



Universidad Carlos III  
Heurística y Optimización  
Práctica 2  
Curso 2024-25

**CSP y Búsqueda Heurística**

Fecha de entrega: **20-12-2024**

Grupo: **82**

Autores:

**David Mancebo Barrena**                      **100495917**

**Ilinca Bianca Mitrea Mitrea**              **100495772**

<b>Introducción</b>	<b>3</b>
<b>Parte 1: Validación con Python constraint</b>	<b>3</b>
1. Conjuntos y parámetros del problema:	3
2. Funciones del problema:	3
3. Variables y dominio del problema:	4
4. Restricciones del problema:	4
5. Análisis de los resultados y casos de prueba	5
5.1. Análisis de los casos de prueba	5
5.2. Análisis de complejidad	6
<b>Parte 2: Planificación con búsqueda heurística</b>	<b>7</b>
1. Conjuntos y parámetros:	7
Mapa	7
Conjunto de aviones	7
Conjunto de posiciones iniciales de los aviones	7
Conjunto de posiciones finales de los aviones	7
2. Estados	7
Estado inicial	8
Estado objetivo	8
3. Operadores	8
Precondiciones verificadas globalmente	8
Precondiciones verificadas individualmente	8
Efectos	9
4. Función objetivo	9
5. Función de coste	9
6. Heurísticas	10
6.1. Heurística 1: máximo de distancias Manhattan	10
6.1.1. Definición de la heurística:	10
6.1.2. Restricciones relajadas:	10
6.1.3. Admisibilidad	10
6.2. Heurística 2: Distancia mínima precalculada con BFS	10
6.2.1. Definición	10
6.2.2. Restricciones relajadas:	11
6.2.3. Admisibilidad:	11
7. Modelo de planificación global y transición de estados	11
Selección global de acciones	11
Transición de estados	12
Actualizar la posición de cada avión	12
8. Análisis comparativo del rendimiento del algoritmo con las heurísticas implementadas	12
a. Caso 1	12
b. Caso 2	13
c. Caso 3	14
Conclusiones de la comparación de heurísticas	14
9. Justificación del uso de heapq en la implementación de A*	14
<b>Conclusiones de la práctica</b>	<b>15</b>

## Introducción

En este documento presentamos el modelado y resolución de la práctica 2, que consiste en dos problemas relacionados con la gestión y planificación de una flota de aviones. En una primera parte, se emplea la técnica de satisfacción de restricciones (CSP) para determinar la asignación horaria y espacial de las tareas de mantenimiento, teniendo en cuenta la disponibilidad de talleres, el número de tareas a realizar (estándar y especialista), el tipo de avión (estándar o jumbo) y las limitaciones espaciales y de maniobrabilidad entre talleres y parkings. En la segunda parte, se aborda el problema de la planificación del rodaje de aviones en un aeropuerto, desde sus posiciones iniciales hasta las pistas asignadas, evitando colisiones y cumpliendo con las condiciones del mapa (celdas transitables, intransitables y celdas de no espera). Este problema se modela como un problema de búsqueda heurística con el objetivo de minimizar el makespan, y se implementa el algoritmo A\* con dos heurísticas admisibles, analizando su rendimiento en distintas configuraciones del entorno.

## Parte 1: Validación con Python constraint

### 1. Conjuntos y parámetros del problema:

- *NFRANJAS*: número de franjas horarias del problema
- *FILAS*: número de filas de la matriz de talleres
- *COLS*: número de columnas de la matriz de talleres
- *Aviones* =  $\{AV_i \mid i \in \mathbb{N}, 1 \leq i \leq |Aviones|\}$ : Aviones involucrados en el problema
- *TipoAvión* =  $\{t_i \in \{STD, JMB\} \mid i \in \mathbb{N}, 1 \leq i \leq |Aviones|\}$ : Tipo de los aviones del conjunto *Aviones*
- *RestrAvión* =  $\{r_i \in \{T, F\} \mid i \in \mathbb{N}, 1 \leq i \leq |Aviones|\}$ : Booleano que representa si se deben hacer las tareas especialistas antes de las estándar en el avión *i* del conjunto *Aviones*
- *T1Avión* =  $\{t1_i \in \mathbb{N} \mid i \in \mathbb{N}, 1 \leq i \leq |Aviones|\}$ : Cantidad de tareas de tipo 1 necesarias en cada uno de los aviones del conjunto *Aviones*
- *T2Avión* =  $\{t2_i \in \mathbb{N} \mid i \in \mathbb{N}, 1 \leq i \leq |Aviones|\}$ : Cantidad de tareas de tipo 2 necesarias en cada uno de los aviones del conjunto *Aviones*

### 2. Funciones del problema:

- *Asignación*( $AV_i, (f, c), t$ )  $\rightarrow \{0, 1\}$ : función booleana que recibe un avión del conjunto *Aviones*, la posición de un taller (*f, c*) y una franja horaria *t* y devuelve 1 si el avión  $AV_i$  está asignado al taller (*f, c*) en la franja horaria *t* y 0 si no.
- *TipoAvión* <sub>$t \in \{STD, JMB\}$</sub> ( $AV_i$ )  $\rightarrow \{0, 1\}$ : función booleana que recibe un avión de conjunto *Aviones* y devuelve 1 si el avión  $AV_i$  es de tipo *t* y 0 si no.
- *TipoTaller* <sub>$t \in \{STD, SPC, PRK\}$</sub> (*f, c*)  $\rightarrow \{0, 1\}$ : función booleana que recibe la posición de un taller (*f, c*) y devuelve 1 si el taller (*f, c*) es de tipo *t* y 0 si no.
- *NAsignaciones* <sub>$t \in \{STD, SPC, PRK\}$</sub> ( $\bigcup_{N \leq j \leq M} Pos_{ij}$ )  $\rightarrow \mathbb{N}$ : función que recibe un conjunto de posiciones de un avión en diferentes franjas horarias y devuelve el número de asignaciones a talleres de tipo *t* para este.

### 3. Variables y dominio del problema:

$X = Pos_{ij} \mid 1 \leq i \leq |Aviones|, 1 \leq j \leq NFRANJAS$ : posición del avión  $i$  en la franja  $j$ ;

$D = \{(f, c) \mid 0 \leq f < FILAS, 0 \leq c < COLS, f, c \in \mathbb{N}\}$ : tupla con la fila y columna del taller/parking asignado a un avión en una franja de tiempo específica. Este dominio es el mismo para todas las variables.

### 4. Restricciones del problema:

**R1:** Máximo de 2 aviones por taller en cada franja horaria.

$$\forall f \in [0, FILAS), \forall c \in [0, COLS), \forall t \in [1, NFRANJAS]: \sum_{i=1}^{|Aviones|} Asignación(AV_i, (f, c), t) \leq 2$$

**R2:** Máximo de 1 avión de tipo JMB por taller en cada franja horaria.

$$\forall f \in [0, FILAS), \forall c \in [0, COLS), \forall t \in [1, NFRANJAS]: \sum_{i=1}^{|Aviones|} (Asignación(AV_i, (f, c), t) \cdot TipoAvión_{JMB}(AV_i)) \leq 1$$

Nota: en las restricciones R1 y R2 hemos asumido que las restricciones de espacio total también se aplican a los parkings, ya que es lo que indica el sentido común y no tiene sentido que en un parking pueda haber infinitos aviones. También hemos considerado que es posible que en un mismo taller/parking haya a la vez un avión JMB y uno STD, ya que el enunciado solo indica que “no podrá haber más de un avión JMB en la misma franja horaria en el mismo taller”.

**R3:** Todas las tareas de todos los aviones tienen que estar hechas a lo largo de las franjas horarias del problema, pudiendo hacerse las tareas  $t1$  en talleres SPC.

$$\forall AV_i \in Aviones: NAsignaciones_{STD}(\bigcup_{1 \leq j \leq NFRANJAS} Pos_{ij}) + NAsignaciones_{SPC}(\bigcup_{1 \leq j \leq NFRANJAS} Pos_{ij}) \geq T2Avión_i + T1Avión_i$$

Nota: Hemos asumido que un avión puede descansar en un taller o un parking sin realizar tareas de mantenimiento, ya que el enunciado no lo especifica. Esto se ve reflejado en el uso de  $\geq$  en vez de  $=$ , lo que permite estar en más talleres que tareas totales en todas las franjas de tiempo.

**R4:** Todas las tareas tipo 2 tienen que estar hechas en talleres SPC a lo largo de las franjas horarias del problema.

$$\forall AV_i \in Aviones: NAsignaciones_{SPC}(\bigcup_{1 \leq j \leq NFRANJAS} Pos_{ij}) \geq T2Avión_i$$

Nota: Al usar  $\geq$  en vez de  $=$  en esta restricción y junto a la restricción R3 permitimos que las tareas  $t1$  se realicen en talleres de tipo SPC.

**R5:** Para todo  $AV_i$  con  $RestrAvión_i = T$ , se deben terminar todas las tareas  $t2$  antes de entrar a un taller STD.

$$\forall t \in [1, NFRANJAS], \forall f \in [0, FILAS), \forall c \in [0, COLS), \forall AV_i \mid RestrAvión_i = T:$$

$$(Asignación(AV_i, (f, c), t) = 1) \wedge (TipoTaller_{STD}(f, c) = 1) \Rightarrow NAsignaciones_{SPC}(\bigcup_{1 \leq j \leq t} Pos_{ij}) \geq T2Avión_i$$

Nota: la restricción puede parecer muy compleja, pero solo comprueba que si se ha asignado a un avión a un taller STD en cualquier franja y cualquier posición y  $Restr = T$ , el número de asignaciones a talleres SPC para ese avión en franjas anteriores debe ser mayor o igual al número de tareas  $t2$  del avión.

**R6:** Si un taller/parking tiene asignado un avión en una franja horaria, debe haber al menos un taller/parking adyacente sin ningún avión asignado en esa franja horaria.

$$\forall t \in [1, NFRANJAS], \forall f \in [0, FILAS), \forall c \in [0, COLS):$$

$$(\exists AV_i \mid Asignación(AV_i, (f, c), t) = 1) \Rightarrow (\forall (f', c') \in Ady_{f,c} (\neg \exists AV_j \mid Asignación(AV_j, (f', c'), t) = 1)),$$

donde  $Ady_{f,c}$  representa el conjunto de posiciones adyacentes a la posición  $(f, c)$  y el operador  $\vee$  representa la disyunción lógica (OR) sobre todas las posiciones adyacentes de  $(f, c)$

**R7:** No puede haber 2 talleres/parkings adyacentes con un avión JMB asignado en la misma franja horaria.

$\forall t \in [1, NFRANJAS], \forall f \in [0, FILAS], \forall c \in [0, COLS]:$

$(\exists AV_i | (Asignación(AV_i, (f, c), t) = 1) \wedge (TipoAvión_{JMB}(AV_i) = 1)) \Rightarrow$

$\Rightarrow (\wedge (f', c') \in Ady_{f,c} (\neg \exists AV_j | (Asignación(AV_j, (f', c'), t) = 1) \wedge (TipoAvión_{JMB}(AV_j) = 1)))$ ,

donde  $Ady_{f,c}$  representa el conjunto de posiciones adyacentes a la posición  $(f, c)$  y el operador  $\wedge$  representa la conjunción lógica (AND) sobre todas las posiciones adyacentes de  $(f, c)$




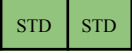
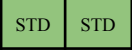


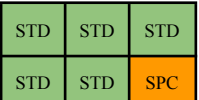
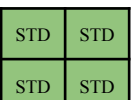
Nota: en la restricción 6 del enunciado de donde obtenemos R7 no se menciona a los parkings, solo “talleres adyacentes”, pero por sentido común hemos decidido extender esta restricción también a los parkings, ya que la restricción 5 del enunciado también trata sobre garantizar la maniobrabilidad de los aviones y si toma en cuenta los parkings.

## 5. Análisis de los resultados y casos de prueba

Una vez desarrollado el modelo de la parte 1, procedimos a implementarlo en el lenguaje de programación Python usando la librería python-constraint para la obtención de soluciones. Creamos también funciones auxiliares para recibir los parámetros del problema mediante un fichero de entrada y guardar en un fichero de salida tanto el número de soluciones como entre 1 y 5 de las soluciones encontradas. Para la comprobación del correcto funcionamiento del programa y del cumplimiento de todas las restricciones hemos desarrollado un total de 13 casos de prueba, cuyos resultados analizaremos en una tabla. Estos casos de prueba son muy simples con la intención de que no tarden mucho en ejecutar, pero sirven para comprobar si las restricciones del modelo se están aplicando correctamente. También destacaremos más adelante la escalabilidad del modelo desarrollado y la evolución de la complejidad de la búsqueda de soluciones con el tamaño del problema.

### 5.1. Análisis de los casos de prueba

ID	Franjas	Cuadrícula	Aviones	Soluciones encontradas	Restricciones comprobadas	Análisis
01	4		1-STD-F-2-2	11	R3, R4	Caso básico para comprobar que realicen todas las tareas de un avión y que las tareas t2 se hagan en el taller SPC. Hay $16 - 1 - 4 = 11$ soluciones ya que para cada franja podemos elegir entre el taller STD y SPC, pero las asignaciones de 4 franjas al taller STD (1) y las de 3 franjas al taller STD (4) no son válidas ya que rompen R2.
02	3		1-STD-F-2-2	0	R3, R4	Caso con más tareas totales (4) para el avión 1 qué franjas horarias (3), por lo que no hay una asignación donde se realicen todas las tareas.
03	3		1-STD-F-1-1	4	R3, R4	Caso con más franjas (3) qué tareas totales para el avión 1. 4 soluciones porque se puede descansar en PRK en cualquiera de las 3 franjas o pasar las 3 franjas en STD(0,0).
04	2		1-STD-F-0-2 2-STD-F-0-2 3-STD-F-0-2	36	R1	Caso más complejo con 3 aviones que comprueba que no puedan ir más de 2 aviones en un mismo taller al mismo tiempo. Cada franja tiene 6 posibles asignaciones válidas, lo que hace que haya $6 \times 6 = 36$ soluciones.

05	1		1-STD-F-0-1 2-STD-F-0-1 3-STD-F-0-1	0	R1	Caso simple que muestra que no puede haber más de 2 aviones en el mismo taller en la misma franja. Ninguna solución porque hay 3 aviones y un solo taller.
06	1		1-JMB-F-1-0 2-JMB-F-1-0	2	R2	Caso que muestra que no puede haber más de un avión JMB por casilla, por lo que solo hay 2 soluciones, cada una con uno de los 2 aviones asignados a (0,0) y (0, 2) y viceversa.
07	1		1-JMB-F-1-0 2-JMB-F-1-0	0	R1	Caso simple que muestra que no puede haber más de 1 avión JMB en el mismo taller en la misma franja. Ninguna solución porque hay 2 aviones JMB y un solo taller.
08	1		1-STD-F-1-0 2-STD-F-1-0 3-STD-F-1-0	0	R1, R6	Caso que muestra que para cada casilla asignada a un avión debe haber una casilla adyacente libre. Como los 3 aviones no caben en una casilla tienen que dividirse pero esto rompe R6.
09	1		1-JMB-F-1-0 2-JMB-F-1-0	0	R2, R7	Caso que muestra que no puede haber 2 aviones JMB en casillas adyacentes. Los 2 aviones se deben dividir en las 2 casillas disponibles pero esto rompe R7, por lo que no hay soluciones.
10	1		1-JMB-F-1-0 2-JMB-F-1-0	4	R2, R7	La restricción R7 obliga a los 2 aviones a ocupar los talleres ocupando las 2 diagonales del cuadrado. Por cada diagonal hay 2 posibles permutaciones de los 2 aviones, por lo que hay un total de $2 \times 2 = 4$ soluciones.
11	3		1-STD-T-1-2	2	R5	Caso que muestra que un avión con restr = T debe realizar todas las tareas t2 antes de entrar a un taller STD. El avión siempre está en (0,1) en las dos primeras franjas y en la última puede realizar la tarea t1 en (0,0) o (0,1) (2 soluciones)
12	3		1-JMB-T-2-1 2-JMB-T-2-1	0	R2, R5	Los 2 aviones tienen restr = T por lo que tienen que ir a SPC(1,2) en la franja 1, pero esto rompe R2 ya que los dos son aviones JMB, por lo que no hay soluciones válidas. Si alguno de los dos tuviese restr = F podría ir primero a un taller STD y si habría soluciones.
13	1		-	0	-	Caso básico con 0 aviones. Como no hay aviones no se crean variables y no se puede encontrar ninguna solución.

## 5.2. Análisis de complejidad

En el caso del modelo diseñado, la magnitud del problema depende solamente del número de franjas, las dimensiones de la cuadrícula de talleres y el número de aviones. El número de variables es  $\#Variables = NFRANJAS \cdot |Aviones|$ , y el dominio tiene tamaño  $FILAS \cdot COLS$ . Como el programa desarrollado tiene que encontrar todas las posibles soluciones del problema, esto conlleva explorar todas las posibles asignaciones para todas las variables, por lo que tenemos una complejidad temporal exponencial de  $O((FILAS \cdot COLS)^{NFRANJAS \cdot |Aviones|})$  si el solver de python-contraint no optimiza la búsqueda. Esto se ve reflejado en un problema con 3 franjas, grid 3x3 y 3 aviones ( $9^9$  asignaciones) tardando 102 segundos, pero añadiendo un avión más ( $9^{12}$  asignaciones) el programa pasa a terminar en 1534 segundos (25 minutos y medio).

## Parte 2: Planificación con búsqueda heurística

### 1. Conjuntos y parámetros:

#### Mapa

Sea  $M$  un mapa bidimensional que representa el conjunto de celdas del aeropuerto, donde cada celda se identifica por sus coordenadas  $(x, y)$ :

$$M = \{(x, y) \mid x \in \{0, 1, \dots, m-1\}, y \in \{0, 1, \dots, n-1\}\}.$$

La coordenada  $x$  representa la fila (0 a  $m-1$ ), donde  $m \in \mathbb{N}$  es el número total de filas del mapa.

La coordenada  $y$  representa la columna (0 a  $n-1$ ), donde  $n \in \mathbb{N}$  es el número total de columnas del mapa.

Definimos particiones de  $M$  en función de las restricciones:

$M_{gris} \subseteq M$ : conjunto de celdas grises (no transitables).

$M_{amarillo} \subseteq M$ : conjunto de celdas amarillas (transitables pero sin posibilidad de espera).

$M_{transitable} = M \setminus M_{gris}$ : celdas transitables (incluye las amarillas y las blancas).

#### Conjunto de aviones

Sea el conjunto de aviones:

$$Aviones = \{a_i\}, 1 \leq i \leq N,$$

donde  $N = |Aviones| \in \mathbb{N}$  es el número total de aviones en el problema.

#### Conjunto de posiciones iniciales de los aviones

$$Init = \{Init_i = (x_i^{init}, y_i^{init}) \mid Init_i \in M_{transitable}, i = 1, \dots, N\}$$

Cada avión  $a_i$  tiene una posición inicial  $Init_i = (x_i^{init}, y_i^{init}) \in M_{transitable}$  donde:

- $x_i^{init}$  es la fila de la posición inicial del avión  $a_i$  en la cuadrícula del mapa.
- $y_i^{init}$  es la columna de la posición inicial del avión  $a_i$  en la cuadrícula del mapa.

#### Conjunto de posiciones finales de los aviones

$$Goals = \{Goal_i = (x_i^{goal}, y_i^{goal}) \mid Goal_i \in M_{transitable}, i = 1, \dots, N\}$$

Cada avión  $a_i$  tiene una posición objetivo:  $Goal_i = (x_i^{goal}, y_i^{goal}) \in M_{transitable}$  donde:

- $x_i^{goal}$  es la fila de la posición inicial del avión  $a_i$  en la cuadrícula del mapa.
- $y_i^{goal}$  es la columna de la posición inicial del avión  $a_i$  en la cuadrícula del mapa.

### 2. Estados

Definimos la función  $PosAv(a_i, t)$  que da la posición  $(x, y)$  del avión  $a_i$  en el tiempo  $t$ .

$$PosAv: Aviones \times \mathbb{N} \rightarrow M_{transitable}$$

donde  $Aviones$  es el conjunto de aviones y  $M_{transitable}$  el conjunto de celdas transitables del mapa

$$\text{Por lo tanto } PosAv(a_i, t = 0) = Init_i$$

Un estado en el tiempo  $t$  se denota por una tupla que contiene la posición actual de cada avión:

$$S(t) = (PosAv(a_1, t), PosAv(a_2, t), \dots, PosAv(a_N, t))$$

donde, por definición,  $PosAv(a_i, t) \in M_{transitable}$

### **Estado inicial**

$S_{inicial}$  es el estado inicial del sistema en  $t = 0$ :

$$S_{inicial} = S(0) = (PosAv(a_1, 0), PosAv(a_2, 0), ..., PosAv(a_N, 0)) = (Init_1, Init_2, ..., Init_N)$$

### **Estado objetivo**

$S_{final}$  es un conjunto de estados, ya que pueden existir varios escenarios válidos en los que los aviones han alcanzado sus objetivos en diferentes configuraciones de tiempo.

$$S_{final} = \{S(t) \mid \forall a_i \in Aviones, PosAv(a_i, t) = Goal_i\}$$

Es decir, cualquier estado que cumple  $S(t) = (Goal_1, Goal_2, ..., Goal_N)$  es un estado final .

Dicho de otra forma, un estado  $S(t)$  es final si y sólo si para cada avión  $a_i$ :

$$PosAv(a_i, t) = Goal_i$$

## **3. Operadores**

Llamaremos *Acciones* al conjunto de operadores que podemos aplicar a un avión individual  $a_i$ :

$$Acciones = \{MoveUp, MoveDown, MoveLeft, MoveRight, Wait\}$$

### **Precondiciones verificadas globalmente**

Las precondiciones globales se aplican al sistema completo de aviones y dependen de todas las acciones y estados de los aviones simultáneamente. Son las siguientes :

#### **1. No debe haber colisión entre aviones en el siguiente estado:**

Al aplicar un operador sobre un avión  $a_i$  en un instante  $t$  se debe cumplir que  $\forall a_i, a_j \in Aviones, i \neq j$ :

$$(PosAv(a_i, t + 1) \neq PosAv(a_j, t + 1))$$

Esto implica que los operadores elegidos para cada avión en  $t$  no ubiquen a dos aviones en la misma celda en  $t + 1$ .

#### **2. No debe haber intercambio de posiciones adyacentes entre aviones:**

Dado el estado del sistema en el tiempo  $t$ :  $S(t) = (PosAv(a_1, t), PosAv(a_2, t), ..., PosAv(a_N, t))$

Para dos aviones  $a_i, a_j \in Aviones, i \neq j$ :

$\forall i, j \in \{1, 2, ..., N\}, i \neq j$ , si  $PosAv(a_i, t)$  es adyacente a  $PosAv(a_j, t)$  entonces no se puede cumplir que:

$$(PosAv(a_i, t + 1) = PosAv(a_j, t)) \wedge (PosAv(a_j, t + 1) = PosAv(a_i, t))$$

### **Precondiciones verificadas individualmente**

Las precondiciones individuales se aplican a cada avión  $a_i$  individualmente y dependen solo de  $a_i$  y su celda destino.

Para aplicar cualquiera de las acciones sobre un avión  $a_i$  , el avión debe estar en una celda transitable:

$PosAv(a_i, t) = (x_i, y_i) \in M_{transitable}$ . Esta restricción no hace falta incluirla como una precondición explícita

de las acciones porque por definición  $PosAv(a_i, t)$  siempre representa la posición actual del avión, la cual es

una celda transitable:  $PosAv: Aviones \times \mathbb{N} \rightarrow M_{transitable}$



## Efectos

A continuación se recoge en la siguiente tabla las precondiciones y efectos de cada operador:

Operador	Precondiciones globales	Precondiciones individuales	Efectos
$MoveUp(a_i)$	$\forall a_i, a_j \in Aviones, i \neq j:$ $(PosAv(a_i, t + 1) \neq PosAv(a_j, t + 1))$  Si $PosAv(a_i, t)$ y $PosAv(a_j, t)$ son adyacentes, no puede darse el caso: $(PosAv(a_i, t + 1) = PosAv(a_j, t)) \wedge$ $(PosAv(a_j, t + 1) = PosAv(a_i, t))$	La celda superior es transitable: $(x_i - 1, y_i) \in M_{transitable}$	$PosAv(a_i, t + 1) = (x_i - 1, y_i)$
$MoveDown(a_i)$		La celda inferior es transitable: $(x_i + 1, y_i) \in M_{transitable}$	$PosAv(a_i, t + 1) = (x_i + 1, y_i)$
$MoveLeft(a_i)$		La celda izquierda es transitable: $(x_i, y_i - 1) \in M_{transitable}$	$PosAv(a_i, t + 1) = (x_i, y_i - 1)$
$MoveRight(a_i)$		La celda derecha es transitable: $(x_i, y_i + 1) \in M_{transitable}$	$PosAv(a_i, t + 1) = (x_i, y_i + 1)$
$Wait(a_i)$		La celda actual no es amarilla: $PosAv(a_i, t) = (x_i, y_i) \in M_{blanco}$	$PosAv(a_i, t + 1) = (x_i, y_i)$

## 4. Función objetivo

Para seleccionar la mejor combinación de acciones para pasar al siguiente estado, el algoritmo A\* utiliza una función heurística  $h$  y una función de coste acumulado  $g$  para evaluar el valor de los estados candidatos. La selección se hace minimizando la función objetivo.

La función objetivo combina dos componentes:

- Coste acumulado  $g(S)$ : Representa el costo total de las acciones realizadas para alcanzar el estado actual  $S(t)$  desde el estado inicial  $S_{inicial}$
- Función heurísticas  $h(S)$ : Proporciona una estimación del coste mínimo restante para llegar desde  $S(t)$  al estado objetivo  $S_{final}$

La función objetivo es:

$$f(S) = g(S) + h(S)$$

donde  $g(S)$  se incrementa con cada acción realizada para llegar al estado actual y  $h(S)$  garantiza que la búsqueda esté informada sobre la proximidad al objetivo.

## 5. Función de coste

El objetivo de nuestro problema es encontrar la solución que minimice el makespan, es decir, el tiempo total necesario para que todos los aviones alcancen sus respectivas posiciones objetivo. En nuestro problema, **cada transición de estado (cada paso de tiempo) tiene un coste unitario**. Esto quiere decir que si la secuencia de acciones para llegar al estado objetivo es de longitud  $T$  (es decir, se requieren  $T$  pasos de tiempo para llegar a un estado en el que todos los aviones han alcanzado sus metas), **el coste total de la solución es  $T$** .

La solución al problema es una secuencia finita de estados y acciones:

$Solución = (Transición(t = 0), Transición(t = 1), ..., Transición(T - 1))$ .

Por lo tanto, la función de coste total para una solución que finaliza en el tiempo T es:  $g(Solución) = T$

## 6. Heurísticas

### 6.1. Heurística 1: máximo de distancias Manhattan

#### 6.1.1. Definición de la heurística:

Para cada avión  $a_i$  calculamos la distancia Manhattan entre su posición actual  $PosAv(a_i, t) = (x_i, y_i)$  y su posición objetivo  $Goal(a_i) = (x_i^{goal}, y_i^{goal})$ . La distancia Manhattan para un avión es:

$$d_i = |x_i - x_i^{goal}| + |y_i - y_i^{goal}|$$

Para calcular la heurística del estado tomamos el máximo de las distancias Manhattan de los aviones:

$$h_1(S) = \max_{1 \leq i \leq N} \{|x_i - x_i^{goal}| + |y_i - y_i^{goal}|\}$$

donde  $N = |Aviones|$

#### 6.1.2. Restricciones relajadas:

Esta heurística asume un desplazamiento directo (en línea recta, sólo en movimientos horizontales y verticales) desde la posición actual hasta la meta, sin considerar ningún tipo de obstáculo o restricción adicional. En concreto, relaja las siguientes restricciones del problema real:

- No considera celdas intransitables (grises)
- No considera las celdas amarillas
- No considera que los aviones no puedan estar los dos en la misma celda (colisiones).
- No considera que no se pueden intercambiar simultáneamente posiciones entre dos aviones adyacentes.

#### 6.1.3. Admisibilidad

La distancia Manhattan es admisible ya que siempre ofrece una estimación que es menor o igual al coste real necesario. En el peor de los casos, el avión puede tener que rodear obstáculos o esperar, incrementando el coste real por encima de la simple distancia Manhattan. Por lo tanto, la distancia Manhattan es una cota inferior del coste real.

### 6.2. Heurística 2: Distancia mínima precalculada con BFS

#### 6.2.1. Definición

Esta heurística precalcula, para cada avión y para cada celda, la distancia mínima hasta su meta utilizando una búsqueda BFS "inversa" desde la meta, teniendo en cuenta las celdas grises (no transitables). Para cada avión  $a_i$ , se define:

$d_{BFS}(a_i, x, y) = \text{longitud del camino más corto desde } (x, y) \text{ hasta } (x_i^{goal}, y_i^{goal}) \text{ evitando celdas no transitables (grises)}$ .

Esto implica:

1. Para cada meta de un avión, calcular la distancia mínima desde esa meta hacia cada celda del mapa utilizando búsqueda BFS inversa. Decimos que el BFS es "inverso" porque en lugar de partir desde la posición inicial del avión y buscar caminos hacia la meta, se comienza desde la meta del avión y se

calcula la distancia mínima hacia todas las celdas transitables del mapa. Así precalculamos para cada posición objetivo un "mapa de distancias" desde la meta hacia el resto del mapa.

2. La heurística considera únicamente las celdas transitables  $(x, y) \in M_{transitable}$ . Cualquier celda en  $M_{gris}$  se considera inalcanzable y tiene una distancia asignada de  $\infty$ .
3. Guardar estos valores en una tabla de búsqueda rápida.
4. Durante la ejecución de A\*, simplemente se debe consultar la tabla para obtener la distancia mínima de cada avión a su meta.

Por lo tanto, la heurística para un estado  $S$  es:

$$h_2(S) = \max_{a_i} d_{BFS}(a_i, x_i, y_i)$$

#### 6.2.2. Restricciones relajadas:

Esta heurística relaja las siguientes restricciones:

- Ignora las posibles colisiones entre aviones.
- No considera las restricciones de no intercambio de posiciones entre aviones.

#### 6.2.3. Admisibilidad:

Dado que la heurística calcula la distancia mínima sin tener en cuenta otros aviones, la heurística es admisible porque nunca sobreestima el coste real desde el estado actual hasta el estado objetivo. Las interacciones con otros aviones sólo pueden aumentar el coste real (por esperas y rodeos).

## 7. **Modelo de planificación global y transición de estados**

Sea la función  $OperadoresAvion: Aviones \times Tiempo \rightarrow P(Acciones)$  donde  $P(Acciones)$  denota el conjunto de todas las partes (o subconjuntos) de  $Acciones$  que se puede aplicar al avión  $a_i$  en el instante  $t$  cumpliendo con las precondiciones que se han explicado antes.

Dado un avión  $a_i \in Aviones$  y un instante  $t \in Tiempo$ ,  $OperadoresAvion(a_i, t)$  devuelve el conjunto de acciones factibles para  $a_i$  en el estado  $S(t)$ :

$$OperadoresAvion(a_i, t) \subseteq Acciones$$

Para pasar de un estado  $S(t)$  al estado  $S(t + 1)$  se aplica una acción  $accion_i$  a cada avión  $a_i$  en el tiempo  $t$ .

### Selección global de acciones

Definimos  $SeleccionarAccionesGlobal(t)$ . Esta función tiene como finalidad elegir una tupla de acciones, una acción para cada avión, que se aplicarán al mismo tiempo sobre los aviones en el instante  $t$ :

$$SeleccionarAccionesGlobal(t) = (accion_1(t), accion_2(t), ..., accion_N(t))$$

donde  $accion_i \in OperadoresAvion(a_i, t)$

La función  $SeleccionarAccionesGlobal(t)$  busca la tupla de acciones que conduce desde  $S(t)$  hacia un estado sucesor  $S(t + 1)$  que minimice  $f(S(t + 1))$ . Teniendo en cuenta que  $S(t + 1) = Transicion(S(t), (accion_1, ..., accion_N))$ :

$$SeleccionarAccionesGlobal(t) = \arg \min_{(accion_1, ..., accion_N)} f(Transición(S(t), (accion_1, ..., accion_N)))$$

Donde el mínimo se toma sobre todas las combinaciones factibles  $(accion_1, ..., accion_N)$  con  $accion_i \in OperadoresAvion(a_i, t)$

### **Transición de estados**

La función de transición de estado *Transición* toma un estado actual  $S(t)$  y una tupla de acciones (seleccionada por *SeleccionarAccionesGlobal*( $t$ )) y produce el siguiente estado  $S(t + 1)$ .

*Transición*:  $Estados \times (Acciones^N) \rightarrow Estados$

Dado:

$S(t) = (PosAv(a_1, t), PosAv(a_2, t), ..., PosAv(a_N, t))$

y

$SeleccionarAccionesGlobales(t) = (accion_1, accion_2, ..., accion_N)$

Entonces:

$S(t + 1) = Transición(S(t), (accion_1, ..., accion_N)) = (PosAv(a_1, t + 1), ..., PosAv(a_N, t + 1))$

Para ello, usa internamente *ActualizarAv* para cada avión.

### **Actualizar la posición de cada avión**

La función *ActualizarAv* actualiza la posición de un avión dado su posición actual y la acción a aplicar. Si  $Pos = PosAv(a_i, t)$  y  $accion_i \in Acciones$

*ActualizarAv*:  $(Posiciones \times Acciones) \rightarrow Posiciones$

$PosAv(a_i, t + 1) = ActualizarAv(PosAv(a_i, t), accion_i)$

Si  $accion_i = Wait \Rightarrow ActualizarAv((x_i, y_i), Wait) = (x_i, y_i)$

Si  $accion_i = MoveUp \Rightarrow ActualizarAv((x_i, y_i), MoveUp) = (x_i - 1, y_i)$

Si  $accion_i = MoveDown \Rightarrow ActualizarAv((x_i, y_i), MoveDown) = (x_i + 1, y_i)$

Si  $accion_i = MoveLeft \Rightarrow ActualizarAv((x_i, y_i), MoveLeft) = (x_i, y_i - 1)$

Si  $accion_i = MoveRight \Rightarrow ActualizarAv((x_i, y_i), MoveRight) = (x_i, y_i + 1)$

## **8. Análisis comparativo del rendimiento del algoritmo con las heurísticas implementadas**

A continuación vamos a hacer una comparación detallada del rendimiento de las dos heurísticas propuestas: la Heurística 1 (Distancia Manhattan) y la Heurística 2 (Distancia mínima precalculada con BFS en el mapa, teniendo en cuenta celdas intransitables). La comparación se basa en el tiempo de cálculo y el número de nodos expandidos por el algoritmo A\* en diferentes escenarios de complejidad creciente.

### **a. Caso 1**

Primero, analizamos un caso en el que el mapa es bastante grande (tenemos 54 celdas) pero representa un problema sencillo de resolver ya que no hay celdas intransitables y todas ellas son blancas. Ambas heurísticas ofrecen el mismo rendimiento: el mismo tiempo y el mismo número de nodos expandidos. Esto es esperable ya que en ausencia de obstáculos, ambas heurísticas proporcionan la misma estimación del coste para alcanzar la meta.

Heurística	Input Mapa	Tiempo (s)	Nodos expandidos
H1. Distancia Manhattan	3	1,77	4252
H2. Distancia precalculada con BFS (teniendo en cuenta celdas intransitables)	(0,0) (5,5) (2,0) (0,5) (0,2) (5,4) B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B B;B;B;B;B;B;B;B;B	1,77	4252

### b. Caso 2

En este caso ya comienzan a apreciarse diferencias importantes pues tenemos un mapa que tiene un 25% de sus celdas intransitables. Al tener en cuenta las celdas intransitables, la heurística 2 es más informada y proporciona una estimación más cercana al coste real, lo que permite a A\* centrar su búsqueda en estados más prometedores y, por ende, expandir menos nodos.

Heurística	Input Mapa	Tiempo (s)	Nodos expandidos												
H1. Distancia Manhattan	5	0,96	4004												
H2. Distancia precalculada con BFS (teniendo en cuenta celdas intransitables)	(0,0) (2,3) (2,0) (2,0) (0,2) (0,1) (0,3) (1,1) (1,1) (0,3) B;B;A;A G;B;G;G B;B;B;A  <table border="1"> <tr> <td>AV1 INIT</td><td>AV3 FIN</td><td>AV3 INIT</td><td>AV4 INIT / AV5 FIN</td></tr> <tr> <td></td><td>AV4 FIN / AV5 INIT</td><td></td><td></td></tr> <tr> <td>AV2 INIT / AV2 FIN</td><td></td><td>AV1 FIN</td><td></td></tr> </table>	AV1 INIT	AV3 FIN	AV3 INIT	AV4 INIT / AV5 FIN		AV4 FIN / AV5 INIT			AV2 INIT / AV2 FIN		AV1 FIN		0,74	2804
AV1 INIT	AV3 FIN	AV3 INIT	AV4 INIT / AV5 FIN												
	AV4 FIN / AV5 INIT														
AV2 INIT / AV2 FIN		AV1 FIN													

**Mejora de tiempo** =  $(0.96 - 0.74)/0.96 \approx 22.9\%$

La heurística 2 reduce el tiempo cerca de un 23%.

**Mejora en número de nodos expandidos** =  $(4004 - 2804)/4004 \approx 30\%$

La heurística 2 expande aproximadamente un 30% menos de nodos.

Vemos que la diferencia en cuanto al rendimiento a la hora de utilizar una heurística u otra es bastante importante. Esto se debe a que en este caso tenemos 3 celdas intransitables en el mapa, lo cual representa un 25% del mapa que es intransitable. Si utilizamos la heurística 1, se expande un mayor número de nodos, ya que el algoritmo explora caminos "prometedores" que en realidad no son válidos debido a los obstáculos.

### c. Caso 3

Aquí la diferencia se hace incluso más notoria pues tenemos aún más celdas intransitables (38% de las celdas del mapa son intransitables). Esto confirma la tendencia: cuantas más celdas intransitables, la heurística 2 que tiene en cuenta obstáculos se vuelve más eficiente.

Heurística	Input Mapa	Tiempo (s)	Nodos expandidos																					
H1. Distancia Manhattan	5 (0,0) (2,6) (2,0) (1,6) (0,2) (0,1) (0,3) (1,1) (1,1) (1,5) B;B;A;A;B;B;G G;B;G;G;G;B;B B;B;B;A;G;G;B	8,93	16710																					
H2. Distancia precalculada con BFS (teniendo en cuenta celdas intransitables)	<table><tr><td>AV1 INIT</td><td>AV3 FIN</td><td>AV3 INIT</td><td>AV4 INIT</td><td></td><td></td><td></td></tr><tr><td></td><td>AV4 FIN /AV5 INIT</td><td></td><td></td><td></td><td>AV5 FIN</td><td>AV2 FIN</td></tr><tr><td>AV2 INIT</td><td></td><td></td><td></td><td></td><td></td><td>AV1 FIN</td></tr></table>	AV1 INIT	AV3 FIN	AV3 INIT	AV4 INIT					AV4 FIN /AV5 INIT				AV5 FIN	AV2 FIN	AV2 INIT						AV1 FIN	1,88	2817
AV1 INIT	AV3 FIN	AV3 INIT	AV4 INIT																					
	AV4 FIN /AV5 INIT				AV5 FIN	AV2 FIN																		
AV2 INIT						AV1 FIN																		

**Mejora de tiempo** =  $(8,93 - 1,88)/8,93 \approx 78.9\%$

¡La heurística 2 reduce el tiempo cerca de un 79%!

**Mejora en número de nodos expandidos** =  $(16710 - 2817)/16710 \approx 83,1\%$

¡La heurística 2 reduce el número de nodos expandidos en torno al 83%!

### Conclusiones de la comparación de heurísticas

La heurística 2 es más informada que la heurística 1 pues tiene en cuenta las celdas intransitables y eso la hace más eficiente a la hora de resolver problemas con un mayor número de celdas intransitables. Cuantas más celdas intransitables haya en el mapa de entrada, la Heurística 2 será más recomendable para optimizar la eficiencia del proceso de búsqueda. Sin embargo, en mapas con pocos obstáculos ambas heurísticas darán resultados muy similares, por lo que se podría ser incluso preferible utilizar la Heurística 1 ya que tiene menor demanda de memoria al no tener que almacenar la estructura auxiliar de las distancias precalculadas con BFS.

## 9. Justificación del uso de heapq en la implementación de A\*

En la implementación de A\* utilizamos heapq, que es una librería que forma parte de la biblioteca estándar de Python y permite manipular de forma eficiente una estructura de datos que funcione como cola de prioridad. Concretamente:

- Inserción de nodos: cada nuevo estado generado se inserta en ABIERTA usando push(ABIERTA, (f, g, estado, padre, acción)), garantizando que la cola mantenga el nodo con menor valor de la función objetivo f en la parte superior.

- Extracción del mejor nodo: en cada iteración del bucle principal, se extrae el nodo con menor valor de la función objetivo mediante  $f_{\text{actual}}$ ,  $g_{\text{actual}}$ ,  $\text{actual}$ ,  $\text{padre}$ ,  $\text{accion} = \text{pop}(\text{ABIERTA})$ . Esto asegura que la expansión se realice siempre sobre el nodo más prometedor según la heurística.

Las razones por las que consideramos muy útil utilizar heapq en lugar de una lista simple son las siguientes:

- heapq implementa un montículo binario mínimo (min-heap), donde el valor más pequeño siempre está en la raíz. Esto es ideal para  $A^*$ , ya que necesitamos procesar los nodos con el menor valor de  $f(n) = g(n) + h(n)$
- heapq tiene operaciones eficientes como push (para insertar elementos) y pop (para extraer el elemento con la mayor prioridad), ambas operaciones en  $O(\log n)$  lo que resulta especialmente eficiente cuando se deben manejar muchos nodos en ABIERTA.
- Un min-heap como el de heapq está perfectamente adaptado al patrón de acceso de  $A^*$  (extracción priorizada e inserción frecuente).
- Con listas simples, mantener la prioridad implica ordenar o buscar la posición adecuada para cada inserción, lo que conlleva costes superiores  $O(n)$  para inserciones directas o hasta  $O(n \log n)$  si se ordena después de cada inserción. Estas alternativas resultan innecesariamente costosas en comparación con heapq que mantiene automáticamente el orden con inserciones en  $O(\log n)$ .

NOTA: Cabe mencionar que el enunciado de la práctica no especifica que no se pueda utilizar la librería heapq o que esto vaya a ser penalizado. heapq simplemente nos proporciona una estructura de datos, pero el diseño y la lógica del algoritmo  $A^*$ , qué es lo realmente importante, siguen siendo responsabilidad del código del estudiante por lo que no consideramos que sea un atajo, sino un uso de herramientas estándar.

## Conclusiones de la práctica

Como conclusiones, lo más valioso que nos llevamos de este proyecto es que nos ha permitido profundizar en el modelado de problemas, que consideramos la parte más difícil e importante de la asignatura.

En la parte 1 hemos aprendido a modelar problemas de satisfacción de restricciones (CSP) con variables y dominio definidos por parámetros y restricciones bastante complejas que abarcan varias variables al mismo tiempo. En cuanto al modelado de esta parte, nos ha sido extremadamente útil la definición y el uso de funciones auxiliares como Asignación() o NAsignaciones(), que han servido para definir las restricciones de una forma más simple y compacta. En el modelo solo hemos indicado la entrada y la salida de estas funciones, pero a la hora de la implementación hemos tenido que diseñar código real para poder usarlas. En cuanto a la implementación, ha sido de gran ayuda el uso del lenguaje de programación Python y la librería python-constraint, ya que ambos han sido muy fáciles de usar para la resolución de problemas de satisfacción de restricciones. El único inconveniente de esta parte es que algunas de las restricciones mencionadas en el enunciado eran algo ambiguas, y hemos tenido que asumir bastantes cosas sobre el problema. Todas estas asunciones las hemos indicado en notas en la parte del modelado de restricciones.

En la parte 2 hemos implementado el algoritmo  $A^*$  en Python, lo que nos ha permitido consolidar nuestra comprensión teórica de este algoritmo ampliamente utilizado. Cabe destacar que la flexibilidad, la sintaxis intuitiva y la sencillez de Python han facilitado significativamente la programación. Una de las partes más interesantes de esta etapa del proyecto ha sido la deducción de heurísticas admisibles, donde nos hemos centrado en buscar un equilibrio entre diseñar heurísticas lo más informadas posible y garantizar su admisibilidad. Elegir una heurística más informada resulta clave en nuestro problema de planificación de rutas para múltiples aviones, ya que reduce el número de nodos expandidos por el algoritmo  $A^*$ . Esto, por tanto, aumenta la eficiencia operativa y escalabilidad del sistema, que son aspectos fundamentales en el contexto práctico de una empresa como la compañía aérea que se nos propone en el proyecto. En este contexto, consideramos que la Heurística 2, basada en las distancias mínimas precalculadas mediante BFS, es la mejor opción para nuestro problema. Esta heurística combina precisión, eficiencia y admisibilidad, optimizando tanto el rendimiento como los resultados del algoritmo.