Técnicas de los Sistemas Inteligentes

Práctica 2:

Satisfacción de restricciones

Curso 2019-20

Javier Gálvez Obispo

javiergalvez@correo.ugr.es

Grupo 2, Viernes 17:30 – 19:30

Ejercicio 1

Representamos los valores que toman las letras en un array de 10 elementos donde la posición 1 representa el valor de la letra 1. El orden que decidamos para las letras no tienen mayor importancia, en este caso es el siguiente:

```
letras = ["F", "E", "S", "T", "D", "I", "N", "K", "R", "A"];
```

Como tenemos 10 letras, cada letra representará un dígito distinto en el intervalo [0, 9] por lo que no se pueden repetir valores en el array, utilizamos *all_different(valores)* para ello.

Lo dificultad de este ejercicio reside en los acarreos que se pueden producir en cada suma. Utilizamos un array *acarreos* que guarda el acarreo que produce cada suma para así poder tenerlos en cuenta en la siguiente suma. Calculamos el acarreo como la diferencia de la suma anterior entre 10 y el resto de dividir la suma anterior entre 10 también divido por 10, por ejemplo:

La suma anterior suma 15, caso de que *E* sea 5 tendríamos

$$\text{acarreo} = 15 / 10 - (15 \bmod 10) / 10 = 1.5 - 0.5 = 1$$

Existen dos soluciones para este problema. La letra *E* puede tomar los valores 0 y 5 produciendo dos resultados totalmente distintos:

30830	95295
+ 60830	+ 35295
+ 50970	+ 45805
142630	176395

Ejercicio 2

Representamos el número de 10 dígitos como un array de 10 posiciones. Sólo necesitamos una restricción para resolver este problema:

```
constraint forall(i in 1..10)(count(numero, i-1) == numero[i]);
```

Obligamos a que el dígito *i* aparezca dentro del array las veces que se indica en la posición *i* del mismo.

Sólo existe una solución para este problema, 6210001000. Si añadimos una restricción para que cualquiera de las posiciones, sólo una, no pueda tomar el valor de la solución dada, el problema se vuelve insatisfactible.

Ejercicio 3

Representamos la horas disponibles de los profesores como una matriz 6x2 en la que la fila 3 representa al profesor 3 y la columnas guardan la hora de inicio y la hora final de su franja de tiempo disponible.

Utilizamos un array de 6 elementos para representar el orden los profesores. Forzando que éste tome valores distintos en cada posición conseguimos que cada profesor imparta una clase de 1 hora. Utilizamos un array unidimensional ya que sólo se puede dar una clase por hora.

Cuando asignamos una hora a un profesor, ésta debe estar en la franja horaria que tiene disponible. Comprobamos que la hora a la que acaba la clase se encuentre en el intervalo (inicio-fin] de la franja horaria del profesor.

El horario resultante es el siguiente:

9:00 – 10:00	Profesor 6
10:00 – 11:00	Profesor 4
11:00 – 12:00	Profesor 5
12:00 – 13:00	Profesor 2
13:00 – 14:00	Profesor 3
14:00 – 15:00	Profesor 1

Ejercicio 4

Representamos el horario como una matriz booleana 12x4, 12 filas porque tenemos 3 asignaturas y 4 grupos dando un total de 12 clases distintas que se pueden impartir. Entonces, si horario[i, j] == 1 significa que la clase *i*, por ejemplo TSI-G3, se da a la hora *j*. Las clases se ordenan de la siguiente manera:

```
clases = ["IA-G1", "IA-G2", "IA-G3", "IA-G4",
          "TSI-G1", "TSI-G2", "TSI-G3", "TSI-G4",
          "FBD-G1", "FBD-G2", "FBD-G3", "FBD-G4"];
```

Ya que sólo disponemos de 4 aulas, haremos que la suma de cada columna de la matriz sea menor o igual que 4, como la matriz solo toma valores 0 ó 1 la suma de una columna indica el número de clases que se dan a esa hora.

Para representar qué clases son impartidas al mismo grupo utilizamos otra matriz booleana de tamaño 4x12, indicando con 1 las clases que se imparten al grupo representado por la fila, es decir, si mismo_grupo[3, 7] == 1 quiere decir que la clase, TSI-G3, se imparte al grupo 3.

Para comprobar que dos clases que se imparten al mismo grupo no se den a la misma hora forzamos que la suma para una hora dadas 2 clases que comparten grupo sea menor o igual que 1, es decir horario[clase1, hora] + horario[clase2, hora] <= 1.

Utilizamos la misma representación para asignar profesores a asignaturas que para los grupos. Evitamos que un profesor imparta dos clases distintas al mismo tiempo siguiendo la misma idea que antes.

A la hora de asignar el horario hay que tener en cuenta que las clases que imparte el profesor 2 no pueden darse en la segunda hora, si profesores[2, clase] == 1 entonces horario[clase, 2] == 0. Lo mismo ocurre con el profesor 4 en la primera hora.

El horario que obtenemos es el siguiente:

9:00 – 10:00	IA-G2 (P1)	FBD-G1 (P2)	FBD-G4 (P3)	
10:00 – 11:00	IA-G4 (P4)	TSI-G2 (P1)	FBD-G3 (P3)	
11:00 – 12:00	IA-G3 (P4)	TSI-G1 (P1)	TSI-G4 (P3)	FBD-G2 (P2)
12:00 – 13:00	IA-G1 (P1)	TSI-G3 (P3)		

La asignación de aulas es arbitraria.

Ejercicio 5

Para este ejercicio representamos el horario como una matriz 5x6 en la que guardamos en cada posición *i, j* la asignatura que se imparte en el día *i* a la hora *j*. Además, guardamos en 2 arrays distintos las horas que se tienen que dar por asignatura y la duración de los bloques.

Para las asignaturas que se imparten en bloques de 2 horas comprobamos que si horario[i, j] = *a*, siendo *a* una asignatura de éstas, entonces, tiene que ocurrir que horario[i, j+1] == *a*, o bien, horario[i, j-1] == *a*, pero no ambas a la vez.

Para asegurarnos que no se imparte más de un bloque de una asignatura al día contamos la veces que aparece una asignatura en una columna y hacemos que sea igual a la duración de un bloque de esa asignatura.

Puesto que los profesores que sólo pueden dar clase una vez por día, todos menos el profesor 4, imparten dos asignaturas, basta con comprobar que si en una columna, un día, ya ha dado la clase de una de las asignaturas entonces no pueda impartir una clase de la otra asignatura. Por ejemplo, para el profesor 1 si horario[1, 3] == 1 entonces horario[1, *i*] != 3 para cualquier *i* en [1, 6].

El recreo lo representamos como un 0 en el horario. Forzamos que horario[i, 4] == 0 para cualquier *i* en [1, 5] e impedimos que el resto de horas de clase sean 0. Además, tenemos que comprobar que se dan las horas exactas de cada asignatura contando las veces que aparece cada una en la matriz.

Uno de los resultados posibles es el siguiente:

A7	A4	A3	A6	A6
A3	A4	A3	A1	A1
A3	A2	A2	A1	A1
RECREO				
A8	A9	A5	A5	A4
A8	A7	A5	A5	A4

Ejercicio 6

Para este ejercicio definimos una variable por persona y le asignamos un número entre 1 y 5 sin repetir. Por ejemplo:

vasco = 1, catalán = 2, gallego = 3, navarro = 4, andaluz = 5

Vamos a representar cada característica en un array distinto. Tendremos un array para las casas, otro para las profesiones, otro para los animales y otro para las bebidas. Definimos variables con valores entre 1 y 5 para cada característica. La forma en la que asignamos los índices es arbitraria pero no podemos repetir ningún valor. En estos arrays la posición *i* representa la característica de la persona *i*, por ejemplo, animales[2] == perro significa que el catalán (según la asignación de arriba) es dueño del perro.

Además, también es necesario representar el orden de las casas. Para ello usamos un array en el que la posición 1 representa la casa más a la izquierda y la posición 5 la casa más a la derecha.

Así podemos representar de forma sencilla las restricciones que nos piden. Por ejemplo, la restricción *a* sería casas[vasco] == roja. Son muchas las restricciones que hay en este problema y ya que son muy fáciles de implementar con ésta representación de la información, es mejor consultar el código proporcionado para ver como se hace cada una.

Respondiendo a la pregunta del problema, la cebra está en la casa del gallego y es el vasco quien bebe agua.

Ejercicio 7

Para este problema asignamos a cada tarea un índice, la tarea A será la tarea 1, la B la 2 y así sucesivamente. Guardamos en un array las duraciones de cada tarea, por tanto, la posición *i* del array indicará la duración de la tarea *i.* Para las tareas predecesoras utilizamos una matriz 9x2 (no hay ninguna tarea que necesite que se completen más de 2 tareas), la fila 5 guarda las tareas que se tienen que realizar para comenzar la tarea 5. No todas las tareas tienen 2 predecesoras y minizinc nos obliga a rellenar todos los valores de la matriz, para ello, creamos una nueva tarea AUX con índice 0 y duración 0 que no tendrá ningún efecto en el problema.

Necesitaremos también un array para guardar la hora de inicio de cada tarea que sólo podrá tomar valores mayores o iguales que 0.

Para resolver el problema tan sólo necesitamos una restricción, una tarea no puede empezar hasta que todas las predecesoras hayan terminado. Ésto lo conseguimos haciendo que la hora de inicio de una tarea tenga que ser mayor o igual a la hora de finalizado de todas sus predecesoras:

inicio[i] >= max(inicio[p1] + duración[p1], inicio[p2] + duración[p2])

donde p1 y p2 son las tareas predecesoras de la tarea *i*.

Ya que queremos construir la casa en el menor tiempo posible minimizamos:

max(inicio[i] + duracion[i])

es decir, el tiempo de acabado de la última tarea.

El menor tiempo posible para construir la casa son 18 días empezando las tareas de la siguiente forma:

Día 0	Día 7	Día 10	Día 15	Día 16
A	B	C	E	I
	D		F	
	H		G	

Ejercicio 8

Nos encontramos ante el mismo problema que en el ejercicio 7 pero ahora sólo tenemos 3 trabajadores. Guardamos en un array los trabajadores que se necesitan para cada tarea siendo el valor almacenado en la posición *i* el número de trabajadores que se necesitan para comenzar con la tarea *i*.

La representación de la información y la restricción que teníamos en el problema anterior no varían para éste. Añadimos la restricción de los trabajadores limitados.

Una tarea no puede comenzar si no hay suficiente trabajadores libres. Para llevar la cuenta de los trabajadores ocupados, cuando vayamos a iniciar una tarea vemos qué otras tareas se están realizando en ese momento. Las tareas que se están llevando a cabo son las tareas que terminan después de la hora de inicio de la tarea que queremos empezar y que empezaron antes que ésta, es decir, para dos tareas *i, j,* con *i* distinto de *j*, *j* está en progreso cuando se va a empezar con *i* si

inicio[j] <= inicio[i] /\ inicio[j] + duración[j] > inicio[i]

entonces, no podremos comenzar la tarea *i* si la diferencia del número de trabajadores que tenemos, 3, y el número de trabajadores ocupados en las tareas que ya se están realizando es menor que el número de trabajadores que necesitamos para la tarea *i*.

El menor tiempo posible para resolver éste problema son 22 días empezando las tareas con el siguiente orden:

Día 0	Día 7	Día 15	Día 18	Día 19	Día 20
A	D	B	C	F	E
	H			G	I

Ejercicio 9

Ahora las tareas requieren un solo trabajador de los tres que hay. Además, cada uno tarda un tiempo distinto en realizar cada tarea.

Modificamos el array de duraciones que teníamos para poder representar ésta información. Lo transformamos en una matriz 3x9 (3x10 si tenemos en cuenta la tarea AUX que necesitamos) en la que cada fila representa un trabajador.

Es necesario saber qué trabajador está asignado a cada tarea en todo momento porque no podemos asignar a la misma persona a 2 tareas distintas. Para ello, utilizamos un array que almacena en la posición *i* el trabajador que realiza la tarea *i*. Entonces, cuando vayamos a iniciar una tarea debemos comprobar que la persona que le asignamos no esté ocupada con otra tarea. Dadas 2 tareas *i, j*, con *i* distinto de *j*, tal que inicio[j] <= inicio[i] /\ inicio[j] + duración[j] > inicio[i], es decir, la tarea *j* está en progreso cuando vamos a iniciar la tarea *i* entonces, trabajando[i] != trabajando[j], la persona que va a trabajar en la tarea *i* debe ser distinta a la que está realizando la tarea *j*.

El menor tiempo posible para resolver este problema son 12 días empezando las tareas y asignado los trabajadores de la siguiente forma:

Trabajadores	Día 0	Día 4	Día 7	Día 9	Día 10
Tr1	A	B	D	G	I
Tr2		H	C	E	
Tr3				F	

Ejercicio 10

Representamos los pesos y la preferencia en dos arrays distintos donde pesos[i] y preferencia [i] representan el peso y el grado de preferencia del turista para el objeto *i* respectivamente.

Utilizamos un array con valores 0 y 1 para representar qué objetos se meten en la mochila y cuáles no.

Para no pasarnos del límite de peso hacemos que la suma del producto de los pesos por su selección no supere el peso máximo, es decir, sum(pesos[i]*seleccion[i]) <= 275.

Para conseguir la mayor preferencia posible maximizamos la suma de las preferencias de los objetos seleccionados, maximizamos sum(preferencia[i]*seleccion[i]).

La mejor solución a éste problema sería meter en la mochila mapa, compás, agua, sandwich, azúcar, queso y protección solar consiguiendo un peso total de 274kg y una preferencia de 705.