Práctica 2
Técnicas de los Sistemas Inteligentes

Carmen Azorín Martí

11 de mayo de 2025

Introducción

En esta práctica se resuelven varios ejercicios relacionados con los contenidos de la asignatura Técnicas de los Sistemas Inteligentes. Cada ejercicio aborda distintos aspectos y algoritmos estudiados en clase.

Ejercicio 1

a) El programa busca encontrar una combinación de monedas que sume un importe exacto que se le indique en céntimos. El array `monedas` representa los valores de las monedas disponibles, mientras que `solucion` es un array de variables que indica cuántas monedas de cada tipo se usan. Las restricciones son:

- La suma del valor total de las monedas seleccionadas debe ser igual al importe.
- No se permiten cantidades negativas de monedas.

Además, se define una variable `total_monedas` que calcula el número total de monedas utilizadas. El modelo intenta encontrar una solución que satisfaga las condiciones con `solve satisfy` y muestra la solución en pantalla.

La tabla con los resultados obtenidos al resolver el problema del cambio de monedas:

Importe	**Primera solución nº de monedas**	**Nº total de soluciones**	**Runtime (ms)**
0.13 Unid	[13, 0, 0, 0, 0, 0, 0, 0, 0] 13	12	137
1.47 Unid	[147, 0, 0, 0, 0, 0, 0, 0, 0] 147	11555	264
2.30 Unid	[230, 0, 0, 0, 0, 0, 0, 0, 0] 230	77206	917
2.99 Unid	[299, 0, 0, 0, 0, 0, 0, 0, 0] 299	258664	20046

b) Para este apartado hemos añadido una nueva restricción:

```
constraint importe - monedas[7]*solucion[7] - monedas[8]*solucion[8] -
monedas[9]*solucion[9] <100;
```

De esta forma, además de buscar una combinación que sume el importe indicado, se fuerza que la parte entera del importe (mayor o igual a 100 céntimos) se asigne

únicamente a las monedas de 100, 200 y 500 céntimos (índices 7, 8 y 9 del array `monedas`).

La tabla de resultados cuando la parte entera de los importes sea asignada únicamente a monedas de unidades:

Importe	**Primera solución nº de monedas**	**Nº total de soluciones**	**Runtime (ms)**
0.13 Unid	[13, 0, 0, 0, 0, 0, 0, 0, 0] 13	12	132
1.47 Unid	[47, 0, 0, 0, 0, 0, 1, 0, 0] 48	230	235
2.30 Unid	[30, 0, 0, 0, 0, 0, 2, 0, 0] 32	142	174
2.99 Unid	[99, 0, 0, 0, 0, 0, 2, 0, 0] 101	5146	589

c) En este apartado se sustituye `solve satisfy;` (para encontrar una o más soluciones válidas que cumplan las restricciones) por `solve minimize total_monedas;`. Ahora se está buscando no solo una solución válida, sino la solución óptima que minimice el número total de monedas usadas.

La tabla de soluciones óptimas:

Importe	**Solución óptima**	**Nº monedas**	**Runtime (ms)**
0.13 Unid	[0, 1, 0, 1, 0, 0, 0, 0, 0]	2	129
1.47 Unid	[2, 0, 0, 2, 1, 0, 1, 0, 0]	6	159
2.30 Unid	[0, 0, 1, 0, 1, 0, 0, 1, 0]	3	140
2.99 Unid	[1, 1, 0, 2, 1, 1, 0, 1, 0]	7	132

d) Respondamos a la preguntas del enunciado.

a) Los tiempos crecen exponencialmente, hasta el punto de que para el valor 2.99 el tiempo es de más 20 segundos. Para el valor de 3.50 unidades llega a tardar casi un minuto y de 5.27, más de 5 minutos, aunque en mi ordenador ni siquiera se ejecuta, obligándome a forzar salida. Esto se debe al aumento del espacio de búsqueda tan grande que hay.

b) Para importes pequeños no hay tantas posibilidades, porque las monedas grandes apenas tienen combinaciones posibles (por ejemplo, 2.30 puede usar dos monedas de 1 unidad como máximo y no puede usar monedas d 3 y 5 unidades). Sin embargo, para importes grandes de millones de unidades el espacio de búsqueda es inmenso.

Por ejemplo, si quisiéramos calcular todas las combinaciones de monedas de 1, 3 y 5 unidades para el importe de 1 millón de unidades, tendríamos que resolver una ecuación como la siguiente:

$$x + 3y + 5z = 1000000$$

Para resolver esta ecuación, podemos ir dando valores a la x y a la y para despejar la z. Y la cantidad de combinaciones posibles es de más de 6 millones. Teniendo en cuenta que ni siquiera estamos contando las monedas de céntimos, el cálculo de todas las soluciones posibles con todas las monedas es inmenso.

c) Podemos evitar que se reduzcan el número de monedas con lo siguiente: no puede haber más de 2 monedas de 1 céntimo, porque se puede sustituir por una de 3; no puede haber más de 1 moneda de 5, porque se puede sustituir por una de 10; no puede haber más de 1 moneda de 25, porque se puede sustituir por una de 50. También podemos conseguir que se usen todas las monedas de 5 unidades posibles, es en un cálculo rápido hacer:

$$122233367 \text{ céntimos } /500 = 2444660 \text{ monedas de 5 unidades}$$

Ejecutando el código con estas restricciones, obtenemos como primera solución:
`monedas = [0, 19, 0, 1, 0, 0, 3, 0, 244466]`
`total monedas = 244489`
Y encuentra en total 54 soluciones en 192 milisegundos.

e) Al introducir coma flotante tanto en las monedas, como en la cantidad de monedas y en el importe, no se ejecuta ni siquiera el importe de 0.13 Unidades. Esto se debe a que la cantidad de soluciones aumenta al permitir valores infinitesimalmente pequeños como `1.48219693752374e-323`.

Por ejemplo, par ael importe 0.13, nos devuelve como una solución solución
`monedas = [13.0, 1.48219693752374e-323, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0]`
Que se interpreta como 13 monedas de 1 céntimo y un valor extremadamente pequeño de monedas de 3 céntimos, que es prácticamente cero. Ese valor es tan insignificante que al realizar la suma en coma flotante, el resultado sigue siendo exactamente 13.0 porque la precisión de tipo `float` no llega a capturar esa diferencia, es decir, car por debajo de la precisión de la máquina.

La explicación de imprecisión de la máquina me la ha explicado ChatGPT cuando le he preguntado por qué la suma de las monedas no da exactamente 13.

Ejercicio 2

Para este programa definimos los conjuntos `CIUDAD`, que guarda las ciudades; `P1` y `P2`, que guarda las personas a emparejar; y `ANIOS`, que guarda los posibles años que pueden llevar casados (5, 10, 15, 20, 25, 30). Y la idea es calcular las variables de `orden_parejas1`, que guarda los años de casados para cada persona del conjunto `P1`; `orden_parejas2`, que guarda los años de casados para cada persona del conjunto `P2`; y `orden_ciudades`, que guarda los años de casados para cada ciudad.

Las restricciones generales son que todos los elementos de `orden_parejas1`, `orden_parejas2` y `orden_ciudades` deben ser diferentes entre sí. Y luego añadimos las restricciones específicas del enunciado. El modelo simplemente busca una solución que satisfaga todas las restricciones con `solve satisfy;`.

El resultado es el siguiente:

```
Barcelona: Lucia y Pedro 25 años
Parma: Sofia y Pablo 20 años
Lyon: Juan y Jose 30 años
Madrid: Cristina y Marta 10 años
Sevilla: Claudia y Marcos 5 años
```

Ejercicio 3

Para eset ejercicio se define la matriz de tamaño 12x12 con índices definidos por los conjuntos `FILAS` y `COLS`. `k=6` indica que en cada fila y cada columna debe haber exactamente 6 ceros y 6 unos. La idea es rellenar la variable `matriz` que se rellena inicialmente con la restricciones del enunciado.

Posteriormente, restringimos que:

- Que haya el mismo número de 0s y de 1s por fila y por columna: para ello se usa la variable `k` que debe ser igual a la suma de toos los elementos de las filas o de las columnas.
- Que las filas y las columnas sean diferentes entre sí: que para cada fila/columna haya algún elemento de todas las filas/columnas siguientes que sea diferente.
- Evitar que haya tres números consecutivos iguales: para los 3 elementos consecutivos de cada fila/columna los comparamos dos a dos. Con un `or` obligamos a que algún par tenga elementos diferentes, de entre los 3 pares.

El modelo usca una solución que satisfaga todas las restricciones.

En este problema me ha aparecido una única solución, que es la siguiente:

1	1	0	0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	0	1	0	0	1	1
1	0	0	1	1	0	1	0	1	0	1	0
1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0	0	1	1	0
1	0	0	1	1	0	0	1	1	0	0	1
1	0	1	1	0	0	1	0	0	1	0	1
0	1	0	0	1	1	0	1	0	1	1	0
0	0	1	1	0	1	1	0	1	0	1	0
1	0	1	0	1	0	0	1	0	1	0	1
0	1	0	1	0	1	0	1	1	0	0	1

Ejercicio 4

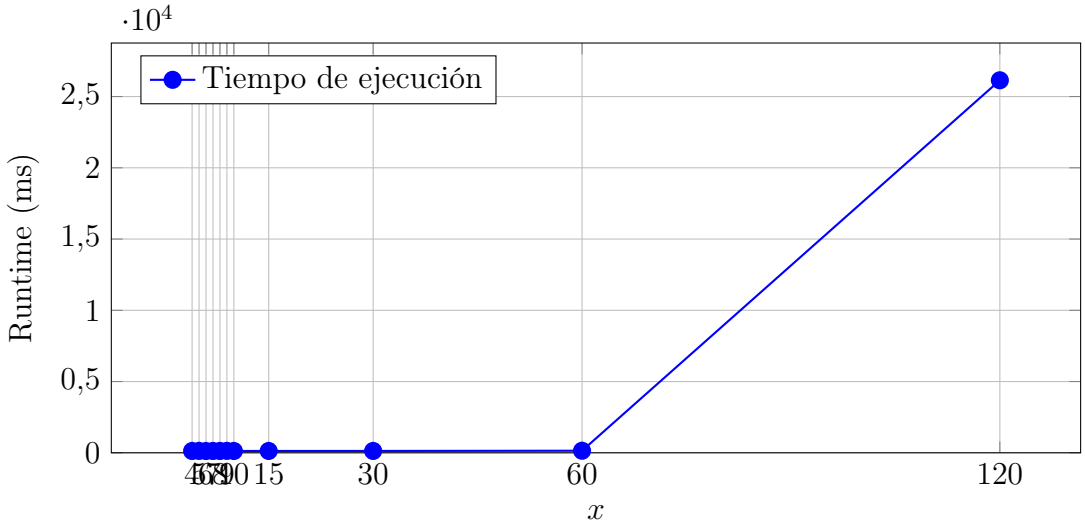
Para descomponer en cuadrados perfectos un número entero, lo guardamos en `x`. Para usar la descomposición en números más pequeños posibles, maximizamos la cantidad de sumandos distintos. Creamos una variable `S` que es un array de `x-1` variables binarias que indica si el valor correspondiente se incluye o no en la descomposición (Si `S[i]=1`, el valor `i` se incluye en la descomposición). Y lo que buscamos es que la suma de los cuadrados de los valores seleccionados (los que tienen `S[i]=1`) debe ser igual al cuadrado de `x`.

También definimos una variable llamada cardinalidad que cuenta cuántos elementos forman parte de del conjunto, como `S` sólo tiene elementos 0 y 1, basta con sumar todos sus elementos. Y para resolver el problema, maximizamos la cardinalidad.

Tabla de resultados:

x	S	$\|S\|$	**Runtime (ms)**
4	-	0	126
5	{3, 4}	2	131
6	-	0	123
7	{2, 3, 6}	3	127
8	-	0	126
9	{2, 4, 5, 6}	4	133
10	{1, 3, 4, 5, 7}	5	126
15	{1, 2, 3, 4, 5, 7, 11}	7	128
30	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15}	13	129
60	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 21, 22, 23}	21	144
120	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30, 31, 32, 33, 34, 38}	34	26142

La gráfica de tiempos es la siguiente:



En la gráfica vemos que desde 4 hasta 60 los tiempos se mantienen prácticamente constantes. Sin embargo, al probar 120, el tiempo llega a ser de más de 26 segundos. Esto indica que el problema no es escalable.

El problema es que cuando más grande es el número x, la cantidad de subconjuntos posibles $S \subset \{1, ..., x-1\}$ cuya suma de cuadrados tenemos que comprobar que sea x^2 crece de forma combinatoria. Para un conjunto de n elementos, el número total de subconjuntos posibles es 2^n, lo que implica complejidad exponencial en el peor caso.

Ejercicio 5

Este programa resuelve el problema de planificación de tareas para ensamblar el DeLorean. Para hacerlo se definen los conjuntos de `Tarea` (A, B, C, D..., J) y `Trabajadores` (Marty, Doc y Biff). Por otro lado, una lista de `Predecesoras` que guarda subconjuntos de tareas que se deben completar antes de empezar otra tarea. Y además, se especifica la duración en días para cada tarea y cada trabajador fijo (Marty y Doc).

Las variables a rellenar son:

- `inicio` y `fin`: arrays de índice `Tarea` que indican el día de inicio y el día de fin de cada tarea.
- `asignacion_marty`: array de índice `Tarea` que indica si ña tarea cuenta con el apoyo de Biff (1 si lo apoya y 0 si no).
- `tiempo_total`: guarda el tiempo total del ensamblado, calculado como el máximo día de finalización entre todas las tareas.

Para calcular el tiempo de finalización de cada tarea tenemos que tener en cuenta quién realiza la tarea y si recibe el apoyo de Biff; si la tarea es crítica para el ensamblaje, se reduce 2 días si está Biff y si no, se reduce 1 día. Si el tiempo de la tarea es menor a 3 días, el apoyo de Biff no reduce la duración. Así que para implementar estas restricciones hemos implementado que para cada tarea, `fin[t]` (su tiempo de finalización) es su tiempo de inicio `inicio[t]` más lo que se tarde, incluyendo un `if` para comprobar quién hace la tarea, otro `if` para ver si recibe la ayuda de Biff, otro `if` para ver si la tarea tarda más de 3 días y otro `if` para ver si la tarea es crítica o no.

También se crea una restricción para obligar a que las tareas empiecen después de que todas sus tareas predecesoras hayan terminado, es decir, `fin[p] <= inicio[t]` para cada tarea `t` y para cada predecesora suya `p`. Por otro lado, Biff solo puede apoyar en 4 tareas como máximo y no pueden solaparse en el tiempo, es decir, cogemos todos los pares de tareas de Biff `i, j` y comprobamos que `(fin[i] <= inicio[j]  fin[j] <= inicio[i])`. Finalmente, si dos tareas son asignadas al mismo trabajador, no pueden: es la misma restricción que la anterior pero para Marty y Doc.

Para resolver el ejercicio, minimizamos el tiempo total de ensamblado, que es el máximo del array `fin`.

a) El resultado es:

```
Tiempo total: 27 días
Asignación de tareas:
"Montar chasisïnicio día 1 fin día 9. Asignada a Marty con apoyo de Biff
Ïnstalar ruedasïnicio día 9 fin día 10. Asignada a Marty
Çableado eléctricoïnicio día 10 fin día 11. Asignada a Doc con apoyo de
Biff
"Motor de fusión nuclearïnicio día 11 fin día 15. Asignada a Doc con apoyo
de Biff
"Tablero de controlïnicio día 15 fin día 18. Asignada a Doc
Ïnstalación y configuración del condensador de fluzoïnicio día 18 fin
día 23. Asignada a Doc
.Ajuste aerodinámicoïnicio día 15 fin día 22. Asignada a Marty
Ïnstalación de puertas de ala de gaviotaïnicio día 22 fin día 23. Asignada
a Marty con apoyo de Biff
"Panel de tiempoïnicio día 23 fin día 25. Asignada a Doc
Ïmplementación y validación de algoritmos de búsqueda heurísticaïnicio
día 25 fin día 27. Asignada a Doc
```

El Diagrama de Gantt lo ha hecho ChatGPT pasándome un código de Python:

b) Para este apartado definimos el entero `tiempo_minimo=27`, para forzar a que el tiempo total sea exactamente 27 (basándonos en el apartado anterior). El objetivo

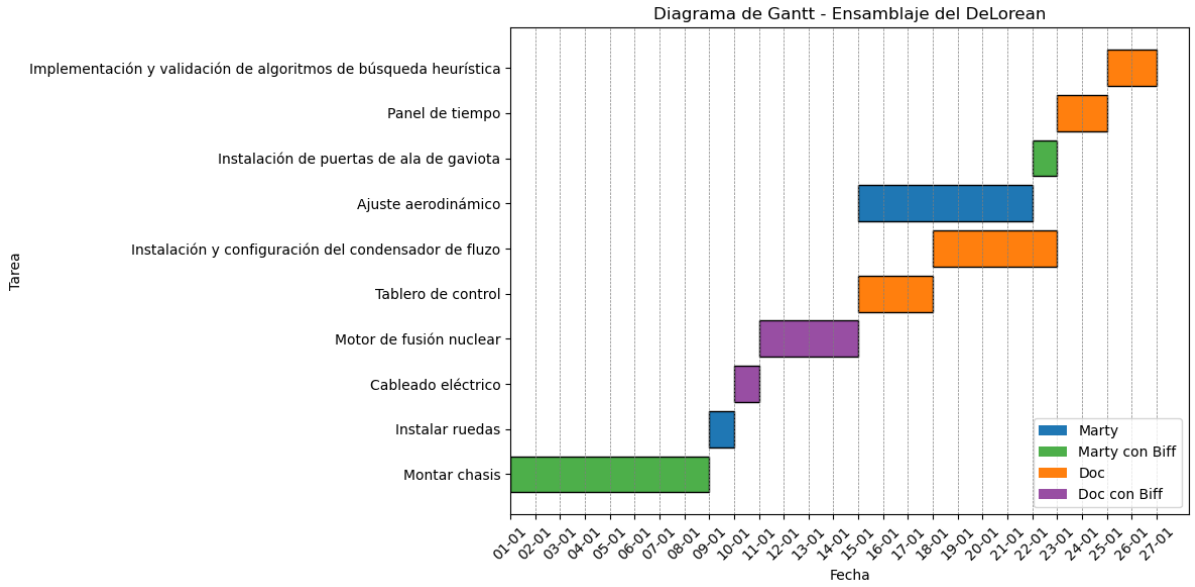


Figura 1: Diagrama de Gantt para el ensamblaje del DeLorean.

es encontrar asignaciones válidas que cumplan esa duración fija, así que usamos `solve satisfy;`.

El resultado es:

```
Tiempo total: 27 días
Asignación de tareas:
"Montar chasisïnicio día 1 fin día 9. Asignada a Marty con apoyo de Biff
Ïnstalar ruedasïnicio día 9 fin día 10. Asignada a Marty
Çableado eléctricoïnicio día 10 fin día 11. Asignada a Doc con apoyo de
Biff
"Motor de fusión nuclearïnicio día 11 fin día 15. Asignada a Doc con apoyo
de Biff
"Tablero de controlïnicio día 15 fin día 18. Asignada a Doc
Ïnstalación y configuración del condensador de fluzoïnicio día 18 fin
día 23. Asignada a Doc
.ªjuste aerodinámicoïnicio día 15 fin día 22. Asignada a Marty
Ïnstalación de puertas de ala de gaviotaïnicio día 22 fin día 23. Asignada
a Marty con apoyo de Biff
"Panel de tiempoïnicio día 23 fin día 25. Asignada a Doc
Ïmplementación y validación de algoritmos de búsqueda heurísticaïnicio
día 25 fin día 27. Asignada a Doc
-----
Tiempo total: 27 días
Asignación de tareas:
"Montar chasisïnicio día 1 fin día 9. Asignada a Marty con apoyo de Biff
Ïnstalar ruedasïnicio día 9 fin día 10. Asignada a Marty
Çableado eléctricoïnicio día 10 fin día 11. Asignada a Doc con apoyo de
Biff
"Motor de fusión nuclearïnicio día 11 fin día 15. Asignada a Doc con apoyo
de Biff
```

```
"Tablero de controlïnicio día 15 fin día 18. Asignada a Doc
Ïnstalación y configuración del condensador de fluzoïnicio día 18 fin
día 23. Asignada a Doc
.ªjuste aerodinámicoïnicio día 15 fin día 22. Asignada a Marty
Ïnstalación de puertas de ala de gaviotaïnicio día 22 fin día 23. Asignada
a Marty con apoyo de Biff
"Panel de tiempoïnicio día 23 fin día 25. Asignada a Marty
Ïmplementación y validación de algoritmos de búsqueda heurísticaïnicio
día 25 fin día 27. Asignada a Doc
```

En este caso los diagramas son el anterior y este nuevo:

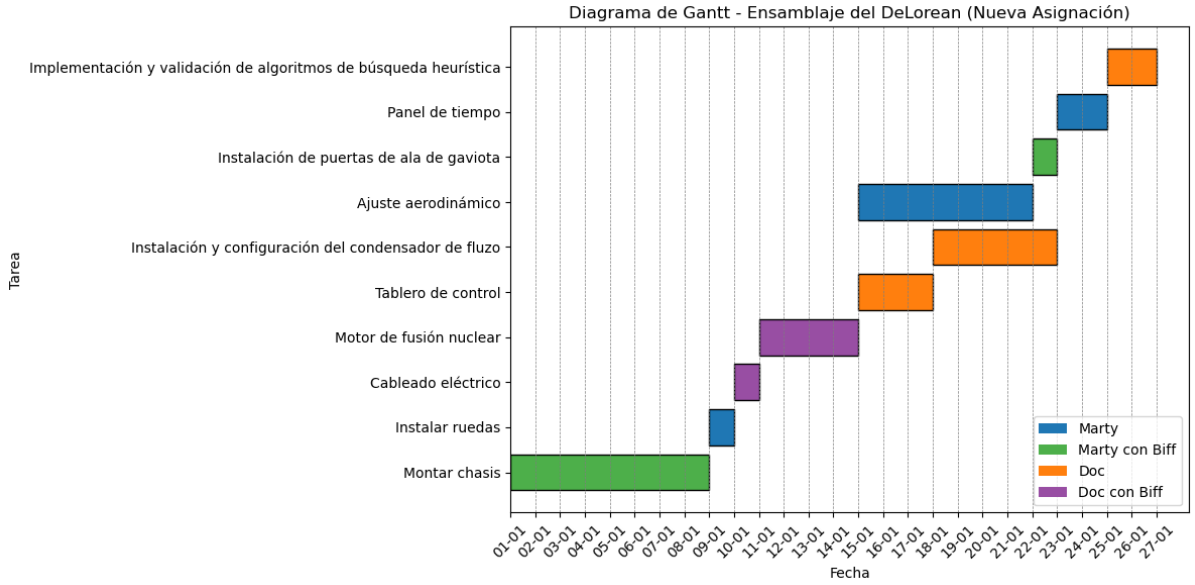


Figura 2: Segundo Diagrama de Gantt para el ensamblaje del DeLorean.

Ejercicio 6

Este ejercicio busca resolver el problema de asignación de tribunales para la defensa de TFGs. Para hacerlo se definen los conjuntos `dias` (lunes, martes,..., viernes), `profesores` (de 1 a 9), `tfgs` (de 1 a 10) y los departamentos `DECSAI` (de 1 a 3), `LSI` (de 4 a 6), `ICAR` (de 7 a 9). También definimos la `agenda` como un array de índices de profesores que guarda subconjuntos de días para los que están disponibles y `supervisados`, también un array de índices de profesores que guarda subconjutnos de TFGs que han supervisado.

Lo que buscamos es asignar valores a:

- `tribunal1` y `tribunal2`: arrays de índices de profesores que indica que si el profesor está o no en cada tribunal (1 ó 0).
- `diaTribunal1` y `diaTribunal2`: variables de días que indican cuándo evalúa cada tribunal.
- `evaluaTribunal1` y `evalauTribunal2`: arrays de indices de tfgs que indican si evalua dicho TFG el tribunal correspondiente o no (1 ó 0).

Las restricciones son:

- Cada TFG debe estar en un único tribunal: para cada tfg `evaluaTribunal1[t]=0` si `evaluaTribunal2[t]=1` o viceversa, es decir, su suma debe dar 1.
- Cada tribunal corrige 5 TFGs: la suma de 1s en `evaluaTribunalN` debe ser 5.
- Un profesor no puede estar en ambos tribunales, pero sí puede no estar en ninguno. Por tanto, la suma de `tribunal1[profe] + tribunal2[profe] <= 1` (dará 0 si no está en ninguno de los tribunales).
- Cada tribunal debe contar con 1 profesor de cada departamento. La suma de profesores de DECSAI en cada tribunal debe sumar 1, y así con todos los departamentos.
- Si un profesor está en un tribunal, el día asignado debe estar dentro de su agenda. Por tanto, para cada profesor asignado al tribunal 1, el dia del tribunal 1 debe estar en la agenda del profesor. Lo mismo para el tribunal 2.
- Un profesor no puede evaluar un TFG que ha supervisado. Por tanto, para cada profesor en el tribunal 1, los TFGs supervisados por dicho profesor no deben estar en `evaluaTribunal1`. Lo mismo para el tribunal 2.
- Para evitar simetría, se añade que el tribunal 1 debe tener el índice de TFG más bajo, es decir, el 1.

Para resolver el problema, se deben satisfacer todas las restricciones con `solve satisfy;`.

1. En total se han obtenido 6 soluciones válidas, es decir, 6 pares de asignaciones para el Tribunal1 y para el Tribunal2. Pero como son intercambiables los nombres de los tribunales, en realidad solamente hay 3 posibles asignaciones diferentes para los tribunales.

```
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 9, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 4, 5, 6, 7, 8
-----
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 5, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 4, 6, 7, 8, 9
-----
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 3, 5, 9, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 2, 4, 6, 7, 8
----- Tribunal 1: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 2, 4, 6, 7, 8
Tribunal 2: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 3, 5, 9, 10
-----
Tribunal 1: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
```

```
TFGs a evaluar: 4, 6, 7, 8, 9
Tribunal 2: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 5, 10
-----
Tribunal 1: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 4, 5, 6, 7, 8
Tribunal 2: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 9, 10
-----
```

2. Hay 6 soluciones válidas pero realmente son 3 soluciones únicas, ya que cada una tiene su solución espejo. Esto se debe a que el Tribunal 1 y el Tribunal 2 son intercambiables; si el Tribunal 1 evalúa en miércoles y el Tribunal 2 en viernes, se podría representar al revés y sería equivalente.

   Una forma de evitar estas simetrías es forzar un criterio de orden entre los tribunales. Algunas opciones son:

   - Obligar a que el Tribunal 1 siempre evalúe día antes que el Tribunal 2.
   - Obligar a que el Tribunal 1 tenga siempre el índice más pequeño de profesor.
   - Obligar a que el TFG de menor índice esté en el Tribunal 1. Esta es la opción que he tratado yo con la siguiente restricción:

     ```
     constraint min([t | t in tfgs where evaluaTribunal1[t] = 1])
     <min([t | t in tfgs where evaluaTribunal2[t] = 1]);
     ```

     De esta forma se consiguen 3 soluciones únicas.

```
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 9, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 4, 5, 6, 7, 8
-----
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 2, 3, 5, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 4, 6, 7, 8, 9
-----
Tribunal 1: 3(DECSAI), 4(LSI), 8(ICAR), (Día: miercoles)
TFGs a evaluar: 1, 3, 5, 9, 10
Tribunal 2: 2(DECSAI), 6(LSI), 7(ICAR), (Día: viernes)
TFGs a evaluar: 2, 4, 6, 7, 8
-----
```