Práctica 2
Satisfacción de restricciones

Técnicas de los Sistemas Inteligentes - GII
Grupo 2
Curso 19-20

Universidad de Granada
Cumbreras Torrente, Paula
49087324-B

Índice de contenidos

Introducción

El objetivo de esta práctica es aprender a codificar problemas de satisfacción de restricciones empleando MiniZinc, un lenguaje de modelado de restricciones de código libre. Para ellos, se pide resolver una relación de diez problemas, los cuales han sido incluidos en ficheros MZN individuales. A continuación se muestra la documentación de los mismos, junto a las soluciones obtenidas.

Ejercicios

Ejercicio 1

Las variables han sido definidas como var int, pudiendo tomar un valor entre 0 y 9 o entre 1 y 9 en caso de T, F, D y K. Para asegurarnos de que a cada letra se le asigna un valor único, se ha definido un array de var int con todas las letras sobre el que hemos aplicado `all_different` (admitirá arrays con todos sus componentes distintos) de la librería "all_different.mzn". Para asegurar la satisfacción de la suma, basta con incluir:

```
constraint  10000*T + 1000*E + 100*S + 10*T + E + 10000*F + 1000*E + 100*S + 10*T + E
            + 10000*D + 1000*E + 100*I + 10*N + E = 100000*K + 10000*R + 1000*A
            + 100*F + 10*T + E;
```

```
Running Ejercicio1.mzn
T=9 F=3 D=4 E=5 S=2 I=8 N=0 K=1 R=7 A=6
   95295
 + 35295
 + 45805
  ======
  176395
----------
Finished in 60msec
```

Ejercicio 2

Se han definido 10 variables int que formarán el array x, pudiendo tomar cada una de ellas un valor entre 0 y 9. Se ha definido una función `f` de tipo predicate que hace uso de `count` de la librería "count_fn.mzn" la cual dará por válido un valor `n` siempre y cuando ése sea el número de veces que aparece otro valor `v` en un array `x`. De esta forma, podemos otorgarle a cada una de las variables el valor que corresponda:

- `constraint f(d0,x,0);`

- constraint f(d1,x,1);
- etc.

```
Running Ejercicio2.mzn
X= 6210001000
----------
Finished in 62msec
```

Ejercicio 3

Se ha definido una variable int por cada profesor, la cual podrá tomar valores en función de sus restricciones horarias (por ejemplo, p1 puede tomar los valores 11, 12, 13 o 14). Las seis variables creadas han sido recogidas en un array clases, sobre el que se ha aplicado all_different de la librería "all_different.mzn" para asegurar que solo pueda dar clase un profesor por hora.

```
Running Ejercicio3.mzn
Profesor 1: 14:00
Profesor 2: 12:00
Profesor 3: 13:00
Profesor 4: 10:00
Profesor 5: 11:00
Profesor 6: 9:00
----------
Finished in 70msec
```

Ejercicio 4

Para la resolución de este problema se han definido las siguientes variables:

- 4 arrays p de tamaño variable en función del número de asignaturas que imparte cada profesor, donde se almacerán las horas a las que impartirá cada una de ellas.
- 4 arrays g de tamaño 3 donde se almacenan las clases en las que se impartirán las asignaturas correspondientes a cada grupo
- 4 arrays g_prof de tamaño 3 donde se almacenan las horas a las que se impartirán las asignaturas correspondientes a cada grupo

Para asegurar que a un profesor no se le puedan asignar dos clases a la vez, se aplica all_different de la librería "all_different.mzn" sobre cada uno de los arrays p. Ésto mismo se aplicará sobre los arrays g_prof para evitar incompatibilidad horaria, es decir, que para un grupo se asignen dos clases a la vez. Se ha definido una función aulas de tipo predicate que comparará dos grupos (tanto sus aulas, como sus horarios) y aceptará únicamente si ninguno de los componentes del primer grupo coincide tanto en hora como en aula con ninguno de los componentes del segundo grupo. Esta función se aplica sobre todos los pares de grupos asegurando de esta forma que no se asigne más de una clase en un aula a la misma hora.

```
Running Ejercicio4.mzn
Grupo 1:
  IA- aula 2 horario 12:00
  TSI- aula 3 horario 11:00
  FBD- aula 3 horario 9:00
Grupo 2:
  IA- aula 3 horario 10:00
  TSI- aula 2 horario 9:00
  FBD- aula 1 horario 13:00
Grupo 3:
  IA- aula 2 horario 11:00
  TSI- aula 1 horario 12:00
  FBD- aula 2 horario 10:00
Grupo 4:
  IA- aula 1 horario 10:00
  TSI- aula 1 horario 11:00
  FBD- aula 1 horario 9:00
----------
Finished in 74msec
```

Ejercicio 5

Este problema ha sido representado con un array horario bidimensional [5x6]. A cada posición de este array se le asignará un int que corresponde a una asignatura (numeradas de 1 a 9). Se han definido varias funciones de tipo predicate:

- establecer_recreo asegura que en todos los días, la cuarta franja horaria está reservada para el recreo, el cual ha sido representado con el número 0.
- bloque_dos_horas hace uso de count de la librería ““count_fn.mzn” para garantizar que si, para cualquiera de los días, la asignatura a ha sido asignada a alguna hora, tendrá que volver a ser asignada a otra hora consecutiva a esta, pues se tratará de una asignatura impartida en bloque de dos horas. Para ello se realizan una serie de condicionales sencillos que garanticen la consecutividad. Esta función se aplicará sobre las asignaturas 1, 3, 4, 5 y 8.
- bloque_una_hora, haciendo uso de count, asegura que una asignatura a puede haber sido asignada, como mucho, una vez por cada día. Se aplica sobre las asignaturas 2, 6, 7 y 9
- bloque_profesor, dadas dos asignaturas a1 y a2, haciendo uso de count, garantiza que en un mismo día no puedan impartirse las dos. Esto se aplicará para los pares de asignaturas 1-3 (profesor 1), 4-5 (profesor 2) y 6-9 (profesor 3).
- horas_semanales, volviendo a hacer uso de count, comprueba que en toda la semana se imparten exactamente t horas de la asignatura a.

```
Running Ejercicio5.mzn
       L  M  X  J  V
 8:00  3  5  3  9  6
 9:00  3  5  3  4  4
10:00  2  7  7  4  4
```

```
11:00 0   0   0   0   0
12:00 5   8   6   1   1
13:00 5   8   2   1   1
----------
Finished in 116msec
```

Ejercicio 6

Este problema ha sido representado con un array matriz bidimensional [5x5] donde cada fila corresponde a una casa, ordenadas de izquierda a derecha. En cada fila quedan representadas las características (región, color de la casa, profesión, animal y bebida preferida) asociadas a cada casa, permitiendo así que al emplear all_different de la librería "all_different.mzn" sobre las filas no coincidan las características en más de una columna. Se han creado dos var int cebra y agua donde se almacenará la casa a la que pertenecen. Para almacenar los datos del enunciado se han generado cuatro funciones de tipo predicate:

- asignar_casa, permite asignar una característica q a una fila caract si conocemos el número de la casa (por ejemplo, para la la restricción "el andaluz vive en la primera casa de la izquierda").
- asignar_si, permite asignar una característica q2 a la fila caract2 de aquella casa en la que la fila caract1 valga q1 (por ejemplo, para la restricción "el de la casa verde bebe café).
- al_lado establece que la casa cuya fila caract1 valga q1, está situada a alguno de los lados de la casa cuya fila caract2 valga q2 (por ejemplo, para la restricción "la casa del andaluz está al lado de la azul").
- a_la_derecha realiza la misma acción que la función al_lado, con la excepción de que solo permite que la casa se encuentre situada a la derecha; esta función fue definida para satisfacer la restricción "la casa verde está al lado de la blanca y a su derecha".

```
Running Ejercicio6.mzn
casa      1  2  3  4  5
region    5  4  1  3  2
color     5  4  1  2  3
profesion 5  3  2  1  4
animal    4  3  2  5  1
bebida    5  1  3  2  4

La cebra la tiene el 3 en la casa 4
Bebe agua el 5 en la casa 1

[Region: 1-vasco, 2-catalan, 3-gallego, 4-navarro,5-andaluz]
[Casas numeradas de izquierda a derecha]
----------
Finished in 90msec
```

Ejercicio 7

Para este problema se han incluido dos versiones: en ambas versiones de este ejercicio obtenemos el mismo resultado, pero una de ella (Ejercicio7c.mzn) utiliza la función cumulative de la librería "cumulative.mzn", que distribuye automáticamente una serie de tareas entre cierto número de trabajadores, teniendo en cuenta las duraciones y los trabajadores necesarios para cada tarea. Dado que esta función ha sido empleada en el ejercicio 8, no se explicará en esta sección.

La versión que no utiliza dicha función almacena en un array de tamaño 9 el orden en el que se ejecutarán las tareas (sobre él se ha ejecutado all_different) y un array costes del mismo tamaño donde se almacenan las duraciones de cada tarea. Para satisfacer las restricciones relativas al orden de ejecución de ha definido una función realizado_antes de tipo predicate que garantiza que una tarea b se realiza antes que otra tarea a. En el orden de ejecución establecido, se rellena un array empieza de tamaño 9, donde se almacenan los tiempos en los que empieza cada tarea; para ello se ha definido otra función empezar de tipo predicate, que, haciendo uso de una función empezar_anteriores de tipo var int, asigna a cada valor del array el tiempo que han tardado en finalizar las tareas precedentes; para ello se almacenan en otro array finaliza de tamaño 9 la suma del tiempo en el que ha empezado una tarea y su correspondiente valor del array costes. El coste total a minimizar ha sido almacenado en una variable int coste_total y tomará el valor máximo del array finaliza.

La asignación del orden de tareas ya es óptima, por lo que aunque tratemos de minimizar el coste, siempre obtendremos la misma asignación. Se puede observar que la versión que utiliza cumulative es ligeramente más rápida que la implementada por mí.

Running Ejercicio7.mzn

Tarea	Empieza	Termina
A	1	8
B	8	11
C	11	12
D	8	16
E	16	18
F	16	17
G	16	17
H	8	11
I	17	19

Coste = 19

==========

Finished in 88msec

Running Ejercicio7c.mzn

Tarea	Empieza	Termina
A	1	8
B	8	11
C	11	12
D	8	16
E	16	18
F	16	17
G	16	17
H	8	11
I	17	19

Coste = 19

==========

Finished in 73msec

Ejercicio 8

Este problema, al ser una modificación del problema 7, mantiene prácticamente las mismas variables y funciones que la versión del mismo (Ejercicio7c.mzn) que usa cumulative de la librería "cumulative.mzn". Se han definido 4 arrays de tamaño 9:

- costes almacena la duración de cada una de las tareas.
- trabajadores almacena el número de trabajadores necesarios para realizar cada tarea (en el ejercicio 7c, este array consistiría en 9 unos).
- empieza almacena el tiempo que tarda en empezar cada tarea.
- finaliza almacena el tiempo que tarda en terminar cada tarea.

Además se define la variable int max_trabajadores, inicializada a 3 (en el ejercicio 7c fue inicializada a 9) que indica el número de trabajadores que pueden realizar alguna tarea simultáneamente. La variable int coste_total a minimizar será el valor máximo del array finaliza. Al hacer uso de cumulative (constraint cumulative (empieza, costes, trabajadores, max_trabajadores);), que realiza una asignación automática de tareas teniendo en cuenta el número de trabajadores necesarios para cada una de ellas y la duración de las mismas, como ya se explicó en la sección correspondiente al ejercicio 7, solo se han tenido que definir funciones para controlar el orden en el que se realizan: para cada tarea, la suma del tiempo en el que comienza y de su coste asociado (duración) ha de ser menor al tiempo en el que comienzan sus tareas precedentes. El array finaliza tomará como valor la suma del tiempo de inicio de cada tarea y de su coste asociado.

```
Running Ejercicio8.mzn
Tarea   Empieza  Termina
A       1       8
B       16      19
C       19      20
D       8       16
E       21      23
F       20      21
G       20      21
H       8       11
I       21      23
Coste = 23
----------
==========
Finished in 72msec
```

Ejercicio 9

Este problema también es una modificación del problema 7, por lo que mantendrá muchas de las variables empleadas para el ejercicio 7 y 8, a excepción de:

- El array costes pasa a ser un array bidimensional [3x9] donde se almacenan los costes asociados a cada trabajador.
- Se ha definido un array trabaja de tamaño 9 en el que se almacenará qué trabajador realizará cada una de las tareas.

Se ha redefinido la función cumulative de tipo predicate de forma que acepte la nueva representación de los datos, eligiendo para cada tarea i la duración que le supondrá al trabajador asignado a dicha tarea realizarla (costes[trabaja[i],i]):

```
predicate cumulative_multi(array[TAREA] of var TIME: e, array[int,TAREA] of var int: c,
array[TAREA] of var int: w, var int: mr) =
        let { set of int: times = lb_array(e)..max([ ub(e[i]) + ub(c[w[i],i]) | i in TAREA ]) }
        in forall( t in times ) (
          mr >= sum( i in TAREA )
          ( e[i] <= t /\ t < e[i] + c[w[i],i] )
        );
```

Para controlar que a un mismo trabajador no se le asignen más de una tarea en cada momento, se ha definido una función varios_trabajos de tipo predicate, que mediante un bucle doble, asegura que para cada una de las tareas de un trabajador tr, no haya ninguna otra asignada al mismo trabajador que coincida con la primera, ya sea por empezar a la vez o por empezar después que la primera pero antes de que ésta finalice, etc. Esta función se aplicará sobre los 3 trabajadores.

La asignación del orden de tareas inicial no es óptima, y por tanto, sí puede minimizarse la variable coste_total, como se puede observar en el resultado (el último resultado mostrado es el optimizado):

```
Running Ejercicio9.mzn
Tarea   Empieza  Termina   Trabajador
A       1       5         1
B       5       8         1
C       8       9         2
D       5       13        3
E       13      19        3
F       13      15        2
G       13      14        1
H       5       8         2
I       14      16        1
Coste = 19
----------
Tarea   Empieza  Termina   Trabajador
A       1       5         1
B       5       8         1
C       8       9         2
D       5       13        3
E       13      15        2
F       13      14        3
G       13      14        1
H       5       8         2
I       14      16        1
Coste = 16
----------
```

```
----------
Tarea   Empieza  Termina   Trabajador
A       1       5         1
B       5       8         1
C       8       9         2
D       8       11        1
E       11      13        2
F       11      12        3
G       11      12        1
H       5       8         2
I       12      14        1
Coste = 14
----------
==========
Finished in 466msec
```

Ejercicio 10

Para este problema se han definido 3 arrays de int de tamaño 12: pesos almacena los pesos de cada objeto, pref almacena la preferencias de los mismos y mochila almacena un

valor 1 o 0 a modo de booleano que indica si el objeto ha sido o no seleccionado (respectivamente). Se han definido 2 variables pref_total (a maximizar) y peso_total.

Se ha definido una función llenar_mochila de tipo predicate encargada de controlar los valores de pref_total y peso_total haciendo uso de otra función aniadir de tipo var int, la cual realiza una sumatoria de los objetos del vector (pref o pesos) que hayan sido seleccionados en mochila. Dicha sumatoria será la que de valor a las variables pref_total y peso_total. La restricción del peso máximo es controlada con otra función pesar del tipo predicate que asegura que el peso_total sea menor a peso_max (275)

```
Running Ejercicio10.mzn
Objeto  |Mapa Compás Agua Sandwich Azúcar Lata Plátano Manzana Queso Cerveza
P.Solar Cámara
S-1/N-0 |1   0     0   0      0     0   0     0     0     0     0     0
Peso total = 9
Suma preferencias = 150
----------
Objeto  |Mapa Compás Agua Sandwich Azúcar Lata Plátano Manzana Queso Cerveza
P.Solar Cámara
S-1/N-0 |1   1     0   0      0     0   0     0     0     0     0     0
Peso total = 22
Suma preferencias = 185
----------
                                [10 soluciones omitidas]
----------
Objeto  |Mapa Compás Agua Sandwich Azúcar Lata Plátano Manzana Queso Cerveza
P.Solar Cámara
S-1/N-0 |1   0     1   1      1     0   1     0     0     0     1     0
Peso total = 265
Suma preferencias = 700
----------
Objeto  |Mapa Compás Agua Sandwich Azúcar Lata Plátano Manzana Queso Cerveza
P.Solar Cámara
S-1/N-0 |1   1     1   1      1     0   0     0     1     0     1     0
Peso total = 274
Suma preferencias = 705
----------
==========
Finished in 70msec
```

Bibliografía

- https://www.minizinc.org
    - *"A MiniZinc Tutorial"*
    - *"MiniSearch - Documentation"*
    - *"2.1. Basic Modelling in MiniZin - The MiniZinc Handbook 2.4.3"*
    - *"2.2. More Complex Models - - The MiniZinc Handbook 2.4.3"*
    - *"2.3. Predicates and Functions - The MiniZinc Handbook 2.3.1"*
    - *"4.1. Specification of MiniZinc - The MiniZinc Handbook 2.4.2"*
    - *"4.2.6. Global Constraints - The MiniZinc Handbook 2.3.0"*
- http://www.hakank.org - *"My MiniZinc Page"*
- AIDA Research Group (YouTube) - *"Tutorial Introduction to MiniZinc"*