Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Deep Learning en Videojuegos: Resolución de Laberintos

Deep Learning in Videogames: Labyrinth Resolution

Pablo Martín González

La Laguna, 11 de septiembre de 2020

D. **Jose Demetrio Piñeiro Vera**, con N.I.F. 43.774.048-B profesor Titular de Universidad adscrito al Departamento de Ingeniera Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Jesús Miguel Torres Jorge**, con N.I.F.43.826.207-Y profesor Titular de Universidad adscrito al Departamento de Ingeniera Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Deep Learning en Videojuegos: Resolución de Laberintos"

ha sido realizada bajo su dirección por D. **Pablo Martín González**,

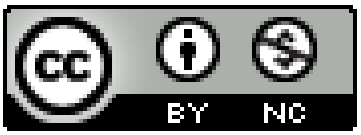
con N.I.F. 79.081.496-K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de septiembre de 2020

Agradecimientos

A mis tutores, por ayudarme a la hora de realizar este trabajo, sin los cuales, este trabajo no se hubiera podido llevar a cabo.

Licencia



Resumen

Los algoritmos de búsqueda de camino son indispensables en la sociedad de hoy en día. Dichos algoritmos son usados por aplicaciones como Google Maps, sistemas GPS, o incluso, en el enrutamiento de paquetes a través de internet. El objetivo de este trabajo es la de estudiar la posibilidad de crear mediante Deep Learning un modelo capaz de rivalizar contra los algoritmos actuales de "pathfinding". Para ello, se hace uso del entorno de desarrollo Unity y del kit de herramientas de machine learning ML-Agents, publicado por el equipo de desarrollo de Unity.

Palabras clave: Laberinto, Deep Learning, Ml-Agents, Búsqueda de caminos

Abstract

Pathfinding algorithms are indispensable today, these algorithms are used by applications such as Google Maps, GPS systems, or even, in the routing of packets through the internet. The objective of this work is to study the possibility of creating through Deep Learning a model capable of competing against current "pathfinding" algorithms. To do this, the Unity development environment, and the ML-Agents machine learning toolkit, published by the Unity development team, are used.

Keywords: Labyrinth, Deep Learning, Ml-Agents, Pathfinding

Índice general

Capítulo 1 Introducción 11
1.1 Motivación y objetivos 11
Capítulo 2 Antecedentes y estado actual del arte 12
2.1 Videojuegos 12
2.1.1 Inteligencia Artificial en Videojuegos 12
2.2 Laberintos 13
2.2.1 Resolución de laberintos 13
2.2.2 Generación de laberintos 14
2.3 Aprendizaje por refuerzo 14
2.3.1 Casos famosos 15
2.3.2 Algoritmos 16
Capítulo 3 Desarrollo 17
3.1 Entorno de desarrollo y herramientas utilizadas 17
3.2 Creación de escenarios 17
3.3 Creación del Agente 18
3.4 Búsqueda de la configuración óptima 20
3.5 Cambio de algoritmo 22
Capítulo 4 Resultados 25
4.1 Comportamiento de los agentes 25
4.2 Problemática del estudio final 26
4.3 Procedimiento del estudio 26
4.4 Resultados modelo LSTM 3.07E6 28
4.5 Resultados LSTM 1.224E7 30
4.6 Laberintos no perfectos 33
4.7 Resultados Laberinto Wilson 34
Capítulo 5 Conclusiones y líneas futuras 35
5.1 Conclusiones 35
5.2 Líneas futuras 35
Capítulo 6 Summary and Conclusions 36

6.1 Final conclusions 36
6.2 Future Works 36
Capítulo 7 Presupuesto 37
7.1 Coste del proyecto 37
Capítulo 8 Repositorio del proyecto 39

Índice de figuras

Figura 1: Encuadre típico de un escenario en Aprendizaje por Refuerzo 15
Figura 2: Rejilla de Laberinto 17
Figura 3: Laberinto completo 18
Figura 4: Corrección de movimiento 18
Figura 5: Normalización de las recompensas 19
Figura 6: Vector de observaciones 19
Figura 7: Ejemplo de Curriculum 21
Figura 8: Ejemplo de actualización de Curriculum 21
Figura 9: Modelo intentando ir hacia el final 25
Figura 10: Atrapado en un laberinto de 14X14 26
Figura 11: Histograma LSTM 3.07E6 29
Figura 12: Histograma LSTM 1.224E7 31

Índice de tablas

Tabla 1: Matriz de discretización de valores 20
Tabla 2: Resultados de tabla de rendimiento, media 21
Tabla 3: Resultados de tabla de rendimiento, desviación típica 22
Tabla 4: Comparación de modelos 22
Tabla 5: Estadísticas LSTM 3.07E6 5X5 23
Tabla 6: Estadísticas LSTM 3.07E6 9X9 23
Tabla 7: Estadísticas LSTM 1.224E7 9X9 23
Tabla 8: Ejemplo de estudio 27
Tabla 9: Estudio LSTM 3.07E6 simplificada 28
Tabla 10: problemática de la dificultad 29
Tabla 11: Estudio LSTM 1.224R7 9X9 Max 30
Tabla 12: Laberintos no resueltos. 32
Tabla 13: Coste Recursos Humanos 37
Tabla 14: Cálculo de horas de trabajo computacional 37
Tabla 15: Coste de electricidad 38
Tabla 16: Coste total del proyecto 38

Capítulo 1
Introducción

1.1 Motivación y objetivos

El algoritmo A* se publicó hace ya cuarenta años y se considera, junto con Dijkstra, uno de los mejores métodos para encontrar el camino mínimo entre dos puntos. Por esa razón, y por los avances que ha habido en el campo de la inteligencia artificial en videojuegos de la última década, hemos decidido ver si es posible que un sistema pueda aprenderlos o acercarse a su funcionamiento en un contexto concreto.

El objetivo principal es el de encontrar un modelo que sea capaz de, igualar el algoritmo A y hacer un estudio de ver en qué condiciones es capaz de replicar su comportamiento.

Este objetivo principal se puede dividir en los siguientes objetivos específicos que se tienen que cumplir a la hora de finalizar el trabajo:

1. Desarrollar un entorno capaz de producir escenarios aleatorios.
2. Estudiar distintos modelos de redes neuronales.
3. Implementar estrategias de entrenamiento para los escenarios.
4. Hacer un estudio estadístico de los resultados obtenidos.
5. Mejorar el modelo hasta que sea capaz de resolver laberintos de manera competente.

Para ello, hacemos uso del entorno de desarrollo Unity, junto con su kit de herramientas de *machine learning*, ML-Agents, en el cual desarrollamos un agente, lo dotamos de la capacidad de captar su entorno y de actuar en consecuencia. Además, se realizó un estudio previo y a partir de ello fuimos cambiando y mejorando los ajustes del modelo.

Capítulo 2
Antecedentes y estado actual del arte

2.1 Videojuegos

Esta industria ha estado en continuo desarrollo durante las últimas décadas, los expertos [1] datan el primer videojuego en el año 1951, y desde entonces hasta el día de hoy, no ha parado de crecer de manera exponencial, tanto así, que ha llegado a ser una industria equiparable al cine o la música.

El principal culpable de esto es la aparición de internet y todas las facilidades que trae consigo, redes sociales, plataformas de "streamings", tiendas virtuales, etc. Gracias a ello es más fácil que nunca el tener interés por la industria de los videojuegos, sobre todo, cuando se organizan eventos que tienen repercusión a nivel mundial como la Gamescom o el E3, el cual tuvo en 2019 la conferencia más vista de todo ese año, la de Microsoft [2].

Además, hay que destacar el precio relativamente accesible de las consolas y de los videojuegos, que están al alcance de cualquier persona en mayor o menor medida. Esto es posible gracias a plataformas de internet que dejan los videojuegos a precios bajos, permitiendo así, que incluso gente sin grandes ingresos pueda disponer de ellos. Esto sumado a la evolución que ha tenido el "hardware" en las últimas décadas, ha dado como resultado que no se requiera de dispositivos altamente costosos para su consumo.

Por último, hay que destacar la facilidad de acceso a juegos competitivos, en los cuales las mecánicas utilizadas no son difíciles de aprender, pero sí de perfeccionar. Consiguiendo así captar tu atención durante más tiempo, mientras mejoras tu habilidad.

2.1.1 Inteligencia Artificial en Videojuegos

Desde el origen de los videojuegos se ha perseguido el intentar implementar algún tipo de inteligencia, ya sea en los NPC (Non-Playable Character), o en la generación procedural de niveles.

Al principio usaban solo mundos ya establecidos y enemigos con patrones de ataque sencillos, que se limitaban a hacer su ataque todo el rato, independientemente de si el jugador estaba cerca o no. Con los años, se consiguieron desarrollar algoritmos de búsqueda de caminos sencillos, para poder perseguir al jugador, mientras que en otros simplemente recorrer todo el espacio disponible de forma aleatoria y cuando encuentran el jugador de frente realizan una acción, ya sea perseguirlo, dispararle o huir de él. Para la generación procedural se usa algo similar y es que se tiene una lista de posibles salas que se van encadenando de manera aleatoria.

En la actualidad existen inteligencias capaces de simular un comportamiento bastante complejo. Aún está lejos de ser realista debido a que aún resulta un poco robótico, con falta de fluidez y de naturalidad.

Hoy en día existen juegos capaces de simular un comportamiento humano, creando estrategias dependiendo del entorno, un caso de esto es BattleField 1 [3], en el cual la IA cambia su objetivo atendiendo a su estado actual, si le falta munición va en busca de ella, si está bajo de vida intenta curarse. Sin embargo, el objetivo de ese proyecto no fue el de enfrentarse a humanos, su intención fue la de ayudar al equipo de desarrollo durante la fase de pruebas y de control de calidad, debido a que como uno no sabe cómo se van a comportar, pueden buscar la forma de romper el juego.

Esto ocurrió recientemente en el equipo de desarrollo de OpenAI, los cuales dieron vida a una

inteligencia capaz de jugar al "Pilla Pilla" [4]. En realidad, se tratan de dos IAs enfrentadas para aprender la una de la otra, durante su proceso de aprendizaje se ve como cuando los *hiders* desarrollan una estrategia, los *seekers* consiguen contrarrestarla. Hasta tal punto fue así, que, tras una estrategia ganadora por parte de los prófugos, los perseguidores consiguieron darles caza utilizando una mecánica que ni los programadores sabían que habían implementado. Para finalmente terminar con que los que huyen, consiguieron impedir que se les pillara. Esto demuestra que la inteligencia artificial es perfecta para el testeo de videojuegos, sin embargo, tiene un coste muy grande.

Hace casi diez años el diseñador de videojuegos Michael Cook desarrolló una IA capaz de diseñar videojuegos de manera automática. Su nombre es Angelina [5], y ha conseguido más de cuarenta juegos distintos. Para la creación de estos su creador le tiene que dar información, como imágenes o textos, y la inteligencia sola desarrollara el juego, busca en internet lo necesario para poder llevarlo a cabo, como modelos y diseños artísticos. También desarrolló una versión donde buscaba en internet temas de actualidad en periódicos y ese era su tema de entrada. Gracias a esa idea nació el juego "*Sex Lies and Rape*" [6], donde un juego de plataformas básico tenía una temática seria, ante ese título, el creador explicó que: "Es importante darse cuenta de que Angelina no puede saber cuan poderoso es algo así, en ese tipo de contexto. Por ahora".

2.2 Laberintos

Un laberinto es un lugar formado por calles y encrucijadas, intencionalmente complejo con la finalidad de dificultar el camino hacia la salida, dando la posibilidad de no encontrar el sendero correcto.

Los laberintos se pueden categorizar en dos tipos [7]: univiarios y multiviarios. Los univiarios son aquellos en los que es imposible perderse, debido a que su intención es la de hacer recorrer todo el espacio hasta llegar al final siguiendo una única vía. En el caso de los multiviarios, cuya finalidad es llegar a la salida, pero en este caso, puedes tomar caminos correctos o incorrectos.

Por otro lado, cada uno de estos dos grupos se dividen a su vez en subcategorías, dependiendo de la forma de construirlo cabe destacar:

- Laberinto Clásico: es de tipo univiario, de forma ovoidal y muy sencillo. El más famoso de este tipo es el Laberinto de Creta, creado para contener al minotauro.
- Laberinto Romano: cuadrado, univiario y dividido en cuatro cuadrantes normalmente simétricas.
- Laberinto Barroco: multiviario, con caminos muertos y un solo camino a la salida.

Un ejemplo de laberinto moderno son los mapas de las ciudades, los cuales serían de tipo multiviario y habría más de un camino posible para llegar de un punto A, hasta otro punto B, que serían nuestros puntos de inicio y fin respectivamente.

Un laberinto se puede decir que es perfecto cuando no tiene puntos muertos, es decir zonas donde no se puede llegar de ninguna manera, es equivalente a un grafo conexo y, por ende, se puede resolver con técnicas de resolución de grafos.

2.2.1 Resolución de laberintos

Desde siempre han existido algoritmos sencillos para resolver laberintos, uno de ellos es la regla de la mano derecha [8], que se basa, como bien indica su nombre, en mantener la mano derecha siempre tocando la primera pared que encontremos, una vez hecho eso, sólo basta con recorrer todo el laberinto sin levantar la mano. A fuerza bruta acabaremos encontrando la salida, esto se basa en la cualidad de que, si quitamos las secciones sin salida, nos queda un único camino que

seguir, en la mayoría de los casos, siendo cualquiera de las paredes la solución, por tanto, se podría usar también la mano izquierda. Pero este método tiene unas limitaciones, una de ellas es que, si un laberinto tiene secciones aisladas o ciclos, no se podrá nunca llegar al final, otra limitación es la lentitud.

Lo bueno del método mencionado anteriormente es que no precisamos ni de memoria, ni de la posición final. Si quisiéramos un algoritmo más eficiente tendríamos que ser capaces de implementar algún tipo de memoria.

Los algoritmos más destacables en su labor son: Dijkstra [9] y A*. El primero de ellos fue concebido en 1959 por Edsger W. Dijkstra, dicho algoritmo determina el camino más corto dado un vértice de origen y el resto de los vértices en un grafo dirigido. Como ya se mencionó con anterioridad, un laberinto puede ser representado como un grafo, incluso uno dirigido, en el cual todos los vértices apuntan al siguiente sin volver atrás. La idea principal es la de ir explorando todos los caminos más cortos que parten del origen hasta el resto de los vértices, la condición de parada de este algoritmo es que ya se haya explorado la totalidad del grafo.

El A* [10] fue desarrollado en 1968 por un equipo de investigación en el Instituto de Investigación de *Stanford*, este tiene la peculiaridad de que siempre es óptimo, mientras no se sobreestime la heurística. Funciona con dos estructuras de datos auxiliares, uno que se usa de cola de prioridad y ordenada por el valor acumulado, y otra que se usa de memoria de los nodos ya visitados. En cada iteración se expande el nodo más prometedor, por eso, su condición de parada es alcanzar el objetivo y, una vez alcanzado, se tiene ya la ruta más optima.

2.2.2 Generación de laberintos

Existen múltiples algoritmos de creación de laberintos [11], pero solo dos son importantes para este proyecto. El algoritmo Wilson y DFS.

El primero de estos se basa en elegir una celda aleatoria(en nuestro caso la casilla de abajo a la izquierda) y marcarla como visitada, después desde otra celda aleatoria se empieza a buscar un camino, eligiendo al azar que dirección seguir, hasta la otra celda. Cuando el camino se envuelve se crea un bucle, este se borra, se elimina del camino, y se continua desde la base del bucle. Por esa razón el primer paso de este algoritmo a veces tarda en completarse.

Una vez encontrado el primer camino, este se vuelve parte del laberinto, y entonces volvemos al paso dos, el elige otro punto y se busca un camino hasta llegar a una sección ya formada del laberinto. Y así hasta que no queden celdas sin formar parte de este.

Y, por último, el DFS, el cual es ejecutar este método y por cada celda que pasa ir eliminando la pared entre estas, para al final generar un laberinto perfecto.

2.3 Aprendizaje por refuerzo

El aprendizaje por refuerzo es un área de la inteligencia artificial, que está centrada en descubrir que acciones se deben tomar para maximizar la señal de recompensa. A este agente no se le dice que decisiones tomar, sino que, por el contrario, él solo debe explorar que tipo de acciones realizar para aumentar la función de recompensa.

Para simular el aprendizaje de sistemas biológicos reales necesitamos hacer una serie de suposiciones que simplificarían el comportamiento de nuestro agente, estas reducciones nos permiten disponer de un sistema más flexible para proyectar diversas situaciones en nuestro sistema y nos permitirá extraer conclusiones más generales acerca de los algoritmos que implementan estos sistemas de aprendizaje.

En general, tenemos que suponer que los agentes siguen un proceso de decisión de Markov [12]:

- El agente debe percibir un conjunto finito de estados distintos de su entorno y dispone de otro conjunto finito de acciones para interactuar con él.
- El tiempo avanza de forma discreta y en cada instante de tiempo, el agente percibe un estado concreto, selecciona una posible acción y la ejecuta, obteniendo un nuevo estado.
- El entorno responde a la acción del agente mediante recompensas, positivas o negativas.
- Tanto la recompensa, como el estado siguiente no tienen por qué ser conocidos de manera previa por el agente.

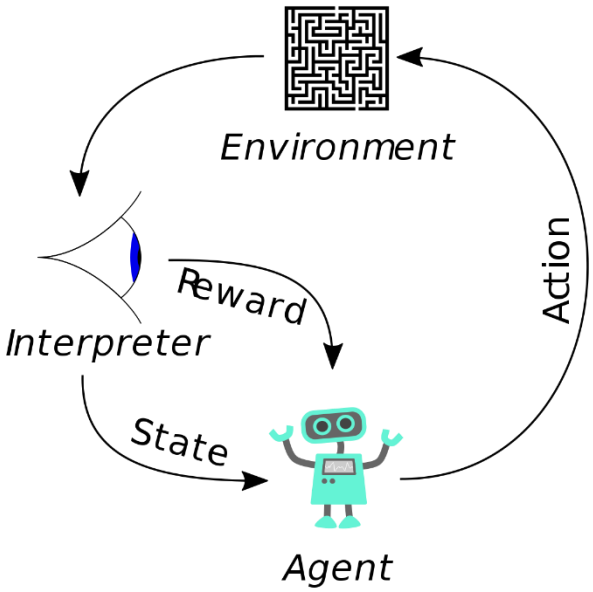


Figura 1: Encuadre típico de un escenario en Aprendizaje por Refuerzo

2.3.1 Casos famosos

Existen multitud de empresas que han desarrollado productos usando esta técnica de inteligencia artificial, pero entre ellos destaca una, *DeepMind*. Esta es una empresa del Reino Unido fundada en 2010, sus productos más famosos han sido *AlphaGo* [13] *y AlphaStar* [14].

En 2015, el *AlphaGo* original, se convirtió en el primer programa de ordenador en ganar a un jugador profesional de GO [15], en un tablero de tamaño completo de 19 × 19. Este es un juego para dos personas originario de China, el cual consiste en controlar una cantidad de territorio mayor al adversario. A pesar de su supuesta simplicidad requiere una gran cantidad de estrategia, siendo así más complicado que el propio ajedrez.

AlphaGo usa un algoritmo de búsqueda de árboles de Montecarlo [16] para encontrar sus movimientos basados en el conocimiento previamente aprendido mediante el aprendizaje automático. Actualmente está considerado el mejor jugador del mundo en Go, así como posiblemente de ajedrez.

AlphaStar es un programa capaz de jugar al StarCraft II, este se dio a conocer en enero de 2019 y para agosto de ese mismo año llego al nivel de Gran Maestro. A diferencia de *AlphaGo*, esta inteligencia

al principio aprendió a través de la imitación de los datos guardados de los juegos de humanos contra humanos, esto fue necesario para resolver el problema de la exploración ya que sería como encontrar "una aguja en un pajar". Mas tarde se perfeccionó jugando contra sí mismo, utilizando técnicas de aprendizaje por refuerzo.

2.3.2 Algoritmos

Existen multitud de algoritmos que se pueden usar en conjunto con el entrenamiento por refuerzo, pero nos vamos a centrar en los dos que se ven en este proyecto, PPO (Proximal Policy Optimization) y LSTM (Long short-term memory).

El primero de ellos PPO, es un tipo de algoritmo de aprendizaje profundo por refuerzo, es decir aprendizaje por refuerzo con la ayuda de redes neuronales. Este algoritmo surge del análisis de los algoritmos ya existentes, observando donde fallan y analizando el por qué. Muchas veces cuando se entrena un algoritmo clásico surge el problema de que intenta mejorar en una dirección errónea, para evitar eso y aportar estabilidad, se planteó la idea de limitar cuan diferente puede ser una nueva política respecto a la anterior, esto se logra a través de una serie de cálculos complejos, el problema de realizar dichas operaciones viene cuando tienen que ejecutar millones de veces.

El algoritmo PPO [17] opta por hacer una adaptación de algoritmos clásicos, para que sus cálculos se acerquen a los ideales, pero añadiendo pequeñas limitaciones con una función llamada "*Clipped surrogate objective function*", que limita los cambios en las políticas de manera constante.

El algoritmo LSTM también es usado en el campo del aprendizaje profundo por refuerzo, el cual tiene como característica que dispone de conexiones de retroalimentación. Cada unidad de LSTM dispone de una celda, una puerta de entrada, una puerta de salida y una puerta de olvido. La celda recuerda los valores en intervalos de tiempo aleatorios y las tres puertas regulan el flujo de información dentro y fuera de la celda

Estas redes, gracias a la memoria que implementa, son adecuadas para los problemas que necesitan almacenar información, o en nuestro caso, la evolución, el punto o recorrido del laberinto donde hemos estado y poder recuperarlo más adelante.

Capítulo 3
Desarrollo

3.1 Entorno de desarrollo y herramientas utilizadas

El entorno de desarrollo elegido fue Unity [18], gracias a su versatilidad a la hora de crear escenarios y facilidad para implementar el apartado gráfico, además cuenta con *ML- Agents Toolkit* [19], que es un kit de herramientas para el aprendizaje automático, el cual facilita la implementación y uso de inteligencia artificial en este entorno.

El lenguaje de programación elegido es C#, debido a su parecido a C++ y su integración en Unity, y el entorno para el desarrollo del código fue Visual Studio.

3.2 Creación de escenarios

Para la realización de este proyecto se precisaba de un método capaz de generar escenarios de manera aleatoria y automática, con solo darle unas dimensiones este sea capaz de generar un laberinto perfecto [20]. Este tipo de escenarios serán la base de nuestros experimentos.

Para esta tarea, primero se implementó una serie de bucles, los cuales generarían un total de X columnas e Y filas, siendo estas las dimensiones de nuestro problema.

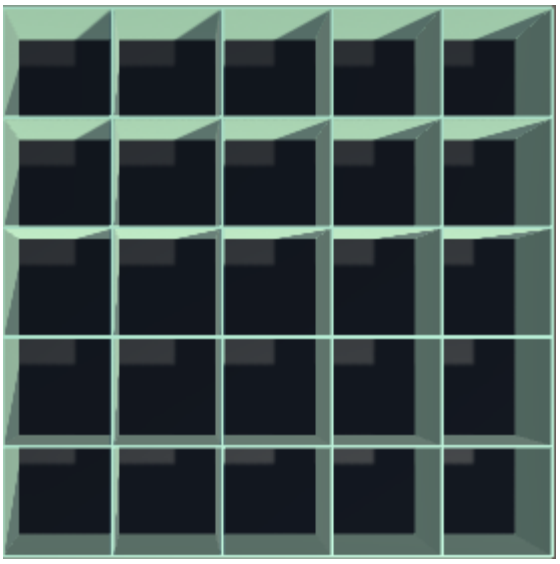


Figura 2: Rejilla de Laberinto

Acto seguido, se crean un total de X * Y células, y se le asigna a cada una cuál es la pared correspondiente a cada punto cardinal: norte, sur, este y oeste, para así tener en un vector todas las celdas de nuestro escenario, con sus correspondientes muros. Después, se ejecuta un algoritmo de DFS (Depth First Search), que parte de un punto aleatorio de nuestro mapa y su objetivo es recorrer todo el espacio disponible, decidiendo de manera aleatoria a qué vecino visitar, destruyendo las paredes que encuentre a su paso. De esta manera, conseguimos un laberinto perfecto, el cual no deja ninguna casilla sin recorrer y ningún punto inaccesible y de tipo multiviario,

ya que existen diversos caminos sin salida que resultan distintos cada vez que se ejecutan.

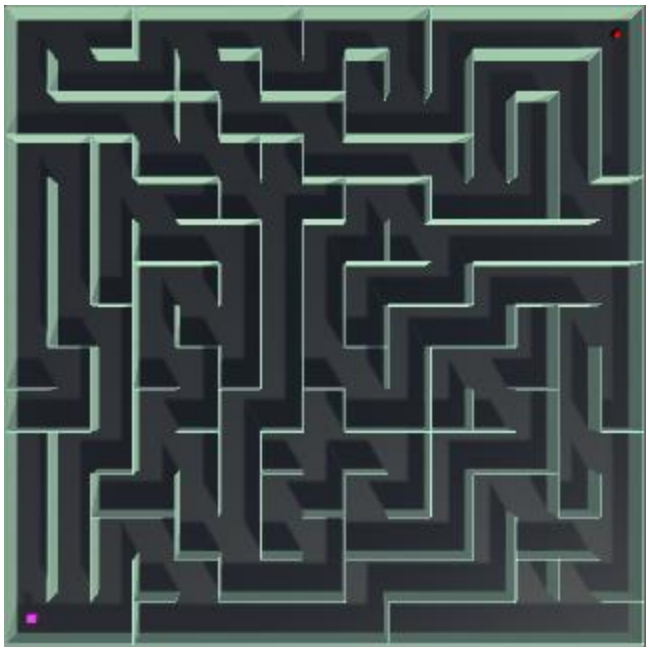


Figura 3: Laberinto completo

Para facilitar el trabajo con este tipo de escenarios, se creó en la misma clase, un vector de caracteres. Esto se hizo con el fin de darle profundidad a las paredes, debido a que, cuando se deciden las dimensiones del problema no se contemplan en él los muros, por tanto, estos carecen de grosor, como se representan en la figura 2. Para ello, se instancia una matriz bidimensional de tamaño: X = X * 2 -1 y de Y = Y * 2 -1, esta fórmula nace de querer dividir nuestro laberinto en X partes iguales, entonces para ello tendremos que usar X – 1 líneas. Este aumento de tamaño se compensa haciendo que el agente se mueva dando el doble de pasos, consiguiendo así, que las dimensiones desde la perspectiva del agente sean las mismas.

```
actualPosX = actualPosX + (int)verticalChange * 2;
actualPosY = actualPosY + (int)horizontalChange * 2;
```

Figura 4: Corrección de movimiento

Posteriormente, se rellena dicho vector con '·' si es un posible camino y de '#' en caso de ser un elemento imposible de traspasar. Con esto, la manipulación y percepción del entorno se hace de una manera mucha más cómoda y eficiente para nuestro agente.

3.3 Creación del Agente

Ambas clases, tanto la encargada de crear el laberinto llamada *MazeArea*, como la responsable de los agentes llamada *MazeAgent*, heredan de clases: *Area* y *Agent,* que pertenecen al toolkit de ML-Agents y contienen los métodos y variables necesarias para manejar el comportamiento durante el entrenamiento.

El agente puede ser instanciado en cualquiera de las cuatro esquinas del mapa, siendo esa la entrada al laberinto, esta decisión se realiza de manera aleatoria, una vez decidido el punto de partida se establece el destino, el cual se escoge de la misma manera, pero solo entre las esquinas sin ocupar. Esto se hizo con la intención de que el agente recorra la máxima distancia posible, pero no siempre es así, pues existe la posibilidad de que exista un camino directo entre el inicio y el fin del laberinto, debido a la aleatoriedad, tanto del escenario, como en la selección de principio y final.

Para determinar cómo es el movimiento de nuestro agente, primero se tiene que conocer cuántas acciones será capaz de realizar. Una vez fijado el número de acciones, hay que especificarlas en el inspector de elementos de Unity, en nuestro caso particular, solo tendrá un tipo de movimiento, el desplazamiento unidimensional, que se puede mover en cualquier dirección de los puntos cardinales.

Además de esto, para que nuestra inteligencia sepa qué acciones están bien y cuáles mal, se le tiene que recompensar, este indicativo de buen comportamiento se recomienda que no exceda el valor 1.0F. Esto, sumado al hecho de que, para forzar al agente a resolver el problema en el menor número de pasos posibles, hay que darle en cada iteración una pequeña recompensa negativa, hizo que determináramos que la mejor opción era que en cada iteración le damos un beneficio negativo de seis veces el tamaño del escenario, normalizando ese valor a uno. De esta manera, conseguimos que si tarda más de lo estipulado será recompensado de manera negativa, y en caso contrario se le premiará positivamente, pero nunca superando el valor máximo de 1.0F [21].

```
AddReward(-1.0f / (area.xSize * area.ySize * 6));
```

Figura 5: Normalización de las recompensas

También, hay que determinar el número máximo de pasos que puede estar nuestra inteligencia resolviendo un mismo escenario, si se queda en una esquina sin poder salir estaría atrapado ahí de manera indefinida, sin poder mejorar y solo empeoraría el modelo actual.

Y, por último, nos queda concretar lo que el agente es capaz de observar, para ello, intentamos darle los mismos conocimientos del entorno del que disponen los algoritmos actuales, su posición actual y la posición final, ambas son dadas en sus componentes X e Y, también se le adjuntan la cantidad de celdas que son capaces de moverse en cada dirección.

```
AddVectorObs(positiveVerticalSpace);   //North
AddVectorObs(negativeVerticalSpace);   //South
AddVectorObs(positiveHorizontalSpace); //East
AddVectorObs(negativeHorizontalSpace); //West
AddVectorObs(actualPosX);              //Agent position
AddVectorObs(actualPosY);
AddVectorObs(finalX);                  //Final position
AddVectorObs(finalY);
```

Figura 6: Vector de observaciones

3.4 Búsqueda de la configuración óptima

Los modelos creados se han entrenado en dos dispositivos de manera paralela, un ordenador de sobremesa y un portátil.

Sobremesa:

- CPU: Intel Core I5 8600K
- GPU: GeForce GTX 1060 3GB
- RAM: 16 GB
- SO: Windows 10 Pro

portátil:

- CPU: Intel Core I7 9750H
- GPU: GeForce GTX 2060 6GB
- RAM: 16 GB
- SO: Windows 10

Existen más de veinte posibles ajustes que poder modificar a la hora de configurar el fichero de entrenamiento [22], de las cuales la mayoría de las variables son de carácter continuo, esto hace que la búsqueda de un modelo óptimo sea una tarea complicada. Por esa razón se decidió acotar el problema.

Al comienzo, empezamos un estudio de rendimiento, para saber que combinación de variables hace que el agente sea más eficiente, pero sólo con dos variables: learning_rate y beta. Éstas fueron elegidas debido a su relevancia, la primera es: "la tasa de aprendizaje utilizada para actualizar el módulo de curiosidad intrínseca", es decir, controla cuanto cambia el modelo en respuesta al error estimado, cada vez que se actualizan los pesos. La segunda variable controla cuan aleatoria es la exploración de un escenario. Según la documentación oficial de ML-Agents dichos ajustes deben estar entre $[1E^{-5}, 1E^{-3}]$ y $[1E^{-4}, 1E^{-2}]$ respectivamente, estas dos opciones son variables continuas, por tanto, para poder realizar un estudio se tiene que discretizar. En consecuencia, se creó una tabla que, divide cada una de las variables en seis posibilidades, dando lugar a una matriz cuadrada de treinta y seis elementos en total.

		Beta					
		1,00E-04	2,08E-03	4,06E-03	6,04E-03	8,02E-03	1,00E-02
Learning Rate	1,00E-05	1 1	1 2	1 3	1 4	1 5	1 6
	2,08E-04	2 1	2 2	2 3	2 4	2 5	2 6
	4,06E-04	3 1	3 2	3 3	3 4	3 5	3 6
	6,04E-04	4 1	4 2	4 3	4 4	4 5	4 6
	8,02E-04	5 1	5 2	5 3	5 4	5 5	5 6
	1,00E-03	6 1	6 2	6 3	6 4	6 5	6 6

Tabla 1: Matriz de discretización de valores

El resto de los ajustes se quedan constantes durante todas las pruebas, entre ellos cabe destacar: el número de capas usada es de cuatro y a su vez cada *layer* tiene quinientas doce unidades ocultas, además, para que el estudio sea justo, todos los modelos son entrenados la misma cantidad de

iteraciones: $3.07E^6$. Cuando se determine cuál es la configuración óptima se entrenará durante un número mayor de iteraciones para saber cuál es el límite.

Cada modelo fue entrenado por un tiempo aproximado de cinco horas, que es el tiempo aproximado que le lleva a los dispositivos terminar el entrenamiento. Para este experimento primero se usó el algoritmo PPO (Proximal Policy Optimization) [23]. En adición, se hizo uso de un *curriculum* [24] de entrenamiento que se usa cuando la tarea a realizar es muy complicada y se necesita de un aprendizaje progresivo para poder llegar a realizarla de manera eficiente. Esta se implementó de manera que, primero resuelva laberinto de tres por dos y, según aumenta la cantidad de recompensa recibida, va aumentando de manera gradual su tamaño. Pasando de tres por dos a tres por tres, luego a tres por cuatro, cuatro por cuatro, hasta llegar a veinte.

```
"measure": "reward",
"thresholds": [ 0.7, 0.7, 0.65, 0.65, 0.65, 0.6, 0.6, 0.6, 0.5, 0.5, 0.45, 0.45, 0.3, 0.3, 0.3],
"min_lesson_length": 200,
"signal_smoothing": true,
"parameters": {
    "x_size": [ 2.0, 3.0 , 4.0, 4.0, 4.0, 5.0, 6.0, 6.0, 6.0, 7.0, 8.0, 8.0, 8.0, 9.0, 10.0, 10.0 ],
    "y_size": [ 3.0, 3.0 , 3.0, 4.0, 5.0, 5.0, 5.0, 6.0, 7.0, 7.0, 7.0, 8.0, 9.0, 9.0,  9.0, 10.0 ]
}
```

Figura 7: Ejemplo de Curriculum

```
void configureAreaCurricula()
{

    int sizeX = (int)academy.FloatProperties.GetPropertyWithDefault("x_size", area.xSize);
    int sizeY = (int)academy.FloatProperties.GetPropertyWithDefault("y_size", area.ySize);
    area.xSize = sizeX;
    area.ySize = sizeY;
    agentParameters.maxStep = (((area.xSize * 2 - 1) * (area.ySize * 2 - 1)) * 40);
}
```

Figura 8: Ejemplo de actualización de Curriculum

		Beta					
		1,00E-04	2,08E-03	4,06E-03	6,04E-03	8,02E-03	1,00E-02
Learning Rate	1,00E-05	No Completa	110	102	119	95	89
	2,08E-04	No Completa	206	138	207	229	158
	4,06E-04	No Completa	394	256	221	402	245
	6,04E-04	No Completa	399	No Completa	No Completa	671	718
	8,02E-04	No Completa	No Completa	No Completa	No Completa	No Completa	No Completa
	1,00E-03	No Completa	No Completa	No Completa	No Completa	No Completa	No Completa

Tabla 2: Resultados de tabla de rendimiento, media

		Beta					
		1,00E-04	2,08E-03	4,06E-03	6,04E-03	8,02E-03	1,00E-02
Learning Rate	1,00E-05	No Completa	377	299	352	314	256
	2,08E-04	No Completa	632	440	587	660	529
	4,06E-04	No Completa	940	731	685	968	688
	6,04E-04	No Completa	935	No Completa	No Completa	1204	1264
	8,02E-04	No Completa	No Completa	No Completa	No Completa	No Completa	No Completa
	1,00E-03	No Completa	No Completa	No Completa	No Completa	No Completa	No Completa

Tabla 3: Resultados de tabla de rendimiento, desviación típica

Como se muestran en las tablas dos y tres la configuración óptima, dentro de nuestro espectro de búsqueda, es la correspondiente a la casilla 1-6, con los valores de: *learning rate* $1.00E^{-5}$ y beta $1.00E^{-2}$. El tamaño máximo de laberinto al que llegaron fue de cinco por cinco, por tanto, aquellos modelos que no llegaron a ese nivel no tienen sentido compararlo con los otros. Una vez determinada la posición ganadora se creó otro modelo con las mismas características, pero aumentando el tiempo de entrenamiento, pasando de $3.07E^6$ *max steps*, a $4.00E^7$ y de cinco horas a casi sesenta y cinco. Para hacer una comparativa justa se generó un estudio para los modelos, donde ambos se enfrentarían a cinco mil laberintos aleatorios de cinco por cinco.

Promedio 3,07E6	Desviación Estándar 3,07E6	Veces que no termina 3,07E6
114,099	355,5153423	32
Promedio 4E7	Desviación Estándar 4E7	Veces que no termina 4E7
105,1078	331,7395714	31

Tabla 4: Comparación de modelos

Al ver los resultados se evidencia que, usando un t-test de muestras independientes (nos hemos basado en el Teorema Central del Límite, ya que disponemos de datos de alta asimetría a la derecha y un tamaño muestras de n=5000), que por término medio los modelos no son estadísticamente diferentes (p = 0.191). En otras palabras, al incrementar el *max_steps* trece veces, usando el algoritmo PPO, no se consiguen, por término medio, mejoras significativas.

3.5 Cambio de algoritmo

Debido a la imposibilidad de conseguir mejores resultados, se optó por cambiar el algoritmo usado para su entrenamiento. En vez de usar el PPO, se decidió usar el LSTM (*Long Short-Term Memory*) [25], el cual dispone de memoria y es capaz de utilizarla. Para usar este tipo de red, hay que añadir los parámetros de: activar el *flag* de que se desea usar este algoritmo, la longitud de las secuencias de experiencias y el tamaño de la memoria. El resto de los ajustes siguen igual, la única recomendación es bajar el número de *layers.*

Primero se entrenó bajo las mismas condiciones, bajo las cuales hicimos el estudio del modelo optimo, para ver si el cambio de red estaba justificado. Al hacer el modelo nos dieron unos resultados que no se pueden comparar con el otro algoritmo.

Promedio LSTM 3,07E6	Desviación Estándar LSTM 3,07E6	Veces que no termina LSTM 3,07E6
29,8936	99,32006834	2

Tabla 5: Estadísticas LSTM 3.07E6 5X5

Además de mejorar notablemente el promedio y la desviación estándar, reduce también las veces que no consigue resolver un laberinto de cinco por cinco, pero lo mejor es que este consigue resolver laberintos cuadrados, de hasta nueve filas de tamaño.

Promedio	Desviación Estándar	Veces que no termina
825,5912	2232,530652	146

Tabla 6: Estadísticas LSTM 3.07E6 9X9

Sin embargo, en la tabla seis podemos comprobar que, a medida que aumenta el tamaño del problema, aumenta de manera drástica el promedio de pasos, la desviación estándar se dispara y no consigue resolver casi el 3% de los escenarios.

Tras ver la mejora obtenida por la nueva red, se decidió, al igual que con la anterior, ponerle un número mayor de iteraciones, para ver hasta dónde podía llegar si le poníamos tres veces el número de *max_steps* que tenía anteriormente.

Promedio	Desviación Estándar	Veces que no termina
356,4626	1479,620141	62

Tabla 7: Estadísticas LSTM 1.224E7 9X9

Como se puede apreciar en la tabla siete, el aumento de las iteraciones, junto con el crecimiento del tiempo de entrenamiento, que pasó de doce horas a casi treinta y unas, es notable que también hay un incremento del rendimiento, disminuyendo a más de la mitad el promedio y las veces que no logra acabar los escenarios y bajando en mil unidades la desviación típica.

Lo mejor es que esta nueva inteligencia es capaz de resolver laberintos cuadrados de hasta quince filas, los cuales hubiera sido imposible alcanzar, de seguir usando la red neuronal PPO.

También se intentó entrenar el agente con otras dos redes, *Behavioral Cloning* y *Soft Actor Critic.* En la primera de ellas los ajustes que tenemos son exactamente iguales a las disponibles en las redes anteriores, por ende, se hizo uso de la configuración optima obtenida con anterioridad, y en las opciones exclusivas de este entrenamiento se dejaron las que vienen de base. Para entrenar a nuestro modelo primero necesitamos un archivo *.demo*, para ello se dejó el ordenador encendido por doce horas, capturando casi cincuenta y tres mil experiencias.

Para el segundo, dado que no comparte los mismos ajustes con los anteriores, se intentó realizar otro estudio, esta vez usando la única característica común que tiene esta red con las anteriores, el *learning rate*, pero dada la diferencia de ajuste se iba a llevar a cabo otro estudio para encontrar la configuración adecuada para este.

Ambos llegaron a entrenar modelos, pero de ninguno de los dos se consiguieron resultados, tras más de doce horas de entrenamiento apenas eran capaces de resolver laberintos pequeños. Se busco el problema y se intentó solucionar, pero no se consiguió dar con el error.

Capítulo 4
Resultados

4.1 Comportamiento de los agentes

Los modelos creados tienen un movimiento bastante peculiar, este patrón de comportamiento cambia mucho, dependiendo de qué tipo de red fue usada para su entrenamiento. La principal diferencia entre ambos es la forma de moverse, mientras que los entrenados con PPO hacen un movimiento ida y vuelta [26], a la vez en que avanzan, en el que es tutelado por la red LSTM tiene un movimiento más decidido [27], el cual solo replica ese mismo tipo de comportamiento una vez encuentra un callejón sin salida e intenta traspasarlo.

Cabe destacar también que el modelo capaz de resolver laberintos mayores imita, en mayor medida el comportamiento de los algoritmos actuales, decidiendo que camino elegir en base a que posición se encuentra más cerca de la salida. Sin embargo, los otros modelos tienen una conducta más errática, estos en algunos escenarios son capaces de tener el final justo enfrente e irse por otro camino, cosa que también pasa con los LSTM, pero en muchas menos ocasiones.

Y en ambos modelos, cuando se quedan atrapados, intentan de manera reiterada ir por el mismo camino, sin encontrar nuevas alternativas, el PPO hace poco recorrido hacia atrás, pero por el contrario el LSTM hace uso de *Backtracking,* hasta casi llegar al principio y vuelve a empezar, consiguiendo así mejores resultados.

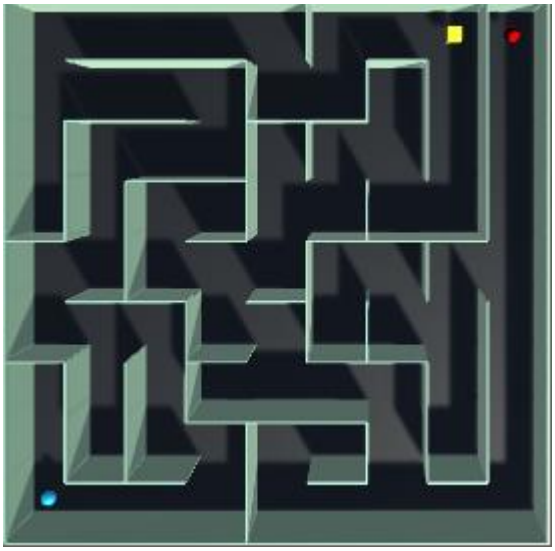


Figura 9: Modelo intentando ir hacia el final

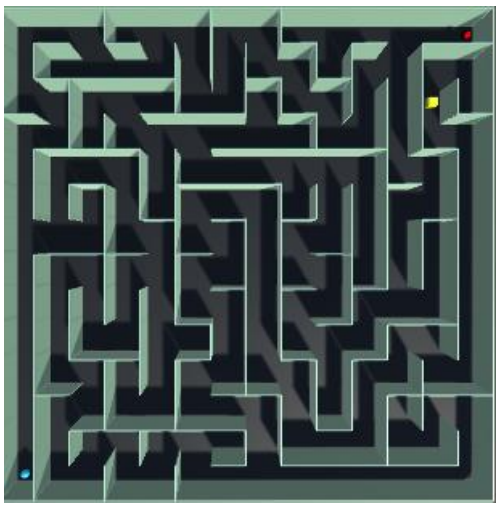


Figura 10: Atrapado en un laberinto de 14X14

Tanto en la figura nueve como en la diez se puede apreciar que el modelo se queda trabado en esas posiciones, al tener la salida justo al lado, y por no empeorar el beneficio, no exploran más, aunque como en la figura seis se muestra, puedan tener la salida simplemente siguiendo el camino que tiene delante.

4.2 Problemática del estudio final

Para poder tener una referencia real de la eficacia de los modelos creados con anterioridad se propuso crear un estudio donde se compararán los pasos que necesita el algoritmo A* para resolver el mismo laberinto que resuelve nuestra inteligencia, pero existen dos problemas:

1. El algoritmo A* no recorre de manera lineal el laberinto, se va teletransportando a los puntos que tienen una posibilidad mayor de ser óptimo.
2. El modelo simula un algoritmo aleatorio debido a que, en un mismo laberinto, cada vez que se ejecute, toma decisiones distintas. Esto complica la realización de un estudio ya que tenemos que tomar como medida de rendimiento un promedio de cuánto tiempo tarda en llegar el final.
3. Dificultad a la hora de saber la complejidad de un laberinto, dado que se generan de manera aleatoria, un escenario a simple vista pequeño puede ser más complicado que uno de un tamaño considerablemente mayor.

4.3 Procedimiento del estudio

Para la realización de este estudio primero se debía de disponer algún tipo de “contrincante” al que enfrentarlo, para ello se decidió usar el algoritmo, el cual como ya se mencionó con anterioridad es uno de los más utilizados para este tipo de tareas. El problema surge cuando este tiene una funcionalidad que no está contemplada en nuestra inteligencia artificial, está capacitado de ir de un lado a otro del laberinto en una sola iteración, esto es debido a su funcionamiento, ya que guarda en su lista abierta todas las posiciones vistas hasta el momento, con su posición en el espacio y su distancia hasta el final, es capaz de ir al lugar que tenga el valor de distancia más pequeño.

Para solucionar esto, se optó por la implementación de un algoritmo de DFS que simule lo máximo posible el funcionamiento del A*, pero sin dicha capacidad. El funcionamiento de este es sencillo, ya que recorre el laberinto en una misma dirección, guardando en memoria el camino recorrido hasta que se encuentra con una intersección, en la cual tomará el camino que le acerque

más a la salida, usando la Distancia Manhattan como la forma de saber la distancia más pequeña hasta un punto, esta es la misma que es usada normalmente por el algoritmo A*.

Así mismo, queríamos probar si el algoritmo DFS implementado era peor que el A* y llegando a la conclusión de que:

Dado un laberinto aleatorio con dimensiones m y n siendo m,n ∈ {5,6,7,8,9} entonces:

1. El número de pasos que el algoritmo A^* excede, en promedio, al número de pasos óptimos es 5, siendo la desviación típica 7.58. Concretamente dicho promedio se encuentre en el intervalo (4.19;7.19) al 95% de confianza [28].
2. El número de pasos que el algoritmo DFS modificado excede, en promedio, al número de pasos óptimos es 10, siendo la desviación típica 16.8. Concretamente dicho promedio se encuentre en el intervalo (6.75;13.4) al 95% de confianza.
3. A través de un t-test unilateral ($p<0.001$) de muestras emparejadas podemos evidenciar que el algoritmo DFS modificado excede, en promedio de pasos, al algoritmo A*.

Una vez determinado con quién comparar nuestros modelos se decidió que la mejor manera de hacerlo era enfrentándolos entre sí, los tres algoritmos (A*, DFS modificado y el modelo), estos intentarían resolver el mismo laberinto, pero dada la aleatoriedad de los modelos generados se concluyó que la mejor manera de poner a prueba su capacidad, era la de hacerlo recorrer el mismo laberinto, que los otros algoritmos, cien veces y de ahí sacar una media que nos proporcionaría el rendimiento de nuestra inteligencia.

En el estudio realizado se sacaron muestras de cien laberintos distintos, los cuales son de tamaño aleatorio, entre cinco y el número máximo que sea capaz de resolver, pudiendo no ser cuadrados. Como ya se dijo anteriormente, se iba a ejecutar el modelo, cien veces en cada laberinto, haciendo así que se tengan que resolver más de diez mil laberintos.

Una vez obtenidos los datos se creó una tabla donde se pueden observar cuantas veces, nuestro modelo igual o mejor que los algoritmos nombrados anteriormente, o si en alguna ocasión es óptimo.

Para poder sacar datos útiles, se llegó a la conclusión de que era mejor hallar el número medio de pasos que nuestro modelo excede, en promedio, al número de pasos óptimos. Esto significa que, a los diez mil datos recabados se les va a restar los pasos óptimos, para así obtener la media de pasos en exceso que hace nuestra inteligencia en ese mismo laberinto. Y después, promediar de nuevo esos datos para sacar el promedio de la media de veces que nos pasamos del óptimo.

Laberinto	Óptimo	Pasos A*	Pasos DFS	Veces óptimo	Veces < A*	Veces = A*	Veces < DFS	Veces = DFS
6x5	11	18	23	0	12	3	24	5
8x6	14	20	30	1	30	13	81	2

Tabla 8: Ejemplo de estudio

En la tabla ocho se muestra un ejemplo de los datos recopilados en nuestro estudio, donde la primera columna representa el tamaño del laberinto a recorrer, la segunda la cantidad de pasos que tiene el recorrido óptimo, la tercera el numero de pasos que necesito el algoritmo A* para resolver dicho escenario, seguido de los pasos que necesitó el DFS. Después contabilizamos cuantas veces nuestro modelo fue capaz de realizar el recorrido óptimo. Para terminar, realizamos un conteo de cuantas veces nuestra inteligencia fue capaz de superar el A* y cuantas veces lo consiguió igualar, luego hacemos exactamente lo mismo con el DFS.

4.4 Resultados modelo LSTM 3.07E6

La tabla obtenida [29] tras la computación de los datos, es demasiado grande para estar en esta memoria, pero en su lugar se desarrolló una a modo de resumen.

Laberinto	Fi	Veces óptimo	% veces óptimo	Veces mejor o Igual A*	% veces mejor o igual A*	Veces igual o mejor DFS modificado	% veces mejor o igual DFS modificado
5x5	2	104	52.00%	113	56.50%	104	52.00%
5x6	3	105	35.00%	160	53.33%	180	60.00%
5x7	7	121	17.29%	229	32.71%	227	32.43%
5x8	4	160	40.00%	175	43.75%	201	50.25%
5x9	2	34	17.00%	50	25.00%	48	24.00%
6x5	3	99	33.00%	104	34.67%	105	35.00%
6x6	7	132	18.86%	155	22.14%	187	26.71%
6x7	6	195	32.50%	231	38.50%	271	45.17%
6x8	3	56	18.67%	79	26.33%	80	26.67%
6x9	3	84	28.00%	84	28.00%	84	28.00%
7x5	7	212	30.29%	226	32.29%	235	33.57%
7x6	3	136	45.33%	163	54.33%	142	47.33%
7x7	1	8	8.00%	13	13.00%	19	19.00%
7x8	2	87	43.50%	89	44.50%	87	43.50%
7x9	3	75	25.00%	91	30.33%	75	25.00%
8x5	8	355	44.38%	425	53.13%	496	62.00%
8x6	6	129	21.50%	176	29.33%	185	30.83%
8x7	3	78	26.00%	90	30.00%	78	26.00%
8x8	2	89	44.50%	89	44.50%	98	49.00%
8x9	5	101	20.20%	109	21.80%	121	24.20%
9x5	3	65	21.67%	65	21.67%	69	23.00%
9x6	3	32	10.67%	57	19.00%	78	26.00%
9x7	5	143	28.60%	154	30.80%	173	34.60%
9x8	5	121	24.20%	160	32.00%	163	32.60%
9x9	4	12	3.00%	72	18.00%	92	23.00%

Tabla 9: Estudio LSTM 3.07E6 simplificada

En la tabla nueve se puede observar el tamaño de los laberintos usados, así como sus frecuencias, junto con: cuántas veces nuestro modelo hace el recorrido óptimo y cuántas veces hace los mismos pasos, o menos, que los algoritmos rivales, junto con sus porcentajes. Pero los datos aquí reunidos no son representativos, por el hecho de que, aunque los escenarios tengan las mismas dimensiones, la naturaleza del problema es distinta, ya que sus patrones no son los mismos. Con dicha tabla se intenta ilustrar los datos recogidos por el estudio de una manera condensada e ilustrativa.

Laberinto	Óptimo	Pasos A* óptimo	Pasos DFS modificado	Veces óptimo	Veces mejor o Igual A*	Veces igual o mejor DFS modificado
9x9	19	67	137	9	65	78
9x9	19	27	35	0	2	9
9x9	45	58	51	0	0	0
5x7	25	33	39	0	10	19

Tabla 10: problemática de la dificultad

En la tabla diez se puede comprobar de manera empírica, la dificultad de determinar que problemas son más complicados, visto que, teniendo tres laberintos de la misma dimensión y dos de ellos resolverse en los mismos pasos, la cantidad de iteraciones dadas por los otros algoritmos se disparan, no obstante, aquel laberinto con un numero de iteraciones mayor por parte del A*, es también el único de los ahí mostrados que ha conseguido hacer el recorrido de manera óptima, mientras que el laberinto, que se resuelve en cuarenta y cinco pasos, fue incapaz de ni siquiera igualar a los competidores.

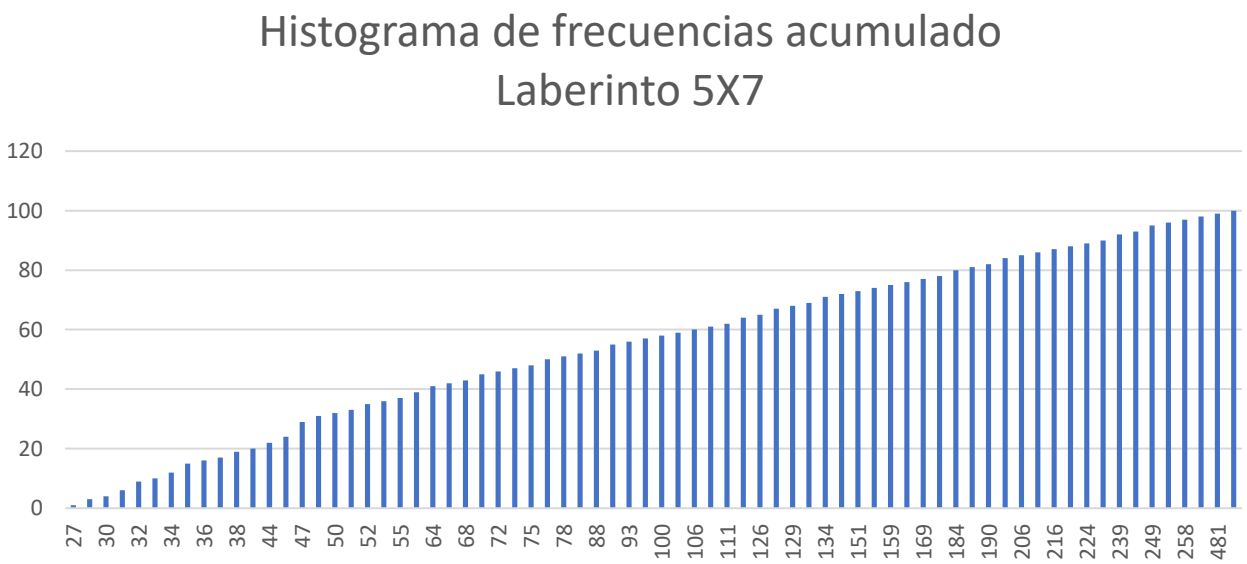


Figura 11: Histograma LSTM 3.07E6

En la figura once se representa el histograma acumulado de un laberinto de 5X7, el cual ha sido resuelto cien veces por nuestro modelo. Y en él se indican el número de pasos que tardó en terminar el escenario, en este se puede apreciar que el sesenta por ciento de los datos está por debajo de ciento once pasos.

Tras calcular la media de pasos en exceso dados por nuestro modelo, podemos concluir que:

Dado un laberinto aleatorio con dimensiones m y n siendo m, n ∈ {5,6,7,8,9}:

- Para un laberinto de 9x9 el número de pasos que el modelo excede, en promedio, al número de pasos óptimos es 56, siendo la desviación típica 77.63. Sin embargo, dado que el modelo es aleatorio, expresaremos el rendimiento del sujeto en promedio, a saber, el número medio de pasos que la inteligencia artificial excede, en promedio, al número de pasos óptimos es 108, siendo la desviación típica de las medias es 200. Concretamente dicho promedio se encuentre en el intervalo (68.3; 148) al 95% de confianza.

4.5 Resultados LSTM 1.224E7

Para comparar los dos modelos de manera más exhaustiva, decidimos hacerle exactamente el mismo estudio, donde se elegirían laberintos aleatorios de entre cinco y nueve. Por las dimensiones del estudio [30] es imposible mostrar su totalidad aquí, por esa razón se representará un estudio el cual no es representativo, debido a que agrupamos los laberintos por tamaño, pero la naturaleza de estos es distinta, no existen dos escenarios iguales en este estudio y por ende al juntarlos no nos sale una tabla fiel a la realidad.

Laberinto	FI	Veces óptimo	% veces óptimo	Veces mejor o Igual A*	% veces mejor o igual A*	Veces igual o mejor DFS modificado	% veces mejor o igual DFS modificado
5x5	5	198	39.6%	307	61.4%	287	57.4%
5x6	3	102	34.0%	179	59.7%	191	63.7%
5x7	2	112	56.0%	116	58.0%	153	76.5%
5x8	5	127	25.4%	196	39.2%	163	32.6%
5x9	8	318	39.8%	405	50.6%	370	46.3%
6x5	4	184	46.0%	212	53.0%	210	52.5%
6x6	4	76	19.0%	91	22.8%	89	22.3%
6x7	2	48	24.0%	93	46.5%	64	32.0%
6x8	3	56	18.7%	101	33.7%	173	57.7%
6x9	4	86	21.5%	136	34.0%	146	36.5%
7x5	4	81	20.3%	166	41.5%	125	31.3%
7x6	2	14	7.0%	56	28.0%	50	25.0%
7x7	4	72	18.0%	175	43.8%	202	50.5%
7x8	3	89	29.7%	180	60.0%	88	29.3%
7x9	3	20	6.7%	149	49.7%	115	38.3%
8x5	3	1	0.3%	11	3.7%	47	15.7%
8x6	4	14	3.5%	37	9.3%	36	9.0%
8x7	2	59	29.5%	75	37.5%	59	29.5%
8x8	8	122	15.3%	298	37.3%	182	22.8%
8x9	5	181	36.2%	230	46.0%	240	48.0%
9x5	5	45	9.0%	93	18.6%	128	25.6%
9x6	3	14	4.7%	71	23.7%	67	22.3%
9x7	5	69	13.8%	170	34.0%	153	30.6%
9x8	4	7	1.8%	88	22.0%	113	28.3%
9x9	5	55	11.0%	120	24.0%	184	36.8%

Tabla 11: Estudio LSTM 1.224R7 9X9 Max

En la tabla once podemos ver que, en comparación con su modelo predecesor, el cual podemos observar en la tabla nueve, este de manera general tiene unos porcentajes de aciertos mayor, así como en también crece cuantas veces es igual o mejor que los otros algoritmos.

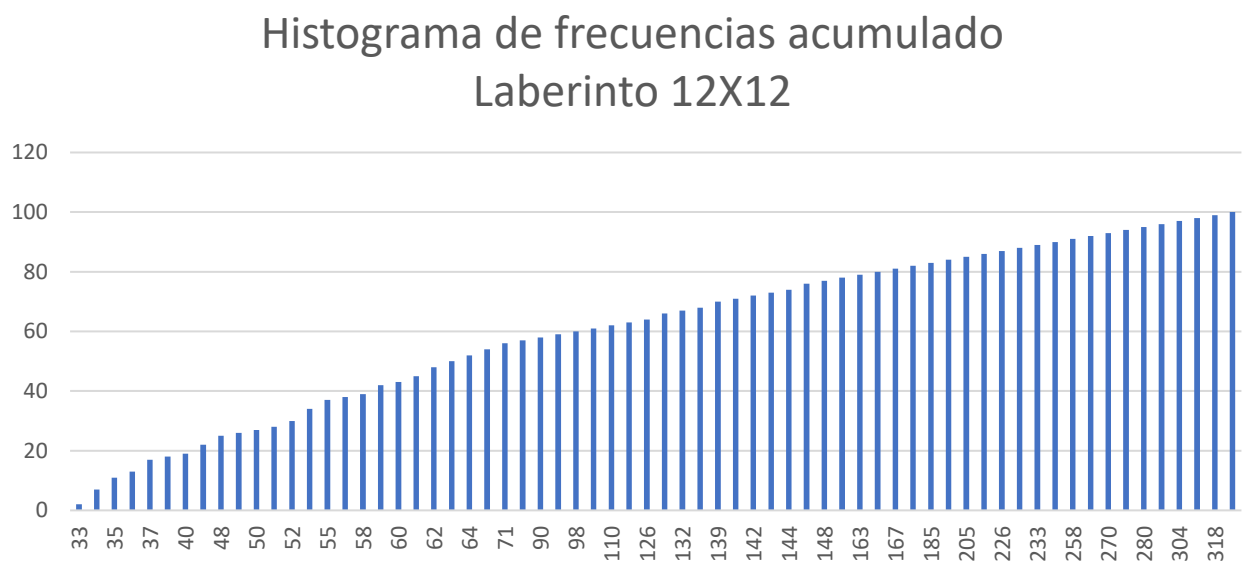


Figura 12: Histograma LSTM 1.224E7

En la figura doce se representa el histograma acumulado de un laberinto de 12X12, el cual ha sido resuelto cien veces por nuestro modelo. Y en él se indican el número de pasos que tardó en terminar el escenario, en este se puede apreciar que el sesenta por ciento de los datos está por debajo de noventa y ocho pasos.

De este estudio, podemos afirmar que:

Dado un laberinto aleatorio con dimensiones m y n siendo m, n ∈ {5,6,7,8,9}:

- Para un laberinto de 9x9 el número de pasos que el modelo excede, en promedio, al número de pasos óptimos es 38, siendo la desviación típica 28.11. Sin embargo, dado que el modelo es aleatorio, expresaremos el rendimiento del sujeto en promedio, a saber, el número medio de pasos que la inteligencia artificial excede, en promedio, al número de pasos óptimos es 85, siendo la desviación típica de las medias es 191. Concretamente dicho promedio se encuentre en el intervalo (47.7; 124) al 95% de confianza.

Con ello podemos decir sin miedo a equivocarnos que el modelo 1.224E7 es mejor en la resolución de laberintos, en tamaños comprendidos entre cinco y nueve, que el modelo 3.07E6.

Sin embargo, dado que esta inteligencia es capaz de resolver laberintos cuadrados de hasta quince filas, optamos por repetir una última vez el estudio, pero, aumentando el tamaño de generación aleatoria de laberintos.

Los resultados obtenidos en este estudio [31] es, al igual que la anterior, son demasiado grandes para ser mostrados en esta memoria, e incluso una tabla simplificada sigue siendo demasiado, dado al número de escenarios distintos, dando lugar a muchas más combinaciones que el anterior. Por eso mismo solo se mostrará una tabla que, disponga únicamente de aquellos laberintos que no fueron resueltos, ni una vez, de manera óptima.

Laberinto	Fi	Veces óptimo	% veces óptimo	Veces mejor o Igual A*	% veces mejor o igual A*	Veces igual o mejor DFS modificado	% veces mejor o igual DFS modificado
5x12	1	0	0.00%	0	0.00%	0	0.00%
6x13	1	0	0.00%	0	0.00%	2	2.00%
6x14	1	0	0.00%	41	41.00%	5	5.00%
6x8	1	0	0.00%	4	4.00%	9	9.00%
7x13	1	0	0.00%	11	11.00%	16	16.00%
7x15	2	0	0.00%	2	1.00%	10	5.00%
7x6	1	0	0.00%	11	11.00%	0	0.00%
8x14	1	0	0.00%	63	63.00%	11	11.00%
9x13	1	0	0.00%	0	0.00%	1	1.00%
9x15	1	0	0.00%	73	73.00%	25	25.00%
9x7	1	0	0.00%	12	12.00%	12	12.00%
9x8	1	0	0.00%	2	2.00%	2	2.00%
10x10	1	0	0.00%	0	0.00%	27	27.00%
10x15	1	0	0.00%	0	0.00%	2	2.00%
10x5	1	0	0.00%	1	1.00%	1	1.00%
10x6	1	0	0.00%	2	2.00%	13	13.00%
10x9	1	0	0.00%	57	57.00%	57	57.00%
11x8	1	0	0.00%	69	69.00%	91	91.00%
12x12	1	0	0.00%	18	18.00%	0	0.00%
12x14	2	0	0.00%	75	37.50%	66	33.00%
13x12	1	0	0.00%	0	0.00%	14	14.00%
13x13	1	0	0.00%	1	1.00%	3	3.00%
13x14	2	0	0.00%	10	5.00%	23	11.50%
13x15	1	0	0.00%	0	0.00%	0	0.00%
13x5	1	0	0.00%	0	0.00%	0	0.00%
14x13	1	0	0.00%	2	2.00%	2	2.00%
14x6	1	0	0.00%	0	0.00%	0	0.00%
14x7	1	0	0.00%	0	0.00%	0	0.00%
14x8	1	0	0.00%	0	0.00%	60	60.00%
15x15	2	0	0.00%	18	9.00%	30	15.00%
15x6	1	0	0.00%	3	3.00%	29	29.00%

Tabla 12: Laberintos no resueltos.

En esta tabla, al igual que la nueve y la once, sigue sin ser representativa, a simple vista parece que los resultados son peores, pero es porque se quedaron otras treinta y seis filas fueras, las pertenecientes a laberintos que si era capaz de resolver. En dicha tabla podemos comprobar que los laberintos que tienen una estructura rectangular en vez de cuadrada son aquellos que no consigue resolver de manera óptima, esto se puede deber a que siempre se entrenó en escenarios semi cuadrados.

Tras calcular la media de pasos en exceso dados por nuestro modelo, podemos concluir que:

Dado un laberinto aleatorio con dimensiones m y n siendo m, n ∈ {5,6,7,8,9,10,11,12,13,14,15}:

- El número medio de pasos que la inteligencia artificial excede, en promedio, al número de pasos óptimos es 506, siendo la desviación típica de las medias es 1172. Concretamente dicho promedio se encuentre en el intervalo (256; 759) al 95% de confianza.

Aunque este número es mucho peor que el anterior, este se debe a la alta probabilidad de generar escenarios rectangulares, pero como pudimos comprobar con anterioridad, este modelo sigue siendo mejor que su antecesor.

4.6 Laberintos no perfectos

Dado que el modelo fue entrenado únicamente en laberintos perfectos, se decidió probar su comportamiento en escenarios con ciclos y con múltiples caminos a la salida.

Para ello primero contamos el número de paredes que tiene un laberinto perfecto y así poder calcular cuantas de estas se desean eliminar, si el número obtenido es decimal, se decide truncar, ya que, consideramos que 4.9 paredes, son cuatro paredes completas y una sin terminar.

Hicimos tres estudios, eliminando el 20%, el 30% y el 40% y nos dio como resultado que:

Dado un laberinto aleatorio con dimensiones m y n siendo m, n ∈ {5,6,7,8,9}:

- El número medio de pasos que la inteligencia artificial excede, en un laberinto no perfecto al que se le eliminó el 20% de las paredes, en promedio, al número de pasos óptimos, es 72, siendo la desviación típica de las medias es 261.55. Concretamente dicho promedio se encuentra en el intervalo (20.81; 124.6) al 95% de confianza.
- El número medio de pasos que la inteligencia artificial excede, en un laberinto no perfecto al que se le eliminó el 30% de las paredes, en promedio, al número de pasos óptimos, es 60.80, siendo la desviación típica de las medias es 227.60. Concretamente dicho promedio se encuentra en el intervalo (15.68; 106.1) al 95% de confianza.
- El número medio de pasos que la inteligencia artificial excede, en un laberinto no perfecto al que se le eliminó el 40% de las paredes, en promedio, al número de pasos óptimos, es 15.35, siendo la desviación típica de las medias es 54. Concretamente dicho promedio se encuentra en el intervalo (4.62; 26.1) al 95% de confianza.

Con estos datos podemos ver que: al eliminar más paredes y el laberinto hacerse más sencillo, nuestro modelo es más eficiente, al reducir el número de veces que se aleja del camino óptimo.

4.7 Resultados Laberinto Wilson

Es conocido que cada método de creación de laberintos introduce su propio sesgo a la hora de crearlos, o tienen una manera particular de hacerlos, por esa razón, se decidió enfrentar nuestro modelo a escenarios creados por Wilson. Con el fin de conocer si su comportamiento sigue un patrón relacionado con los laberintos con los que fue entrenado, o si, por el contrario, sigue un esquema general.

Una vez instanciado dicho objeto, se hizo el mismo estudio hecho con anterioridad, para ver si es capaz de resolver dichos laberintos y en cuantos pasos.

Dado un laberinto aleatorio con dimensiones m y n siendo m, n ∈ {5,6,7,8,9}:

- El número medio de pasos que la inteligencia artificial, en un laberinto Wilson, excede, en promedio, al número de pasos óptimos es 180.98, siendo la desviación típica de las medias es 416. Concretamente dicho promedio se encuentra en el intervalo (98.4; 264) al 95% de confianza.

Al ver estos resultados podemos ver que, en los laberintos de tipo Wilson, nuestro modelo creado es más lento a la hora de encontrar la salida, por lo que podemos concluir en que nuestro modelo no sigue un patrón general.

Capítulo 5
Conclusiones y líneas futuras

5.1 Conclusiones

La resolución de laberintos es una tarea muy complicada, esto se debe a la multitud de posibilidades y a la incertidumbre de saber si estas yendo en la dirección correcta, ambas cosas dificultan notablemente las investigaciones en este sector.

Como se puede comprobar en los apartados 4.4 y 4.5 de esta memoria, los resultados generados por nuestro modelo no son lo suficientemente competentes, como era de esperar, si se les compara con los algoritmos usualmente utilizados para la resolución de estos escenarios, sin embargo, tanto los datos recaudados por el modelo, así como su comportamiento son interesantes de analizar.

Tras realizar la tarea de experimentación se puede concluir que la razón principal, por la que los modelos no consiguen terminar de forma satisfactoria los laberintos, es la incapacidad de estos de memorizar el ambiente que les rodea. Esto influye negativamente a la hora de escoger sus caminos, puesto que vuelve a pasar por los caminos que ya una vez recorrió, intentando obtener un resultado distinto. Pero, el problema trascendental, es que las dimensiones de los escenarios escalan de manera exponencial y, aunque a la red LSTM se le asigne una gran cantidad de memoria, en algún momento dicha cantidad, se quedará corta y volverá a cometer los mismos errores que ha estado cometiendo, hasta ahora, el modelo actual.

Este TFG me ha dado la oportunidad de introducirme al área de la Inteligencia Artificial, aprendiendo sus múltiples casos de uso y su funcionamiento, también ha hecho que ponga a prueba todos los conocimientos adquiridos en estos años de carrera, especialmente los de Unity y C#.

5.2 Líneas futuras

Este trabajo ha dejado una infinidad de posibles líneas futuras sobre las que seguir trabajando y experimentando, destacando sobre todo las siguientes líneas:

- Aumentar el rango de búsqueda de una configuración optima, pudiendo desglosar los trabajados en este proyecto en más intervalos, o añadir nuevas dimensiones a la matriz, lo cual aumentaría de manera exponencial el número de posibilidades a testear.
- Aumentar el tamaño de la memoria en la red LSTM, con estos cambios se podrían obtener mejores resultados, pero con un coste de tiempo de cómputo muy elevado.
- Actualizar la versión de *ML-Agents* utilizada en este proyecto y probar nuevas configuraciones con la *curiosity* [32] activada, la cual se utiliza en escenarios donde la cantidad de recompensa recibidas por el agente son raras o escasas
- Utilizar el entrenamiento por imitación [33] de *ML-Agents*, que, con el cual podría imitar el comportamiento de los algoritmos aquí usados para su comparación y ver si tras imitar es capaz de aprender mejor.
- Clasificar la complejidad de los laberintos por el diámetro de su grafo asociado, el camino óptimo.

Capítulo 6
Summary and Conclusions

6.1 Final conclusions

Solving mazes is a very complicated task, this is because to the multitude of possibilities and the uncertainty of knowing if you are going in the right direction, both things make research in this sector significantly more difficult.

As can be seen in sections 4.4 and 4.5 of this report, the results generated by our model are not competent enough as expected, when compared to the algorithms usually used to solve these scenarios, however, all data collected by the model and its behaviour are interesting to analyse.

After doing this experiment, it can be concluded that the main reason, why the models cannot successfully complete the mazes, is their inability to memorize the environment around them. This has a negative influence when choosing your paths, since it goes back through the paths that you once travelled, trying to obtain a different result. But the transcendental problem is that the dimensions of the scenarios scale exponentially and, although the LSTM network is assigned a large amount of memory, at some point that amount will fall short and will make the same mistakes that it has been committing, until now.

This TFG has given me the opportunity to enter the area of Artificial Intelligence, learning its multiple use cases and how it works, it has also put me to the test of all the knowledge acquired in these years of my career, especially that of Unity and C #.

6.2 Future Works

This work has left an infinity of possible future lines on which to continue working and experimenting, highlighting above all the following lines:

- Increase the search range of an optimal configuration, being able to break down those worked on in this project in more intervals, or add new dimensions to the matrix, which would exponentially increase the number of possibilities to test.
- Increase the memory size in the LSTM network, with these changes better results could be obtained, but with a very high computation time cost.
- Update the version of ML-Agents used in this project and test new configurations with curiosity [32] activated, which is used in scenarios where the amount of reward received by the agent is rare or scarce.
- Use ML-Agents' imitation training [33], which, with which you could imitate the behaviour of the algorithms used here for comparison and see if after imitating you are able to learn better.
- Classify the complexity of the mazes by the diameter of its associated graph, the optimal path.

Capítulo 7
Presupuesto

7.1 Coste del proyecto

Para la realización de este proyecto han sido necesarias alrededor de 500 horas de trabajo efectivo, si nos ponemos en el puesto de trabajo de una persona con formación universitaria, ciñéndonos al promedio de 10 €/hora nos daría un coste aproximado de 5000€.

Además, hay que añadir que se hizo un uso aproximado de 380 horas del ordenador para el entrenamiento de los modelos usados y para la realización de las estadísticas, también habría que sumar el coste de los equipos utilizados, también debemos tener en cuenta que todo el trabajo podría haberse desarrollado en un solo ordenador de gama media, que disponga de grafica dedicada, el cual suele costar alrededor de 800 euros.

Número de horas trabajadas	600
Coste por hora	10
Total	6000

Tabla 13: Coste Recursos Humanos

Modelos elección de la configuración	180	Recolección de datos modelo mejorado PPO	4
Mejor modelo de configuración final PPO	65	Recolección de datos modelo LSTM 5X5	2
Mejor modelo de configuración final LSTM	12	Recolección de datos modelo LSTM 9X9	7
Aumento tamaño modelo LSTM	30	Recolección de datos investigación final LSTM	18
Recolección de datos selección de configuración	24	recolección de datos modelo mejorado LSTM 9X9	12
Recolección de datos modelo PPO	4	Recolección de datos investigación final modelos mejorado LSTM	24

Tabla 14: Cálculo de horas de trabajo computacional

Número de horas computando	382
Media de KW por hora	0,332
Coste medio por KW hora	0,1199
Total	15,2061

Tabla 15: Coste de electricidad

Coste se RRHH	6000
Coste de electricidad	15,2
Coste del *hardware*	800
Total	6815,2

Tabla 16: Coste total del proyecto

Capítulo 8
Repositorio del proyecto

El código del proyecto ha sido subido a un repositorio público, junto con los estudios realizados, y con gifs de su comportamiento.

- https://github.com/alu0100981506/TFG-Maze

Bibliografía

[1] “Se lanza “Tennis for two”, considera el primer videojuego de la historia - History”. [Online]. Available: https://latam.historyplay.tv/hoy-en-la-historia/se-lanza-tennis-two-considerado-el-primer-videojuego-de-la-historia. [Accessed: 22-June-2020]

[2] ”Microsoft’s E3 2019 Presser Was the Most Watched Conference Stream for a Third Straight Year - Twinfinite”. [Online]. Available: https://twinfinite.net/2019/06/microsofts-e3-2019-presser-was-the-most-watched-conference-stream-for-a-third-straight-year/. [Accessed: 22-June-2020]

[3] ”Experimental Self-Learning AI in Battlefield 1 - Youtube”. [Online]. Available: https://youtu.be/ZZsSx6kAi6Y. [Accessed 23-June-2020]

[4] ”Emergent Tool Use from Multi-Agent Interaction - OpenAI”. [Online]. Available: https://openai.com/blog/emergent-tool-use/. [Accessed: 23-June-2020]

[5] “Games by Angelina - Games by Aangelina”. [Online]. Available: http://www.gamesbyangelina.org/. [Accessed: 23-June-2020]

[6] ”Sex, Lies and Rape – Games By Angelina”. [Online]. Available: http://www.gamesbyangelina.org/2012/05/new-game-sex-lies-and-rape/.

[Accessed: 30-June-2020]

[7] ”Laberinto – Wikipedia, la enciclopedia libre”. [Online]. Available: https://es.wikipedia.org/wiki/Laberinto. [Accedes: 22-June-2020]

[8] ”Cómo salir de un laberinto. Método de la mano derecha – Ingeniería Basica”. [Online]. Available: https://ingenieriabasica.es/como-salir-de-un-laberinto/#:~:text=Laberinto%20de%20prueba-,El%20m%C3%A9todo%20de%20la%20mano%20derecha,que%20lleguemos%20a%20la%20salida. [Accessed: 30-June-2020]

[9] ”Dijkstra's algorithm - Wikipedia, the free encyclopedia”. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. [Accessed: 30-June-2020]

[10] ”A* search algorithm- Wikipedia, the free encyclopedia”. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm#History. [Accessed: 30-June-2020]

[11] ” Maze Generation Algorithms - An Exploratio”. [Online]. Available: https://professor-l.github.io/mazes/. [Accessed: 30-June-2020]

[12] ”Procesos de decisión de Markov y aprendizaje por refuerzo- Libro IA”. [Online]. Available: http://ccc.inaoep.mx/~emorales/Papers/2018/2018ESucar.pdf. [Accessed: 30-June-2020]

[13] ”AlphaGo - DeepMind”. [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far. [Accessed: 30-June-2020]

[14] ”AlphaStar - DeepMind”. [Online]. Available: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii. [Accessed: 30-June-2020]

[15] ”Go (game) - Wikipedia, the free encyclopedia”. [Online]. Available: https://en.wikipedia.org/wiki/Go_(game). [Accessed: 30-June-2020]

[16] ”Monte Carlo tree search - Wikipedia, the free encyclopedia”. [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. [Accessed: 30-June-2020]

[17] ”Proximal Policy Optimization Algorithm”. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf. [Accessed: 30-June-2020]

[18] ”Unity”. [Online]. Available: https://unity.com/. [Accessed: 30-June-2020]

[19] ”ML-Agents - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents. [Accessed: 28-June-2020]

[20] ”Creating a maze generator - Creating a Closed Grid - Unity 3D - Youtube”. [Online]. Available: https://youtu.be/OzENv_ZRA1g. [Accessed: 30-June-2020]

[21] ”Environment Design Best Practices - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/0.13.0/docs/Learning-Environment-Best-Practices.md#rewards. [Accessed: 30-June-2020]

[22] ”Training Config File - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/0.13.0/docs/Training-ML-Agents.md#training-config-file. [Accessed: 30-June-2020]

[23] ”Training with Proximal Policy Optimization - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/0.13.0/docs/Training-PPO.md#training-with-proximal-policy-optimization. [Accessed: 30-June-2020]

[24] ”Training with Curriculum Learning - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/0.13.0/docs/Training-Curriculum-Learning.md#training-with-curriculum-learning. [Accessed: 30-June-2020]

[25] ”Memory-enhanced agents using Recurrent Neural Networks - GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/0.13.0/docs/Feature-Memory.md#memory-enhanced-agents-using-recurrent-neural-networks. [Accessed: 30-June-2020]

[26] ”Comportamiento PPO- GitHub”. [Online]. Available: https://github.com/alu0100981506/TFG-Maze/tree/master#ppo. [Accessed: 30-June-2020]

[27] ”Comportamiento LSTM- GitHub”. [Online]. Available: https://github.com/alu0100981506/TFG-Maze/tree/master#lstm. [Accessed: 30-June-2020]

[28] ”One-sample t-test — jamovi - YouTube”. [Online]. Available: https://youtu.be/DrBT4ezYIL8. [Accessed: 2-September-2020]

[29] ”Estudio General TFG Todos los datos 3.07E6- GitHub”. [Online]. Available: https://github.com/alu0100981506/TFG-Maze/blob/master/Estudios/Estudio%20General%20TFG%20Todos%20los%20datos%203.07E6.xls. [Accessed: 30-June-2020]

[30] ”Estudio General TFG 9x9 Max 1.224E7- GitHub”. [Online]. Available: https://github.com/alu0100981506/TFG-Maze/blob/master/Estudios/Estudio%20General%20TFG%209x9%20Max%201.224E7.xls. [Accessed: 30-June-2020]

[31] ”Estudio General TFG Todos los datos 1.224E7.xls- GitHub”. [Online]. Available: https://github.com/alu0100981506/TFG-Maze/blob/master/Estudios/Estudio%20General%20TFG%20Todos%20los%20datos%201.224E7.xls. [Accessed: 30-June-2020]

[32] ”Curiosity for Sparse-reward Environments-Git Hub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/release_3_docs/docs/ML-Agents-Overview.md#curiosity-for-sparse-reward-environments. [Accessed: 30-June-2020]

[33] ”Imitation Learning- GitHub”. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/release_3_docs/docs/ML-Agents-Overview.md#imitation-learning. [Accessed: 30-June-2020]