

## Tema 2: Introducción a la Programación con Restricciones

[0 Antes de empezar](#)

[1 Primer programa](#)

[2 Segundo programa](#)

[Apéndice A: Instalación y uso de un minizinc](#)

[Apéndice B: Otros sistemas para la programación con restricciones](#)

[Apéndice C: Para saber más](#)

---

[0 Antes de empezar](#)

[0.1 ¿Qué es la programación con restricciones?](#)

[0.2 Objetivos del curso](#)

---

### 0.1 ¿Qué es la programación con restricciones?

#### Lugar en el territorio de la programación

La programación con restricciones es un estilo de programación que agrupa a muchos lenguajes. Se puede encuadrar dentro del paradigma de la **programación declarativa**. En programación declarativa, el programador indica qué quiere conseguir sin detallar los pasos que hay que seguir para lograrlo. Es el sistema el que dispone de forma interna de los métodos que se utilizarán para lograr el objetivo propuesto por el programador. En particular la programación con restricciones ha venido asociándose a la **programación lógica**; ambos tipos de programación se combinan de forma natural en la llamada **programación lógica con restricciones** (Constraint Logic Programming o paradigma CLP). Sin embargo, la programación con restricciones (**PR** a partir de ahora) tiene características propias.

Ni el más entusiasta PR-adicto diría que la PR es un modelo de programación de propósito general. Se trata más bien de un tipo de programación que se utiliza cuando encuentra problemas muy característicos:

- 🕒 Se trata de representar **problemas combinatoriamente complejos**, donde la solución depende de relaciones entre varios objetos. Un ejemplo típico son los Sudokus. Otro la elaboración de horarios en una facultad. En estos casos hay un número finito de posibles soluciones, y podríamos programar código en cualquier lenguaje para ir probando una a una. Sin embargo, en la mayoría de problemas nos encontraríamos con aplicaciones muy lentas y nada escalables. Gracias a la PR nos ahorramos el trabajo de programar la enumeración de posibles soluciones. Además los resolutores que utiliza el sistema aplicarán técnicas de optimización que reducirán considerablemente, a veces espectacularmente, el tiempo requerido para encontrar una solución.
- 🕒 En algunos problemas el número de posibles soluciones puede ser incluso potencialmente infinito. Esto sucede por ejemplo en algunos problemas de optimización en los que aparecen **ecuaciones sobre números reales**. En estos casos resulta imprescindible disponer de métodos que nos permitan encontrar las soluciones (en ocasiones queremos la *mejor* solución bajo un cierto criterio), y la PR nos ofrece justo esto.

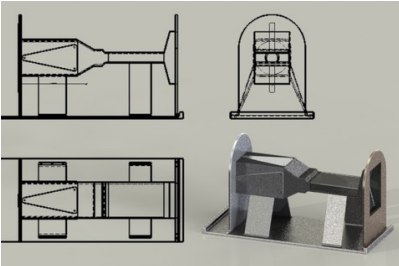
💡 A los programas de PR se les llama *modelos*

Podemos decir que la programación con restricciones es un reino pequeño dentro del territorio de la programación, pero un reino lleno de sorpresas.

#### Los dos niveles

Al estudiar la programación con restricciones podemos distinguir dos niveles:

- 🕒 La modelización del problema. En principio el programador solo tiene que modelar el problema en un lenguaje de restricciones adecuado. El sistema de restricciones que estemos usando toma el modelo y se encarga de buscar las soluciones. Para ello utilizará diversos resolutores: para enteros, para reales, para booleanos. Pero en principio (van dos en principio, hay que empezar a sospechar) esto es transparente para el programador; él solo tiene que modelar, representar el problema, y sentarse a obtener el resultado. ¿Fácil? No tanto. En realidad modelar es todo un arte, y un buen programador con restricciones tiene que conocer:
  - 🕒 El lenguaje de restricciones utilizado, sus limitaciones y sus posibilidades. Algunas técnicas como la reificación ayudarán a modelar de forma elegante y eficiente los problemas.
  - 🕒 Aunque idealmente al modelar no hace falta saber cómo funciona el sistema (el segundo nivel), en la práctica en PR es importante tener al menos una idea general de las posibilidades de los distintos resolutores. Tanto es así que el sistema permite al usuario elegir el resolutor que desee, e incluso darle indicaciones concretas acerca del método de resolución que mejor se adapte al problema.



💡 El desarrollo de resolutores. En este campo se trata de definir resolutores para distintos dominios (booleanos, cadenas, etc.). Los resolutores tienen como objetivo ideal ser correctos (las soluciones que den deben ser soluciones del problema modelado, es decir no vale engañar), eficientes y a ser posible completos (si es posible fuertemente completos: capaces de enumerar todas las soluciones, o si no débilmente completos, capaces de asegurar que si hay solución se encontrará al menos una). No todos los resolutores cumplen estas 3 propiedades. La corrección parece una propiedad necesaria, pero la completitud no siempre es posible. Incluso en ciertos casos se puede sacrificar la completitud por la eficiencia: el sistema encuentra casi siempre solución cuando la hay de forma rápida, pero en ciertos casos particulares falla y no es capaz de encontrar solución.

## 0.2 Objetivos del curso

En este curso nos centraremos en la parte de la modelización. Pretendemos aprender a modelar problemas en diversos dominios, utilizando principalmente el lenguaje de modelado del sistema [MiniZinc](#). Como indicamos en el punto anterior para modelar bien tenemos que conocer algo de lo que hay "por debajo" los resolutores, sus principios, sus limitaciones. Esto nos permitirá modelar mejor, de forma más eficiente, pero también llegado el caso ser capaces de definir nuestros propios resolutores.

El objetivo final es que al programar seamos capaces de reconocer problemas PR cuando aparezcan, podamos modelarlos adecuadamente, e integrar un resolutor con nuestra aplicación (los resolutores se pueden combinar con Java, C++, etc.)



## 1 Primer Programa

Vamos a ver un programa ejemplo.

Enunciado: una imprenta quiere conocer el mínimo número de colores que precisa para colorear el mapa autonómico de España:



La idea es conseguir que dos autonomías con límite común estén siempre pintadas de color diferente, de otra forma sería confuso al parecer que se trata de una sola. En 1976 Kenneth Appel y Wolfgang Haken demostraron que todo mapa plano se podía pintar con solo 4 colores. Sin embargo en ocasiones basta con menos ¿se podrá pintar con 3 colores el mapa autonómico de España?



Para pensar:

¿Cómo programaríamos este problema en C++ o en Java?

En PR, en particular utilizando [MiniZinc](#) podemos emplear un programa similar al siguiente:

```
##### España coloreada

% número máximo de colores
int: nc = 3;

% color de cada comunidad
var 1..nc: ga;  var 1..nc: can; var 1..nc: pv;  var 1..nc: nav; var 1..nc: ara;   var 1..nc: cat;
var 1..nc: as; var 1..nc: cyl; var 1..nc: rio; var 1..nc: mad; var 1..nc: clm; var 1..nc: val;
var 1..nc: mur; var 1..nc: and; var 1..nc: ext;

% restricciones
constraint ga!=as; constraint ga!=cyl; constraint as!=cyl; constraint as!=can;
constraint can!=pv; constraint can!=cyl; constraint pv != nav; constraint pv!= rio;
constraint nav != rio; constraint nav!=ara; constraint ara != cat; constraint ara != val;
constraint rio!= cyl; constraint pv != cyl; constraint cat != val;
constraint cyl != mad;
constraint cyl != ext; constraint cyl!= clm; constraint ext != clm; constraint ara != clm;
constraint mad != clm; constraint clm != val;  constraint ext != and; constraint clm != mur;
constraint and != mur; constraint mur != val; constraint rio!=ara;
```

```
% resolver
solve satisfy;

% mostrar la solución
output ["galicia=", show(ga), "\t cantabria=", show(can), "\t país vasco=", show(pv), "\n",
"aragón=", show(ara), "\t cataluña=", show(cat), "\t asturias=", show(as), "\t navarra=", show(nav),  "\n",
"castilla la mancha=", show(clm), "\t castilla y león=", show(cyl), "\t rioja=", show(rio), "\n",
"madrid=", show(mad), "\t valencia=", show(val), "\t murcia=", show(mur), "\n",
"andalucía=", show(and), "\t extremadura=", show(ext), "\n"];
```

Vamos a explicar un poco el programa:

### Declaración de parámetros



```
int: nc = 3;
```

La variable nc es un parámetro. En este caso indicamos que es un entero y que toma el valor 3.

### Declaración de variables de decisión y sus dominios.

```
var 1..nc: ga;  var 1..nc:can; var 1..nc: pv;  var 1..nc:nav; var 1..nc:ara;   var 1..nc:cat;
var 1..nc: as; var 1..nc: cyl; var 1..nc: rio; var 1..nc:mad; var 1..nc: clm; var 1..nc:val;
var 1..nc: mur; var 1..nc:and; var 1..nc: ext;
```

Estas variables son llamadas variables de decisión. Hay una para cada autonomía (ga para Galicia, can para Cantabria) exceptuando las islas (que no influyen en el problema). Cada variable tiene un dominio inicial de 1..nc. La idea es que el valor "1" representa el primer color a usar, "2" el segundo color, etc. Por tanto cada variable indica de qué color se pintará la autonomía. El objetivo es encontrar valores para estas variables tales que:

-  Sean valores del dominio 1..nc.
-  Se satisfagan (es decir se cumplan) las restricciones sobre las variables.

### Declaración de restricciones.

```
% restricciones
constraint ga!=as; constraint ga!=cyl; constraint as!=cyl; constraint as!=can;
constraint can!=pv; constraint can!=cyl; constraint pv != nav; constraint pv!= rio;
constraint nav != rio; constraint nav!=ara; constraint ara != cat; constraint ara != val;
constraint  rio!= cyl; constraint pv != cyl; constraint cat != val;
constraint cyl != mad;
constraint cyl != ext; constraint cyl != clm; constraint ext != clm; constraint ara != clm;
constraint mad != clm; constraint clm != val;  constraint ext != and; constraint clm != mur;
constraint and != mur; constraint mur != val; constraint rio!=ara;
```

Ya llegamos a las restricciones. Aquí decimos lo que deben cumplir las variables en toda solución de nuestro problema. En nuestro caso debemos asegurarnos de que dos regiones limítrofes tengan distintos colores. Esto lo hacemos mediante restricciones de desigualdad. Por ejemplo `constraint as!=cyl`; indica que asturial (as) y Castilla La Mancha (cyl) deben tener colores distintos.

### Llamada al resolutor.

```
solve satisfy;
```

indica al sistema que ya puede buscar una solución que satisfaga todos los constraints. La sentencia `solve` también puede ser utilizada con los parámetros `maximize` y `minimize` para buscar los valores máximos o mínimos de una función objetivo. Esta simple llamada reemplaza todo el códigoque tendríamos que escribir para buscar una solución.

### Mostrar solución.

```
output ["galicia=", show(ga), "\t cantabria=", show(can), "\t país vasco=", show(pv), "\n",
"aragón=", show(ara), "\t cataluña=", show(cat), "\t asturias=", show(as), "\t navarra=", show(nav),  "\n",
"castilla la mancha=", show(clm), "\t castilla y león=", show(cyl), "\t rioja=", show(rio), "\n",
"madrid=", show(mad), "\t valencia=", show(val), "\t murcia=", show(mur), "\n",
"andalucía=", show(and), "\t extremadura=", show(ext), "\n"];
```

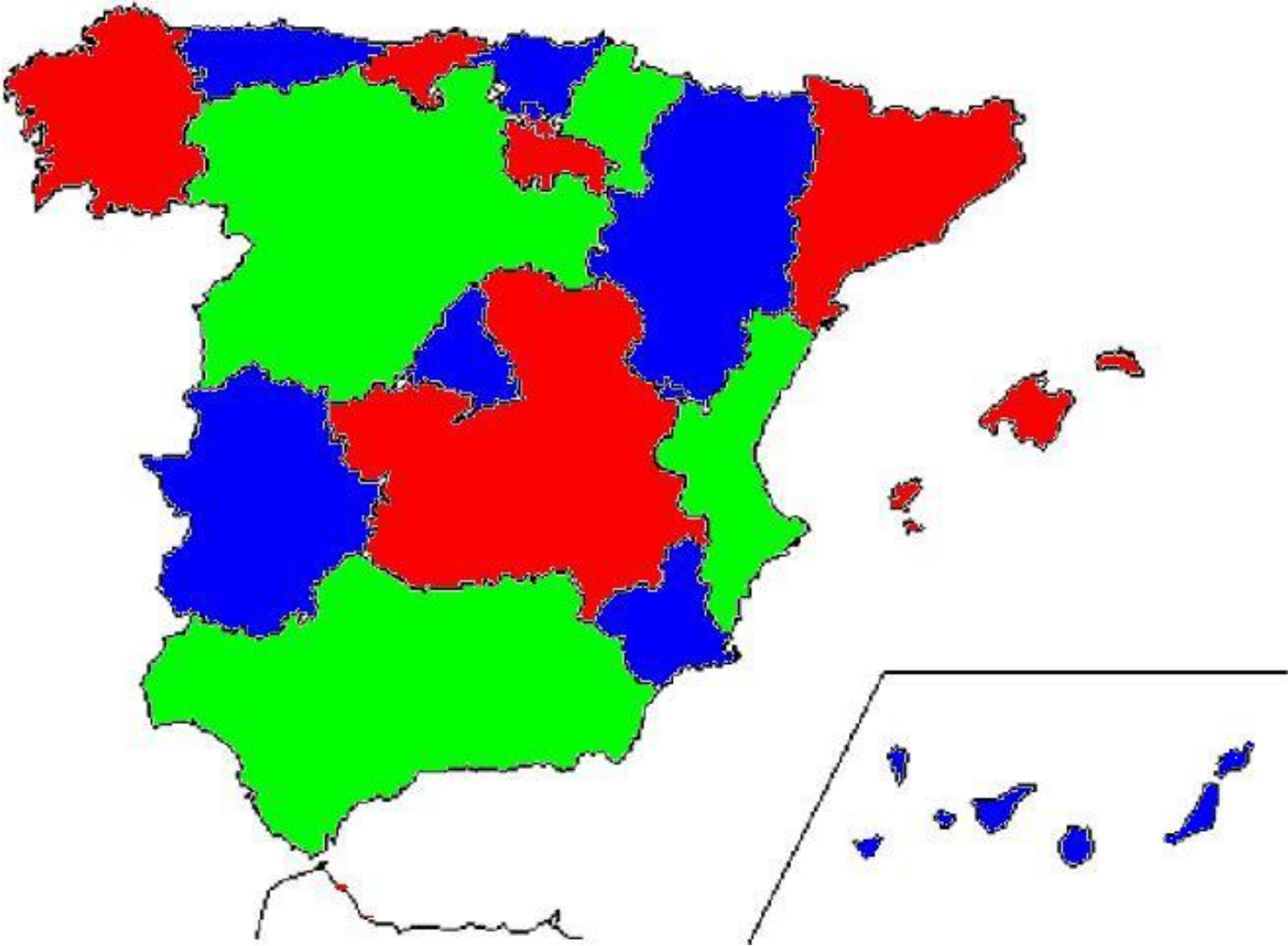
Finalmente la instrucción `output` se utiliza para mostrar por pantalla una serie de valores, es este caso el contenido de las variables a las que `solve` ha dado valores. Acepta una lista de strings como parametro. Los valores enteros de las variables son convertidos en strings mediatne la función `show`

Ahora podemos ejecutar el programa desde la consola utilizando `minizinc`:

```
[~/docencia/1213/pr/ejemplos] $ minimizinc coloreas.mzn
galicia=3          cantabria=3          país vasco=2
aragón=2           cataluña=3           asturias=2         navarra=1
castilla la mancha=3      castilla y león=1      rioja=3
madrid=2           valencia=1           murcia=2
andalucía=1        extremadura=2
```

Es decir el sistema ha encontrado una solución, confirmando que el mapa autonómico de España solo necesita 3 colores. Si hacemos corresponder el número 1 con el verde, el 2 con el azul y el 3 con el rojo podemos pintar el mapa como sigue:





El esquema que sigue el programa: declaración de variables / llamada al resolutor / mostrar resultado es muy habitual en la programación con restricciones.



Ahora se puede entender mejor porqué la programación con restricciones pertenece a la *programación declarativa*: se declaran variables, se declaran restricciones, pero no se dice cómo encontrar los valores; eso es cosa del resolutor.

Antes de continuar se recomienda resolver las siguientes preguntas



Supongamos que descubrimos que Navarra (nav) tiene frontera con Castilla y León ¿Cómo incorporamos esta información? ¿qué nos devuelve ahora MiniZinc? (probarlo)

Para aprender más sobre MiniZinc se recomienda el tutorial: <http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-1.6/minizinc-tute.pdf>

## 2 Segundo programa

¿Fácil verdad? Bueno, es fácil pero no tanto. Vamos a ver un ejemplo un poco más complicado, y a tropezarnos con las primeras dificultades.

Enunciado: una fundición fabrica dos tipos de materiales, A y B. Cada tonelada de material A requiere 3 días de horno, además consume 14 toneladas de mineral de hierro y 2 toneladas de carbón. En cambio, cada tonelada de material B requiere 4 días de horno, precisa 2.1 toneladas de hierro y 11 toneladas de carbón. El precio de venta del material A es de 18.5 euros la tonelada, y el precio de B 20.7 euros por tonelada. La fundición dispone de un único horno y tiene almacenadas 40 toneladas de hierro y 13 de carbón. En los próximos 15 días no se recibirán más suministros de estos materiales. Calcular cuántas toneladas de A y cuantas de B hay que producir en esos días para maximizar el beneficio.



No parece difícil. Solo que en este caso hay que maximizar la función de ganancias (*solve maximize*) en lugar de resolver sin más. Además las cantidades de material A y B de la solución pueden no ser enteras; puede que lo más rentable sea producir 1.5 toneladas de cada uno, por ejemplo. Hacemos una primera versión del modelo:



Fundición. Versión 1:

```
##### fundición

int: dias = 15;
int: cant_h = 40;
int: cant_c=13;

% cantidades a fabricar de cada producto
var float: a;
var float: b;

% límite de tiempo de uso de horno
```

```
constraint a*3+b*4<=dias;
% no se puede superar la cant. de hierro almacenado
constraint 14*a+2.1*b<=cant_h;
% ni la de carbón
constraint 2*a+11*b<=cant_c;
% cantidades no negativas
constraint a>=0;
constraint b>=0;


% maximizar el beneficio
solve maximize 18.5*a+20.7*b;

%mostrar el resultado
output ["A=", show(a), "\t B=", show(b), "\tganancia:", show(18.5*a+20.7*b),
        "\nSobra carbón: ", show(cant_c-(2*a+11*b)), "\tSobra hierro: ", show(cant_h-(14*a+2.1*b)),
        "\n Sobran días de horno:", show(dias-(a*3+b*4)),"\n"];
```

Utilizamos minizinc para compilar y ejecutar el modelo:

```
[~/docencia/1213/pr] $ minizinc fundicion.mzn
....
fundicion.mzn:27:
  type-inst error: operator `*' argument list has invalid type-inst: expected
  `(int, int)' or `(var int, var int)' or `(float, float)' or
  `(var float, var float)', actual `(int, var float)'
....
```

Obtenemos un error de compilación ¿por qué? MiniZinc nos está diciendo que estamos mezclando enteros con reales. Esto se suele hacer en muchos lenguajes, donde la conversión de enteros a reales es automática, pero no así en un resolutor. La razón es que en el caso de valores enteros se utilizará un resolutor con una técnicas determinadas, y en el caso de números reales otro resolutor. Y no podemos mezclar.



Error común: escribir modelos que combinen tipos de datos distintos.

Para solucionarlo tenemos que convertir todos los datos a tipo *float* incluso las constantes enteras. Así 11 se escribirá 11.0.

 Fundición. Versión 2:

```
%%%%%%%% fundición

float: dias = 15.0;
float: cant_h = 40.0;
float: cant_c=13.0;

% cantidades a fabricar de cada producto
var float: a;
var float: b;

% límite de tiempo de uso de horno
constraint a*3.0+b*4.0<=dias;
% no se puede superar la cant. de hierro almacenado
constraint 14.0*a+2.1*b<=cant_h;
% ni la de carbón
constraint 2.0*a+11.0*b<=cant_c;
% cantidades no negativas
constraint a>=0.0;
constraint b>=0.0;


% maximizar el beneficio
solve maximize 18.5*a+20.7*b;

%mostrar el resultado
output ["A=", show(a), "\t B=", show(b), "\tganancia:", show(18.5*a+20.7*b),
        "\nSobra carbón: ", show(cant_c-(2.0*a+11.0*b)), "\tSobra hierro: ", show(cant_h-(14.0*a+2.1*b)),
        "\n Sobran días de horno:", show(dias-(a*3.0+b*4.0)),"\n"];
```

A ver si ahora....

```
[~/docencia/1213/pr] $ minizinc fundicion.mzn
flatzinc: error: variables of type `var float' are not supported by the FD solver backend.
```

¿qué ocurre? pues que el comando *minizinc* solo es válido si se está utilizando el resolutor para enteros (el llamado *resolutor de dominios finitos* o *FD solver* en inglés).



Error común: utilizar un resolutor que no se corresponde con el *dominio de restricciones*


Para restricciones sobre reales tenemos que utilizar otro resolutor de los que vienen con MiniZinc, en este caso *mzn-g12mip*:

```
[~/docencia/1213/pr] $ mzn-g12mip fundicion.mzn
A=2.7550066755674227      B=0.6809078771695596      ganancia:65.06241655540721
Sobra carbón: 0.0        Sobra hierro: 7.105427357601002e-15
Sobran días de horno:4.011348464619493
```

¡¡Lo hemos conseguido!! Supongamos sin embargo que hay un requerimiento adicional:

Requerimiento: Solo se pueden vender toneladas enteras de mineral. El valor 2.75 toneladas de A significa que tiramos 0.75 toneladas.

El problema de convertir a entero ya lo hemos visto, nos daría un error por mezclar reales (por ejemplo el valor 2.1) con enteros. Lo primero que se nos ocurre es:

 Idea:  
¿bastaría con truncar la solución obtenida por el modelo definido sobre reales?

---

Así obtendríamos que se deben fabricar 2 toneladas de A y 0 de B. La ganancia total será de  $18.5 \times 2 = 37$ .

 Error común: pensar que la solución en los números enteros es la que se obtiene al truncar la solución en los reales.



La solución: comprobar que todas las restricciones y el objetivo de maximización son equivalentes si se multiplican por 10 los dos lados de cada desigualdad. Como todos los números reales que aparecen tiene un solo valor decimal, al multiplicar por 10 tenemos un problema equivalente en enteros.

 Fundición. Versión 3:

```
##### fundición

int: dias = 150;
int: cant_h = 400;
int: cant_c=130;

% cantidades a fabricar de cada producto
var int: a;
var int: b;

% límite de tiempo de uso de horno
constraint a*30+b*40<=dias;
% no se puede superar la cant. de hierro almacenado
constraint 140*a+21*b<=cant_h;
% ni la de carbón
constraint 20*a+110*b<=cant_c;
% cantidades no negativas
constraint a>=0;
constraint b>=0;

% maximizar el beneficio
solve maximize 185*a+207*b;

%mostrar el resultado
output ["A=", show(a), "\t B=", show(b), "\tganancia: (/10)", show(185*a+207*b),
        "\nSobra carbón: (/10)", show(cant_c-(20*a+110*b)), "\tSobra hierro: (/10) ", show((cant_h-(140*a+20*b))),
        "\n Sobran días de horno: (/10)", show(dias-(a*30+b*40)),"\n"];
```

A mostrar el resultado tenemos que a y b serán los correctos, pero que el resto de los valores habría que dividirlos por 10 para que tengan sentido:


```
[~/docencia/1213/pr] $ mzn-gl2mip fundicionent.mzn
A=1      B=1      ganancia: (/10)392
Sobra carbón: (/10)0      Sobra hierro: (/10) 240
Sobran días de horno: (/10)80
```

Ahora solo se admite una tonelada de cada material, y la ganancia baja de 65.06 a 39.2. También se nos indica que nos sobrarán 8 días de horno. Este tipo de "trucos" son inevitables en muchos casos y constituyen parte del "arte de programar con restricciones".


## Apéndice A: Instalación y uso de un minizinc

Los siguientes pasos indican cómo instalar MiniZinc en Linux (en particular yo lo he instalado en Ubuntu).


 Bajar el tar.gz de la última versión disponible en: <http://www.g12.csse.unimelb.edu.au/minizinc/download.html>

 tar xzvf fichero.tar.gz

 sh SETUP

 Añadir el path la carpeta bin (al .profile). En mi caso añadí al final:

```
PATH="$PATH:$HOME/docencia/1213/pr/minizinc-1.5.2/bin"
export PATH
```

 Para ver si funciona ir a la carpeta examples de la distribución y probar minizinc zebra.mzn; si da una salida que no sea error está bien instalado.

---


## Apéndice B: Otros sistemas para la programación con restricciones

 [Gecode](#): una librería para restricciones que se integra con C++.

 [Minion](#): resolutor escalable para dominios finitos

 [Choco](#): librería Java para PR.

 [JaCoP](#): otra librería Java para PR.

 [Comet](#): un lenguaje para PR similar a C++.

 [Sugar](#): un resolutor tipo SAT.

---

## Apéndice C: Para saber más

 Sobre el teorema de los 4 colores: [http://en.wikipedia.org/wiki/Four\\_color\\_theorem](http://en.wikipedia.org/wiki/Four_color_theorem)

 La página principal de MiniZinc: <http://www.g12.csse.unimelb.edu.au/minizinc/>

 Tutorial de MiniZinc: <http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-1.6/minizinc-tute.pdf>

---

