uc3m | Universidad Carlos III de Madrid

Heurística y Optimización

Práctica 2: Satisfacción de Restricciones y Búsqueda Heurística

Inés Fuai Guillén Peña | 100495752@alumnos.uc3m.es | g85

Nicola Stefania Cindea | 100495713@alumnos.uc3m.es | g85

uc3m | Universidad Carlos III de Madrid

ÍNDICE

1. Introducción

En este proyecto nos enfocamos en resolver el tema del mantenimiento de una flota de aviones. La aerolínea hace todo el mantenimiento en un solo día, organizando las tareas por franjas horarias. Para que sea más práctico, dividimos la flota en dos tipos de aviones: los estándar y los jumbo. Igual pasa con los talleres, que están clasificados en dos tipos: los estándar y los especializados.

El reto está en asignar los aviones a los talleres respetando una serie de restricciones. Estas restricciones incluyen la capacidad de cada taller (no pueden trabajar con más aviones de los que soportan al mismo tiempo), el tipo de tareas que pueden hacer, el orden en el que se tienen que realizar esas tareas, y también que haya suficiente espacio para que los aviones puedan moverse sin problemas. Además, tenemos que planificar el rodaje de los aviones dentro del aeropuerto, o sea, cómo se van a mover desde sus posiciones iniciales hasta las pistas de despegue sin chocar entre ellos. Para esto, planteamos el problema como una búsqueda heurística, tratando de minimizar el makespan. Usamos el algoritmo A* y probamos con varias heurísticas admisibles para asegurarnos de que las soluciones sean válidas y eficientes.

Para abordar todo esto, usamos herramientas como Python, programación por restricciones y búsqueda heurística. Estas nos ayudan a modelar tanto el tema del mantenimiento como el del rodaje y a encontrar las mejores soluciones cumpliendo todas las condiciones. Así podemos evaluar qué tan viables son los modelos y qué tan bien funcionan las técnicas que aplicamos.

2. Parte 1: Implementación del modelo

Variables. Posición de cada avión por franja horaria en la matriz de talleres.

- *Ai,j*: Representa el taller o parking asignado al avión *i* en la franja horaria *j*

Dominio. Posibles asignaciones de los aviones en cada franja horaria.

- *Talleres estándar (STD):* tareas de mantenimiento tipo 1 → (0,1),(1,0),(1,1),(1,2),(1,3),(2,0),(2,2),(3,3),(4,1),(4,2)
- *Talleres especialistas (SPC):* tareas de mantenimiento tipo 1 y 2 → (0,3),(2,1),(2,3),(3,0),(4,3)
- *Parkings (PRK):* almacenar aviones → (0,0),(0,2),(0,4),(1,4),(2,4),(3,1),(3,2),(3,4),(4,0),(4,4)

Restricciones y modelados.

1. **Restricción 1:** Cada avión debe estar asignado a una única posición en la matriz en cada franja horaria. Modelado → $\forall i,\ \forall j,\ Ai,j \in DOMINIO$

2. **Restricción 2:** Cada taller puede atender a un máximo de 2 aviones por franja horaria, pero en ningún caso podrá haber más de un avión JUMBO en la misma franja horaria en el mismo taller. Modelado:
   - Para cada taller *t* y franja *j*: count(*Ai,j* = *t*) ≤ 2
   - Si hay un avión JUMBO asignado: count(*Ai,j=t y tipo(i)* = *JUMBO*) ≤ 1

3. **Restricción 3:** Un taller especialista puede realizar tareas de tipo 1 y tipo 2, pero un taller estándar solo puede realizar tareas de tipo 1. Modelado:
   - Si el avión *i* tiene una tarea de tipo 2 en *j*, entonces: *Ai,j* ∈ *SPC*
   - Si el avión *i* tiene tareas de tipo 1 y tipo 2, primero debe completarse la tarea de tipo 2: *Ai,j2* ∈ *SPC, j2 < j1* donde *j1* corresponde a la tarea tipo 1

4. **Restricción 4:** Orden de las tareas de mantenimiento (todas las tareas de tipo 2 se realizan antes que todas las tareas de tipo 1). Modelado:
    - Las tareas de tipo 2 deben realizarse consecutivamente en talleres especialistas → Si *Ti,1,Ti,2,…,Ti,m* son las tareas de tipo 2, entonces: $\forall j \in [1, m],\ Ai, j \in SPC$. Estas tareas deben completarse en franjas horarias consecutivas: $j2 = j1 + 1,\ j3 = j2 + 1$
    - Las tareas de tipo 1 sólo pueden realizarse después de completar todas las tareas de tipo 2 → Si *Ti,m+1,…,Ti,m+n* son las tareas de tipo 1, entonces: $\forall j \in [m + 1, m + n],\ Ai, j \in STD\ U\ SPC$
    - No puede haber alternancia entre tareas de tipo 2 y tipo 1: Si *Ti,j* es de tipo 2, entonces cualquier tarea posterior de tipo 1 debe comenzar después de la última tarea de tipo 2: *j1 < j2* ⟹ *Ti,j1* es tipo 2, *Ti,j2* es tipo 1
    - El avión no puede asignarse a un parking (PRK) si tiene tareas pendientes. Durante franjas horarias asignadas para tareas:$\forall j \in [1, m + n],\ Ai, j \notin PRK$

5. **Restricción 5:** Al menos uno de los talleres o parkings adyacentes (vertical y horizontal) debe estar vacío.
   Modelado: Para cada franja *j*, si *Ai,j = (x,y)* →
   $\rightarrow \forall (x', y') \in \{(x - 1, y),\ (x + 1, y), (x, y - 1), (x, y + 1)\}, Ak, j \neq (x', y')$

6. **Restricción 6:** Dos aviones JUMBO no pueden estar asignados a talleres adyacentes en la misma franja horaria. Modelado: Para cada franja *j*, si $Ai, j = (x, y)$ y $tipo(i) = JUMBO$ $\rightarrow \forall k,\ Ak, j, \neq (x',\ y') \in \{(x - 1, y),\ (x + 1, y), (x, y - 1), (x, y + 1)\}$

Implementación en Python: Para implementar el modelo CSP en Python y resolverlo, usamos una librería llamada python-constraint.

El objetivo principal de este modelado es asignar una posición a cada avión en cada franja horaria siguiendo una serie de restricciones. Estas posiciones pueden ser: un parking, un taller estándar o un taller especialista.

La arquitectura del código se divide en 3 partes: la lectura y parseo del archivo de entrada (donde recibimos toda la información necesaria para solucionar el problema), la creación del modelo CSP, y la escritura de los resultados en un archivo csv.

LECTURA Y PARSEO

parse_input(file_path): esta función recibe el archivo en el que se encuentran los datos y lo abre en forma de lectura. Comprueba que el archivo tenga mínimo 5 líneas y devuelve los 6 argumentos que se usarán en el modelo csp (el número de franjas horarias, el tamaño de la matriz, las posiciones de los talleres estándar, las posiciones de los talleres especialistas, las posiciones de los parkings y los aviones con sus correspondientes datos).

parse_positions(line): extrae las coordenadas en formato (x,y) y las devuelve como una lista de tuplas de números enteros para que sea más fácil de manejar.

parse_planes(line): realiza algo parecido pero con cada línea de los aviones. Divide las líneas en partes separadas por un guión y devuelve la lista de los aviones y sus datos (id del avión, tipo, restricción, tiempo para mantenimiento estándar y tiempo de mantenimiento especializado).

MODELO CSP

model_csp(): recibe los 6 parámetros anteriores y crea variables para cada avión por franja horaria. Asigna un dominio para cada variable y se asegura de que todas las restricciones se cumplan. Primero se asegura de que en cada franja horaria los aviones estén en posiciones diferentes.

- validar_capacidad(*positions): se asegura de que haya un máximo de dos aviones por taller agrupando las posiciones y añadiendolas a un diccionario de talleres. Una vez hecho, se asegura de que devuelva true si hay dos o menos aviones por taller y false si hay más de 2. También se encarga de que no haya más de un avión JUMBO por taller.

Nos aseguramos de que un taller especialista pueda realizar tareas tanto del tipo 1 como del 2, mientras que un taller estándar solo puede realizar tareas del tipo 1.

- ordenar_tareas(*args): llamamos a esta función para ordenar las tareas y que la restricción 4 se cumpla. Las tareas de tipo 2 tienen preferencia sobre las tareas de tipo 1.
- validar_maniobrabilidad(pos, *adj_pos): esta función se encarga de que los aviones no ocupen posiciones adyacentes si tienen restricciones de maniobrabilidad.
- no_jumbos_adyacentes(*positions): se encarga de que la última restricción se cumpla y no haya dos aviones jumbos en talleres adyacentes.

ESCRITURA DE RESULTADOS

Utilizamos una función (write_output) para escribir las soluciones generadas por el modelo CSP en un archivo de salida.

2. Parte 1: Resolución y análisis de los casos de prueba

Caso 1: Un solo avión

Input:

```
Franjas: 5
2x2
STD: (1,1)
SPC: (0,0)
PRK: (0,1) (1,0)
1-STD-T-0-3
```

Output:

```
N. Sol: 16
Solucion 1:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(1, 0), PRK(1, 0)
Solucion 2:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(1, 0), PRK(0, 1)
Solucion 3:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(1, 0), SPC(0, 0)
Solucion 4:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(1, 0), STD(1, 1)
Solucion 5:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(0, 1), PRK(1, 0)
Solucion 6:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(0, 1), PRK(0, 1)
Solucion 7:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(0, 1), SPC(0, 0)
Solucion 8:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(0, 1), STD(1, 1)
Solucion 9:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(1, 0)
Solucion 10:
1-STD-T-0: SPC(0, 0), SPC(0, 0), SPC(0, 0), SPC(0, 0), PRK(0, 1)
```

Las primeras franjas se asignan a talleres especialistas (SPC), y las últimas a parkings (PRK) o talleres estándar (STD). Las variaciones en las soluciones se deben a las diferentes combinaciones posibles para las asignaciones de las últimas franjas, respetando las reglas de capacidad y restricciones del problema.

Caso 2: 2 aviones

Input:

```
Franjas: 2
3x3
STD: (0,1) (1,0) (1,1)
SPC: (2,1)
PRK: (0,0)
1-STD-F-1-0
2-JMB-T-0-1
```

Output:

```
N. Sol: 80
Solucion 1:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), SPC(2, 1)
Solucion 2:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(1, 1)
Solucion 3:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(1, 0)
Solucion 4:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(0, 1)
Solucion 5:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), PRK(0, 0)
Solucion 6:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), STD(1, 1)
Solucion 7:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), STD(1, 0)
Solucion 8:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), STD(0, 1)
Solucion 9:
1-STD-F-1: PRK(0, 0), STD(1, 1), 2-JMB-T-0: SPC(2, 1), SPC(2, 1)
Solucion 10:
1-STD-F-1: PRK(0, 0), STD(1, 1), 2-JMB-T-0: SPC(2, 1), PRK(0, 0)
```

En el caso 2 observamos que obtenemos 80 soluciones válidas. El número de franjas es 2 lo que restringe el número total de combinaciones. El avión 2-JMB-T-0-1 debe estar asignado al taller especializado (2,1) mientras que el avión 1-STD-F-1-0 puede ocupar tanto talleres estándar como el parking en diferentes combinaciones.

Caso 3: 3 aviones

Input:

```
Franjas: 2
3x3
STD: (0,1) (1,0) (1,1) (2,0)
SPC: (2,1)
PRK: (0,0) (0,2)
1-STD-F-1-0
2-JMB-T-0-1
3-STD-F-1-1
```

Output:

```
N. Sol: 5544
Solucion 1:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), SPC(2, 1), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 2:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(2, 0), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 3:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(1, 1), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 4:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(1, 0), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 5:
1-STD-F-1: PRK(0, 0), PRK(0, 0), 2-JMB-T-0: SPC(2, 1), STD(0, 1), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 6:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), PRK(0, 0), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 7:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), STD(2, 0), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 8:
1-STD-F-1: PRK(0, 0), SPC(2, 1), 2-JMB-T-0: SPC(2, 1), STD(1, 1), 3-STD-F-1: PRK(0, 2), PRK(0, 2)
Solucion 9:
```

En el caso de que haya 3 aviones se observa un aumento significativo en el número de soluciones. En este caso el avión 2-JMB-T-0-1 siempre ocupa (2,1) en al menos una franja horaria cumpliendo así con la restricción de uso del taller especializado. El avión 1-STD-F-1-0 alterna entre talleres estándar y parkings en ambas franjas y el avión 3-STD-F-1-1 sigue una lógica similar al avión 1.

El modelo genera todas las combinaciones válidas posibles, respetando las restricciones de capacidad, exclusividad y uso de talleres especializados. El número de soluciones refleja la complejidad creciente del problema al aumentar el número de aviones.

3. Parte 2: Implementación del algoritmo

Para modelar este problema, se deben representar varios aspectos clave: el estado del sistema, las acciones disponibles, los costos asociados y la forma en que evaluamos las soluciones. Esto permite diseñar un modelo estructurado que capture las interacciones necesarias para evitar colisiones y gestionar el movimiento eficiente de los aviones.

El **estado** del problema se define por la posición de cada avión en el mapa del aeropuerto y el tiempo transcurrido. Formalmente, se representa como $S = (P1,P2,\ldots,Pn,T)$, donde $P1,P2,\ldots,Pn$ son las posiciones actuales de los aviones $A1,A2,\ldots,An$, cada una expresada como una coordenada (x,y), y T es el tiempo total transcurrido hasta ese momento. Este modelo proporciona una descripción completa de la situación actual del sistema en un momento dado.

Las **acciones** disponibles para los aviones son simples: pueden moverse a una celda adyacente (en dirección vertical u horizontal) siempre que la celda de destino no esté ocupada ni bloqueada, o pueden esperar en su posición actual. Cada acción se representa como un par $a = (Pi,T)$, donde Pi es la nueva posición del avión tras realizar la acción y T el tiempo asociado al movimiento o espera. Estas acciones permiten modelar todas las posibles decisiones que los aviones pueden tomar en cada instante.

La **función de transición** $\delta(S,a)$ describe cómo evoluciona el sistema al aplicar una acción a en un estado S. Esta función genera un nuevo estado S', lo que permite simular los cambios en el sistema a medida que los aviones se mueven o esperan. De esta manera, se pueden explorar secuencias de acciones que conduzcan a soluciones óptimas.

En cuanto a la **función de costo**, cada movimiento o espera incurre en un costo de una unidad de tiempo. Este modelo simplificado garantiza que el tiempo sea la única métrica considerada, lo que es consistente con el objetivo de minimizar el tiempo total necesario para que todos los aviones lleguen a sus destinos.

El **objetivo** del problema es minimizar el makespan, es decir, reducir el tiempo total T hasta que el último avión llegue a su destino sin colisiones. Esto asegura que el sistema opere de manera eficiente y segura.

El **espacio de búsqueda** del problema incluye todos los posibles estados de los aviones en el aeropuerto a lo largo del tiempo. Para explorar este espacio de manera eficiente, hemos empleado dos heurísticas admisibles. La heurística basada en la distancia de Manhattan calcula la distancia mínima entre la posición actual de un avión y su destino, considerando solo movimientos horizontales y verticales. Esto garantiza que nunca sobreestima el costo real. Por otro lado, la heurística máximo de Manhattan toma el valor máximo entre las distancias de Manhattan de todos los aviones a sus destinos. Esta variante también es admisible y asegura que el avión más lejano reciba prioridad en la estimación.

Implementación en Python: Este código tiene la intención de resolver un problema de planificación de trayectorias de aviones utilizando el algoritmo A*. Lo que hace es leer un archivo

CSV con la información sobre la ubicación de los aviones y sus destinos, y luego calcula el mejor camino para cada avión, asegurándose de que no se crucen entre sí.

Primero, la función *leer_csv* es bastante clara. Lee el archivo y separa las posiciones iniciales, finales y el mapa. No es tan difícil de entender, pero hay que tener cuidado con los datos en el archivo, porque si algo no está bien formado, puede haber errores. Luego, las heurísticas son dos opciones bastante comunes: la distancia Manhattan y el máximo de Manhattan. En general, las heurísticas están bien implementadas y ayudan a guiar la búsqueda de manera eficiente. Sin embargo, elegir entre una u otra puede influir en el rendimiento dependiendo del tipo de mapa y las distancias entre los aviones. El algoritmo A* utiliza una cola de prioridad para expandir los nodos según el costo acumulado y la heurística, lo cual es eficiente. El código para expandir los movimientos posibles de los aviones y generar todas las combinaciones de movimientos es algo complejo y podría optimizarse. Actualmente, está evaluando todas las combinaciones de movimientos de los aviones, lo cual es costoso cuando el número de aviones crece.

Cabe mencionar que no se ha podido implementar una restricción adicional en la que un avión no pueda pasar a una celda inmediatamente después de que otro avión haya estado en ella; es decir, un avión debe esperar antes de ocupar una celda que acaba de ser utilizada por otro avión.

En cuanto a los archivos de salida, el manejo de los resultados está bien. Las trayectorias calculadas son almacenadas adecuadamente, lo que facilita la revisión y análisis de las soluciones generadas.

3. Parte 2: Resolución de casos de prueba y análisis comparativo

Caso 1: Mapa simple sin obstáculos

Input:

```
2
(0,0) (3,3)
(3,3) (0,0)
B;B;B;B
B;B;B;B
B;B;B;B
B;B;B;B
```

Estadísticas: h1

```
Tiempo total: 0.01s
Makespan: 6
h inicial: 12
Nodos expandidos: 170
```

h2

```
Tiempo total: 0.01s
Makespan: 6
h inicial: 6
Nodos expandidos: 158
```

Output: h1

```
(0,0) → (0,1) → (0,2) → (0,3) ↓ (1,3) ↓ (2,3) ↓ (3,3)
(3,3) ↑ (2,3) ↑ (1,3) ← (1,2) ↑ (0,2) ← (0,1) ← (0,0)
```

h2

```
(0,0) → (0,1) → (0,2) → (0,3) ↓ (1,3) ↓ (2,3) ↓ (3,3)
(3,3) ↑ (2,3) ↑ (1,3) ← (1,2) ↑ (0,2) ← (0,1) ← (0,0)
```

Con el input proporcionado y el resultado obtenido, el algoritmo A* ha encontrado trayectorias válidas y sin colisiones para los aviones en el escenario planteado. Podemos ver en base a los resultados que la heurística 2 es más eficiente que la 1: menos nodos expandidos y h inicial menor. Las rutas escogidas para cada avión es la misma independientemente de la heurística aplicada.

Caso 2: Ejemplo del enunciado

Input:

```
2
(3,3) (0,2)
(0,1) (3,3)
B;B;B;B
B;G;G;G
A;B;G;G
A;A;B;B
```

Estadísticas: h1

```
Tiempo total: 0.00s
Makespan: 8
h inicial: 9
Nodos expandidos: 73
```

h2

```
Tiempo total: 0.00s
Makespan: 8
h inicial: 5
Nodos expandidos: 78
```

Output: h1

```
(3,3) ← (3,2) ← (3,1) ↑ (2,1) ← (2,0) ↑ (1,0) ↑ (0,0) → (0,1) → (0,2)
(0,1) ← (0,0) ↓ (1,0) ↓ (2,0) ↓ (3,0) → (3,1) → (3,2) → (3,3) w (3,3)
```

h2

```
(3,3) ← (3,2) ← (3,1) ↑ (2,1) ← (2,0) ↑ (1,0) ↑ (0,0) → (0,1) → (0,2)
(0,1) ← (0,0) ↓ (1,0) ↓ (2,0) ↓ (3,0) → (3,1) → (3,2) w (3,2) → (3,3)
```

Como observamos en la heurística 1, el algoritmo decide avanzar hasta el final y hacer una espera en (3,3), mientras que la heurística 2 hace que el avión espere en la celda anterior (3,2). También la h2 expande más nodos y la h inicial es menor.

Caso 3: 2 aviones y objetivos distintos

Input:

```
2
(3,0) (0,1)
(0,2) (2,3)
B;B;B;A
G;B;A;B
G;G;B;B
B;A;A;B
```

Estadísticas: h1

```
Tiempo total: 0.00s
Makespan: 6
h inicial: 7
Nodos expandidos: 49
```

h2

```
Tiempo total: 0.00s
Makespan: 6
h inicial: 4
Nodos expandidos: 77
```

Output: h1

```
(3,0) → (3,1) → (3,2) ↑ (2,2) ↑ (1,2) ↑ (0,2) ← (0,1)
(0,2) → (0,3) ↓ (1,3) ↓ (2,3) w (2,3) w (2,3) w (2,3)
```

h2

```
(3,0) → (3,1) → (3,2) ↑ (2,2) ↑ (1,2) ↑ (0,2) ← (0,1)
(0,2) w (0,2) ← (0,1) → (0,2) → (0,3) ↓ (1,3) ↓ (2,3)
```

En ambas heurísticas, el avión 1 sigue los mismos pasos, pero en el avión 2 cambia: la heurística 1 decide avanzar hasta el final y esperar en su celda objetivo hasta q el otro avión llegue a su goal, mientras que la heurística 2 hace que el avión haga pasos de más en lugar de esperar. Las estadísticas también tienen algunas diferencias como la h inicial y el número de nodos expandidos.

Caso 4: 1 solo avión

Input:

```
1
(0,0) (3,3)
B;B;B;B
B;B;B;B
B;B;B;B
B;B;B;B
```

Estadísticas: h1

```
Tiempo total: 0.00s
Makespan: 6
h inicial: 6
Nodos expandidos: 16
```

h2

```
Tiempo total: 0.00s
Makespan: 6
h inicial: 6
Nodos expandidos: 16
```

Output: h1 `(0,0) → (0,1) → (0,2) → (0,3) ↓ (1,3) ↓ (2,3) ↓ (3,3)`

h2 `(0,0) → (0,1) → (0,2) → (0,3) ↓ (1,3) ↓ (2,3) ↓ (3,3)`

Ambas heurísticas presentan las mismas estadísticas y pasos del avión.

Caso 5: Mismo inicio, mismo objetivo

Input:

```
2
(0,0) (3,3)
(0,0) (3,3)
B;B;B;B
B;B;B;B
B;B;B;B
B;B;B;B
```

Estadísticas: -

Output:

```
(.venv) PS C:\Users\Ines\Documents\heu2> python ASTARRodaje.py ASTAR-tests/mapa03.csv 1
No se encontró solución.
(.venv) PS C:\Users\Ines\Documents\heu2> python ASTARRodaje.py ASTAR-tests/mapa03.csv 2
No se encontró solución.
```

Ambos aviones inician en la misma celda y tienen como celda objetivo la misma. Dado que nuestro algoritmo tiene la restricción de no permitir que dos aviones coincidan en una misma celda, no encuentra solución a este input.

Caso 6: Misma celda como objetivo

Input:

```
3
(0,0) (3,3)
(2,0) (3,3)
(1,0) (3,3)
B;B;B;B
B;G;B;B
B;B;B;B
B;B;B;B
```

Estadísticas: -

Output:

```
(.venv) PS C:\Users\Ines\Documents\heu2> python ASTARRodaje.py ASTAR-tests/mapa04.csv 1
No se encontró solución.
(.venv) PS C:\Users\Ines\Documents\heu2> python ASTARRodaje.py ASTAR-tests/mapa04.csv 2
No se encontró solución.
```

Dado que los tres aviones tienen como objetivo la misma celda, es imposible encontrar una solución.

4. Conclusión

En conclusión, hemos logrado modelar y resolver dos problemas complejos en la operación de una aerolínea: el mantenimiento de aviones y la planificación de su rodaje en el aeropuerto. Usamos técnicas de Satisfacción de Restricciones (CSP) y búsqueda heurística para cumplir con las restricciones y optimizar los procesos en cuanto a eficiencia y manejabilidad.