Práctica 1

Alejandro Luque Villegas, Maria Solórzano Gómez
Comentarios sobre la Práctica 1

April 5, 2025

1 Codificación de la solución

Lo primero que se va a comentar es la codificación de la solución que se ha decidido usar. En la sección de abajo podemos observar cómo se ha implementado la solución. Se usa una matriz de días por total_turnos, donde total_turnos es la suma de todos los turnos que puede haber en un día.

```
array[1..D, 1..total_turnos] of var 1..T : sol;
```

1.1 Problemas de satisfacción

En esta sección se va a comentar cómo se han implementado las restricciones del apartado de satisfacción. También se comentarán las distintas entradas que ayudan a comprobar la corrección de dichas restricciones.

1.1.1 Turnos correspondientes

Para esta sección se ha usado un *assert* para comprobar que hay suficientes trabajadores para cumplir con todos los turnos posibles. Para comprobar la corrección de las restricciones, se ha hecho un archivo de prueba. El archivo $trabajadores\_insuficientes.dzn$ es un archivo muy simple que comprueba que el *assert* es válido. El *assert* comprueba que haya suficientes trabajadores para que cada uno trabaje solo un turno al día si fuera necesario, ya que esa es la restricción que se trata en la siguiente sección. Para la corrección de las restricciones no hace falta un archivo de prueba, ya que es necesario que se cumpla, pues usa un alldifferent.

Para el alldifferent se usa x, donde x es uno de los días (usando un forall para comprobar todos los días), e y, donde y va desde el inicio del turno al final del turno. Comprobando que para cada turno, todos los trabajadores son distintos.

```
constraint assert (T >= total_turnos,
                   "No hay suficientes trabajadores para cubrir los turnos.");
constraint forall (x in 1..D)(alldifferent([sol[x, y] | y in 1..N1]));
constraint forall (x in 1..D)(alldifferent([sol[x, y] | y in N1+1..N1+1+N2]));
constraint forall (x in 1..D)(alldifferent([sol[x, y] | y in N1+N2+2..total_turnos]));
```

1.1.2 Un solo turno

La segunda restricción es que cualquier trabajador solo pueda trabajar un turno cada día. En esta restricción también se tiene en cuenta la segunda restricción adicional que añade que un trabajador solicite la posibilidad de doblar turnos, es decir, trabajar 2 turnos consecutivos, lo que implica que al trabajador que ha solicitado esta modalidad no se le pueda aplicar la restricción.

Por lo tanto, se ha usado un alldifferent con la adición de un where donde se comprueba que solo se pueda aplicar la restricción a aquellos trabajadores que no doblan turnos. El código que queda es el siguiente:

```
constraint forall(x in 1..D) (alldifferent([sol[x, y] | y in 1..total_turnos
                                                  where TrabDob[sol[x,y]] != 1]));
```

1.1.3 Máximo de días trabajados consecutivos

Para esta restricción usamos una ventana, que va de 1 a $D - MaxDT$. Esto se hace para eliminar comparaciones innecesarias. Después, en la ventana se comprueba, para cada trabajador, que ese trabajador no tenga más de $MaxDT$ días de trabajo consecutivos. Para probar esto se ha creado un archivo de prueba, *dias_consecutivos.dzn* en el que se comprueba que si no se cumple la restricción da unsatisfiable.

1.1.4 Máximo de días libres consecutivos

Vamos comprobando para todos los trabajadores que no han solicitado doblar turno y vamos comprobando en una ventana de tamaño $MaxDL$ y vamos sumando 1 si el trabajador coincide con el actual y no es un día indeseado para el mismo.

1.1.5 Mínimo de días trabajados

Antes de implementar esta restricción tenemos un *assert* para comprobar si siquiera es posible que todos trabajen el mínimo de días. Para ello, se usa el siguiente *assert*:

```
constraint assert ((T * MinDT/total_turnos) <= D,
                   "Hay demasiados trabajadores para que todos trabajen el minimo.");
```

Además de este *assert*, se recorren todos los días teniendo en cuenta cada trabajador y comprobamos que la suma de los días trabajados. Para todos estos la suma tiene que ser mayor o igual que $MinDT$.

Para comprobar la validez de este *assert* se ha creado un archivo de prueba *minimo_dias.dzn*

1.1.6 Tiempo entre turnos

Para esta restricción decidimos que solo es válida la negación de que el trabajador trabaje el último turno del día actual, y el primer turno del siguiente.

1.1.7 Dos turnos de noche

Para esta restricción se ha usado un principio similar al de la restricción anterior. Para cada trabajador, no trabaja 2 turnos de noche seguidos, o no trabaja el tercer día (el de después de los 2 turnos de noche).

1.1.8 Afinidad entre trabajadores

Antes de implementar esta restricción, tenemos 2 *assert*:

```
constraint assert(A<=T, "Hay demasiado pocos trabajadores para que sean afines");
constraint assert(A<= N1 /\ A <= N2 /\ A <= N3,
                  "Hay demasiado pocos turnos para que trabajen afines");
```

1. Se comprueba que no se tenga que tener más afinidad por turno que trabajadores disponibles.
2. Se comprueba que no se tenga que tener más afinidad que huecos en un turno.

Seguido esto tenemos 3 restricciones, una por turno, que confirma que la suma de los trabajadores afines de cada trabajador es mayor que A. Para comprobar esta restricción se ha creado $afinidad.dzn$ para mostrar la corrección de las restricciones.

1.1.9 Encargado

Tenemos 3 restricciones, una para cada turno que comprueba que existe un encargado en cada turno para cada uno de los días. De nuevo, para comprobar se ha creado *encargados.dzn* para mostrar la funcionalidad.

1.2 Restricciones adicionales

1.2.1 Turnos antiestrés

Para esta restricción comprobamos todos los trabajadores que hayan solicitado un turno antiestrés. Para ello se ha usado la misma idea que en las restricciones 6 y 7. Usando una

ventana de $D-6$ días, se mira que no trabaje 2 turnos de mañana seguidos (d y $d+1$), 2 turnos de tarde seguidos ($d+2$ y $d+3$) y uno de noche($d+4$)o que libre 2 días siguientes ($d+5$ y $d+6$).

1.2.2 Doblar Turnos

Aquí hemos tenido que usar 2 restricciones distintas, ya que también se tiene que tener en cuenta los días libres. Para cada trabajador que ha solicitado doblar un turno se hace lo siguiente. En una ventana de $D-MaxDL$ el trabajador tiene que trabajar al menos un día desde d hasta $d+MaxDL$ o haya trabajado 2 turnos el día anterior ($d-1$).

La segunda restricción comprueba que para cada día, hasta $D-1$ el trabajador no doble turno o no trabaje el día siguiente ($d+1$).

1.3 Problemas de optimización

1.3.1 Días indeseados

Para minimizar el número de días indeseados, hacemos que la función a minimizar sea cuantas veces trabaja un trabajador en un dia que no desea trabajar (almacenado en un array de tmaño trabajadores x dias). Si deseasemos que los incumplimientos de estos deseos se repartiesen entre los trabajadores, se podría reducir la media, o la diferencia entre el trabajador que mas veces trabaja en dias indeseados y el trabajador que menos dias trabaja en esos dias.

1.3.2 Turnos indeseados

Para minimizar el número de turnos indeseados hacemos que la función a minimizar sea el número de turnos que trabaja un trabajador que no desea trabajar ese truno, así asegurando que el número de turnos indeseados sea el mínimo. Se optó por hacer los turnos indeseados generales, en lugar de específicos de cada dia, solo por simplicidad, pero podría expandirse haciendo la matriz tridimensional trabajadores x dias x turnos, en lugar de la actual, que es trabajadores x dias.

1.4 Otros comentarios

Una restricción que se ha añadido para reducir el espacio de búsqueda en caso de que no sea una solución válida es que los trabajadores se coloquen en orden de identificador por cada turno. Para ello se ha usado la siguiente restricción:

```
constraint forall(d in 1..D)(forall(t in 1..N1-1)(sol[d,t] < sol[d, t+1]));
constraint forall(d in 1..D)(forall(t in N1+1..N1+N2)(sol[d,t] < sol[d, t+1]));
constraint forall(d in 1..D)(forall(t in N1+N2+2..total_turnos-1)(sol[d,t] < sol[d, t+1]));
```

En cuanto a los resolutores se ha ido probando varios resolutores distintos y por prueba y error el que mejor resultados nos da es. Usando Geocode 6.3.0 para los problemas de satisfacción, la media es de entorno 320ms. Usando Chuffed 0.13.2 obtenemos resultados en tiempos en torno a 350ms. Para HiGHS la media rondaba en torno a los 18 segundos.

Cabe destacar que esto cambia cuando se trata de los problemas de optimización, ya que para estos HiHGS es el más rápido.