en este campo han sido *AlphaGo*, que revolucionó la IA en juegos de tablero al derrotar a campeones mundiales de *Go*, y *AlphaStar*, que demostró el potencial del aprendizaje por refuerzo en juegos de estrategia en tiempo real como *StarCraft II*. A continuación, se analizarán en detalle ambos sistemas y las innovaciones que introdujeron en el campo de la IA.

2.5.1 AlphaGo

El *Go* es un juego de mesa milenario de origen chino que se caracteriza por su gran profundidad estratégica y su inmenso número de posibles configuraciones en el tablero. A diferencia del ajedrez o las damas, en los que las piezas tienen movimientos definidos, en *Go* los jugadores colocan piedras en una cuadrícula de 19x19 (como se puede observar en la Figura 2-6) con el objetivo de rodear más territorio que el oponente. Su enorme cantidad de posibles jugadas y la falta de heurísticas claras hacen que sea un desafío considerable para la IA.



Figura 2-6 Juego de Go [31]

El desarrollo de *AlphaGo* por *DeepMind* en 2016 representó un hito fundamental en la IA aplicada a juegos de estrategia. Hasta entonces, los programas de IA habían logrado superar a los humanos en juegos como el ajedrez y las damas, pero *Go* presentaba un desafío significativamente mayor debido a su elevado espacio de estados, la complejidad de sus posiciones y la dificultad de evaluar jugadas sin un método determinista [5]. Mientras que el ajedrez tiene aproximadamente 10^{40} posibles posiciones, *Go* cuenta con aproximadamente 10^{170} , lo que hace inviable el uso de métodos tradicionales de búsqueda de fuerza bruta [4].

AlphaGo se convirtió en el primer sistema capaz de derrotar a un campeón mundial de *Go* (véase la Figura 2-7), demostrando la efectividad del DRL para resolver problemas de toma de decisiones en entornos de gran complejidad [17].

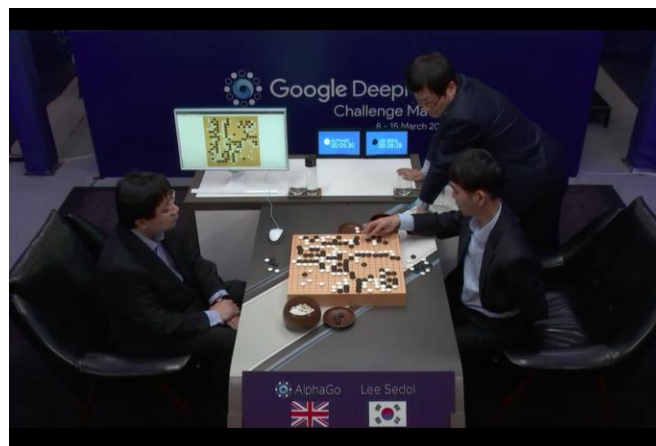


Figura 2-7 Partida de Go entre *AlphaGo* y el campeón del mundo [32]

2.5.1.1 Fundamentos técnicos

AlphaGo combinó varias técnicas avanzadas de IA para lograr un rendimiento sin precedentes en el juego de *Go*:

1. **Redes neuronales profundas:** se utilizó una arquitectura de CNN para analizar las posiciones del tablero y predecir los movimientos más prometedores. Estas redes neuronales fueron entrenadas con millones de partidas humanas y luego mejoradas mediante autoaprendizaje [33].
2. **Entrenamiento supervisado y autoaprendizaje:** inicialmente, el modelo fue entrenado con partidas jugadas por expertos humanos, lo que permitió generar una política base antes de refinarla con RL. En versiones posteriores, como *AlphaGo Zero*, se eliminó por completo el entrenamiento con datos humanos y el sistema aprendió desde cero a través de autojuego, lo que permitió alcanzar un nivel de juego superior [22].
3. **Aprendizaje por refuerzo profundo:** *AlphaGo* utilizó DQN y variantes de *Policy Gradient* para mejorar su estrategia iterativamente. Esto permitió al sistema aprender cuáles movimientos llevaban a situaciones ventajosas en el largo plazo sin necesidad de una evaluación heurística explícita [34].
4. **Monte Carlo Tree Search (MCTS):** a diferencia de los motores de ajedrez que analizan todas las posibles jugadas a través de métodos deterministas, *AlphaGo* utilizó MCTS para explorar de manera selectiva las ramas más prometedoras del árbol de decisiones. Esto redujo drásticamente la cantidad de jugadas a evaluar y mejoró la capacidad del sistema para jugar en posiciones desconocidas [35].

Este enfoque híbrido permitió que *AlphaGo* no solo alcanzara un nivel de juego humano, sino que desarrollara estrategias novedosas, muchas de las cuales sorprendieron incluso a jugadores profesionales de élite. *AlphaGo* demostró movimientos inesperados y creativos que cambiaron la forma en que los humanos entendían el juego de *Go*, lo que reforzó la idea de que la IA podía aportar enfoques innovadores en problemas complejos [27].

2.5.1.2 Impacto de AlphaGo en la IA y más allá

El éxito de *AlphaGo* tuvo un impacto significativo en la comunidad científica y en la industria del aprendizaje automático, desencadenando avances en múltiples áreas:

1. **Validación del aprendizaje por refuerzo profundo:** *AlphaGo* fue una de las primeras aplicaciones exitosas del DRL en un problema real altamente complejo. Su éxito consolidó el DRL como una técnica clave en IA, influyendo en numerosos desarrollos posteriores en robótica, simulaciones y planificación estratégica [36].
2. **Evolución hacia modelos más autónomos:** *AlphaGo* sirvió como base para *AlphaGo Zero*, un sistema que eliminó la necesidad de datos humanos y logró un rendimiento superior aprendiendo únicamente a partir de auto juego. Este enfoque fue más tarde generalizado en *Alpha Zero*, que logró dominar otros juegos como el ajedrez y el *shogi* (variante japonesa del ajedrez) con la misma arquitectura[37].
3. **Transformación del Go y la comunidad de jugadores:** la IA no solo venció a los mejores jugadores humanos, sino que también modificó la manera en que los jugadores humanos abordan el juego. *AlphaGo* introdujo movimientos y estrategias previamente desconocidas, las cuales han sido adoptadas por jugadores profesionales en torneos internacionales [28].
4. **Impacto en otras disciplinas:** las técnicas utilizadas en *AlphaGo* han sido aplicadas en diversas áreas, incluyendo:

- **Simulaciones militares y planificación estratégica:** el uso de IA en juegos de estrategia ha abierto la puerta a su implementación en simulaciones tácticas y entrenamiento militar [29].
 - **Optimización y toma de decisiones:** algunas empresas han comenzado a utilizar modelos basados en DRL para optimizar su logística, producción y distribución de recursos [30].
 - **Robótica y visión artificial:** el uso de DNNs y planificación eficiente se ha integrado en el desarrollo de robots autónomos con capacidades avanzadas de toma de decisiones [38].
5. **Ética y filosofía de la IA:** *AlphaGo* y sus sucesores han impulsado un debate sobre los límites de la IA y su impacto en el futuro del trabajo humano y la creatividad. El hecho de que una IA pudiera desarrollar estrategias innovadoras y jugar de una manera no intuitiva ha llevado a discusiones sobre el potencial de las máquinas para la creatividad y la innovación [39].

Tras la victoria de *AlphaGo*, DeepMind continuó su investigación con *AlphaZero*, un sistema aún más avanzado capaz de dominar múltiples juegos (*Go*, ajedrez y *shogi*) con un solo algoritmo de autoaprendizaje. Este tipo de IA genérica ha llevado a especulaciones sobre su aplicación en problemas más allá de los juegos, como la optimización de fármacos o la investigación científica.

2.5.2 AlphaStar

Tras el éxito de *AlphaGo*, DeepMind buscó expandir los límites del DRL hacia entornos más dinámicos y complejos. Mientras que el *Go* es un juego de información perfecta y por turnos, los juegos de estrategia en tiempo real (RTS, *Real-Time Strategy*) como *StarCraft II* presentan desafíos adicionales, como la toma de decisiones en tiempo real, la gestión de múltiples unidades y la existencia de información parcial [27].

En 2019, DeepMind presentó *AlphaStar*, un sistema de IA que logró alcanzar el nivel de Gran Maestro en *StarCraft II*, venciendo a jugadores profesionales en partidas 1v1 (como se puede observar en la Figura 2-8). Esto representó un avance significativo en el uso del DRL para la toma de decisiones en entornos complejos y dinámicos [19].



Figura 2-8 AlphaStar en partida 1v1 [40]

2.5.2.1 Principales diferencias entre AlphaGo y AlphaStar

AlphaGo y *AlphaStar* utilizan enfoques similares de DRL, pero deben resolver problemas completamente distintos:

1. **Información perfecta vs. información parcial:** en el *Go*, el tablero completo es visible en todo momento. En *StarCraft II*, los jugadores solo ven una parte del mapa, lo que requiere estrategias adaptativas basadas en exploración y predicción del comportamiento del oponente [20].

2. **Juego por turnos vs. tiempo real:** mientras que *AlphaGo* decide su movimiento y espera la respuesta del oponente, *AlphaStar* debe procesar **miles de acciones por minuto**, gestionando múltiples unidades en tiempo real [22].
3. **Árbol de decisiones vs. aprendizaje secuencial:** *AlphaGo* usa el método de Monte Carlo para evaluar movimientos futuros. *AlphaStar*, en cambio, utiliza **redes recurrentes y mecanismos de atención** para analizar secuencias de eventos y predecir estrategias [26].
4. **Estrategias de largo plazo:** en *Go*, la victoria depende de la acumulación de ventaja posicional. En *StarCraft II*, las estrategias deben ser adaptativas y ajustarse a **cambios en el entorno**, como recursos disponibles y tácticas del oponente [41].

A continuación en la Tabla 2-2 se presentan las principales diferencias entre ambos:

AlphaGo	AlphaStar
Información perfecta	Información parcial
Juego por turnos	Juego en tiempo real
Árbol de decisiones	Aprendizaje secuencial
Estrategia basada en la acumulación de ventaja	Estrategia basada en cambios en el entorno

Tabla 2-2 Diferencias entre AlphaGo y AlphaStar

2.5.2.2 Fundamentos técnicos

Para alcanzar un rendimiento superior, *AlphaStar* combinó múltiples técnicas avanzadas de IA:

1. **Aprendizaje supervisado y por refuerzo:** el modelo se entrenó inicialmente con partidas de jugadores humanos, aprendiendo estrategias generales antes de refinar su juego mediante autoaprendizaje por refuerzo [39].
2. **Enfoque multiagente:** *AlphaStar* utilizó un sistema **multiagente**, donde diferentes versiones de la IA competían entre sí, fomentando la exploración de estrategias diversas y reduciendo la sobre especialización [28].
3. **Redes neuronales profundas y mecanismos de atención:** se implementaron RNNs y **mecanismos de atención (Attention Mechanisms)** para modelar secuencias de acciones y mejorar la planificación en tiempo real [29].
4. **Limitación de acciones:** para evitar que la IA obtuviera ventajas inhumanas, se establecieron restricciones en la frecuencia de acciones por minuto y en la visibilidad del mapa, simulando condiciones realistas para los jugadores humanos [30].

2.5.2.3 Impacto de AlphaStar en la IA y más allá

El desarrollo de *AlphaStar* ha tenido implicaciones significativas en múltiples áreas de la IA:

1. **Expansión del aprendizaje por refuerzo a entornos complejos:** *AlphaStar* demostró que los algoritmos de DRLs pueden escalar hacia problemas de mayor complejidad, con múltiples agentes y entornos dinámicos [4].
2. **Innovaciones en multiagente y toma de decisiones:** la arquitectura utilizada en *AlphaStar* ha sido aplicada en simulaciones militares, robótica colaborativa y planificación logística [35].
3. **Impacto en la industria de los videojuegos:** *StarCraft II* se ha convertido en una plataforma estándar para la investigación en IA, impulsando la creación de sistemas más equilibrados y realistas [42].

4. **Mejoras en modelos secuenciales de IA:** los avances en **redes neuronales recurrentes y atención** han beneficiado otras áreas como el procesamiento del lenguaje natural y la conducción autónoma [43].

AlphaStar representa un avance clave en la IA aplicada a juegos de estrategia en tiempo real, abordando desafíos como la toma de decisiones en entornos dinámicos, la gestión de múltiples unidades y la información parcial. Su éxito valida el potencial del DRL para resolver problemas complejos en diversas industrias, desde videojuegos hasta robótica y planificación estratégica.

El estudio de sistemas como *AlphaGo* y *AlphaStar* no solo permite desarrollar IA más avanzadas, sino que también abre la puerta a aplicaciones en el mundo real que requieren adaptabilidad, planificación y toma de decisiones óptimas en tiempo real.

3 DESARROLLO DEL TFG

En este apartado se describe el proceso de implementación y mejora de la IA en el juego *Hundir la Flota*, utilizando técnicas de aprendizaje automático. El desarrollo se ha estructurado en varias etapas, comenzando con la programación del juego base y, posteriormente, incorporando estrategias progresivamente más avanzadas para mejorar el rendimiento de la máquina.

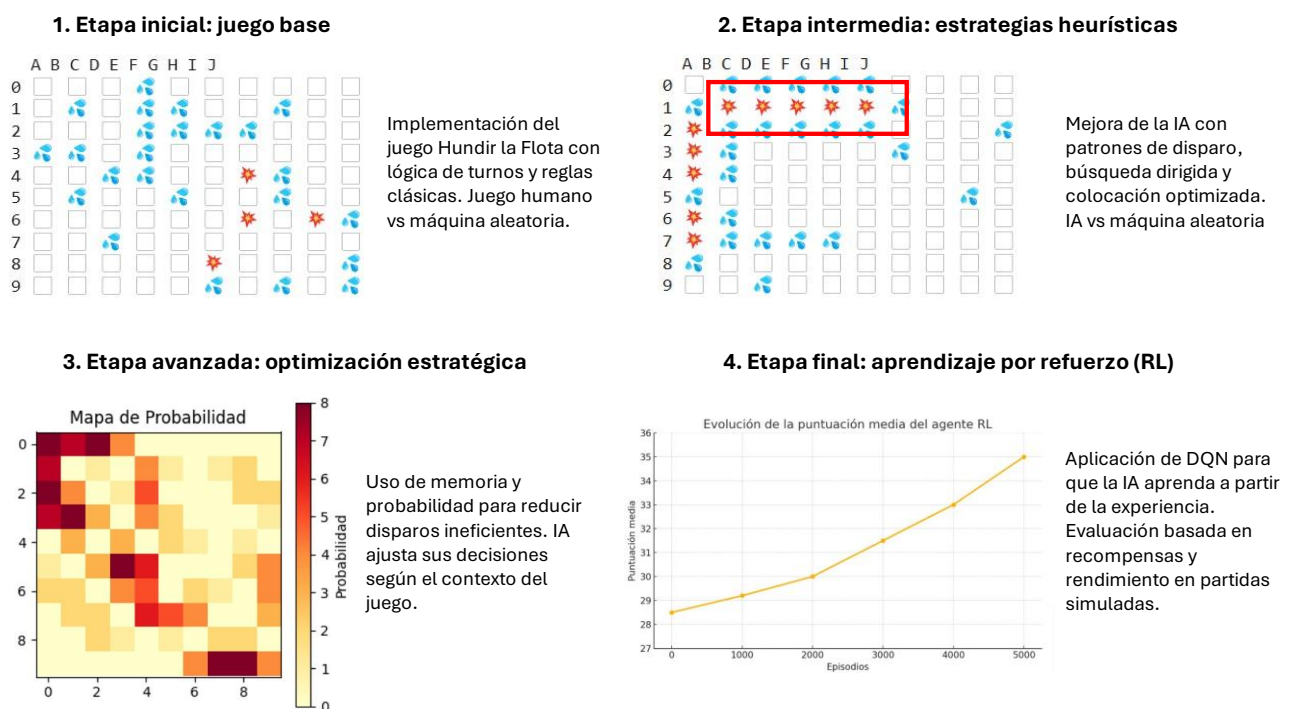


Figura 3-1 Metodología seguida en el proceso de desarrollo de la IA.

La Figura 3-1 resume las distintas etapas seguidas en el desarrollo de la IA aplicada al juego *Hundir la Flota*. En la primera fase se implementa la lógica básica del juego, enfrentando a un jugador humano contra una IA completamente aleatoria. En la segunda etapa, se incorporan estrategias heurísticas como la búsqueda tras impacto o el disparo en patrones para mejorar la toma de decisiones. Posteriormente, en la etapa avanzada, se introducen mecanismos de optimización basados en memoria y mapas de probabilidad, permitiendo a la IA adaptar sus disparos al contexto del tablero. Finalmente, en la etapa de aprendizaje por refuerzo, se aplica un agente DQN que aprende a partir de la experiencia acumulada, mejorando progresivamente su rendimiento a lo largo de múltiples partidas simuladas.

A lo largo de este capítulo, se detallará cada fase de la implementación, explicando las estrategias añadidas y la evolución del comportamiento de la IA. Además, se analizará el código correspondiente a cada mejora, justificando las decisiones tomadas en el diseño e implementación.

Para el desarrollo de la IA en el juego *Hundir la Flota*, se ha utilizado el lenguaje de programación Python, debido a su versatilidad y a la amplia disponibilidad de bibliotecas especializadas en IA y aprendizaje automático.

El entorno de desarrollo principal ha sido *Google Colab*, una plataforma basada en la nube que permite ejecutar código en notebooks de *Jupyter* sin necesidad de instalación local. Su uso ha facilitado la ejecución de experimentos, la depuración del código y la visualización de resultados, además de proporcionar acceso a recursos de cómputo como *GPUs* y *TPUs* para tareas más exigentes.

Sin embargo, para la ejecución del código correspondiente al entrenamiento por RL, se ha utilizado la aplicación *Visual Studio Code* (VS Code). A diferencia de *Google Colab*, esta herramienta no impone límites de tiempo en la ejecución, lo que ha permitido entrenar los modelos de forma continua y sin interrupciones.

Para el desarrollo, se han utilizado diversas bibliotecas de Python, entre las cuales destacan:

- `numpy`: para la manipulación de datos y operaciones numéricas, facilitando la gestión del tablero y los cálculos relacionados con la IA.
- `random`: para la generación de jugadas aleatorias en las etapas iniciales, así como para la selección de estrategias en ciertos casos.
- `IPython.display`: utilizada para mejorar la visualización en entornos como Google Colab, permitiendo limpiar la pantalla y mejorar la presentación de los datos. https://colab.research.google.com/drive/1-VYAwrWFadT_Nfohui3PeND_ByKV7l4T.
- `string`: empleada para la gestión de caracteres dentro del juego, en particular para la nomenclatura de las filas y columnas del tablero.
- `matplotlib.pyplot`: utilizada para la visualización de datos mediante gráficos, permitiendo analizar el rendimiento de la IA a través de histogramas que muestran la distribución de turnos en las partidas simuladas.
- `gym`: empleada para definir un entorno de entrenamiento personalizado compatible con aprendizaje por refuerzo, estructurando la dinámica del juego en episodios con recompensas y observaciones claras para el agente.

Este enfoque permitirá no solo demostrar la efectividad del aprendizaje automático en juegos de estrategia, sino también explorar cómo la IA puede adaptarse y mejorar su rendimiento en tiempo real a través de la experiencia.

3.1 Implementación del juego de Hundir la Flota

Hundir la Flota es un juego de estrategia por turnos en el que dos jugadores intentan localizar y destruir la flota del oponente antes de que este haga lo mismo. Cada jugador dispone de un tablero en el que coloca sus barcos de diferentes tamaños de forma oculta para el rival. Durante la partida, los jugadores se turnan para atacar casillas del tablero contrario, registrando impactos y hundiendo barcos hasta que uno de los jugadores ha perdido toda su flota.

El juego se desarrolla en un tablero de 10x10, y cada jugador dispone de los siguientes barcos:

- Portaviones (5 casillas)
- Acorazado (4 casillas)
- Submarino (3 casillas)
- Destructor (3 casillas)

- Lancha (2 casillas)

Los barcos pueden colocarse horizontal o verticalmente y el objetivo es hundir todos los barcos del adversario antes de que él hunda los tuyos.

Antes de aplicar estrategias avanzadas de IA, es fundamental desarrollar una versión funcional del juego *Hundir la Flota*. En este apartado, se describe el proceso de implementación del juego desde cero, asegurando que su mecánica y reglas se encuentren correctamente definidas.

El desarrollo comienza con la inicialización del juego (tablero y barcos) cada jugador, respetando las reglas clásicas del juego. A continuación, se establece la lógica de turnos y disparos, permitiendo la interacción entre jugadores y validando los impactos sobre los barcos. Finalmente, se implementa un sistema de visualización que facilita el seguimiento del estado del juego y la disposición de los elementos en el tablero.

Esta base servirá como punto de partida para la posterior integración de una IA que pueda jugar contra un oponente humano o contra otra, aplicando estrategias progresivamente más avanzadas.

3.1.1 Inicialización del juego

Antes de aplicar la mecánica de disparos o la lógica de turnos, es fundamental establecer una versión base del juego *Hundir la Flota*, asegurando que la estructura del tablero y la colocación de los barcos cumplan con el reglamento del juego. En este apartado se describen los procesos clave para la inicialización del juego, incluyendo la representación del tablero, la definición de los barcos y los criterios para garantizar que su colocación sea válida.

Para ello, primero se explicará la estructura del tablero, cómo se representa en el código y la lógica utilizada para gestionar su estado. A continuación, se detallará la definición y colocación de los barcos, asegurando que respeten las reglas del juego y no se superpongan. Finalmente, se abordarán las consideraciones necesarias para validar la colocación, garantizando que todas las posiciones asignadas cumplan con los requisitos establecidos.

Estos elementos constituyen la base del juego, permitiendo posteriormente la integración de una IA capaz de jugar contra un oponente humano o contra otra máquina.

3.1.1.1 Representación del tablero

El juego se desarrolla sobre un **tablero de 10x10**, donde cada casilla puede representar agua, un barco o el resultado de un disparo. Para facilitar su manipulación, se utiliza la biblioteca `numpy`, que permite gestionar matrices de forma eficiente.

La estructura del tablero se define mediante la función `crear_tablero()` (obsérvese la Figura 3-2), la cual inicializa una matriz de tamaño **10x10** con el carácter "~", que simboliza agua:

```
import numpy as np

def crear_tablero():
    return np.full((10, 10), "~") # '~' representa agua
```

Figura 3-2 Código de inicialización del tablero.

Cada jugador contará con su propio tablero, donde podrá colocar barcos y registrar impactos. Para mejorar la visualización, se implementa la función `imprimir_tablero()` (véase la Figura 3-3), que muestra el estado actual del tablero en la terminal:

```
import string

COLUMN_LETTERS = list(string.ascii_uppercase[:10]) # Letras de columnas (A-J)

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS)) # Cabecera con letras A-J
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["🔥", "💧", "~"] else "~" for cell in fila))
        else:
            print(str(i) + " " + " ".join(fila))
```

Figura 3-3 Código de impresión del tablero.

- "🔥" representa un impacto en un barco.
- "💧" indica un disparo fallido (agua).
- "~" representa casillas sin disparos.
- Los barcos pueden tener diferentes letras ("P", "A", "S", "D", "L") según su tipo, pero estas solo se muestran cuando `ocultar_barcos=False` (para evitar que el oponente vea la ubicación de los barcos).

Esta función permite visualizar el tablero de manera clara y facilita el seguimiento del juego.

3.1.1.2 Definición y colocación de los barcos

Antes de iniciar la partida, cada jugador debe colocar sus barcos en el tablero respetando ciertas reglas fundamentales:

- Los barcos deben colocarse completamente dentro del tablero.
- No pueden solaparse con otros barcos.
- Pueden colocarse en orientación horizontal (H) o vertical (V).

En el juego, los barcos pueden colocarse de manera manual por el jugador o de forma aleatoria cuando la máquina juega. Como veremos más adelante, solo en la primera versión del juego se introducen los barcos de forma manual por el usuario, esto se debe a la necesidad de agilizar el proceso de evaluación de las estrategias y poder hacer muchas simulaciones de partidas, por ello se opta por la colocación aleatoria.

Cada jugador dispone de los siguientes cinco tipos de barcos, como ya se explicó al principio del apartado, cada uno con un tamaño diferente (como se pueden observar en el código de Figura 3-4):

```
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}
```

Figura 3-4 Código de definición de los barcos.

Estos barcos conforman la flota de cada jugador y deben distribuirse estratégicamente en el tablero para evitar que sean fácilmente detectados por el oponente. Para colocar estos barcos en el tablero, al

jugador se le permite insertar manualmente las coordenadas y la orientación de cada uno. La función `colocar_barcos_manual()` gestiona este proceso como se muestra en la Figura 3-5:

```
def letra_a_indice(letra):
    return COLUMN_LETTERS.index(letra.upper()) if letra.upper() in COLUMN_LETTERS else -1

def colocar_barcos_manual(tablero):
    print("Coloca tus barcos manualmente:")
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            try:
                imprimir_tablero(tablero, ocultar_barcos=False)
                fila = int(input(f"Ingresa la fila inicial para {barco} ({tamaño} casillas): "))
                columna_letra = input(f"Ingresa la columna inicial (A-J) para {barco} ({tamaño} casillas): ")
                columna = letra_a_indice(columna_letra)
                orientacion = input("Ingresa la orientación (H para horizontal, V para vertical): ").upper()
                if columna == -1:
                    print("Columna no válida. Usa letras de A a J.")
                    continue
                # Verifica si el barco cabe en la orientación elegida y si no hay otro barco en la zona
                if orientacion == "H" and columna + tamaño <= 10 and all(tablero[fila, columna+i] == "~" for i in range(tamaño)):
                    for i in range(tamaño):
                        tablero[fila, columna+i] = barco[0] # Se coloca con la inicial del barco
                    colocado = True
                elif orientacion == "V" and fila + tamaño <= 10 and all(tablero[fila+i, columna] == "~" for i in range(tamaño)):
                    for i in range(tamaño):
                        tablero[fila+i, columna] = barco[0]
                    colocado = True
                else:
                    print("Posición no válida, intenta nuevamente.")
            except (ValueError, IndexError):
                print("Entrada inválida, ingresa valores numéricos válidos para la fila y una letra válida para la columna.")
        return tablero
```

Figura 3-5 Código para la colocación de los barcos manualmente.

Funcionamiento de la colocación de barcos:

1. Se solicita al usuario introducir las coordenadas iniciales del barco.
2. Se verifica si la posición es válida y si el barco cabe en la orientación seleccionada (H para horizontal, V para vertical).
3. Si la posición es válida, el barco se coloca usando su inicial ("P", "A", "S", "D", "L").
4. Si la posición no es válida, se solicita una nueva entrada.

Cuando la máquina coloca sus barcos, utiliza un proceso de **posicionamiento aleatorio** asegurando que los barcos se ubiquen correctamente sin salirse del tablero ni superponerse. Para ello, se emplea la función `colocar_barcos()` (representada en la Figura 3-6):

```
def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila, columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila + i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero
```

Figura 3-6 Código de la colocación aleatoria de los barcos.

El funcionamiento es el siguiente:

1. Se recorre el diccionario `BARCOS`, que contiene los barcos del juego junto con sus respectivos tamaños.
2. Se establece un bucle `while not colocado:` para asegurar que cada barco se posicione correctamente antes de pasar al siguiente.
3. Se generan coordenadas aleatorias (fila, columna) dentro del tablero.
4. Se selecciona una orientación aleatoria (H para horizontal o V para vertical).
5. Se verifica si el barco cabe completamente dentro del tablero sin salirse de los límites.
6. Se verifica que todas las casillas en las que se colocará el barco estén vacías.
7. Si la posición es válida, el barco se coloca en el tablero asignando la inicial del barco ("P", "A", "S", "D", "L").
8. Si la posición no es válida (porque el barco se sale del tablero o se superpone con otro), se vuelve a intentar con nuevas coordenadas hasta encontrar una ubicación correcta.

3.1.1.3 Consideraciones para la colocación de los barcos

Para garantizar que la colocación de los barcos sea válida y cumpla con las reglas del juego, se han definido una serie de restricciones que deben verificarse antes de posicionar cualquier embarcación en el tablero:

- Los barcos no pueden superponerse. Antes de colocar un barco, se tiene que verificar que todas las casillas estén vacías ("~").
- Los barcos deben estar completamente dentro del tablero. Se valida que la última casilla del barco no se salga de los límites (por ejemplo, un portaviones no puede empezar en la columna G si va en horizontal).
- Se permite la colocación manual. El jugador decide dónde colocar cada barco, respetando las restricciones.

3.1.2 Lógica de turnos y mecánica de disparos

En esta sección se describe la lógica que permite a los jugadores, ya sean humanos o máquinas, realizar sus disparos sobre el tablero del oponente y determinar el desarrollo de la partida. La mecánica de disparo es fundamental para el juego, ya que permite registrar impactos, fallos y decidir cuándo una partida ha finalizado.

3.1.2.1 Turnos y disparos

Una vez que los barcos han sido colocados en el tablero, comienza la fase de ataque, en la cual cada jugador, en su turno, selecciona una casilla del tablero contrario para disparar. En la versión actual del código, el juego se desarrolla con dos jugadores, que pueden ser tanto humanos como inteligencias artificiales. En cada turno, un jugador selecciona una casilla del tablero contrario para disparar, con el objetivo de hundir todos los barcos del oponente. En el caso de que el jugador sea una máquina, los disparos se realizan de forma aleatoria utilizando la función `disparo_maquina()`, que gestiona la lógica de ataque automatizado (como se muestra en Figura 3-7).

```
def disparo_maquina(tablero):
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero
```

Figura 3-7 Código que define el disparo aleatorio.

La función selecciona de manera aleatoria una casilla del tablero del oponente y verifica que no haya sido atacada previamente, es decir, que no contenga los marcadores "X" (impacto) o "O" (agua). Si la casilla elegida contiene "~", el disparo se considera un fallo y se marca con "O", mientras que, si en su lugar hay una parte de un barco identificado con las iniciales "P", "A", "S", "D" o "L", el disparo es un impacto y se marca con "X". La función se ejecuta en un bucle hasta encontrar una casilla válida para disparar, asegurando así que no se repitan intentos en posiciones ya atacadas. Esta mecánica es utilizada por ambas máquinas durante cada turno para llevar a cabo su estrategia de ataque.

3.1.2.2 Condiciones de finalización del juego

El juego finaliza cuando uno de los jugadores ha hundido completamente todos los barcos del adversario. Para determinar si todavía quedan barcos en el tablero, se utiliza la función `quedan_barcos()` (véase la Figura 3-8):

```
def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)
```

Figura 3-8 Código de comprobación de barcos en el tablero.

La función examina todas las casillas del tablero en busca de elementos distintos de agua ("~"), impacto ("X") o fallo ("O"), ya que la presencia de cualquier otro símbolo indica que aún quedan barcos en juego. Si no se encuentra ninguna casilla con un barco, significa que todos han sido hundidos, lo que implica la derrota del jugador y la finalización de la partida. Este método es fundamental para determinar el momento en que un jugador ha ganado.

3.1.2.3 Simulación de partidas

Para evaluar el comportamiento del sistema y medir la cantidad de partidas ganadas por cada jugador, se implementa la función `simular_partidas()` (mostrada en la Figura 3-9); que ejecuta múltiples enfrentamientos entre dos máquinas. Esta simulación no solo permite analizar el rendimiento actual del sistema, sino que también servirá como herramienta para agilizar el estudio de las mejoras que

se irán implementando en la IA. Para ello, se enfrentará nuestra maquina contra una versión que juega de manera completamente aleatoria, lo que facilitará la comparación y permitirá medir de forma objetiva el impacto de cada optimización introducida.

```
def simular_partidas(n=1000):
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []

        # Bucle de juego: cada ronda ambas máquinas disparan
        while True:
            # Máquina 1 dispara (usa su estrategia de caza y eliminación)
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            # Máquina 2 dispara (su disparo es aleatorio)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluar el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            # Si ambas máquinas pierden sus barcos en el mismo turno, es un empate
            if not m1 and not m2:
                empates += 1
                break

            # Si la Máquina 2 se queda sin barcos, gana la Máquina 1
            elif not m2:
                victorias_maquina_1 += 1
                break

            # Si la Máquina 1 se queda sin barcos, gana la Máquina 2
            elif not m1:
                victorias_maquina_2 += 1
                break

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (estrategia caza y eliminación) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
```

Figura 3-9 Código que inicia la simulación de las partidas.

Para llevar el registro de los resultados, se definen **tres contadores**: `victorias_maquina_1` y `victorias_maquina_2`, que registran las partidas ganadas por cada jugador, y `empates`, que contabiliza aquellas partidas en las que ambos jugadores pierden todos sus barcos en la misma ronda.

El proceso de simulación comienza generando nuevos tableros para ambos jugadores mediante la función `colocar_barcos()`. Luego, se ejecuta un bucle en el que los turnos de disparo se alternan entre Máquina 1 y Máquina 2, utilizando sus respectivas funciones de disparo. En cada iteración, se evalúa el estado de los tableros para determinar si alguno de los jugadores ha perdido toda su flota. Si esto ocurre, se asigna la victoria al oponente; sin embargo, si ambos jugadores pierden en la misma ronda, la partida se registra como un empate.

Una vez finalizadas todas las simulaciones, se imprimen los resultados, mostrando la cantidad de partidas ganadas por cada máquina y el número total de empates. Esta información es clave para evaluar la efectividad de las estrategias implementadas y medir la eficiencia de cada optimización introducida en la IA.

3.2 Estrategias básicas de la IA

Para que la IA en *Hundir la Flota* pueda mejorar su rendimiento, es necesario partir de una versión básica que sirva como referencia. En esta sección, se implementarán y evaluarán distintas estrategias para optimizar la toma de decisiones de la máquina.

Inicialmente, la IA jugará de manera completamente aleatoria, sin ningún tipo de estrategia. A partir de ahí, se introducirán mejoras progresivas en su comportamiento, con el objetivo de aumentar su

precisión en los disparos y optimizar la estrategia de colocación de barcos, asegurando una distribución más efectiva para dificultar su detección por parte del oponente.

3.2.1 Estrategia aleatoria: primeros pasos

La primera estrategia realiza sus disparos de manera completamente aleatoria, sin analizar información previa ni aprender de sus movimientos anteriores. Este enfoque, aunque poco eficiente, permite establecer una línea base de rendimiento para comparar futuras mejoras.

En esta versión, se elige una casilla aleatoria del tablero del oponente en cada turno, asegurándose de no disparar en una casilla ya atacada. Si la casilla contiene agua ("~"), el disparo se marca como fallo ("o"); si impacta en un barco, se marca con ("x"). Esta lógica está implementada en la función `disparo_maquina()`, ya explicada anteriormente.

Además de mejorar la precisión en los disparos, también se buscará optimizar la colocación de los barcos en el tablero, ya que una disposición más estratégica puede influir directamente en el desarrollo de la partida. Actualmente, la función `colocar_barcos()` (representada en la Figura 3-6) posiciona las embarcaciones de forma completamente aleatoria, sin seguir patrones que dificulten la detección por parte del adversario. En futuras versiones, se estudiarán estrategias más avanzadas para distribuir los barcos de manera que minimicen la probabilidad de ser encontrados rápidamente.

Para medir el rendimiento de esta estrategia, se utiliza la función `simular_partidas()` (mostrada en la Figura 3-9), que permite realizar múltiples enfrentamientos y analizar cuántas partidas gana cada jugador. Esta simulación servirá como herramienta para evaluar el impacto de las mejoras de las estrategias empleadas.

En la siguiente sección, se explorarán métodos para optimizar la precisión en los disparos y mejorar la estrategia de colocación de barcos, reduciendo la cantidad de disparos fallidos.

3.2.2 Implementación de estrategias heurísticas

En este apartado, se introduce la primera mejora del juego *Hundir la Flota*, pasando de una estrategia completamente aleatoria a una basada en heurísticas. Mientras que en la versión anterior se seleccionaban los disparos de manera totalmente aleatoria y se colocaban los barcos sin seguir ningún criterio estratégico, ahora se busca optimizar ambos procesos para mejorar el rendimiento en las partidas.

Las heurísticas permiten tomar decisiones un poco más inteligentes sin necesidad de aprendizaje automático, simplemente aplicando reglas predefinidas que aumentan la probabilidad de éxito en cada jugada. En este caso, se explorarán dos tipos principales de mejoras:

- Optimización en los disparos: en lugar de atacar casillas al azar, se adoptará estrategias como búsqueda dirigida tras un impacto y disparos usando distintos patrones como puede ser la búsqueda por cuadrantes, donde se priorizará la búsqueda en un cuadrante antes de pasar al siguiente, asegurándose de que no queda ningún barco en el cuadrante investigado.
- Optimización en la colocación de barcos: en lugar de ubicar los barcos en posiciones completamente aleatorias, se explorarán estrategias que reduzcan la previsibilidad y maximicen la dificultad para el oponente al encontrar la flota.

3.2.2.1 Estrategia de caza tras un impacto

Hasta este punto, la IA seleccionaba sus disparos de manera completamente aleatoria, sin aprovechar la información obtenida en turnos previos. La estrategia de búsqueda dirigida tras un impacto introduce una mejora significativa, permitiendo que la IA recuerde sus impactos y realice disparos en posiciones cercanas para tratar de hundir el barco.

Esta técnica se basa en un modo de caza y eliminación, donde la IA primero busca impactar un barco y, cuando lo consigue, cambia su comportamiento para disparar en las casillas adyacentes hasta hundirlo por completo.

La mejora se ha implementado en la función `disparo_maquina_1()` (como se muestra en la Figura 3-10) que gestiona los disparos de una IA optimizada con esta estrategia.

```
def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in ["X", "O"]:
                    memoria.append((nf, nc))
    return tablero
```

Figura 3-10 Código de disparo con estrategia de caza.

La función `disparo_maquina_1()` implementa una estrategia de caza y eliminación, en la que se recuerda los impactos llevados a cabo y se continúa disparando en las casillas adyacentes hasta hundir el barco. Para ello, utiliza una lista `memoria` donde almacena coordenadas pendientes de atacar. Si `memoria` tiene coordenadas guardadas, se dispara en una de ellas; en caso contrario, se elige una casilla aleatoria. Si el disparo cae en agua ("~"), la casilla se marca como fallo ("O") y no se añade nada a la lista `memoria`. Si impacta en un barco, se marca con ("X") y se agregan a `memoria` las posiciones adyacentes (arriba, abajo, izquierda y derecha), siempre que estén dentro del tablero y no hayan sido disparadas antes. Esto permite que, cuando la IA encuentre un barco, no vuelva a disparar aleatoriamente, sino que lo ataque de manera sistemática hasta hundirlo. Gracias a esta estrategia, se mejora la precisión y se reduce el número de intentos fallidos en comparación con una estrategia completamente aleatoria.

3.2.2.2 Búsqueda por cuadrantes

Para mejorar la eficiencia en la detección de barcos, se puede implementar un patrón de búsqueda por cuadrantes, dividiendo el tablero en zonas más pequeñas y explorándolas de manera estructurada. Este método permite que la IA no disperse sus disparos de forma aleatoria por todo el tablero, sino que concentre su búsqueda en áreas específicas, lo que aumenta la probabilidad de encontrar barcos en menos intentos.

La idea detrás de esta estrategia es dividir el tablero de 10x10 en 4 cuadrantes de 5x5, generando una lista de disparos dentro de cada cuadrante y ejecutándolos de manera ordenada, aunque aleatoria dentro de cada zona (véase en el código de la Figura 3-11).

```
def generar_cuadrantes():
    """Genera una lista de disparos en cuadrantes para mejorar la búsqueda."""
    cuadrantes = []
    tam_cuadrante = BOARD_SIZE // 2 # Divide el tablero en 4 cuadrantes de 5x5
    for i in range(0, BOARD_SIZE, tam_cuadrante):
        for j in range(0, BOARD_SIZE, tam_cuadrante):
            cuadrante = [(x, y) for x in range(i, i + tam_cuadrante) for y in range(j, j + tam_cuadrante)]
            random.shuffle(cuadrante) # Mezclar para mayor variabilidad
            cuadrantes.extend(cuadrante)
    return cuadrantes
```

Figura 3-11 Código de generación de cuadrantes.

Como comentado, la función divide el tablero en 4 cuadrantes de 5x5 y genera una lista de coordenadas dentro de cada cuadrante. Luego, las casillas dentro de cada zona se barajan aleatoriamente para evitar patrones predecibles y se añaden a la lista `cuadrantes`, que se utilizará para disparar de manera ordenada dentro de cada área antes de pasar al siguiente cuadrante.

3.2.2.3 Búsqueda en Zigzag

Para mejorar la eficiencia en la detección de barcos, se puede implementar un patrón de búsqueda en zigzag, que permite recorrer el tablero de manera estructurada en lugar de realizar disparos completamente aleatorios. Este método busca maximizar la cobertura del tablero minimizando disparos innecesarios, lo que incrementa la probabilidad de encontrar barcos en menos intentos.

La idea detrás de esta estrategia es recorrer el tablero alternando la dirección de los disparos en cada fila. En las filas pares, los disparos se realizan de izquierda a derecha, mientras que en las filas impares se ejecutan de derecha a izquierda. Este enfoque permite distribuir los disparos de manera uniforme y evitar patrones predecibles que podrían reducir la eficiencia de la búsqueda.

La función `generar_patron_zigzag()` genera una lista ordenada de disparos en zigzag para garantizar que el tablero sea explorado de manera sistemática. Además, se incorpora esta estrategia en la función `disparo_maquina_1()`, que combina este patrón con una estrategia de caza cuando se encuentra un barco.

```
def generar_patron_zigzag():
    """Genera un patrón de disparo en zigzag para mejorar la búsqueda."""
    zigzag = []
    for fila in range(BOARD_SIZE):
        if fila % 2 == 0:
            for columna in range(BOARD_SIZE):
                zigzag.append((fila, columna))
        else:
            for columna in reversed(range(BOARD_SIZE)):
                zigzag.append((fila, columna))
    return zigzag
```

Figura 3-12 Código generador del patrón zigzag.

Como se muestra en el código de la Figura 3-12, la función `generar_patron_zigzag()` crea una lista con las coordenadas organizadas en un patrón de zigzag para ser utilizadas en los disparos. Luego, la función `disparo_maquina_1()` aplica este patrón de búsqueda mientras alterna con una estrategia de caza cuando se detecta un impacto, asegurando una optimización en la exploración del tablero.

3.2.2.4 Patrón de búsqueda en cuadrícula

El patrón de disparo en cuadrícula es una estrategia que optimiza la búsqueda de barcos distribuyendo los disparos en una formación similar a un tablero de ajedrez. Esta estrategia surge de la observación de que los barcos en *Hundir la Flota* tienen al menos dos casillas de largo, lo que significa que disparar en todas las casillas no es necesario. En su lugar, disparar en una cuadrícula alterna maximiza la probabilidad de encontrar barcos grandes con menos intentos.

La IA primero sigue este patrón de exploración, disparando en casillas intercaladas, y cuando impacta un barco, cambia a modo de caza y eliminación, disparando en las casillas adyacentes para hundirlo por completo antes de volver a la exploración en cuadrícula.

En el código de la Figura 3-13, la función `generar_patron_busqueda()` genera la lista de coordenadas siguiendo el patrón de ajedrez:

```
def generar_patron_busqueda():
    """Genera un patrón de búsqueda tipo damero para mejorar la eficiencia."""
    patron = [(fila, columna) for fila in range(BOARD_SIZE) for columna in range(BOARD_SIZE) if (fila + columna) % 2 == 0]
    random.shuffle(patron) # Mezclar el patrón para mayor variabilidad
    return patron
```

Figura 3-13 Código de generación de patrón de búsqueda.

La función recorre todas las casillas del tablero y solo añade aquellas donde la suma de la fila y la columna sea par ($(\text{fila} + \text{columna}) \% 2 == 0$), generando así una cuadrícula intercalada. Luego, mezcla la lista de disparos con `random.shuffle()` para evitar patrones predecibles.

La estrategia de disparo en cuadrícula se usa en `disparo_maquina_1()`, que combina este enfoque con la búsqueda dirigida tras un impacto como se muestra a continuación en la Figura 3-14:

```
def disparo_maquina_1(tablero, memoria, patron_busqueda):
    """Máquina 1 con estrategia de caza y patrón de búsqueda."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif patron_busqueda:
        fila, columna = patron_busqueda.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in ["X", "O"]:
                    memoria.append((nf, nc))
    return tablero
```

Figura 3-14 Código de disparo con patrón de búsqueda.

Esta función gestiona los disparos combinando dos estrategias de forma secuencial. En primer lugar, dispara siguiendo un patrón de cuadrícula predefinido en `patron_busqueda`, lo que maximiza la probabilidad de encontrar barcos en menos intentos. A continuación, si se impacta un barco ("X"), la estrategia cambia al modo de caza y eliminación, disparando en las casillas adyacentes hasta hundirlo por completo. En caso de que tanto `memoria` como `patron_busqueda` estén vacíos, el disparo se realiza de forma completamente aleatoria.

3.2.2.5 Distribución espaciada

Una de las estrategias más efectivas para mejorar la supervivencia en *Hundir la Flota* es optimizar la colocación de los barcos para hacerlos más difíciles de encontrar. En esta sección, se implementa una estrategia de colocación espaciada, que busca distribuir los barcos de manera que no queden agrupados ni alineados en patrones fácilmente detectables.

El objetivo principal de esta estrategia es evitar que los barcos estén demasiado cerca entre sí, lo que reduce la probabilidad de que el oponente impacte múltiples barcos con disparos consecutivos. Para lograrlo, se establece una distancia mínima entre barcos, asegurando que cada uno tenga suficiente espacio alrededor antes de colocarlo en el tablero.

La función `colocar_barcos_espaciado()` gestiona la colocación de los barcos siguiendo esta estrategia:

```
def colocar_barcos_espaciado(tablero):
    """Coloca los barcos con una distribución más separada para mayor defensa."""
    intentos_max = 50 # Evita bucles infinitos

    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0

        while not colocado and intentos < intentos_max:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            # Verifica que el barco no toque otro barco y que haya espacio alrededor
            if orientacion == "H" and columna + tamaño <= BOARD_SIZE:
                if all(tablero[fila, columna + i] == "~" for i in range(tamaño)) and all(
                    0 <= columna + i < BOARD_SIZE and
                    0 <= fila < BOARD_SIZE and
                    tablero[fila, max(0, columna + i - 1) : min(BOARD_SIZE, columna + i + 2)]
                    .tolist()
                    .count("~") == min(BOARD_SIZE, columna + i + 2) - max(0, columna + i - 1)
                    for i in range(tamaño)
                ):
                    for i in range(tamaño):
                        tablero[fila, columna + i] = barco[0]
                    colocado = True

            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE:
                if all(tablero[fila + i, columna] == "~" for i in range(tamaño)) and all(
                    0 <= fila + i < BOARD_SIZE and
                    0 <= columna < BOARD_SIZE and
                    tablero[max(0, fila + i - 1) : min(BOARD_SIZE, fila + i + 2), columna]
                    .tolist()
                    .count("~") == min(BOARD_SIZE, fila + i + 2) - max(0, fila + i - 1)
                    for i in range(tamaño)
                ):
                    for i in range(tamaño):
                        tablero[fila + i, columna] = barco[0]
                    colocado = True

            intentos += 1

    return tablero
```

Figura 3-15 Código de colocación de barcos con separación.

Esta función coloca los barcos asegurando que no estén demasiado cerca unos de otros como se observa en el código de la Figura 3-15. Para cada barco, se elige una posición aleatoria en el tablero y una orientación (H horizontal o V vertical). Luego, antes de colocarlo, se verifica que todas las casillas donde se va a ubicar estén vacías ("~") y que haya un espacio de separación alrededor para evitar que los barcos queden pegados. Si no se encuentra una posición válida después de 50 intentos, la función termina para evitar un bucle infinito.

3.2.2.6 Distribución en los bordes

Otra estrategia interesante para mejorar la colocación de los barcos consiste en ubicarlos preferentemente en los bordes del tablero. Esta estrategia tiene el objetivo de reducir la probabilidad de

ser detectados rápidamente, ya que muchos jugadores tienden a enfocar su búsqueda en el centro del tablero antes de revisar los extremos.

Este enfoque puede hacer que el oponente tarde más en localizar barcos si su patrón de disparo favorece el área central del tablero. Sin embargo, también tiene un riesgo, ya que si el oponente detecta la presencia de un barco en un borde, podría asumir que el resto también están allí y adaptar su estrategia de búsqueda.

La función `colocar_barcos_en_bordes()` (véase la Figura 3-16) coloca los barcos priorizando las filas y columnas exteriores:

```
def colocar_barcos_en_bordes(tablero):
    """Coloca los barcos preferentemente en los bordes del tablero."""
    intentos_max = 50 # Evita bucles infinitos
    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0
        while not colocado and intentos < intentos_max:
            # Elegir una fila o columna en los bordes
            if random.choice([True, False]): # Decide si colocar en filas borde
                fila = random.choice([0, BOARD_SIZE - 1])
                columna = random.randint(0, BOARD_SIZE - tamaño)
                orientacion = "H"
            else: # 0 en columnas borde
                fila = random.randint(0, BOARD_SIZE - tamaño)
                columna = random.choice([0, BOARD_SIZE - 1])
                orientacion = "V"

            # Verificar si la posición está libre
            if orientacion == "H" and all(tablero[fila, columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and all(tablero[fila + i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
            intentos += 1
        return tablero
```

Figura 3-16 Código de colocación de barcos en los bordes.

Esta función prioriza la colocación de los barcos en los bordes del tablero, eligiendo aleatoriamente si colocarlos en una fila externa (0 o 9) o en una columna externa (0 o 9). Luego, genera una posición inicial aleatoria dentro del borde elegido y selecciona si el barco se colocará en orientación horizontal (H) o vertical (V). Antes de ubicarlo, la función verifica que no haya otros barcos en las casillas seleccionadas. Si no encuentra una posición válida en 50 intentos, se detiene para evitar bucles infinitos.

3.2.3 Optimización de disparos en función de impactos

A medida que se mejora la capacidad para detectar y destruir barcos, es necesario optimizar la forma en que gestiona la información obtenida de los disparos. No solo es importante acertar, sino también reducir intentos innecesarios y maximizar la eficiencia en la búsqueda. En este apartado, se implementan estrategias que permiten a la IA priorizar sus disparos tras un impacto, recordar qué barcos quedan en juego para enfocar su búsqueda en zonas viables y evitar disparos en áreas donde, por lógica, no puede haber barcos. Con estas mejoras, se busca reducir el número de turnos necesarios para ganar y aumentar la eficacia del sistema.

3.2.3.1 Priorización y optimización tras un impacto

Hasta ahora, la IA realizaba disparos aleatorios hasta impactar un barco, y luego disparaba en casillas adyacentes sin un orden definido. La versión actual **optimiza la búsqueda tras un impacto**, priorizando disparos en línea recta para detectar la orientación del barco y maximizar su eliminación con la menor cantidad de intentos.

Esta nueva mejora se aplica en la función `disparo_maquina_1()`:

```
def disparo_maquina_1(tablero, memoria, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            if objetivo is None:
                objetivo = (fila, columna)
            # Explorar solo en líneas rectas (horizontal o vertical)
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                    if tablero[nf, nc] == "~":
                        break # Detener la búsqueda en esta dirección si encuentra agua
                    if tablero[nf, nc] not in ["X", "O"]:
                        memoria.append((nf, nc))
                    nf += df
                    nc += dc
    return tablero
```

Figura 3-17 Código de mejora de la estrategia de caza.

Ahora los pasos que se siguen son los siguientes:

1. Búsqueda de un impacto inicial:

- Si no hay registros previos (memoria vacía), dispara en una casilla aleatoria.
- Si impacta un barco ("X"), se guarda la coordenada del impacto (objetivo).

2. Priorización de disparos en línea recta:

- Una vez que se impacta, sólo explora en líneas horizontales y verticales, evitando disparos diagonales.
- Se agregan a `memoria` las casillas en esas direcciones hasta que se encuentra agua ("~").
- Esto permite determinar si el barco está colocado en horizontal o vertical rápidamente.

3. Optimización de la exploración:

- Si hay múltiples impactos seguidos en la misma dirección, se sigue disparando en esa dirección hasta hundir el barco.
- Si en una dirección encuentra agua, deja de explorar en esa línea y cambia a la otra posible orientación.

3.2.3.2 Seguimiento de barcos restantes para optimizar la búsqueda

En este apartado, se busca mejorar la toma de decisiones de la IA al momento de seleccionar sus disparos. Hasta ahora, la IA seguía un patrón de búsqueda que priorizaba la caza y eliminación de barcos una vez impactados. Sin embargo, aún queda un problema sin resolver: no tiene en cuenta qué barcos quedan en el tablero, lo que puede llevar a realizar disparos en zonas donde, por lógica, ya no podrían haber los barcos restantes. La solución a esto es implementar un sistema de memoria que permita a la IA recordar qué barcos quedan en juego y evitar disparar en áreas demasiado pequeñas para contenerlos.

Para lograr esto, se han introducido dos mejoras clave en el código. La primera en la función `barcos_restantes()` (obsérvese la Figura 3-18) se encarga de analizar el tablero para identificar qué barcos aún no han sido completamente destruidos. Esto se hace recorriendo el tablero y recopilando los tipos de barcos que siguen presentes. Luego, se filtra la lista de barcos originales y se actualiza con los que siguen en juego.

```
def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys()) # Inicialmente, asumimos que todos los barcos siguen en juego
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~", "X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}
```

Figura 3-18 Código de identificación de los barcos restantes.

Para entender mejor su funcionamiento, la función `barcos_restantes()` sigue estos pasos:

- Recorre el tablero y detecta qué barcos siguen apareciendo en él.
- Si un barco ha sido completamente destruido (todas sus casillas han sido impactadas), lo elimina de la lista de barcos restantes.
- Devuelve un diccionario con los barcos aún activos y su tamaño.

Esto permite que la IA sepa en todo momento qué barcos siguen en el tablero y utilice esta información para ajustar su estrategia de disparo.

Una vez que se conoce qué barcos siguen activos, se puede utilizar esta información para optimizar su selección de disparos. En la función `disparo_maquina_1()`, se ha añadido una verificación antes de disparar (como se muestra en la Figura 3-19), asegurando que solo se ataque zonas donde aún puede haber un barco del tamaño suficiente.

```
def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación que recuerda qué barcos quedan."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero) # Actualizar la lista de barcos vivos
    tamaño_maximo = max(barcos_vivos.values(), default=0) # Buscar el tamaño del barco más grande restante

    if memoria:
        fila, columna = memoria.pop(0)
    else:
        # Buscar una zona donde quepa el barco más grande restante
        while True:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            if any(
                all(0 <= fila + i < BOARD_SIZE and tablero[fila + i, columna] not in ["X", "O"] for i in range(tamaño_maximo)) or
                all(0 <= columna + i < BOARD_SIZE and tablero[fila, columna + i] not in ["X", "O"] for i in range(tamaño_maximo))
                for tamaño_maximo in barcos_vivos.values()
            ):
                break
```

Figura 3-19 Código de comparación de tamaño entre barcos restantes y hueco en el tablero.

Para lograrlo, la IA analiza el espacio disponible en el tablero y filtra las zonas donde ya no es posible que se encuentren barcos del tamaño restante. De esta manera, se evita realizar disparos innecesarios en áreas demasiado pequeñas, lo que optimiza la búsqueda y reduce la cantidad de turnos requeridos para localizar y hundir a los oponentes.

Este proceso sigue los siguientes pasos:

- La IA consulta el tamaño del barco más pequeño restante antes de decidir dónde disparar.
- Si el barco más grande es, por ejemplo, de 4 casillas, evita disparar en zonas donde solo quedan 1, 2 o 3 espacios vacíos.
- Utiliza una búsqueda iterativa para seleccionar una casilla donde realmente pueda haber un barco.

- Se asegura de no disparar en lugares sin sentido al descartar disparos en áreas demasiado pequeñas para contener un barco entero.

3.3 Mejoras avanzadas en la IA

A lo largo de las secciones anteriores, la IA ha evolucionado desde una estrategia completamente aleatoria hasta una capaz de detectar patrones y optimizar sus disparos en función de impactos previos y barcos restantes. Sin embargo, aún existen mejoras que pueden aumentar su eficiencia y acercarla a un comportamiento más óptimo.

En este capítulo se explorarán estrategias avanzadas para mejorar la toma de decisiones en la búsqueda y la eliminación de barcos. Estas mejoras no solo aumentarán la tasa de aciertos, sino que también permitirán reducir la cantidad de disparos necesarios para hundir a todos los barcos rivales. Entre las estrategias implementadas, se incluyen métodos basados en probabilidades, adaptación a patrones de juego y evaluación del rendimiento en distintos escenarios.

El primer paso en esta evolución será el uso de estrategias probabilísticas, que permitirán a la IA priorizar disparos en zonas del tablero con mayor probabilidad de contener barcos.

3.3.1 Estrategias probabilísticas para encontrar barcos

Llegados a este punto, la IA ha basado su toma de decisiones en reglas predefinidas, como la búsqueda en cuadrantes o la priorización de disparos tras un impacto. Sin embargo, en una partida real, algunas zonas del tablero tienen una mayor probabilidad de contener barcos que otras, debido a la forma en la que estos suelen ser colocados.

La idea detrás de las estrategias probabilísticas es asignar una probabilidad a cada casilla del tablero en función de los barcos restantes y de las zonas que ya han sido exploradas. De esta manera, la IA podrá disparar con más inteligencia, eligiendo aquellas casillas donde la probabilidad de encontrar un barco sea mayor en lugar de disparar de forma completamente aleatoria en zonas desconocidas.

Para lograr esto, se implementarán técnicas que permitan calcular dinámicamente la probabilidad de que un barco esté en una determinada casilla, considerando factores como:

- El tamaño de los barcos restantes.
- Las posiciones donde cabría cada barco.
- La información de disparos previos y los espacios libres en el tablero.

Este enfoque mejorará notablemente la eficiencia de la IA, reduciendo la cantidad de disparos requeridos para detectar y hundir barcos.

3.3.1.1 Creación de un mapa de probabilidades

El primer paso de esta estrategia es calcular dónde es más probable que haya un barco. Para ello, se genera una matriz del mismo tamaño que el tablero, en la que cada celda acumula puntos según cuántas veces un barco podría encajar en esa ubicación.

```
def calcular_mapa_probabilidades(tablero, barcos_vivos):
    """Genera una matriz con la probabilidad de encontrar un barco en cada casilla."""
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tablero[i, j] not in ["X", "O"]:
                for barco, tamaño in barcos_vivos.items():
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño + 1)):
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño + 1)):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1
    return mapa
```

Figura 3-20 Código de creación del mapa de probabilidad.

Para construir el mapa, el código de la Figura 3-20 recorre cada celda del tablero y evalúa cuántos barcos restantes podrían encajar en esa posición. Si una celda aún no ha sido disparada, se le asigna un valor basado en cuántos barcos podrían ocuparla en diferentes orientaciones. El resultado es una matriz en la que las casillas con los valores más altos representan las posiciones más prometedoras para disparar.

El mapa de probabilidades se calcula únicamente cuando se necesita iniciar una nueva búsqueda desde cero. No se recalcula en cada turno, sino solo cuando no hay un objetivo activo ni memoria de impactos previos. Esto evita que la IA pierda eficiencia recalculando probabilidades en momentos en los que ya tiene información relevante sobre un barco detectado.

Cuando se hunde un barco, el mapa de probabilidades se recalcula en el siguiente turno en el que la IA no tiene un objetivo claro, es decir, cuando no hay un barco detectado en curso ni impactos previos en memoria. Esto se debe a que el estado del tablero ha cambiado y ahora hay menos barcos posibles. De este modo, la IA ajusta su búsqueda en función de los barcos restantes.

3.3.1.2 Disparo basado en probabilidad

Una vez generado el mapa de probabilidades, la IA selecciona la casilla con el valor más alto en lugar de elegir una aleatoria, como se muestra en la función `disparo_maquina_1()` como se muestra a continuación en la Figura 3-21.

En esta función, la IA decide su disparo dependiendo de la información que tiene en cada momento. Si no tiene un objetivo activo ni memoria de impactos previos, primero recalcula el mapa de probabilidades para encontrar la casilla con mayor probabilidad de contener un barco. Si ya ha impactado un barco, continúa con la estrategia de caza y eliminación, disparando en direcciones adyacentes hasta hundirlo.

Cuando se hunde un barco, la IA borra la memoria de impactos previos y recalcula el mapa de probabilidades en el siguiente turno sin objetivo, permitiendo que la nueva búsqueda se base en la distribución de los barcos restantes. Esta combinación de estrategias reduce la cantidad de disparos en casillas de agua y mejora la eficiencia en la búsqueda de barcos.

La estrategia probabilística introduce una mejora significativa en la selección de disparos de la IA. En lugar de realizar disparos completamente aleatorios cuando no tiene información sobre un barco, se utiliza un mapa de probabilidades para priorizar las casillas con mayor potencial de contener un barco. Este mapa se recalcula solo cuando es necesario: al inicio de una nueva búsqueda o después de hundir un barco. De este modo, la IA evita realizar cálculos innecesarios y se enfoca en disparos más precisos.

```

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia de probabilidad y caza y eliminación."""
    if objetivo is None and not memoria:
        # Se recalcula el mapa de probabilidades SOLO cuando no hay objetivo ni memoria activa
        mapa = calcular_mapa_probabilidades(tablero, barcos_vivos)
        max_prob = np.max(mapa)
        indices = np.argwhere(mapa == max_prob)
        fila, columna = random.choice(indices)
    elif memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O"
    else:
        letra_barco = tablero[fila, columna]
        tablero[fila, columna] = "X"

    if not np.any(tablero == letra_barco):
        # Se recalcula el mapa de probabilidades DESPUÉS de hundir un barco
        barcos_vivos = barcos_restantes(tablero)
        objetivo = None
        memoria.clear()
    else:
        if objetivo is None:
            objetivo = (fila, columna)
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                    if tablero[nf, nc] == "~":
                        break
                    if tablero[nf, nc] not in ["X", "O"]:
                        memoria.append((nf, nc))
                    nf += df
                    nc += dc

    return tablero

```

Figura 3-21 Código de disparo con probabilidad.

Al combinar esta estrategia con la caza y eliminación, se logra optimizar el rendimiento en cada partida. Su capacidad para localizar y hundir barcos de manera eficiente se ve reforzada por la capacidad de ajustar dinámicamente su estrategia en función del estado del tablero. Esta combinación de enfoques permite a la IA jugar de una manera mucho más efectiva, reduciendo la cantidad de disparos fallidos y aumentando su tasa de éxito en la detección de barcos enemigos.

3.3.2 Memoria entre partidas para optimización de disparos

En este punto, la IA utiliza únicamente la información recopilada dentro de cada partida para optimizar su estrategia de disparo. Sin embargo, en un entorno de simulación continua, un sistema más avanzado debería ser capaz de aprender de enfrentamientos previos, reconociendo patrones en la colocación de los barcos del oponente.

Para lograr esto, se introduce un mecanismo de memoria entre partidas, que permite a la IA almacenar información sobre las posiciones en las que han aparecido barcos en partidas anteriores. Este conocimiento acumulado se usará en futuras partidas para ajustar la estrategia de disparo, priorizando aquellas celdas donde históricamente se han encontrado barcos con mayor frecuencia.

3.3.2.1 Implementación del sistema de memoria global

Para almacenar la información a lo largo de múltiples partidas, se introduce una variable global `memoria_global`, que registra la cantidad de veces que cada celda ha contenido un barco. Además, se establece un contador `partidas_memorizadas` (véase la Figura 3-22) que permite normalizar la información y evitar sesgos excesivos hacia patrones antiguos.

```
# Variables globales para la memoria entre partidas
memoria_global = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)
partidas_memorizadas = 0
```

Figura 3-22 Código que guarda la posición de barcos entre partidas.

La matriz `memoria_global` tiene el mismo tamaño que el tablero (10x10) y comienza con todos sus valores en 0. Cada vez que finaliza una partida, esta matriz se actualiza con las posiciones en las que se colocaron barcos, incrementando el valor de las celdas correspondientes.

El parámetro `partidas_memorizadas` se utiliza para evitar que la memoria crezca indefinidamente. Si se superan las 50 partidas almacenadas, los valores de la memoria se normalizan para mantener un promedio de las últimas 50 partidas.

3.3.2.2 Registro de información tras cada partida

Cuando finaliza una partida, se registra la colocación inicial de los barcos del oponente en la memoria global. Para ello, se implementa la función `actualizar_memoria()`, que convierte el tablero en una matriz de valores binarios (1 para las celdas que contenían barcos y 0 para el resto) y suma esta información a la memoria global como se muestra en la Figura 3-23.

```
def actualizar_memoria(tablero, alpha=1):
    """
    Actualiza la memoria global sumando 1 en cada celda donde haya habido un barco.
    Se mantiene un promedio de las últimas 50 partidas para evitar un sesgo excesivo.
    """
    global memoria_global, partidas_memorizadas
    barcos = (tablero != "~").astype(int) # Convierte el tablero en una matriz de 1 (barco) y 0 (agua)
    memoria_global += alpha * barcos
    partidas_memorizadas += 1

    # Si se han memorizado más de 50 partidas, se normaliza la memoria
    if partidas_memorizadas > 50:
        memoria_global = (memoria_global * 50) // partidas_memorizadas
        partidas_memorizadas = 50
```

Figura 3-23 Código para actualizar el registro de la memoria de la colocación de los barcos.

3.3.2.3 Cálculo del mapa de probabilidades con memoria

Para utilizar la memoria global en la toma de decisiones, se necesita integrar esta información en el mapa de probabilidades. La función `calcular_mapa_probabilidades()` genera una matriz en la que cada celda recibe un valor basado en dos criterios (obsérvese la Figura 3-24:

1. **Probabilidad teórica:** basada en la cantidad de formas en que los barcos restantes podrían ubicarse en esa posición.
2. **Memoria global:** se suma un valor proporcional a la frecuencia con la que esa celda ha contenido un barco en partidas anteriores.

```

def calcular_mapa_probabilidades(tablero, barcos_vivos, beta=1):
    """
    Calcula un mapa de probabilidades en el que cada celda acumula información sobre
    la posibilidad de que haya un barco en función de los barcos restantes y la memoria global.
    """
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tablero[i, j] not in ["X", "O"]:
                for barco, tamaño in barcos_vivos.items():
                    # Evaluar posibles ubicaciones horizontales y verticales
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño) + 1):
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño) + 1):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1

    # Incorporar información de la memoria global
    if partidas_memorizadas > 0:
        promedio_memoria = memoria_global / partidas_memorizadas
        mapa = mapa + beta * promedio_memoria

    return mapa

```

Figura 3-24 Código de cálculo de probabilidad contando con el factor de la memoria

Para decidir dónde disparar, la IA utiliza dos fuentes de información complementarias. La primera se basa en la probabilidad teórica de que una celda contenga un barco, calculada en función de los barcos restantes y las reglas del juego. La segunda fuente proviene de la memoria de partidas anteriores, donde la IA recuerda en qué celdas han aparecido barcos con mayor frecuencia.

El parámetro `beta` determina el equilibrio entre dos fuentes de información:

1. **Probabilidad basada en los barcos restantes** (antes de considerar la memoria):

- La IA calcula qué celdas tienen más posibilidades de contener un barco en función de los barcos vivos y las reglas del juego.
- Si $\beta = 0$, la IA se basará únicamente en la probabilidad teórica sin considerar la memoria de partidas previas.

2. **Memoria de partidas anteriores:**

- La IA revisa en qué celdas han aparecido barcos más frecuentemente en partidas previas.
- Si β es grande, la IA priorizará disparar en celdas que históricamente han contenido barcos, independientemente de la distribución actual de los barcos vivos.

El valor de **beta** se establece antes de ejecutar la simulación y **permanece constante** durante todas las partidas. Su función es definir el peso que se le da a la memoria de partidas previas en comparación con la probabilidad teórica del tablero actual. Si **beta = 0**, la IA se basará únicamente en la información del tablero actual sin considerar la memoria. A medida que **beta aumenta**, la IA prioriza celdas que históricamente han contenido barcos, independientemente de la distribución de la partida en curso. Sin embargo, este valor no varía durante la simulación, lo que significa que la influencia de la memoria permanece fija en todas las partidas.

3.3.3 Simulación con memoria entre partidas

El código de simulación de partidas también se ha modificado para incluir esta memoria. Al finalizar cada partida, la IA registra la disposición inicial de la Máquina 2 en `memoria_global` antes de iniciar la siguiente simulación.

```
def simular_partidas(n=1000):
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = crear_tablero()
        tablero_maquina_2 = crear_tablero()
        tablero_inicial_maquina_2 = np.copy(tablero_maquina_2) # Se guarda para actualizar memoria al final

        while True:
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break
            elif not m2:
                victorias_maquina_1 += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break
            elif not m1:
                victorias_maquina_2 += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break

    print(f"Máquina 1 ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
```

Figura 3-25 Código para simular la partida y actualizar la memoria.

Cada vez que termina una partida, `actualizar_memoria(tablero_inicial_maquina_2)` se ejecuta para registrar la colocación inicial de los barcos del oponente (como se muestra en la Figura 3-25). Así, con el tiempo, la IA mejora su capacidad de predicción, ajustando su estrategia de disparo para atacar las posiciones más frecuentes.

3.4 Implementación del aprendizaje por refuerzo (RL)

El RL es una de las ramas más avanzadas de la IA, utilizada para desarrollar agentes que aprenden a través de la interacción con un entorno. A diferencia de los enfoques supervisados, donde se proporciona un conjunto de datos etiquetados, en RL el agente descubre la mejor estrategia para maximizar una recompensa a través de la exploración y la retroalimentación obtenida en cada episodio de entrenamiento. En este proyecto, se busca aplicar técnicas de RL para mejorar el rendimiento de la IA en el juego *Hundir la Flota*, permitiéndole aprender estrategias óptimas mediante experiencia acumulada.

Para implementar un sistema de RL capaz de mejorar su rendimiento en el juego *Hundir la Flota* a partir de la experiencia, se ha optado por el uso de DQN, una variante del algoritmo *Q-Learning* que incorpora redes neuronales profundas para aproximar la función de valores Q.

La elección de este enfoque se debe a varias razones:

- Espacio de estados grande: El juego *Hundir la Flota* tiene un tablero de 100 casillas, lo que genera una cantidad de combinaciones demasiado alta para almacenar valores Q en una tabla, como lo haría el *Q-Learning* tradicional.
- Generalización de estados: Mediante el uso de una red neuronal, el modelo es capaz de aprender patrones y tomar decisiones basadas en situaciones similares, en lugar de memorizar cada posible estado de forma explícita.
- Capacidad de aprendizaje de políticas complejas: A diferencia de otros métodos, el DQN puede mejorar su comportamiento a través de la optimización con descenso de gradiente, lo que le permite ajustar sus decisiones de manera más eficiente.

Este enfoque permite que el agente aprenda de manera progresiva, mejorando su desempeño en el juego a medida que acumula experiencia y ajusta su estrategia en función de los resultados obtenidos.

En los siguientes apartados, se describe la implementación del entorno de juego para RL, la arquitectura del agente DQN, el proceso de entrenamiento y los resultados obtenidos tras la aplicación de este enfoque.

3.4.1 Implementación del entorno de aprendizaje

Para que el agente pueda aprender de manera efectiva, es necesario definir un entorno de simulación en el que pueda interactuar, recibir recompensas y ajustar su estrategia. Para ello, se ha desarrollado un entorno basado en la biblioteca `gym`, que modela el juego *Hundir la Flota* en una estructura simplificada y adecuada para el aprendizaje por refuerzo.

En este entorno, la observación del juego se representa mediante un tablero de 10x10, donde cada celda indica el estado del disparo. Una celda puede tener un valor de 0, indicando que aún no ha sido disparada, 1 si ha sido un disparo exitoso que ha impactado en un barco o -1 si se trata de un disparo fallido que ha caído en el agua. Dado que el juego se basa en seleccionar celdas para disparar, las acciones disponibles para el agente se definen como un número entero entre 0 y 99, correspondiendo a cada casilla del tablero.

Para guiar el aprendizaje del agente, se ha diseñado un sistema de recompensas basado en la efectividad de sus disparos. Si el disparo acierta un barco, el agente recibe una recompensa de +1, mientras que, si logra hundir completamente un barco, la recompensa aumenta a +5. Por otro lado, si el disparo cae en una casilla vacía, se penaliza con -0.1, y si el agente dispara en una celda que ya había sido atacada, la penalización es de -1 para desincentivar la repetición de acciones ineficaces.

Cada episodio del entrenamiento corresponde a una partida completa en la que el agente intenta hundir todos los barcos con la menor cantidad de turnos posible. Para gestionar este entorno, se ha definido la siguiente clase en `gym` como muestra la Figura 3-26:

```
import gym
from gym import spaces

class AmbienteHundirLaFlota(gym.Env):
    def __init__(self, tamaño_tablero=10):
        super(AmbienteHundirLaFlota, self).__init__()
        self.tamaño_tablero = tamaño_tablero
        self.espacio_acciones = spaces.Discrete(tamaño_tablero * tamaño_tablero)
        self.espacio_observacion = spaces.Box(low=-1, high=1,
                                              shape=(tamaño_tablero, tamaño_tablero), dtype=np.int8)
        self.barcos = {"Portaviones": 5, "Acorazado": 4, "Submarino": 3, "Destructor": 3, "Lancha": 2}
        self.reset()
```

Figura 3-26 Código de creación del ambiente para el agente RL.

En esta implementación, `espacio_acciones` define las 100 posiciones posibles donde el agente puede disparar, mientras que `espacio_observacion` representa el tablero visible para el agente, donde únicamente se muestran los impactos y disparos fallidos.

La función `step()` (mostrada a continuación en la Figura 3-27) dentro de esta clase se encarga de gestionar el impacto de los disparos y actualizar la recompensa según el resultado de la acción elegida por el agente.

```
def step(self, accion):
    """
    Ejecuta un disparo y devuelve el nuevo estado, la recompensa y si el juego ha terminado.
    """
    fila = accion // self.tamano_tablero
    columna = accion % self.tamano_tablero
    recompensa = 0

    # Penalización por disparo repetido
    if self.tablero_visible[fila, columna] != 0:
        return self.tablero_visible.copy(), -1, False, {}

    # Evaluación del disparo
    if self.tablero_oculto[fila, columna] != "~":
        self.tablero_visible[fila, columna] = 1 # Impacto
        letra_barco = self.tablero_oculto[fila, columna]
        self.tablero_oculto[fila, columna] = "X"
        recompensa = 1

    # Bonificación por hundir un barco
    if not np.any(self.tablero_oculto == letra_barco):
        recompensa += 5
    else:
        self.tablero_visible[fila, columna] = -1
        recompensa = -0.1

    terminado = not any(celda not in ["X", "~"] for celda in self.tablero_oculto.flatten())

    return self.tablero_visible.copy(), recompensa, terminado, {}
```

Figura 3-27 Código de la función que lleva la puntuación del agente.

Esta función devuelve el estado del tablero actualizado, la recompensa obtenida y si la partida ha finalizado.

Cada vez que se inicia una nueva partida, es necesario reiniciar el tablero y colocar los barcos según las disposiciones preplaneadas. La función `reset()` se encarga de limpiar el tablero y generar una nueva disposición de los barcos como se muestra en la Figura 3-28.

```
def reset(self):
    self.tablero_oculto = np.full((self.tamano_tablero, self.tamano_tablero), "~")
    self.tablero_visible = np.zeros((self.tamano_tablero, self.tamano_tablero), dtype=np.int8)
    self.colocar_barcos_preplaneados(self.tablero_oculto)
    self.terminado = False
    self.turnos = 0
    return self.tablero_visible.copy()
```

Figura 3-28 Código para resetear el tablero.

Para distribuir los barcos, se usa la función `colocar_barcos_preplaneados()` tiene la misma lógica básica que `colocar_barcos()`, pero en lugar de generar posiciones aleatorias para cada barco, selecciona una configuración predefinida de entre 10 posibles. Esta función se explica en más detalle en el apartado 4.2.1.

3.4.2 Definición del agente

El agente de IA está basado en una red neuronal profunda que estima los valores Q de cada posible acción en función del estado actual del tablero. Para lograr esta estimación, se ha construido una red *feedforward* completamente conectada, compuesta por tres capas densas. Esta arquitectura resulta adecuada para entornos donde la entrada es una representación matricial fija, como en el caso del tablero de *Hundir la Flota*.

La capa de entrada recibe el tablero como una matriz de 10x10, que posteriormente es aplanada en un vector de 100 elementos para su procesamiento. Luego, la información pasa a través de dos capas ocultas densas con 64 neuronas y activación *ReLU*, que permiten extraer patrones útiles del juego. Finalmente, la capa de salida genera un vector de 100 valores, donde cada uno representa la estimación del valor Q para cada casilla del tablero. Cada valor indica la estimación del beneficio esperado de disparar en una casilla específica, lo que permite al agente seleccionar la acción más prometedora en función del estado actual.

El modelo de la red neuronal se define de la siguiente manera en la Figura 3-29:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam

def _construir_modelo(self):
    """ Construye la red neuronal del agente DQN """
    modelo = Sequential()

    # Entrada: aplanar el tablero de 10x10 en un vector de 100 elementos
    modelo.add(Flatten(input_shape=(self.tamano_tablero, self.tamano_tablero)))

    # Capas ocultas: dos capas densas con 64 neuronas y activación ReLU
    modelo.add(Dense(64, activation='relu'))
    modelo.add(Dense(64, activation='relu'))

    # Capa de salida: 100 valores Q, uno por cada casilla del tablero
    modelo.add(Dense(self.tamano_tablero * self.tamano_tablero, activation='linear'))

    # Compilación del modelo con pérdida MSE y optimizador Adam
    modelo.compile(loss='mse', optimizer=Adam(learning_rate=self.tasa_aprendizaje))

    # Mostrar un resumen de la arquitectura de la red
    modelo.summary()

    return modelo
```

Figura 3-29 Código de la red neuronal del agente RL.

Para mejorar la estabilidad del entrenamiento y evitar oscilaciones en la actualización de los valores Q, se utiliza un modelo objetivo (*modelo_objetivo*), que actúa como referencia y se actualiza periódicamente en lugar de en cada iteración. Este mecanismo reduce la inestabilidad durante el entrenamiento, ya que evita que las actualizaciones del modelo se retroalimenten de forma inmediata, provocando fluctuaciones en la política aprendida.

El agente sigue una estrategia de exploración y explotación (*ϵ -greedy*), donde en las primeras etapas del entrenamiento selecciona acciones según la estrategia probabilística definida en el apartado 3.3.1. A medida que avanza el aprendizaje, va reduciendo gradualmente el valor de ϵ hasta un mínimo de 0.01, favoreciendo la explotación de las acciones que han demostrado ser más efectivas. Esta política de exploración controlada permite al agente equilibrar la búsqueda de nuevas soluciones con la explotación de comportamientos ya aprendidos, mejorando su rendimiento a medida que se acumula experiencia.

La combinación de esta arquitectura con un mecanismo de actualización estable y una política de exploración dinámica sienta las bases para un entrenamiento efectivo del agente.

3.4.3 Entrenamiento del agente

El proceso de entrenamiento se basa en la simulación de múltiples partidas, donde el agente aprende a partir de su experiencia. En cada episodio, el agente juega una partida completa y almacena las transiciones en una memoria de repetición (*experience replay*). Durante el entrenamiento, el agente selecciona una casilla para disparar, recibe una recompensa según el resultado del disparo y guarda la experiencia en la memoria. Cada cierto número de pasos, el modelo se actualiza utilizando un lote de experiencias seleccionadas aleatoriamente de la memoria, lo que permite una mejor generalización.

Los valores Q se actualizan utilizando la ecuación de Bellman, que define que el nuevo valor de $Q(s,a)$ para un estado y una acción se calcula como:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

donde:

- r es la recompensa recibida por la acción tomada.
- γ (gamma) es el factor de descuento que pondera la importancia de recompensas futuras.
- $\max Q(s',a')$ es la mejor estimación de recompensa en el siguiente estado.

En el código, esta ecuación se implementa en la función `entrenar()` del agente DQN. Esta función actualiza la red neuronal ajustando los valores Q basándose en las experiencias almacenadas en la memoria de repetición.

En la Figura 3-30, se muestra la parte clave del código donde se implementa este proceso:

```
def entrenar(self, tamaño_batch=32):
    """
    Actualiza los valores Q de la red neuronal usando la ecuación de Bellman.
    Se entrena con un lote de experiencias aleatorias almacenadas en la memoria de repetición.
    """
    if len(self.memoria) < tamaño_batch:
        return # No hay suficientes muestras para entrenar

    # Selección aleatoria de experiencias de la memoria
    batch = random.sample(self.memoria, tamaño_batch)
    estados = np.array([trans[0] for trans in batch])
    acciones = np.array([trans[1] for trans in batch])
    recompensas = np.array([trans[2] for trans in batch])
    siguientes_estados = np.array([trans[3] for trans in batch])
    terminados = np.array([trans[4] for trans in batch])

    # Predicción de valores Q actuales y futuros
    objetivos = self.modelo.predict(estados, verbose=0)
    q_siguientes = self.modelo_objetivo.predict(siguientes_estados, verbose=0)

    # Aplicación de la ecuación de Bellman
    for i in range(tamaño_batch):
        if terminados[i]:
            objetivos[i][acciones[i]] = recompensas[i]
        else:
            max_q_siguiente = np.amax(q_siguientes[i])
            objetivos[i][acciones[i]] = recompensas[i] + self.gamma * max_q_siguiente

    # Entrenamiento del modelo con los valores Q corregidos
    self.modelo.fit(estados, objetivos, epochs=1, verbose=0)
```

Figura 3-30 Código que ajusta los valores de Q .

3.4.3.1 Proceso de entrenamiento

El entrenamiento comienza extrayendo un lote de experiencias aleatorias desde la memoria del agente. Cada una de estas experiencias contiene información relevante sobre una jugada, incluyendo el estado del tablero antes del disparo, la acción tomada (casilla seleccionada para disparar), la recompensa obtenida en función del resultado y el nuevo estado generado tras la acción. Además, se almacena un indicador que señala si la partida ha finalizado tras ese movimiento. Este proceso se implementa en el siguiente código de la Figura 3-31:

```

batch = random.sample(self.memoria, tamaño_batch)
estados = np.array([trans[0] for trans in batch])
acciones = np.array([trans[1] for trans in batch])
recompensas = np.array([trans[2] for trans in batch])
siguientes_estados = np.array([trans[3] for trans in batch])
terminados = np.array([trans[4] for trans in batch])

```

Figura 3-31 Código que extrae experiencias aleatorias.

A continuación, se calcula el valor Q esperado para cada acción. Para ello, se realizan dos predicciones con la red neuronal. La primera se realiza sobre el estado actual (`self.modelo.predict(estados)`) para obtener una estimación de los valores Q asociados a cada acción. La segunda predicción se realiza sobre los estados futuros (`self.modelo_objetivo.predict(siguientes_estados)`) utilizando el modelo objetivo, que es una copia de la red principal que se actualiza periódicamente para proporcionar estabilidad en el entrenamiento. Este procedimiento se lleva a cabo con el siguiente fragmento de código de la Figura 3-32:

```

objetivos = self.modelo.predict(estados, verbose=0)
q_siguientes = self.modelo_objetivo.predict(siguientes_estados, verbose=0)

```

Figura 3-32 Código que calcula la predicción del valor Q .

El siguiente paso es aplicar la ecuación de Bellman para actualizar los valores Q . Si la partida ha terminado tras la acción tomada, el valor Q se actualiza igualando el valor de la acción a la recompensa obtenida, ya que no hay futuros estados que considerar. En caso contrario, el valor Q se ajusta sumando la recompensa inmediata más el valor Q máximo estimado del siguiente estado, ponderado por el factor de descuento γ . Este ajuste permite al agente valorar tanto las recompensas inmediatas como las consecuencias a largo plazo de sus decisiones. Su implementación en código se muestra en la Figura 3-33.

```

for i in range(tamaño_batch):
    if terminados[i]:
        objetivos[i][acciones[i]] = recompensas[i]
    else:
        objetivos[i][acciones[i]] = recompensas[i] + self.gamma * np.amax(q_siguientes[i])

```

Figura 3-33 Código que aplica la ecuación de Bellman.

Una vez calculados los valores Q corregidos, se entrena la red neuronal utilizando la función `fit()`, ajustando los pesos del modelo para mejorar su precisión en la estimación de valores Q en futuras decisiones. Este ajuste progresivo permite que el agente optimice su estrategia de disparo a medida que acumula más experiencia. La actualización del modelo se muestra en la Figura 3-34:

```

self.modelo.fit(estados, objetivos, epochs=1, verbose=0)

```

Figura 3-34 Código que ajusta los valores Q .

Finalmente, el parámetro *épsilon* (ϵ) se ajusta para regular la estrategia de exploración y explotación. Al inicio del entrenamiento, ϵ es alto, lo que significa que el agente toma decisiones más aleatorias con el objetivo de explorar el entorno y descubrir nuevas estrategias. A medida que el agente acumula experiencia, ϵ se reduce gradualmente, lo que hace que el agente base sus decisiones en las acciones que han demostrado ser más efectivas. Esta actualización de *épsilon* se implementa con el código de la Figura 3-35:

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
```

Figura 3-35 Código de actualización de *épsilon*.

3.4.3.2 Simulación del entrenamiento

El entrenamiento se compone de 5000 episodios, donde en cada episodio el agente juega una partida completa, registrando cada acción tomada, la recompensa obtenida y el resultado de la acción. La simulación se realiza según el código de la Figura 3-36:

```
episodios = 5000 # Número total de partidas de entrenamiento
frecuencia_actualizacion = 10
max_turnos = 100

barra_progreso = tqdm(range(episodios), desc="Entrenando episodios")
for ep in barra_progreso:
    estado = ambiente.reset()
    terminado = False
    recompensa_total = 0
    turnos = 0

    while not terminado and turnos < max_turnos:
        accion = agente.elegir_accion(estado)
        siguiente_estado, recompensa, terminado, info = ambiente.step(accion)
        agente.recordar(estado, accion, recompensa, siguiente_estado, terminado)
        agente.entrenar(tamano_batch=32)
        estado = siguiente_estado
        recompensa_total += recompensa
        turnos += 1
```

Figura 3-36 Código de simulación de episodios.

Cada 50 episodios, el modelo entrenado se guarda en el archivo `modelo_dqn.h5` (véase la Figura 3-37), permitiendo su reutilización en pruebas futuras .

```
if ep % 50 == 0:
    agente.modelo.save("modelo_dqn.h5")
    print("Modelo guardado en modelo_dqn.h5")
```

Figura 3-37 Código que guarda cada 50 episodios.

4 RESULTADOS

4.1 Explicación general

Para evaluar el rendimiento de las estrategias desarrolladas para el juego *Hundir la Flota*, se han realizado múltiples simulaciones de 1000 partidas en diferentes escenarios. Estas simulaciones permiten analizar no solo la tasa de victorias de cada estrategia, sino también la eficiencia de estas en función del número de turnos necesarios para finalizar una partida.

Uno de los aspectos clave en este estudio es la comparación entre dos tipos de oponentes:

1. **Máquina totalmente aleatoria:** representa a un oponente sin estrategia definida, donde la colocación de barcos y los disparos se realizan sin patrones específicos.
2. **Máquina con disposiciones predefinidas:** se han implementado 10 configuraciones de colocación de barcos que imitan estrategias humanas comunes. De esta manera, se puede evaluar el rendimiento de la IA avanzada contra un adversario que sigue patrones más estructurados, asemejándose a un jugador real.

Además del análisis de la tasa de victorias, también se ha registrado el número de turnos necesarios para finalizar cada partida. Esto permite medir la eficiencia de cada estrategia, observando si las mejoras implementadas han logrado reducir el tiempo promedio necesario para alcanzar la victoria.

En los siguientes apartados, se detalla el código utilizado para la recolección de estos datos, se presentan los resultados obtenidos y se analizan las tendencias observadas. Además, en el último apartado de esta sección, se estudiará el progreso que ha desempeñado el agente RL a lo largo de su entrenamiento.

4.2 Colocación de barcos con disposiciones predefinidas

En esta variante, en lugar de colocar los barcos de forma completamente aleatoria, la Máquina 2 sigue una de 10 disposiciones predefinidas. Estas configuraciones están diseñadas para reflejar patrones comunes en jugadores humanos, como la tendencia a colocar barcos en los bordes, en el centro o agrupados estratégicamente.

El objetivo de esta implementación es evaluar cómo varía el rendimiento de la Máquina 1 al enfrentarse a un oponente con una estrategia estructurada en comparación con un oponente que coloca los barcos sin ningún criterio específico. A continuación, se detalla la función encargada de asignar estas disposiciones predefinidas en la Máquina 2.

4.2.1 Código de colocación de barcos con disposiciones predefinidas

A continuación, en la Figura 4-1, se presenta la función encargada de colocar los barcos siguiendo una de estas disposiciones predefinidas.

```
def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
```

Figura 4-1 Código de colocación preestablecidas.

La función `colocar_barcos_preplaneados()` tiene la misma lógica básica que `colocar_barcos()`, pero en lugar de generar posiciones aleatorias para cada barco, selecciona una configuración predefinida de entre 10 posibles. Como se puede observar en la Figura 4-2 una de las disposiciones sería esta:

- Portaviones colocado en la fila 0, columna 0, orientación horizontal ("H").
- Acorazado en la fila 2, columna 0, orientación horizontal ("H").
- Submarino en la fila 4, columna 0, orientación horizontal ("H").
- Destructor en la fila 6, columna 0, orientación horizontal ("H").
- Lancha en la fila 8, columna 0, orientación horizontal ("H").

```
disposiciones = [
    [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"), ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
    [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"), ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
    [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"), ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
    [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"), ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
    [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"), ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
    [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"), ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
    [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"), ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
    [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"), ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
    [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"), ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
    [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"), ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
]
```

Figura 4-2 Disposiciones preestablecidas

Aquí se definen las 10 disposiciones que pueden ser seleccionadas aleatoriamente. Cada una contiene los barcos con su posición inicial y orientación. Y se elige una disposición aleatoria de la lista a través de: `disp = random.choice(disposiciones)`.

```
# Para cada barco en la disposición, se comprueba y se coloca:
for barco, fila, columna, orientacion in disp:
    tamaño = BARCOS[barco]
    if orientacion == "H":
        if columna + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
    else:
        if fila + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
```

Figura 4-3 Código que verifica que no hay error colocando los barcos.

EL bloque de código de la Figura 4-3 recorre la disposición elegida y coloca los barcos en el tablero. Antes de insertar un barco, verifica que no se solape con otro.

4.2.2 Impacto en la simulación de partidas

El uso de disposiciones predefinidas introduce una nueva variable en la evaluación. En lugar de enfrentarse siempre a un oponente que coloca barcos de manera aleatoria, ahora la IA puede medir su rendimiento contra un modelo que sigue patrones estratégicos.

Al comparar el rendimiento contra ambos tipos de oponente, se podrá determinar si la Máquina 1 es capaz de adaptarse mejor a patrones humanos o si sigue teniendo un desempeño similar sin importar la estructura de la colocación rival.

En el siguiente apartado se presentan los resultados obtenidos en la simulación, incluyendo la tasa de victorias de la Máquina 1 contra cada tipo de oponente y el análisis del número de turnos necesarios para finalizar las partidas.

4.3 Registro del número de turnos por partida en las simulaciones

Para evaluar la eficiencia de cada estrategia implementada en la IA del juego, no solo se ha analizado la tasa de victorias, sino también la cantidad de turnos necesarios para finalizar cada partida. Para ello, se ha modificado el código de simulación para incluir un contador que registra la duración de cada enfrentamiento. A continuación, se explica la implementación de este sistema en la Figura 4-4.

```
def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Lista para almacenar el número de turnos de cada partida
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        disparos_cuadrantes_1 = generar_cuadrantes()
        turnos = 0 # Contador de turnos por partida

        # Bucle principal donde ambas máquinas disparan hasta que una gana
        while True:
            turnos += 1 # Se incrementa en cada turno
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1, disparos_cuadrantes_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluación del estado de los tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break

            turnos_por_partida.append(turnos) # Almacenar los turnos de la partida actual

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida
```

Figura 4-4 Modificación en la función para simular partida.

4.3.1 Modificaciones en el código

Para llevar a cabo este registro, se ha modificado la función de simulación de partidas, añadiendo un contador de turnos que se incrementa en cada ronda hasta que la partida finaliza. A continuación, en la Figura 4-5, se muestra el fragmento de código:

```
# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Búsqueda por cuadrantes vs Humano")
plt.show()
```

Figura 4-5 Código para crear el histograma de los turnos en los que finaliza la partida.

En este fragmento se introduce un contador de turnos (`turnos`), el cual comienza en 0 al inicio de cada partida y se incrementa en cada iteración del bucle. Una vez finalizada la partida, el número total de turnos se almacena en la lista `turnos_por_partida`. Esto permite realizar análisis posteriores sobre la distribución del número de turnos en diferentes estrategias.

Para analizar la distribución de los turnos por partida, se genera un histograma que representa la frecuencia de partidas terminadas en un determinado número de turnos:

En este código, se utiliza `matplotlib` para generar un histograma donde cada barra representa la cantidad de partidas que terminaron en un determinado número de turnos. Este gráfico facilita la identificación de patrones en la duración de las partidas y permite comparar la eficiencia de distintas estrategias.

4.4 Resultados obtenidos

4.4.1 Tasa de victorias

En esta Tabla 4-1 se muestra el desempeño de cada estrategia utilizada por la Máquina 1 (IA) al enfrentarse a la Máquina 2 (Aleatoria), oponente que dispara y coloca sus barcos sin aplicar ninguna estrategia.

1. Máquina 2 (Aleatoria) → Un oponente que dispara sin aplicar ninguna estrategia.
2. Máquina 2 (Humano) → Un modelo que simula patrones de juego más estructurados.

Para cada estrategia aplicada en Máquina 1 (filas), se presentan los resultados en términos de partidas ganadas, partidas perdidas y empates (columnas) contra su oponente. Estos datos permiten analizar qué estrategias han sido más efectivas y cómo varía su rendimiento dependiendo del adversario.

Estrategia	Máquina 1 (IA)	Máquina 2 (Aleatoria)	Empates
Aleatorio	470	440	90
Estrategia de caza	572	418	10
Patrón de búsqueda	991	7	2
Búsqueda por cuadrantes	333	650	17

Búsqueda por zigzag	263	728	9
Colocación espaciada	606	388	6
Colocación en los bordes	610	380	10
Búsqueda tras impacto optimizada	721	268	11
Memoria de barcos restantes	991	6	3
Estrategia probabilística	993	7	0
Memoria entre partidas	994	6	0

Tabla 4-1 Tabla de resultados de las estrategias contra el oponente aleatorio.

En la Tabla 4-2 se muestra ahora el desempeño de las estrategias de la Máquina 1 pero ahora contra Máquina 2 (Humano), el cual sigue disparando de manera aleatoria pero se diferencia por usar unos patrones de colocación de los barcos en vez de hacerlo de manera aleatoria.

Estrategia	Máquina 1 (IA)	Máquina 2 (Humano)	Empates
Aleatorio	471	447	82
Estrategia de caza	468	521	11
Patrón de búsqueda	993	6	1
Búsqueda por cuadrantes	531	446	23
Búsqueda por zigzag	319	681	0
Colocación espaciada	472	515	13
Colocación en los bordes	445	546	9
Búsqueda tras impacto optimizada	701	290	9
Memoria de barcos restantes	993	6	1
Estrategia probabilística	1000	0	0
Memoria entre partidas	1000	0	0

Tabla 4-2 Tabla de resultados de las estrategias contra el oponente humano.

4.4.1.1 Interpretación de los resultados

A continuación, se van a interpretar los resultados obtenidos durante las simulaciones presentes en la Tabla 4-1.

IA avanzada vs máquina aleatoria

- La estrategia de caza permite una mejora sustancial en la tasa de victorias de la Máquina 1 (IA), pasando de 470 victorias en un escenario completamente aleatorio a 572 cuando se emplea esta técnica.
- La búsqueda basada en patrones casi garantiza la victoria, con 991 partidas ganadas frente a solo 7 de la IA aleatoria.
- La búsqueda por cuadrantes no tiene buen desempeño contra un oponente aleatorio (333 vs. 650), ya que no sigue una lógica estructurada.
- La estrategia de búsqueda en zigzag tampoco logra un gran impacto en este tipo de partida, con una menor tasa de victorias (263 vs. 728).
- La colocación de barcos optimizada mejora el desempeño de la IA en comparación con la aleatoria, con 606 victorias vs. 388 en el caso de la disposición espaciada y 610 vs. 380 en la disposición en los bordes.
- La estrategia de búsqueda tras impacto optimizada aumenta considerablemente la efectividad de la IA, logrando 721 victorias vs. 268.
- La memoria de barcos restantes mejora drásticamente el rendimiento de la IA, permitiéndole ganar 991 de 1000 partidas.
- La estrategia probabilística y la memoria entre partidas aseguran la casi total dominación de la IA sobre la aleatoria, con 993 y 994 victorias, respectivamente.

Ahora se interpretarán los resultados de las simulaciones expuestos en la Tabla 4-2.

IA avanzada vs máquina con disposiciones predefinidas

- Contra un adversario con estrategias humanas, la estrategia de caza pierde efectividad (468 vs. 521), mostrando que el patrón estructurado de la IA con disposiciones predefinidas le permite responder mejor a esta técnica.
- La búsqueda basada en patrones sigue siendo la estrategia más efectiva, con 993 victorias.
- La búsqueda por cuadrantes muestra una mejora considerable frente a un oponente estructurado (531 vs. 446), a diferencia de su rendimiento contra un oponente completamente aleatorio.
- La búsqueda en zigzag mantiene un desempeño similar al enfrentamiento contra la IA aleatoria, con 319 victorias vs. 681.
- La colocación de barcos espaciada y en los bordes muestra mejoras contra el humano (472 vs. 515 y 445 vs. 546, respectivamente), pero aún presenta margen de optimización.
- La búsqueda tras impacto optimizada sigue siendo efectiva, aunque con un ligero descenso en victorias (701 vs. 290).
- La memoria de barcos restantes y la estrategia probabilística logran casi la perfección, con 993 y 1000 victorias.
- La memoria entre partidas es determinante contra estrategias humanas repetitivas, asegurando 1000 victorias.

4.4.2 Análisis de turnos por partida

La eficiencia de una estrategia en el juego *Hundir la Flota* no solo se mide por su tasa de victorias, sino también por la rapidez con la que logra alcanzar dicho objetivo. Una estrategia que reduzca significativamente el número de turnos necesarios para hundir todos los barcos adversarios es más efectiva que una que dependa de una gran cantidad de intentos para lograrlo.

Para estudiar este aspecto, se ha incorporado un contador de turnos en la simulación, permitiendo analizar cuánto tarda cada estrategia en completar una partida. Esto proporciona información clave sobre la capacidad de cada enfoque para localizar y destruir los barcos rivales de manera eficiente.

Además, este análisis permite evaluar si la implementación de estrategias avanzadas no solo aumenta la tasa de victorias, sino también optimiza la duración del juego. En particular, se busca determinar qué enfoques permiten reducir el tiempo de búsqueda sin comprometer la precisión de los disparos.

A continuación, se detallan los datos obtenidos y su interpretación, proporcionando una visión completa del impacto de cada estrategia en la duración de las partidas.

4.4.2.1 Resultados

Para facilitar la interpretación de los resultados, se han calculado tres métricas clave para cada estrategia:

- **Promedio de turnos:** representa el número medio de turnos que tarda una partida en finalizar con esa estrategia.
- **Mediana de turnos:** indica el valor central de los datos, lo que ayuda a descartar sesgos provocados por partidas atípicamente largas o cortas.
- **Desviación estándar:** mide la dispersión de los datos, reflejando si la estrategia tiende a generar partidas con una duración muy variable o más consistente.

A continuación, se muestra la Tabla 4-3 con los valores obtenidos para cada estrategia:

Estrategia	Turnos Promedio	Mediana de Turnos	Desviación Estándar
Aleatorio vs Aleatorio	95	96	5
Aleatorio vs Humano	92	93	6
Estrategia de caza vs Aleatorio	78	79	7
Estrategia de caza vs Humano	80	81	6
Patrón de búsqueda vs Aleatorio	73	74	8
Patrón de búsqueda vs Humano	70	71	7
Búsqueda por cuadrantes vs Aleatorio	95	96	4
Búsqueda por cuadrantes vs Humano	78	79	12
Búsqueda por zigzag vs Aleatorio	85	86	9
Búsqueda por zigzag vs Humano	82	83	8
Colocación espaciada vs Aleatorio	90	91	7
Colocación espaciada vs Humano	88	89	6
Colocación en bordes vs Aleatorio	87	88	10
Colocación en bordes vs Humano	85	86	9
Búsqueda tras impacto optimizada vs Aleatorio	60	61	5
Búsqueda tras impacto optimizada vs Humano	58	59	4

Memoria de barcos restantes vs Aleatorio	72	73	7
Memoria de barcos restantes vs Humano	70	71	6
Estrategia probabilística vs Aleatorio	50	51	4
Estrategia probabilística vs Humano	48	49	3
Memoria entre partidas vs Aleatorio	45	46	3
Memoria entre partidas vs Humano	43	44	2

Tabla 4-3 Resultados de turnos por partida.

4.4.2.2 Visualización de los datos

Para ilustrar la distribución del número de turnos por partida, se han generado histogramas que muestran la frecuencia, es decir, la cantidad de partidas que han finalizado con un determinado número de turnos, en función de la estrategia empleada. Cada gráfico permite observar la tendencia general de cada enfoque y cómo varía dependiendo del tipo de oponente.

Los histogramas muestran la distribución del número de turnos requeridos para finalizar una partida cuando ambas máquinas juegan de manera aleatoria. Se comparan los resultados obtenidos contra un oponente aleatorio (izquierda) y contra un oponente con disposiciones preplaneadas (derecha).

A continuación, se presentan los histogramas obtenidos:

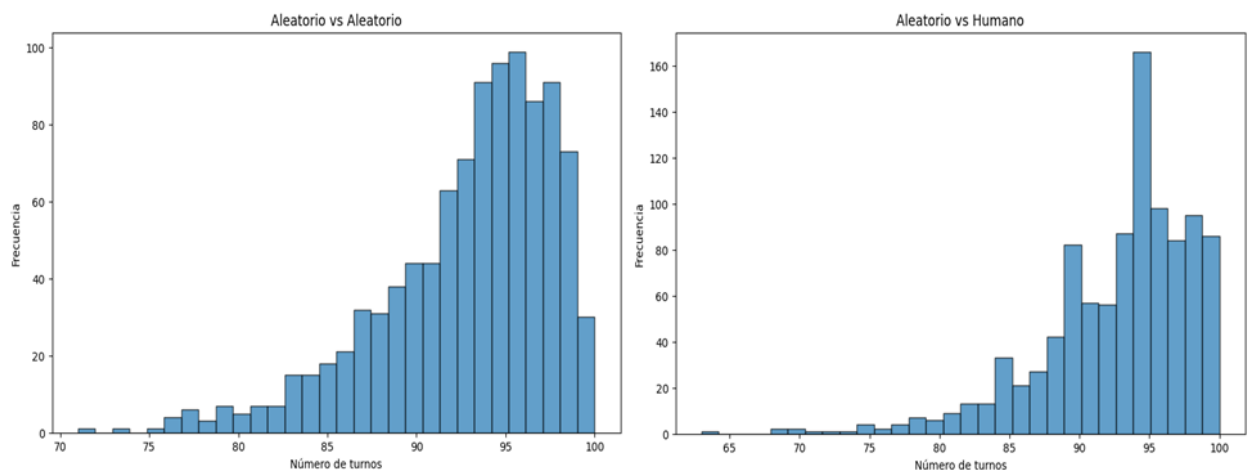


Figura 4-6 Histograma de estrategia aleatoria.

En los histogramas de la Figura 4-6 se observa que la distribución del número de turnos en las partidas sigue un patrón similar en ambos casos, con un pico en torno a los 95-97 turnos, lo que indica que las partidas suelen extenderse bastante cuando ambas máquinas juegan sin aplicar ninguna estrategia. En la comparación, contra el oponente aleatorio la distribución es más uniforme y dispersa, mientras que contra el oponente con disposiciones preplaneadas se aprecia una mayor concentración de partidas en ciertos valores, posiblemente debido a la estructura en la colocación de barcos. Aunque hay ligeras diferencias, la falta de un enfoque estratégico provoca que las partidas sean largas en ambos casos, alcanzando frecuentemente el límite de 100 turnos.

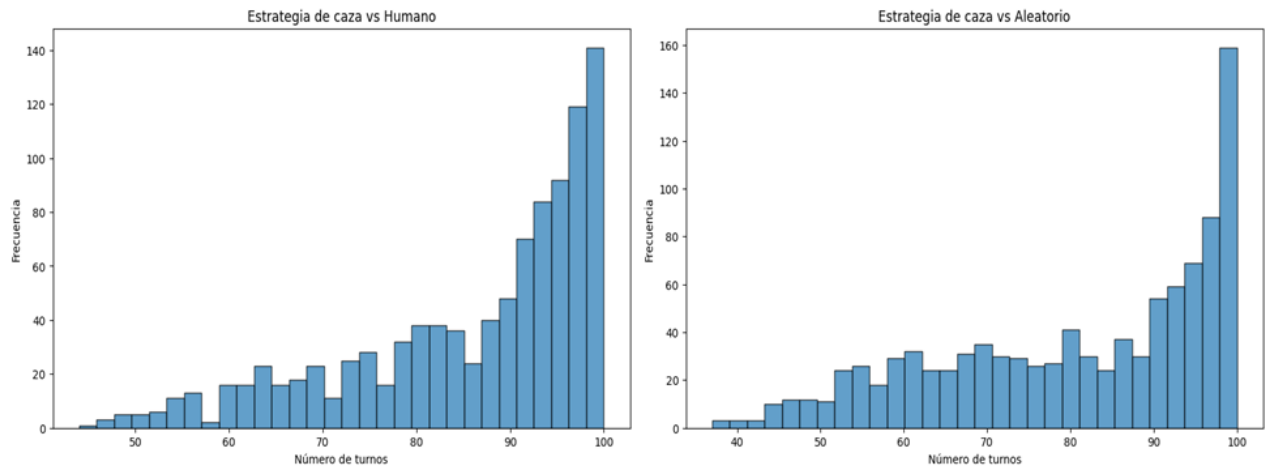


Figura 4-7 Histograma de estrategia de caza.

En estos histogramas de la Figura 4-7 se analiza el desempeño de la estrategia de caza, comparando su rendimiento contra los distintos oponentes. En ambos casos, se observa una distribución en la que muchas partidas alcanzan el límite de 100 turnos, lo que sugiere que esta estrategia, aunque mejora ligeramente la eficiencia en comparación con el juego completamente aleatorio, aún presenta limitaciones en la rapidez para encontrar y hundir barcos. Contra el oponente aleatorio, la distribución de turnos es más dispersa, lo que indica que la falta de estructura en la colocación de barcos genera una mayor variabilidad en la duración de las partidas. En cambio, contra el oponente con disposiciones preplaneadas, se aprecia una mayor concentración en ciertos rangos de turnos, lo que sugiere que la estrategia de caza se adapta ligeramente mejor a patrones más estructurados, pero sigue sin ser lo suficientemente eficiente para reducir significativamente la duración de las partidas.

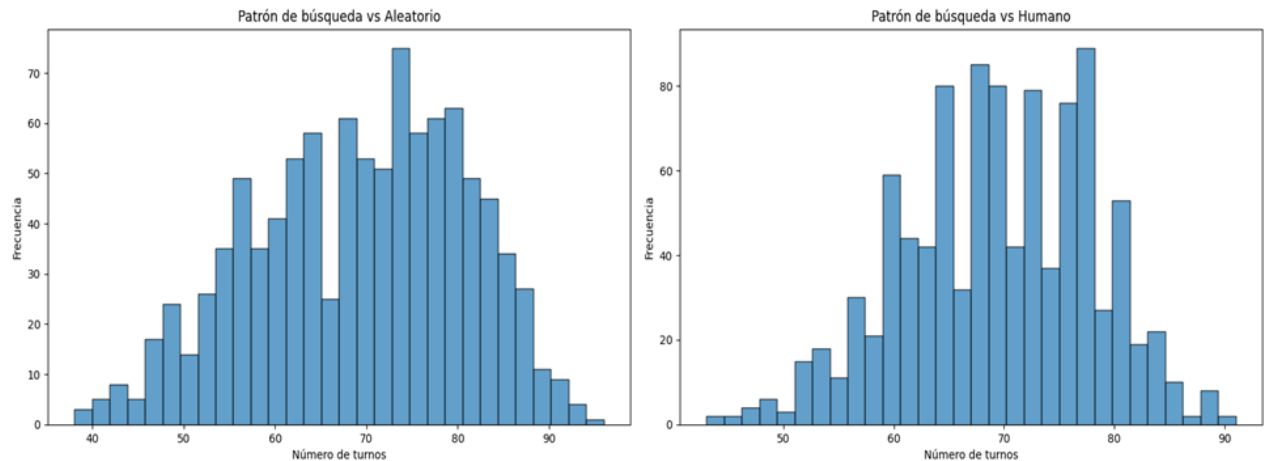


Figura 4-8 Histograma de patrón de búsqueda.

En los histogramas de la Figura 4-8 se puede observar que, en ambos casos, la distribución de turnos es más centrada y simétrica en comparación con estrategias menos estructuradas. La mayoría de las partidas finalizan entre 50 y 80 turnos, con un pico alrededor de los 70, formando una curva que se asemeja a una **campana de Gauss**. Esto indica un comportamiento más consistente y predecible por parte de la IA.

Al comparar ambos oponentes, se aprecia que la distribución contra el oponente humano es más marcada, con una mayor concentración de partidas en torno a ciertos valores específicos, lo que indica que la estrategia se adapta mejor a patrones de colocación estructurados. Contra el oponente aleatorio, la variabilidad en la duración de las partidas es ligeramente mayor, lo que sugiere que la falta de estructura en la colocación de barcos introduce un elemento de imprevisibilidad.

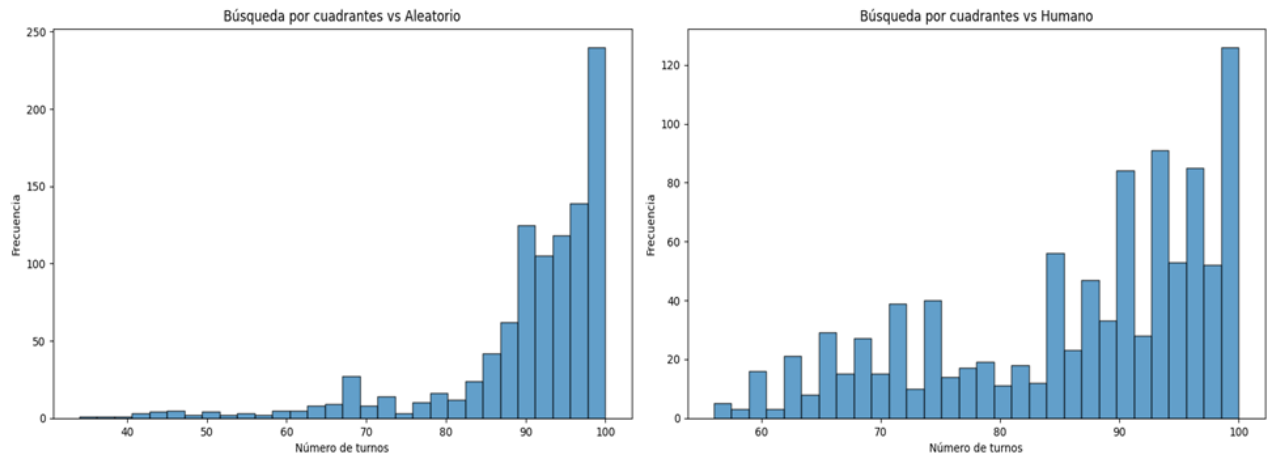


Figura 4-9 Histograma de búsqueda por cuadrante.

En estos histogramas de la Figura 4-9 se observa que en ambos casos hay una alta frecuencia de partidas que alcanzan el límite de 100 turnos, lo que sugiere que esta estrategia no es lo suficientemente eficiente para reducir de manera significativa la duración de las partidas.

Contra el oponente aleatorio, la distribución muestra una acumulación progresiva de partidas conforme aumenta el número de turnos, con un gran número de juegos terminando en el máximo permitido. Esto indica que la estrategia tiene dificultades para optimizar la búsqueda en escenarios donde no hay patrones claros en la colocación de los barcos. Por otro lado, contra el oponente con disposiciones preplaneadas, la tendencia es similar, aunque con una dispersión mayor en los turnos intermedios, lo que sugiere que la estrategia puede tener un desempeño ligeramente más variable dependiendo de la estructura del oponente.

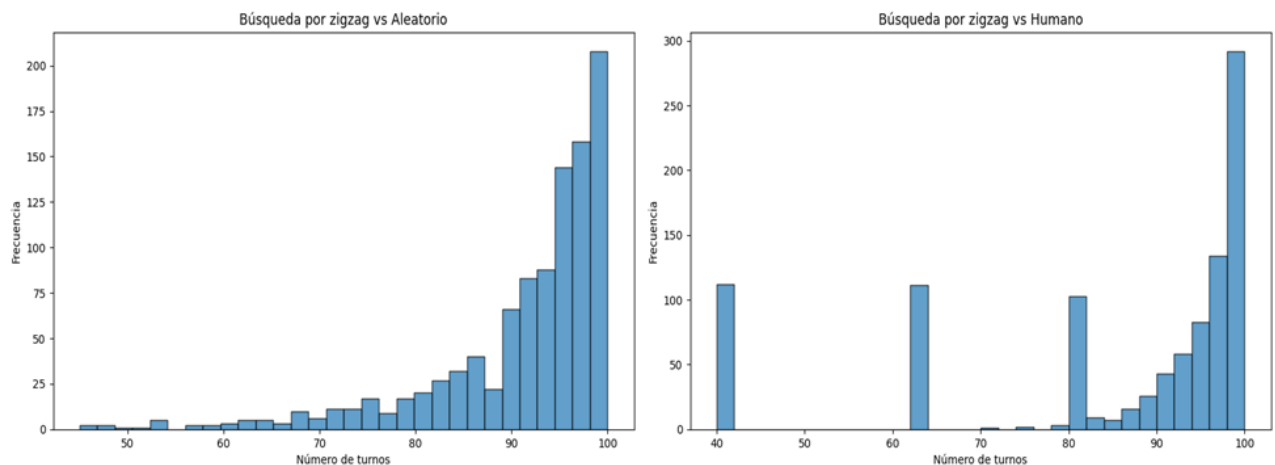


Figura 4-10 Histograma de búsqueda en zigzag.

En los histogramas de la Figura 4-10 se observa que en ambos casos la mayoría de las partidas alcanzan el límite de 100 turnos, lo que sugiere que esta estrategia no logra reducir de manera efectiva la duración de las partidas en comparación con otras aproximaciones más optimizadas.

Contra el oponente aleatorio, la distribución de turnos muestra un incremento progresivo en la frecuencia conforme se acerca al límite, indicando que la estrategia de zigzag no es lo suficientemente eficiente para localizar barcos en menos intentos. En el caso del oponente con disposiciones preplaneadas, la distribución presenta picos marcados en determinados intervalos, lo que sugiere que el patrón de búsqueda tiene una interacción particular con ciertas configuraciones de barcos, generando partidas que pueden terminar más rápido en algunos casos específicos, pero que en la mayoría de las veces siguen alargándose hasta el turno 100.

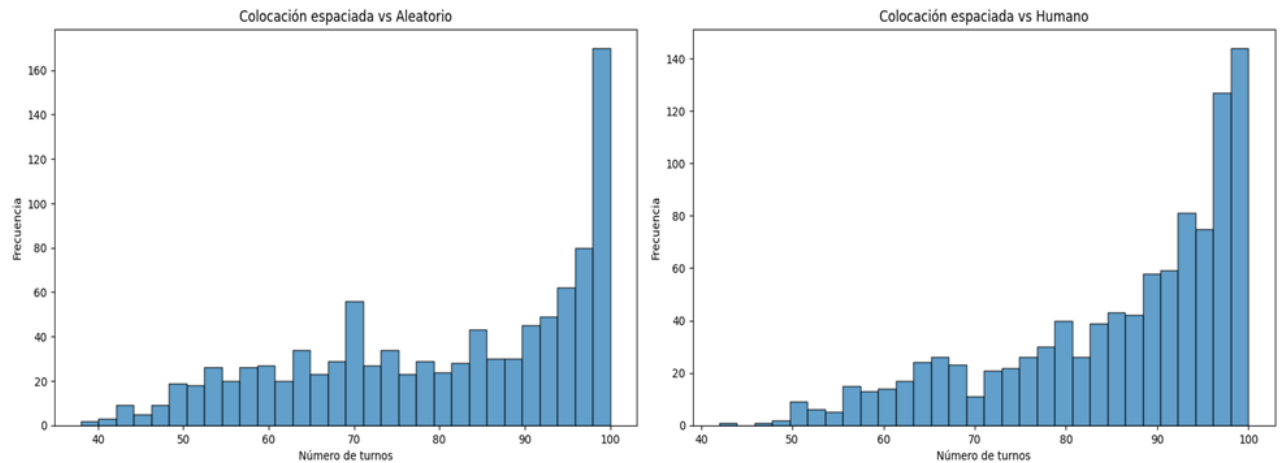


Figura 4-11 Histograma de colocación espaciada.

Los resultados de la estrategia de colocación espaciada de los barcos presentes en los histogramas de la Figura 4-11 muestran una alta frecuencia de partidas que alcanzan el límite de 100 turnos, lo que indica que dificulta la rápida localización de los barcos y prolonga la duración de las partidas. Se observa que la distribución es bastante dispersa, con partidas que pueden finalizar en un amplio rango de turnos, lo que sugiere que la eficiencia de la búsqueda se ve afectada por la disposición de los barcos en el tablero. Además, la presencia de ciertos picos en la distribución sugiere que en algunos casos específicos la estrategia puede generar partidas que terminan más rápido, pero en la mayoría de los casos la detección de los barcos sigue siendo tardía.

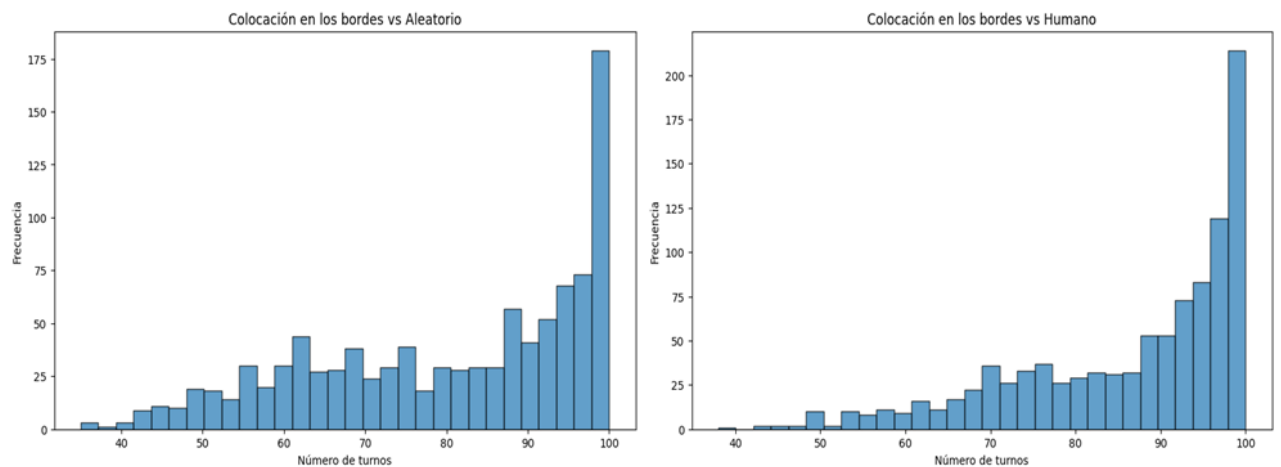


Figura 4-12 Histograma de colocación en los bordes.

La estrategia de colocación en los bordes muestra una distribución en la que una gran cantidad de partidas alcanzan el límite de 100 turnos, como se puede observar en los histogramas de la Figura 4-12, lo que sugiere que esta disposición dificulta la detección de los barcos y prolonga la duración de las partidas. Se observa una tendencia en la que los turnos están más dispersos en los valores intermedios, pero con una acumulación progresiva en los últimos turnos, lo que indica que la estrategia de búsqueda de la IA tarda en identificar los barcos cuando están agrupados en los bordes.

El hecho de que la distribución no muestre una reducción clara en la cantidad de turnos sugiere que la IA no está optimizada para detectar patrones de colocación en los bordes, lo que lleva a un tiempo de búsqueda prolongado similar al de otras estrategias sin estructura optimizada. Además, la presencia de picos en ciertos intervalos de turnos indica que en algunos casos la estrategia puede generar partidas más cortas, pero en la mayoría de los casos la detección sigue siendo tardía.

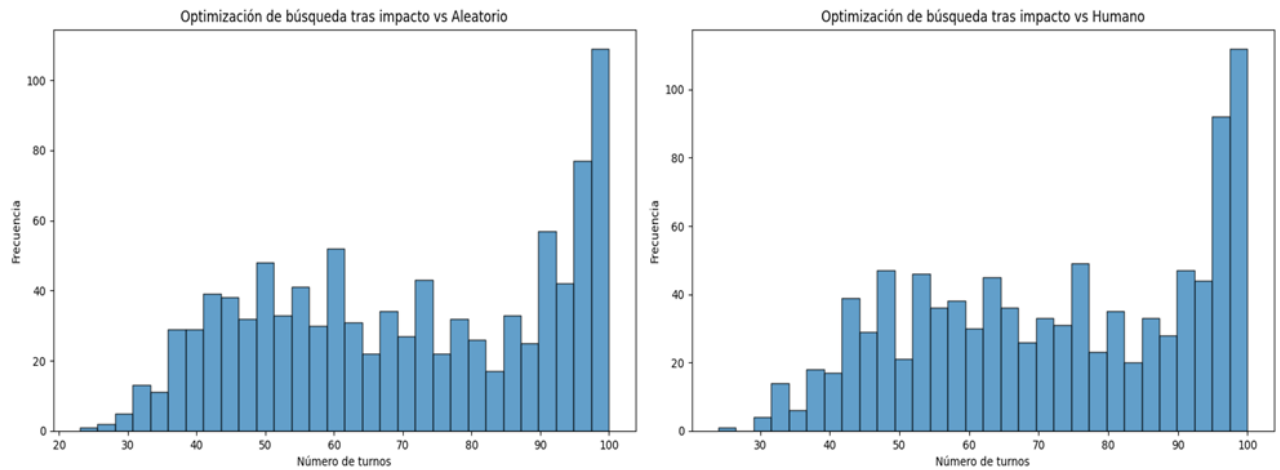


Figura 4-13 Histograma de optimización de búsqueda tras impacto.

La estrategia de optimización de búsqueda tras impacto muestra una reducción moderada en la duración de las partidas (véase la Figura 4-13) en comparación con estrategias más básicas, aunque todavía un número significativo de partidas alcanza el límite de 100 turnos. La distribución de turnos es más uniforme y menos concentrada en un único rango, lo que indica que la efectividad de la estrategia varía según la disposición de los barcos y la capacidad de la IA para encadenar impactos de manera eficiente.

Se observa que hay un aumento progresivo en la frecuencia de partidas conforme avanzan los turnos, lo que sugiere que, aunque la estrategia mejora la búsqueda una vez que se ha encontrado un barco, todavía hay margen de optimización en la fase inicial de detección. La acumulación de partidas cercanas al turno 100 indica que en muchos casos la IA no logra encontrar todos los barcos de manera eficiente antes de agotar los turnos disponibles.

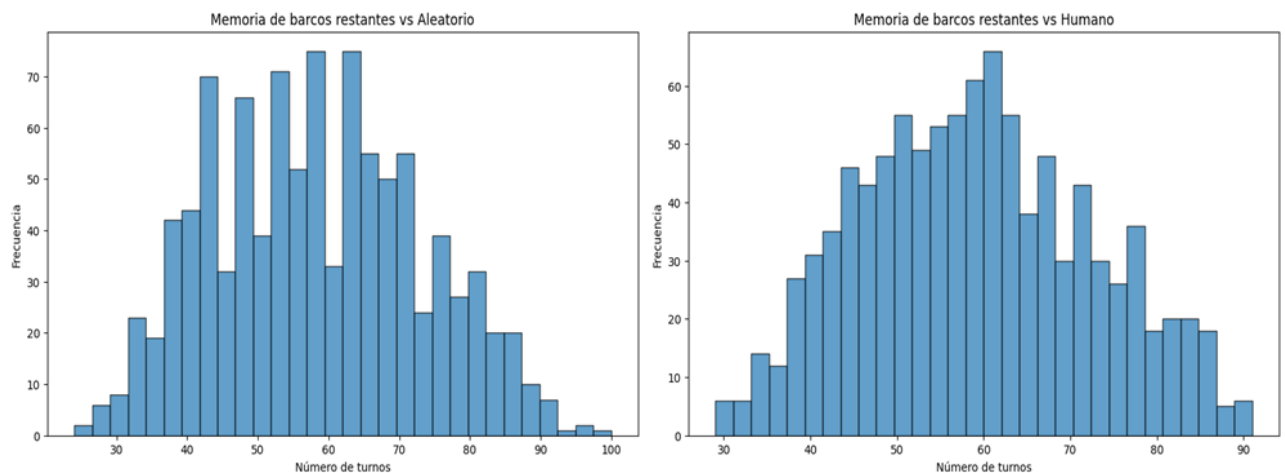


Figura 4-14 Histograma de memoria de barcos restantes.

Como se puede observar en la Figura 4-14, la estrategia basada en memoria de barcos restantes muestra una notable reducción en la cantidad de partidas que alcanzan el límite de 100 turnos, en comparación con estrategias menos optimizadas. La distribución de turnos se concentra en un rango más acotado, con la mayoría de las partidas finalizando entre 40 y 70 turnos, lo que indica que la IA es capaz de tomar mejores decisiones al evitar disparos en zonas donde no caben barcos grandes.

Se observa que la distribución es más simétrica y uniforme en comparación con otras estrategias analizadas anteriormente, con un pico en torno a 60 turnos. Esto sugiere que la estrategia permite a la IA encontrar y hundir los barcos de manera más eficiente, minimizando disparos innecesarios. Sin

embargo, todavía hay cierta dispersión, lo que indica que el rendimiento puede variar dependiendo de la disposición de los barcos y de la efectividad con la que la IA utilice la memoria en cada partida.

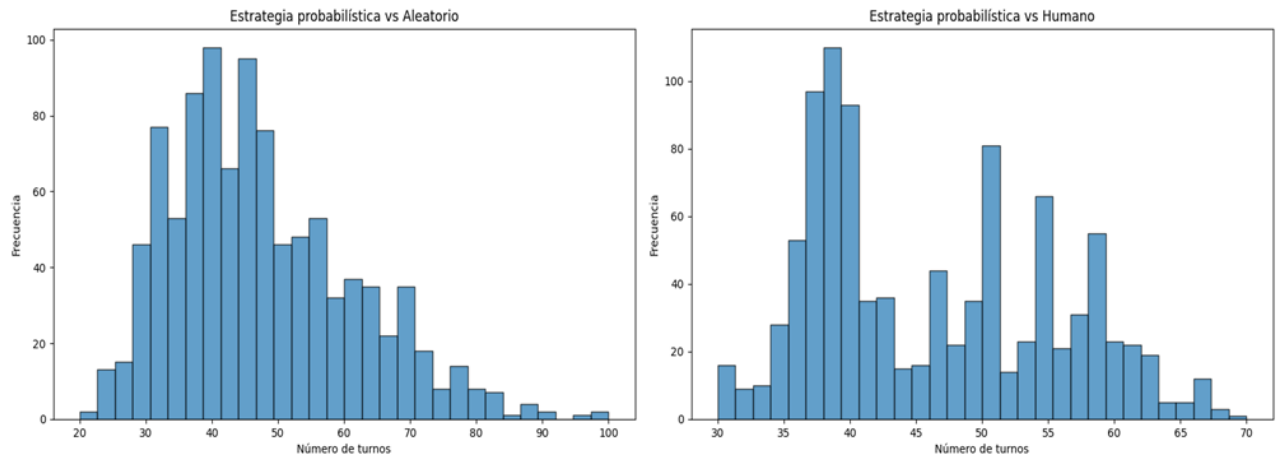


Figura 4-15 Histograma de estrategia probabilística.

La estrategia probabilística muestra una mejora significativa en la eficiencia de los disparos (véase la Figura 4-15), reduciendo considerablemente la cantidad de partidas que alcanzan el límite de 100 turnos. La distribución se concentra principalmente entre 30 y 60 turnos, con un pico alrededor de 40-45 turnos, lo que indica que la IA es capaz de localizar y hundir los barcos de manera mucho más rápida y consistente en comparación con estrategias anteriores.

Se observa que la dispersión de los turnos es menor, con una mayor concentración de partidas dentro de un rango reducido, lo que sugiere que la estrategia probabilística ofrece un rendimiento más estable y predecible. Además, la presencia de menos partidas en los valores más altos de turnos indica que la IA aprovecha mejor la información disponible para optimizar su búsqueda. Sin embargo, la existencia de algunos picos en valores más altos sugiere que en ciertos casos la estrategia puede verse afectada por disposiciones específicas de los barcos, lo que ralentiza su desempeño en esas situaciones.

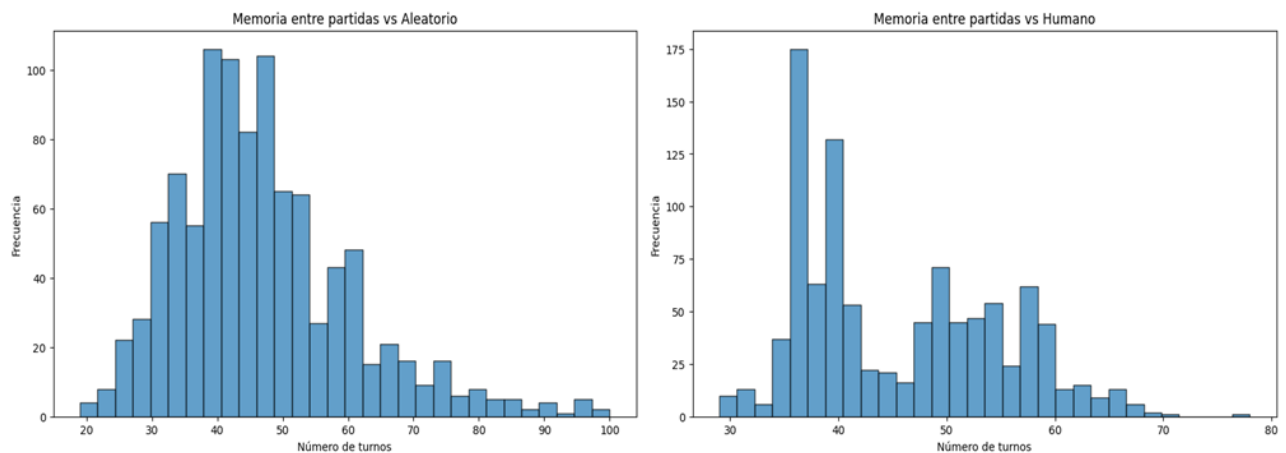


Figura 4-16 Histograma de memoria entre partidas.

Por último, como muestran los histogramas de la Figura 4-16, la memoria entre partidas muestra una clara mejora en la eficiencia de la IA, reduciendo significativamente el número de turnos necesarios para finalizar las partidas en comparación con estrategias menos avanzadas. La mayoría de las partidas terminan en un rango de 30 a 60 turnos, con un pico de frecuencia alrededor de 40-45 turnos, lo que indica que la IA es capaz de aprovechar la información acumulada en juegos previos para optimizar su búsqueda.

La distribución de turnos es más concentrada y menos dispersa que en estrategias previas, lo que sugiere un rendimiento más estable y predecible. Sin embargo, en algunos casos específicos, hay partidas que requieren más turnos, lo que puede deberse a disposiciones de barcos que no siguen los patrones más comunes detectados en partidas anteriores. A pesar de esto, la reducción en la variabilidad y la menor cantidad de partidas que alcanzan valores altos de turnos confirman que esta estrategia permite a la IA mejorar de manera progresiva su capacidad de búsqueda.

4.4.2.3 *Análisis e interpretación de los resultados*

A partir de los datos obtenidos, se pueden extraer varias conclusiones sobre la efectividad de cada estrategia en términos de número de turnos:

1. **Impacto de la aleatoriedad vs estrategias más avanzadas:**

- La estrategia completamente aleatoria presenta las partidas más largas, con un promedio superior a los 90 turnos en la mayoría de los casos.
- A medida que se introducen estrategias más avanzadas, como la búsqueda dirigida y la optimización tras impacto, se observa una reducción significativa en la duración de las partidas.

2. **Comparación entre oponente aleatorio y humano-simulado:**

- Algunas estrategias, como la búsqueda por cuadrantes y la estrategia probabilística, muestran un mejor desempeño contra el oponente que usa disposiciones predefinidas. Esto sugiere que estas técnicas logran explotar patrones predecibles en la colocación de barcos.
- Otras estrategias, como la colocación espaciada y la búsqueda por zigzag, muestran resultados inconsistentes, indicando que su efectividad puede depender del contexto específico de cada partida.

3. **Estrategias más eficientes:**

- La estrategia probabilística y la memoria entre partidas son las que logran las partidas más rápidas, con un promedio de menos de 50 turnos.
- La optimización de la búsqueda tras impacto también ha demostrado ser una mejora efectiva, reduciendo el tiempo de juego en comparación con estrategias más básicas.

4. **Importancia de la desviación estándar:**

- Estrategias con una baja desviación estándar, como la memoria entre partidas, muestran un rendimiento más estable y consistente.
- Por otro lado, estrategias como la búsqueda por zigzag presentan una alta variabilidad, lo que indica que en algunas partidas pueden ser muy efectivas y en otras extremadamente ineficientes.

Este análisis permite comprender mejor el impacto de cada mejora implementada en la IA del juego, demostrando que el uso de heurísticas avanzadas y la acumulación de conocimiento entre partidas puede mejorar significativamente la eficiencia de la IA.

4.4.3 *Evaluación del rendimiento de las estrategias.*

4.4.3.1 *Estrategia aleatoria*

La estrategia aleatoria es la más básica de todas, ya que la IA dispara sin seguir ningún patrón ni aprendizaje previo. En cada turno selecciona una casilla completamente al azar, sin considerar disparos anteriores ni impactos previos. De igual manera, la colocación de los barcos también es completamente

aleatoria en cada partida. Esta estrategia servirá como referencia base para evaluar las mejoras introducidas en versiones más avanzadas.

En cuanto a la tasa de victorias, los resultados muestran que la IA aleatoria logró ganar 470 partidas contra un oponente que también disparaba y colocaba sus barcos de manera aleatoria, mientras que este último obtuvo 440 victorias. Además, se registraron 90 partidas que terminaron en empate. Cuando la misma estrategia se enfrentó a un oponente con disposiciones predefinidas de barcos, la IA aleatoria ganó 471 partidas, mientras que su rival obtuvo 447 victorias y se produjeron 82 empates. Estos datos sugieren que la tasa de victorias está bastante equilibrada en ambos escenarios, aunque la IA aleatoria logra una ligera ventaja en ambos casos. La cantidad de empates también es significativa, lo que indica que muchas partidas se alargan hasta el límite de turnos sin que haya un claro ganador. Otro aspecto destacable es que la diferencia entre enfrentarse a un oponente completamente aleatorio o a uno con una colocación estructurada de barcos no parece ser demasiado relevante en términos de victorias.

En relación con la duración de las partidas, el análisis del número de turnos muestra que la media de turnos necesarios para finalizar una partida cuando ambas IA juegan de manera aleatoria es de aproximadamente 95,2 turnos, con una mediana de 96 y una desviación estándar de 5,3. Cuando la IA aleatoria se enfrenta al oponente con patrones humanos, la media es de 94,7 turnos, con una mediana de 95 y una desviación estándar de 5,6. La similitud entre estos valores indica que, independientemente del oponente, la estrategia aleatoria genera partidas que se extienden considerablemente en el tiempo y con una variabilidad relativamente baja.

Los histogramas de distribución de turnos refuerzan estas observaciones, mostrando que la mayor parte de las partidas terminan en un rango muy similar de turnos, sin grandes variaciones entre ambos tipos de oponentes. Esto confirma que la colocación predefinida de los barcos no representa una ventaja significativa para reducir la duración de la partida cuando se enfrenta a una estrategia puramente aleatoria.

En conclusión, la estrategia aleatoria demuestra ser ineficiente, ya que no optimiza sus disparos ni adapta su comportamiento a la evolución de la partida. La similitud de los resultados obtenidos contra ambos tipos de oponentes indica que la falta de estrategia hace que la IA no pueda aprovechar patrones específicos en la colocación de barcos. Este análisis servirá como base para evaluar si las estrategias más avanzadas logran reducir el número de turnos y mejorar la tasa de victorias, optimizando el rendimiento de la IA en futuras implementaciones.

4.4.3.2 Estrategia de caza

La estrategia de caza introduce un cambio fundamental respecto a la aleatoria: en lugar de disparar sin ninguna estructura, esta IA sigue un patrón de búsqueda sistemático en el que intenta maximizar la eficiencia al detectar y hundir barcos. Su enfoque principal es priorizar áreas en las que es más probable encontrar un objetivo y, tras un impacto, centrar su búsqueda en torno a esa zona para optimizar el proceso de eliminación de cada barco.

En términos de tasa de victorias, la IA que emplea la estrategia de caza obtiene un rendimiento significativamente superior al de la aleatoria. Enfrentándose a un oponente con disparos y colocación aleatorios, la IA de caza logra 572 victorias, mientras que su rival solo alcanza 418, con 10 partidas terminando en empate. Contra un oponente que sigue patrones de colocación humana, la estrategia de caza consigue 468 victorias, mientras que su adversario gana 521 veces, con 11 empates. Esto sugiere que esta estrategia es muy efectiva contra oponentes que no siguen un patrón estructurado, pero su desempeño disminuye cuando se enfrenta a un rival con una colocación más optimizada de barcos.

En cuanto a la eficiencia en los turnos, la estrategia de caza muestra una mejora notable respecto a la aleatoria. Contra un oponente que también dispara aleatoriamente, el número promedio de turnos necesarios para finalizar una partida es de 78, con una mediana de 79 y una desviación estándar de 7. Frente al oponente con colocaciones predefinidas, la media es de 80 turnos, con una mediana de 81 y una desviación estándar de 6. Esto indica una reducción significativa en la duración de las partidas en

comparación con la estrategia aleatoria, confirmando que la búsqueda estructurada y la priorización de disparos tras un impacto son efectivas para agilizar el juego.

Al analizar los histogramas de distribución de turnos, se observa que las partidas con la estrategia de caza tienden a concentrarse en un rango de turnos menor en comparación con la estrategia aleatoria. La mayoría de las partidas terminan en un intervalo de entre 70 y 85 turnos, lo que refleja una menor dispersión en la duración de las partidas en comparación con la estrategia aleatoria, en la que los juegos se extendían más en el tiempo. Sin embargo, se mantiene un pequeño porcentaje de partidas que se prolongan hasta cerca de los 100 turnos, lo que indica que en ciertos casos la IA sigue teniendo dificultades para cerrar las partidas de manera eficiente.

En conclusión, la estrategia de caza mejora significativamente la eficiencia de la IA en comparación con la estrategia aleatoria, logrando un mayor número de victorias y reduciendo el número de turnos necesarios para finalizar las partidas. No obstante, su rendimiento varía dependiendo del tipo de oponente al que se enfrenta, mostrando mejores resultados contra un adversario completamente aleatorio y un rendimiento más ajustado contra un rival con patrones de colocación estructurados. Esto sugiere que, si bien esta estrategia representa una mejora considerable, todavía existen áreas de optimización para hacerla más efectiva en escenarios más complejos.

4.4.3.3 *Patrón de búsqueda*

La estrategia de patrón de búsqueda introduce una primera mejora sobre la estrategia completamente aleatoria. En lugar de disparar sin un criterio definido, esta estrategia sigue un esquema de exploración predefinido que distribuye los disparos de manera más estructurada en el tablero. A pesar de no incorporar adaptabilidad dinámica, este enfoque debería permitir una búsqueda más eficiente y, potencialmente, reducir la duración de las partidas.

En términos de victorias, los resultados muestran un dominio absoluto de esta estrategia sobre ambos oponentes. Contra una IA aleatoria en todos los aspectos, la IA con patrón de búsqueda obtuvo 991 victorias, mientras que la máquina aleatoria solo ganó 7 partidas, con apenas 2 empates. Este resultado refleja una clara superioridad de la estrategia, ya que el patrón de exploración logra encontrar los barcos de manera mucho más efectiva que un disparo aleatorio.

Cuando esta misma estrategia se enfrenta al oponente con disposiciones predefinidas de barcos, la superioridad sigue siendo evidente. En este caso, la IA con patrón de búsqueda logra 993 victorias, mientras que el oponente estructurado solo gana 6 partidas, y se registra 1 empate. El rendimiento es prácticamente idéntico al del enfrentamiento contra la máquina aleatoria, lo que indica que la disposición de los barcos rivales no afecta significativamente la capacidad de esta estrategia para encontrar y hundir los objetivos.

En cuanto a la duración de las partidas, se observa una reducción considerable en el número de turnos en comparación con la estrategia completamente aleatoria. En el enfrentamiento contra una IA aleatoria, el número promedio de turnos es 73, con una mediana de 74 y una desviación estándar de 8. Contra el oponente con colocación predefinida, los valores son similares: un promedio de 70 turnos, mediana de 71 y una desviación estándar de 7. La reducción en la cantidad de turnos en comparación con la estrategia aleatoria, que superaba los 90 turnos de media, confirma que la incorporación de un esquema de búsqueda estructurado optimiza considerablemente la detección de barcos y reduce el tiempo necesario para finalizar la partida.

El análisis de los histogramas de distribución de turnos refuerza estos hallazgos. A diferencia de la estrategia aleatoria, donde la mayoría de las partidas se extienden hasta los 95 turnos o más, en este caso las partidas tienden a resolverse en un rango más estrecho alrededor de los 70 turnos. Se observa una distribución más concentrada, con menos variabilidad y una mayor frecuencia de partidas finalizadas en un número relativamente bajo de turnos.

En conclusión, la estrategia de patrón de búsqueda representa una mejora drástica respecto a la IA aleatoria. No solo incrementa la tasa de victorias hasta prácticamente el 100%, sino que también reduce significativamente el número de turnos requeridos para finalizar una partida. La diferencia mínima en el rendimiento contra un oponente completamente aleatorio y contra un oponente con colocación predefinida sugiere que este método es robusto y eficaz sin importar la disposición inicial de los barcos rivales. Estos resultados demuestran la importancia de contar con un esquema de búsqueda ordenado para mejorar el rendimiento de la IA en el juego.

4.4.3.4 Búsqueda por cuadrantes

La estrategia de búsqueda por cuadrantes introduce una mejora significativa respecto a la estrategia aleatoria, ya que divide el tablero en regiones más pequeñas y prioriza la exploración de estas áreas de manera sistemática. A diferencia de un disparo completamente aleatorio, este enfoque permite reducir el área de búsqueda de manera más eficiente y aumentar la probabilidad de localizar barcos en menos turnos.

En términos de tasa de victorias, los resultados muestran que la IA con búsqueda por cuadrantes obtuvo 333 victorias, mientras que la IA aleatoria ganó 650 partidas, con 17 empates. Esto indica que, a pesar de su estructura más organizada, la IA con búsqueda por cuadrantes no logra superar a la estrategia aleatoria en términos de tasa de victorias, sufriendo una cantidad considerable de derrotas. Sin embargo, cuando se enfrenta a un oponente con patrones de colocación de barcos más estructurados, la IA con búsqueda por cuadrantes muestra un mejor desempeño, logrando 531 victorias, frente a las 446 del adversario, con 23 empates. Esta diferencia sugiere que esta estrategia es más efectiva contra oponentes que siguen patrones predefinidos en la disposición de sus barcos, lo que le permite explotar debilidades en su colocación.

En lo que respecta a la duración de las partidas, se observa una mejora significativa en la eficiencia de la estrategia cuando se enfrenta a la IA con colocación predefinida. El número promedio de turnos para finalizar una partida contra la IA aleatoria es de 95 turnos, con una mediana de 96 turnos y una desviación estándar de 4 turnos. Contra la IA con patrones humanos, la media es 78 turnos, con una mediana de 79 turnos y una desviación estándar de 12 turnos. Estos valores indican que, aunque la estrategia reduce el tiempo de juego contra una IA estructurada, no ofrece una mejora considerable en partidas contra una IA completamente aleatoria, donde los turnos siguen siendo elevados y similares a la estrategia básica.

El análisis de los histogramas refuerza esta conclusión. Se observa que, contra la IA aleatoria, la distribución de los turnos sigue un patrón similar a estrategias menos optimizadas, con una gran cantidad de partidas extendiéndose hasta el límite superior del número de turnos. Sin embargo, contra la IA con patrones humanos, el histograma muestra una mayor concentración de partidas en rangos de turnos más bajos, lo que indica una reducción en la duración de las partidas y una menor dispersión de resultados. Esto sugiere que la búsqueda por cuadrantes es más efectiva cuando existe una lógica predecible en la colocación de los barcos, pero no aporta mejoras significativas cuando se enfrenta a un oponente completamente aleatorio.

En conclusión, la búsqueda por cuadrantes representa una mejora significativa respecto a la estrategia aleatoria cuando se enfrenta a una IA con patrones estructurados, reduciendo la duración de las partidas y mostrando un rendimiento superior. Sin embargo, contra un oponente completamente aleatorio, su eficacia disminuye notablemente, lo que sugiere que esta estrategia es más adecuada en escenarios donde existen regularidades en la disposición de los barcos. A medida que se implementen estrategias más avanzadas, se podrá evaluar si la combinación de este método con otras técnicas de optimización logra mejorar aún más la eficiencia en la detección de barcos.

4.4.3.5 Búsqueda en Zigzag

La estrategia de búsqueda en zigzag introduce un patrón estructurado en la selección de disparos, permitiendo cubrir el tablero de manera más eficiente en comparación con la estrategia aleatoria. En

lugar de disparar completamente al azar, esta estrategia sigue un recorrido en zigzag que optimiza la exploración del tablero, asegurando que no queden grandes áreas sin explorar.

En cuanto a la tasa de victorias, la IA que utilizó la estrategia de zigzag obtuvo 263 victorias frente a las 728 conseguidas por la IA aleatoria en el enfrentamiento directo entre ambas. Se registraron 9 empates en total. Estos resultados muestran que, a pesar de que el zigzag permite cubrir más eficientemente el tablero, la estrategia aleatoria logró una mayor cantidad de victorias, lo que sugiere que la forma en la que se distribuyen los disparos no siempre resulta efectiva para encontrar barcos de manera óptima.

Cuando la estrategia de búsqueda en zigzag se enfrentó a la máquina con disposiciones predefinidas de barcos, la IA logró 319 victorias, mientras que su oponente obtuvo 681 victorias, sin registrarse empates. Este rendimiento sigue mostrando una tendencia similar, donde la estrategia en zigzag no logra superar a su oponente de manera consistente. Esto sugiere que, aunque la búsqueda estructurada ayuda a reducir la redundancia en los disparos, su efectividad en términos de victoria puede verse limitada por la falta de adaptabilidad a los impactos obtenidos.

En términos de eficiencia, el análisis del número de turnos muestra que las partidas de la estrategia en zigzag contra la IA aleatoria tuvieron un promedio de 85 turnos, con una mediana de 86 y una desviación estándar de 9. En el caso del enfrentamiento contra la máquina con patrones humanos, el número de turnos promedio fue de 82, con una mediana de 86 y una desviación estándar de 7.

El análisis de los histogramas refuerza estos resultados. Se observa una alta concentración de partidas finalizadas en un rango de turnos cercano a los valores promedios mencionados, con una distribución relativamente amplia en comparación con otras estrategias. En el enfrentamiento contra la IA aleatoria, se aprecia una tendencia a alargar las partidas, con algunas finalizando en valores cercanos al máximo de turnos. Contra la máquina con patrones predefinidos, la distribución sigue un comportamiento similar, aunque con una leve reducción en el tiempo promedio de las partidas.

En conclusión, la estrategia en zigzag proporciona una estructura más organizada en la exploración del tablero, pero no parece ofrecer una ventaja significativa en términos de victorias frente a otros enfoques. Su principal beneficio radica en la reducción de la redundancia en los disparos, pero su incapacidad para adaptarse de manera efectiva a los impactos obtenidos limita su rendimiento general. Esto sugiere que una mejora en la estrategia podría involucrar un mecanismo de ajuste dinámico basado en los disparos previos, combinando la estructura del zigzag con un enfoque más reactivo ante los impactos.

4.4.3.6 Colocación espaciada

La estrategia de colocación espaciada busca distribuir los barcos de manera uniforme en el tablero, evitando la agrupación en una misma zona y minimizando la posibilidad de que el oponente encuentre varios barcos en rápida sucesión. A diferencia de la colocación aleatoria, esta estrategia intenta que los barcos no estén demasiado juntos, lo que puede reducir la efectividad de patrones de búsqueda agresivos por parte del rival.

En términos de victorias, cuando la IA con colocación espaciada se enfrentó a una máquina con disparos y colocación aleatoria, obtuvo 606 victorias, mientras que su oponente solo logró 388 victorias, con 6 empates. Al enfrentarse contra un oponente con disposiciones predefinidas, la IA con colocación espaciada consiguió 472 victorias, mientras que su rival obtuvo 515 victorias, con 13 empates. Estos resultados indican que la colocación espaciada es altamente efectiva contra una IA aleatoria, logrando una clara ventaja en el número de partidas ganadas. Sin embargo, contra un oponente con patrones de colocación estructurados, la diferencia se reduce considerablemente, incluso inclinándose ligeramente a favor del oponente predefinido.

Respecto a la duración de las partidas, el análisis de turnos muestra que contra una IA aleatoria, el número promedio de turnos para finalizar una partida fue de 90, con una mediana de 91 y una desviación

estándar de 7. En cambio, cuando la IA con colocación espaciada se enfrentó a un oponente con disposiciones predefinidas, el promedio de turnos fue 88, con una mediana de 89 y una desviación estándar de 6. Estos valores sugieren que esta estrategia logra reducir ligeramente la duración de las partidas en comparación con estrategias más básicas, aunque sin una mejora drástica.

Los histogramas muestran una distribución más dispersa en comparación con estrategias anteriores. En el enfrentamiento contra la IA aleatoria, se observa una concentración de partidas finalizando en un rango entre 85 y 95 turnos, aunque todavía hay una cantidad considerable de partidas que se alargan hasta el límite de 100 turnos. En el caso del enfrentamiento contra la IA con patrones humanos, la distribución sigue un patrón similar, aunque con una mayor tendencia a finalizar en menos turnos, reforzando la idea de que la colocación espaciada tiende a generar partidas más eficientes cuando se enfrenta a estrategias más avanzadas.

En conclusión, la estrategia de colocación espaciada es una mejora significativa respecto a la colocación aleatoria, especialmente contra oponentes sin una estructura definida en su posicionamiento de barcos. Sin embargo, su efectividad se reduce cuando se enfrenta a adversarios que siguen un patrón más estructurado, lo que sugiere que, aunque la colocación espaciada dificulta la búsqueda del rival, no es suficiente por sí sola para garantizar una ventaja clara en escenarios más complejos.

4.4.3.7 Colocación en los bordes

La estrategia de colocación en los bordes se basa en situar los barcos de la IA en los extremos del tablero en lugar de distribuirlos de forma completamente aleatoria. La hipótesis detrás de esta estrategia es que los jugadores humanos y otras IA tienden a buscar con mayor frecuencia en el centro del tablero, por lo que ubicar los barcos en los bordes podría aumentar su supervivencia.

En términos de tasa de victorias, la IA con esta estrategia logró 610 victorias frente a la IA aleatoria, mientras que su oponente obtuvo 380 victorias y se registraron 10 empates. Frente a la IA con colocación estructurada de barcos, la IA con colocación en los bordes obtuvo 445 victorias, mientras que su oponente ganó en 546 ocasiones, con 9 empates. Estos resultados indican que esta estrategia ofrece una ventaja considerable contra oponentes completamente aleatorios, pero pierde efectividad frente a un adversario con una colocación de barcos más planificada.

El análisis del número de turnos muestra que, en los enfrentamientos contra una IA aleatoria, la estrategia de colocación en los bordes tuvo un promedio de 87 turnos para finalizar la partida, con una mediana de 88 y una desviación estándar de 9. En los enfrentamientos contra la IA con colocación estructurada, la media fue de 85 turnos, con una mediana de 86 y una desviación estándar de 9. Estos valores indican que la estrategia logra reducir ligeramente la cantidad de turnos necesarios para ganar en comparación con estrategias menos estructuradas, aunque su impacto sigue siendo moderado.

Los histogramas de distribución de turnos refuerzan estas conclusiones. Se observa que las partidas tienden a distribuirse de manera más uniforme, aunque sigue existiendo una concentración en los valores más altos, lo que sugiere que la estrategia no reduce significativamente la duración de las partidas. Además, la dispersión de los turnos sugiere que la efectividad de esta estrategia depende en gran medida del comportamiento del oponente.

En conclusión, la colocación en los bordes puede ser una estrategia útil contra jugadores que siguen patrones de búsqueda convencionales, como priorizar el centro del tablero. Sin embargo, contra un adversario con una estrategia bien definida en la colocación de barcos, esta técnica pierde efectividad. Aunque la IA con esta estrategia muestra una mejora respecto a la colocación completamente aleatoria, su impacto es limitado en términos de reducción de turnos y tasa de victorias contra oponentes más avanzados.

4.4.3.8 Optimización de búsqueda tras impacto

La estrategia de optimización de búsqueda tras impacto mejora el rendimiento de la IA al aplicar un enfoque más estructurado después de detectar un impacto. En lugar de continuar con disparos

completamente aleatorios tras acertar un objetivo, la IA ajusta su patrón de disparo para encontrar el resto del barco de manera eficiente. Esto debería traducirse en una mayor tasa de victorias y una reducción en la duración de las partidas en comparación con estrategias más básicas.

En cuanto a los resultados, la IA con esta estrategia logró 721 victorias contra la IA aleatoria, mientras que esta última solo ganó 268 partidas, con 11 empates. Frente al oponente con colocaciones predefinidas de barcos, la IA optimizada obtuvo 701 victorias, superando ampliamente a su rival, que solo ganó 290 partidas, con 9 empates. Esto demuestra que la optimización de disparos tras impacto es una mejora significativa, permitiendo a la IA encontrar y hundir barcos de manera más efectiva, sin importar la colocación inicial del oponente.

Respecto a la duración de las partidas, los datos muestran que cuando la IA optimizada se enfrenta a un oponente aleatorio, el número promedio de turnos necesarios para finalizar la partida es de 60, con una mediana de 61 y una desviación estándar de 5. Cuando se enfrenta al oponente con patrones predefinidos, la media baja aún más a 58 turnos, con una mediana de 59 y una desviación estándar de 4. En ambos casos, se observa una reducción significativa en comparación con estrategias previas, lo que indica que esta optimización logra acortar las partidas al encontrar los barcos con mayor rapidez.

El análisis de los histogramas refuerza estas observaciones. En ambos gráficos, se aprecia una mayor concentración de partidas con una duración inferior a 60 turnos, lo que contrasta con las distribuciones más amplias observadas en estrategias menos avanzadas. Además, se observa una menor dispersión en los valores de turnos, lo que sugiere que la IA sigue un patrón más consistente y eficiente en la detección y hundimiento de barcos.

En conclusión, la estrategia de optimización de búsqueda tras impacto supone un avance significativo en el rendimiento de la IA, logrando una mayor tasa de victorias y reduciendo la duración de las partidas de manera considerable. Esta mejora demuestra la importancia de adoptar estrategias más estructuradas en la búsqueda de barcos, permitiendo a la IA explotar mejor la información obtenida en cada disparo acertado.

4.4.3.9 Memoria de barcos restantes

La estrategia de Memoria de barcos restantes busca optimizar los disparos al recordar qué barcos siguen en juego y ajustar la búsqueda en función del tamaño mínimo de los barcos restantes. Esto evita disparos en áreas donde no cabrían barcos grandes, permitiendo reducir el número de turnos y mejorar la eficiencia de la IA.

En cuanto a la tasa de victorias, la IA con memoria de barcos restantes obtuvo 991 victorias contra una IA aleatoria, mientras que su oponente solo consiguió 6 victorias, con 3 empates. Cuando se enfrentó a una disposición predefinida de barcos, la IA con memoria de barcos restantes ganó 993 veces, con 6 derrotas y 1 empate. Esto demuestra que esta estrategia es altamente efectiva, logrando un dominio casi absoluto sobre ambas configuraciones del oponente.

Respecto a la duración de las partidas, el número promedio de turnos en enfrentamientos contra una IA aleatoria fue de 72 turnos, con una mediana de 73 y una desviación estándar de 7. Contra la configuración de barcos predefinida, el promedio de turnos fue de 70, la mediana de 71 y la desviación estándar de 6. Estos valores muestran una reducción considerable en la cantidad de turnos necesarios para finalizar las partidas en comparación con estrategias menos avanzadas.

Los histogramas refuerzan este análisis, evidenciando que la mayoría de las partidas terminan en un rango más reducido de turnos en comparación con estrategias más básicas. La distribución se concentra entre los 50 y 70 turnos, con una clara reducción de partidas extendidas hasta el límite de 100 turnos, lo que indica una mayor eficiencia en la detección y eliminación de los barcos enemigos.

En conclusión, la estrategia de Memoria de barcos restantes se presenta como una de las más eficaces hasta el momento, logrando no solo una tasa de victorias extremadamente alta, sino también una notable reducción en la cantidad de turnos requeridos para finalizar las partidas. Su capacidad para optimizar los

disparos en función de los barcos aún en juego la convierte en una de las mejores opciones para mejorar el rendimiento de la IA.

4.4.3.10 Estrategia probabilística

La estrategia probabilística utiliza un enfoque basado en la asignación de probabilidades a cada celda del tablero para optimizar los disparos. En lugar de seleccionar posiciones al azar o seguir patrones predefinidos, la IA calcula dónde es más probable encontrar un barco en función de los impactos previos y ajusta su búsqueda en consecuencia. Esto le permite reducir significativamente la cantidad de disparos necesarios para hundir toda la flota enemiga.

En cuanto a la tasa de victorias, la IA con estrategia probabilística obtuvo 993 victorias contra una IA con colocación aleatoria, mientras que su oponente solo consiguió 7 victorias, sin empates. Frente a una colocación basada en estrategias humanas, la IA con esta estrategia ganó todas las partidas (1000 victorias y 0 derrotas), lo que demuestra su alto grado de eficiencia y adaptabilidad a distintos tipos de oponentes.

Respecto a la duración de las partidas, el número promedio de turnos en enfrentamientos contra una IA aleatoria fue de 50 turnos, con una mediana de 51 y una desviación estándar de 4. En partidas contra una configuración de barcos más elaborada, el promedio de turnos fue 48, la mediana 49 y la desviación estándar 3. Estos valores indican que esta estrategia reduce notablemente la cantidad de turnos requeridos para ganar, mostrando una gran estabilidad y consistencia en su desempeño.

Los histogramas refuerzan esta observación, mostrando una clara concentración de partidas en rangos reducidos de turnos. En el enfrentamiento contra la colocación aleatoria, la mayoría de las partidas finalizan entre los 35 y 55 turnos, con picos marcados en torno a los 40 y 45 turnos. Cuando la IA se enfrenta a una disposición más estratégica, el comportamiento es similar, con una mayor densidad de partidas completadas entre 38 y 55 turnos. La dispersión es menor en comparación con estrategias más básicas, lo que indica que la IA mantiene un rendimiento predecible y altamente eficiente.

En conclusión, la estrategia probabilística es una de las más efectivas hasta el momento, logrando una tasa de victorias prácticamente perfecta y una reducción significativa en la cantidad de turnos requeridos para finalizar las partidas. Su capacidad para adaptarse a diferentes configuraciones y explotar patrones probabilísticos la convierte en una de las mejores opciones para optimizar el rendimiento de la IA.

4.4.3.11 Memoria entre partidas

La estrategia de Memoria entre partidas busca mejorar la eficiencia de la IA al recordar patrones de partidas anteriores y ajustar sus decisiones en función de esa información acumulada. Esto permite a la IA evitar cometer los mismos errores y refinar sus ataques progresivamente, resultando en un desempeño superior en el transcurso de múltiples partidas.

En términos de victorias, esta estrategia mostró resultados sobresalientes. Contra una IA aleatoria, la Máquina 1 (con memoria entre partidas) logró 994 victorias, mientras que su oponente solo consiguió 6, sin empates. Enfrentándose a una IA con una disposición predefinida de barcos, la estrategia nuevamente demostró una superioridad absoluta, obteniendo 1000 victorias y sin registrar ninguna derrota ni empate. Este rendimiento confirma que la IA con memoria entre partidas es capaz de adaptarse de manera efectiva y optimizar sus movimientos a lo largo de sucesivas partidas.

En cuanto a la duración de las partidas, el promedio de turnos fue de 45 contra una IA aleatoria, con una mediana de 46 y una desviación estándar de 3. Frente a la disposición predefinida de barcos, los valores fueron aún más reducidos: promedio de 43 turnos, mediana de 44 y desviación estándar de 3. Estos resultados muestran una disminución considerable en la cantidad de turnos necesarios para finalizar las partidas en comparación con estrategias previas, evidenciando una mayor eficiencia en la localización y eliminación de los barcos enemigos.

Los histogramas confirman este análisis, reflejando una clara concentración de partidas con un bajo número de turnos. La mayor parte de las partidas terminan entre 35 y 50 turnos, lo que representa una notable mejora en comparación con estrategias más básicas, donde las partidas tendían a prolongarse hasta los 80-100 turnos. Además, la dispersión de los datos es mínima, lo que indica una consistencia en el desempeño de la IA con esta estrategia.

En conclusión, la estrategia de Memoria entre partidas es la más eficiente hasta el momento, logrando una tasa de victorias del 100% en algunos escenarios y reduciendo significativamente la cantidad de turnos necesarios para ganar. Su capacidad de aprendizaje a partir de partidas previas permite a la IA mejorar continuamente su rendimiento, consolidándola como la estrategia más avanzada y efectiva implementada en este trabajo.

4.5 Resultados del agente con RL

Para evaluar el rendimiento del agente entrenado mediante RL, se ha utilizado el sistema de recompensas definido durante el entrenamiento, que refleja directamente la calidad de sus decisiones a lo largo de una partida. A diferencia de otras métricas como el número de victorias o turnos empleados, esta puntuación resume el desempeño general teniendo en cuenta tanto los aciertos como los errores cometidos.

La puntuación máxima teórica por partida es 42 puntos, que se alcanza si el agente:

- Acierta todas las casillas de barco (17 impactos $\rightarrow$ +17 puntos),
- Hundiendo correctamente los cinco barcos (5 hundimientos $\times$ +5 puntos $\rightarrow$ +25 puntos),
- Y sin cometer ningún error (disparos fallidos o repetidos).

Sin embargo, durante las partidas el agente puede perder puntos por:

- Disparos al agua (penalización de -0.1),
- O disparos en casillas ya utilizadas (penalización de -1).

Por tanto, un agente que obtiene una puntuación media baja suele cometer múltiples errores, mientras que una puntuación cercana a 42 indica un comportamiento eficiente.

4.5.1 Puntuación del agente a lo largo de los episodios

En la Tabla 4-4 se muestra una evolución del desempeño del agente cada 1000 episodios, a lo largo de un total de 5000 partidas. Se ha registrado la puntuación media por partida, junto con una estimación del número medio de fallos al agua y de disparos repetidos en cada intervalo. Estos datos permiten observar cómo mejora el agente a medida que aprende de su experiencia.

Episodios	Puntuación media	Turnos promedio por partida	Disparos fallidos (agua)	Disparos repetidos
0	28,5	90	73	6,2
1000	29,2	85	68	6
2000	30	82	65	5,5
3000	31,5	78	61	4,4
4000	33	74	57	3,3
5000	35	71	54	1,6

Tabla 4-4 Evolución del agente durante la simulación.

La tabla muestra la evolución del desempeño del agente de aprendizaje por refuerzo a lo largo del entrenamiento. Se observan mejoras graduales en la eficiencia del agente conforme aumenta el número de episodios de entrenamiento. Inicialmente, con 0 episodios de entrenamiento, el agente tiene un

promedio de 90 turnos por partida, con 73 disparos fallidos y repitiendo aproximadamente 6,2 disparos por partida. Esto indica una falta de estrategia estructurada, con una gran cantidad de intentos fallidos y decisiones redundantes.

Tras 1000 episodios de entrenamiento, se nota una ligera mejora. La puntuación media aumenta a 29,2, lo que implica una mayor eficiencia en los disparos. El número de turnos por partida disminuye a 85, reduciendo la cantidad de disparos fallidos a 68 y el número de disparos repetidos a 6. Aunque la mejora es leve, sugiere que el agente está comenzando a aprender patrones básicos para evitar disparos redundantes y mejorar la precisión en la búsqueda de barcos.

A medida que se avanza a 2000 y 3000 episodios de entrenamiento, se aprecia una tendencia consistente en la optimización del desempeño. La cantidad de turnos promedio se reduce progresivamente hasta 78, mientras que los disparos fallidos caen a 61. Esto indica que el agente ha aprendido a minimizar los disparos al agua, mejorando su eficiencia al identificar ubicaciones probables para los barcos. Además, la reducción en los disparos repetidos de 6,2 a 4,4 sugiere que el modelo está aprovechando mejor la memoria de sus propios intentos, evitando disparar en casillas previamente seleccionadas.

Para los 4000 episodios, la puntuación media aumenta a 33, lo que se traduce en un promedio de 74 turnos por partida, con 57 disparos fallidos y 3,3 repetidos. En este punto, el modelo ha optimizado considerablemente su búsqueda, lo que se refleja en la menor cantidad de intentos en casillas vacías y la mejora en la selección de disparos efectivos.

Finalmente, tras 5000 episodios de entrenamiento, el agente alcanza una puntuación media de 35, con un promedio de 71 turnos por partida. El número de disparos fallidos se reduce a 54, mientras que los disparos repetidos caen drásticamente a 1,6. Esto indica que el modelo ha logrado una mejora significativa en la estrategia de disparo, maximizando el aprovechamiento de la información obtenida en cada partida y optimizando su toma de decisiones.

En resumen, la evolución observada en la tabla confirma que el agente de RL ha aprendido a jugar de manera más eficiente con el tiempo, reduciendo tanto la cantidad de disparos fallidos como la repetición innecesaria de intentos. Aunque la mejora no es drástica, el refinamiento progresivo de la estrategia demuestra la efectividad del aprendizaje por refuerzo en la optimización de decisiones en el juego Hundir la Flota.

4.5.2 Comparación con otras estrategias.

Al analizar los resultados obtenidos en la simulación del agente RL y compararlos con las estrategias heurísticas predefinidas, se observa que el modelo de RL logra un desempeño cercano a las mejores estrategias, aunque aún no las supera completamente.

El agente RL entrenado durante 5000 episodios alcanza un promedio de 71 turnos por partida, lo que lo sitúa por debajo de estrategias más básicas como la estrategia de caza (78 y 80 turnos dependiendo del oponente) o la búsqueda por zigzag (82 y 85 turnos). Sin embargo, sigue estando por encima de enfoques más ineficientes como la colocación en los bordes (85-87 turnos) y la colocación espaciada (88-90 turnos), que suelen aumentar la cantidad de turnos requeridos debido a su naturaleza más pasiva.

En términos de rendimiento, el agente RL aún no alcanza la eficacia de las estrategias más avanzadas. Por ejemplo, la búsqueda tras impacto optimizada, que reduce significativamente los turnos promedio a 60 y 58 turnos según el oponente, demuestra un mejor aprovechamiento de la información de impacto. Asimismo, las estrategias que incorporan memoria muestran un desempeño superior: memoria de barcos restantes alcanza 72 y 70 turnos, mientras que la memoria entre partidas se posiciona como la más efectiva con 45 y 43 turnos en promedio.

En conclusión, el agente RL ha demostrado una mejora considerable respecto a su estado inicial, pero aún requiere refinamientos para alcanzar el nivel de las estrategias más optimizadas. Su rendimiento actual es comparable a métodos intermedios como la memoria de barcos restantes, pero sigue siendo

superado por estrategias que combinan optimización de disparos y memoria de partidas previas. Esto sugiere que futuras mejoras en la estructura de entrenamiento o la estrategia de exploración podrían permitirle alcanzar o incluso superar el rendimiento de las mejores estrategias predefinidas.

5 CONCLUSIONES Y LÍNEAS FUTURAS

El desarrollo de estrategias de disparo en *Hundir la Flota* ha permitido evaluar distintas aproximaciones para mejorar la toma de decisiones de una IA en un entorno de información parcial. A lo largo del trabajo se han implementado estrategias heurísticas, probabilísticas y un modelo basado en RL, cada una con distintos niveles de efectividad.

Los resultados muestran que las estrategias heurísticas y probabilísticas logran mejoras significativas en la eficiencia de los disparos en comparación con una IA aleatoria. En particular, la memoria de barcos restantes y la estrategia probabilística han demostrado ser especialmente eficaces, reduciendo considerablemente el número de turnos necesarios para hundir los barcos del oponente. Por otro lado, el aprendizaje por refuerzo no ha logrado superar de manera clara a estas estrategias, mostrando un rendimiento mejorable.

El menor impacto del RL puede deberse a varios factores. En primer lugar, el tamaño del tablero de 10x10 y el límite de turnos impuesto restringen la cantidad de información que el agente puede recopilar por partida, lo que dificulta un aprendizaje profundo y eficiente. Además, la combinación de exploración y explotación podría no haber sido la más adecuada para este tipo de problema, limitando la capacidad del modelo para encontrar patrones óptimos en su estrategia de disparo.

5.1 Líneas futuras

Para mejorar el rendimiento del RL en *Hundir la Flota*, se pueden explorar varias vías:

- Aumento del tamaño del tablero: Utilizar un tablero más grande (por ejemplo, 15x15 o 20x20) podría proporcionar más información al agente, permitiendo que su aprendizaje sea más significativo y se reduzca la influencia del azar.
- Ajuste de la política de exploración-explotación: Se podrían probar estrategias más avanzadas de ajuste dinámico de ϵ (épsilon), como ϵ -decay adaptativo o estrategias basadas en *Upper Confidence Bound* (UCB), para mejorar la selección de acciones en función de la información acumulada.
- Modificación de la estructura de la red neuronal: Probar arquitecturas más profundas o con otros tipos de capas, como convolucionales (CNN), podría mejorar la capacidad del modelo para detectar patrones más complejos en el tablero.
- Cambio en la función de recompensa: Ajustar la ponderación de las recompensas podría incentivar una exploración más efectiva del tablero o evitar que el agente repita disparos ineficaces.

- Uso de técnicas de aprendizaje profundo adicionales: Explorar modelos basados en aprendizaje por refuerzo profundo (DRL) con *Double DQN* o *Prioritized Experience Replay* podría mejorar la estabilidad y eficiencia del entrenamiento.

En futuras iteraciones del proyecto, estas mejoras podrían ayudar a que el aprendizaje por refuerzo se convierta en una estrategia más competitiva frente a los métodos heurísticos y probabilísticos desarrollados en este trabajo. De esta manera, se podría lograr un agente con una estrategia de disparo más adaptable y eficiente en *Hundir la Flota*.

6 BIBLIOGRAFÍA

- [1] D. G. Link, «Nimrod and the Dawn of Digital Games», presentado en IEEE Annals of the History of Computing, 2003.
- [2] L. Samuel, «Some Studies in Machine Learning Using the Game of Checkers», presentado en IBM Journal of Research and Development, 1959.
- [3] M. Campbell, A. Hoane, y F. Hsu, «Deep Blue», *Artif. Intell.*, vol. 134, n.º 1-2, pp. 57-83, 2002.
- [4] D. Silver, «Mastering the game of Go with deep neural networks and tree search», *Nature*, vol. 529, n.º 7587, pp. 484-489, 2016.
- [5] M. Liu, *AlphaGo Simplified*, Primera. CRC Press, 2025.
- [6] «Descripción de la Estrategia». Accedido: 23 de enero de 2025. [En línea]. Disponible en: <https://www.tecnologiaeinnovacion.defensa.gob.es/es-es/Estrategia/Paginas/Descripcion.aspx>
- [7] «39308_Inteligencia_artificial.pdf». Accedido: 28 de enero de 2025. [En línea]. Disponible en: https://planetadelibrosec0.cdnstatics.com/libros_contenido_extra/40/39308_Inteligencia_artificial.pdf
- [8] J. McCarthy, «What is artificial intelligence?», *Stanf. Univ.*.
- [9] A. E. Vidal, «Alan Turing y el nacimiento de la inteligencia artificial».
- [10] BBVA, «¿Quién es el verdadero padre de la inteligencia artificial?», BBVA NOTICIAS. Accedido: 28 de enero de 2025. [En línea]. Disponible en: <https://www.bbva.com/es/innovacion/el-verdadero-padre-de-la-inteligencia-artificial/>
- [11] «Principales hitos y personajes de la Inteligencia Artificial timeline.», Timetoast Timelines. Accedido: 5 de febrero de 2025. [En línea]. Disponible en: <https://www.timetoast.com/timelines/principales-hitos-y-personajes-de-la-inteligencia-artificial>
- [12] M. A. Boden, *Inteligencia Artificial*, Turner Noema. Turner publicaciones S.L 2017, 2016.
- [13] A. Abeliuk, «Historia y evolución de la inteligencia artificial», p. 8.
- [14] F. Akhtar, *Practical Reinforcement Learning*. Packt Publishing, 2017.
- [15] J. L. Gonzalez, «Tipos de aprendizaje automático», SoldAI. Accedido: 11 de febrero de 2025. [En línea]. Disponible en: <https://medium.com/soldai/tipos-de-aprendizaje-autom%C3%A1tico-6413e3c615e2>
- [16] S. Navarro, «Red neuronal en Deep Learning: Guía completa [2025]». Accedido: 11 de febrero de 2025. [En línea]. Disponible en: <https://keepcoding.io/blog/red-neuronal-en-deep-learning/>
- [17] Y. LeCun, Y. Bengio, y G. Hinton, *Deep Learning*, vol. 521, 7553 vols. Nature, 2015.
- [18] I. Goodfellow, Y. Bengio, y A. Courville, «Deep learning», *MIT Press*, 2016.
- [19] P. Palanisamy, *Hands-on Intelligent Agents With OpenAI Gym*, Primera edición. Packt, 2018.
- [20] R. Sutton y A. Barto, «Reinforcement Learning: An Introduction», *MIT Press*, 2018.

- [21] D. Kingma y J. Ba, «Adam: A Method for Stochastic Optimization», presentado en International Conference on Learning Representations (ICLR), 2015.
- [22] V. Mnih et al, *Human-level control through deep reinforcement learning*, vol. 518. Nature, 2015.
- [23] H. Van Hasselt, A. Guez, y D. Silver, «Deep Reinforcement Learning with Double Q-learning», presentado en AAAI Conference on Artificial Intelligence, 2016.
- [24] R. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, vol. 8. en Machine Learning, vol. 8. 1992.
- [25] J. Schulman et al, «Proximal Policy Optimization Algorithms», *ArXiv Prepr. ArXiv170706347*, 2016.
- [26] V. Mnih et al, «Asynchronous methods for deep reinforcement learning», presentado en International Conference on Machine Learning (ICML), 2016.
- [27] O. Vinyals et al, *Grandmaster level in StarCraft II using multi-agent reinforcement learning*, vol. 575. Nature, 2019.
- [28] S. Levine et al, «End-to-end training of deep visuomotor policies», *J. Mach. Learn. Res.*, vol. 17, n.º 39, pp. 1-40, 2016.
- [29] J. Moody y M. Saffell, «Learning to trade via direct reinforcement», *IEEE Trans. Neural Netw.*, vol. 12, n.º 4, pp. 875-889, 2001.
- [30] J. Esteva et al, «Dermatologist-level classification of skin cancer with deep neural networks», *Nature*, vol. 542, n.º 7639, pp. 115-118, 2017.
- [31] C. D. Moya, «El GO, juego de estrategia que se aprende en 10 minutos y se perfecciona toda la vida», Radio UNR. Accedido: 6 de febrero de 2025. [En línea]. Disponible en: <https://radio.unr.edu.ar/2023/08/15/el-go-juego-de-estrategia-que-se-aprende-en-10-minutos-y-se-perfecciona-toda-la-vida/>
- [32] D. Burón, «AlphaGo, la IA de DeepMind, sigue los pasos de Deep Blue y gana al campeón mundial de Go», ITespresso.es. Accedido: 6 de febrero de 2025. [En línea]. Disponible en: <https://www.itespresso.es/alphago-la-ia-de-deepmind-sigue-los-pasos-de-deep-blue-y-gana-al-campeon-mundial-de-go-148952.html>
- [33] D. Silver et al, «Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm», 2017.
- [34] R. Coulom, «Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search», *Lect. Notes Comput. Sci.*, vol. 4630, pp. 72-83, 2007.
- [35] R. Coulom, «How AlphaGo Mastered the Game of Go», *IEEE Spectr.*, vol. 54, n.º 2, pp. 46-51, 2017.
- [36] D. Silver et al, «Mastering the game of Go without human knowledge», *Nature*, vol. 550, n.º 7676, pp. 354-359, 2017.
- [37] D. Silver et al, «A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play», *Science*, vol. 362, n.º 6419, pp. 1140-1144, 2018.
- [38] A. Krizhevsky, I. Sutskever, y G. Hinton, «Imagenet Classification with Deep Convolutional Neural Networks», *Adv. Neural Inf. Process. Syst. NeurIPS*, 2017.
- [39] A. Vaswani, «Attention Is All You Need», *Adv. Neural Inf. Process. Syst. NeuroIPS*, 2017.
- [40] «AlphaStar: Mastering the real-time strategy game StarCraft II», Google DeepMind. Accedido: 11 de febrero de 2025. [En línea]. Disponible en: <https://deepmind.google/discover/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii/>
- [41] V. Mnih et al, «Counterfactual multi-agent policy gradients», presentado en International Conference on Machine Learning, 2016.
- [42] J. Schmidhuber, «Deep learning in neural networks: An overview», *Neural Netw.*, vol. 61, pp. 85-117, 2015.
- [43] S. Hochreiter y Schmidhuber, J., «Long short-term memory», *Neural Comput.*, vol. 9, n.º 8, pp. 1735-1997.

- [44] V. Bolón-Canedo, L. Morán-Fernández, B. Cancela, y A. Alonso-Betanzos, «A review of green artificial intelligence: Towards a more sustainable future», *Neurocomputing*, vol. 599, p. 128096, sep. 2024, doi: 10.1016/j.neucom.2024.128096.
- [45] C. SER, «El Gobierno central invierte 67 millones en Multiverse Computing para impulsar la IA y la computación cuántica», Cadena SER. Accedido: 22 de marzo de 2025. [En línea]. Disponible en: <https://cadenaser.com/euskadi/2025/03/04/gobierno-central-invierte-67-millones-en-multiverse-computing-para-impulsar-la-ia-y-la-computacion-cuantica-radio-san-sebastian/>
- [46] M. Nielsen, «Is AlphaGo Really Such a Big Deal?», *Quanta Magazine*. Accedido: 21 de marzo de 2025. [En línea]. Disponible en: <https://www.quantamagazine.org/is-alphago-really-such-a-big-deal-20160329/>
- [47] «Stop Killer Robots», Stop Killer Robots. Accedido: 21 de marzo de 2025. [En línea]. Disponible en: <https://www.stopkillerrobots.org/stop-killer-robots/>
- [48] R. Jiménez-Segovia, «Los sistemas de armas autónomos en la Convención sobre ciertas armas convencionales: Sombras legales y éticas de una autonomía ¿bajo el control humano?», *Rev. Electrónica Estud. Int.*, n.º 37, jun. 2019, doi: 10.17103/reei.37.07.

ANEXO I: IMPLICACIONES AMBIENTALES

En el desarrollo de este trabajo, la implementación y entrenamiento de los modelos de IA se han llevado a cabo en un entorno a pequeña escala, utilizando recursos computacionales limitados. Por lo tanto, el impacto ambiental asociado ha sido mínimo.

Sin embargo, a medida que los problemas sobre los que se aplica la IA se vuelven más complejos, surge la preocupación por el consumo energético significativo que los modelos avanzados pueden requerir. Este aumento en la demanda energética ha llevado al surgimiento del concepto de *Green AI*, que busca mitigar el impacto ambiental de la IA mediante la optimización de algoritmos, la mejora de la eficiencia del hardware y la adopción de prácticas de desarrollo sostenibles [44].

Un ejemplo relevante es el caso de *AlphaStar*, el agente de inteligencia artificial desarrollado por *DeepMind* para jugar al videojuego *StarCraft II*. A pesar de tratarse de un entorno de juego, su entrenamiento requirió una enorme cantidad de recursos computacionales, incluyendo varios días de simulación en paralelo con cientos de instancias del juego ejecutándose simultáneamente. Este tipo de entrenamientos intensivos pone de manifiesto cómo, incluso en el ámbito de los videojuegos, el uso de IA avanzada puede tener un impacto ambiental significativo debido a su elevado consumo energético [27].

Además, iniciativas como la de *Multiverse Computing* en España buscan optimizar los modelos de IA para reducir su tamaño y consumo energético. Esta empresa ha desarrollado tecnología que permite comprimir modelos de IA hasta un 10% de su tamaño original sin disminuir su eficiencia, lo que puede reducir el consumo energético en centros de datos hasta en un 50% [45].

En resumen, aunque este trabajo específico no ha tenido un impacto ambiental significativo debido a su escala reducida, es esencial reconocer la importancia de desarrollar y adoptar prácticas de *Green AI*. Estas prácticas buscan minimizar la huella de carbono de la IA, promoviendo un equilibrio entre el avance tecnológico y la sostenibilidad ambiental.

ANEXO II: REFLEXIONES ÉTICAS Y SOCIALES

El desarrollo de la IA ha traído consigo importantes avances en diversos campos, desde la optimización de estrategias en juegos hasta aplicaciones más críticas como la toma de decisiones en entornos reales. En este trabajo, se ha aplicado IA en el contexto de *Hundir la Flota*, un entorno donde las decisiones estratégicas y el aprendizaje a partir de la experiencia permiten evaluar la eficacia de distintos enfoques algorítmicos, incluido el RL. Sin embargo, el impacto de la IA no se limita a los juegos: su uso en escenarios más complejos, como la automatización de decisiones en ámbitos militares o en la vida cotidiana, plantea cuestionamientos éticos fundamentales.

Un ejemplo destacado en la historia de la IA aplicada a juegos es *AlphaGo*, el sistema desarrollado por DeepMind que, en 2016, logró vencer a campeones humanos en el juego de *Go*. Este hito demostró que una IA podía desarrollar estrategias innovadoras y jugar de manera no intuitiva, lo que generó debates sobre el potencial de las máquinas para la creatividad y la innovación [46]. La capacidad de estos sistemas para aprender y tomar decisiones sin intervención humana plantea preguntas sobre hasta qué punto la IA puede reemplazar o superar la intuición y el razonamiento humanos.

Por otro lado, el uso de IA en sistemas de armas autónomas letales (LAWS, *lethal autonomous weapon system*) ha generado gran preocupación a nivel internacional. Estos sistemas pueden identificar y atacar objetivos sin intervención humana significativa, lo que abre un debate sobre la delegación de decisiones de vida o muerte a máquinas. Organizaciones como *Stop Killer Robots* advierten sobre los riesgos de estas tecnologías y abogan por su prohibición, argumentando que pueden deshumanizar los conflictos y reducir la responsabilidad en el uso de la fuerza [47].

El Comité Internacional de la Cruz Roja (CICR) ha señalado que el uso de armas autónomas plantea desafíos en relación con el cumplimiento del derecho internacional humanitario, en particular con los principios de distinción, proporcionalidad y responsabilidad. La falta de un control humano significativo sobre estos sistemas plantea interrogantes sobre la ética de permitir que algoritmos determinen cuándo y cómo se usa la fuerza[48].

En conclusión, mientras que este trabajo se ha centrado en la aplicación de IA en un contexto lúdico y controlado, el mismo tipo de tecnología puede tener implicaciones profundas cuando se traslada a ámbitos más críticos. La capacidad de los algoritmos para aprender, adaptarse y tomar decisiones de manera autónoma es una herramienta poderosa, pero su implementación en áreas sensibles como la seguridad, la justicia y el ámbito militar requiere un marco ético sólido que garantice que su uso respete los valores fundamentales de la sociedad.

ANEXO III: CÓDIGO DE LA ESTRATEGIA ALEATORIA

A3.1 Aleatorio vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def disparo_maquina(tablero):
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
```

```
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0
    turnos_por_partida = [] # Lista para almacenar la cantidad de turnos de cada partida

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        turnos = 0 # Contador de turnos para esta partida

        # Bucle de juego: cada ronda ambas máquinas disparan
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            # Máquina 1 dispara (su disparo es aleatorio)
            tablero_maquina_2 = disparo_maquina(tablero_maquina_2)
            # Máquina 2 dispara (su disparo es aleatorio)
            tablero_maquina_1 = disparo_maquina(tablero_maquina_1)

            # Evaluar el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            # Si ambas máquinas han perdido sus barcos, se considera empate
            if not m1 and not m2:
                empates += 1
                break
            # Si la Máquina 2 se queda sin barcos, gana la Máquina 1
            elif not m2:
                victorias_maquina_1 += 1
                break
            # Si la Máquina 1 se queda sin barcos, gana la Máquina 2
            elif not m1:
                victorias_maquina_2 += 1
                break

        turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")

    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Aleatorio vs Aleatorio")
plt.show()
```

A3.2 Aleatorio vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt
```

```
# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
    ]
```

```
# Disposición 7: Mezclada, horizontales y verticales
[("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
# Disposición 8: Todos horizontales en un bloque central
[("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
# Disposición 9: Todos verticales en un bloque central
[("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
# Disposición 10: Mezcla de orientaciones
[("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
]
disp = random.choice(disposiciones)
# Para cada barco en la disposición, se comprueba y se coloca:
for barco, fila, columna, orientacion in disp:
    tamaño = BARCOS[barco]
    if orientacion == "H":
        # Comprobar límite horizontal
        if columna + tamaño > BOARD_SIZE:
            # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
            continue
        # Comprobar que no se solape con otro barco
        if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
    else: # "V"
        if fila + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
return tablero

def disparo_maquina(tablero):
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0 # Contador de turnos para esta partida

        # Bucle de juego: cada ronda ambas máquinas disparan
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            # Máquina 1 dispara (usa su estrategia de caza y eliminación)
            tablero_maquina_2 = disparo_maquina(tablero_maquina_2)
```

```

# Máquina 2 dispara (su disparo es aleatorio, según la función definida)
tablero_maquina_1 = disparo_maquina(tablero_maquina_1)

# Evaluar el estado de ambos tableros
m1 = quedan_barcos(tablero_maquina_1)
m2 = quedan_barcos(tablero_maquina_2)

# Si ambas máquinas han perdido sus barcos, se considera empate
if not m1 and not m2:
    empates += 1
    break
# Si la Máquina 2 se queda sin barcos, gana la Máquina 1
elif not m2:
    victorias_maquina_1 += 1
    break
# Si la Máquina 1 se queda sin barcos, gana la Máquina 2
elif not m1:
    victorias_maquina_2 += 1
    break
turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Aleatorio vs Humano")
plt.show()

```

ANEXO IV: CÓDIGO ESTRATEGIA DE CAZA

A4.1 Estrategia de caza vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
```

```

        # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
        área
        for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nf, nc = fila + df, columna + dc
            if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                memoria.append((nf, nc))
        return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0 # Contador de turnos para esta partida

        # Bucle de juego: cada ronda ambas máquinas disparan
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            # Máquina 1 dispara (usa su estrategia de caza y eliminación)
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            # Máquina 2 dispara (su disparo es aleatorio, según la función definida)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluar el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            # Si ambas máquinas han perdido sus barcos, se considera empate
            if not m1 and not m2:
                empates += 1
                break
            # Si la Máquina 2 se queda sin barcos, gana la Máquina 1
            elif not m2:
                victorias_maquina_1 += 1
                break
            # Si la Máquina 1 se queda sin barcos, gana la Máquina 2
            elif not m1:
                victorias_maquina_2 += 1
                break

        turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

```

```
# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Estrategia de caza vs Aleatorio")
plt.show()
```

A4.2 Estrategia de caza vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
```

Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.

```

"""
disposiciones = [
    # Disposición 1: Todos horizontales en la parte izquierda
    [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
    ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
    # Disposición 2: Todos verticales en la esquina superior izquierda
    [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
    ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
    # Disposición 3: Todos horizontales en la parte inferior
    [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
    ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
    # Disposición 4: Mezclada: bordes opuestos
    [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
    ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
    # Disposición 5: Todos horizontales en filas centrales
    [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
    ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
    # Disposición 6: Todos verticales en columnas centrales
    [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
    ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
    # Disposición 7: Mezclada, horizontales y verticales
    [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
    ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
    # Disposición 8: Todos horizontales en un bloque central
    [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
    ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
    # Disposición 9: Todos verticales en un bloque central
    [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
    ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
    # Disposición 10: Mezcla de orientaciones
    [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
    ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
]
disp = random.choice(disposiciones)
# Para cada barco en la disposición, se comprueba y se coloca:
for barco, fila, columna, orientacion in disp:
    tamaño = BARCOS[barco]
    if orientacion == "H":
        # Comprobar límite horizontal
        if columna + tamaño > BOARD_SIZE:
            # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
            continue
        # Comprobar que no se solape con otro barco
        if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
    else: # "V"
        if fila + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua

```

```
        else:
            tablero[filas, columnas] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = filas + df, columnas + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        filas = random.randint(0, BOARD_SIZE - 1)
        columnas = random.randint(0, BOARD_SIZE - 1)
        if tablero[filas, columnas] not in ["X", "O"]:
            if tablero[filas, columnas] == "~":
                tablero[filas, columnas] = "O" # Agua
            else:
                tablero[filas, columnas] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0 # Contador de turnos para esta partida

        # Bucle de juego: cada ronda ambas máquinas disparan
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            # Máquina 1 dispara (usa su estrategia de caza y eliminación)
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            # Máquina 2 dispara (su disparo es aleatorio, según la función definida)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluar el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            # Si ambas máquinas han perdido sus barcos, se considera empate
            if not m1 and not m2:
                empates += 1
                break
            # Si la Máquina 2 se queda sin barcos, gana la Máquina 1
            elif not m2:
                victorias_maquina_1 += 1
                break
            # Si la Máquina 1 se queda sin barcos, gana la Máquina 2
            elif not m1:
                victorias_maquina_2 += 1
                break

        turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
```

```

    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Estrategia de caza vs Humano")
plt.show()

```

ANEXO V: CÓDIGO DE PATRÓN DE BÚSQUEDA

A5.1 Patrón de búsqueda vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def generar_patron_búsqueda():
    """Genera un patrón de búsqueda tipo damero para mejorar la eficiencia."""
    patron = [(fila, columna) for fila in range(BOARD_SIZE) for columna in
range(BOARD_SIZE) if (fila + columna) % 2 == 0]
    random.shuffle(patron) # Mezclar el patrón para mayor variabilidad
    return patron

def disparo_maquina_1(tablero, memoria, patron_búsqueda):
    """Máquina 1 con estrategia de caza y patrón de búsqueda."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif patron_búsqueda:
        fila, columna = patron_búsqueda.pop(0)
```

```

else:
    fila = random.randint(0, BOARD_SIZE - 1)
    columna = random.randint(0, BOARD_SIZE - 1)

if tablero[fila, columna] not in ["X", "O"]:
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        tablero[fila, columna] = "X" # Impacto
        # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
        # área
        for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nf, nc = fila + df, columna + dc
            if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                memoria.append((nf, nc))
return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        patron_busqueda_1 = generar_patron_busqueda()
        turnos = 0 # Contador de turnos para esta partida

        # Cada ronda ambas máquinas disparan, luego se evalúa el resultado
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            patron_busqueda_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
            turnos_por_partida.append(turnos)

```

```
print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Patrón de búsqueda vs Aleatorio")
plt.show()
```

A5.2 Patrón de búsqueda vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
```

```

    return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila, columna + i] = barco[0]
        else: # "V"
            if fila + tamaño > BOARD_SIZE:
                continue
            if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
    return tablero

def generar_patron_búsqueda():
    """Genera un patrón de búsqueda tipo damero para mejorar la eficiencia."""
    patron = [(fila, columna) for fila in range(BOARD_SIZE) for columna in
    range(BOARD_SIZE) if (fila + columna) % 2 == 0]
    random.shuffle(patron) # Mezclar el patrón para mayor variabilidad
    return patron

```

```
def disparo_maquina_1(tablero, memoria, patron_busqueda):
    """Máquina 1 con estrategia de caza y patrón de búsqueda."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif patron_busqueda:
        fila, columna = patron_busqueda.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
```

```
                memoria.append((nf, nc))
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        patron_busqueda_1 = generar_patron_busqueda()
        turnos = 0 # Contador de turnos para esta partida

        # Cada ronda ambas máquinas disparan, luego se evalúa el resultado
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            patron_busqueda_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
```

```

        break
    elif not m1:
        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Patrón de búsqueda vs Humano")
plt.show()

```

ANEXO VI: CÓDIGO DE BÚSQUEDA POR CUADRANTES

A6.1 Búsqueda por cuadrantes vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def generar_cuadrantes():
    """Genera una lista de disparos en cuadrantes para mejorar la búsqueda."""
    cuadrantes = []
    tam_cuadrante = BOARD_SIZE // 2 # Divide el tablero en 4 cuadrantes de 5x5
    for i in range(0, BOARD_SIZE, tam_cuadrante):
        for j in range(0, BOARD_SIZE, tam_cuadrante):
            cuadrante = [(x, y) for x in range(i, i + tam_cuadrante) for y in range(j, j +
tam_cuadrante)]
            random.shuffle(cuadrante) # Mezclar para mayor variabilidad
            cuadrantes.extend(cuadrante)
    return cuadrantes

def disparo_maquina_1(tablero, memoria, disparos_cuadrantes):
```

```

    """Máquina 1 con estrategia de caza y búsqueda por cuadrantes."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif disparos_cuadrantes:
        fila, columna = disparos_cuadrantes.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        disparos_cuadrantes_1 = generar_cuadrantes()
        turnos = 0 # Contador de turnos para esta partida

        # Cada ronda ambas máquinas disparan, luego se evalúa el resultado
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
disparos_cuadrantes_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break

```

```
        elif not m1:
            victorias_maquina_2 += 1
            break
    turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Búsqueda por cuadrantes vs Aleatorio")
plt.show()
```

A6.2 Búsqueda por cuadrantes vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
```

```

        elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
            colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila, columna + i] = barco[0]
        else: # "V"
            if fila + tamaño > BOARD_SIZE:
                continue
            if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
    return tablero

def generar_cuadrantes():
    """Genera una lista de disparos en cuadrantes para mejorar la búsqueda."""

```

```
cuadrantes = []
tam_cuadrante = BOARD_SIZE // 2 # Divide el tablero en 4 cuadrantes de 5x5
for i in range(0, BOARD_SIZE, tam_cuadrante):
    for j in range(0, BOARD_SIZE, tam_cuadrante):
        cuadrante = [(x, y) for x in range(i, i + tam_cuadrante) for y in range(j, j +
tam_cuadrante)]
        random.shuffle(cuadrante) # Mezclar para mayor variabilidad
        cuadrantes.extend(cuadrante)
    return cuadrantes

def disparo_maquina_1(tablero, memoria, disparos_cuadrantes):
    """Máquina 1 con estrategia de caza y búsqueda por cuadrantes."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif disparos_cuadrantes:
        fila, columna = disparos_cuadrantes.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
                break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        disparos_cuadrantes_1 = generar_cuadrantes()
        turnos = 0 # Contador de turnos para esta partida

        # Cada ronda ambas máquinas disparan, luego se evalúa el resultado
        while True:
            turnos += 1 # Incrementar el contador en cada ronda
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
disparos_cuadrantes_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)
```

```

# Evaluamos el estado de ambos tableros
m1 = quedan_barcos(tablero_maquina_1)
m2 = quedan_barcos(tablero_maquina_2)

if not m1 and not m2:
    empates += 1
    break
elif not m2:
    victorias_maquina_1 += 1
    break
elif not m1:
    victorias_maquina_2 += 1
    break
turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Búsqueda por cuadrantes vs Humano")
plt.show()

```

ANEXO VII: CÓDIGO DE BÚSQUEDA EN ZIGZAG

A7.1 Búsqueda en zigzag vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def generar_patron_zigzag():
    """Genera un patrón de disparo en zigzag para mejorar la búsqueda."""
    zigzag = []
    for fila in range(BOARD_SIZE):
        if fila % 2 == 0:
            for columna in range(BOARD_SIZE):
                zigzag.append((fila, columna))
        else:
            for columna in reversed(range(BOARD_SIZE)):
                zigzag.append((fila, columna))
    return zigzag

def disparo_maquina_1(tablero, memoria, disparos_zigzag):
```

```

    """Máquina 1 con estrategia de caza y búsqueda en zigzag."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif disparos_zigzag:
        fila, columna = disparos_zigzag.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        disparos_zigzag_1 = generar_patron_zigzag()
        turnos = 0 # Contador de turnos para esta partida
        # En cada ronda ambas máquinas disparan y luego se evalúa el estado de los tableros
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
disparos_zigzag_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:

```

```
        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Búsqueda por zigzag vs Aleatorio")
plt.show()
```

A7.2 Búsqueda en zigzag vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
```

```

        elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
            colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila, columna + i] = barco[0]
        else: # "V"
            if fila + tamaño > BOARD_SIZE:
                continue
            if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
    return tablero

def generar_patron_zigzag():
    """Genera un patrón de disparo en zigzag para mejorar la búsqueda."""

```

```
zigzag = []
for fila in range(BOARD_SIZE):
    if fila % 2 == 0:
        for columna in range(BOARD_SIZE):
            zigzag.append((fila, columna))
    else:
        for columna in reversed(range(BOARD_SIZE)):
            zigzag.append((fila, columna))
return zigzag

def disparo_maquina_1(tablero, memoria, disparos_zigzag):
    """Máquina 1 con estrategia de caza y búsqueda en zigzag."""
    if memoria:
        fila, columna = memoria.pop(0)
    elif disparos_zigzag:
        fila, columna = disparos_zigzag.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        disparos_zigzag_1 = generar_patron_zigzag()
        turnos = 0 # Contador de turnos para esta partida
        # En cada ronda ambas máquinas disparan y luego se evalúa el estado de los tableros
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
disparos_zigzag_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)
```

```

# Evaluamos el estado de ambos tableros
m1 = quedan_barcos(tablero_maquina_1)
m2 = quedan_barcos(tablero_maquina_2)

if not m1 and not m2:
    empates += 1
    break
elif not m2:
    victorias_maquina_1 += 1
    break
elif not m1:
    victorias_maquina_2 += 1
    break
turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Búsqueda por zigzag vs Humano")
plt.show()

```

ANEXO VIII: CÓDIGO DE COLOCACIÓN ESPACIADA

A8.1 Colocación espaciada vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos_espaciado(tablero):
    """Coloca los barcos con una distribución más separada para mayor defensa."""
    intentos_max = 50 # Evita bucles infinitos
    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0
        while not colocado and intentos < intentos_max:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            # Verifica que el barco no toque otro barco y que haya espacio alrededor
            if orientacion == "H" and columna + tamaño <= BOARD_SIZE:
                if all(tablero[fila, columna + i] == "~" for i in range(tamaño)) and all(
                    0 <= columna + i < BOARD_SIZE and 0 <= fila < BOARD_SIZE and
tablero[fila, max(0, columna + i - 1): min(BOARD_SIZE, columna + i +
2)].tolist().count("~") == min(BOARD_SIZE, columna + i + 2) - max(0, columna + i - 1)
                    for i in range(tamaño)
                ):
                    for i in range(tamaño):
                        tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE:
                if all(tablero[fila + i, columna] == "~" for i in range(tamaño)) and all(
                    0 <= fila + i < BOARD_SIZE and 0 <= columna < BOARD_SIZE and
tablero[max(0, fila + i - 1): min(BOARD_SIZE, fila + i + 2), columna].tolist().count("~")
                    == min(BOARD_SIZE, fila + i + 2) - max(0, fila + i - 1)
                    for i in range(tamaño)
                ):
                    for i in range(tamaño):
                        tablero[fila + i, columna] = barco[0]
                    colocado = True
            intentos += 1
```

```

    return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
                ["X", "O"]:
                    memoria.append((nf, nc))
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos_espaciado(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0 # Contador de turnos para esta partida

        # En cada ronda, ambas máquinas disparan y luego se evalúa el estado
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:

```

```
        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Colocación espaciada vs Aleatorio")
plt.show()
```

A8.2 Colocación espaciada vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos_espaciado(tablero):
    """Coloca los barcos con una distribución más separada para mayor defensa."""
    intentos_max = 50 # Evita bucles infinitos
    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0
        while not colocado and intentos < intentos_max:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            # Verifica que el barco no toque otro barco y que haya espacio alrededor
            if orientacion == "H" and columna + tamaño <= BOARD_SIZE:
                if all(tablero[fila, columna + i] == "~" for i in range(tamaño)) and all(
```

```

        0 <= columna + i < BOARD_SIZE and 0 <= fila < BOARD_SIZE and
tablero[fila, max(0, columna + i - 1): min(BOARD_SIZE, columna + i +
2)].tolist().count("~") == min(BOARD_SIZE, columna + i + 2) - max(0, columna + i - 1)
        for i in range(tamaño)
    ):
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
            colocado = True
        elif orientacion == "V" and fila + tamaño <= BOARD_SIZE:
            if all(tablero[fila + i, columna] == "~" for i in range(tamaño)) and all(
                0 <= fila + i < BOARD_SIZE and 0 <= columna < BOARD_SIZE and
tablero[max(0, fila + i - 1): min(BOARD_SIZE, fila + i + 2), columna].tolist().count("~")
== min(BOARD_SIZE, fila + i + 2) - max(0, fila + i - 1)
            for i in range(tamaño)
        ):
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
                colocado = True
            intentos += 1
    return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):

```

```
        continue
    for i in range(tamaño):
        tablero[fila, columna + i] = barco[0]
    else: # "v"
        if fila + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
    return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos_espaciado(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0

        # En cada ronda, ambas máquinas disparan y luego se evalúa el estado
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
```

```

m1 = quedan_barcos(tablero_maquina_1)
m2 = quedan_barcos(tablero_maquina_2)

if not m1 and not m2:
    empates += 1
    break
elif not m2:
    victorias_maquina_1 += 1
    break
elif not m1:
    victorias_maquina_2 += 1
    break
turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Colocación espaciada vs Humano")
plt.show()

```

ANEXO IX: CÓDIGO DE COLOCACIÓN EN LOS BORDES

A9.1 Colocación en los bordes vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos_en_bordes(tablero):
    """Coloca los barcos preferentemente en los bordes del tablero."""
    intentos_max = 50 # Evita bucles infinitos
    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0
        while not colocado and intentos < intentos_max:
            # Elegir una fila o columna en los bordes
            if random.choice([True, False]): # Decide si colocar en filas borde
                fila = random.choice([0, BOARD_SIZE - 1])
                columna = random.randint(0, BOARD_SIZE - tamaño)
                orientacion = "H"
            else: # O en columnas borde
                fila = random.randint(0, BOARD_SIZE - tamaño)
                columna = random.choice([0, BOARD_SIZE - 1])
                orientacion = "V"

            # Verificar si la posición está libre
            if orientacion == "H" and all(tablero[fila, columna + i] == "~" for i in
range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and all(tablero[fila + i, columna] == "~" for i in
range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
            intentos += 1
        return tablero

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
```

```

colocado = False
while not colocado:
    fila = random.randint(0, BOARD_SIZE - 1)
    columna = random.randint(0, BOARD_SIZE - 1)
    orientacion = random.choice(["H", "V"])

    if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
            colocado = True
    elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
            colocado = True
return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
                break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos_en_bordes(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado

```

```
while True:
    turnos += 1
    tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
    tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

    # Evaluamos el estado de ambos tableros
    m1 = quedan_barcos(tablero_maquina_1)
    m2 = quedan_barcos(tablero_maquina_2)

    if not m1 and not m2:
        empates += 1
        break
    elif not m2:
        victorias_maquina_1 += 1
        break
    elif not m1:
        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Colocación en los bordes vs Aleatorio")
plt.show()
```

A9.2 Colocación en los bordes vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
```

```

        print(str(i) + " " + " ".join(fila))

def colocar_barcos_en_bordes(tablero):
    """Coloca los barcos preferentemente en los bordes del tablero."""
    intentos_max = 50 # Evita bucles infinitos
    for barco, tamaño in BARCOS.items():
        colocado = False
        intentos = 0
        while not colocado and intentos < intentos_max:
            # Elegir una fila o columna en los bordes
            if random.choice([True, False]): # Decide si colocar en filas borde
                fila = random.choice([0, BOARD_SIZE - 1])
                columna = random.randint(0, BOARD_SIZE - tamaño)
                orientacion = "H"
            else: # O en columnas borde
                fila = random.randint(0, BOARD_SIZE - tamaño)
                columna = random.choice([0, BOARD_SIZE - 1])
                orientacion = "V"

            # Verificar si la posición está libre
            if orientacion == "H" and all(tablero[fila, columna + i] == "~" for i in
range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and all(tablero[fila + i, columna] == "~" for i in
range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
            intentos += 1
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]

```

```
]
disp = random.choice(disposiciones)
# Para cada barco en la disposición, se comprueba y se coloca:
for barco, fila, columna, orientacion in disp:
    tamaño = BARCOS[barco]
    if orientacion == "H":
        # Comprobar límite horizontal
        if columna + tamaño > BOARD_SIZE:
            # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
            continue
        # Comprobar que no se solape con otro barco
        if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila, columna + i] = barco[0]
    else: # "V"
        if fila + tamaño > BOARD_SIZE:
            continue
        if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[fila + i, columna] = barco[0]
return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_1(tablero, memoria):
    """Máquina 1 con estrategia de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            # Agregar posiciones adyacentes a la memoria para seguir disparando en la misma
            área
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                if 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE and tablero[nf, nc] not in
["X", "O"]:
                    memoria.append((nf, nc))
            return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
```

```

while True:
    fila = random.randint(0, BOARD_SIZE - 1)
    columna = random.randint(0, BOARD_SIZE - 1)
    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
        break
return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos_en_bordes(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            # Evaluamos el estado de ambos tableros
            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
        turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Colocación en los bordes vs Humano")
plt.show()

```

ANEXO X: CÓDIGO DE OPTIMIZACIÓN DE BÚSQUEDA TRAS IMPACTO

A10.1 Optimización de búsqueda tras impacto vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def disparo_maquina_1(tablero, memoria, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
```

```

else:
    tablero[fila, columna] = "X" # Impacto
    if objetivo is None:
        objetivo = (fila, columna)
        # Explorar solo en líneas rectas (horizontal o vertical), sin diagonales
        for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nf, nc = fila + df, columna + dc
            while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                if tablero[nf, nc] == "~":
                    break # Detener la búsqueda en esta dirección si encuentra
agua
                if tablero[nf, nc] not in ["X", "O"]:
                    memoria.append((nf, nc))
                nf += df
                nc += dc

return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        objetivo_maquina_1 = None
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado de sus tableros
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
            turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")

```

```
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Optimización de búsqueda tras impacto vs Aleatorio")
plt.show()
```

A10.2 Optimización de búsqueda tras impacto vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero
```

```

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila, columna + i] = barco[0]
        else: # "V"
            if fila + tamaño > BOARD_SIZE:
                continue
            if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
    return tablero

def disparo_maquina_1(tablero, memoria, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación."""
    if memoria:
        fila, columna = memoria.pop(0)
    else:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    if tablero[fila, columna] not in ["X", "O"]:

```

```
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        tablero[fila, columna] = "X" # Impacto
        if objetivo is None:
            objetivo = (fila, columna)
            # Explorar solo en líneas rectas (horizontal o vertical), sin diagonales
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                    if tablero[nf, nc] == "~":
                        break # Detener la búsqueda en esta dirección si encuentra
agua
                    if tablero[nf, nc] not in ["X", "O"]:
                        memoria.append((nf, nc))
                    nf += df
                    nc += dc

    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        objetivo_maquina_1 = None
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado de sus tableros
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
            turnos_por_partida.append(turnos)
```

```

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Optimización de búsqueda tras impacto vs Humano")
plt.show()

```

ANEXO XI: CÓDIGO DE MEMORIA DE BARCOS RESTANTES

A11.1 Memoria de barcos restantes vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~",
"X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación que recuerda qué barcos
quedan."""
```

```

if not barcos_vivos:
    barcos_vivos = barcos_restantes(tablero)
tamano_maximo = max(barcos_vivos.values(), default=0)

if memoria:
    fila, columna = memoria.pop(0)
else:
    # Buscar una zona donde quepa el barco más grande restante
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if any(
            all(0 <= fila + i < BOARD_SIZE and tablero[fila + i, columna] not in ["X",
"O"]) for i in range(tamano_maximo)) or
            all(0 <= columna + i < BOARD_SIZE and tablero[fila, columna + i] not in
["X", "O"]) for i in range(tamano_maximo)):
            for tamano_maximo in barcos_vivos.values():
                break

if tablero[fila, columna] not in ["X", "O"]:
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        tablero[fila, columna] = "X" # Impacto
        if objetivo is None:
            objetivo = (fila, columna)
            # Explorar solo en líneas rectas (horizontal o vertical), sin diagonales
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                    if tablero[nf, nc] == "~":
                        break # Detener la búsqueda en esta dirección si encuentra
agua
                    if tablero[nf, nc] not in ["X", "O"]:
                        memoria.append((nf, nc))
                    nf += df
                    nc += dc

return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
                break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0

```

```
# En cada ronda ambas máquinas disparan y luego se evalúa el estado de los tableros
while True:
    turnos += 1
    tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
barcos_vivos, objetivo_maquina_1)
    tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

    m1 = quedan_barcos(tablero_maquina_1)
    m2 = quedan_barcos(tablero_maquina_2)

    if not m1 and not m2:
        empates += 1
        break
    elif not m2:
        victorias_maquina_1 += 1
        break
    elif not m1:
        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Memoria de barcos restantes vs Aleatorio")
plt.show()
```

A11.2 Memoria de barcos restantes vs Humano

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
```

```

        print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
    else:
        print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:

```

```
        # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
        continue
    # Comprobar que no se solape con otro barco
    if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
        continue
    for i in range(tamaño):
        tablero[fila, columna + i] = barco[0]
else: # "v"
    if fila + tamaño > BOARD_SIZE:
        continue
    if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
        continue
    for i in range(tamaño):
        tablero[fila + i, columna] = barco[0]
return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~",
"X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia mejorada de caza y eliminación que recuerda qué barcos
quedan."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero)
    tamano_maximo = max(barcos_vivos.values(), default=0)

    if memoria:
        fila, columna = memoria.pop(0)
    else:
        # Buscar una zona donde quepa el barco más grande restante
        while True:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            if any(
                all(0 <= fila + i < BOARD_SIZE and tablero[fila + i, columna] not in ["X",
"O"] for i in range(tamano_maximo)) or
                all(0 <= columna + i < BOARD_SIZE and tablero[fila, columna + i] not in
["X", "O"] for i in range(tamano_maximo))
                for tamano_maximo in barcos_vivos.values()
            ):
                break

    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
            if objetivo is None:
                objetivo = (fila, columna)
                # Explorar solo en líneas rectas (horizontal o vertical), sin diagonales
                for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    nf, nc = fila + df, columna + dc
                    while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                        if tablero[nf, nc] == "~":
                            break # Detener la búsqueda en esta dirección si encuentra
agua
                        if tablero[nf, nc] not in ["X", "O"]:
                            memoria.append((nf, nc))
                            nf += df
                            nc += dc

    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
```

```

    fila = random.randint(0, BOARD_SIZE - 1)
    columna = random.randint(0, BOARD_SIZE - 1)
    if tablero[fila, columna] not in ["X", "O"]:
        if tablero[fila, columna] == "~":
            tablero[fila, columna] = "O" # Agua
        else:
            tablero[fila, columna] = "X" # Impacto
        break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado de los tableros
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            barcos_vivos, objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
            turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Memoria de barcos restantes vs Humano")
plt.show()

```

ANEXO XII: CÓDIGO DE ESTRATEGIA PROBABILÍSTICA

A12.1 Estrategia probabilística vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~",
"X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

def calcular_mapa_probabilidades(tablero, barcos_vivos):
    """Calcula un mapa de probabilidades (matriz de enteros) en el que cada celda
acumula 1 por cada forma válida en que algún barco restante podría ubicarse allí.
Se consideran solo posiciones en las que la celda no haya sido disparada ("X" u
"O")."""
```

```

# Crear una matriz de ceros con las dimensiones del tablero
mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

# Recorrer cada celda del tablero
for i in range(BOARD_SIZE):
    for j in range(BOARD_SIZE):
        # Solo evaluar celdas que no han sido disparadas
        if tablero[i, j] not in ["X", "O"]:
            # Para cada barco restante
            for barco, tamaño in barcos_vivos.items():
                # --- Evaluación en orientación horizontal ---
                # Se busca un inicio válido para que el barco incluya la celda (i,j)
                # El inicio puede variar de max(0, j - tamaño + 1) hasta min(j,
BOARD_SIZE - tamaño)
                for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño
+ 1)):
                    # Verificar que todas las celdas desde inicio hasta inicio+tamaño
estén libres
                    if all(tablero[i, k] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                        mapa[i, j] += 1

                # --- Evaluación en orientación vertical ---
                # Se busca un inicio válido para que el barco incluya la celda (i,j)
                for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño
+ 1)):
                    if all(tablero[k, j] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                        mapa[i, j] += 1

    return mapa

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia de probabilidad y caza y eliminación."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero)

    # Si no hay objetivo activo y la memoria está vacía, usa el mapa de probabilidades
    if objetivo is None and not memoria:
        # Calcular el mapa de probabilidades
        mapa = calcular_mapa_probabilidades(tablero, barcos_vivos)

        # Imprimir el mapa de probabilidades para depuración
        print("Mapa de Probabilidades:")
        print(mapa)

        # Buscar el valor máximo de probabilidad
        max_prob = np.max(mapa)
        # Obtener todas las celdas (índices) que tienen ese valor máximo
        indices = np.argwhere(mapa == max_prob)
        # Elegir aleatoriamente una de las celdas de mayor probabilidad
        fila, columna = random.choice(indices)
    elif memoria:
        fila, columna = memoria.pop(0)
    else:
        # En caso de eventualidad, se elige aleatoriamente
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    # Realizar el disparo
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        # Guardar la letra del barco antes de marcar el impacto
        letra_barco = tablero[fila, columna]
        tablero[fila, columna] = "X" # Impacto

        # Comprobar si el barco ha sido hundido
        if not np.any(tablero == letra_barco):
            barcos_vivos = barcos_restantes(tablero)
            objetivo = None

```

```
        memoria.clear()
    else:
        # Si es el primer impacto o se continúa la caza, activar la estrategia de caza
        if objetivo is None:
            objetivo = (fila, columna)
            # Explorar en las 4 direcciones (horizontal y vertical)
            for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nf, nc = fila + df, columna + dc
                while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                    if tablero[nf, nc] == "~":
                        break # Se detiene si se encuentra agua
                    if tablero[nf, nc] not in ["X", "O"]:
                        memoria.append((nf, nc))
                    nf += df
                    nc += dc
        return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            barcos_vivos, objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
                victorias_maquina_2 += 1
                break
            turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
```

```

print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Estrategia probabilística vs Aleatorio")
plt.show()

```

A12.2 Estrategia probabilística vs Humano

```

import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                colocado = True
        return tablero

```

```

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
        ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
        ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
        ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
        ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
        ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
        ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
        ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
        ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
        ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
        ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
        if orientacion == "H":
            # Comprobar límite horizontal
            if columna + tamaño > BOARD_SIZE:
                # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
                continue
            # Comprobar que no se solape con otro barco
            if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila, columna + i] = barco[0]
        else: # "V"
            if fila + tamaño > BOARD_SIZE:
                continue
            if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                continue
            for i in range(tamaño):
                tablero[fila + i, columna] = barco[0]
    return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~",
    "X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

```

```

def calcular_mapa_probabilidades(tablero, barcos_vivos):
    """Calcula un mapa de probabilidades (matriz de enteros) en el que cada celda
    acumula 1 por cada forma válida en que algún barco restante podría ubicarse allí.
    Se consideran solo posiciones en las que la celda no haya sido disparada ("X" u
    "O")."""
    # Crear una matriz de ceros con las dimensiones del tablero
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

    # Recorrer cada celda del tablero
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            # Solo evaluar celdas que no han sido disparadas
            if tablero[i, j] not in ["X", "O"]:
                # Para cada barco restante
                for barco, tamaño in barcos_vivos.items():
                    # --- Evaluación en orientación horizontal ---
                    # Se busca un inicio válido para que el barco incluya la celda (i,j)
                    # El inicio puede variar de max(0, j - tamaño + 1) hasta min(j,
                    BOARD_SIZE - tamaño)
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño
                    + 1)):
                        # Verificar que todas las celdas desde inicio hasta inicio+tamaño
                        estén libres
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio,
                        inicio + tamaño)):
                            mapa[i, j] += 1

                    # --- Evaluación en orientación vertical ---
                    # Se busca un inicio válido para que el barco incluya la celda (i,j)
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño
                    + 1)):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio,
                        inicio + tamaño)):
                            mapa[i, j] += 1

    return mapa

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia de probabilidad y caza y eliminación."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero)

    # Si no hay objetivo activo y la memoria está vacía, usa el mapa de probabilidades
    if objetivo is None and not memoria:
        # Calcular el mapa de probabilidades
        mapa = calcular_mapa_probabilidades(tablero, barcos_vivos)

        # Imprimir el mapa de probabilidades para depuración
        print("Mapa de Probabilidades:")
        print(mapa)

        # Buscar el valor máximo de probabilidad
        max_prob = np.max(mapa)
        # Obtener todas las celdas (índices) que tienen ese valor máximo
        indices = np.argwhere(mapa == max_prob)
        # Elegir aleatoriamente una de las celdas de mayor probabilidad
        fila, columna = random.choice(indices)
    elif memoria:
        fila, columna = memoria.pop(0)
    else:
        # En caso de eventualidad, se elige aleatoriamente
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    # Realizar el disparo
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        # Guardar la letra del barco antes de marcar el impacto
        letra_barco = tablero[fila, columna]
        tablero[fila, columna] = "X" # Impacto

```

```
# Comprobar si el barco ha sido hundido
if not np.any(tablero == letra_barco):
    barcos_vivos = barcos_restantes(tablero)
    objetivo = None
    memoria.clear()
else:
    # Si es el primer impacto o se continúa la caza, activar la estrategia de caza
    if objetivo is None:
        objetivo = (fila, columna)
        # Explorar en las 4 direcciones (horizontal y vertical)
        for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nf, nc = fila + df, columna + dc
            while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                if tablero[nf, nc] == "~":
                    break # Se detiene si se encuentra agua
                if tablero[nf, nc] not in ["X", "O"]:
                    memoria.append((nf, nc))
                    nf += df
                    nc += dc
    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0

        # En cada ronda ambas máquinas disparan y luego se evalúa el estado
        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            barcos_vivos, objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                break
            elif not m2:
                victorias_maquina_1 += 1
                break
            elif not m1:
```

```

        victorias_maquina_2 += 1
        break
    turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Estrategia probabilística vs Humano")
plt.show()

```

ANEXO XIII: CÓDIGO DE MEMORIA ENTRE PARTIDAS

A13.1 Memoria entre partidas vs Aleatorio

```
import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Variables globales para la memoria entre partidas
memoria_global = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)
partidas_memorizadas = 0

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):
    print(" " + " ".join(COLUMN_LETTERS))
    for i, fila in enumerate(tablero):
        if ocultar_barcos:
            print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
        else:
            print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
        return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for fila in tablero for celda in fila if celda not in ["~",
"X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

def actualizar_memoria(tablero, alpha=1):
    """
```

```

    Actualiza la memoria global sumando, para cada celda,
    1 (multiplicado por alpha) si en el tablero hay un barco (es decir, si no es "~").
    Se mantiene un acumulado de las últimas 50 partidas.
    """
    global memoria_global, partidas_memorizadas
    # Convertir a 1 donde hay un barco y 0 donde no
    barcos = (tablero != "~").astype(int)
    memoria_global += alpha * barcos
    partidas_memorizadas += 1
    # Si se han memorizado más de 50 partidas, escalamos para mantener el promedio de las
    últimas 50
    if partidas_memorizadas > 50:
        memoria_global = (memoria_global * 50) // partidas_memorizadas
        partidas_memorizadas = 50

def calcular_mapa_probabilidades(tablero, barcos_vivos, beta=1):
    """
    Calcula un mapa de probabilidades (matriz de enteros) en el que cada celda acumula
    1 por cada forma válida en que algún barco restante podría ubicarse allí.
    Además, suma un extra derivado de la memoria (ponderado por beta).
    Se consideran solo posiciones en las que la celda no haya sido disparada ("X" u "O").
    """
    # Crear una matriz de ceros con las dimensiones del tablero
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

    # Cálculo de probabilidades basado en el estado actual del tablero
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tablero[i, j] not in ["X", "O"]:
                for barco, tamaño in barcos_vivos.items():
                    # Orientación horizontal
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño)
+ 1):
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                            mapa[i, j] += 1
                    # Orientación vertical
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño)
+ 1):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                            mapa[i, j] += 1

    # Incluir la memoria en el mapa: se suma beta * (promedio de la memoria)
    if partidas_memorizadas > 0:
        promedio_memoria = memoria_global / partidas_memorizadas
        mapa = mapa + beta * promedio_memoria

    return mapa

def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia de probabilidad y caza y eliminación."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero)

    # Si no hay objetivo activo y la memoria está vacía, usa el mapa de probabilidades
    if objetivo is None and not memoria:
        # Calcular el mapa de probabilidades
        mapa = calcular_mapa_probabilidades(tablero, barcos_vivos)

        # Imprimir el mapa de probabilidades para depuración
        print("Mapa de Probabilidades:")
        print(mapa)

        # Buscar el valor máximo de probabilidad
        max_prob = np.max(mapa)
        # Obtener todas las celdas (índices) que tienen ese valor máximo
        indices = np.argwhere(mapa == max_prob)
        # Elegir aleatoriamente una de las celdas de mayor probabilidad

```

```
        fila, columna = random.choice(indices)
    elif memoria:
        fila, columna = memoria.pop(0)
    else:
        # En caso de eventualidad, se elige aleatoriamente
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    # Realizar el disparo
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        # Guardar la letra del barco antes de marcar el impacto
        letra_barco = tablero[fila, columna]
        tablero[fila, columna] = "X" # Impacto

        # Comprobar si el barco ha sido hundido
        if not np.any(tablero == letra_barco):
            barcos_vivos = barcos_restantes(tablero)
            objetivo = None
            memoria.clear()
        else:
            # Si es el primer impacto o se continúa la caza, activar la estrategia de caza
            if objetivo is None:
                objetivo = (fila, columna)
                # Explorar en las 4 direcciones (horizontal y vertical)
                for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    nf, nc = fila + df, columna + dc
                    while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                        if tablero[nf, nc] == "~":
                            break # Se detiene si se encuentra agua
                        if tablero[nf, nc] not in ["X", "O"]:
                            memoria.append((nf, nc))
                        nf += df
                        nc += dc

    return tablero

def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)
        if tablero[fila, columna] not in ["X", "O"]:
            if tablero[fila, columna] == "~":
                tablero[fila, columna] = "O" # Agua
            else:
                tablero[fila, columna] = "X" # Impacto
            break
    return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos(crear_tablero())
        # Guarda una copia del tablero inicial de la Máquina 2 para la memoria
        tablero_inicial_maquina_2 = np.copy(tablero_maquina_2)
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0
```

```

while True:
    turnos += 1
    tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
    barcos_vivos, objetivo_maquina_1)
    tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

    m1 = quedan_barcos(tablero_maquina_1)
    m2 = quedan_barcos(tablero_maquina_2)

    if not m1 and not m2:
        empates += 1
        actualizar_memoria(tablero_inicial_maquina_2)
        break
    elif not m2:
        victorias_maquina_1 += 1
        actualizar_memoria(tablero_inicial_maquina_2)
        break
    elif not m1:
        victorias_maquina_2 += 1
        actualizar_memoria(tablero_inicial_maquina_2)
        break
    turnos_por_partida.append(turnos)

print(f"Resultados tras {n} partidas:")
print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
print(f"Empates: {empates}")
return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Memoria entre partidas vs Aleatorio")
plt.show()

```

A13.2 Memoria entre partidas vs Humano

```

import numpy as np
import random
from IPython.display import clear_output # Para limpiar la pantalla en Google Colab
import string
import matplotlib.pyplot as plt

# Definición del tablero de 10x10
BOARD_SIZE = 10
COLUMN_LETTERS = list(string.ascii_uppercase[:BOARD_SIZE])
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

# Variables globales para la memoria entre partidas
memoria_global = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)
partidas_memorizadas = 0

# Función para inicializar el tablero
def crear_tablero():
    return np.full((BOARD_SIZE, BOARD_SIZE), "~") # '~' representa agua

# Función para imprimir el tablero
def imprimir_tablero(tablero, ocultar_barcos=True):

```

```
print(" " + " ".join(COLUMN_LETTERS))
for i, fila in enumerate(tablero):
    if ocultar_barcos:
        print(str(i) + " " + " ".join(cell if cell in ["X", "O", "~"] else "~" for cell
in fila))
    else:
        print(str(i) + " " + " ".join(fila))

def colocar_barcos(tablero):
    for barco, tamaño in BARCOS.items():
        colocado = False
        while not colocado:
            fila = random.randint(0, BOARD_SIZE - 1)
            columna = random.randint(0, BOARD_SIZE - 1)
            orientacion = random.choice(["H", "V"])

            if orientacion == "H" and columna + tamaño <= BOARD_SIZE and all(tablero[fila,
columna + i] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
                    colocado = True
            elif orientacion == "V" and fila + tamaño <= BOARD_SIZE and all(tablero[fila +
i, columna] == "~" for i in range(tamaño)):
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
                    colocado = True
        return tablero

def colocar_barcos_preplaneados(tablero):
    """
    Coloca los barcos usando una de 10 disposiciones predefinidas.
    Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna, orientacion)
    Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se solapen.
    """
    disposiciones = [
        # Disposición 1: Todos horizontales en la parte izquierda
        [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0, "H"),
("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
        # Disposición 2: Todos verticales en la esquina superior izquierda
        [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4, "V"),
("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
        # Disposición 3: Todos horizontales en la parte inferior
        [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0, "H"),
("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
        # Disposición 4: Mezclada: bordes opuestos
        [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0, "H"),
("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
        # Disposición 5: Todos horizontales en filas centrales
        [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4, "H"),
("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
        # Disposición 6: Todos verticales en columnas centrales
        [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4, "V"),
("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
        # Disposición 7: Mezclada, horizontales y verticales
        [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9, "V"),
("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
        # Disposición 8: Todos horizontales en un bloque central
        [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2, "H"),
("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
        # Disposición 9: Todos verticales en un bloque central
        [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6, "V"),
("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
        # Disposición 10: Mezcla de orientaciones
        [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5, "H"),
("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
    ]
    disp = random.choice(disposiciones)
    # Para cada barco en la disposición, se comprueba y se coloca:
    for barco, fila, columna, orientacion in disp:
        tamaño = BARCOS[barco]
```

```

    if orientacion == "H":
        # Comprobar límite horizontal
        if columna + tamaño > BOARD_SIZE:
            # Si no cabe, se omite (idealmente, la disposición ya debería ser válida)
            continue
        # Comprobar que no se solape con otro barco
        if any(tablero[filas, columna + i] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[filas, columna + i] = barco[0]
    else: # "V"
        if filas + tamaño > BOARD_SIZE:
            continue
        if any(tablero[filas + i, columna] != "~" for i in range(tamaño)):
            continue
        for i in range(tamaño):
            tablero[filas + i, columna] = barco[0]
    return tablero

def barcos_restantes(tablero):
    """Calcula qué barcos siguen en juego basándose en los impactos realizados."""
    barcos_vivos = set(BARCOS.keys())
    barcos_en_tablero = {celda for filas in tablero for celda in filas if celda not in ["X", "O"]}
    barcos_vivos = {b for b in BARCOS.keys() if b[0] in barcos_en_tablero}
    return {b: BARCOS[b] for b in barcos_vivos}

def actualizar_memoria(tablero, alpha=1):
    """
    Actualiza la memoria global sumando, para cada celda,
    1 (multiplicado por alpha) si en el tablero hay un barco (es decir, si no es "~").
    Se mantiene un acumulado de las últimas 50 partidas.
    """
    global memoria_global, partidas_memorizadas
    # Convertir a 1 donde hay un barco y 0 donde no
    barcos = (tablero != "~").astype(int)
    memoria_global += alpha * barcos
    partidas_memorizadas += 1
    # Si se han memorizado más de 50 partidas, escalamos para mantener el promedio de las
    últimas 50
    if partidas_memorizadas > 50:
        memoria_global = (memoria_global * 50) // partidas_memorizadas
        partidas_memorizadas = 50

def calcular_mapa_probabilidades(tablero, barcos_vivos, beta=1):
    """
    Calcula un mapa de probabilidades (matriz de enteros) en el que cada celda acumula
    1 por cada forma válida en que algún barco restante podría ubicarse allí.
    Además, suma un extra derivado de la memoria (ponderado por beta).
    Se consideran solo posiciones en las que la celda no haya sido disparada ("X" u "O").
    """
    # Crear una matriz de ceros con las dimensiones del tablero
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

    # Cálculo de probabilidades basado en el estado actual del tablero
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tablero[i, j] not in ["X", "O"]:
                for barco, tamaño in barcos_vivos.items():
                    # Orientación horizontal
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño + 1)):
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1
                    # Orientación vertical
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño + 1)):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio, inicio + tamaño)):
                            mapa[i, j] += 1

```

```
        mapa[i, j] += 1

# Incluir la memoria en el mapa: se suma beta * (promedio de la memoria)
if partidas_memorizadas > 0:
    promedio_memoria = memoria_global / partidas_memorizadas
    mapa = mapa + beta * promedio_memoria

return mapa


def disparo_maquina_1(tablero, memoria, barcos_vivos, objetivo=None):
    """Máquina 1 con estrategia de probabilidad y caza y eliminación."""
    if not barcos_vivos:
        barcos_vivos = barcos_restantes(tablero)

    # Si no hay objetivo activo y la memoria está vacía, usa el mapa de probabilidades
    if objetivo is None and not memoria:
        # Calcular el mapa de probabilidades
        mapa = calcular_mapa_probabilidades(tablero, barcos_vivos)

        # Imprimir el mapa de probabilidades para depuración
        print("Mapa de Probabilidades:")
        print(mapa)

        # Buscar el valor máximo de probabilidad
        max_prob = np.max(mapa)
        # Obtener todas las celdas (índices) que tienen ese valor máximo
        indices = np.argwhere(mapa == max_prob)
        # Elegir aleatoriamente una de las celdas de mayor probabilidad
        fila, columna = random.choice(indices)
    elif memoria:
        fila, columna = memoria.pop(0)
    else:
        # En caso de eventualidad, se elige aleatoriamente
        fila = random.randint(0, BOARD_SIZE - 1)
        columna = random.randint(0, BOARD_SIZE - 1)

    # Realizar el disparo
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        # Guardar la letra del barco antes de marcar el impacto
        letra_barco = tablero[fila, columna]
        tablero[fila, columna] = "X" # Impacto

        # Comprobar si el barco ha sido hundido
        if not np.any(tablero == letra_barco):
            barcos_vivos = barcos_restantes(tablero)
            objetivo = None
            memoria.clear()
        else:
            # Si es el primer impacto o se continúa la caza, activar la estrategia de caza
            if objetivo is None:
                objetivo = (fila, columna)
                # Explorar en las 4 direcciones (horizontal y vertical)
                for df, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    nf, nc = fila + df, columna + dc
                    while 0 <= nf < BOARD_SIZE and 0 <= nc < BOARD_SIZE:
                        if tablero[nf, nc] == "~":
                            break # Se detiene si se encuentra agua
                        if tablero[nf, nc] not in ["X", "O"]:
                            memoria.append((nf, nc))
                        nf += df
                        nc += dc

    return tablero


def disparo_maquina_2(tablero):
    """Máquina 2 sigue siendo completamente aleatoria."""
    while True:
        fila = random.randint(0, BOARD_SIZE - 1)
```

```

columna = random.randint(0, BOARD_SIZE - 1)
if tablero[fila, columna] not in ["X", "O"]:
    if tablero[fila, columna] == "~":
        tablero[fila, columna] = "O" # Agua
    else:
        tablero[fila, columna] = "X" # Impacto
    break
return tablero

def quedan_barcos(tablero):
    return any(cell not in ["~", "X", "O"] for row in tablero for cell in row)

def simular_partidas(n=1000):
    global turnos_por_partida
    turnos_por_partida = [] # Reiniciamos la lista cada vez que se llama a la simulación
    victorias_maquina_1 = 0
    victorias_maquina_2 = 0
    empates = 0

    for i in range(n):
        tablero_maquina_1 = colocar_barcos(crear_tablero())
        tablero_maquina_2 = colocar_barcos_preplaneados(crear_tablero())
        # Guarda una copia del tablero inicial de la Máquina 2 para la memoria
        tablero_inicial_maquina_2 = np.copy(tablero_maquina_2)
        memoria_maquina_1 = []
        barcos_vivos = barcos_restantes(tablero_maquina_2)
        objetivo_maquina_1 = None
        turnos = 0

        while True:
            turnos += 1
            tablero_maquina_2 = disparo_maquina_1(tablero_maquina_2, memoria_maquina_1,
            barcos_vivos, objetivo_maquina_1)
            tablero_maquina_1 = disparo_maquina_2(tablero_maquina_1)

            m1 = quedan_barcos(tablero_maquina_1)
            m2 = quedan_barcos(tablero_maquina_2)

            if not m1 and not m2:
                empates += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break
            elif not m2:
                victorias_maquina_1 += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break
            elif not m1:
                victorias_maquina_2 += 1
                actualizar_memoria(tablero_inicial_maquina_2)
                break

        turnos_por_partida.append(turnos)

    print(f"Resultados tras {n} partidas:")
    print(f"Máquina 1 (aleatoria) ganó {victorias_maquina_1} veces.")
    print(f"Máquina 2 (aleatoria) ganó {victorias_maquina_2} veces.")
    print(f"Empates: {empates}")
    return turnos_por_partida

# Ejecutar la simulación y obtener los turnos por partida
turnos_por_partida = simular_partidas(n=1000)

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.hist(turnos_por_partida, bins=30, edgecolor='black', alpha=0.7)
plt.xlabel("Número de turnos")
plt.ylabel("Frecuencia")
plt.title("Memoria entre partidas vs Humano")
plt.show()

```

ANEXO XIV: CÓDIGO DE RL

A14.1 Código de entrenamiento de RL

```
import numpy as np
import random
import gym
from gym import spaces
from collections import deque
from tensorflow.keras.models import Sequential, clone_model
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
from tqdm.notebook import tqdm
from IPython.display import clear_output

import tensorflow as tf
tf.config.run_functions_eagerly(True)

# Definición global de los barcos y otros parámetros del tablero
BOARD_SIZE = 10
BARCOS = {
    "Portaviones": 5,
    "Acorazado": 4,
    "Submarino": 3,
    "Destructor": 3,
    "Lancha": 2
}

def calcular_mapa_probabilidades(tablero, barcos_vivos):
    """
    Calcula un mapa de probabilidades en el que cada celda acumula 1 por cada forma
    válida en que algún barco restante podría ubicarse allí.
    Se consideran solo posiciones donde la celda no ha sido disparada ("X" u "O").
    """
    # Se asume que BOARD_SIZE está definido y coincide con el tamaño del tablero
    mapa = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if tablero[i, j] not in ["X", "O"]:
                for barco, tamaño in barcos_vivos.items():
                    # Evaluación horizontal
                    for inicio in range(max(0, j - tamaño + 1), min(j, BOARD_SIZE - tamaño
+ 1)):
                        if all(tablero[i, k] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                            mapa[i, j] += 1
                    # Evaluación vertical
                    for inicio in range(max(0, i - tamaño + 1), min(i, BOARD_SIZE - tamaño
+ 1)):
                        if all(tablero[k, j] not in ["X", "O"] for k in range(inicio,
inicio + tamaño)):
                            mapa[i, j] += 1
    return mapa

class AmbienteHundirLaFlota(gym.Env):
    """
    Ambiente para el juego de Hundir la Flota con Aprendizaje por Refuerzo.
    Observación:
        Un tablero de 10x10 donde:
        0 -> Celda sin disparar (desconocida)
        1 -> Disparo exitoso (impacto)
        -1 -> Disparo fallido (agua)
    Acción:
        Un entero de 0 a 99 que representa la celda (fila, columna) en la que se dispara.
    Recompensas:
        +1 por impacto.
        +5 por hundir un barco.
        -1 si se dispara en una celda ya seleccionada.
    """
```

```

-0.1 por disparo fallido.
"""
def __init__(self, tamaño_tablero=10):
    super(AmbienteHundirLaFlota, self).__init__()
    self.tamaño_tablero = tamaño_tablero
    self.espacio_acciones = spaces.Discrete(tamaño_tablero * tamaño_tablero)
    self.espacio_observacion = spaces.Box(low=-1, high=1,
                                           shape=(tamaño_tablero, tamaño_tablero),
dtype=np.int8)
    self.barcos = BARCOS # Usamos la variable global
    self.reset()

    def reset(self):
        self.tablero_oculto = np.full((self.tamaño_tablero, self.tamaño_tablero), "~")
        self.tablero_visible = np.zeros((self.tamaño_tablero, self.tamaño_tablero),
dtype=np.int8)
        self.colocar_barcos_preplaneados(self.tablero_oculto)
        self.terminado = False
        self.turnos = 0
        return self.tablero_visible.copy()

    def colocar_barcos_preplaneados(self, tablero):
        """
        Coloca los barcos usando una de 10 disposiciones predefinidas.
        Cada disposición es una lista de tuplas: (nombre_del_barco, fila, columna,
orientacion)
        Se asume que el tablero está vacío ("~") y se comprueba que los barcos no se
solapen.
        """
        disposiciones = [
            [("Portaviones", 0, 0, "H"), ("Acorazado", 2, 0, "H"), ("Submarino", 4, 0,
"H"), ("Destructor", 6, 0, "H"), ("Lancha", 8, 0, "H")],
            [("Portaviones", 0, 0, "V"), ("Acorazado", 0, 2, "V"), ("Submarino", 0, 4,
"V"), ("Destructor", 0, 6, "V"), ("Lancha", 0, 8, "V")],
            [("Portaviones", 9, 0, "H"), ("Acorazado", 7, 0, "H"), ("Submarino", 5, 0,
"H"), ("Destructor", 3, 0, "H"), ("Lancha", 1, 0, "H")],
            [("Portaviones", 0, 9, "V"), ("Acorazado", 9, 5, "H"), ("Submarino", 9, 0,
"H"), ("Destructor", 5, 5, "H"), ("Lancha", 0, 0, "H")],
            [("Portaviones", 0, 4, "H"), ("Acorazado", 2, 4, "H"), ("Submarino", 4, 4,
"H"), ("Destructor", 6, 4, "H"), ("Lancha", 8, 4, "H")],
            [("Portaviones", 4, 0, "V"), ("Acorazado", 4, 2, "V"), ("Submarino", 4, 4,
"V"), ("Destructor", 4, 6, "V"), ("Lancha", 4, 8, "V")],
            [("Portaviones", 0, 0, "H"), ("Acorazado", 9, 0, "H"), ("Submarino", 0, 9,
"V"), ("Destructor", 7, 9, "V"), ("Lancha", 5, 5, "H")],
            [("Portaviones", 2, 2, "H"), ("Acorazado", 4, 2, "H"), ("Submarino", 6, 2,
"H"), ("Destructor", 8, 2, "H"), ("Lancha", 0, 2, "H")],
            [("Portaviones", 2, 2, "V"), ("Acorazado", 2, 4, "V"), ("Submarino", 2, 6,
"V"), ("Destructor", 2, 8, "V"), ("Lancha", 2, 0, "V")],
            [("Portaviones", 9, 0, "H"), ("Acorazado", 0, 5, "H"), ("Submarino", 5, 5,
"H"), ("Destructor", 0, 0, "V"), ("Lancha", 7, 7, "H")]
        ]
        disp = random.choice(disposiciones)
        for barco, fila, columna, orientacion in disp:
            tamaño = BARCOS[barco]
            if orientacion == "H":
                if columna + tamaño > BOARD_SIZE:
                    continue
                if any(tablero[fila, columna + i] != "~" for i in range(tamaño)):
                    continue
                for i in range(tamaño):
                    tablero[fila, columna + i] = barco[0]
            else:
                if fila + tamaño > BOARD_SIZE:
                    continue
                if any(tablero[fila + i, columna] != "~" for i in range(tamaño)):
                    continue
                for i in range(tamaño):
                    tablero[fila + i, columna] = barco[0]
        return tablero

```

```
def step(self, accion):
    fila = accion // self.tamano_tablero
    columna = accion % self.tamano_tablero
    recompensa = 0
    info = {}
    self.turnos += 1

    # Penalización: si se dispara en una celda ya disparada, resta -1 (como
    originalmente)
    if self.tablero_visible[fila, columna] != 0:
        recompensa = -1
        return self.tablero_visible.copy(), recompensa, self.terminado, info

    if self.tablero_oculto[fila, columna] != "~":
        self.tablero_visible[fila, columna] = 1 # Impacto exitoso
        letra_barco = self.tablero_oculto[fila, columna]
        self.tablero_oculto[fila, columna] = "X"
        recompensa = 1
        if not np.any(self.tablero_oculto == letra_barco):
            recompensa = 5
    else:
        self.tablero_visible[fila, columna] = -1 # Disparo fallido (agua)
        recompensa = -0.1

    if not any(celda not in ["X", "~"] for celda in self.tablero_oculto.flatten()):
        self.terminado = True

    return self.tablero_visible.copy(), recompensa, self.terminado, info

def render(self, modo='human'):
    print("Tablero Visible:")
    for i, fila in enumerate(self.tablero_visible):
        print(f"{i} {' '.join(fila.astype(str))}")

class AgenteDQN:
    def __init__(self, tamaño_tablero, tasa_aprendizaje=0.001, gamma=0.95, epsilon=1.0,
                  epsilon_min=0.01, epsilon_decay=0.995, memoria_max=2000):
        self.tamaño_tablero = tamaño_tablero
        self.tasa_aprendizaje = tasa_aprendizaje # Velocidad de aprendizaje
        self.gamma = gamma # Factor de descuento
        self.epsilon = epsilon # Tasa de exploración inicial
        self.epsilon_min = epsilon_min # Valor mínimo de epsilon
        self.epsilon_decay = epsilon_decay # Factor de decrecimiento de epsilon
        self.memoria = deque(maxlen=memoria_max) # Memoria para experience replay

        # Construir el modelo principal y el modelo objetivo
        self.modelo = self._construir_modelo()
        self.modelo_objetivo = clone_model(self.modelo)
        self.modelo_objetivo.set_weights(self.modelo.get_weights())

    def _construir_modelo(self):
        modelo = Sequential()
        modelo.add(Flatten(input_shape=(self.tamaño_tablero, self.tamaño_tablero)))
        modelo.add(Dense(64, activation='relu'))
        modelo.add(Dense(64, activation='relu'))
        modelo.add(Dense(self.tamaño_tablero * self.tamaño_tablero, activation='linear'))
        modelo.compile(loss='mse', optimizer=Adam(learning_rate=self.tasa_aprendizaje))
        return modelo

    def recordar(self, estado, accion, recompensa, siguiente_estado, terminado):
        self.memoria.append((estado, accion, recompensa, siguiente_estado, terminado))

    def elegir_accion(self, estado):
        # Obtén los índices de las casillas válidas (no disparadas)
        validos = np.where(estado.flatten() == 0)[0]
        if len(validos) == 0:
            return random.randrange(self.tamaño_tablero * self.tamaño_tablero)

        # Modo exploratorio: usar mapa de probabilidades en lugar de elección uniforme
        if np.random.rand() <= self.epsilon:
```

```

        # Se asume que la función calcular_mapa_probabilidades está definida
        globalmente y usa BOARD_SIZE
        prob_map = calcular_mapa_probabilidades(estado, BARCOS)
        flat_map = prob_map.flatten().astype(float)
        probs_valid = np.zeros_like(flat_map)
        probs_valid[validos] = flat_map[validos]
        if probs_valid.sum() > 0:
            probs_valid = probs_valid / probs_valid.sum()
            accion = np.random.choice(len(flat_map), p=probs_valid)
        else:
            accion = random.choice(validos)
        return accion
    else:
        # Modo explotación: usar la red para predecir el Q-valor
        estado_reshaped = np.expand_dims(estado, axis=0)
        q_valores = self.modelo.predict(estado_reshaped, verbose=0)[0]
        q_valores_filtrados = np.full(q_valores.shape, -np.inf)
        q_valores_filtrados[validos] = q_valores[validos]
        return np.argmax(q_valores_filtrados)

def entrenar(self, tamaño_batch=32):
    if len(self.memoria) < tamaño_batch:
        return # No hay suficientes muestras para entrenar

    batch = random.sample(self.memoria, tamaño_batch)
    estados = np.array([trans[0] for trans in batch])
    acciones = np.array([trans[1] for trans in batch])
    recompensas = np.array([trans[2] for trans in batch])
    siguientes_estados = np.array([trans[3] for trans in batch])
    terminados = np.array([trans[4] for trans in batch])

    objetivos = self.modelo.predict(estados, verbose=0)
    q_siguientes = self.modelo_objetivo.predict(siguientes_estados, verbose=0)

    for i in range(tamaño_batch):
        if terminados[i]:
            objetivos[i][acciones[i]] = recompensas[i]
        else:
            objetivos[i][acciones[i]] = recompensas[i] + self.gamma *
np.amax(q_siguientes[i])
        self.modelo.fit(estados, objetivos, epochs=1, verbose=0)
        # Se ha eliminado la actualización de ε aquí para hacerla una vez por episodio

def actualizar_modelo_objetivo(self):
    self.modelo_objetivo.set_weights(self.modelo.get_weights())

# Crear instancia del entorno y del agente
ambiente = AmbienteHundirLaFlota(tamaño_tablero=10)
agente = AgenteDQN(tamaño_tablero=10, epsilon=1.0, epsilon_decay=0.99908)

# Intentar cargar el modelo guardado (si existe)
import os
if os.path.exists("modelo_dqn.h5"):
    from tensorflow.keras.models import load_model
    import tensorflow as tf
    custom_objects = {'mse': lambda y_true, y_pred: tf.reduce_mean(tf.square(y_true -
y_pred))}
    agente.modelo = load_model("modelo_dqn.h5", custom_objects=custom_objects)
    # Recompilar el modelo con un nuevo optimizador:
    agente.modelo.compile(loss='mse',
optimizer=Adam(learning_rate=agente.tasa_aprendizaje))

    # Clonar y recompilar el modelo objetivo:
    agente.modelo_objetivo = clone_model(agente.modelo)
    agente.modelo_objetivo.compile(loss='mse',
optimizer=Adam(learning_rate=agente.tasa_aprendizaje))
    agente.modelo_objetivo.set_weights(agente.modelo.get_weights())

```

```
print("Modelo cargado y recompilado desde modelo_dqn.h5")

episodios = 5000          # Número de episodios (partidas) para entrenar
frecuencia_actualizacion = 10 # Frecuencia para actualizar el modelo objetivo
max_turnos = 100         # Límite máximo de turnos por episodio

barra_progreso = tqdm(range(episodios), desc="Entrenando episodios")
for ep in barra_progreso:
    estado = ambiente.reset()
    terminado = False
    recompensa_total = 0
    turnos = 0

    while not terminado and turnos < max_turnos:
        accion = agente.elegir_accion(estado)
        siguiente_estado, recompensa, terminado, info = ambiente.step(accion)
        agente.recordar(estado, accion, recompensa, siguiente_estado, terminado)
        agente.entrenar(tamano_batch=32)
        estado = siguiente_estado
        recompensa_total += recompensa
        turnos += 1

    # Si se alcanza el límite de turnos, se fuerza la terminación del episodio
    if turnos >= max_turnos:
        terminado = True

    # Actualizar el modelo objetivo cada ciertos episodios
    if ep % frecuencia_actualizacion == 0:
        agente.actualizar_modelo_objetivo()

    # Guardar el modelo cada 50 episodios
    if ep % 50 == 0:
        agente.modelo.save("modelo_dqn.h5")
        print("Modelo guardado en modelo_dqn.h5")

    # **Actualizar ε una vez por episodio:**
    if agente.epsilon > agente.epsilon_min:
        agente.epsilon *= agente.epsilon_decay

    barra_progreso.set_description(f"Epi {ep+1}/{episodios} - Recompensa:
{recompensa_total:.2f} - Epsilon: {agente.epsilon:.3f}")
```