

MEMORIA PRÁCTICA 2

HEURÍSTICA Y OPTIMIZACIÓN

DICIEMBRE 2025



Gonzalo Vela Sacristán & Guillermo García Ortega
100472334 & 100472109



Introducción.....	2
1. Satisfacción de restricciones.....	2
1.1 Modelización del problema.....	2
1.2 Implementación en código.....	3
1.2.1 parte-1.py.....	3
1.2.2 aux1.py.....	3
2. Algoritmos de búsqueda.....	4
Modelización.....	4
grafo.py (Estructura de Datos).....	4
abierta.py (Lista Abierta).....	5
cerrada.py (Lista Cerrada).....	5
algoritmo.py (Lógica de Resolución).....	5
parte-2.py (Orquestación e I/O).....	6
pruebas.py (Validación y Análisis).....	6
Conclusión.....	7
3. Análisis de Resultados.....	8
Satisfacción de restricciones.....	8
Búsqueda.....	9



Introducción

El objetivo de la práctica es modelar y resolver una tarea de satisfacción de restricciones, así como modelar e implementar eficientemente la resolución de un problema de búsqueda heurística.

1. Satisfacción de restricciones

Para este apartado se describe un pasatiempo llamado BINARIO, que se juega sobre una rejilla cuadrangular de dimensiones $n \times n$, y en la que es preciso disponer discos blancos y negros, de tal modo que:

- No debe quedar ninguna posición vacía.
- El número de discos blancos y negros en cada fila, y en cada columna, debe ser el mismo.
- No es posible disponer más de dos discos del mismo color consecutivamente en una fila o columna.

1.1 Modelización del problema

Variables:

$X_i = \text{Posición } X \text{ del tablero } n * n \in [0, \dots, (n * n) - 1]$ que toma el valor $i \in [X, 0, .]$

Dominio:

$D_X = [X, 0, .]$ Donde X representa un disco negro, 0 un disco blanco y . sin disco

Restricciones:

- $R_1 = \text{Mismo número de discos blancos y negros en cada fila y en cada columna}$

$$\forall_{\text{columnas,filas}} \in i; B_i = N_i$$

B_i son el número de blancos en la fila o columna i

N_i son el número de negros en la fila o columna i

- $R_2 = \text{No más de dos colores iguales seguidos en una columna o fila}$

$$\forall_{\text{columnas,filas}} \in i; C_{ij} = C_{i(j+1)} \neq C_{i(j+2)}$$

- $R_3 = \text{Ninguna celda vacía}$

Esta restricción se va a satisfacer una vez que las R_1 y R_2 se hayan satisfecho



1.2 Implementación en código

El código implementado para la resolución del problema BINARIO se puede dividir en tres partes:

- Obtener los datos del estado inicial a través de un archivo,
- Crear las variables y restricciones del problema con los datos obtenidos.
- Resolver usando python-constraint e imprimir el resultado por pantalla y en un archivo

1.2.1 [parte-1.py](#)

Este archivo es usado en la resolución del problema, siendo su uso el siguiente:

```
"Uso: python parte-1.py <fichero_entrada> <fichero_salida>"
```

Donde recibe dos parámetros, el fichero de entrada que tiene la forma indicada en el enunciado.

Una vez leído el fichero de entrada, este se mete en una lista para su posterior tratamiento. A continuación se van añadiendo en orden las variables del problema en su estado inicial, según la celda sea X, O ó . , se crean las variables con su respectivos dominios.

Después, se crean las filas y columnas del tablero para su procesamiento en las restricciones, ya que es la mejor manera de comprobar ambas. Luego se crea un loop por las columnas y otro por las filas, creando las restricciones de *equal_colors* y *two_equal_colors* (*definidas en aux1.py*) respectivamente.

Por último, se llama a la función de *getSolutions()* de python-constraint para obtener todas las soluciones posibles. Se escoge la primera de todas, que será añadida junto al estado inicial en el fichero de salida; *salida.txt*

Este fichero *salida.txt* tiene la estructura pedida en el enunciado de la práctica.

1.2.2 [aux1.py](#)

En este archivo, simplemente tenemos las funciones auxiliares usadas en [parte-1.py](#).

Las funciones implementadas son:

- **print_decorated_matrix(board, size)**: Dibuja el estado inicial en un tablero decorado como se describe en la práctica.
- **make_rows(n) / make_cols(n)**: crean las filas y columnas necesarias para las restricciones.
- **equal_colors(*args)**: Correspondiente a R1, sirve de argumento en `.addConstraint(func(), vars)`
- **two_equal_colors(*args)**: Correspondiente a R2, sirve de argumento en `.addConstraint(func(), vars)`



2. Algoritmos de búsqueda

El propósito de la segunda parte de la práctica consiste en encontrar la ruta más corta entre dos puntos de alguno de los mapas de los Estados Unidos que están disponibles en la 9th DIMACS Shortest-Path Challenge.

Modelización

El problema del camino más corto sobre el mapa se define formalmente como una tupla $\langle S, s_0, G, O, c, h \rangle$:

- **Estado (S):** Cada estado $s \in S$ corresponde a un identificador único de vértice (id) presente en el grafo del mapa (ficheros .gr).
- **Estado Inicial (s_0):** El vértice de origen especificado por el usuario: start_node.
- **Test de Meta ($G(s)$):** Una función booleana que devuelve Verdadero si el estado actual s coincide con el vértice destino (goal_node).
- **Operadores (O):** Dado un estado u, los operadores generan el conjunto de sucesores $Suc(u) = \{v \mid \exists(u, v) \in E\}$, donde E son las aristas del grafo.
- **Función de Coste ($c(u, v)$):** El coste de aplicar un operador es la distancia física (peso de la arista) entre el vértice u y v, tal como se define en el fichero .gr. El coste del camino g(n) es la suma de los costes de las aristas recorridas.
- **Función Heurística ($h(n)$):** Se utiliza la distancia Haversine (distancia del gran círculo) entre las coordenadas geográficas (latitud, longitud) del nodo n y el nodo meta. Esta heurística es **admisible** ($h(n) \leq c * (n)$) porque la distancia en línea recta (o vuelo de pájaro) nunca es mayor que la distancia real por carretera.

Como nuestro objetivo es encontrar una solución óptima al problema DIMACS, tenemos que usar un algoritmo que no solo sea completo, sino que también sea admisible. Además, el problema sugiere usar heurísticas para resolverlo con un algoritmo de búsqueda informada.

Con esta definición, sólo tenemos dos opciones de algoritmos admisibles que usen heurísticas, A* e IDA*. Como el objetivo es usar una heurística admisible, por tanto no debería haber transposiciones, usaremos A* en este apartado.

Dada la magnitud de los datos (millones de nodos y aristas), el diseño se ha centrado prioritariamente en la optimización de la memoria RAM y la velocidad de ejecución, cumpliendo estrictamente con la modularización en ficheros exigida por el enunciado.

grafo.py (Estructura de Datos)

Este es el componente crítico que permite cargar el mapa completo de EE.UU. en una máquina estándar (16GB RAM) sin colapsar.

Problema detectado: El uso de diccionarios estándar de Python (dict) y listas de tuplas para representar el grafo genera una carga de memoria inasumible para el sistema. Cada



objeto en Python consume bytes adicionales de metadatos, lo que hacía inviable cargar 24 millones de nodos y 58 millones de aristas con estructuras convencionales.

Solución de diseño: Se implementó una estructura basada en arrays (módulo array)

- **Almacenamiento de Vecinos:** En lugar de objetos tupla, se usa un array lineal de enteros sin signo para almacenar secuencialmente [vecino, peso, vecino, peso...]. Esto reduce el consumo de memoria por arista drásticamente.
- **Acceso Directo:** Se sustituyeron los diccionarios por listas indexadas. El índice de la lista corresponde al ID del nodo, permitiendo acceso en tiempo $O(1)$.
- **Gestión de Coordenadas:** Las coordenadas se almacenan como enteros en arrays nativos, evitando la conversión a float (que ocupa más memoria) hasta el momento exacto en que son necesarias para el cálculo heurístico.
- **Reserva de Memoria:** Se lee la cabecera del fichero DIMACS y se reserva el espacio exacto en memoria antes de la carga. Esto elimina el coste computacional de redimensionar las estructuras millones de veces durante la lectura.

abierta.py (Lista Abierta)

Esta clase gestiona los nodos candidatos a ser expandidos por el algoritmo A*.

Decisión de diseño: Se ha encapsulado una Cola de Prioridad (Min-Heap) utilizando una implementación de heap propia.

Justificación:

Para que A* sea eficiente, es crítico obtener el nodo con el menor coste estimado ($f = g + h$) de forma rápida. Una lista normal requeriría recorrer todos los elementos para encontrar el mínimo $O(n)$, lo que haría el algoritmo extremadamente lento.

El uso de un heap permite insertar nuevos nodos y extraer el mínimo con una complejidad logarítmica $O(\log n)$, garantizando un rendimiento óptimo incluso con millones de nodos en espera.

cerrada.py (Lista Cerrada)

Esta clase almacena los nodos que ya han sido evaluados para evitar ciclos y re-evaluaciones. La clase es bastante sencilla, la lógica simplemente se basa en la creación de un array de n elementos False, que indican si un nodo ha sido expandido o no.

En el caso de expandirlo, se llama al método de meter en la lista cerrada que simplemente lo que hace es poner en True la posición del nodo n, así que la 'inserción' es O(1)

algoritmo.py (Lógica de Resolución)

Contiene la implementación del algoritmo A* y la función heurística.

Algoritmo A*: Implementación del algoritmo como en clase.



Heurística: Dado que los nodos representan coordenadas geográficas (latitud/longitud) sobre la superficie de la Tierra, la distancia Euclídea (línea recta en un plano 2D) introduciría errores significativos debido a la curvatura terrestre.

Se implementó la fórmula de **Haversine** (distancia del círculo máximo) para calcular la distancia aérea real en metros. Esto garantiza que la heurística sea **admisible**, condición necesaria para que A* encuentre siempre la solución óptima.

Para intentar que fuese algo más rápida, se han precalculado en radianes las coordenadas, ya que la función heurística tardaba mucho tiempo en comparación con un algoritmo de fuerza bruta.

parte-2.py (Orquestación e I/O)

Actúa como punto de entrada y controlador principal del sistema.

Gestión de Entrada/Salida: Se encarga de parsear los argumentos de línea de comandos y gestionar la lectura robusta de los ficheros .gr (grafo) y .co (coordenadas), manejando posibles errores de archivos no encontrados.

Doble pasada de lectura: Implementa la lógica para leer primero las cabeceras de los archivos y comunicar al objeto Grafo el tamaño necesario para reservar memoria, coordinando la optimización descrita en **grafo.py**.

Formato de Salida: Es responsable de reconstruir el camino devuelto por el algoritmo y formatearlo según el requisito estricto <nodo> - (coste) - <nodo>, lo cual implica realizar consultas al grafo para recuperar los pesos de las aristas utilizadas en la solución final.

Métricas: Calcula y presenta estadísticas de rendimiento (tiempo de ejecución, nodos expandidos, nodos por segundo) para el análisis de resultados.

pruebas.py (Validación y Análisis)

Este script adicional se diseñó como una herramienta de soporte para la fase de análisis de resultados y validación de la robustez del sistema.

La carga de grafos masivos (I/O y construcción de estructuras en memoria) consume una cantidad de tiempo significativa y fija. Ejecutar el script principal (parte-2.py) repetidamente para probar casos individuales resultaba ineficiente, ya que obligaba a recargar el grafo completo para cada consulta.

Se implementó una arquitectura de ejecución por lotes (batch processing).

- **Persistencia en Memoria:** El script carga la estructura del grafo una única vez al inicio de la ejecución.



- **Batería de Tests:** Una vez el grafo reside en la RAM, el script lanza múltiples consultas secuenciales (casos cortos, medios, largos y aleatorios) contra la misma instancia del grafo.
- **Recolección de Métricas:** Permite calcular el tiempo promedio de respuesta del algoritmo A* de forma aislada, separando el tiempo de cómputo puro del tiempo de carga de datos. Esto es fundamental para realizar el análisis de complejidad y rendimiento solicitado en la memoria del proyecto, permitiendo demostrar la eficiencia del algoritmo en cientos de consultas sin la penalización de la lectura de disco.

Conclusión

El diseño resultante prioriza la eficiencia en el uso de recursos. La combinación de estructuras de datos de bajo nivel (arrays) para el almacenamiento masivo, junto con una estructura de heap para la abierta, permite resolver instancias masivas como el mapa de USA entero (el que tiene más nodos y aristas del DIMACS) del problema, que serían imposibles de tratar con implementaciones estándar de alto nivel como listas o tuplas.

Debido a la prohibición de `heapq` y `set` para las listas, se ha visto mermado el tiempo de ejecución con implementaciones propias.

Además, la heurística que tenemos no está tan informada, por lo que no hay resultados tan exitosos como los esperados (ver a continuación). Usando distintas heurísticas para disminuir el tiempo de ejecución y mejorando la lógica de las listas abiertas y cerradas, no se ha llegado a los niveles de rapidez que otro lenguaje de programación compilado como c++ hubiera llegado.

3. Análisis de Resultados

Satisfacción de restricciones

Análisis de Complejidad del Problema BINAIRO:

Para un tablero de tamaño NxN:

- **Variables:** Se define una variable por cada celda del tablero, por lo que el número total de variables es N^2 .
- **Dominios:** Cada variable tiene un dominio máximo de 2 valores ($D = \{X, O\}$), salvo las casillas preasignadas que tienen dominio de tamaño 1.
- **Restricciones:** El modelo define restricciones sobre filas y columnas.
 - Para las filas, generamos N grupos de variables. A cada grupo se le aplican 2 restricciones (equal_colors y two_equal_colors). Total filas: 2N.
 - Para las columnas, generamos N grupos de variables. A cada grupo se le aplican 2 restricciones idénticas. Total columnas: 2N.
 - **Número total de restricciones:** 4N.

El espacio de estados crece exponencialmente con el número de variables (2^{N^2}), pero la propagación de restricciones reduce drásticamente el espacio de búsqueda efectivo.

Para analizar el modelado e implementación de la primera parte, se han usado tres entradas del juego para comprobar si resuelve con éxito el problema. Se incluirá una captura de pantalla con el fichero de salida.

Primero con un tablero de 4x4:

```
Ejecutando prueba 4x4...
Estado inicial del tablero:
+---+---+---+---+
|   | X | X |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   | X |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| 0 |   |   |   |
+---+---+---+---+
3 soluciones encontradas
```

+---+---+---+---+
0 X X 0
+---+---+---+---+
X 0 0 X
+---+---+---+---+
X 0 0 X
+---+---+---+---+
0 X X 0
+---+---+---+---+

Después con un tablero de 6x6, para comprobar que la restricción de no más de dos colores seguidos se cumple (no es posible en el 4x4)

```
Ejecutando prueba 6x6 (Ejemplo PDF)...
Estado inicial del tablero:
+---+---+---+---+---+
|   | X | O |   |   |
+---+---+---+---+---+
|   |   | X |   |   |
+---+---+---+---+---+
| O | X |   |   |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
| O |   | X |   | X |   |
+---+---+---+---+---+
|   | X |   | O |   |
+---+---+---+---+---+
12 soluciones encontradas
```

+---+---+---+---+---+---+	O X O X O X
+---+---+---+---+---+---+	X O X O O X
+---+---+---+---+---+---+	O X O X X O
+---+---+---+---+---+---+	X O X X O O
+---+---+---+---+---+---+	O O X O X X
+---+---+---+---+---+---+	X X O O X O

Y por último, un tablero de 8x8 para comprobar en tableros más grandes.

```
Ejecutando prueba 8x8...
Estado inicial del tablero:
+---+---+---+---+---+---+---+
|   | X |   | X |   | O |   |
+---+---+---+---+---+---+---+
| O | O | X |   | O | X | O |
+---+---+---+---+---+---+---+
|   |   | X | O |   |   |   |
+---+---+---+---+---+---+---+
| X |   |   | O | O |   | X |
```

5 soluciones encontradas

+---+---+---+---+---+---+---+	X X O X O O X O
+---+---+---+---+---+---+---+	O O X X O X O X
+---+---+---+---+---+---+---+	O X X O X O X O
+---+---+---+---+---+---+---+	X O O X O O X X
+---+---+---+---+---+---+---+	O O X O X X O X
+---+---+---+---+---+---+---+	O X X O X X O O
+---+---+---+---+---+---+---+	X O O X O O X X
+---+---+---+---+---+---+---+	X X O O X X O O

Búsqueda

Comparativa A vs Fuerza Bruta (Dijkstra)*

Para validar la eficiencia de la heurística, se ha implementado una versión del algoritmo de fuerza bruta (Dijkstra) desactivando la heurística ($h(n)=0$) y se ha comparado con A*.

Se puede comprobar que aunque el algoritmo A* expanda menos nodos en por lo general, tarda algo más que el algoritmo de fuerza bruta **Dijkstra**.

Todo ello se puede achacar a una heurística poco informada, ya que al probar con esta misma heurística pero multiplicada por 10, se ven mejoras de x4 a x8 en algunos trazados, pero siempre con mejora.

Los tiempos se redujeron en general una vez se aclaró que no se podía usar heapq ni set para las listas abierta y cerrada; pues después de implementar un heap casero y una lista cerrada más sencilla, seguía funcionando el algoritmo, pero algo más lento.

```
--- Cargando USA-road-d.USA ---
Leyendo coordenadas (USA-road-d.USA.co)...
Leyendo estructura del grafo (USA-road-d.USA.gr)...
Carga completada en 62.61 s.
Grafo en memoria: 23947347 nodos, 58333344 arcos.

=====
Caso Inicio -> Fin | A* Exp. A* T(s) | Dijkstra Exp. Dijk. T(s) | Mejora
=====
1 1 -> 500 | 7417 1.0893 | 10029 0.9411 | 0.9x
2 100 -> 2000 | 1065 0.9039 | 1266 0.9101 | 1.0x
3 1 -> 10000 | 454545 3.5340 | 584290 3.1320 | 0.9x
4 21455604 -> 3735651 | 3102204 17.1858 | 3102204 13.6820 | 0.8x
5 839222 -> 9228453 | 2391883 12.5353 | 2391883 10.1943 | 0.8x
6 8217208 -> 7489710 | 2668810 14.3470 | 2668810 11.1924 | 0.8x
=====
Tiempo medio A*: 8.2659 s
```

Y aquí con el más pequeño de NY:

```
--- Cargando USA-road-d.NY ---
Leyendo coordenadas (USA-road-d.NY.co)...
Leyendo estructura del grafo (USA-road-d.NY.gr)...
Carga completada en 0.70 s.
Grafo en memoria: 264346 nodos, 733846 arcos.

=====
Caso Inicio -> Fin | A* Exp. A* T(s) | Dijkstra Exp. Dijk. T(s) | Mejora
=====
1 1 -> 500 | 2856 0.0186 | 3139 0.0155 | 0.8x
2 100 -> 2000 | 827 0.0100 | 1008 0.0096 | 1.0x
3 1 -> 10000 | 44567 0.1868 | 48047 0.1584 | 0.8x
4 58370 -> 13113 | 144738 0.6832 | 150940 0.5822 | 0.9x
5 144195 -> 128394 | 86225 0.4365 | 105529 0.4480 | 1.0x
6 117027 -> 73159 | 8537 0.0387 | 9939 0.0351 | 0.9x
=====
Tiempo medio A*: 0.2290 s
```

Ahora veamos cómo se comporta la heurística cuando está más informada de manera artificial (multiplicandola por 10)



```
--- Cargando USA-road-d.NY ---
Leyendo coordenadas (USA-road-d.NY.co)...
Leyendo estructura del grafo (USA-road-d.NY.gr)...
Carga completada en 0.64 s.
Grafo en memoria: 264346 nodos, 733846 arcos.

=====
Caso Inicio -> Fin | A* Exp. A* T(s) | Dijkstra Exp. Dijk. T(s) | Mejora
=====
1 1 -> 500 | 736 0.0107 | 3139 0.0155 | 1.4x
2 100 -> 2000 | 127 0.0089 | 1008 0.0113 | 1.3x
3 1 -> 10000 | 24745 0.1261 | 48047 0.1534 | 1.2x
4 58370 -> 13113 | 40869 0.2344 | 150940 0.5653 | 2.4x
5 144195 -> 128394 | 9824 0.0584 | 105529 0.4351 | 7.5x
6 117027 -> 73159 | 3268 0.0202 | 9939 0.0353 | 1.7x
=====
Tiempo medio A*: 0.0765 s
```

Y con USA

```
--- Cargando USA-road-d.USA ---
Leyendo coordenadas (USA-road-d.USA.co)...
Leyendo estructura del grafo (USA-road-d.USA.gr)...
Carga completada en 62.72 s.
Grafo en memoria: 23947347 nodos, 58333344 arcos.

=====
Caso Inicio -> Fin | A* Exp. A* T(s) | Dijkstra Exp. Dijk. T(s) | Mejora
=====
1 1 -> 500 | 930 0.9508 | 10029 0.9335 | 1.0x
2 100 -> 2000 | 318 0.9009 | 1266 0.9047 | 1.0x
3 1 -> 10000 | 27638 1.4758 | 584290 3.1042 | 2.1x
4 21455604 -> 3735651 | 3102204 17.7793 | 3102204 13.6304 | 0.8x
5 839222 -> 9228453 | 2391883 13.5501 | 2391883 10.0713 | 0.7x
6 8217208 -> 7489710 | 2668810 16.0943 | 2668810 11.2916 | 0.7x
=====
Tiempo medio A*: 8.4586 s
```

Podemos ver aquí que se han expandido los mismo nodos que Dijkstra, por tanto tarda más ya que hay que sumar el factor de cálculo de la función heurística.

Sabemos que no sobreestima ya que como mucho la expansión de nodos es la misma