UNIVERSIDAD DE GRANADA

Técnicas de los Sistemas Inteligentes
grado en ingeniería informática

Práctica 2

Satisfacción de Restricciones

Autor
José María Sánchez Guerrero

Rama
Computación y Sistemas Inteligentes || Grupo 2 - Pablo Mesejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

Introducción

Esta práctica consiste en la resolución de una serie de ejercicios utilizando el lenguaje de modelado de restricciones ***MiniZinc***. En este programa, simplemente tendremos que escribir las variables y restricciones que se nos pidan en el ejercicio, y posteriormente ejecutarlo. Sin embargo, tendremos distintas configuraciones para el solucionador o "*solver*" a utilizar, y tras varias pruebas, el que mejor funcionaba es el "***Chuffed 0.10.4***", ya que nos ofrece la solución completa y óptima.

Ejercicio 1

En este problema se nos pide codificar 9 letras con un dígito diferente para cada una (entre el 0 y el 9). Esta asignación de dígitos tiene que satisfacer la siguiente suma:

$$TESTE + FESTE + DEINE = KRAFTE$$

Para resolverlo hemos utilizado dos *sets* de enteros, uno con el número de dígitos/letras que hay, y el otro con los posibles valores que pueden tomar. También hemos utilizado un array con los posibles valores para aplicarle las posteriores restricciones; junto a otro pero de string para poder imprimir las letras en el resultado (este último no intervendrá en las operaciones).

Las primera restricción necesaria ha sido la función "*all_ different()*", que servirá para que todos los valores que puede adoptar el array solución, sean distintos. La segunda restricción consiste en la propia suma, de forma que asigno a cada posición un valor de la suma, segmentando el número en unidades, decenas, centenas....

Finalmente, imprimo la solución gracias al array de string declarado anteriormente: **T=7 | E=0 | S=6 | F=8 | D=9 | I=5 | N=3 | K=2 | R=4 | A=1**

$$30.830 + 60.830 + 50970 = 142.630$$

Ejercicio 2

En este ejercicio tenemos que encontrar un número de 10 cifras, en el cual se cumpla que el primer dígito sea el número de 0s que hay en el propio número, el segundo sea el número de 1s, (...), y el último sea el número de 9s.

Para ello hemos utilizado un array de enteros de tamaño 10 y que valores de cada posición tengan un valor entre el 0 y el 9. Para asignar este valor, utilizamos

la función "*forall()*" para recorrer todas las posiciones del array. En cada una de ellas, realizaremos un conteo de las todas las posiciones con la función "*count()*", y comprobamos que el número contado es igual al de la posición en el array.

Si ejecutamos, el resultado que obtenemos es el mismo que en el ejemplo: $\mathbf{X} =$ **6210001000**

Ejercicio 3

Tenemos que encontrar un horario para distribuir a 6 profesores en un aula durante 6 horas. Cada clase dura 1 hora, pero cada profesor tiene distintas restricciones en sus horarios. Se ha representado también como un array, donde cada posición representa a un profesor y sus valores serán las horas disponibles.

La primera restricción la vamos a solventar con la función "*all_ different()*" vista anteriormente, para que ningún profesor pueda estar en el aula a la misma hora que otro. Las siguientes líneas del código corresponderán con las expresiones lógicas que satisfacen los distintos horarios de los profesores.

El resultado final que obtendríamos es el siguiente:

Profesor 1 da 1h de clase a las 14:00h

Profesor 2 da 1h de clase a las 11:00h

Profesor 3 da 1h de clase a las 13:00h

Profesor 4 da 1h de clase a las 10:00h

Profesor 5 da 1h de clase a las 12:00h

Profesor 6 da 1h de clase a las 9:00h

Ejercicio 4

Este ejercicio también consiste en la creación de un horario para los profesores, pero esta vez con distintas condiciones. Dispondremos de 4 aulas, 4 profesores y 3 asignaturas con 4 grupos cada una, por lo que la representación en este ejercicio será distinta al anterior.

Utilizaremos una matriz donde las filas serán los periodos de tiempo, y las

columnas los profesores. Los valores que tomará esta matriz vienen determinados por otro array que representa a las distintas asignaturas con sus grupos correspondientes; sin embargo, como tenemos menos asignaturas que horas disponibles, he añadido 4 más, llamadas *'EMPTY'*, y serán horas libre para el profesor al que le corresponda.

También he implementado otra matriz igual que la anterior, pero que sus valores representan el grupo que está dando una clase. Gracias a esto, y a la funnción "*alldifferent()*", que en este caso la aplico por filas, evitamos que el mismo grupo este dando dos clases a la misma hora.

En cuanto a las restricciones, he utilizado una estructura similar a la del ejercicio anterior, con expresiones lógicas. Al tener dos matrices, vamos asignando asignatura y grupo paralelamente; y también he utilizado "*forall()*" para recorrerlas más fácil, por ejemplo:

```
forall(h in HOURS) ( (schedule[h,1] == 2 ∩ groups[h,1] == 2)
```

Para todas las horas, el profesor 1 puede cursar la asignatura 2 ('IA') en el grupo 2. Finalmente, el resultado de la matriz es el siguiente:

	1	**2**	**3**	**4**
9:00-10:00	IA-G2	FBD-G1	FBD-G3	Empty
10:00-11:00	TSI-G1	Empty	TSI-G4	Empty
11:00-12:00	IA-G1	FBD-G2	TSI-G3	IA-G4
12:00-13:00	TSI-G2	Empty	TSI-G4	IA-G3

Ejercicio 5

Este ejercicio es muy similar al ejercicio anterior. Tenemos que encontrar una asignación de horarios que satisfagan una serie de condiciones. Lo que cambia en este ejercicio son las asignaturas y el tiempo, ya que tendremos que cumplir un número de horas semanales, impartirlas por bloques. También tendremos más restricciones en cuanto a profesores u horas que tienen que estar libres.

Lo primero que haremos es definir las variables horas y días, los arrays asignaturas, las horas por asignatura a la semana, y el profesor que imparte cada asignatura. Después creamos dos matrices que representen el horario semanal (horas como filas y días como columnas), uno para las asignaturas y otro para los profesores, los cuales rellenaremos gracias a los valores de la matriz asignatura y el número del profesor, respectivamente.

Las primeras restricciones serán dos "*all_ different()*" para evitar que en un dia se de más de una clase del mismo bloque, y el otro para evitar que un profesor imparta más de un bloque al día. A continuación, ponemos que el cuarto periodo sea el recreo para todos los días, y que cada una de las asignaturas imparta sus horas correspondientes gracias al array declarado previamente.

Pasamos a poner las restricciones de horarios. Empezamos poniendo que las clases que duran 2 horas no puedan emepzar antes del recreo o antes de irse a casa. Para ello simplemente recorremos la matriz por días y con una expresión lógica decimos que esas horas no puedan ser iguales a las asignaturas correspondientes. La otra restricción de horario que hay que tener en cuenta es que cuando comience una asignatura de 2 horas, la siguiente sea esta misma asignatura. En nuestro caso, para facilitarnos las cosas, hemos decidido poner en el propio array de las asignaturas una asignatura AX - parte 2, por ejemplo, *A1-2*. Gracias a esto, recorriendo todos los valores de la matriz (excepto los últimos y los recreos) y asignamos las segundas partes correspondientes.

Por último nos queda la restricción de los profesores, para la cual utilizamos la otra matriz creada al principio. Ésta la rellenaremos gracias a la matriz de las asignaturas ya obtenida, y dependiendo de que valores tengamos en ella, asignamos un valor u otro a la de profesores. Aclarar que no se asignan profesores, si no valores. Esto lo hacemos para tener un profesor distinto gracias a las primeras restricciones. Las condiciones implementadas en el código nos servirán para poner el mismo valor a las asignaturas dadas por el mismo profesor.

El resultado obtenido tras ejecutar es el siguiente:

	1	**2**	**3**	**4**	**5**
8:00-9:00	A7 (Prof:4)	A3 (Prof:1)	A1 (Prof:1)	A5 (Prof:2)	A4 (Prof:2)
9:00-10:00	A9 (Prof:3)	A3 (Prof:1)	A1 (Prof:1)	A5 (Prof:2)	A4 (Prof:2)
10:00-11:00	A2 (Prof:4)	A6 (Prof:3)	A6 (Prof:3)	A7 (Prof:4)	A2 (Prof:4)
11:00-12:00	Recreo	Recreo	Recreo	Recreo	Recreo
12:00-13:00	A4 (Prof:2)	A5 (Prof:2)	A8 (Prof:4)	A1 (Prof:1)	A3 (Prof:1)
13:00-14:00	A4 (Prof:2)	A5 (Prof:2)	A8 (Prof:4)	A1 (Prof:1)	A3 (Prof:1)

Ejercicio 6

Ahora se nos plantea un problema que consiste en adivinar dónde está la cebra y qué persona bebe agua en un vecindario. Este vecindario esta formado por 5 personas, las cuales tienen una serie de características que las diferencian unas de otras: región, profesión, animal, bebida, color de la casa y posición de la casa con respecto a los demás (están en una línea recta).

La forma en la que se ha resuelto el problema es muy sencilla. He creado una matriz de tamaño $5x6$ (número de vecinos x características) y cada uno de los valores que tomará una celda vienen definidos por lo siguiente. Las características serán arrays de strings con todas las características disponibles, asi que cada valor de la matriz será la posición (int) que tiene esa característica en el array.

Una vez hecho esto, lo siguiente será simplemente poner las expresiones lógicas correspondientes:

```
% El vasco vive en la casa roja.
constraint matrix[1,4] == 1;
```

siendo la primera fila de la matriz el vasco, la cuarta columna el color de la casa y '*colors[1]*' el rojo. El resultado final es el siguiente:

El vasco es escultor, tiene caracoles, bebe leche, su casa en roja y vive en la posicion centro

El catalan es violinista, tiene perro, bebe zumo, su casa en blanca y vive en la posicion centroderecha

El gallego es pintor, tiene cebra, bebe cafe, su casa en verde y vive en la posicion derecha

El navarro es medico, tiene caballo, bebe te, su casa en azul y vive en la posicion centroizquierda

El andaluz es diplomatico, tiene zorro, bebe agua, su casa en amarilla y vive en la posicion izquierda

Por lo tanto, la **CEBRA** la tiene el gallego, y el andaluz bebe **AGUA**.

Ejercicio 7

En este ejercicio disponemos de la informaciónn necesaria para construir una casa paso a paso. Viene representada en una tabla, donde la primera columna son los identificadores de las tareas necesarias para construirla, la segunda sus nombres, la tercera el tiempo que se tarda en hacerla y la cuarta las tareas que tienen que estar hechas antes de realizar la actual (relación de precedencia).

Empezamos declarando un array de string con todas las tareas que llevaremos a cabo, y otro con la duración de cada una de ellas. A continuación, guardamos en una variable el tiempo máximo que pueden tardar las tareas (si las ejecutamos una

tras otra).

También tenemos una matriz de $9x2$ para las precedencias, donde la primera columna es la tarea actual y la segunda es la tarea que no se puede ejecutar hasta que termine. El resto de variables que declaramos son los tiempos en los que empieza a construir (`start`)(ponemos 0 porque el día 1 ya lo estaríamos contando como un día de trabajo), los tiempos en los que finalizan las tareas (`end`), y el tiempo total que lleva construyéndose la casa.

Ahora vamos a pasar a las restricciones utilizadas. La primera de ellas es la función "`maximum()`", y con la cual nos quedamos siempre con el valor máximo que tiene la variable '`end`', que será el valor a minimizar.

Pasamos a la función "`cumulative()`", que es la más importante en este ejercicio. Ésta acepta como parámetros nuestros arrays 'start' y el de duración de cada tarea, y con ellos calcular el tiempo que tardarían una serie de trabjadores en realizarlas. Estos trabjadores se declaran en los siguientes parámetros, pero como en nuestro caso tenemos siempre uno disponible, se ha asignado uno para cada tarea y 1000 disponibles (un número ilimitado de éstos, como dice en el enunciado).

Por último nos queda hacer dos "`forall()`". Uno que recorra las tareas y vaya sumando a la variable `end` la suma del inicio más la duración; y otro que recorra las precedencias y evite que se ejecuten las que tienen un tiempo más alto que sus predecesoras. También nos queda comentar que, como se nos dice el enunciado, utilizamos la función "**solve minimize total_time**" con la variable del tiempo máximo obtenido antes.

El resultado final es el siguiente:

Tiempo total: 18
Levantar_muros: empieza el dia 0, tarda 7 (termina el dia 7)
Carpinteria_de_tejado: empieza el dia 7, tarda 3 (termina el dia 10)
Tejado : empieza el dia 11, tarda 1 (termina el dia 12)
Instalacion_electrica : empieza el dia 7, tarda 8 (termina el dia 15)
Pintado_fachada : empieza el dia 16, tarda 2 (termina el dia 18)
Ventanas : empieza el dia 15, tarda 1 (termina el dia 16)
Jardin : empieza el dia 17, tarda 1 (termina el dia 18)
Techado : empieza el dia 13, tarda 3 dias (termina el dia 16)
Pintado_interior : empieza el dia 16, tarda 2 dias (termina el dia 18)

Ejercicio 8

Este ejercicio es exactamente igual que el anterior, con el único cambio de que esta vez sí que tenemos un número de trabajadores limitados (3), y además, cada una de las tareas requiere una cantidad de trabajadores determinada.

Para solventar esta nueva restricción, añadimos una variable para los trabajadores disponibles y otro array con los necesarios para realizar una determinada tarea. Estos tiempos son los que están en la tabla del enunciado. Como dijimos en el ejercicio anterior, la función encargada de manejar estos tiempos es la función "`cumulative()`", en la que pusimos infinitos trabajadores y que cada tarea sólo necesitaba a uno. Pues en este caso, en el parámetro que nos dice los trabajdores necesarios por tarea meteremos el array, y en el siguiente los trabajdores disponibles (3 en este caso).

```
cumulative(start, duration, workers, available_workers)
```

El resto del programa se mantendría exactamente igual. El resultado final si ejecutamos es el siguiente:

Tiempo total: 22
Levantar_muros: empieza el dia 0, tarda 7 (termina el dia 7)
Carpinteria_de_tejado: empieza dia 15, tarda 3 (termina el dia 18)
Tejado : empieza el dia 18, tarda 1 (termina el dia 19)
Instalacion_electrica : empieza el dia 7, tarda 8 (termina el dia 15)
Pintado_fachada : empieza el dia 19, tarda 2 (termina el dia 21)
Ventanas : empieza el dia 19, tarda 1 (termina el dia 20)
Jardin : empieza el dia 21, tarda 1 (termina el dia 22)
Techado : empieza el dia 12, tarda 3 dias (termina el dia 15)
Pintado_interior : empieza el dia 20, tarda 2 dias (termina el dia 22)

Ejercicio 9

Ahora tendremos que resolver los ejercicios anteriores pero con otra tabla de trabajadores. Esta vez tenemos 3 trabajadores, para las tareas solo nos hace falta 1, pero cada uno las realiza en un tiempo distinto. Tendremos que asignar un trabajador a cada tarea de forma que minimizen el tiempo al realizarlas todas.

También partiremos de los ejercicios anteriores, por lo que sólo comentaré los

cambios respecto a estos. Tendremos 3 trabajadores disponibles, pero no tendremos un array de trabajdores necesarios como en el anterior, ya que sólo se necesita 1 por tarea. El array de tiempos ahora se transformará en una matriz, siendo las filas las tareas, las columnas el trabajador y los valores el tiempo que tarda cada trabajador por tarea. La duración total será, en este caso, la suma de todas las duraciones, y necesitaremos dos variables correspondientes a la duración máxima en la que se puede hacer una tarea determinada (`maxd`) y la duración mínima (`mind`). Con esto ya podremos crear las variables `start` y `end` que teníamos anteriormente.

Como tenemos opcionales puntos de inicio y duraciones opcionales para cada tarea, tenemos que definir dos variables más que representen estos conjuntos. La primera será un array de duraciones acotado entre `mind` y `maxd` y el segundo un array con distintos tiempos. Este último es una matriz formada por 'var opt', el cual es un tipo de variable de decisión que puede representar la propia decisión pero con otra posibiladad T. En el manual de MiniZinc encontraremos más información de esta variable, ejemplificada además con un ejemplo muy similar a este ejercicio. A continuación, explicaremos cómo utilizamos estas dos nuevas variables.

Las restricciones en este ejercicio son un poco más complejas que en los anteriores. Aun así, tanto la función que asigna el tiempo total utilizado ("`maximum()`"), como los dos "`forall()`" que actualizan las duraciones de cada tarea y comprueban las precedencias, sólo han cambiado la variable duración por la de duraciones opcionales, asi que nos ahorraremos explicarlas nuevamente. La función "`cumulative()`" debido a las siguientes restricciones no la necesitaremos.

Por último, tenemos las dos nuevas restricciones necesarias para resolver este ejercicio correctamente. En la primera utilizamos la función "`disjunctive()`" para todos los trabajadores. Esta función recibe dos parámetros, el array con distintos tiempos de inicio y la matriz de duraciones. Esta función hace que un trabajador pueda estar solo en una tarea por franja de tiempo, de forma que comprueba para cada tarea y punto de inicio específicos, que no se solape con ninguna otra (no pueden superponerse el tiempo de inicio + duración).

Para la segunda restricción utilizamos la función "`alternative()`", la cual nos asegura que cada tarea se ejecuta una sola vez. Comprueba que los dos primeros argumentos pasados a la función se encuentren en los arrays pasados como terceros y cuartos argumentos. Se recorren todas las tareas y se comprueba si el comienzo de cada una de ellas se encuentra en el array de comienzos opcionales, y lo mismo con sus duraciones asociadas. Gracias a esto relacionamos la tarea y su duración a su trabajador correspondiente, el que la va a realizar.

La solución óptima es la siguiente:

Tiempo total: 12
Levantar_muros: empieza el dia 0, tarda 4 (termina el dia 4)
Carpinteria_de_tejado: empieza el dia 4, tarda 3 (termina el dia 7)
Tejado : empieza el dia 8, tarda 1 (termina el dia 9)
Instalacion_electrica : empieza el dia 7, tarda 8 (termina el dia 9)
Pintado_fachada : empieza el dia 10, tarda 2 (termina el dia 12)
Ventanas : empieza el dia 9, tarda 1 (termina el dia 10)
Jardin : empieza el dia 11, tarda 1 (termina el dia 12)
Techado : empieza el dia 9, tarda 1 dias (termina el dia 10)
Pintado_interior : empieza el dia 10, tarda 2 dias (termina el dia 12)

Ejercicio 10

Este problema consiste en el clásico "problema de la mochila" en el cual tenemos una serie de objetos con un determinado peso y valor, y tendremos que meterlos en una mochila con un límite de capacidad. Tendremos que seleccionar el conjunto de objetos que más valor tenga y que, a su vez, nos entre en la mochila (la cual tiene 275 kg de capacidad). Los pesos y el nivel de preferencia los tendremos en la tabla del enunciado.

Al igual que en los ejercicios anteriores, veamos primero la declaración de variables. Hemos creado tres arrays, uno para cada columna de la tabla proporcionada (nombres, pesos y preferencia), una variable con el peso de la mochila, y un vector solución de 0s y 1s, donde 0 significa que el objeto no está en la mochila y el 1 si.

La restricción que utilizaremos en este ejercicio es muy sencilla. Simplemente hacemos una sumatoria de todos los objetos de forma que multiplicamos el peso por el vector solución, es decir, si el objeto está en la mochila, lo añade a la suma (multiplica por 1) y si no está no lo añade (multiplica por 0). Tras esto hay que comprobar si nos hemos pasado o no del peso máximo.

Finalmente, haremos lo mismo, pero en la función "`solve maximize`", y esta vez con las preferencias en vez de con el peso. De esta forma, la soluciópn será maximizando esta operación. El resultado óptimo obtenido es el siguiente:

Preferencia: 705

Mochila: Mapa, Compas, Agua, Sandwich, Azucar, Queso, Protector