Práctica: *Satisfacción de Restricciones y Búsqueda Heurística.*
Heurística y Optimización.
Curso 2025-2026

1. Objetivo

El objetivo de esta práctica es que el alumno aprenda a modelar y resolver tareas de satisfacción de restricciones, así como aprender a modelar e implementar eficientemente la resolución de problemas de búsqueda heurística.

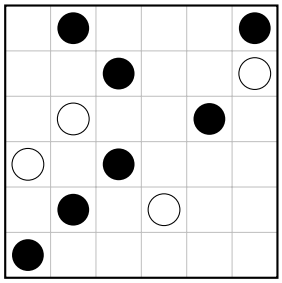
2. Enunciado del problema

La práctica consiste en tres partes, detalladas a continuación, cuya solución debe describirse técnicamente de acuerdo a las indicaciones de la sección 3, y que tienen pesos y valoraciones diferentes, tal y como se indica en la sección 4. La práctica *debe* realizarse en parejas.

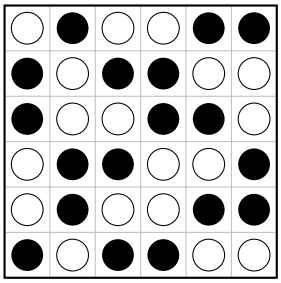
2.1. Parte 1: Satisfacción de restricciones

BINAIRO es un pasatiempo que se juega sobre una rejilla cuadrangular de dimensions $n \times n$, 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.



(a) Instancia a resolver en 6×6



(b) Solución de la misma instancia

Figura 1: Ejemplo de un problema de BINAIRO, a la izquierda, junto con una posible solución, mostrada a la derecha.

Inicialmente, algunas casillas pueden estar, o no, ya ocupadas con discos blancos, o negros, tal y como se puede ver en el ejemplo de la figura 1a, cuya solución se muestra en la figura 1b.

Se pide implementar un *script* en Python 3.8 (o posterior) llamado `parte-1.py` que resuelva cualquier instancia de BINAIRO usando `python-constraint`. El *script* debe recibir exactamente dos argumentos: `fichero-entrada`, y `fichero-salida`. A continuación se muestran diferentes casos de uso:

```
$ ./parte-1.py ejemplo.in ejemplo.out
$ ./parte-1.py /tmp/ejemplo.in ejemplo.out
$ ./parte-1.py ejemplo.in ../../datos/ejemplo.out
$ ./parte-1.py /tmp/ejemplo.in ../../datos/ejemplo.out
```

Como se ve, cada fichero puede indicarse con una ruta absoluta o relativa, y si no contiene ninguna entonces se asume aquella en la que se encuentra el *script*, que debe:

- Leer el fichero de entrada indicado en primer lugar que debe consistir en n líneas, cada una de las cuales contiene n símbolos que sólo pueden ser::
  - '`.`': indica que el contenido de esa casilla es desconocido.
  - '`X`': indica que el contenido de esa casilla es un disco negro.
  - '`O`': indica que el contenido de esa casilla es un disco blanco.
- Resolver el problema de satisfacción de restricciones especificado en el fichero de entrada, y mostrar la solución en el fichero indicado de salida mostrando, primero la instancia a resolver, e inmediatamente después una solución del problema.

  Además, el *script* `parte-1.py` debe mostrar en pantalla la instancia a resolver, y a continuación el número de soluciones encontradas.

Considérense, por ejemplo, los siguientes contenidos para el fichero de entrada `ejemplo.in`:

```
.X...X
..X..O
.O..X.
O.X...
.X.O..
X.....
```

de modo que la ejecución del *script* podría producir una salida como la siguiente:

```
+---+---+---+---+---+---+
|   | X |   |   |   | X |
|   |   | X |   |   | O |
|   | O |   |   | X |   |
| O |   | X |   |   |   |
|   | X |   | O |   |   |
| X |   |   |   |   |   |
+---+---+---+---+---+---+
9 soluciones encontradas
```

y, además, debe escribir *una* solución en el fichero indicado de salida, `ejemplo.out`, precedida de la instancia resuelta. Una solución factible podría ser:

```
+---+---+---+---+---+---+
|   | X |   |   |   | X |
|   |   | X |   |   | O |
|   | O |   |   | X |   |
| O |   | X |   |   |   |
|   | X |   | O |   |   |
| X |   |   |   |   |   |
+---+---+---+---+---+---+
+---+---+---+---+---+---+
| O | X | O | O | X | X |
| X | O | X | X | O | O |
| X | O | O | X | X | O |
```

```
| O | X | X | O | O | X |
| O | X | O | O | X | X |
| X | O | X | X | O | O |
+---+---+---+---+---+---+
```

donde se ha tenido cuidado en delimitar cada problema (la instancia original, y su solución), para poder reconocerlas ambas fácilmente.

2.2. Parte 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[1]. La especificación de cada grafo consiste en dos ficheros con sufijo `.gr` y `.co`:

- El primero describe un grafo *dirigido* indicando los arcos entre dos pares cualquiera de vértices identificados inequívocamente por su índice, con la distancia entre ellos sobre la superficie de la Tierra en metros, y que se especifican con líneas que tienen el formato:

  `a <id1><id2><coste>`

  siendo posible ignorar el resto de las líneas.

- El segundo contiene la longitud y latitud de cada vértice sobre la superficie de la Tierra multiplicados por 10^6, en líneas que tienen el formato:

  `v <id><longitud><latitud>`

  siendo posible ignorar el resto de las líneas.

Por ejemplo, las líneas 8 y 9 del fichero `USA-road-d.BAY.gr` son:

```
a 1 2 1988
a 2 1 1988
```

que indica que desde el vértice `1` hasta el `2` hay un arco con una distancia de 1.988 metros e, igualmente, es posible ir desde el segundo vértice hasta el primero, recorriendo una distancia también igual a 1.988 metros.

El fichero de coordenadas especifica, también en las líneas 8 y 9, que la longitud y latitud de cada uno de estos puntos es:

```
v 1 -121745853 37608914
v 2 -121745591 37610691
```

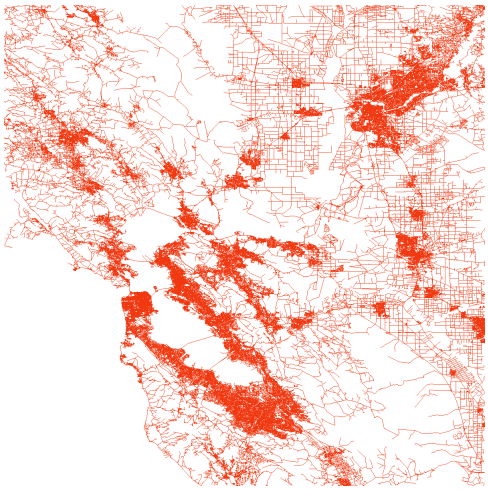
es decir, el vértice `1` se encuentra en (37,608914, -121,745853), mientras que el vértice 2 se encuentra en (37,610691, -121,745591) —donde se han invertido intencionadamente la longitud y latitud para expresar puntos sobre la superficie de la Tierra de acuerdo al convenio internacional.

Las figura 2 muestra los dos grafos más pequeños de la 9th DIMACS Shortest-Path Challenge con 264.346 y 321.270 vértices cada uno, respectivamente.

Se pide implementar un *script* en Python 3.8 (o posterior) llamado `parte-2.py` que resuelva cualquier instancia del camino más corto entre dos puntos cualesquiera sobre cualquiera de los mapas de la 9th DIMACS Shortest-Path Challenge. El *script* debe recibir exactamente cuatro argumentos: `vertice-1` `vertice-2` `nombre-del-mapa`, y `fichero-salida`. A continuación se muestran diferentes casos de uso:

[1] Ver `https://www.diag.uniroma1.it/~challenge9/download.shtml` en las columnas *Distance graph* y *Coordinates*.

(a) Mapa de New York (`NY`)



(b) Mapa de Bay Area (`BAY`)

Figura 2: Vistas de los dos grafos más pequeños de la 9th DIMACS Shortest-Path Challenge

```
$ ./parte-2.py 1 309 USA-road-d.BAY solucion
$ ./parte-2.py 1 309 /tmp/USA-road-d.BAY solucion
$ ./parte-2.py 1 309 USA-road-d.BAY ../../datos/solucion
$ ./parte-2.py 1 309 /tmp/USA-road-d.BAY ../../datos/solucion
```

Nótese que el nombre `USA-road-d.BAY` identifica inequívocamente a los ficheros de grafo, `USA-road-d.BAY.gr`, y de coordenadas, `USA-road-d.BAY.co` que deben estar en el mismo directorio. Como se ve, cada fichero puede indicarse con una ruta absoluta o relativa, y si no contiene ninguna entonces se asume aquella en la que se encuentra el *script*, que debe:

- Leer los ficheros de entrada de grafo y coordenadas indicado en tercer lugar, y que tienen el formato indicado anteriormente.
- Resolver el problema del camino más corto desde el vértice indicado en primer lugar, hasta el vértice indicado en segundo lugar, y escribir la solución en el fichero indicado en cuarto lugar con el siguiente formato:

  ```
  <inicio> - ... - <vertice-i> - (coste arco <i,i+1>) - <vértice-i+1> - ... - <final>
  ```

  Además, el *script* `parte-2.py` debe mostrar en pantalla la siguiente información: número de vértices procesados del fichero de coordenadas, número de arcos procesados del fichero de grafo, coste de la sólución óptima encontrada y, a continuación, el número de nodos expandidos y el tiempo dedicado para resolver el problema.

Por ejemplo, la ejecución:

```
$ ./parte-2.py 1 309 USA-road-d.BAY solucion
```

podría mostrar en pantalla la siguiente información:

```
# vertices: 321270
# arcos   : 800172
Solución óptima encontrada con coste 10216

Tiempo de ejecución: 0.15 segundos
# expansiones      : 4 (25.12 nodes/sec)
```

y el fichero de salida `solucion` tendría los contenidos siguientes:

```
1 - (1498) - 308 - (8718) - 309
```

de modo que la suma de los costes de los arcos, 1498 + 8718, es exactamente igual al coste de la solución óptima encontrada, 10216, mostrada en la salida del *script*.

Para resolver el problema se pide elegir un algoritmo de búsqueda, e implementar una heurística que resuelva el problema *óptimamente* expandiendo menos nodos que el mismo algoritmo basado en fuerza bruta, que también debes implementar, y documentar en el análisis de resultados —ver sección 2.3. La implementación del *script* `parte-2.py` debe contener la implementación más eficiente posible usando, para ello, la mejor selección de algoritmo, estructuras de datos y heurística que sea posible. Se exige, en particular, que la implementación de esta parte contenga al menos los siguientes ficheros:

- `parte-2.py`: *script* principal que resuelve el problema.
- `grafo.py`: definición de una clase que sirva para leer los contenidos de un grafo y representarlos eficientemente en memoria.
- `abierta.py`: definición de una clase con la implementación de la lista abierta elegida, si tu algoritmo la necesitara.
- `cerrada.py`: definición de una clase con la implementación de la lista cerrada elegida, si tu algoritmo la necesitara.
- `algoritmo.py`: definición de una clase que resuelve *óptimamente* la instancia indicada en los parámetros del *script* `parte-2.py`.

2.3. Parte 3: Análisis de Resultados

En este apartado se deben analizar todos los resultados obtenidos de cada parte, describiendo la solución obtenida (comprobando que cumple con las restricciones del enunciado) y analizando cómo progresa el tiempo de ejecución en función de las variables del problema de entrada. ¿Qué explica el crecimiento del tiempo de ejecución? ¿Cuál es la instancia más grande que resuelve tu algoritmo?

Análisis de la complejidad del problema:

1. En relación con la primera parte, ¿cuántas variables y restricciones has definido en la primera parte?
2. En relación con la segunda parte, ¿cómo crece el tamaño de las estructuras de datos definidas en la segunda parte? ¿Qué reducción es posible en el número de nodos expandidos y el tiempo de ejecución en la segunda parte gracias al uso de una heurística? ¿Qué algoritmo has elegido como el mejor para resolver este problema?

Crea diferentes casos y demuestra su resolución con el uso de tus *scripts*, variando los parámetros, y explica cómo estas modificaciones afectan a la dificultad de resolución del problema resultante.

3. Directrices para la Memoria

La memoria debe entregarse en formato .pdf y tener un máximo de 15 hojas en total, incluyendo la portada, contraportada e índice, si los tuviera. **Debe tratarse de un documento técnico**, en vez de un manual de usuario que contenga, al menos:

1. Breve introducción explicando los contenidos del documento.
2. Descripción de los modelos, argumentando las decisiones tomadas, usando para ello una notación algebraica correcta, clara y concisa.
3. Análisis de los resultados, según se indica en la sección 2.3.

La memoria **no debe incluir código fuente** en ningún caso.

4. Evaluación

La evaluación de la práctica se realizará sobre 10 puntos. Para que la práctica sea evaluada deberá realizarse al menos la primera parte de la práctica:

1. Parte 1 (3 puntos)
    - Modelización del problema (1 punto).
    - Implementación del modelo (1 puntos).
    - Resolución y análisis de los casos de prueba (1 punto).
2. Parte 2 (7 puntos)
    - Modelización del problema (2 punto).
    - Implementación del algoritmo (3 puntos).
    - Resolución de diferentes casos de prueba y análisis de resultados (2 puntos).

En la evaluación de la modelización del problema, un modelo correcto supondrá la mitad de los puntos. Para obtenerse el resto de puntos, la modelización del problema deberá:

- Ser formalizada correctamente en la memoria.
- Ser, preferiblemente, sencilla y concisa.
- Estar bien explicada (ha de quedar clara cuál es la utilidad de cada variable/restricción).
- Justificarse en la memoria todas las decisiones de diseño tomadas.

En la evaluación de la implementación del modelo, un modelo correcto supondrá la mitad de los puntos. Para obtenerse el resto de puntos, la implementación del problema deberá:

- Corresponderse íntegramente con el modelo propuesto en la memoria.
- Entregar código fuente correctamente organizado y comentado. Los nombres deben ser descriptivos. Deberán añadirse comentarios en los casos en que sea necesario mejorar la legibilidad y comprensión.
- Contener casos de prueba que muestren diversidad para la validación de la implementación.

Importante: los modelos implementados deben ser correctos. Esto es, han de funcionar y obtener soluciones óptimas a cada problema propuesto. En ningún caso se obtendrá una calificación superior a 1 punto por un modelo que no lo haga.

IA generativa: el uso de herramientas de Inteligencia Artificial para realizar la práctica está expresamente prohibido, si bien es posible usarlas con el propósito de mejorar o razonar sobre modelos desarrollados previamente.
En particular, la sospecha del uso fraudulento de herramientas de este tipo, o cualquier otra consideración que anime a creer que algún miembro de un equipo, o ambos, no han realizado la práctica íntegramente, es motivo suficiente **para hacer una evaluación oral paralela a la corrección de esta práctica**.

5. Entrega

Se tiene de plazo para entregar la práctica hasta el 18 de diciembre de 2025, Jueves, a las 23:55. Este límite es fijo y no se extenderá de ningún modo. Ten en cuenta que los entregadores de cada grupo reducido han sido configurados para poder sobreescribir versiones anteriores.

Sólo un miembro de cada pareja de estudiantes debe subir:

- Un único fichero `.zip` a través del punto de entrega de 'Aula Global' llamado "*Segunda práctica*".

  El fichero debe nombrarse `p2-NIA1-NIA2.zip`, donde `NIA1` y `NIA2` son los últimos 6 dígitos del NIA (rellenando con `0`s por la izquierda si fuera preciso) de cada miembro de la pareja.

  Ejemplo: `p2-054000-671342.zip`.

- La memoria debe subirse separadamente a través del punto de entrega Turnitin de 'Aula Global' llamado "*Segunda práctica (sólo pdf)*".

  La memoria debe llamarse `NIA1-NIA2.pdf` —después de sustituir adecuadamente los NIAs de cada estudiante como en el caso anterior.

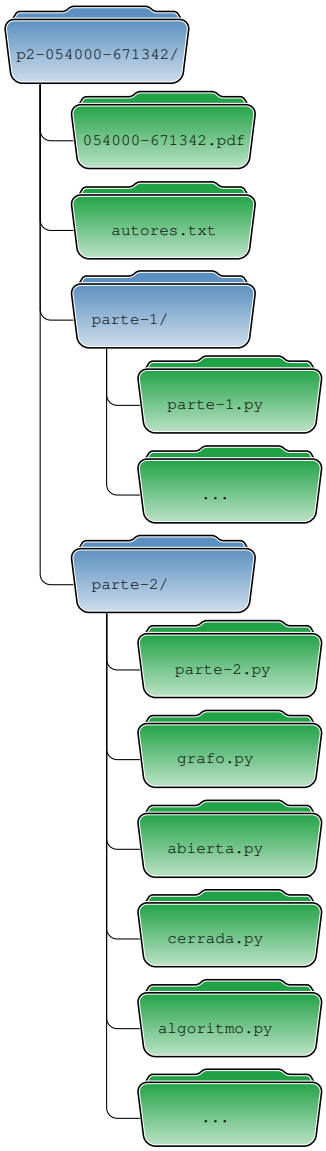
  Ejemplo: `054000-671342.pdf`.

La descompresión del fichero `.zip` descrito en el primer punto debe producir un directorio llamado `p2-NIA1-NIA2`, donde `NIA1` y `NIA2` son los últimos 6 dígitos del NIA de cada estudiante rellenando con `0`s si fuera preciso. Este directorio debe contener: primero, la misma memoria entregada a través del segundo enlace descrito anteriormente que debe nombrarse como `NIA1-NIA2.pdf` —después de sustituir convenientemente los NIAs de cada estudiante; segundo, un fichero llamado `autores.txt`, codificado en UTF-8, que identifica a los miembros de cada pareja, con una línea por cada uno con el siguiente formato: `NIA Apellidos, Nombre`. Por ejemplo:

```
054000 Von Neumann, John
671342 Turing, Alan
```

Además, este directorio debe contener un directorio por cada parte realizada llamados "`parte-1`" y "`parte-2`". Las soluciones de cada parte deben incluirse en sus respectivos directorios.

La siguiente figura muestra una distribución posible de los ficheros después de descomprimir el fichero `.zip`:



Como se puede ver, los directorios "`parte-1`" y "`parte-2`" pueden contener otros ficheros adicionales (indicados en la figura con puntos suspensivos). Son ejemplos válidos: ficheros adicionales de Python necesarios para la ejecución de los *scripts*, o ficheros de datos de ejemplo de casos resueltos que se discutan en la memoria.

Importante: no seguir las normas de entrega puede suponer una pérdida de hasta 1 punto en la calificación final de la práctica.