

GENERACIÓN Y RESOLUCIÓN DE PUZLES AUTOMÁTICA COMO  
APOYO A DISEÑADORES DE VIDEOJUEGOS  
AUTOMATIC PUZZLE GENERATION AND SOLVING AS A SUPPORT  
FOR GAME DESIGNERS



TRABAJO FIN DE GRADO  
CURSO 2023-2024

AUTOR  
Esther Babon Arcauz

DIRECTOR  
Ismael Sagredo Olivenza

GRADO EN INGENIERÍA INFORMÁTICA  
FACULTAD DE INFORMÁTICA  
UNIVERSIDAD COMPLUTENSE DE MADRID

GENERACIÓN Y RESOLUCIÓN DE PUZLES AUTOMÁTICA COMO  
APOYO A DISEÑADORES DE VIDEOJUEGOS  
AUTOMATIC PUZZLE GENERATION AND SOLVING AS A SUPPORT  
FOR GAME DESIGNERS

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

AUTOR

ESTHER BABON ARCAUZ

DIRECTOR

ISMAEL SAGREDO OLIVENZA

**CONVOCATORIA: SEPTIEMBRE 2024**

GRADO EN INGENIERÍA INFORMÁTICA  
FACULTAD DE INFORMÁTICA  
UNIVERSIDAD COMPLUTENSE DE MADRID

13 DE SEPTIEMBRE DE 2024



## DEDICATORIA

A mi amona



## **AGRADECIMIENTOS**

A Guille, Óscar y todos los que han hecho que este camino sea más llevadero.



# RESUMEN

## GENERACIÓN Y RESOLUCIÓN DE PUZLES AUTOMÁTICA COMO APOYO A DISEÑADORES DE VIDEOJUEGOS

En este Trabajo de Fin de Grado modelamos, resolvemos y generamos cinco tipos de rompecabezas de la conocida saga de videojuegos el Profesor Layton. Para ello utilizamos diferentes técnicas de Inteligencia Artificial, como la búsqueda heurística A\*, la búsqueda metaheurística de optimización Enfriamiento Simulado y los algoritmos genéticos. Además incorporamos una interfaz gráfica para que el usuario pueda recibir pistas para llegar a la solución del puzle en tiempo real y para que pueda generar los puzles con diferentes dificultades. Finalmente comprobamos que los algoritmos utilizados son óptimos mediante graficas de medidas de tiempos de ejecución y validez de las soluciones generadas.

### Palabras clave

Puzle, IA, Modelar, Resolver, Generar, Heurística, Búsqueda, A\*, Genéticos, Algoritmos





# **ABSTRACT**

## **AUTOMATIC PUZZLE GENERATION AND SOLVING AS A SUPPORT FOR GAME DESIGNERS**

In this Final Degree Project we model, solve, and generate five types of puzzles from the well-known video game saga Professor Layton. In order to achieve this, we use different Artificial Intelligence techniques, such as heuristic search A\*, metaheuristic search optimization Simulated Annealing and genetic algorithms. We also incorporate a graphical interface so that the user can receive hints to reach the solution of the puzzle in real time and can generate puzzles with different difficulties. Finally, we verify that the algorithms used are optimal by means of graphical measures of execution times and validity of the generated solutions.

### **Keywords**

Puzzle, AI, Model, Resolve, Generate, Heuristic, Search, A\*, Genetic, Algorithms





# ÍNDICE DE CONTENIDOS

Capítulo 1 - Introducción .....	1
1.1 Motivación .....	1
1.2 Objetivos.....	2
1.3 Plan de trabajo .....	3
Capítulo 2 - Estado de la cuestión .....	5
2.1 Algoritmos de generación de puzles .....	5
2.1.1 Algoritmos basados en grafos .....	5
2.1.2 Algoritmo de Tutte [11] .....	7
2.1.3 Algoritmo Answer Set Programming with Propositional Schema [13].....	7
2.2 Algoritmos de resolución de puzles.....	8
2.2.1 A* con Manhattan [15].....	8
2.2.2 A* con Chebyshev [18].....	9
2.2.3 A* acotado por tiempo e inyección.....	9
2.2.4 Redes neuronales para aprender a combinar heurísticas.....	10
2.3 Justificación de la investigación .....	11
2.4 Tecnologías aplicadas .....	11
2.4.1 Búsqueda A* [23] .....	11
2.4.2 Búsqueda Simulated Annealing [24] .....	12
2.4.3 Algoritmos genéticos [25] .....	13
Capítulo 3 - Herramientas .....	15
3.1 AIMA [26] .....	15
3.2 Numpy [28] .....	15

3.3 Tkinter [29].....	16
3.4 Time [30] .....	16
3.5 Matplotlib [31] .....	17
Capítulo 4 - Modelado de problemas.....	19
4.1 Puzle 1: Jarras (Water Pitchers) .....	19
4.1.1 Algoritmo utilizado .....	20
4.1.2 Modelado .....	20
4.1.3 Generación .....	22
4.1.4 Tiempos .....	22
4.1.5 Interfaz gráfica .....	24
4.2 Puzle 2: 8-Puzzle y 16-Puzzle (A Worms Dream) .....	27
4.2.1 Algoritmo utilizado .....	27
4.2.2 Modelado .....	27
4.2.3 Generación .....	29
4.2.4 Tiempos .....	29
4.2.5 Interfaz grafica .....	31
4.3 Puzle 4: Flip panels: Stomp on it! .....	34
4.3.1 Algoritmo utilizado .....	34
4.3.2 Modelado .....	34
4.3.3 Generación .....	35
4.3.4 Tiempos .....	36
4.3.5 Interfaz gráfica .....	37
4.4 Puzle 3: Sliding: Get the ball out! .....	40
4.4.1 Algoritmo utilizado .....	40
4.4.2 Modelado .....	41

4.4.3 Generación .....	42
4.4.4 Tiempos .....	43
4.4.5 Interfaz gráfica .....	44
4.5 Puzle 5: Hatgram.....	48
4.5.1 Algoritmo utilizado .....	48
4.5.2 Modelado .....	49
4.5.3 Parámetros.....	53
4.5.4 Tiempos .....	61
4.5.5 Interfaz gráfica .....	61
Capítulo 5 - Conclusión y trabajo futuro .....	64
Introduction.....	66
Conclusions and future work .....	71
Bibliografía.....	73

## ÍNDICE DE FIGURAS

Figura 1: Water Pitchers [43] .....	19
Figura 2: Tiempo generación Jarras.....	23
Figura 3: Tiempo resolución Jarras .....	23
Figura 4: GUI menú Jarras .....	24
Figura 5: GUI resolver Jarras .....	24
Figura 6: GUI resolver Jarras pista .....	25
Figura 7: GUI generar Jarras .....	26
Figura 8: A worm's Dream [33] .....	27
Figura 9: Tiempo generación 8/16-puzle .....	30
Figura 10: Tiempo resolución 8/16-puzle.....	30
Figura 11: GUI menú 8/16-Puzle.....	31
Figura 12: GUI resolver 8/16-Puzle.....	31
Figura 13: GUI generar 8-Puzle.....	32
Figura 14: GUI generar 16-Puzle.....	33
Figura 15: Stomp on it! [34].....	34
Figura 16: Tiempo generar Stomp .....	36
Figura 17: Tiempo resolver Stomp.....	37
Figura 18: GUI menú Stomp .....	37
Figura 19: GUI resolver Stomp .....	38
Figura 20: GUI generación Stomp 1 .....	39
Figura 21: GUI generación Stomp 2 .....	39
Figura 22: Get the Ball Out! [35] .....	40



Figura 23: Tiempo generación Pelota.....	43
Figura 24: Tiempo resolución Pelota .....	44
Figura 25: GUI menú Pelota .....	44
Figura 26: GUI resolver Pelota 1 .....	45
Figura 27: GUI resolver Pelota 2 .....	46
Figura 28: GUI generar Pelota 1 .....	46
Figura 29: GUI generar Pelota 2.....	47
Figura 30: Hatgram [37] .....	48
Figura 31: Fitness Número de generaciones .....	53
Figura 32: Tiempo ejecución número de generaciones .....	54
Figura 33: Fitness por tamaño de población.....	55
Figura 34: Tiempo por tamaño de población.....	55
Figura 35: Fitness por tamaño de torneo .....	56
Figura 36: Tiempo por tamaño de torneo .....	57
Figura 37: Fitness por probabilidad de cruce.....	58
Figura 38: Tiempo por probabilidad de cruce .....	58
Figura 39: Fitness por probabilidad de mutación .....	59
Figura 40: Tiempo por probabilidad de mutación .....	60
Figura 41: Fitness solución 100 Hatgram.....	60
Figura 42: GUI menú Hatgram .....	61
Figura 43:GUI resolver Hatgram.....	62
Figura 44: GUI resolver Hatgram piezas .....	63

## Índice de tablas

Tabla 1: Características PC.....	16
Tabla 2: Dificultades generación Jarras .....	22
Tabla 3: Dificultades generación puzzle8/16.....	29
Tabla 4: Dificultades generación Stomp .....	36
Tabla 5: Dificultades generación Pelota .....	43

# Capítulo 1 - Introducción

## 1.1 Motivación

El campo de la Inteligencia Artificial ha ganado mucha atención en los últimos años mayoritariamente debido a la popularización y accesibilidad pública de sistemas de chat basados en Inteligencia Artificial como *ChatGPT* o generadores de contenido multimedia como *Stable Diffusion* o *Sora*. A medida que a la población se le ha facilitado el uso de este tipo de herramientas, se ha interesado más por cómo funcionan y a que otros sectores se pueden aplicar.

Sin embargo, existen múltiples aplicaciones donde el uso de la IA simbólica clásica tiene cabida. Por ejemplo, ChatGPT tiene problemas para realizar cálculos matemáticos complejos [1], y además es poco eficiente a la hora resolver de forma óptima puzzles como el 8-Puzzle o el cubo de Rubik. Concretamente, en un estudio reciente [2] configuraron LLMs para la resolución de puzzles, estas LLMs especializadas, conseguían una tasa de acierto del 93.2% en el 8-puzzle, cuando A\* puede resolver este tipo de puzzle en poco tiempo y con una tasa de acierto del 100%. A pesar de los avances conseguidos con la utilización de los LLMs, en este tipo de problemas sigue siendo más conveniente recurrir a la IA simbólica clásica.

En este Trabajo de Fin de Grado queremos resolver y generar puzzles o problemas de la conocida saga de juegos El Profesor Layton [3], mediante el uso de algoritmos de búsqueda heurística.

La saga de videojuegos del Profesor Layton se centra en resolver misterios, para avanzar en la historia debemos resolver una multitud de puzzles de todo tipo y de diferentes dificultades. Todo ello amenizado por una historia que da consistencia a la sucesión de puzzles que vamos resolviendo.

La motivación tras este TFG es permitir a los diseñadores de juegos de este tipo, disponer de un asistente que en el caso de que los jugadores se queden atascados, les ayude con algún tipo de pista. La ayuda puede ser proporcionada mediante un botón que, al

ser pulsado, proporciona al usuario el siguiente paso que debe hacer para llegar a la solución. Para puzles más complejos, podríamos requerir de otro tipo de ayudas como veremos más adelante.

Desde el punto de vista de diseñadores de videojuegos, queremos que puedan generar puzles de manera automática para que puedan usar sus capacidades en otros aspectos del diseño de videojuegos. Si deciden generar los puzles ellos mismos, podrán comprobar si son resolubles mediante esta herramienta.

La generación y resolución de rompecabezas es un tema de interés popular, y se ve reflejado en el número de artículos publicados en los últimos años. Mencionamos y discutimos estos artículos en el siguiente capítulo.

## 1.2 Objetivos

El objetivo principal de este trabajo de fin de grado es resolver y generar ciertos tipos de puzles originarios de la saga de videojuegos *El Profesor Layton* [3], mediante técnicas de Inteligencia Artificial en un tiempo suficientemente bueno como para poder dar pistas o soluciones parciales al usuario en tiempo real.

Para poder cumplir el objetivo principal, hemos tenido que cumplir los siguientes subobjetivos:

- Modelar algunos tipos de puzles de ejemplo.
- Para cada tipo de puzle, determinar qué algoritmo de búsqueda es el más óptimo para resolverlo.
- Determinar los parámetros ideales de los diferentes modelos para que el tiempo de ejecución sea el mínimo y la solución sea óptima.
- Poder visualizar los problemas y soluciones mediante una interfaz gráfica.
- Generar los diferentes tipos de puzle con variedad de dificultades.

## **1.3 Plan de trabajo**

### **Octubre 2023:**

- Investigar la existencia de algoritmos de generación de puzles.

### **Noviembre 2023:**

- Investigar los diferentes tipos de algoritmos de resolución basados en Inteligencia Artificial.

### **Diciembre 2023:**

- Modelar y resolver algunos rompecabezas mediante búsqueda heurística.

### **Enero 2024:**

- Investigar otros tipos de búsqueda para los problemas que no se pueden resolver de esta manera.

### **Febrero 2024:**

- Aplicar algoritmos genéticos para la resolución de uno de los rompecabezas.

### **Marzo 2024:**

- Terminar de modelar y resolver todos los rompecabezas

### **Abril 2024:**

- Generar mediante la aleatoriedad los diferentes puzles.

**Mayo 2024:**

- Empezar a generar una interfaz gráfica.
- Comenzar a redactar la memoria.

**Junio 2024:**

- Testear y arreglar la resolución y la generación de los diferentes modelos.

**Julio 2024:**

- Investigar y encontrar los valores ideales de los parámetros del algoritmo genético

**Agosto 2024:**

- Terminar la interfaz grafica
- Sacar graficas de tiempo de generación y resolución de los diferentes rompecabezas

## Capítulo 2 - Estado de la cuestión

La generación y resolución automática de puzzles o problemas ha ganado considerable atención tanto en la industria del videojuego como en la informática. Tal y como se declara en el artículo *Automatic Generation and Analysis of Physics-Based Puzzle Games* [4], esto es debido a la ventaja que proporciona en la rapidez de generación de este tipo de contenido. Además de rebajar los costes de producción, introduce una gran cantidad de variaciones en los juegos.

La capacidad de generar y resolver puzzles de forma automática permite la creación de contenido adaptativo que puede ajustarse en tiempo real a las habilidades del jugador, proporcionando una experiencia de juego personalizada.

En este capítulo revisamos la literatura actual sobre los algoritmos y técnicas utilizados tanto en la generación como en la resolución de puzzles.

### 2.1 Algoritmos de generación de puzzles

#### 2.1.1 Algoritmos basados en grafos

En el artículo: *Automated maze generation and human interaction* [5], Foltin consigue generar laberintos utilizando algoritmos basados en grafos. Un laberinto es un pasatiempo que consiste en llegar de un punto origen a un punto destino, su dificultad reside en que contienen caminos sin salida y generalmente, un solo recorrido correcto.

Los algoritmos basados en grafos son aquellos que tienen como tipo de datos un grafo. Un grafo está definido por un conjunto de aristas, un conjunto de vértices y una matriz de adyacencia. La matriz de adyacencia define las conexiones entre los vértices mediante las aristas.

Utiliza los siguientes algoritmos para generar laberintos:

- **Algoritmo de Prim** [6]: comienza desde un nodo arbitrario y se expande seleccionando las aristas de menor peso que conectan nodos incluidos con nodos no incluidos. De esta manera encuentra el árbol de expansión mínima en un grafo conectado y no dirigido. Un árbol de expansión mínima, conocido como Minimum Spanning Tree [7], es un subconjunto de un grafo que conecta todos los nodos con el menor costo posible.
- **Algoritmo de Kruskal** [8]: comienza ordenando todas las aristas por peso y luego, siempre que no formen un ciclo, las va añadiendo al árbol. De esta manera encuentra el árbol de expansión mínima.
- **Backtracking** [9]: es una técnica de resolución de problemas que construye soluciones incrementales y retrocede cuando se determina que la solución parcial no puede llevar a una solución completa válida, explorando todas las posibles opciones.
- **Hunt and Kill** [10]: Es un algoritmo utilizado en la generación de laberintos. Funciona alternando entre dos fases: "cazar", donde busca una celda no visitada, y "matar", donde extiende un camino aleatoriamente desde esa celda hasta que ya no puede continuar.
- **Bacterial Growth** [5]: este algoritmo simula el crecimiento de bacterias, expandiéndose desde un punto inicial y replicándose hacia áreas adyacentes, a menudo usado en la generación de laberintos o patrones orgánicos.

El análisis es muy exhaustivo, ya que analiza los tiempos que tarda en generar los laberintos, el tiempo que tarda el usuario a llegar a la solución, el tiempo de la solución óptima, el camino a la mejor solución, el número de movimientos del usuario y el número mínimo de movimientos para llegar a la solución. Cabe señalar la falta de estudio sobre la dificultad de los laberintos generados.



### 2.1.2 Algoritmo de Tutte [11]

En el artículo *Puzzle generators and symmetric puzzle layout* [12], logran producir de manera automática puzles cíclicos. Un problema es cíclico cuando utilizamos un mismo conjunto de acciones que aplicamos a cada estado para llegar a la solución, que es a su vez el estado inicial. Un ejemplo de un problema cíclico es el conocido cubo de Rubik.

El algoritmo utilizado para la generación es el algoritmo de Tutte [11]. Este algoritmo basado en principios geométricos garantiza que un grafo se pueda dibujar de manera que todas sus aristas se representan como líneas rectas que no se cruzan, manteniendo así la simetría y la claridad del diseño del puzle.

La investigación incluye una implementación y explora cómo se pueden crear y modificar los niveles de dificultad de los problemas.

### 2.1.3 Algoritmo Answer Set Programming with Propositional Schema [13]

El artículo *Generating celular puzzles with logic programs* [14] se centra en la producción automática de problemas celulares. Los problemas celulares son aquellos en los que los valores de las celdas están restringidos por normas que involucran grupos de celdas. Un ejemplo popular de este tipo de puzles es el Sudoku.

El algoritmo utilizado para generar problemas celulares es el de Answer Set Programming with Propositional Schema [13], este se basa en definir problemas a través de reglas lógicas y encontrar respuestas validas que satisfagan las reglas, mediante el planteamiento declarativo. Mediante este algoritmo se expresan las restricciones del puzle y se generan soluciones completas de forma aleatoria. Después, teniendo la solución, se utiliza el algoritmo de reducción de pistas para determinar el número mínimo de pistas necesarias para que el problema sea resoluble y único. Siendo las pistas las celdas con valores predefinidos.

La dificultad de los problemas generados es controlada mediante la manipulación de las pistas.

## 2.2 Algoritmos de resolución de puzles

### 2.2.1 A\* con Manhattan [15]

En el artículo *A comparative study of the A\* heuristic search algorithm used to solve efficiently a puzzle game* [16], se busca la resolución óptima de un problema en el que se deben mover poliedros en un tablero para alcanzar un estado objetivo a partir de un estado inicial. La dificultad del problema reside en el tamaño del espacio de búsqueda, por lo que es necesario el uso de la búsqueda heurística para encontrar la solución óptima, la que conlleva el menor número de movimientos.

Se comparan dos funciones heurísticas: la heurística de Hamming [17] y la de Manhattan. Las funciones heurísticas estiman como de cerca está el estado actual del estado objetivo. La heurística de Hamming calcula la cantidad de piezas que están fuera de su posición respecto al estado objetivo. Por otro lado, la heurística de Manhattan suma las distancias Manhattan de cada pieza desde su posición actual hasta su posición objetivo. La distancia Manhattan es la distancia en línea recta a lo largo de los ejes horizontales y verticales. Se llega a la conclusión de que la heurística Manhattan resulta ser más informada y eficiente en la reducción del espacio de búsqueda.

El algoritmo de resolución que utiliza es el algoritmo de búsqueda A\*. Esta combina la búsqueda en amplitud y la búsqueda voraz utilizando una función de evaluación  $f(S) = g(S) + h(S)$ , donde  $g(S)$  es el coste del camino desde el nodo inicial hasta el nodo actual y,  $h(S)$  es la estimación del costo restante hasta el objetivo utilizando las funciones heurísticas mencionadas.

### 2.2.2 A\* con Chebyshev [18].

En el artículo *A comparative study of three heuristic functions used to solve the 8-puzzle* [19], modela y resuelve el Puzzle-8, un problema que consiste en un tablero 3x3 y fichas numeradas del 1 al 8 con un espacio vacío. Consiste en ordenar las fichas mediante los movimientos del espacio vacío.

Mencionan el uso de tres heurísticas. La Heurística de Hamming, que se determina mediante el número de fichas mal colocadas. La heurística Manhattan, que se define por la suma de las distancias Manhattan de cada ficha desde su posición actual al objetivo. Y finalmente, la heurística de Chebyshev, que es la razón de estudio de este artículo. Esta se calcula como el doble de la distancia Chebyshev [18]. La distancia Chebyshev es la máxima distancia entre los ejes horizontal y vertical de una ficha desde la posición actual al objetivo. En este artículo se llega a la conclusión de que la Heurística Chebyshev es la más informada y supera a las otras dos tanto en complejidad espacial como en temporal.

### 2.2.3 A\* acotado por tiempo e inyección

En el artículo *Solving general game playing puzzles using heuristic search* [20] consigue modelar y resolver varios rompecabezas, entre ellos el Puzzle-8. La heurística aplicada es el número de pasos necesarios para alcanzar el objetivo, ignorando algunas restricciones y penalizar subobjetivos que no se han alcanzado en el estado actual.

El algoritmo que se aplica es la búsqueda A\* acotado por tiempo e inyección, que es una variación del algoritmo A\*. Este se diferencia de la búsqueda A\* porque interrumpe la búsqueda tras un límite de tiempo. Además, mantiene y actualiza dos caminos, el actual y el potencial, y utiliza inyecciones heurísticas para favorecer la exploración de buenos caminos descubiertos. También, reinicia las listas de búsqueda cuando es necesario.

Finalmente, el autor concluye diciendo que la búsqueda A\* acotado por tiempo e inyección tiene buen rendimiento al resolver rompecabezas en escenarios de tiempo real en varios puzzles, llegando a superar la búsqueda en árbol de Monte Carlo [21].

#### 2.2.4 Redes neuronales para aprender a combinar heurísticas

En el artículo *Learning from Multiple Heuristics* [22] buscan la manera de mejorar el proceso de búsqueda combinando varias heurísticas a través de una red neuronal.

Los rompecabezas que buscan resolver son los del puzle 15 y 25 y la Torre de Hanoi con 4 postes. El puzle 15 y 25 son variaciones del 8-Puzzle, pero con 15 y 25 piezas en vez de 8. El problema de la Torre de Hanoi con 4 postes consiste en mover las fichas perforadas de un poste a otro, cumpliendo que una ficha de mayor tamaño no puede estar sobre una ficha más pequeña y que solo se puede desplazar la ficha que se encuentre en la cima de cada poste.

Para estos problemas definen un conjunto de heurísticas basadas en la distancia Manhattan y en las bases de datos de patrones. Las Bases de datos de patrones son heurísticas donde se almacenan las soluciones óptimas para reducciones del problema original. Por ejemplo, en el puzle 15 se particiona el tablero en dos y se calculan las soluciones de las dos partes.

Construyen un conjunto de datos de entrenamiento de diferentes tipos de rompecabezas y calculan los valores de las diferentes heurísticas para varios estados de los problemas comparándolas con las soluciones óptimas. De esta manera, la red neuronal se entrena para predecir el costo óptimo de un problema específico a partir de múltiples heurísticas.

Para la búsqueda utilizan  $A^*$ , donde el valor heurístico es calculado utilizando la red neuronal, que devuelve una estimación del costo de la solución.

Finalmente, llegan a la conclusión de que, de esta manera se logra reducir el número de nodos generados en la búsqueda sin sacrificar la calidad de la solución, superando el método de búsqueda  $A^*$  tradicional. Por otro lado, el resultado producido por la red neuronal no garantiza la admisibilidad porque puede sobreestimar el costo real de la solución.

## 2.3 Justificación de la investigación

En los apartados anteriores nos hemos encontrado con que hay lagunas en la generación de cierto tipo de puzles y en las funciones heurísticas a aplicar para puzles diferentes a los mencionados. La investigación realizada en este trabajo aborda estas lagunas. Intentaremos usar algoritmos diferentes para generar los tipos de problemas que no tienen cabida en los ya investigados, a la par que determinamos funciones heurísticas admisibles aplicables a otros tipos de problemas resolubles con el algoritmo de búsqueda local heurística y buscamos otros enfoques para problemas que no acepten las configuraciones de estos algoritmos.

## 2.4 Tecnologías aplicadas

### 2.4.1 Búsqueda A\* [23]

La búsqueda A\* es un tipo de búsqueda heurística. Los algoritmos de búsqueda heurística guían la búsqueda aplicando conocimiento del dominio sobre la proximidad de cada estado al estado objetivo. Las heurísticas proporcionan recomendaciones sobre la elección del sucesor más prometedor de un estado, asociando a cada estado un valor numérico que evalúa lo prometedor que es el estado de alcanzar el estado objetivo. Se estima la distancia de un estado a un estado objetivo.

La búsqueda A\*, además de considerar la heurística, considera también el coste del camino desde el nodo inicial al nodo actual. Representa una estimación del coste total del mejor camino desde el estado inicial al estado objetivo pasando por un nodo n.

$$f'(n) = g(n) + h'(n)$$

Combina la búsqueda primero en anchura con primero en profundidad, cambiando el camino de la solución cada vez que haya nodos más prometedores. Si la heurística aplicada es admisible, la búsqueda A\* es óptima. A\* es óptimamente eficiente ya que ningún otro algoritmo óptimo garantiza expandir menos nodos que A\*. La complejidad de A\* sigue siendo exponencial en el caso peor.

#### **2.4.1.1 Ventajas**

Las principales ventajas de A\* son la garantía de encontrar el camino más corto, siempre que la heurística aplicada sea admisible, y la posibilidad de utilizar diferentes heurísticas según el problema.

#### **2.4.1.2 Desventajas**

Las principales desventajas de A\* son el alto consumo de memoria, ya que almacena todos los nodos generados, su dependencia a la heurística aplicada, ya que, si esta no es exacta, pierde eficiencia.

### **2.4.2 Búsqueda Simulated Annealing [24]**

El algoritmo de enfriamiento simulado se utiliza comúnmente con problemas con espacios de búsqueda grandes y complejos. Comienza con una solución inicial aleatoria y la variable temperatura inicializada a un valor alto. En cada iteración se genera una solución vecina un poco diferente a la actual. Si esta nueva solución es mejor se acepta, pero si es peor depende de la variable de temperatura aceptarla o no. Si la temperatura es alta, la probabilidad de aceptarla es mayor. La variable de temperatura va perdiendo valor mediante la función de enfriamiento. Finalmente, el algoritmo termina cuando la temperatura llega a un valor mínimo o si no se encuentra una solución mejor en un número dado de interacciones.

De esta manera el algoritmo de enfriamiento simulado consigue escapar de los mínimos locales y explorar todo el espacio de búsqueda.

#### **2.4.2.1 Ventajas**

Las principales ventajas de Simulated Annealing son la posibilidad de escapar de mínimos locales aceptando soluciones peores temporalmente y la flexibilidad de aplicación a una gran variedad de problemas.

#### **2.4.2.2 Desventajas**

Las principales desventajas son que requiere de la elección del valor de la tasa de enfriamiento y no garantiza encontrar el óptimo global.

#### **2.4.3 Algoritmos genéticos [25]**

Los algoritmos genéticos son parte de los algoritmos de búsqueda local. La búsqueda es un proceso que dada la solución actual en la que se encuentra, selecciona iterativamente una solución de su entorno para continuar la búsqueda.

En los algoritmos genéticos se parte de una población inicial de soluciones candidatas de la cual se seleccionan los individuos más adaptados o capacitados para luego reproducirlos y mutarlos para finalmente obtener la siguiente generación de individuos que estarán más adaptados que la anterior generación. Se genera generación hasta un número máximo de generaciones marcado o hasta encontrar la solución ideal.

Este algoritmo lo utilizamos para la resolución de uno de los rompecabezas, mencionamos más adelante los funcionamientos y definiciones de los diferentes parámetros y funciones auxiliares necesarias.

##### **2.4.3.1 Ventajas**

Las ventajas de los algoritmos genéticos son la capacidad de encontrar soluciones para problemas complejos sin necesidad de un conocimiento extenso y la adaptación de las soluciones a lo largo del tiempo mediante los operadores genéticos.

##### **2.4.3.2 Desventajas**

La principal desventaja de los algoritmos genéticos es que requieren de ajustar múltiples parámetros para el funcionamiento ideal del algoritmo y no garantizan encontrar el óptimo global.





## Capítulo 3 - Herramientas

En este capítulo enumeramos las herramientas que han sido indispensables para la realización de este trabajo. Las herramientas que hemos utilizado son librerías de código abierto de Python. El código de este proyecto ha sido realizado utilizando la versión 3.8.10 de Python, puede que utilizando versiones superiores no cumpla con las funcionalidades aquí descritas.

### 3.1 AIMA [26]

El código de la librería AIMA está basado en el libro de Artificial Intelligence: A Modern Approach [27]

AIMA contiene las funcionalidades necesarias para poder modelar un problema y resolverlo utilizando diferentes algoritmos de Inteligencia Artificial, entre otras muchas. En nuestro caso, lo hemos utilizado para modelar problemas y resolverlos utilizando en la gran mayoría de casos, el algoritmo A\*. Para modelar los problemas hemos utilizado la clase Problem del módulo search.

### 3.2 Numpy [28]

Numpy es una librería de Python utilizada para la computación científica. Proporciona un tipo de array multidimensional llamado *ndarray*. Este tipo de datos facilita la manipulación de grandes volúmenes de datos numéricos. Incluye operaciones vectorizadas y funciones matemáticas y estadísticas.

Hemos utilizado la biblioteca Numpy para la manipulación de datos numéricos.

### 3.3 Tkinter [29]

Tkinter es una librería de Python utilizada para generar interfaces gráficas. Tkinter [29] permite crear todos los elementos gráficos necesarios para generar una interfaz gráfica accesible.

Hemos utilizado Tkinter para generar la interfaz gráfica del proyecto. Contiene menús para seleccionar que tipo de puzzle queremos resolver o generar, permite al usuario introducir los datos de los estados iniciales y objetivos de los problemas de una manera interactiva y permite una visualización de los problemas generados automáticamente más amigable con el usuario. Asimismo, empaqueta las excepciones que se puedan generar en el código.

### 3.4 Time [30]

Time es un módulo de Python que proporciona funciones correspondientes al tiempo.

Hemos utilizado la librería time para medir los tiempos de ejecución de los diferentes algoritmos de generación y resolución que hemos definido.

Las medidas de tiempo se han tomado en un ordenador de sobremesa convencional con las siguientes características (Ver Tabla 1):

Procesador	AMD Ryzen 7 2700X Eight-Core 3.7GHz
RAM	16 GB
Tarjeta gráfica	NVIDIA GeForce GTX 1050 Ti
Sistema Operativo	Windows 10 Pro 64bits

Tabla 1: Características PC

### **3.5 Matplotlib [31]**

Matplotlib es una librería de Python que proporciona las herramientas de visualización de datos para el ecosistema científico de Python. Esta herramienta facilita la exploración de datos interactiva, produce resultados adecuados para publicaciones, proporciona una interfaz gráfica simple, facilita los diagramas habituales y posibilita visualizaciones complejas.

Hemos utilizado esta herramienta para visualizar los tiempos de generación y resolución de los diferentes modelos, entre otras.



## Capítulo 4 - Modelado de problemas

De todos los problemas presentes en el Doctor Layton, hemos seleccionado algunos de ellos, en orden creciente de complejidad, para probar nuestros algoritmos. A continuación, enumeramos los diferentes puzzles implementados y que algoritmos y heurísticas hemos utilizado para resolverlos.

### 4.1 Puzle 1: Jarras (Water Pitchers)

El puzle de las jarras consiste en que hay tres jarras de las que sabemos la capacidad máxima de líquido que pueden contener. Se nos da un estado inicial y un estado objetivo, en los que se determina la cantidad de líquido que tiene cada jarra. El juego consiste en llegar del estado inicial al estado objetivo mediante las acciones de mover el líquido contenido en las diferentes jarras de una a otra.

Este rompecabezas está incluido en el juego *Professor Layton and the Curious Village* [32].



Figura 1: Water Pitchers [43]

#### **4.1.1 Algoritmo utilizado**

Hemos decidido utilizar el algoritmo de búsqueda heurística A\* porque la heurística definida es admisible y por tanto el algoritmo encuentra la solución si existe. Además, este algoritmo nos proporciona una lista de las acciones que ha tomado para llegar del estado inicial al objetivo, por lo que podemos ir dando pistas al usuario en tiempo real.

#### **4.1.2 Modelado**

Para modelar este tipo de puzle hemos creado una instancia de la clase Problem de la librería AIMA [26]. Definimos el nodo estado como una tupla de tres posiciones en la que el valor de la primera posición es la cantidad de líquido que contiene la primera jarra, el valor de la segunda posición es la cantidad que contiene la segunda jarra y el valor de la tercera posición es la cantidad que contiene la tercera jarra.

Inicializamos la instancia de la clase Problem pasándole por parámetros el estado inicial, el estado objetivo y una tupla que indica la cantidad máxima que puede contener cada jarra.

Para poder utilizar la clase Problem tenemos que definir también las funciones de acciones, resultados y heurística.

##### **4.1.2.1 Acciones**

En la función de acciones definimos todas las acciones que se pueden tomar para un estado.

En este caso, comprobamos para cada jarra "a" si contiene líquido, si es así, comprobamos si el resto de las jarras "b" están al máximo de su capacidad, si no es así añadiremos a la lista de acciones las acciones de: verter contenido de la jarra "a" en la jarra "b", para todas las combinaciones de jarras que cumplan lo mencionado.

#### **4.1.2.2 Resultados**

En la función de resultados definimos el estado que produce el aplicar las diferentes acciones.

Teniendo en cuenta que las acciones son, “verter el contenido de la jarra “a” en la jarra “b”, calcularemos el mínimo entre el contenido de la jarra “a” y el contenido que cabe en la jarra “b”, teniendo en cuenta su capacidad máxima y la cantidad que tiene actualmente. Este mínimo que hemos calculado es la cantidad de líquido que será vertido de la jarra “a” a la jarra “b”. De esta manera nos aseguramos de que las jarras no desborden.

La función de resultado devuelve una tupla estado con los valores de cantidades de líquidos actualizados.

#### **4.1.2.3 Heurística**

Hemos determinado la heurística como la diferencia absoluta mínima de la cantidad de líquido en las jarras en el estado actual respecto al estado final.

Ejemplo: Tenemos 3 jarras con capacidades máximas de 16,9 y 7 litros, el estado objetivo es que las primeras dos jarras contengan cada una exactamente 8 litros, y el estado inicial es que la primera jarra está totalmente llena. El valor de la heurística la calculamos de la siguiente manera:

$$h' (16,0,0) = \min ( |16-8| , |0-8| , |0-0| ) = 8$$

La heurística es admisible porque subestima o iguala el coste real desde el estado actual al estado objetivo.

### 4.1.3 Generación

Teniendo en cuenta que en los puzles de este tipo el estado inicial consiste siempre en que la jarra 1 contiene su capacidad máxima y las otras dos jarras están vacías, Basta con generar números aleatorios para los valores de capacidad máxima de cada jarra y después generar valores aleatorios desde cero a la capacidad máxima para los valores del estado final de cada jarra.

Hemos querido añadir también una variable de dificultad, el usuario puede seleccionar qué dificultad tiene el problema generado. La dificultad del problema está determinada por el número de acciones que hay del estado inicial al objetivo. Ver Tabla 2.

Dificultad	Número de pasos hasta la solución
Fácil	0-7
Intermedio	8-14
Difícil	15-19
Muy difícil	20-24

Tabla 2: Dificultades generación Jarras

### 4.1.4 Tiempos

En nuestro trabajo, el tiempo que tarda el algoritmo en generar y resolver puzles es importante, ya que queremos que el usuario interactúe a tiempo real. Estas medidas de tiempo muestran también, lo bien que hemos modelado el problema y definido la función heurística. En la Figura 2, podemos observar que el tiempo en el que generamos 100 puzles validos del tipo Jarra es, en el caso peor, de algo más de 1 segundo.



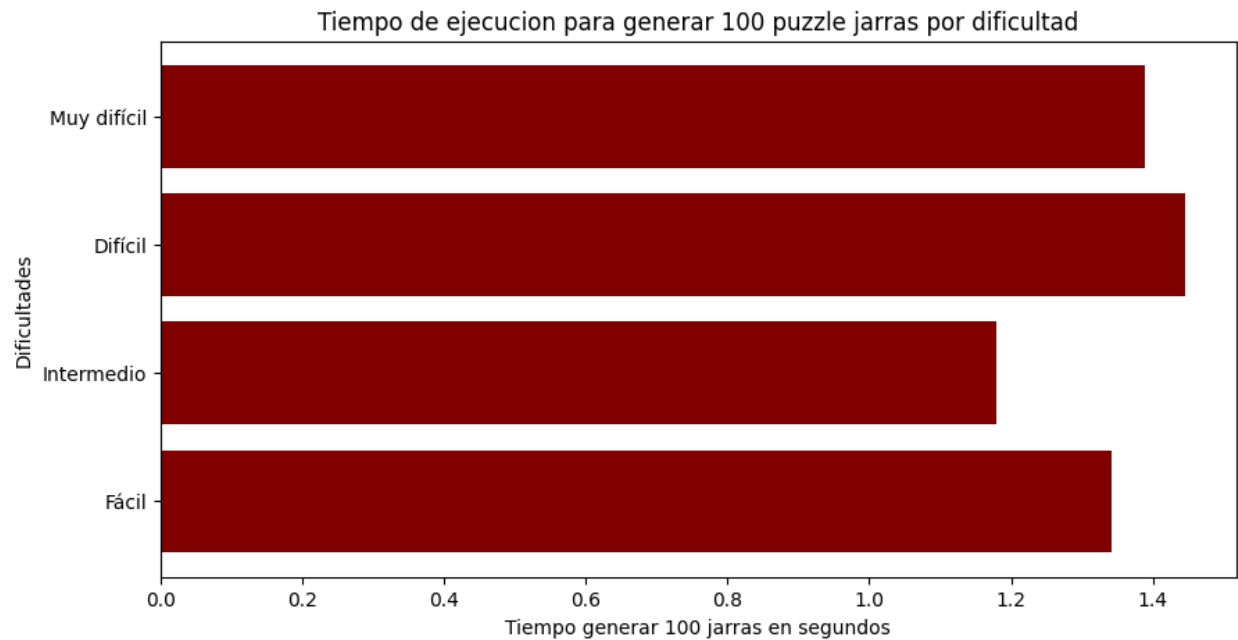


Figura 2: Tiempo generación Jarras

Por otro lado, podemos observar en la Figura 3, que el tiempo de resolución de 100 puzzles de tipo Jarra es, en el caso peor, de 0'014 segundos. Este tiempo es lo suficientemente bueno como para poder dar al usuario respuesta en tiempo real.

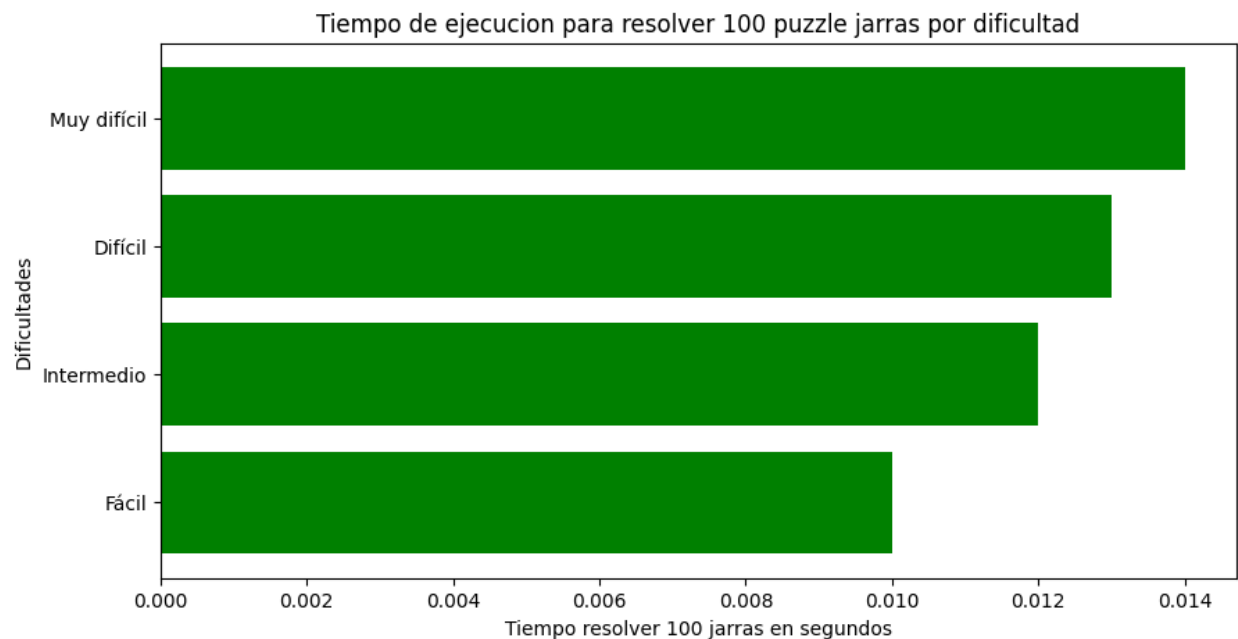


Figura 3: Tiempo resolución Jarras

#### 4.1.5 Interfaz gráfica

Mediante la interfaz gráfica el usuario puede resolver y generar este tipo de puzzles (Ver Figura 4). Si el usuario quiere resolverlo, la interfaz le solicitará los datos de las capacidades máximas de las jarras, el estado actual y el estado objetivo, tal y como se ver en la Figura 5.

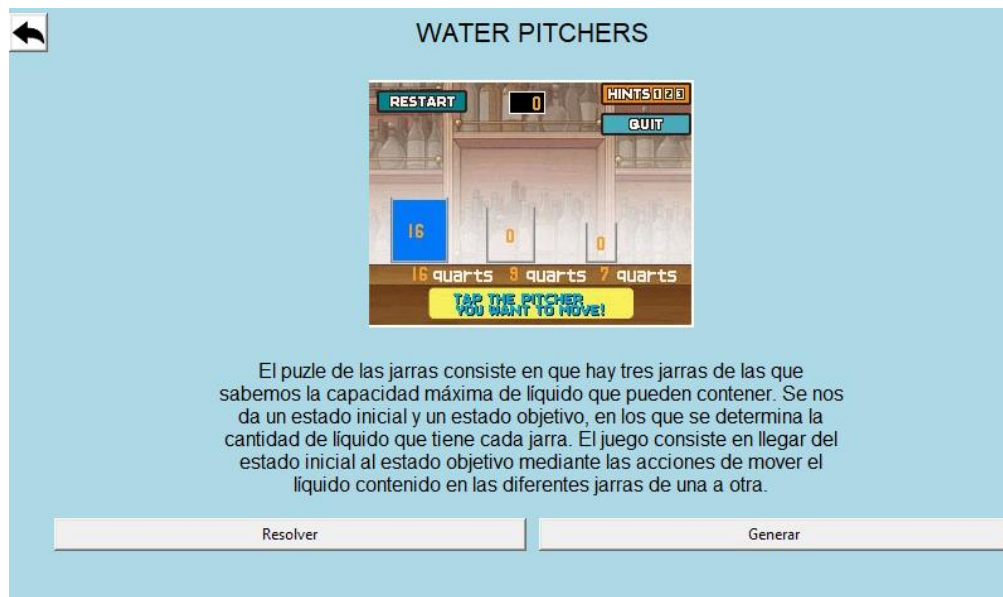


Figura 4: GUI menú Jarras

Figura 5: GUI resolver Jarras

Una vez el usuario presione el botón de resolver, aparecerá en pantalla el siguiente paso que debe realizar junto con el número de pasos que faltan hasta llegar a la solución, además de un botón para pedir otra pista, que actualizará el siguiente paso a realizar y el número de pasos que faltan hasta llegar al último. De esta manera el usuario puede ver todos los pasos que tiene que realizar para resolver el puzle. Ver Figura 6.

The screenshot shows a GUI titled "RESOLVER WATER PITCHERS" with a back arrow icon in the top left. It contains three rows of input fields for three jars. The first row shows maximum capacities: Jarra 1 (16), Jarra 2 (9), and Jarra 3 (7). The second row shows current contents: Jarra 1 (16), Jarra 2 (0), and Jarra 3 (0). The third row shows final target contents: Jarra 1 (8), Jarra 2 (8), and Jarra 3 (4). Below these fields is a "Resolver" button. Under the button, text indicates the next step: "El siguiente paso debes hacer es: Verter jarra 1 en jarra 2" and the remaining steps: "Faltan 15 para llegar a la solución". At the bottom is an "Otra pista" button.

Capacidad máxima de la jarra 1:	Capacidad máxima de la jarra 2:	Capacidad máxima de la jarra 3:
16	9	7
Contenido actual de la jarra 1:	Contenido actual de la jarra 2:	Contenido actual de la jarra 3:
16	0	0
Contenido final de la jarra 1:	Contenido final de la jarra 2:	Contenido final de la jarra 3:
8	8	4

Resolver

El siguiente paso debes hacer es: Verter jarra 1 en jarra 2

Faltan 15 para llegar a la solución

Otra pista

Figura 6: GUI resolver Jarras pista

Por otro lado, si el usuario quiere generar este tipo de puzle, se encontrará con un menú en el que tiene que seleccionar un nivel de dificultad. Una vez seleccionado el nivel de dificultad, tendrá que pulsar el botón de generar puzle y se mostrarán por pantalla las capacidades máximas de las jarras, el estado inicial y el estado objetivo, tanto escrito como dibujado. Ver Figura 7.

GENERAR WATER PITCHERS

Seleccione la dificultad del puzle generado

Intermedio

Generar Puzzle

ESTADO INICIAL

ESTADO OBJETIVO

25

Max: 25

0

Max: 2

0

Max: 18

15

Max: 25

2

Max: 2

8

Max: 18

Figura 7: GUI generar Jarras

## 4.2 Puzle 2: 8-Puzzle y 16-Puzzle (A Worms Dream)

El puzle del 8 consiste en completar un tablero cuadrado de 9 celdas con 8 fichas numeradas del 1 al 8 y una celda vacía. En realidad, a nivel de usuario, se le suele sustituir los números por partes de una imagen, aunque internamente deben tener un orden. Se nos da un estado inicial y mediante el movimiento de las fichas, deslizamiento mediante el uso de la celda vacía, hay que ordenar las fichas numeradas. Existe también otra variante de este puzle llamado el puzle 15, donde la única diferencia es el tamaño del tablero que es de 4x4.

Este juego está incluido en el juego de *Professor Layton and the Curious Village* [32]



Figura 8: A worm's Dream [33]

### 4.2.1 Algoritmo utilizado

Hemos decidido utilizar el algoritmo de búsqueda heurística A\* porque la consideramos optima y nos proporciona la solución en un formato que nos viene bien, como hemos comentado en el puzzle anterior.

### 4.2.2 Modelado

Para modelar este tipo de puzle hemos creado una instancia de la clase Problem de la librería AIMA.

Definimos el nodo estado como una tupla de 9 posiciones, donde cada posición indica el número de fragmento de la imagen.

Inicializamos la instancia de la clase Problem pasándole por parámetros el estado inicial y el estado objetivo.

Para poder utilizar la clase Problem tenemos que definir también las funciones de acciones, resultados y heurística.

#### **4.2.2.1 Acciones**

Para definir las acciones de este puzle, debemos tener en cuenta que hemos simplificado la complejidad haciendo mover la posición vacía en vez de mover cada pieza.

Por lo que, las acciones consisten en mover el hueco arriba, abajo, izquierda y derecha, siempre y cuando se cumplan las restricciones del tablero.

#### **4.2.2.2 Resultados**

Los resultados de las acciones consisten en intercambiar el valor del hueco con el valor de la posición superior, inferior, a la izquierda o a la derecha y devolver la tupla estado actualizada.

#### **4.2.2.3 Heurística**

Hemos decidido aplicar la heurística definida y estudiada en el artículo de Iordan A. C. [19]. La heurística basada en la distancia Chebyshev. Esta se calcula como la suma de la máxima de la distancia entre la celda actual y la celda objetivo, entre los ejes x e y de las fichas mal colocadas. Esta heurística es admisible porque subestima o iguala el coste real desde el estado actual al estado objetivo.

### 4.2.3 Generación

Para generar este tipo de puzle pedimos al usuario la dificultad que quiere que tenga el problema. Si la dificultad seleccionada es fácil o intermedia, generamos una tupla que contiene los números del 0 al 8 de manera desordenada, después, aplicamos aleatoriamente las acciones que están a disposición desde ese estado, creando así el estado objetivo. Depende de la dificultad aplicamos más o menos acciones. Si la dificultad seleccionada es difícil o muy difícil, hacemos lo mismo, pero con una tupla que contiene los números del 0 al 15, generando así un puzle-16. Ver Tabla 3.

Dificultad	Número de pasos hasta la solución	Tamaño del tablero
Fácil	5	3x3
Intermedio	6-10	3x3
Difícil	6-10	4x4
Muy difícil	10-15	4x4

Tabla 3: Dificultades generación puzle8/16

### 4.2.4 Tiempos

El tiempo transcurrido para generar 100 puzles resolubles de esto tipo es de 0.1 segundos en el caso peor (Ver Figura 9). A la hora de resolver, el algoritmo aplicado tarda en el caso peor menos de un segundo en resolver 100 puzles de este tipo (Ver Figura 10). Por lo que el tiempo de resolución nos permite utilizar este algoritmo en tiempo real.

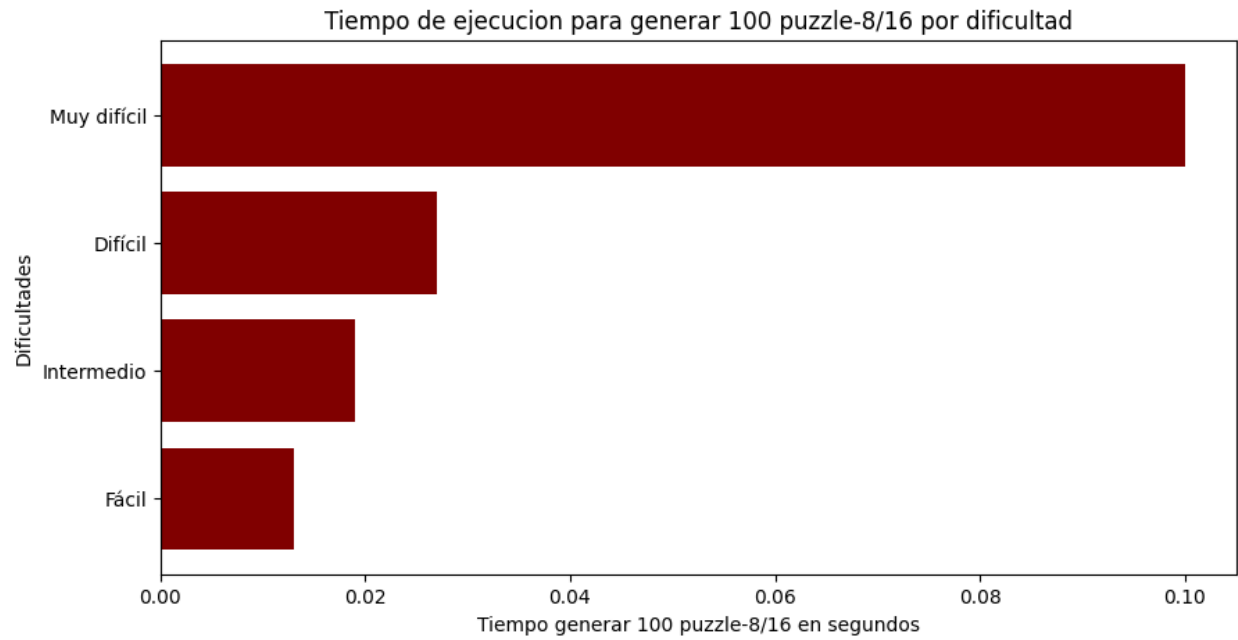


Figura 9: Tiempo generación 8/16-puzle

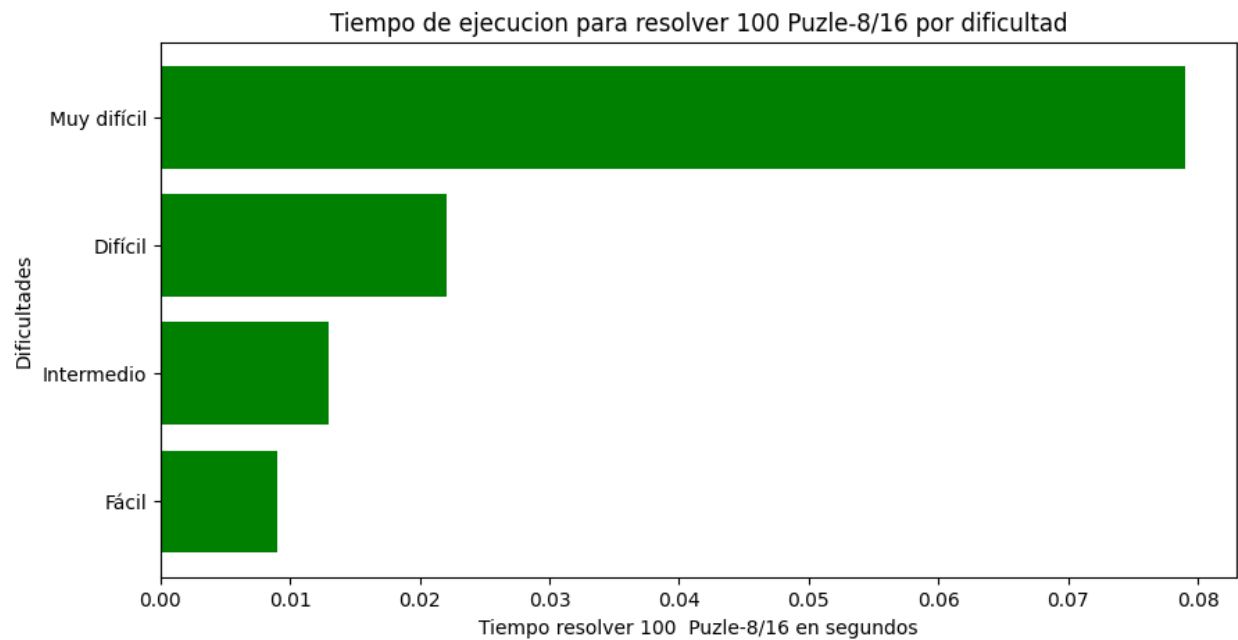


Figura 10: Tiempo resolución 8/16-puzle



### 4.2.5 Interfaz grafica

La interfaz gráfica permite al usuario tanto resolver como generar puzzles de este tipo (Ver Figura 11). Si el usuario quiere resolver un puzzle tendrá que ingresar el estado inicial y el estado objetivo, después procederá a pulsar en el botón de resolver y la interfaz mostrara el siguiente paso a la solución junto con el número de pasos que hace falta dar para llegar a ella (Ver Figura 12). También puede pedir la siguiente pista hasta llegar a la solución.



Figura 11: GUI menú 8/16-Puzzle

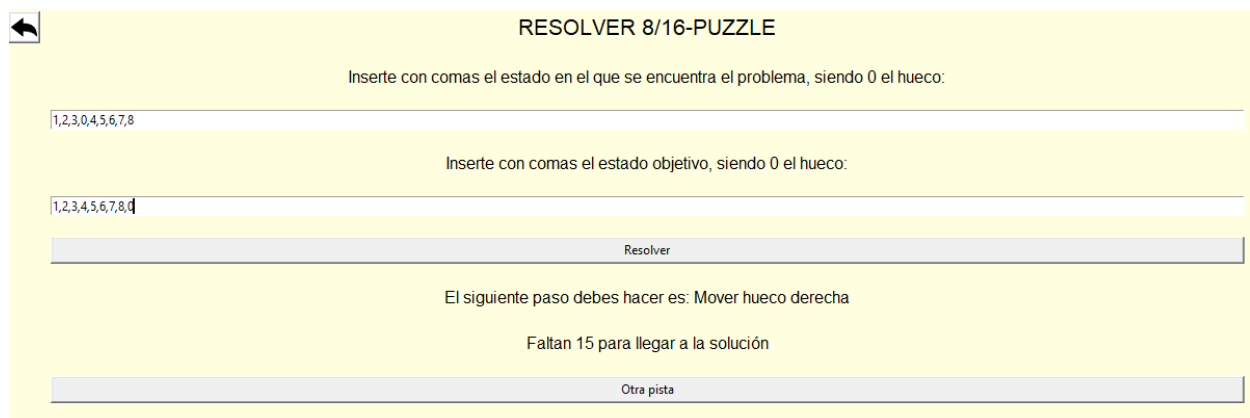


Figura 12: GUI resolver 8/16-Puzzle

Por otro lado, si el usuario desea generar este tipo de problemas, seleccionará una dificultad y la interfaz le mostrará tanto el estado inicial como el estado objetivo del problema generado. Si escoge las opciones fácil o intermedio, la interfaz genera un puzle-8 en pantalla (Ver Figura 13), si escoge difícil o muy difícil la interfaz genera un puzle-16 (Ver Figura 14).


**GENERAR 8/16-PUZZLE**

Seleccione la dificultad del puzzle a generar

Intermedio Generar Puzzle

ESTADO INICIAL				ESTADO OBJETIVO		
4		6			4	6
8	1	3		8	2	3
2	7	5		7	1	5

Figura 13: GUI generar 8-Puzle



## GENERAR 8/16-PUZZLE

Seleccione la dificultad del puzzle a generar

Difícil

Generar Puzzle

ESTADO INICIAL

3	14	2	7
9	12	4	
8	11	15	13
10	1	6	5

ESTADO OBJETIVO

14	2	12	7
3	9		13
8	11	4	15
10	1	6	5

Figura 14: GUI generar 16-Puzle

### 4.3 Puzle 4: Flip panels: Stomp on it!

Este tipo de puzle consiste en: dado un tablero donde las diferentes posiciones pueden tomar dos valores, dos colores, conseguir formar la configuración de colores que se da en el estado objetivo mediante la acción de pisar casillas. Al pisar una casilla cambian de color sus cuatro casillas adyacentes y la misma.

Este rompecabezas está incluido en el juego *Professor Layton vs. Phoenix Wright: Ace Attorney* [33]



Figura 15: Stomp on it! [34]

#### 4.3.1 Algoritmo utilizado

Hemos decidido utilizar el algoritmo de búsqueda heurística A\* por las mismas razones que los dos rompecabezas anteriores.

#### 4.3.2 Modelado

Para modelar este tipo de puzle hemos creado una instancia de la clase Problem de la librería AIMA.

Definimos el nodo estado como un array del tamaño del tablero, los valores que pueden tomar las posiciones del array son 0 para el color blanco y 1 para el color negro.

Inicializamos la instancia de la clase Problem pasándole por parámetros el estado inicial y el estado objetivo.

Para poder utilizar la clase Problem tenemos que definir también las funciones de acciones, resultados y heurística.

#### **4.3.2.1 Acciones**

Las acciones consisten en pisar cada posición del tablero.

#### **4.3.2.2 Resultados**

Los resultados de las acciones consisten en cambiar el valor de las casillas adyacentes a la pisada, si son blancas se actualizan a negro y viceversa. Las casillas que cambian su valor son la casillas superior, inferior, derecha e izquierda de la pisada, siempre que existan, cumpliendo las restricciones del tamaño del tablero.

#### **4.3.2.3 Heurística**

Hemos definido la heurística como la suma de las piezas que tienen un color diferente al del estado objetivo.

### **4.3.3 Generación**

Para la generación de este puzzle, el usuario deberá introducir una dificultad. Dependiendo de la dificultad seleccionada generamos un tablero de un tamaño específico, en el aleatoriamente asignamos los colores blanco y negro a las celdas. Este tablero generado será el estado inicial del problema. Para generar el estado objetivo aplicamos aleatoriamente un número de acciones de la lista de posibles acciones hasta generarlo. La dificultad está determinada por el número de pasos necesario para llegar a la solución y el tamaño del tablero. Ver Tabla 4.

Dificultad	Número de pasos hasta la solución	Tamaño del tablero
Fácil	1-5	3x3
Intermedio	6-20	3x3
Difícil	1-5	4x4
Muy difícil	6-20	4x4

Tabla 4: Dificultades generación Stomp

#### 4.3.4 Tiempos

En el caso de este problema, vemos que tarda más que el problema de las jarras en generar los problemas. Aunque sea bastante superior nos parece un tiempo suficientemente bueno como para dar respuesta en tiempo real al usuario. Ver Figura 16.

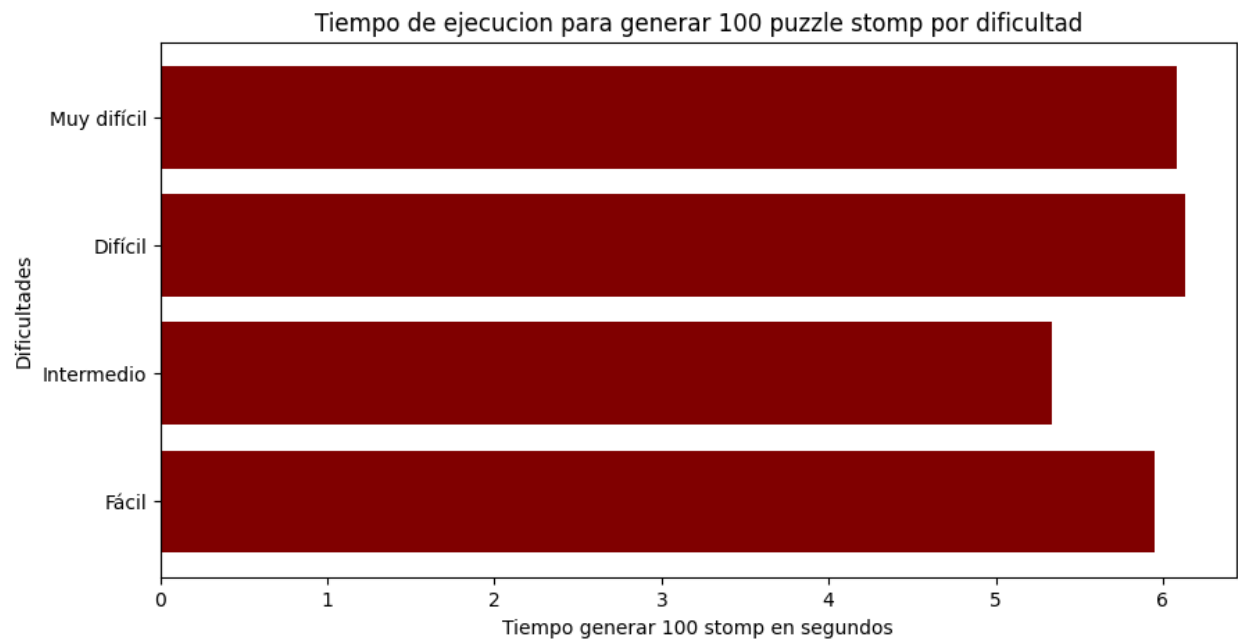


Figura 16: Tiempo generar Stomp

En el caso de resolver los problemas de este tipo, podemos observar que lo hace muy rápido (Ver Figura 17), esto quiere decir que la heurística que hemos aplicado es admisible.

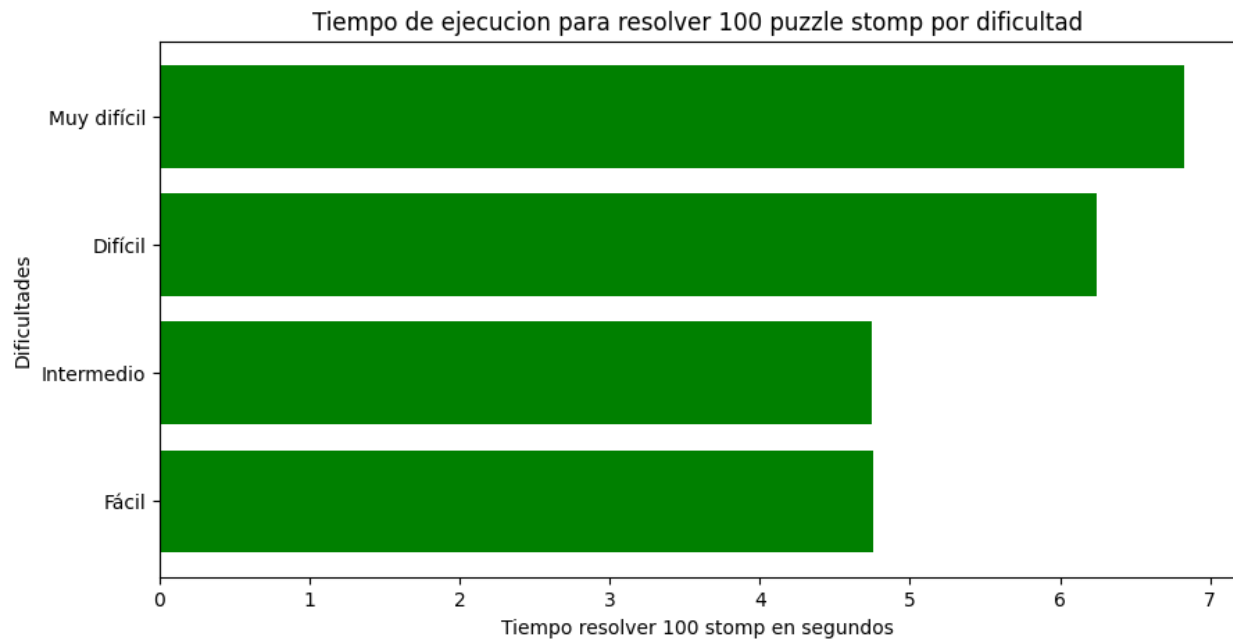


Figura 17: Tiempo resolver Stomp

#### 4.3.5 Interfaz gráfica

El usuario puede escoger si resolver o generar un puzzle de este tipo en el menú principal de la interfaz gráfica que hemos generado. Ver Figura 18.



Figura 18: GUI menú Stomp

Si el usuario escoge resolver el puzle, la interfaz pedirá que introduzca el número de filas y columnas del tablero. Se generan dos tableros del tamaño indicado por el número de filas y columnas, una para describir el estado inicial y otra para describir el estado objetivo. El usuario describe el estado pulsando en las diferentes celdas del tablero, al pulsar sobre ellas se cambia de color. Una vez haya terminado pulsará el botón de resolver y se generará la solución con la opción de enseñar los siguientes pasos. Ver Figura 19.

RESOLVER STOMP ON IT: FLIP PANELS

Numero de filas del tablero: 3

Numero de columnas del tablero: 4

Dibujar tablero

Dibuja el estado inicial


Dibuja el estado objetivo


Resolver

El siguiente paso debes hacer es: stomp on position 1,1

Faltan 5 para llegar a la solución

Otra pista

Figura 19: GUI resolver Stomp

Por otro lado, si el usuario quiere generar un puzle de este tipo, debe seleccionar una dificultad y pulsar en generar puzle. Se dibuja en pantalla, el estado inicial y el estado objetivo del problema generado. Ver Figura 20 y Figura 21.



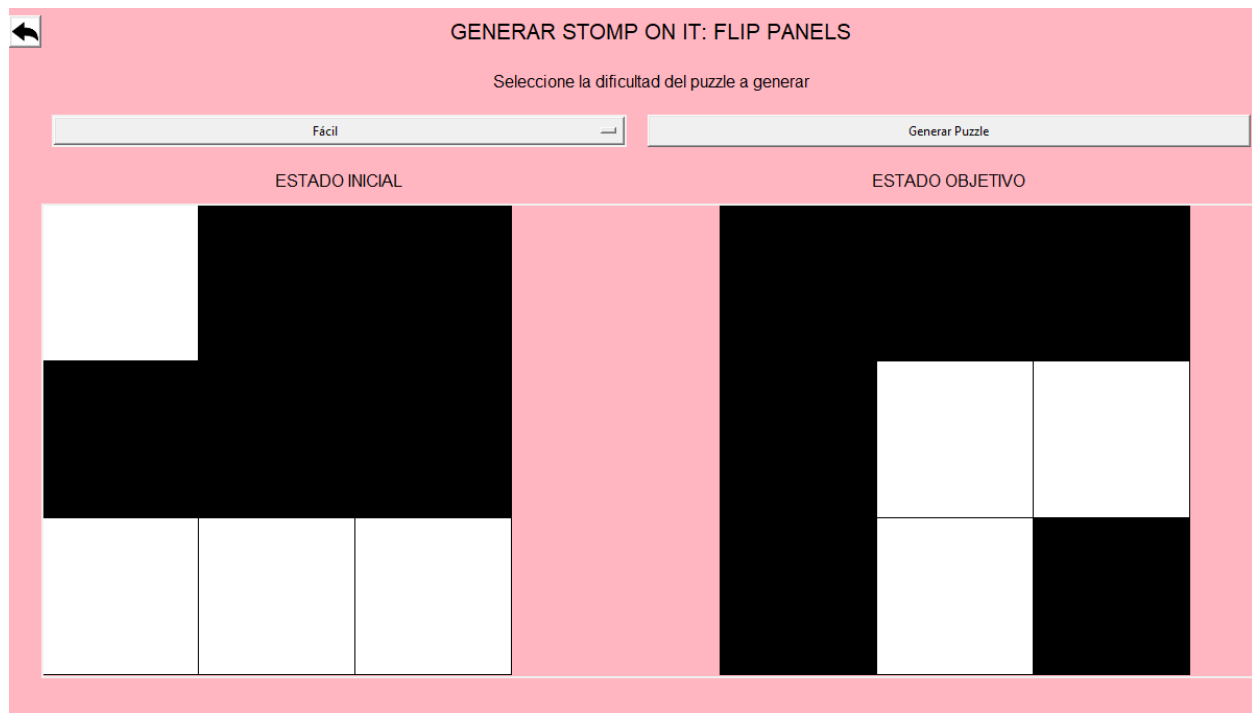


Figura 20: GUI generación Stomp 1

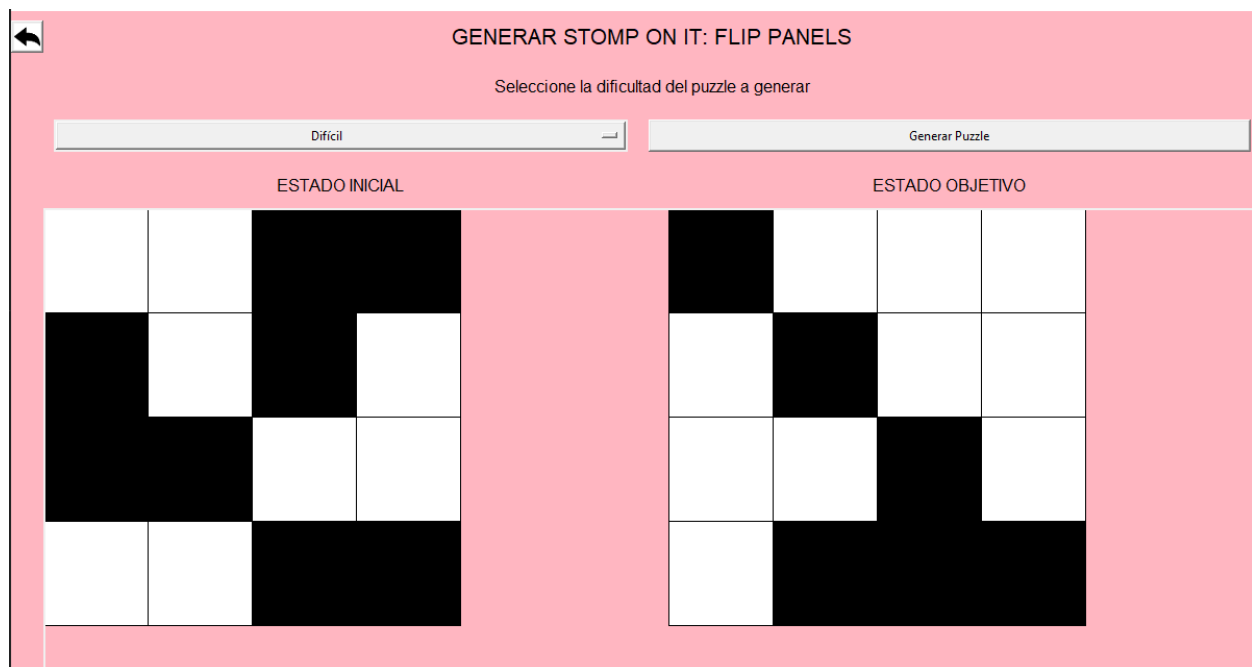


Figura 21: GUI generación Stomp 2

## 4.4 Puzle 3: Sliding: Get the ball out!

Este tipo de puzle consiste en, dado un tablero con piezas, una pelota, posiciones vacías y una posición objetivo para la pelota, conseguir mover la pelota de la posición inicial a la posición objetivo. Para ello hay que mover la pelota y/o mover las piezas para abrir camino a la pelota.

Este rompecabezas está incluido en el juego *Professor Layton and the Curious Village* [32]



Figura 22: Get the Ball Out! [35]

### 4.4.1 Algoritmo utilizado

En un principio decidimos utilizar el algoritmo de búsqueda heurística A\*, para problemas con pocas piezas conseguimos resultados buenos, pero una vez tenía más de 2 piezas, no conseguía encontrar la solución. Esto era de esperar debido a la complejidad que empieza a generarse en las funciones de acciones y resultados a medida que hay más piezas.

Finalmente hemos decidido que primero se intenta resolver con A\*. Si esta búsqueda heurística no da resultado, intentamos resolver el problema con el algoritmo de enfriamiento simulado, conocido como Simulated Annealing [24].

Hemos preferido aplicar primera la búsqueda A\* por dos razones, la generación de la solución es más rápida, y si la heurística es admisible la solución es óptima, y A\* nos proporciona las acciones que ha tenido que tomar para llegar al resultado. El algoritmo de enfriamiento simulado simplemente llega a la solución final. Por lo que, si la búsqueda A\* no resulta satisfactoria, daremos al usuario en forma de pista la posición final de alguna de las piezas del tablero. Aunque, el algoritmo de enfriamiento simulado puede no encontrar una solución.

#### **4.4.2 Modelado**

Para modelar este tipo de puzle hemos creado una instancia de la clase Problem de la librería AIMA.

Definimos el nodo estado como una tupla que describe el tablero, las posiciones vacías toman valor 0, la posición de la pelota toma el valor 1, las posiciones que ocupan las piezas toman valores de 2 en adelante, diferenciando las piezas por número y las posiciones no accesibles toman valor -1.

Inicializamos la instancia de la clase Problem pasándole por parámetros el estado inicial y la posición objetivo de la pelota. En este caso, para comprobar si hemos llegado a una solución solo debemos comprobar que la pelota esté en la posición objetivo, la configuración del resto de las piezas no importa.

Para poder utilizar la clase Problem tenemos que definir también las funciones de acciones, resultados y heurística.

##### **4.4.2.1 Acciones**

En este tipo de puzle tenemos muchas acciones, ya que podemos mover tanto la pelota como todas las piezas del tablero. Las acciones suponen mover, tanto la pelota como todas las piezas disponibles en el tablero, arriba, abajo, derecha e izquierda. Teniendo siempre en cuenta las restricciones del tablero.

#### **4.4.2.2 Resultados**

Las acciones resultan en actualizar el nodo estado dependiendo de si se ha movido la pelota o si se ha movido una pieza, ya que las piezas pueden variar en forma.

#### **4.4.2.3 Heurística**

Hemos decidido aplicar la heurística Chebyshev [19]. Esta heurística calcula el máximo de los movimientos que tiene que hacer la pelota desde el estado inicial al estado final entre el eje x y el eje y, y multiplica por dos este número. La heurística es admisible porque subestima o iguala el coste real desde el estado actual al estado objetivo, esto es debido a que no tiene en cuenta el coste de mover las piezas que pueden estar impidiendo el camino.

#### **4.4.3 Generación**

Para generar este tipo de rompecabezas generamos un tablero de 3x3 o 4x4 dependiendo de la dificultad seleccionada. Sobre ese tablero generamos un número de piezas, que depende también de la dificultad, sobre posiciones aleatorias del tablero. Después comprobamos que las piezas tienen la forma adecuada. Ya que esta implementación solo acepta piezas de forma 1x1, 2x1, 1x2 y 2x2. Finalmente comprobamos si el tablero generado tiene solución y si el número de pasos hasta la solución es el adecuado según marca la dificultad, si no es así, se vuelve a generar el tablero automáticamente hasta generar uno que sea resoluble y que cumpla las restricciones descritas en la Tabla 5.

Dificultad	Número de piezas	Tamaño de tablero	Nº de pasos hasta la solución
Fácil	0-1	3x3	0-5
Intermedio	2	3x3	5-10
Difícil	3	4x4	5-15
Muy difícil	<u>4+</u>	<u>4x4</u>	<u>5+</u>

Tabla 5: Dificultades generación Pelota

#### 4.4.4 Tiempos

Respecto al tiempo de generación vemos que cuantas más piezas tiene el tablero, y mayor es su tamaño, mayor es el tiempo que tardamos en generar 100 rompecabezas de este tipo. Podemos observar en la Figura 23, que, en el caso peor, el tiempo de ejecución es de casi un minuto.

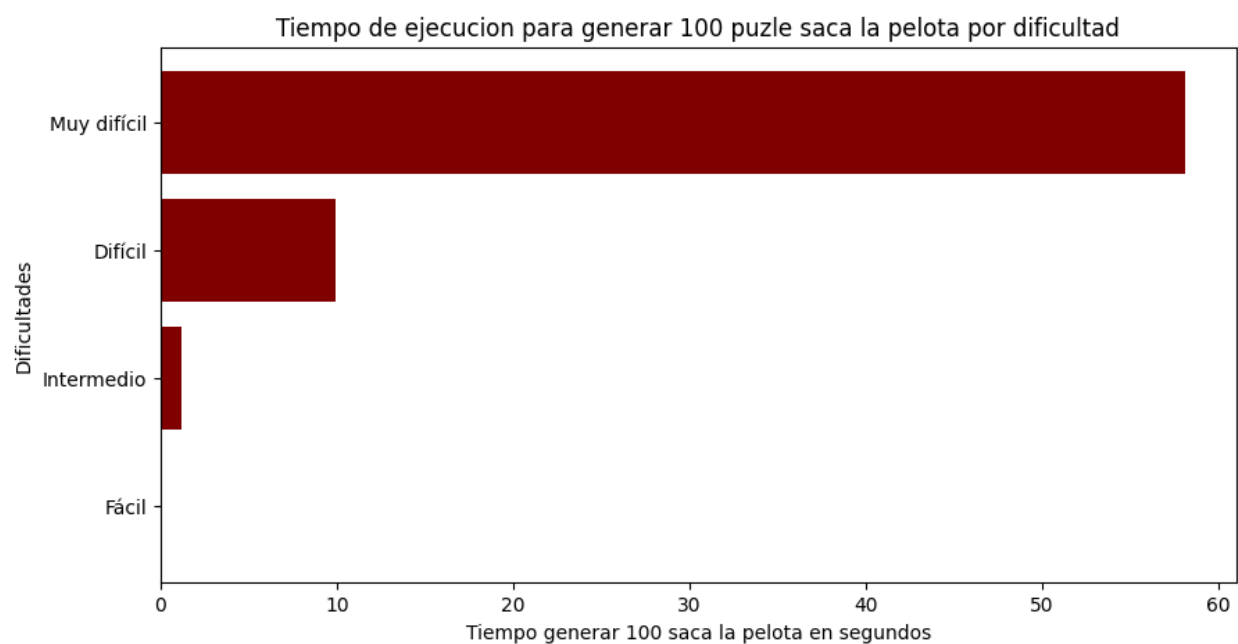


Figura 23: Tiempo generación Pelota

Respecto a los tiempos de resolución, vemos en la Figura 24 que, en el caso peor, el tiempo de ejecución es de 2'3 segundos. Esto implica que la heurística aplicada permite resolver el puzle en tiempo real para generar pistas.

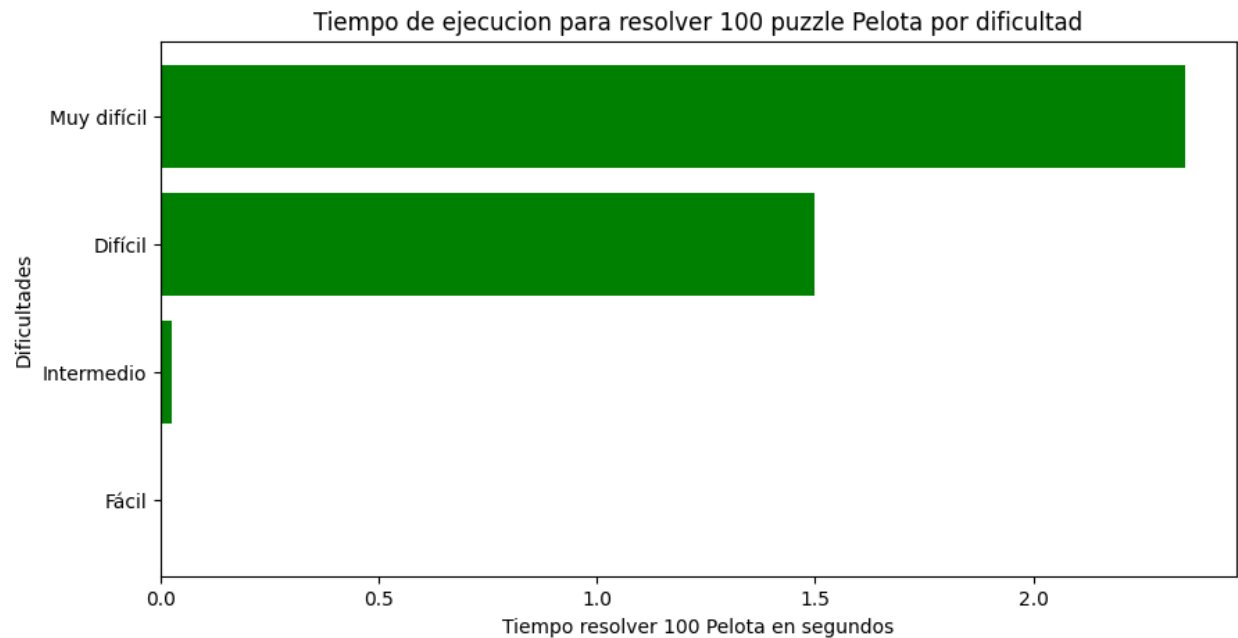


Figura 24: Tiempo resolución Pelota

#### 4.4.5 Interfaz gráfica

Mediante la interfaz gráfica el usuario puede resolver y generar este tipo de problemas (Ver Figura 25).



Figura 25: GUI menú Pelota

Si el usuario quiere resolver este tipo de puzle, debe ingresar el número de filas y columnas del tablero y pulsar en dibujar tablero. Entonces, la interfaz gráfica mostrara un tablero en el que el usuario debe pulsar sobre las celdas para cambiarles el color. Cada color de la celda indica algo diferente: el color rojo indica la posición de la pelota, el color negro indica que es una casilla no valida, el color amarillo indica la posición objetivo de la pelota, y el resto de los colores están reservados para describir las formas y posiciones de las piezas. Ver Figura 26.

**RESOLVER GET THE BALL OUT**

Numero de filas del tablero:

Numero de columnas del tablero:

**Dibujar tablero**

Haz click en las casillas para cambiar su color.(blanco:casilla vacia,  
negro:casilla no accesible, rojo:pelota, amarillo:objetivo de la pelota,  
resto de colores: piezas )

**Dibuja el estado inicial**


**Resolver**

Figura 26: GUI resolver Pelota 1

En este caso la solución puede venir dada de dos maneras, como hemos comentado en el apartado de algoritmo utilizado de este problema puede ser resuelto por un segundo algoritmo si el primero falla. Si la búsqueda A\* es satisfactoria el usuario verá por pantalla cual es el siguiente paso que debe tomar (Ver Figura 27). Pero si la búsqueda A\* no es satisfactoria, al usuario se le mostrara como queda el tablero en el estado objetivo, para que pueda ver la configuración de las piezas.



Figura 27: GUI resolver Pelota 2

Por otro lado, si el usuario quiere generar este tipo de rompecabezas, deberá seleccionar una dificultad y la interfaz mostrará por pantalla el puzle generado. Ver Figura 28 y Figura 29.

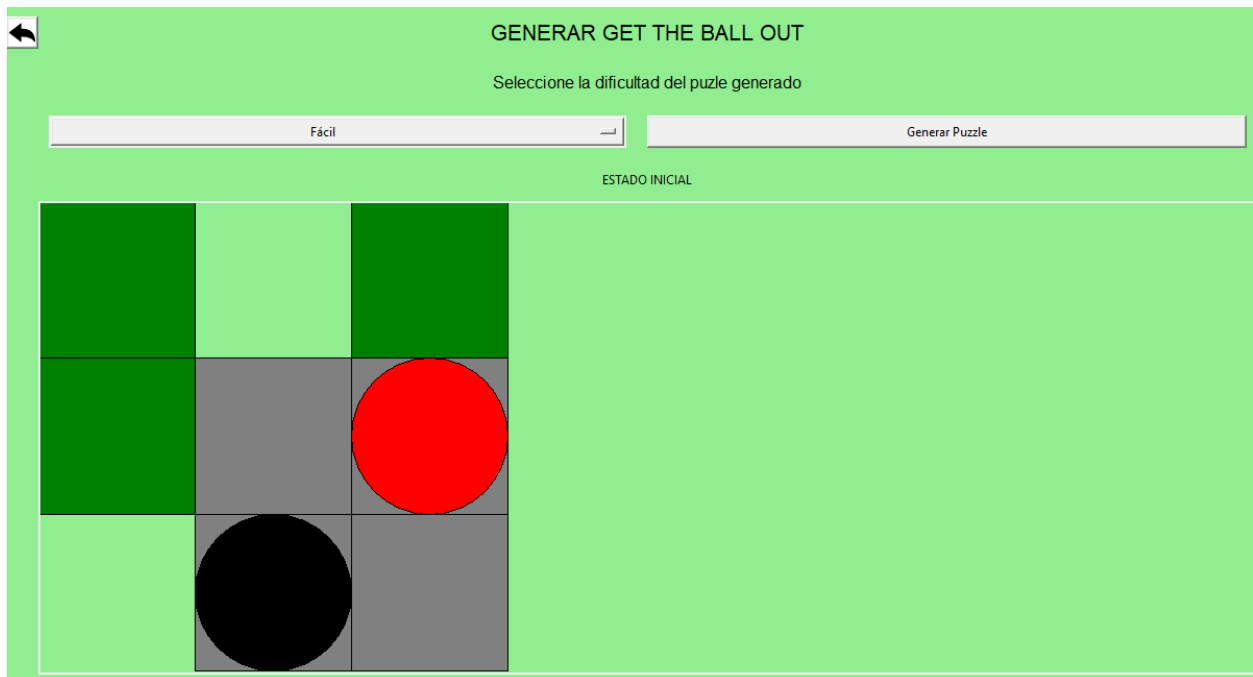


Figura 28: GUI generar Pelota 1



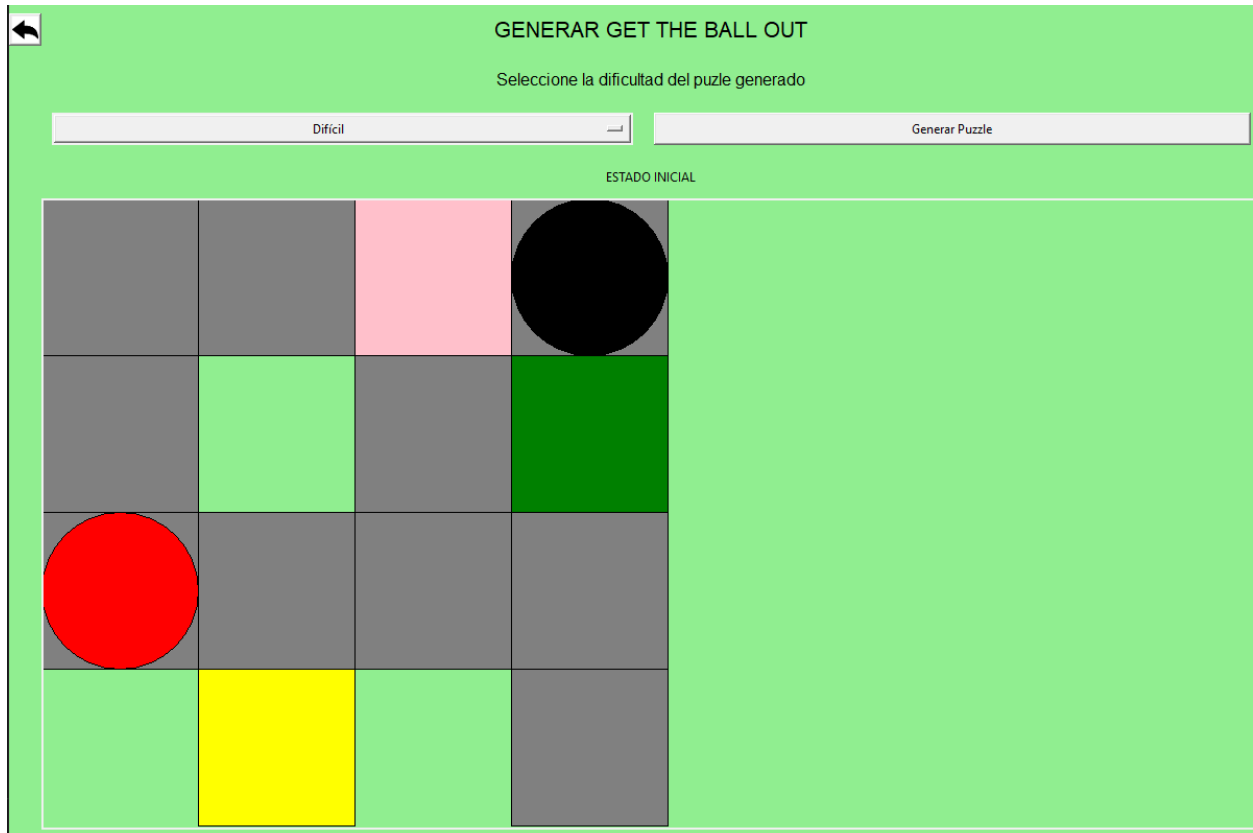


Figura 29: GUI generar Pelota 2

## 4.5 Puzle 5: Hatgram

Este tipo de puzle consiste en, dado un tablero con posiciones vacías y posiciones en las que no se pueden poner piezas, y una lista de piezas de diferentes tamaños y formas, llenar el tablero utilizando todas las piezas teniendo en cuenta que las piezas se pueden rotar y voltear.

Este rompecabezas está incluido en el juego *Professor Layton and the Unwound Future* [36]



Figura 30: Hatgram [37]

### 4.5.1 Algoritmo utilizado

A la hora de decidir qué algoritmo utilizar para encontrar solución a este tipo de problema consideramos utilizar el algoritmo de búsqueda A\* pero decidimos que no era una buena opción porque el número de piezas y las diferentes rotaciones harían que el espacio de búsqueda fuese demasiado grande como para poder resolverlo usando este algoritmo. Por lo que decidimos que el enfoque apropiado era usar algoritmos genéticos [25].

Aunque existen librerías de Python que implementan el algoritmo genético, decidimos implementarlo nosotros mismos para tener más control sobre el modelado de los individuos, generación de la población y los algoritmos de cruce, mutación y selección utilizados.

El esquema que sigue un algoritmo genético es el siguiente:

Generar aleatoriamente una población inicial de tamaño X

- Calcular la valoración de cada individuo mediante la función fitness
- Seleccionar Y individuos mediante la selección por torneo
- Aplicar el operador genético de cruce, teniendo en cuenta la probabilidad de cruce
- Aplicar el operador genético de mutación, teniendo en cuenta la probabilidad de mutación

Se cicla sobre los puntos superiores hasta que se encuentra una solución, hasta que se supera el número de generaciones previamente definido o hasta que se supera un tiempo determinado.

El tipo de solución que nos brinda un algoritmo genético es la solución total del problema, no nos da los pasos intermedios para llegar a la solución. Esto podría suponer un problema para nosotros, ya que queremos generar pistas o pasos para el usuario. Hemos decidido que damos como pista la manera correcta de colocar una pieza concreta.

#### 4.5.2 **Modelado**

Para empezar, hemos definido un array que describe el tablero, donde las posiciones no válidas toman valor -1 y las posiciones a llenar toman valor 0, y una lista de arrays, que consiste en la lista de piezas disponibles.

Hemos especificado también las rotaciones que puede tomar una pieza pueden tomar valores del 0 al 7, teniendo estos valores el siguiente significado:

- 0: la pieza tal y como viene definida
- 1: La pieza girada 90°
- 2: La pieza girada 180°

- 3: La pieza girada 270°
- 4: La pieza volteada horizontalmente
- 5: La pieza girada 90° y volteada horizontalmente
- 6: La pieza girada 180° y volteada horizontalmente
- 7: La pieza girada 270° y volteada horizontalmente

Para ciertas piezas algunas rotaciones resultan en la misma configuración de la pieza, esto no genera ningún problema.

Hemos definido los individuos como una lista de longitud igual al número de piezas a colocar, donde para cada pieza, generamos una lista que contiene las coordenadas donde se posiciona la pieza y la rotación que toma.

#### **4.5.2.1 Generación de la población**

Al generar una población, creamos muchos individuos no válidos, sea porque dos piezas se pisan entre ellas o sea porque hay piezas colocadas en posiciones no válidas. Para no descartar del todo estos individuos, hemos decidido actualizar las coordenadas de la pieza a  $[-1,-1]$  y decir que no están colocadas.

De manera que, generamos aleatoriamente la población, dando valores a las coordenadas y rotaciones de las piezas, después la pasamos por una función de validez para que marque las piezas que no se pueden posicionar como no posicionadas.

#### **4.5.2.2 Función fitness**

La función de evaluación establece una medida numérica de la bondad de la solución. En el caso de este problema la hemos especificado como el número de casillas del tablero que están ocupadas por piezas dividido entre el número de casillas del

tablero. De esta manera, damos prioridad a los individuos que más piezas del puzle tengan colocadas.

#### **4.5.2.3 Selección de individuos**

Hemos optado por utilizar el método de selección por torneo, conocido como Tournament Selection [38], de esta manera escogemos los mejores candidatos para el cruce de individuos.

En la selección por torneo escogemos subgrupos de tamaño  $p$  de individuos de la población. Los miembros de cada subgrupo se someten a la función de evaluación para ver cuál de ellos es el mejor, se elige a un individuo de cada subgrupo para el cruce. Variando el número  $p$  de individuos que participan en cada torneo se modifica la presión de selección.

Si  $p$  es alto hay muchos individuos en cada torneo y la presión de selección es alta, los peores individuos tienen pocas oportunidades y se centra la búsqueda de las soluciones en un entorno próximo a las mejores soluciones actuales.

Si  $p$  es bajo el tamaño del torneo es reducido y la presión de selección disminuye, los peores individuos tienen más oportunidades. Se deja el camino abierto para la exploración de nuevas regiones del espacio de búsqueda.

#### **4.5.2.4 Cruce**

Una vez seleccionados los individuos, éstos son recombinados mediante algoritmos de cruce para producir la descendencia que se insertará en la siguiente generación.

Hemos resuelto el cruce en cruce de un punto, conocido como SPX (Single Point Crossover) [39], que consiste en cortar los individuos por un punto escogido aleatoriamente, diferenciando así la cabeza y la cola, después intercambiamos las

colas de los dos individuos a cruzar generando de esta manera dos descendientes que contienen información de ambos padres.

#### **4.5.2.5 Mutación**

Los algoritmos de mutación consisten en modificar unos genes del cromosoma con una probabilidad ( $> 10\%$ ) para garantizar que ningún punto del espacio de búsqueda tenga una probabilidad nula de ser examinado.

Para conseguir mejores resultados hemos decidido utilizar una mutación dirigida. La mutación dirigida intenta solo mutar las partes de los individuos que tienen la capacidad de mejorar la solución.

En este caso calculamos para cada pieza la contribución que hacen al fitness general, para que las mutaciones se enfoquen en las zonas del tablero que tienen potencial de mejorar. Mutamos los individuos que tienen impacto negativo.

La mutación de un individuo consiste en cambiar los valores del individuo aleatoriamente. Hemos utilizado esta técnica para no caer en mínimos locales. Los mínimos locales ocurren cuando los individuos de nuestra población son parecidos entre ellos y no ocupan todo el espacio de soluciones, llegando de esta manera a la mejor solución dentro de un segmento del espacio de soluciones, pero que no son la solución óptima. Mediante la mutación generamos individuos más diversos explorando el espacio de soluciones de forma más amplia, consiguiendo de esta manera llegar a soluciones óptimas. Además, la mutación ayuda a mantener la diversidad genética lo que contribuye a encontrar nuevas soluciones.

### 4.5.3 Parámetros

Al algoritmo genético diseñado, debemos introducirle los valores de los parámetros de la probabilidad de cruce, probabilidad de mutación, el número de generaciones, el tamaño del torneo y el tamaño de la población.

Para que nuestro algoritmo genere en el mínimo tiempo posible las mejores soluciones, vamos a buscar el valor óptimo de estos parámetros.

#### 4.5.3.1 Número de generaciones

El número de generaciones es el parámetro que determina el número de veces que evoluciona el algoritmo genético. Un número de generaciones mayor puede proporcionar más y mejores resultados, aumentando el tiempo de ejecución.

Vemos que el número de generaciones no afecta en el resultado de la función fitness del mejor individuo (Ver Figura 31). Respecto al tiempo de ejecución vemos en la Figura 32, que es menor con el valor de 50. Como queremos que el usuario obtenga la respuesta en el mínimo tiempo posible, el valor que vamos a dar al número de generaciones es de 50.

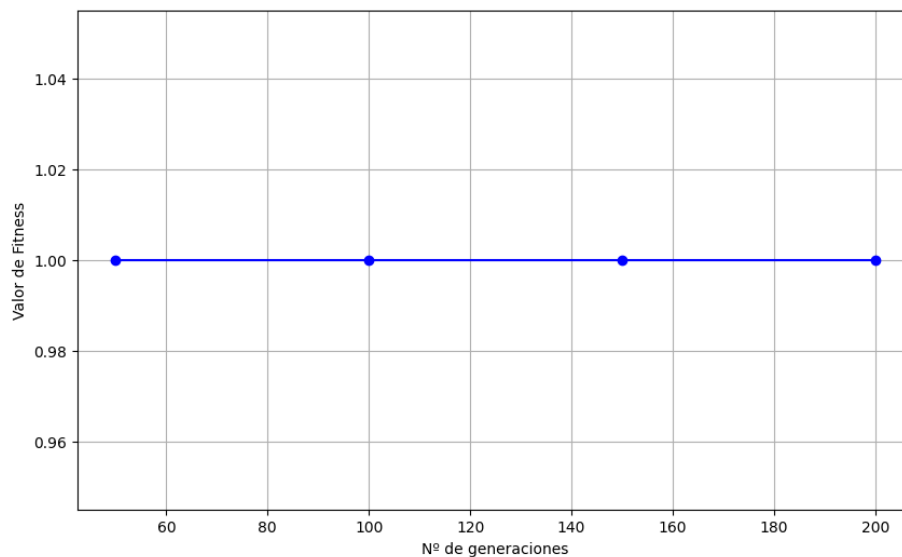


Figura 31: Fitness Número de generaciones

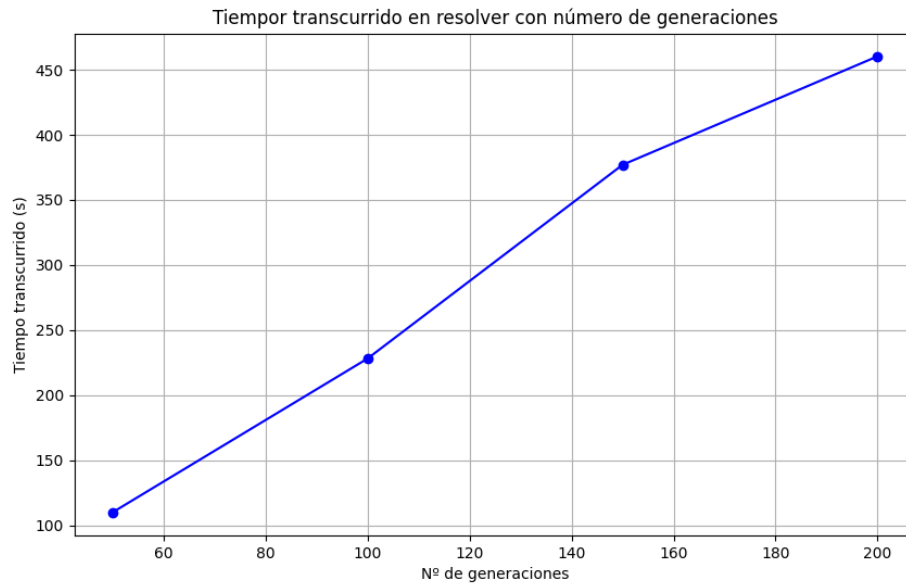


Figura 32: Tiempo ejecución número de generaciones

#### 4.5.3.2 Tamaño de la población

El tamaño de la población es el parámetro que determina cuantos individuos hay en cada generación. Un tamaño de la población mayor tolera una mayor diversidad de los individuos, reduciendo de esta manera la posibilidad de caer en mínimos locales, y aumentando el tiempo de ejecución por generación.

En nuestro caso vemos que el tamaño de la población no afecta en el valor de la función fitness (Ver Figura 33). Respecto al tiempo de ejecución vemos que cuanto mayor es la población, menor es el tiempo (Ver Figura 34). Hemos dicho que, cuanto mayor es el tamaño de la población, mayor es el tiempo de ejecución por generación, pero en nuestro caso, al generar una población mayor, tenemos mayor diversidad genética y podemos llegar a la solución con menos generaciones. Es decir, el tiempo de ejecución por generación es mayor, pero hacen falta menos generaciones para llegar a la solución por lo que el tiempo total es menor. Como el tamaño de la población no afecta a la calidad de la solución y cuanto mayor es el tamaño de la población menor es el tiempo de ejecución, decidimos dar el valor de 1500 al tamaño de la población.



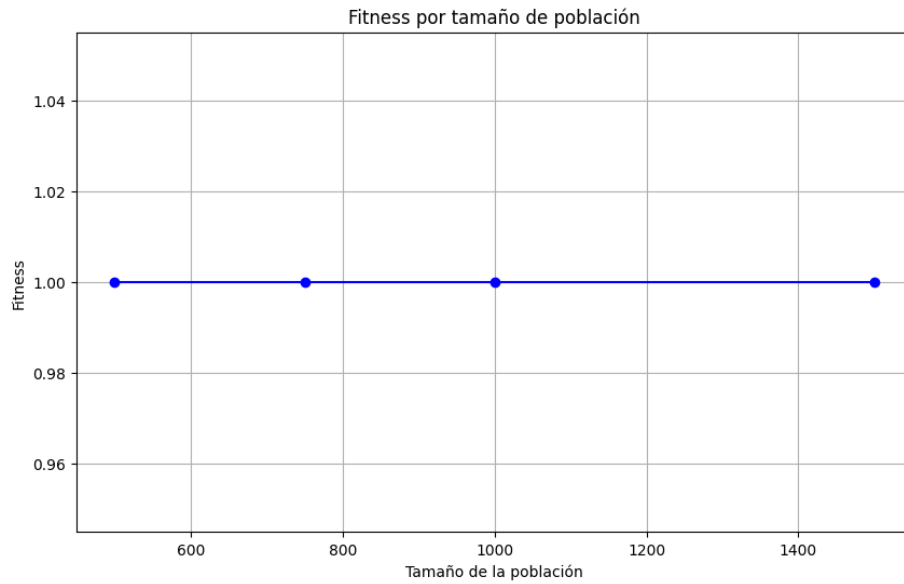


Figura 33: Fitness por tamaño de población

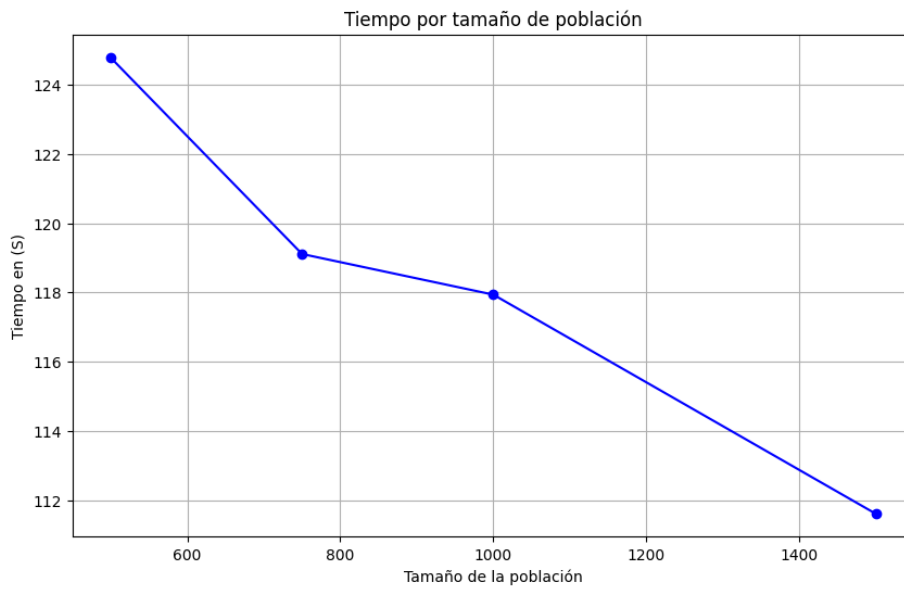


Figura 34: Tiempo por tamaño de población

#### 4.5.3.3 Tamaño del torneo

El tamaño del torneo es el parámetro que determina cuantos individuos participan en la selección por torneo. Si el tamaño del torneo es mayor los individuos con mejor valor fitness tienen más probabilidad de ser seleccionados. Por otro lado, si el tamaño del torneo es menor individuos con peor valor fitness pueden ser seleccionados, permitiendo de esta manera la exploración.

Vemos que el tamaño del torneo no afecta en el valor fitness de la solución (Ver Figura 35). Respecto al tiempo de ejecución, vemos que aumenta a medida que crece el tamaño del torneo porque tiene que comparar más individuos (Ver Figura 36). Como el tamaño del torneo no afecta a la calidad de la solución y buscamos el menor tiempo de ejecución, decidimos que el parámetro del tamaño del torneo tome valor 2.

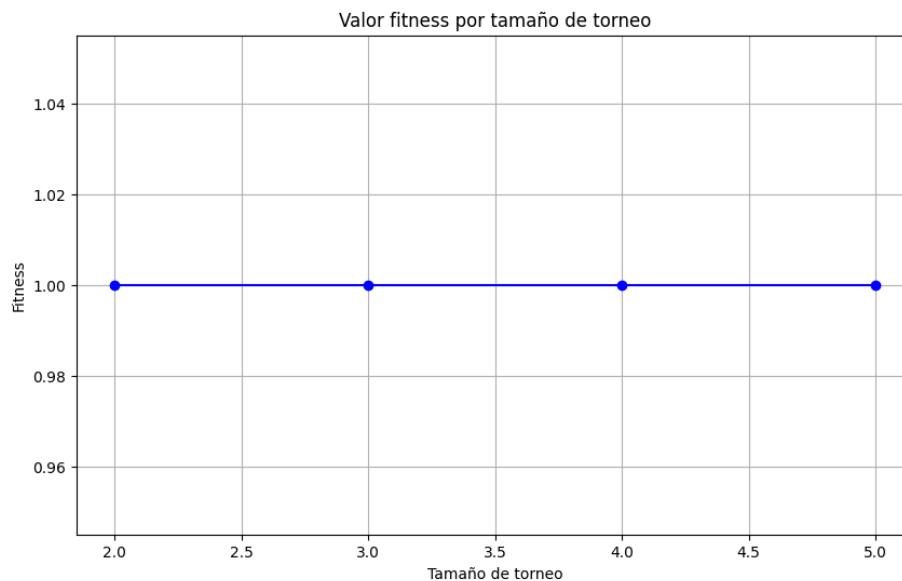


Figura 35: Fitness por tamaño de torneo

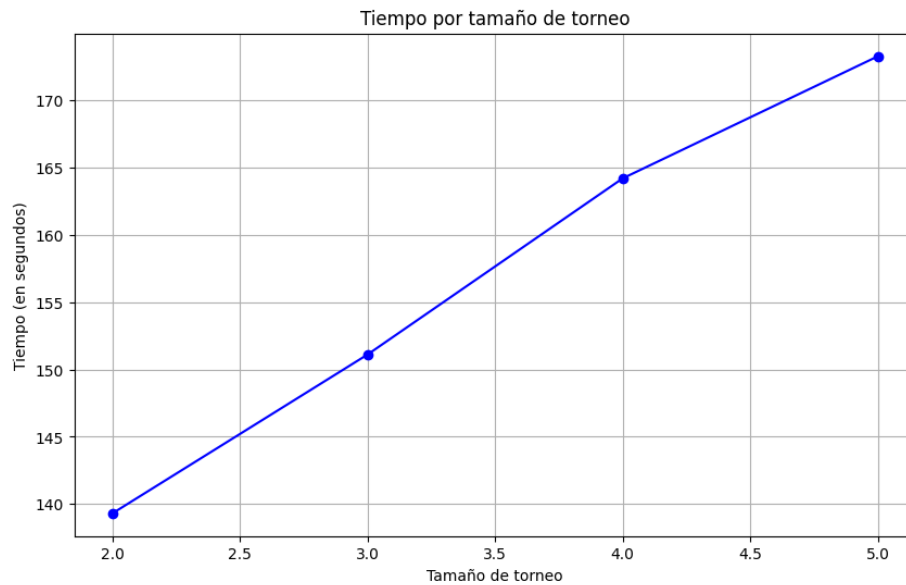


Figura 36: Tiempo por tamaño de torneo

#### 4.5.3.4 Probabilidad de cruce

La probabilidad de cruce es el parámetro que determina la frecuencia en el que se realiza el cruce entre dos individuos. Cuando la probabilidad de cruce toma un valor mayor, aumenta la diversidad de los individuos, pudiendo incluso limitar la preservación de buenos individuos. Por otro lado, si es menor no hay diversidad genética y podemos no llegar a soluciones óptimas.

Hemos decidido probar los valores 0.5, 0.7, 0.8 y 0.9 para el parámetro de probabilidad de cruce. Son relativamente altos para que haya diversidad genética. Respecto a la valoración fitness de la solución, vemos que para todos los casos es óptima (Ver Figura 37). Por lo que se refiere al tiempo de ejecución vemos en la Figura 38, que claramente hay un pico para el valor de 0.7. Con la probabilidad de cruce a este valor conseguimos encontrar las soluciones óptimas en menos tiempo, en menos generaciones.

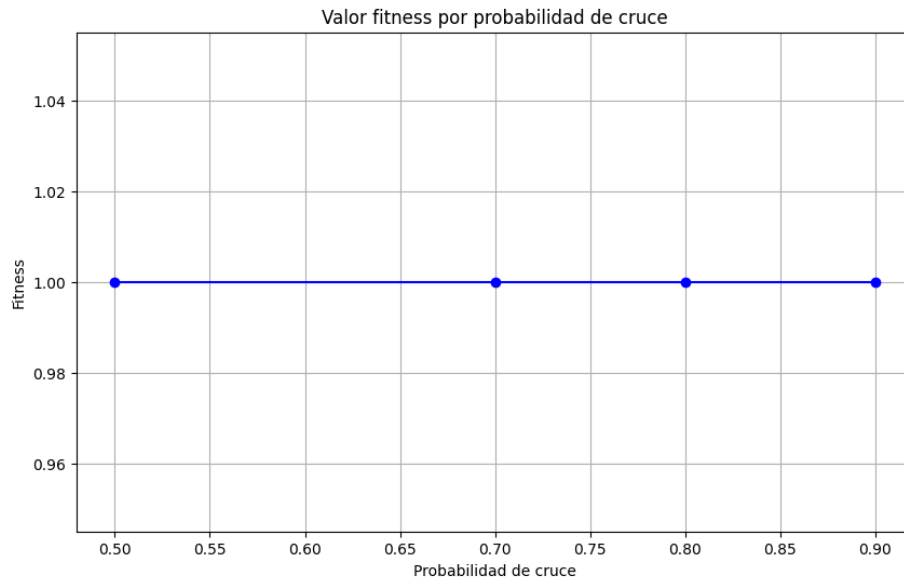


Figura 37: Fitness por probabilidad de cruce

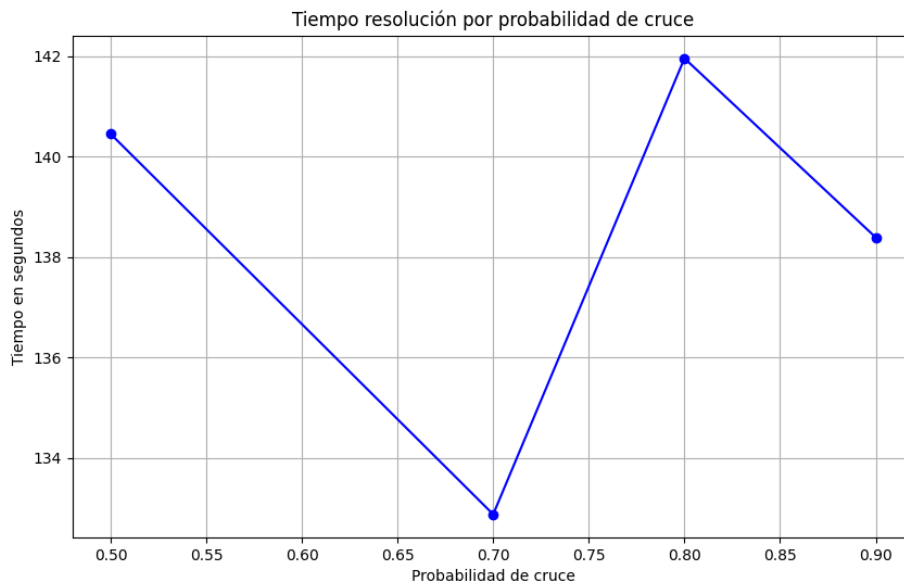


Figura 38: Tiempo por probabilidad de cruce

#### 4.5.3.5 Probabilidad de mutación

La probabilidad de mutación es el parámetro que determina la frecuencia en la que se realiza la mutación de un individuo. Si el valor de este parámetro es alto la diversidad de los individuos aumenta, pero si es demasiado alto, podemos limitar la preservación de buenos individuos. Si el valor de este parámetro es menor, el algoritmo genético no explora adecuadamente el espacio de soluciones.

En la Figura 39, podemos observar que, para los valores de mutación examinados, todos producían una solución óptima. Respecto al tiempo de ejecución, observamos en la Figura 40 que el valor ideal es el de 0.5. Este valor nos parece un poco alto y puede limitar la preservación de buenos individuos, por lo que determinamos que el valor ideal para este parámetro es de 0.15.

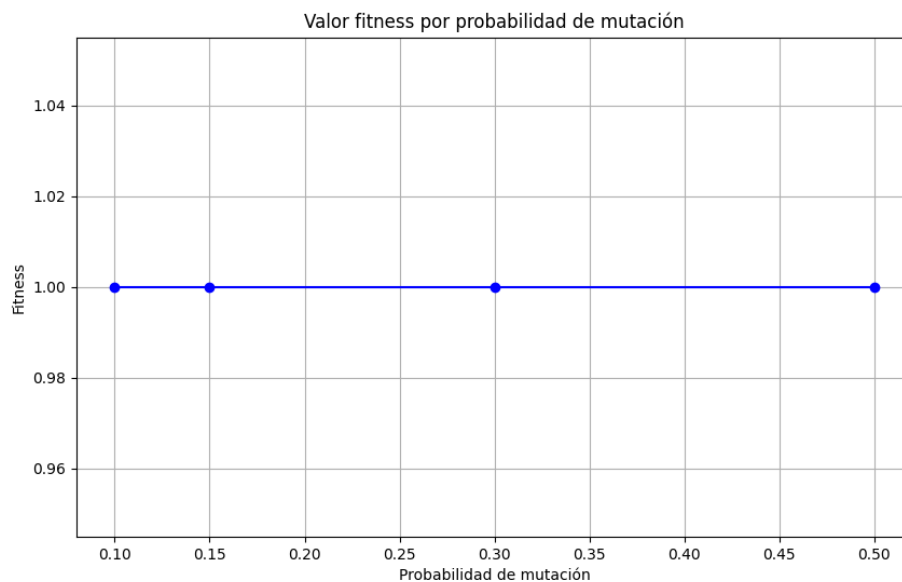


Figura 39: Fitness por probabilidad de mutación

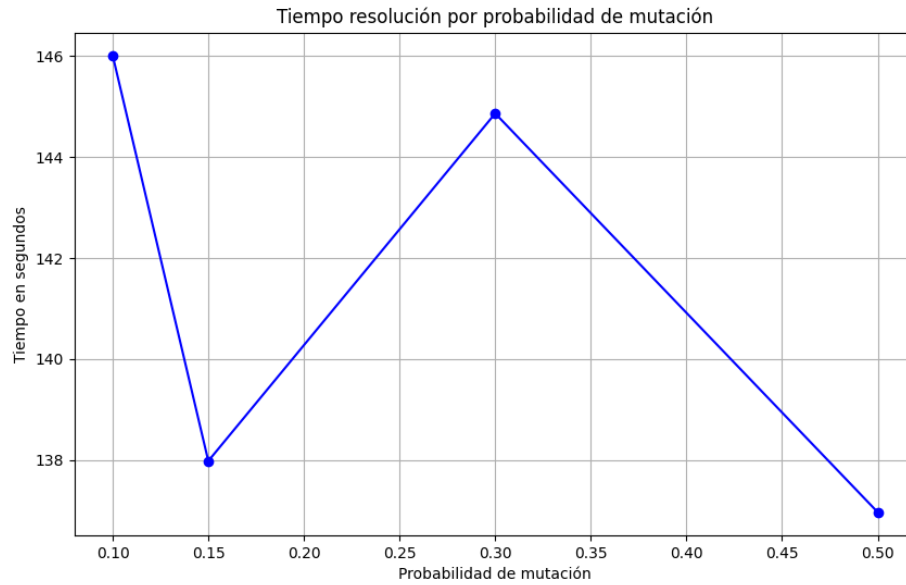


Figura 40: Tiempo por probabilidad de mutación

Tras aplicar los parámetros ideales, vemos en la Figura 41, que para 100 problemas de este tipo el valor de la función fitness de la solución es de 1 en todos los casos, es decir, el algoritmo genético genera en el 100% de las veces la mejor solución.

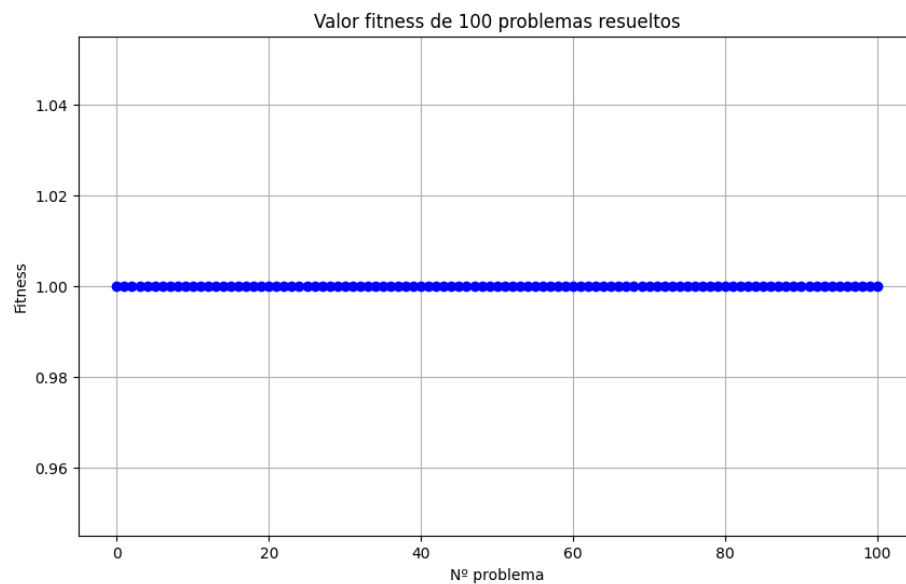


Figura 41: Fitness solución 100 Hatgram

#### 4.5.4 Tiempos

Respecto al tiempo de resolución medio es de 151 segundos, 2 minutos y 32 segundos. El tiempo de ejecución no es tan bajo como el que nos gustaría. Esto imposibilita la posibilidad de dar una solución en tiempo real. Para solucionarlo podemos iniciar el cálculo de la solución en el juego en segundo plano desde el inicio de este para así disponer de la solución cuando se solicite la pista. Pero, en cualquier caso, esto dificulta enormemente la utilización de este algoritmo en la práctica para un juego.

Se podría solucionar ejecutando el algoritmo antes del lanzamiento del juego y que la solución este guardada junto al puzzle. Esto implica que en el juego debería haber un conjunto de puzzles ya resueltos para minimizar el tiempo de generación de la pista.

#### 4.5.5 Interfaz gráfica

La interfaz gráfica muestra un menú con la explicación del puzzle y con la opción de resolverlo. Ver Figura 42.



Figura 42: GUI menú Hatgram

Si el usuario decide que quiere resolver este tipo de puzle, pulsa en el menú del puzle en resolver. La interfaz pide que el usuario proporcione el número de filas y columnas del tablero. El usuario entonces debe describir el tablero, poniendo las posiciones no válidas en negro, las vacías en blanco y si hay alguna pieza colocada pintarla de otro color. Esto se hace como hemos visto en anteriores problemas, pulsando las celdas del tablero. Por ejemplo, en la Figura 43, vemos que hay posiciones no válidas y hay una pieza ya colocada en el puzle.

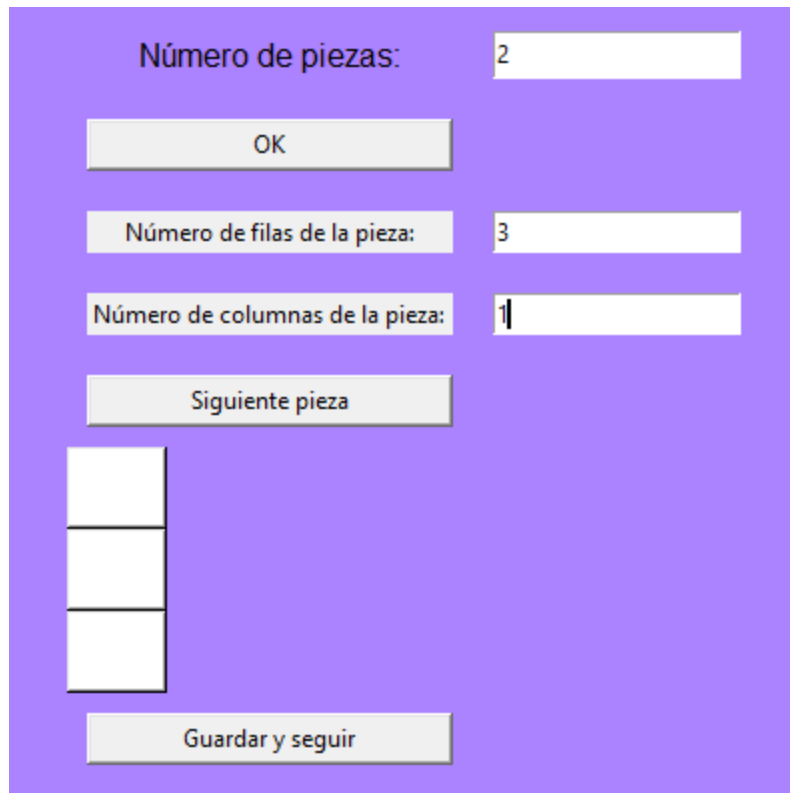


Black	Yellow	White
Yellow	Yellow	White
White	White	White

Figura 43:GUI resolver Hatgram

Cuando el usuario pulsa el botón de guardar tablero, la interfaz pedirá el número de piezas a colocar. Para cada una de las piezas la interfaz pide el número de filas y de columnas que ocupan y habrá que dibujarlas, poniendo en negro las posiciones no válidas. Ver Figura 44.





The image shows a graphical user interface (GUI) for solving a Hatgram puzzle. The background is a solid purple color. At the top, there is a label "Número de piezas:" followed by a text input field containing the number "2". Below this is a button labeled "OK". Further down, there are two more input fields: "Número de filas de la pieza:" with the value "3" and "Número de columnas de la pieza:" with the value "1". Below these is a button labeled "Siguiente pieza". At the bottom left, there is a vertical stack of three empty square boxes. At the bottom center, there is a button labeled "Guardar y seguir".

*Figura 44: GUI resolver Hatgram piezas*

Una vez el usuario ha descrito todas las piezas, pulsa en resolver para que la interfaz le muestre el resultado.

## Capítulo 5 - Conclusión y trabajo futuro

Como conclusión, hemos conseguido modelar, resolver y generar cinco tipos de puzle de manera eficaz. Los tiempos de resolución son lo bastante buenos como para considerarse en tiempo real. Además, hemos creado una interfaz gráfica sencilla para poder probar estos problemas con diferentes niveles de dificultad y ver como los algoritmos resuelven los puzles y proporcionan pistas a los usuarios.

Por otro lado, hay varios aspectos del proyecto que no hemos podido implementar por falta de tiempo. A continuación, enumeramos en orden de prioridad las mejoras que nos gustaría haber añadido en este proyecto:

- Respecto a la generación de los diferentes problemas, los algoritmos que utilizamos no son óptimos, ya que están condicionados por la aleatoriedad. En los artículos investigados sobre la generación, utilizaban algoritmos más precisos. Dejamos para trabajo futuro la investigación e implementación de algoritmos más informados para la generación de los rompecabezas investigados.
- Siguiendo con la generación, hemos definido las dificultades de los diferentes puzles con nuestro propio criterio, cada usuario debería ser capaz de definir la dificultad. Por eso, pensamos que sería conveniente que el usuario pueda escoger el número de pasos que se tienen que hacer para llegar a la solución, el tamaño del tablero y el número de fichas del problema que generamos.
- Acabando con la generación, no hemos podido generar el tipo de puzle de Hatgram, lo dejamos como trabajo futuro.
- Respecto a los tiempos de resolución de los diferentes problemas, opinamos que casi todos son óptimos, el único que difiere de la definición de respuesta en tiempo real es el rompecabezas de Hatgram. Nos gustaría mejorar el tiempo de resolución del algoritmo genético sin sacrificar los valores adecuados de las soluciones. También, como comentamos en el apartado de tiempo, se puede optar en el juego por incluir un set de puzles previamente resueltos para conseguir tener la pista en tiempo real o lanzar el cálculo en

paralelo mientras el usuario observa el puzle, para evitar el tiempo de espera de 2 minutos en encontrar la solución.

- Con relación a los problemas modelados, son unos pocos entre cientos, nos agradaría modelar más tipos de problemas o de alguna manera unificar la resolución de muchos tipos de problema, sin tener que aplicar diferentes heurísticas.
- En lo que corresponde a la interfaz gráfica, quisiéramos hacerla más amigable con el usuario. Sobre todo, en los aspectos de introducir el estado en el que se encuentra el problema.
- Por otro lado hubiese sido de gran interés investigar el uso de redes neuronales para las diferentes aplicaciones de las heurísticas. Tal y como se comenta en el artículo mencionado en el estado del arte.
- Para finalizar, nos hubiese gustado usar variantes de A\* más optimas como las descritas en el estado del arte, para aquellos puzles donde A\* no ha podido resolverlos en un tiempo satisfactorio y hemos utilizado algoritmos de búsqueda local. Esto es debido a que A\* nos puede proporcionar la secuencia de pasos, pero otros algoritmos como el algoritmo genético solo nos proporciona la solución final, lo que dificulta la generación de pistas para el usuario.

# Introduction

## Motivation

Artificial Intelligence field has attracted significant attention for the last few years, mostly because of the popularization and public access of AI based chat systems as ChatGPT or multimedia content generators as Stable Diffusion and Sora. As people have become more familiar with these tools, they have become more interested in how they work and to what other sectors may be applied.

However, there is still room for the use of classical symbolic AI in multiple applications. For example, ChatGPT has troubles doing complex mathematical calculations [1], furthermore, it is not efficient for optimal puzzle solving, such as the 8-Puzzle and the Rubik's cube. In particular, in a recent study [2] LLMs were configured specifically for puzzle Solving, achieving a 93.2% success in the 8-Puzzle, when A\* algorithm is known to solve it in Little time with a 100% success rate. Despite the advantages achieved with the use of LLMs, in this type of problems it still is more convenient to turn to the classical symbolic AI.

In this final project we want to solve and generate puzzles from the well-known video game saga Professor Layton [3], via the use of heuristic search algorithms.

The video game saga of Professor Layton focuses on Solving mysteries; to move forward in the story, we have to solve a multitude of puzzles of all kind and difficulties. All this enlivened by a story that gives consistency to the succession of puzzles that we solve.

The Motivation behind this Project is to allow puzzle game developers to have an assistant that in the case that players get stuck, help them with some kind of hint. This help can be proportioned through a button that when pressed, provides the user with the next step if

must do to get to the solution. For more complex puzzles, we may require other types of aids as we will see below.

From game developers' point of view, we want to allow them Generating puzzles automatically, so that they can use their capacities in other aspects of game designing. If they decide they want to generate their own puzzles, they can check if the puzzles are solvable through this tool.

Puzzle generation and Solving is a topic of popular interest, which is reflected in the number of articles published in recent years. We mention and discuss these articles in the next chapter.

## **Goals**

The main objective of this project is to solve and generate certain puzzle types from the video game saga Professor Layton through Artificial Intelligence techniques in a Good enough time to be able to give clues or partial solutions to the user in real time.

In order to achieve the main goal, we had to achieve the following subobjectives:

- Model some types of sample puzzles.
- For each type of puzzles, determine which search algorithm is the most optimum one to resolve it.
- Establish the ideal parameters of the different models so that the execution time is minimum, and the solution is optimum.
- Being able to display problems and solutions via a graphic interface.
- Generate a variety of puzzles with different difficulties.

## **Work plan**

### **October 2023:**

- Study the existence of puzzle generation algorithms.

### **November 2023:**

- Investigate several types of AI based problem Solving algorithms.

### **December 2023:**

- Model and solve some puzzles through heuristic search.

### **January 2024:**

- Study other types of searches for puzzles that cannot be resolved through heuristic search.

### **February 2024:**

- Apply genetic algorithms for resolving one of the puzzles.

### **March 2024:**

- Finish modeling and solving all the puzzles.

### **April 2024:**

- Generate via randomization the diverse types of puzzles.

**May 2024:**

- Start to generate a graphic interface.
- Start to draft the dissertation.

**June 2024:**

- Testing and fixing the resolution and generation of the different models.

**July 2024:**

- Study the ideal values for the genetic algorithm parameters.

**August 2024:**

- Finish the graphical interface.
- Get plots for the execution time so the solution and generation of the different puzzles modeled.





## Conclusions and future work

To conclude, we have managed to model, resolve, and generate five types of puzzles efficiently. Moreover, the solving times are Good enough to be considered real-time. In addition, we have created a simple graphical interface to be able to test these problems at different difficulty levels and see how the algorithms solve the puzzles and provide hints to the users.

On the other hand, there are various aspects of the project we have not been able to implement. Below, we list in order of priority the improvements we would like to have added to the project:

- Regarding the generation of the problems, the algorithms we have used are not optimal due to their reliance on randomness. On the articles researched about generation, they used more precise algorithms. We leave for future work the investigation and implementation of more informed algorithms for the generation of the investigated puzzles.
- While on the subject of generation, we have defined the difficulties of the different puzzles with our own criterion, each user should be able to define the difficulty. Therefore, we thought it would be convenient that the user can choose the number of steps that have to be done to reach the solution, the size of the board and the number of tiles of the problem generated.
- Putting an end to the subject of generation, we have not been able to generate the puzzle called Hatgram, we leave it as future work.
- Concerning resolution times of the different problems, we think all of them are optimal, the only one that differs from the definition of real-time Answer is the one for solving the Hatgram puzzle. We would like to improve the solving time of the genetic algorithm without sacrificing the adequate solution values. Also, as mentioned, the developer could choose to include a set of previously solved puzzles in the game to get the clue in real time or to launch the

calculation in parallel while the user observes the puzzle, to avoid the waiting time of 2 minutes to find the solution.

- In relation to the modeled puzzles, they are a few among hundreds, we would like to model more types of problems or somehow unify the resolution of many types of problems without having to apply different Heuristics.
- As far as the graphical interface is concerned, we would like to make it more user-friendly. Especially, in the aspects of entering the initial state of the problem.
- On the other hand, it would have been of great interest to investigate the use of neural networks for the different applications of heuristics. As discussed in the article mentioned in the state of the art.
- Finally, we would have liked to use more optimal variants of A\* as the ones described in the state of the art, for those puzzles where A\* has not been able to solve them in a satisfactory time and we have used local search algorithm. This is because A\* can provide us with the sequence of steps, but other algorithms such as the genetic algorithm only provide us with the final solution, which makes it difficult to generate clues for the user.

# Bibliografía

- [1] S. Frieder, L. Pinchetti, Griffiths, R. R., T. Salvatori, T. Lukasiewicz, P. Petersen y J. Berner, «Mathematical capabilities of chatgpt,» *Advances in neural information processing systems*, vol. 36, 2024.
- [2] R. Ding, C. Zhang, L. Wang, Y. Xu, M. Ma, W. Zhang, S. Rajmohan, Q. Lin y D. Zhang, «Everything of thoughts: Defying the law of penrose triangle for thought generation,» *arXiv preprint*, p. arXiv:2311.04254, 2023.
- [3] Level-5, «The Layton Series,» Nintendo, 2007.
- [4] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker y J. Togelius, «Automatic Generation and Analysis of Physics-Based Puzzle Games,» *IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 1-8, 2013.
- [5] M. Foltin, «Automated maze generation and human interaction,» *Brno: Masaryk University Faculty Of Informatics*, 2011.
- [6] R. C. Prim, «Prim's algorithm,» 1957.
- [7] R. Sedgewick y K. Wayne, «Minimum Spanning Tree,» 2007. [En línea]. Available: <https://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/19MST.pdf>.
- [8] J. Kruskal, «Kruskal's algorithm,» 1956.
- [9] D. H. Lehmer, «Backtracking,» 1950s.
- [10] «The Buckblog,» 2011. [En línea]. Available: <https://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm>.

- [11] W. T. Tutte, «Tutte algorithm».
- [12] K. Sugiyama, R. Osawa y S.-H. Hong, «Puzzle generators and symmetric puzzle layout,» *ACM International Conference Proceeding Series*, pp. 97-105, 2005.
- [13] D. East y M. Truszczyński, «aspss -- and implementation of answer-set programming with propositional schemata,» *Logic Programming and Nonmonotonic Reasoning: 6th International Conference, LPNMR*, pp. 402-4025, 2001.
- [14] R. Finkel, W. Marek y M. Truszczynski, «Generating cellular puzzles with logic programs,» [En línea]. Available: [https://d1wqtxts1xzle7.cloudfront.net/72115350/Generating\\_Cellular\\_Puzzles\\_with\\_Logic\\_P20211010-10778-7adcca.pdf?1633903994=&response-content-disposition=inline%3B+filename%3DGenerating\\_Cellular\\_Puzzles\\_with\\_Logic\\_P.pdf&Expires=1725387626&Signature=DPv8KM](https://d1wqtxts1xzle7.cloudfront.net/72115350/Generating_Cellular_Puzzles_with_Logic_P20211010-10778-7adcca.pdf?1633903994=&response-content-disposition=inline%3B+filename%3DGenerating_Cellular_Puzzles_with_Logic_P.pdf&Expires=1725387626&Signature=DPv8KM). [Último acceso: 1 09 2024].
- [15] S. Edelkamp, *Heuristic search: theory and applications*, Elsevier, 2011.
- [16] A. Iordan, «A comparative study of the A\* heuristic search algorithm used to solve efficiently a puzzle game,» de *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, 2018.
- [17] F. Rothlauf, *Design of modern heuristics: principles and application*, Berlin: Springer, 2011.
- [18] R. Coghetto, «Chebyshev distance,» *Formalized Mathematics*, vol. 24, nº 2, pp. 121-141, 2016.
- [19] A.-E. Iordan, «A comparative study of three heuristic functions used to solve the 8-puzzle,» *British Journal of Mathematics & Computer Science*, vol. 16, nº 1, pp. 1-18, 2016.

- [20] G. Þ. Guðmundsson, Solving general game playing puzzles using heuristics search, Doctoral dissertation, 2009.
- [21] M. Świechowski, K. Godlewski, B. Sawicki y J. Mańdziuk, «Monte Carlo tree search: A review of recent modifications and applications,» *Artificial Intelligence Review*, vol. 56, n° 3, pp. 2497-2562, 2023.
- [22] M. Samadi, A. Felner y J. Schaeffer, «Learning from Multiple Heuristics,» *AAAI*, pp. 357-362, 2008.
- [23] P. E. Hart, N. J. Nilsson y B. Raphael, «A\* algorithm,» 1968.
- [24] P. J. Van Laarhoven, Simulated annealing, Netherlands: Springer, 1987.
- [25] J. H. Holland, «Genetic algorithms,» *Scientific american*, vol. 267, n° 1, pp. 66-73, 1992.
- [26] «Aima code python,» [En línea]. Available: <https://github.com/aimacode/aima-python>.
- [27] P. Norvig y S. J. Russell, Artificial Intelligence: A Modern Approach, Prentice Hall, 2020.
- [28] C. Harris, K. Millman y S. van der Walt, «Array programming with NumPy».
- [29] Python Software Foundation, «Tkinter».
- [30] Python Software Foundation, «Time».
- [31] H. J. D., «Matplotlib: A 2D graphics environment,» *Computing in Science & Engineering*, vol. 9, n° 3, pp. 90-95, 2007.
- [32] Level-5, «Professor Layton and the Curious Village,» Nintendo, 2007.

- [33] Level-5;Capcom, Nintendo, 2012.
- [34] Level-5, «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:Stomp\\_on\\_It!?file=VS38.png](https://layton.fandom.com/wiki/Puzzle:Stomp_on_It!?file=VS38.png).
- [35] Level-5, «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:Get\\_the\\_Ball\\_Out!\\_1?file=CV058.gif](https://layton.fandom.com/wiki/Puzzle:Get_the_Ball_Out!_1?file=CV058.gif).
- [36] Level-5, «Professor Layton and the Unwound Future,» 2008.
- [37] Level-5, «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:The\\_Professor%27s\\_Hat?file=UF011.png](https://layton.fandom.com/wiki/Puzzle:The_Professor%27s_Hat?file=UF011.png).
- [38] B. L. Miller y D. E. Goldberg, «Genetic algorithms, tournament selection, and the effects of noise,» *Complex systems*, vol. 9, n° 3, pp. 193-212, 1995.
- [39] P. Kora y P. Yadlapalli, «Crossover operators in genetic algorithms: A review,» *International Journal of Computer Applications*, vol. 162, n° 10, 2017.
- [40] «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:Water\\_Pitchers?file=CV078.gif](https://layton.fandom.com/wiki/Puzzle:Water_Pitchers?file=CV078.gif).
- [41] Level-5, «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:A\\_Worm%27s\\_Dream?file=CV107.gif](https://layton.fandom.com/wiki/Puzzle:A_Worm%27s_Dream?file=CV107.gif).
- [42] G. Meehan y P. Gray, «Constructing crosswoed grids: Use of heuristics vs constraints,» *Proceeding of Expert Systems*, vol. 97, pp. 159-174, 1997.
- [43] Level-5, «layton.fandom.com,» [En línea]. Available: [https://layton.fandom.com/wiki/Puzzle:Water\\_Pitchers?file=CV078.gif](https://layton.fandom.com/wiki/Puzzle:Water_Pitchers?file=CV078.gif).

