UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

PROGRAMACIÓN CON RESTRICCIONES

Memoria de la Práctica 1. Satisfactibilidad y Optimización de Gestión de Turnos.

Realizado por:
Arturo Melero Ortiz

Profesores:
Alberto Rubio Gimeno, Pablo Gordillo Alguacil

Curso: **2023/2024**

Índice

1. Introducción.

1.1. Contextualización de la práctica.

El presente documento se corresponde con la memoria de la Práctica 1 de la asignatura *Programación con Restricciones* de la Facultad de Informática de la UCM. El objetivo de la misma consiste en la resolución de un problema de asignación de turnos en una fábrica, considerando diversas restricciones y condiciones específicas. Para la resolución del problema, se llevaron a cabo dos iteraciones del mismo:

1) Resolución del problema de satisfacción.
2) Resolución del problema aplicando diferentes estrategias de optimización.

En el documento, abordaremos primero las explicaciones pertinentes a las restricciones básicas del problema de satisfacción, incluyendo los *constraint asserts* propuestos, dejando los métodos de optimización para el último apartado. Se expondrá, además, un análisis de resultados de los casos de prueba creados y testados en diferentes resolutores.

1.2. Justificación de los ficheros de entrega.

La entrega realizada a través del portal virtual *Campus UCM* se compone de los siguientes archivos:

- Un archivo modelo ***satisfaccion.mzn*** que contiene la resolución de la primera parte del problema, con las restricciones base y los constraint asserts.
- Un archivo modelo ***optimizacion.mzn*** que contiene la optimización del problema base aplicando una restricción propia, consiste en tratar de distribuir equitativamente el número de días de descanso entre los empleados.
- Dos archivos modelo ***optimizacion1.mzn*** y ***optimizacion2.mzn*** que contienen las optimizaciones 1) y 2) del enunciado de la práctica respectivamente.
- Un fichero de texto ***casos.txt*** que incluye, de forma ordenada, todos los casos de prueba utilizados para realizar los test.

Se proporcionan diferentes archivos del apartado de optimización pues, dependiendo del problema, es necesario realizar cambios en las restricciones base para que funcionen correctamente, por lo que no basta con cambiar la función de optimización. En el presente documento se examinarán dichos cambios.

Por otro lado, *casos.txt* tiene ordenados los casos de prueba en el mismo orden en que se han expuesto los archivos en el listado anterior.

1.3. Modificaciones en la entrega.

Durante la revisión de la entrega, se observó que el "*grado de justicia"* en la distribución de los incumplimientos durante la optimización era cuestionable. Al final, al distribuir es necesario plantear qué se considera justo y a qué se le otorga mayor valor. El objetivo de la modificación de la entrega es precisamente discutir esta problemática, añadiendo nuevos planteamientos para realizar diferentes distribuciones. Para estos, no se proporcionan resultados de casos de prueba, solo el planteamiento de las funciones de minimización.

2. Problema de Satisfacción.

2.1. *Constraint Asserts* implementadas.

Durante la ejecución de los casos de prueba, el resolutor explora primero el árbol de soluciones en busca de alguna solución válida. De esta forma, si el árbol es lo suficientemente grande y el problema es insatisfacible para los datos de entrada, el resolutor tardará minutos en concluir que no hay solución. Por este motivo los *constraint asserts* son importantes: para facilitar la conclusión de insatisfacibilidad sin explorar el árbol de soluciones. De entre las posibles, se han planteado los siguientes:

1) Restricciones para los valores de los parámetros de entrada. Comprueba que los valores de los parámetros sean coherentes con la naturaleza del problema.

```
constraint assert (D >= 0 /\ T >= 0 /\ N1 >= 0 /\ N2 >= 0 /\ N3 >= 0 /\ MaxDT >= 0 /\ MaxDL >= 0 /\ MinDT >= 0 /\ A >= 0,
                                                                    "\n\t---> Los parámetros de entrada deben ser >= 0...");
```

2) Restricciones para la cantidad de encargados. El número de encargados debe ser, como mínimo, mayor o igual al número de turnos.

```
constraint assert (length(R) >= 3, "\n\t---> No hay suficientes encargados para cubrir los tres turnos...");
```

3) Debe haber, al menos, suficientes trabajadores para cubrir los turnos.

```
constraint assert (N1+N2+N3 <= T, "\n\t---> No hay suficientes trabajadores para cubrir los turnos...");
```

Sea la media de trabajo realizado por cada trabajador $m = \frac{D*(N1+N2+N3)}{T}$.

4) El valor de MinDT debe ser menor o igual a *m*, para que pueda cumplirse que todos trabajan al menos MinDT días. Si fuera mayor a *m*, significaría que al menos un trabajador no estaría cumpliendo con los días mínimos establecidos.

```
constraint assert (MinDT <= (D*(N1+N2+N3) / T),
                   "\n\t---> Imposible satisfacer que todos los trabajadores trabajen al menos MinDT dias...");
```

5) Para MaxDT es muy difícil encontrar alguna expresión válida que contemple todos los casos, por lo que nos limitamos a truncar dos casos conocidos donde no se cumple la restricción: si la función techo (*ceiling*) de la media es igual a D o el tamaño de R es 3, entonces MaxDT debe ser necesariamente igual o mayor a D.

```
% 5) Todos los trabajadores pueden trabajar como mucho MaxDT dias consecutivos
constraint assert (if (D = ceil(D*(N1+N2+N3) / T) \/ length(R) = 3) then MaxDT >= D else true endif,
                   "\n\t---> Imposible satisfacer que todos los trabajadores trabajen como mucho MaxDT dias...");
```

6) El número de personas por turno siempre debe ser mayor al número de personas afines por turno. Además, de forma muy sencilla se comprueba que, como mínimo, todas las personas tengan afinidad con otras A personas.

```
% 6) El numero de personas por turno siempre debe ser mayor o igual al numero de personas afines por turno
constraint assert (A < N1 /\ A < N2 /\ A < N3 /\ forall(i in 1..T) (sum(j in 1..T)(afines[i,j]) >= A),
                   "\n\t---> Imposible satisfacer la afinidad de los empleados en turno...");
```

Así, por ejemplo, considerando el caso proporcionado con MaxDT = 4 se puede verificar que $m = \frac{5*(3+3+4)}{12} = 4.167 > D - 1$, ahorrándonos la ejecución al pre-calcular de antemano su insatisfacibilidad.

2.2. Restricciones implementadas.

Se han implementado todas las restricciones propuestas del apartado de satisfacción.

```
% 1) Cada turno tiene el num de trabajadores ("N1", "N2" o "N3") que le corresponde.
constraint forall(i in 1..D)( sum(j in 1..T)(sol[i,j] = 1) = N1 /\
                              sum(j in 1..T)(sol[i,j] = 2) = N2 /\
                              sum(j in 1..T)(sol[i,j] = 3) = N3
);

% 2) Un trabajador solo puede estar en un turno cada dia. --> Se cumple por construccion

% 3) Dado un numero "MaxDT", garantizar que nadie trabaja "MaxDT" días consecutivos.
constraint forall(j in 1..T, i in 1..D - MaxDT) ((sum (k in i..i + MaxDT) (sol[k,j] != 0)) <= MaxDT);

% 4) Dado un numero "MaxDL", garantizar que nadie tiene "MaxDL" dias libres consecutivos
constraint forall(j in 1..T, i in 1..D - MaxDL) (sum (k in i..i + MaxDL) (sol[k,j] = 0) <= MaxDL);

% 5) Dado un numero "MinDT", garantizar que todos trabajan como minimo "MinDT" en los "D" dias.
constraint forall (j in 1..T) ( (sum(i in 1..D) (sol[i, j] != 0)) >= MinDT );

% 6) Si un trabajador hace el ultimo turno de un dia entonces no puede tener el primero del dia siguiente.
constraint forall (j in 1..T, i in 1..D-1) (sol[i,j] = 3 -> sol[i+1,j] != 1);

% 7) Si un trabajador hace el ultimo turno dos dias seguidos entonces tiene que librar el dia siguiente.
constraint forall (j in 1..T, i in 1..D-2) (sol[i,j] = 3 /\ sol[i+1,j] = 3 -> sol[i+2,j] = 0);

% 8) Dada una serie de parejas de trabajadores afines, que se indicarán con una matriz 1..T x 1..T de Booleanos "afines",
%    y un número "A", cada trabajador de un turno tiene que estar con al menos A trabajadores afines en ese turno.
constraint forall(i in 1..D, j in 1..T where sol[i,j] != 0) ( sum(k in 1..T where afines[j,k] /\ j!= k) (sol[i,j] = sol[i,k]) >= A );

% 9) Sea "R" el conjunto de trabajadores (que se obtendrá como un set of números de trabajador) que tienen la categoría de
%    encargados. En cada turno debe haber al menos un responsable.
constraint forall(i in 1..D, t in 1..3) ( exists(j in R)(sol[i,j] == t) );
```

Como nota, la restricción 2) para el sistema de representación seleccionado ("*qué turno tiene asignado cada trabajador cada día o si libra, identificado con el 0*"), se cumple por las restricciones implícitas del problema.

```

% Representacion: matriz de D filas x T columnas, con sol[i,j] turno que realiza el trabajador j en el dia i
array [1..D, 1..T] of var 0..3: sol;            % Turno asociado a cada trabajador por dia con | 0 -> libre    2 -> T2
                                                %                                             | 1 -> T1       3 -> T3

```

2.3. Resultados y tiempos de ejecución para los casos de prueba.

Tras garantizar la corrección de las restricciones aplicando variaciones del caso de prueba proporcionado con el enunciado, se presentan los resultados obtenidos para los 9 casos de prueba propuestos en los distintos resolutores para el problema base.

	Chuffed	COIN-BC	Gecode	HiGHS
Caso 1	77ms	99ms	169ms	108ms
Caso 2	104ms	129ms	155ms	139ms
Caso 3	97ms	160ms	149ms	191ms
Caso 4	**112ms**	**5s 744ms**	**244ms**	**1s 475ms**
Caso 5	339ms	329ms	1s 891ms	243ms
Caso 6	94ms	172ms	171ms	137ms

Caso 7	119ms	43s 451ms	248ms	2s 187ms
Caso 8	209ms	453ms	2s 991ms	413ms
Caso 9	54s 469ms	+15 min	19s 932ms	11s 865

Observaciones:

1) Los casos han sido ordenados primero en tamaño por número de días y segundo por número de trabajadores.
2) El caso 4, marcado en negrita, se corresponde con el proporcionado junto con el enunciado de la práctica.
3) Los casos 2 y 5 son casos insatisfacibles, siendo el 5 un caso de tamaño mayor.
4) Distintos resolutores recorren el árbol de exploración de forma diferente, por lo que los tiempos de ejecución difieren en favor de unos u otros según dónde se encuentre la primera solución posible.

Analizando los resultados de cada resolutor, concluimos:

- ***Chuffed***. En general, parece ser uno de los resolutores más eficientes en términos de tiempo de ejecución para encontrar la primera solución. En el caso 9, sin embargo, experimenta un tiempo de ejecución bastante largo (54 segundos) en comparación con los tiempos obtenidos por *Gecode* y *HiGHS*.
- ***COIN-BC***. Parece ser bastante rápido en la mayoría de los casos, pero muestra una variabilidad significativa en los tiempos de ejecución. Así, por ejemplo, en los casos 4, 7 y 9 experimenta tiempos de ejecución excepcionalmente largos.
- ***Gecode***. Muestra tiempos de ejecución moderados en la mayoría de los casos, pero en general es más lento que *Chuffed* y *COIN-BC*. No parece presentar tanta variabilidad como ese último, aunque sus tiempos de ejecución en los casos 5 y 8 son los mayores.
- ***HiGHS***. Parece ser bastante competitivo en términos de tiempo de ejecución, con resultados similares a *Chuffed* en varios casos. Tiempos de ejecución mayores para los casos 4 y 7.

De esta forma, los resultados para las pruebas planteadas sugieren que *Chuffed* es el resolutor con mejores tiempos, seguido de *HiGHS*. No obstante, los resultados ponen de manifiesto que todos ellos presentan debilidades ante casos específicos.

Empleando los casos de prueba más pequeños (casos 1, 3, 6) para buscar todas las soluciones posibles, al medir los tiempos concluimos lo mismo: *Chuffed* parece operar mejor, pero el resultado dependerá siempre del caso concreto que se ejecute.

	Chuffed	**Gecode**
Caso 1	78ms	154ms
Caso 3	2s 130ms	2s 330ms
Caso 6	685ms	427ms

3. Problema de Optimización.

3.1. Restricciones implementadas.

Para resolver el problema aplicando optimización, según se expone en 1.2. Justificación de los ficheros de entrega, se han implementado las dos restricciones propuestas y una adicional, todas ellas realizadas sobre el problema de satisfacción anterior.

3.1.1. Distribución uniforme de los días libres de los trabajadores.

No es necesario adaptar el código, ni añadiendo variables ni modificando restricciones. Se emplea una función que minimiza la diferencia entre el máximo y el mínimo número de días que descansa cualquier trabajador. Como veremos en el siguiente apartado, dado que los encargados suelen tener que trabajar más días que el resto de trabajadores, aunque no tarde mucho en encontrar una primera solución tiene dificultades en optimizar.

```
88
89 % Resolucion: ------------------------------------------------------------------------------------------------------------------------
90
91 solve minimize let { array [1..T] of var 0..D: libres = [sum(i in 1..D)(sol[i,j] = 0) | j in 1..T]; }
92                  in max(libres) - min(libres);
93
```

3.1.2. Solicitud de días libres de los trabajadores.

Se corresponde con el primero de los apartados propuestos de optimización, debiendo realizar cambios en algunas restricciones para su implementación. Se añade, además, un nuevo parámetro *peticion*, consistente en una matriz $D * T$ de booleanos donde $peticion[i, j] = true$ significa que el trabajador *j* solicita tomarse el día *i* de descanso.

```
19
20 array [1..D, 1..T] of bool: peticion;    % Peticion dias de vacaciones -> Optimizacion 1
21
```

Modificaciones de las restricciones 4) y 5) .

```
66
67 % 4) Dado un numero “MaxDL”, garantizar que nadie tiene “MaxDL” dias libres consecutivos
68 constraint forall(j in 1..T, i in 1..D - MaxDL) (
69                 sum (k in i..i + MaxDL) (sol[k,j] = 0 /\ not peticion[k,j]) <= MaxDL);
70
71 % 5) Dado un numero “MinDT”, garantizar que todos trabajan como minimo “MinDT” en los “D” dias.
72 constraint forall (j in 1..T) ( (sum(i in 1..D) (sol[i, j] != 0)) >= MinDT - sum(i in 1..D)(peticion[i,j]));
73
```

Por último, en cuanto a la función de optimización se consideraron diferentes estrategias para intentar no solo minimizar el número de incumplimientos, sino también distribuirlos. Primero, se consideró sólo minimizar los incumplimientos:

```
88
89 solve minimize ( sum(i in 1..D, j in 1..T)(peticion[i,j] /\ sol[i,j]!=0) )
90
```

Es decir, minimizar el número de veces que, habiendo pedido alguien el día libre, no se le había concedido. Después, se consideró además tratar de minimizar la diferencia entre la persona con el mayor y la persona con el menor número de incumplimientos.

```
88
89 solve minimize ( let { array [1..T] of var 0..D: incumplimientos = [sum(i in 1..D)(peticion[i,j] /\ sol[i,j] != 0) | j in 1..T]; }
90                  in sum(j in 1..T) (incumplimientos[j]) + (max(incumplimientos) - min(incumplimientos)) );
91
```

Siendo *incumplimientos* un array [1..T], donde incumplimientos[i] almacena el nº de incumplimientos para el trabajador *i*. Sin embargo, esta distribución no resulta justa para aquellos casos en los que algunos trabajadores realicen muchas peticiones frente a otros que soliciten pocas. Se decidió asignarle un peso relativo a los incumplimientos en función del nº de peticiones realizadas por dicho trabajador.

Finalmente, se optimizó aplicando la siguiente función:

```
93
94 solve minimize ( let { array [1..T] of var 0..D: incumplimientos = [sum(i in 1..D)(peticion[i,j] /\ sol[i,j] != 0) | j in 1.
95                        array [1..T] of var 0..D: peticiones = [sum(i in 1..D)(peticion[i,j]) | j in 1..T];
96                        var int: pTotal = sum(j in 1..T)(peticiones[j]); }
97                  in if pTotal != 0 then sum(j in 1..T) (incumplimientos[j] * peticiones[j]/pTotal) else 0 endif);
98
```

Se calcula un array incumplimientos[1..T] y otro peticiones[1..T] para almacenar el nº de incumplimientos y peticiones de cada trabajador y un entero pTotal con el nº total de peticiones. Si hay al menos alguna petición (para evitar división por cero), se minimiza la suma de los valores $\frac{incumplimientos[j]*peticiones[j]}{pTotal}$.

Tal como se menciona en 1.3. Modificaciones en la entrega, la distribución anterior puede no resultar justa para todos los trabajadores, pues prioriza minimizar los incumplimientos entre los trabajadores con mayor número de peticiones. Aunque este no fuera el objetivo inicial de la función de minimización anterior, pues se quería una distribución lo más justa posible, tiene sentido considerando:

1) El descontento aumenta cuantas más peticiones se incumplan, independientemente del número de peticiones que se realicen.
2) Todos los incumplimientos tienen el mismo valor.

Es especialmente importante el punto 2), pues no tiene la misma importancia la solicitud de un día libre por incapacidad de asistir al trabajo que el deseo de tener vacaciones en una franja temporal concreta. Al no tener en cuenta la restricción anterior, la distribución propuesta puede ser considerada justa si tenemos en cuenta el coste de oportunidad de un trabajador al decidir tomarse un día libre, pues son limitados, favoreciendo a aquellos que deciden gastar un mayor número de días de libre disposición. Además, esta distribución no castiga aleatoriamente a los trabajadores, sino que favorece a aquellos con un comportamiento determinado (solicitud de mayor número de días).

Para garantizar una distribución más justa, podría tomarse como dato de entrada otra matriz que incluya un código numérico con la importancia relativa de la solicitud, de forma que se prioricen las peticiones más urgentes primero y se repartan el resto equitativamente después. Por ejemplo, podría considerarse:

- Sin petición: valor 0.
- Petición urgente: valor 1.
- Petición por vacaciones: valor 2 (cinco o más días libres).
- Resto de peticiones: valor 3.

De esta forma, al realizar las distribuciones podríamos valorar también priorizar a aquellos trabajadores que deciden tomarse unas vacaciones para no darles un mal reparto durante la distribución de los incumplimientos (considerando, por ejemplo, que o se le conceden todos los días consecutivos o no se le concede ninguno).

Otra alternativa sería asignar para cada empleado un valor numérico que representase bien su antigüedad en la empresa o bien su rendimiento, de forma que al repartir se tuviese en cuenta dicho valor. Podría emplearse una escala de valores enteros, donde los más altos se correspondiesen con un rendimiento/antigüedad mayores. Así, al minimizar, el algoritmo trataría de evitar los incumplimientos con el factor multiplicativo mayor.

```
89
90 array [1..T] of int: rendimiento;    % En escala 1-10
91
92 solve minimize ( let { array [1..T] of var 0..D: incumplimientos = [sum(i in 1..D)(peticion[i,j] /\ sol[i,j] != 0) | j in 1..T]; }
93                  in sum(j in 1..T) (incumplimientos[j] * rendimiento[j])  );
94
```

Para la distribución anterior, si quisiéramos darle incluso más peso al valor asignado por trabajador, podría elevarse este parámetro al cuadrado. También, además, podríamos incorporar la matriz de urgencia de las peticiones y combinar ambas para conseguir una distribución más justa.

3.1.3. Solicitud de turnos que los empleados no desean realizar.

Se corresponde con el segundo de los apartados propuestos de optimización. Su resolución es similar al anterior salvo que no se requieren modificaciones en las restricciones anteriores. Por ello, se emplea una notación similar.

Como parámetro adicional, se añade una matriz $D * T$ de enteros, con el turno que no se desea trabajar por día (utilizando el 0 para indicar indiferencia).

```
19
20 array [1..D, 1..T] of 0..3: peticion;    % Peticion dias de vacaciones -> Optimizacion 1
21
```

Se ha escogido implementar esta representación por ser la otra una particularización de esta (escoger qué turno único no quiere un trabajador es una columna repleta del mismo valor). Además, se considera, por simplicidad, que si un trabajador desea no trabajar en el turno T2 en el día *i*, darle el día *i* libre es una alternativa válida para el trabajador.

Teniendo en cuenta las formas de distribuir los incumplimientos son similares al punto anterior, se presenta directamente la minimización teniendo en cuenta los pesos. En este caso, el nº de peticiones son todos aquellos valores distintos de 0, pues el 0 denota indiferencia.

```
88
89 % Resolucion: ------------------------------------------------------------------------------------------
90 solve minimize ( let { array [1..T] of var 0..D: incumplimientos = [sum(i in 1..D)(peticion[i,j] != 0 /\ sol[i,j] = peticion[i,j] ) | j in 1..T];
91                       array [1..T] of var 0..D: peticiones = [sum(i in 1..D)(peticion[i,j] != 0) | j in 1..T];
92                       var int: pTotal = sum(j in 1..T)(peticiones[j]); }
93                  in if pTotal != 0 then sum(j in 1..T) (incumplimientos[j] * peticiones[j]/pTotal) else 0 endif);
94
```

Ahora, el incumplimiento se produce cuando el turno asignado se corresponde con el valor del turno pedido. Esta representación permite, añadiendo un solo símbolo en la primera línea de la sección *let*, resolver un problema distinto. Cambiando

```
sol[i,j] != peticion[i,j] ) | j in 1..T];
```
→
```
sol[i,j] = peticion[i,j] ) | j in 1..T];
```

en la forma de calcular los incumplimientos, consideramos que un incumplimiento se produce cuando el valor pedido no coincide con el valor dado, por lo que el parámetro *peticiones* pasa a ser la **solicitud de turnos que los trabajadores desean realizar**. Así, con una misma representación y un solo cambio de signo, resolvemos tanto el problema de minimizar los incumplimientos en los turnos que los trabajadores no desean realizar y en los turnos que los trabajadores sí quieren.

3.2. Resultados y tiempos de ejecución para los casos de prueba.

Los casos de prueba son los mismos que para el problema de satisfacción, con modificaciones en aquellos en los que $N1 + N2 + N3 = T$, que han sido modificados para permitir pedir días libres. Recordar, además, que los casos 2 y 5 continúan siendo insatisfacibles.

Por otro lado, señalar que se concederán 20 minutos de ejecución a las pruebas realizadas sobre el caso proporcionado junto con la práctica y hasta 5 minutos para el resto de ejecuciones.

Se indicará con un "-" cuando la ejecución encuentre en dicho tiempo la solución óptima y determine que no hay una mejor. Para los casos 2 y 5 no se indicará, pues no hay soluciones válidas.

3.2.1. Distribución uniforme de los días libres de los trabajadores.

Es, de los problemas de optimización, el más parecido al problema de satisfacción, pues no se añaden parámetros ni se modifican restricciones: solamente minimizamos la diferencia entre el trabajador que descansa más y el que descansa menos. Esta forma de minimizar tiene el inconveniente de que los encargados generalmente descansan menos que el resto de trabajadores y el número de días que trabajan no puede ser minimizado muchas veces por cumplir con las restricciones del problema. Aún así, se obtienen asignaciones más justas en términos de descanso como primera solución que en el problema de satisfacción.

	Chuffed	**COIN-BC**	**Gecode**	**HiGHS**
Caso 1	71ms -	120ms -	138ms -	118ms -
Caso 2	107ms	166ms	174ms	141ms
Caso 3	93ms -	731ms -	154ms -	140ms -
Caso 4	**+20min****	**5min 30s**	**+20min ***	**1s 179ms -**
Caso 5	290ms	532ms	1s 665ms	271ms
Caso 6	114ms -	164ms -	261ms -	136ms -

Caso 7	146ms -	+5 min	1s 386ms -	3s 613ms -
Caso 8	+1s sol opt +5min sin fin	3s 195ms -	+3min sol opt +5min sin fin	577ms -
Caso 9	+7s sol opt +5 min sin fin	5min 19s -	+5 min	25s 339ms -

* < 500ms obtiene una primera solución no óptima. Tras 20 minutos de ejecución, no encuentra ninguna mejor.
** < 500ms en encontrar una solución óptima, +20min en determinar que no hay ninguna solución mejor.

De los resultados obtenidos, concluimos:

- ***Chuffed***. Tiene un tiempo de ejecución rápido en la mayoría de los casos, aunque en el Caso 4 parece haber una situación de atasco que requiere mucho más tiempo para resolver. Dificultades para casos grandes (Caso 8 y Caso 9).
- ***COIN-BC***. Tiene tiempos de ejecución más variables, con algunos casos en los que muestra un rendimiento comparable a Chuffed y otros en los que es significativamente más lento. Parece tener dificultades en algunos casos para encontrar soluciones óptimas o mejorarlas dentro de un límite de tiempo razonable.
- ***Gecode***. También muestra tiempos de ejecución variables, con resultados competitivos en algunos casos pero más lentos en otros. En general, parece tener dificultades en algunos casos para encontrar soluciones óptimas o mejorarlas dentro de un límite de tiempo razonable, similar a COIN-BC. Ofrece más pasos intermedios.
- ***HiGHS***. Tiene tiempos de ejecución ligeramente inferiores a *Chuffed* para los casos pequeños, pero muestra un rendimiento excepcional al encontrar soluciones óptimas rápidamente en los casos grandes donde el resto de resolutores se atascan.
- No hay diferencias significativas en los tiempos obtenidos para los casos 2 y 5 (insatisfacibles) salvo, quizás, para *COIN-C* (siendo estos menores).

En resumen, *HiGHS* parece ser el resolutor más consistente en términos de rendimiento y rapidez para encontrar soluciones, seguido de cerca por *Chuffed*. *COIN-BC* y *Gecode* muestran tiempos de ejecución más variables y pueden tener dificultades para encontrar soluciones óptimas en algunos casos.

3.2.2. Solicitud de días libres de los trabajadores.

Durante las ejecuciones, se descubrió que el resolutor *Chuffed* reportaba el siguiente error al optimizar:

```
Error: syntax error, unexpected FLOAT_LIT in line no. 90
=====ERROR=====
```

Dicho error se producía porque, en la función de minimización, la división entre *pTotal* al calcular los pesos generaba valores flotantes en lugar de enteros. Redondeando el resultado de la división $\frac{peticiones[j]}{pTotal}$ se generan resultados nuevamente, pero el comportamiento de la optimización ya no es el esperado. Dado que en la mayoría de los

casos el número de peticiones es elevado, el resultado de redondear termina siendo 0, por lo que no minimiza al redondear. Empleando la función techo, el resultado termina siendo contraproducente, pues no se produce una distribución equitativa de los incumplimientos.

Por este motivo, dado que no tiene sentido comparar tiempos de ejecución para funciones de optimización con complejidades distintas, ni medir los resultados en base a una distribución menos justa que la propuesta, no emplearemos *Chuffed* para medir los tiempos de ejecución.

	COIN-BC	**Gecode**	**HiGHS**
Caso 1	114ms -	144ms -	146ms -
Caso 2	177ms -	215ms -	167ms -
Caso 3	510ms -	224ms -	990ms -
Caso 3'	2s 682ms -	275ms -	562ms -
Caso 4	**31s 338ms -**	**5min 22s -**	**1s 549ms -**
Caso 4'	**1min 10s -**	**405ms -**	**842ms -**
Caso 5	370ms -	2s 140ms -	368ms -
Caso 6	173ms -	224ms -	204ms -
Caso 7	2 min 1ª respuesta 5min 20s -	1s 45ms -	5s 523ms -
Caso 8	550ms -	+7min	415ms -
Caso 9	+5 min	+10 min	7s 891ms -

En base a los resultados obtenidos, en líneas generales concluimos:

- ***COIN-BC*** muestra buenos tiempos de ejecución, aunque parece tener problemas con Casos 4' y 7 en comparación con los otros resolutores.
- ***Gecode*** tiene buenos tiempos en general, normalmente peores a *COIN-BC*. También sufre atascos en algunos casos, con tiempos de ejecución mayores al resto. Es el que ofrece más pasos intermedios.
- ***HiGHS*** Aunque no posea los tiempos más rápidos, en los casos más complejos funciona excepcionalmente bien.

En general, Gecode parece ser el resolutor más lento, mientras que HiGHS muestra resultados consistentes y ejecuciones rápidas para los casos complejos. Aún así, se necesitarían más pruebas para determinar con precisión cuál es el mejor resolutor en todos los escenarios posibles.

3.2.3. Solicitud de turnos que los empleados no desean realizar.

Al optimizar de un modo parecido, empleando pesos para valorar los incumplimientos de las peticiones de los trabajadores, tampoco se analiza el rendimiento de *Chuffed*.

Salvo para los dos últimos casos y los insatisfacibles, para cada caso de prueba se realizan dos, una con algunas peticiones y otra con columnas repletas del mismo valor, simulando la implementación alternativa donde los trabajadores seleccionan un turno que no desean realizar nunca.

	COIN-BC	**Gecode**	**HiGHS**
Caso 1	137ms -	187ms -	129ms -
Caso 1'	136ms -	186ms -	167ms -
Caso 2	162ms -	206ms -	157ms -
Caso 3	769ms -	231ms -	201ms -
Caso 3'	738ms -	315ms -	271ms -
Caso 4	**36s 593ms -**	**2s 151ms -**	**1s 706ms -**
Caso 4'	**4min 17s -**	**57s 132ms -**	**4s 24ms -**
Caso 5	320ms -	1s 590ms	328ms -
Caso 6	187ms -	209ms -	206ms -
Caso 6'	147ms -	215ms -	162ms -
Caso 7	+5min	42s 614ms -	11s 842ms -
Caso 7'	3min 28s -	50s 10ms -	20s 197ms -
Caso 8	562ms -	+7min	403ms -
Caso 9	+7min sin encontrar sol	+7min sin encontrar sol	1ªsol en 11s 3min 49s -

De los resultados obtenidos, observamos:

- ***COIN-BC*** muestra buenos resultados, aunque con tiempos más largos en Caso 4 y Caso 7 en comparación con otros resolutores.
- ***Gecode*** tiene tiempos de ejecución generalmente buenos, pero parece tener problemas con Caso 9, donde no se encuentra la solución.
- ***HiGHS*** tiene indudablemente los mejores tiempos de ejecución, además de los más consistentes.

En definitiva, con las pruebas realizadas se manifiesta la importancia de la elección de un resolutor apropiado para obtener un buen tiempo de ejecución y que, además, esta elección no depende únicamente de las características del problema a resolver, sino también de la entrada específica. Aún así, para los problemas de optimización HiGHS parece ser el resolutor más rápido y Gecode el que muestra el mayor número de pasos intermedios.