Tema 3: Lenguajes de modelado de problemas de optimización.

Objetivos del tema:

- ❑ Conocer las diferentes alternativas que existen en la actualidad para expresar y resolver problemas de optimización.
- ❑ Estudiar la estructura general de un lenguaje de modelado a través de OPL
- ❑ Conocer los principios operativos de los dos tipos de problemas expresables en OPL: CP y MP
- ❑ Utilizar las construcciones iterativas y operadores de agregación sobre variables de decisión y datos indexados
- ❑ Intercambiar datos entre OPL y una hoja de cálculo Excel

Alternativas para el desarrollo de modelos de optimización

Varias son las alternativas que se presentan a la hora de implementar un modelo de optimización lineal en un computador. Cada una presenta sus ventajas e inconvenientes, por lo que la elección deberá hacerse en cada caso en función del uso que se le vaya a dar al modelo. Podemos señalar cuatro alternativas principales:

1. **Hojas de cálculo con resolutor asociado.** Se trata de la alternativa más simple y asequible para especificar y resolver modelos pequeños de optimización lineal. La entrada y salida de datos se beneficia de todas las facilidades que presenta el entorno de la hoja, especialmente la posibilidad de representar gráficamente los resultados. El principal inconveniente proviene de la dispersión en que queda la especificación del modelo, al tener que expresarse a través de fórmulas asociadas a las diferentes celdas. Si el modelo tiene cierto tamaño, la implantación y mantenimiento se hace muy tediosa. Esta alternativa resulta válida a la hora de ensayar pequeños prototipos de modelos.

2. **Entornos de cálculo numérico y/o simbólico.** Muchos entornos de desarrollo matemático -numéricos y simbólicos- como MatLab, Maple o Mathematica, disponen de resolutores asociados que se pueden utilizar desde el propio medio de programación del entorno. La presentación de los resultados, como en el caso de las hojas de cálculo, se ve potenciada por las abundantes herramientas de visualización gráfica de que disponen. El principal inconveniente proviene de la ausencia de recursos específicos para la expresión de los modelos y la depuración de su funcionamiento.

3. **Biblioteca de algoritmos de optimización + lenguaje de programación de propósito general (Java, Fortran, C#, etc.).** Esta alternativa se ha venido utilizando cuando el modelo debía integrarse en una aplicación y requería interfaces específicas para la entrada y salida de datos. Se benefician de las facilidades que proporcionan los entornos de desarrollo para estos lenguajes, pero carecen de recursos de depuración orientados a un problema de optimización con restricciones. Requieren un tiempo largo de desarrollo y presentan dificultades a la hora del mantenimiento y modificación del modelo.

4. **Lenguajes algebraicos de modelado.** Estos lenguajes permiten expresar el modelo con una sintaxis próxima a la propia especificación matemática. Suelen disponer de recursos específicos para ayudar a la depuración del modelo. Las modificaciones y ampliaciones del modelo resultan relativamente fáciles, permitiendo una estrategia incremental de desarrollo. Los modernos lenguajes como OPL permiten la integración del modelo final en cualquier entorno de programación, por lo que en este aspecto no existe diferencia con la alternativa anterior. El principal inconveniente proviene de la limitación que presentan para el acceso a determinados recursos del resolutor sobre el que ejecuta el modelo. Por ejemplo, OPL no tiene acceso a las restricciones SOS1 y SOS2 ni a las variables indicadoras de reificación de CPLEX, muy importantes en la expresión eficiente de comportamientos no lineales. Esto se puede superar manipulando el modelo generado con OPL desde un lenguaje(C#, Java, C++) que disponga de una interfaz (API) con OPL.

Lenguajes de modelado

Los lenguajes de modelado de problemas de optimización son lenguajes declarativos con una sintaxis próxima a la especificación matemática de estos problemas. En esencia constituyen una interfaz de programación declarativa a través de la cual se utiliza el resolutor lineal de una forma más cómoda y productiva para el diseñador. En el siguiente esquema hemos representado su ubicación en el proceso de resolución de un problema de optimización:

Una de las principales aportaciones de estos lenguajes es el sistema de indexación de variables y datos que posibilita el uso de expresiones iterativas y funciones de agregación.

Existe un buen número de lenguajes de modelado y entornos de desarrollo asociados, muchos de ellos presentan características comunes, aunque difieren en las posibilidades de conexión e integración con fuentes de datos y entornos profesionales de desarrollo software. Una lista de los más importantes y utilizados sería la siguiente:

- **GAMS** http://www.gams.com/
- **Mosel** http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Mosel.aspx
- **AMPL** http://www.ampl.com/
- **AIMS** http://www.aimms.com/
- **Lingo** LINGO and optimization modeling (lindo.com)
- **MPL** http://www.maximal-usa.com/
- **OML** http://msdn.microsoft.com/en-us/library/ff524507(VS.93).aspx
- **OPL** https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio

En este curso, por motivos de eficiencia, utilizaremos exclusivamente el lenguaje OPL. Se trata de un lenguaje moderno, con abundantes recursos expresivos, un resolutor muy potente, CPLEX, con amplias facilidades de conexión a diferentes fuentes de datos (bases de datos, hojas de cálculo, archivos de texto) y con la posibilidad de integración directa de los modelos desarrollados en prácticamente todos los entornos modernos de software.

Entorno de desarrollo para OPL

• OPL es un lenguaje de modelado diseñado para facilitar la expresión de problemas de programación matemática continua y entera (MP) y de programación con restricciones (CP). Estos dos paradigmas de optimización con restricciones tienen orígenes y fundamentos teóricos bastante diferentes, aunque participan de la estructura general de los problemas de optimización.

• Para los problemas de programación matemática OPL dispone de un resolutor denominado CPLEX que integra un conjunto de algoritmos de optimización matemática (simplex, punto interior, red, B&B, etc.) .

• Para los problemas de programación con restricciones OPL dispone de un resolutor de propagación de consistencia y búsqueda denominado CP (*Constraint Programming*).

• Aunque ambos paradigmas de optimización son diferentes, comparten muchos recursos expresivos a nivel del modelado, por ello conviven en el mismo lenguaje y entorno de desarrollo. Pero conviene tener muy claro que el modelo que se ejecute deberá estar diseñado para un solo paradigma, que por defecto es el de la programación matemática.

• El objetivo de este tema es introducir las posibilidades de expresión de los lenguajes de modelado a través de OPL. No obstante daremos en las siguientes transparencias unas ideas muy generales e intuitivas de los mecanismos operacionales que utilizan ambos paradigmas, aunque el conocimiento de estos mecanismos no es indispensable para utilizar el lenguaje.

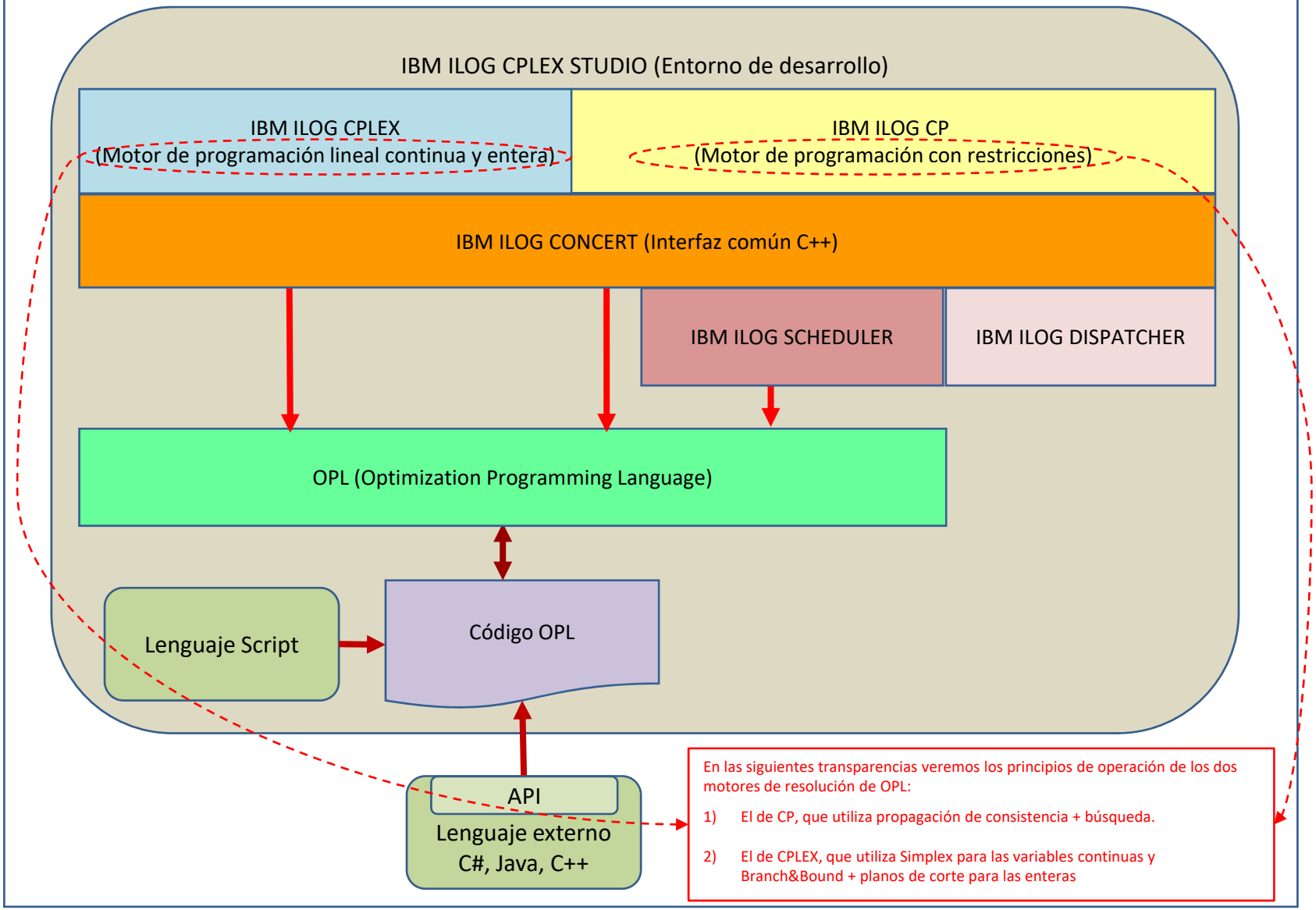
• La programación con restricciones opera sobre dominios finitos de enteros utilizando de forma entrelazada dos fases diferentes: propagación de consistencia para eliminar valores de los dominios de las variables que son inconsistentes con las restricciones, y búsqueda de soluciones sobre espacios de búsqueda reducidos por la propagación.

• La programación matemática opera sobre números reales o enteros utilizando principalmente el algoritmo del Simplex para los primeros y métodos de bifurcación y acotación (*branch&bound)* para los segundos. Estos métodos se potencian con la introducción de planos de corte, dando lugar a los métodos de bifurcación y corte (*branch&cut*)

• El entorno de desarrollo de OPL se denomina IBM ILOG CPLEX STUDIO y está compuesto por los dos resolutores mencionados (CP y CPLEX), una interfaz de programación (API) común denominada CONCERT, escrita en C++, con un conjunto de clases cuya funcionalidad facilita la especificación de problemas CP de *scheduling* y *dispatching.* El lenguaje OPL se sitúa como una interfaz de CONCERT que permite utilizar sus recursos de optimización de una forma declarativa. Además, el entorno dispone de un lenguaje de tipo *script* que permite manipular los modelos desde una semántica imperativa, haciendo posible la implementación de técnicas avanzadas de descomposición de problemas lineales. Finalmente existen interfaces de programación (APIs) que permiten modificar los datos, ejecutar modelos OPL, y obtener resultados desde muchos lenguajes de programación (C#, Java, C++, etc.)

• En la siguiente transparencia se presenta un esquema de bloques del entorno de desarrollo de OPL.

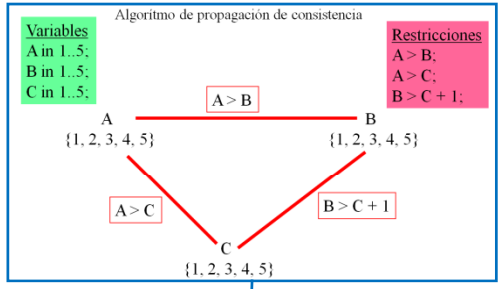
Lenguaje OPL (*Optimization Programming Language*)



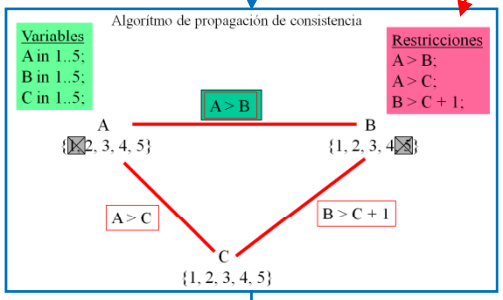
Perfil operativo de la programación con restricciones (CP): propagación de consistencia + búsqueda

Vamos a seguir el perfil operativo de CP utilizando el siguiente ejemplo de restricciones sin función de coste:

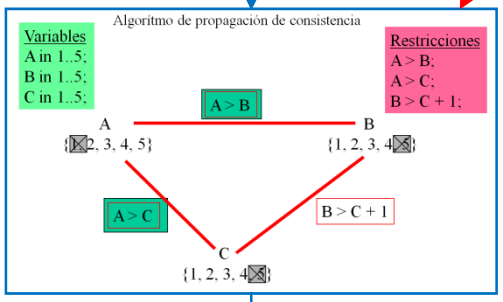
Variables de decisión
A in 1..5, B in 1..5, C in 1..5
Restricciones
A>B;
A>C;
B>C+1



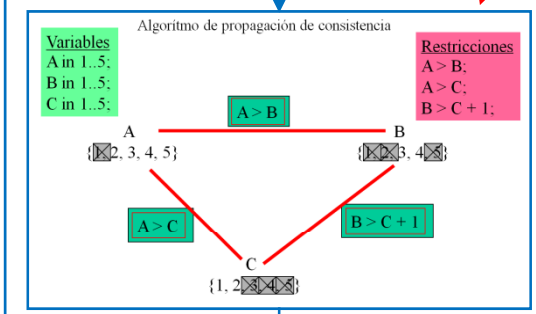
El problema queda definido por la declaración de las variables enteras y sus dominios de variación, y por el conjunto de restricciones. Se representa como un grafo con las variables en los nodos y las restricciones en los arcos.



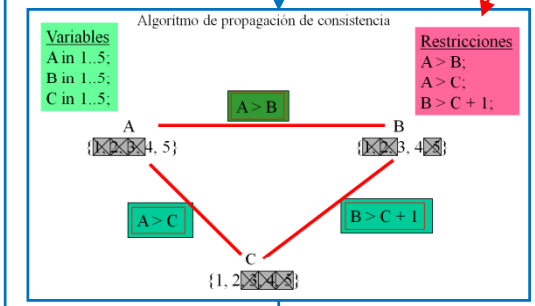
Al aplicar propagación de consistencia entre los nodos A y B utilizando la restricción A>B se eliminan de los dominios de A y B aquellos valores que no son consistentes en ningún caso con la restricción, es decir, el 1 en A (no es mayor que algún valor en B) y el 5 en B (no es menor que algún valor de A).



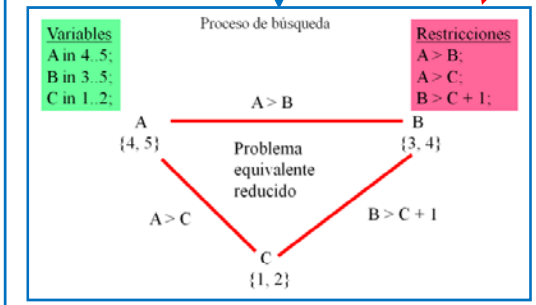
Se aplica propagación entre A y C utilizando la restricción de su arco común A>C. Se elimina el valor 5 de C por no ser soportado por ningún valor de A a través de A>C.



Se aplica propagación entre B y C utilizando la restricción de su arco común B>C+1. Se elimina el valor 3 de C por no ser soportado por ningún valor de B a través de B>C+1, y los valores 1 y 2 de B por no ser soportados por ningún valor de C a través de la misma restricción.

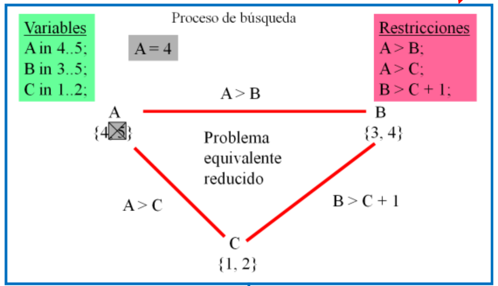


Después de aplicar consistencia entre variables surgen nuevas oportunidades de volver a aplicarla a las mismas variables al haberse reducido sus dominios en propagaciones de otras restricciones. En este caso se vuelve a aplicar A>B y se elimina 3 de A.

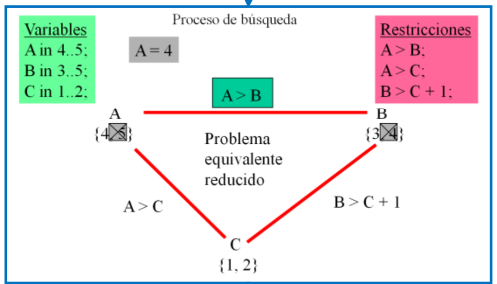


En un momento determinado se alcanza un punto fijo que no permite reducir más los dominios de las variables aplicado consistencia local entre variables a través de las restricciones. El problema original ha sido reducido a otro equivalente (con las mismas soluciones) pero con dominios reducidos

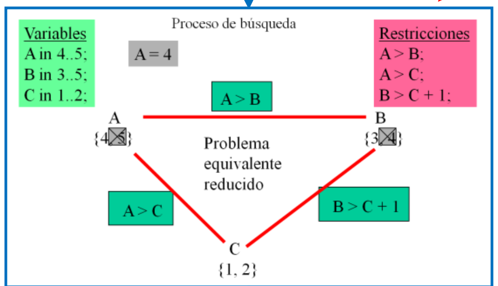
Perfil operativo de la programación con restricciones (CP): propagación de consistencia + búsqueda



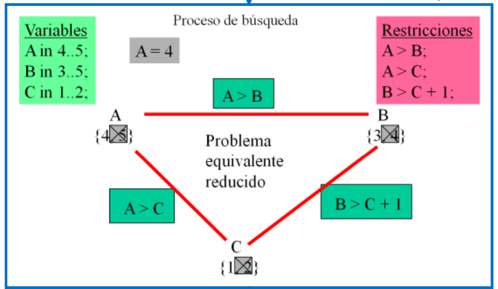
Cuando se alcanza un punto fijo se pasa a la fase de búsqueda: se elige una variable y se fija a algún valor de su dominio. En este caso elegimos A y el valor 4, lo que significa que eliminamos los demás valores de A (5 en este caso) y volvemos a aplicar propagación de consistencia.



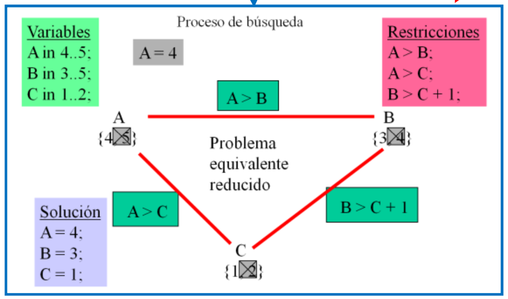
Se elimina el valor 4 de B al aplicar consistencia entra A y B a través de A>B.



La propagación de consistencia entre A y C a través de A>C no elimina ningún valor de los dominios de A o C.



La propagación de consistencia entre B y C a través de B>C+1 elimina el valor 2 de C



Se alcanza otro punto fijo pero en este caso con una solución (dominios de las variables con un solo valor): A=4, B=3, C=1.

Como se puede observar en CP es posible utilizar desigualdades estrictas (<,>) en las restricciones (cosa que no ocurre en programación lineal).

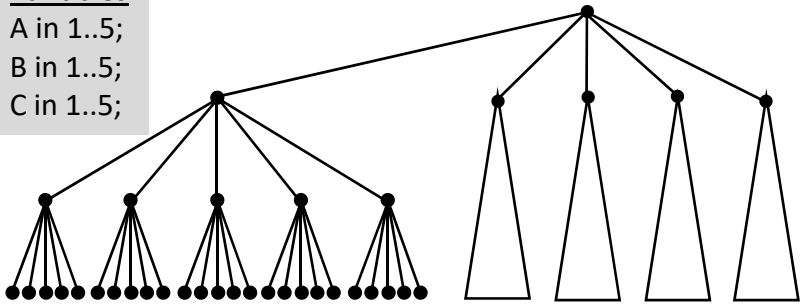
También son posibles términos no lineales e incluso restricciones predefinidas muy útiles en la expresión de determinados problemas, por ejemplo, *alldifferent(X1, X2,…)* que impone que las variables X1, X2, … tomen todas valores diferentes.

La eficiencia de CP depende de la capacidad de los propagadores diseñados para cada restricción y del proceso de búsqueda, que puede ser controlado por el programador con expresiones adecuadas que determinan el orden de las variables y el valor que se elige en cada paso de búsqueda.

Programación con restricciones: efecto de la propagación de consistencia entre dos búsquedas

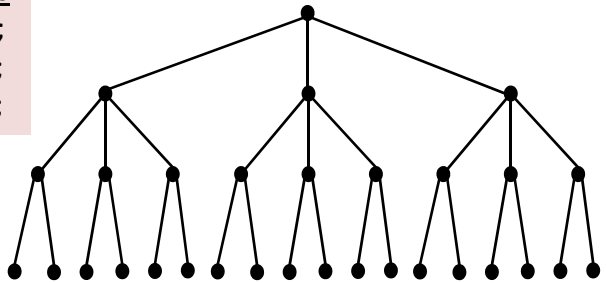
La propagación de consistencia reduce los espacios de búsqueda. En el ejemplo anterior hemos reducido un espacio de búsqueda original 5 x 5 x 5= 125 a un espacio de 1 x 3 x 2 = 6 después de la primera fase de propagación.

Variables
A in 1..5;
B in 1..5;
C in 1..5;



Espacios de búsqueda antes de la propagación

Variables
A in 4..5;
B in 3..5;
C in 1..2;



Espacios de búsqueda después de la propagación

Programación con restricciones: programa OPL

```
using CP;

dvar int A in 1..5;
dvar int B in 1..5;
dvar int C in 1..5;

subject to
{
  A > B;
  A > C;
  B > C + 1;
}
```

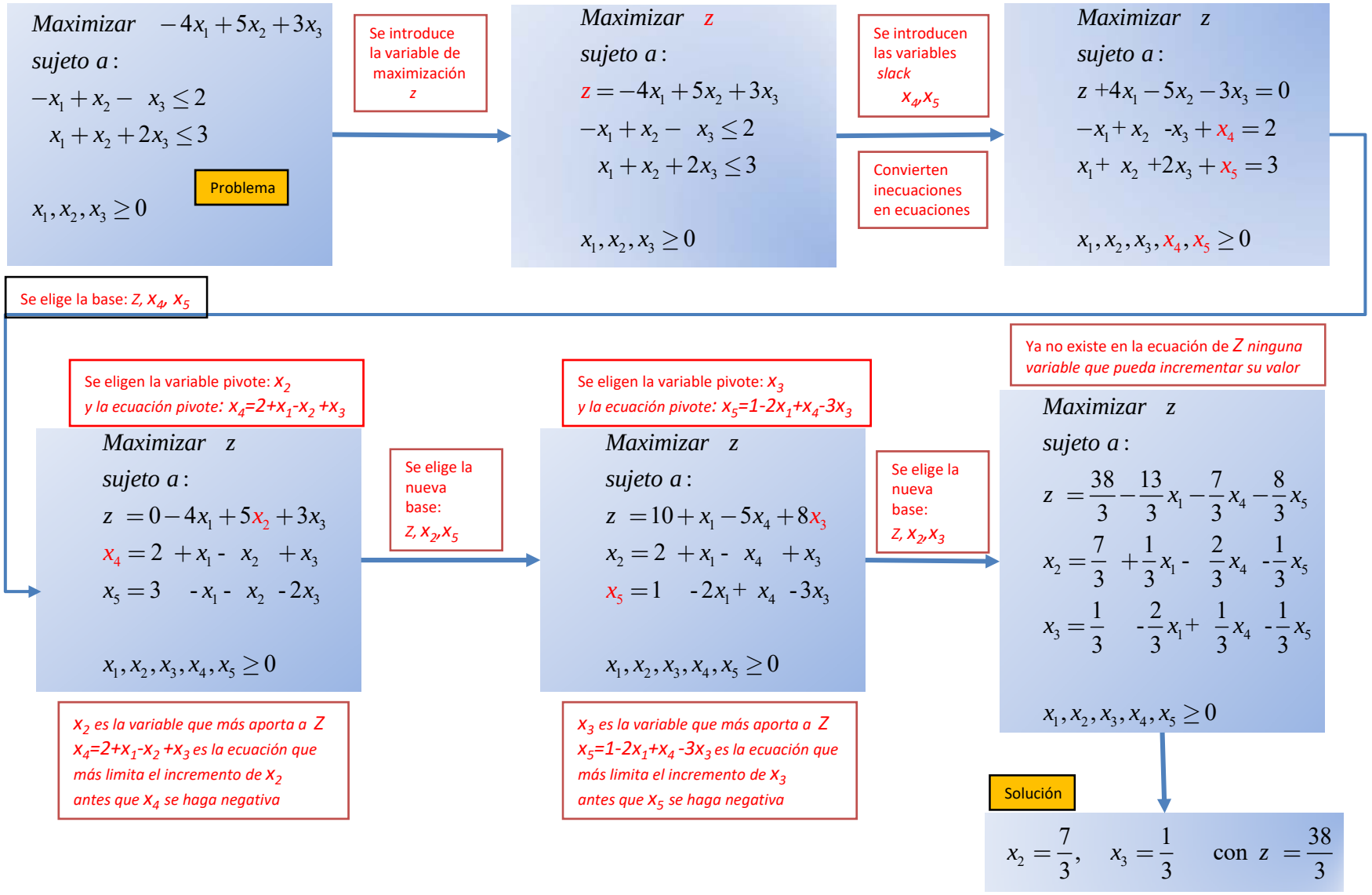
Por defecto los programas OPL se interpretan como programas lineales, para que se interpreten como programación con restricciones (CP) hay que escribir esta directiva.

Una desigualdad estricta produciría un error en un programa lineal

```
// solution
A = 4;
B = 3;
C = 1;
```

Perfil operativo de la programación lineal continua (CPLEX): algoritmo del Simplex

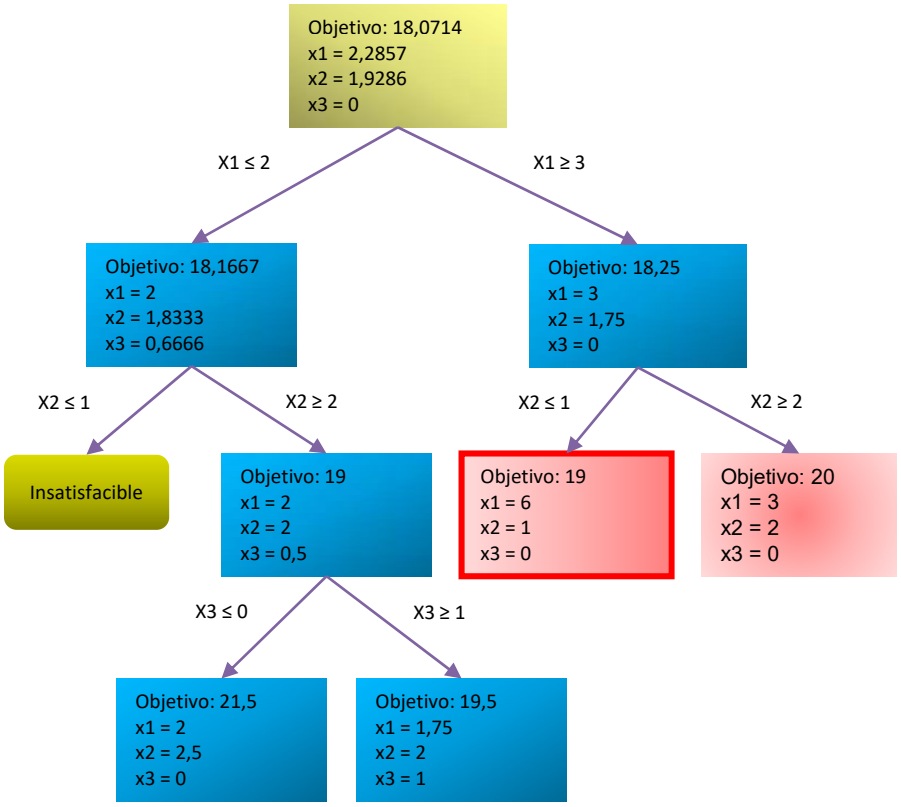
Este algoritmo resuelve problemas lineales con variables continuas realizando una serie de transformaciones sucesivas sobre el problema original:



Perfil operativo de la programación lineal entera (CPLEX): método de *Branch&Bound*

Los problemas lineales enteros se resuelven utilizando métodos de bifurcación y acotación. Se inicia el método relajando el carácter entero de las variables y resolviéndolo como si fuesen continuas. Si la solución óptima produce valores enteros para las variables, ésta es la solución del problema. En caso contrario se elige una variable cuyo valor no haya resultado entero, por ejemplo x1=2,2857 en el problema de la figura, y se crean dos sub-problemas (nodos), uno el problema original más la restricción x1=2, y otro el problema original más la restricción x1=3. Este proceso se denomina de bifurcación sobre la variable x1. El proceso continúa sobre los sub-problemas (nodos) pendientes de resolución. Cuando se obtenga una solución entera en algún nodo mejor que la última obtenida (y aún quedan nodos pendientes de proceso) se guarda como solución provisional. Cuando se obtenga una solución no entera pero con función de coste peor que la solución provisional, se aborta esa rama de proceso. También se aborta cuando el problema resulta insatisfacible o no acotado. La diferencia entre la función de coste de la solución entera provisional y la de la mejor relajación lineal no entera (nodo) constituye el *gap* de convergencia. Cuando este *gap* se hace cero o menor que un cierto valor umbral, la solución provisional se convierte en solución final.

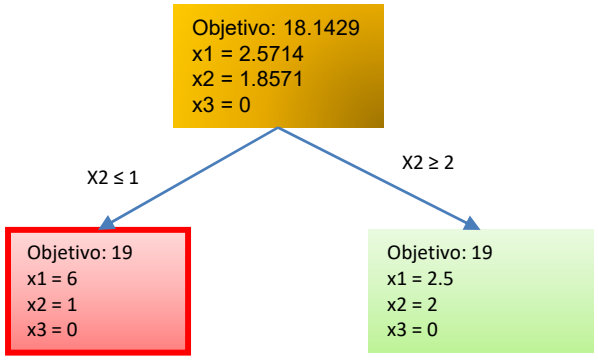
$$\begin{aligned} &\textit{Minimize} \quad 2x_1 + 7x_2 + 2x_3 \\ &s.t. \\ &x_1 + 4x_2 + x_3 \geq 10 \\ &4x_1 + 2x_2 + 2x_3 \geq 13 \\ &x_1 + x_2 - x_3 \geq 0 \\ &\\ &x_1, x_2, x_3 \geq 0 \;\; y \text{ enteras} \end{aligned}$$



Perfil operativo de la programación lineal entera (CPLEX): introducción de planos de corte (método de *Branch&Cut*)

El árbol de búsqueda del método de bifurcación y acotación se puede reducir introduciendo en los nodos restricciones que no eliminen soluciones enteras pero que reduzcan la región factible del los problemas relajados. Estas restricciones se denominan planos de corte (cortan la región factible) y se pueden obtener a partir de las restricciones del problema. Por ejemplo, si dividimos los dos miembros de la restricción $4x_1 + 2x_2 + 2x_3 \geq 13$ por 2 tenemos $2x_1 + 1x_2 + 1x_3 \geq 6,5$, y si las variables deben ser enteras, esa restricción es equivalente a $2x_1 + 1x_2 + 1x_3 \geq 7$, es decir, tiene las mismas soluciones enteras. Por tanto, la introducción de esta restricción en el problema no elimina soluciones enteras. Si ahora ejecutamos la relajación lineal con la restricción añadida tenemos:

$$
\begin{aligned}
&\textit{Minimize} \quad 2x_1 + 7x_2 + 2x_3 \\
&s.t. \\
&x_1 + 4x_2 + x_3 \geq 10 \\
&4x_1 + 2x_2 + 2x_3 \geq 13 \\
&2x_1 + x_2 + x_3 \geq 7 \\
&x_1 + x_2 - x_3 \geq 0 \\
&\\
&x_1, x_2, x_3 \geq 0 \ \ y \text{ enteras}
\end{aligned}
$$



Que evidentemente reduce el árbol de búsqueda.

Introducción a OPL

Para introducir la sintaxis y semántica operacional de OPL vamos a partir del modelo lineal con variables continuas de un problema sencillo de planificación de la producción. Después iremos viendo diferentes formas de expresión (versiones) del modelo en OPL. Esto nos permitirá introducir de forma gradual y motivada los principales recursos expresivos de OPL.

Ejemplo: **planificación de la producción 1**

Se trata de una factoría que produce amoniaco (NH_3) y cloruro amónico (NH_4Cl) a partir de nitrógeno (N), hidrógeno (H) y cloro (Cl). La cantidad disponible de cada uno de los tres componentes así como el beneficio por unidad que se obtiene de cada producto aparecen en la siguiente tabla. Se trata de planificar la producción para obtener el máximo beneficio.

Componentes	Productos		Disponibilidad componentes
	Cantidad de componente en cada producto		
	amoniaco (NH_3)	cloruro_amónico (NH_4Cl)	
nitrógeno	1*N	1*N	50
hidrógeno	3*H	4*H	180
cloro	0*Cl	1*Cl	40
Beneficio por unidad de producto	40	50	

Modelo lineal

Variables de decisión

```
amoniaco =        cantidad de amoniaco a producir
cloruro_amonico = cantidad de cloruro amónico a producir
```

Restricciones

```
Sujeto a       amoniaco + cloruro_amonico <= 50
               3 * amoniaco + 4 * cloruro_amonico <= 180
               cloruro_amonico <= 40;
               amoniaco >=  0
               cloruro_amonico >= 0
```

Función objetivo

```
Maximizar      40 * amoniaco + 50 * cloruro_amonico
```

Primera versión OPL

OPL permite escribir el modelo de forma casi literal a la especificación anterior, utilizando nombres independientes para cada variable de decisión (como ya hemos practicado en los dos primeros temas aunque utilizando nombres genéricos de variables).

p1.mod

```
// Variables de decisión
dvar float+ amoniaco;
dvar float+ cloruro_amonico;

// Función objetivo
Maximize    40*amoniaco+50*cloruro_amonico;

// Restricciones
subject to
   {
     amoniaco+cloruro_amonico <= 50;
     3*amoniaco+4*cloruro_amonico <= 180;
     cloruro_amonico <= 40;
   };
```

```
Optimal Solution with Objective Value: 2300

amoniaco = 20
cloruro_amonico = 30
```

Segunda versión OPL

Si queremos generalizar el modelo para que se pueda adaptar a otra situación en la que cambiemos o ampliemos los productos, OPL permite definir conjuntos de cadenas de caracteres (*strings*) para utilizar como índices de un único nombre para las variables de decisión. Al conjunto de índices podemos llamarle *Productos* y a la variable (indexada con Productos) *produccion*. La nueva versión del programa sería la siguiente:

p2.mod

```
// Indices
{string} Productos = {"amoniaco","cloruro_amonico"};

// Variables de decisión
dvar  float+  produccion[Productos];

// Función objetivo
maximize
  40*produccion["amoniaco"] + 50 * produccion["cloruro_amonico"];

// Restricciones
subject to {
  produccion ["amoniaco"] + produccion ["cloruro_amonico"] <= 50;
  3*produccion ["amoniaco"] + 4 * produccion ["cloruro_amonico"] <= 180;
  produccion ["cloruro_amonico"] <= 40;
};
```

```
Optimal Solution with Objective Value: 2300

produccion[amoniaco] = 20
produccion[cloruro_amonico] = 30
```

Tercera versión OPL

Podemos seguir el proceso de generalización del modelo indexando también los datos del problema *demanda*, *beneficio* y *stock* , previa introducción de un nuevo conjunto de índices denominado *Componentes*. De esta forma podremos utilizar una expresión genérica de sumatorio (sum) para la función de coste y una iteración (forall) más un sumatorio para la expresión de las restricciones. En ambos casos se introducen unos parámetros formales (*p* y *c*) que recorren los respectivos conjuntos.

p3.mod

```
// Indices
{string} Productos = {"amoniaco", "cloruro_amonico"};
{string} Componentes = {"nitrogeno", "hidrogeno", "oxigeno"};

// Datos
float demanda[Productos][Componentes] = [[1, 3, 0], [1, 4, 1]];
float beneficio[Productos] = [40, 50];
float stock[Componentes] = [50, 180, 40];

// Variables de decisión
dvar float+ produccion[Productos];

// Función objetivo
maximize
       sum(p in Productos) beneficio[p] * produccion[p];

// Restricciones
subject to {
       forall(c in Componentes)
          sum(p in Productos) demanda[p][c] * produccion[p] <= stock[c];
};
```

Equivalente a la expresión matemática:

$$\sum_{p \in Productos} beneficio_p \cdot produccion_p$$

Equivalente a la expresión matemática:

$$\forall c \in Componentes$$

Optimal Solution with Objective Value: 2300

produccion[amoniaco] = 20
produccion[cloruro_amonico] = 30

Cuarta versión OPL

Un grado más de independencia del modelo se consigue si se separan datos y código en archivos independientes. OPL permite descomponer el modelo en dos archivos, uno para el modelo propiamente dicho con extensión *.mod*, y otro para los datos con extensión *.dat*. La declaración de datos se sigue realizando en el archivo *.mod* pero de una forma elíptica, utilizando el símbolo (=…). Los dos archivos se pueden asociar en un proyecto utilizando el entorno de desarrollo. De esta forma podemos tener más de una instancia de datos (más de un archivo *.dat*) asociable a un mismo modelo (archivo *.mod*). Para nuestro problema de planificación de la producción tendríamos la siguiente descomposición:

p4.mod

```
// Declaración de índices
{string}Productos =…;
{string}Componentes …;

// Declaración de datos
float demanda[Productos][Componentes] = …;
float beneficio[Productos] = …;
float stock[Componentes] = …;

// Variables de decisión
dvar float+ produccion[Productos];

// Función objetivo
maximize
     sum(p in Productos) beneficio[p] * produccion[p];

// Restricciones
subject to {
      forall(c in Componentes)
            sum(p in Productos) demanda[p][c] * produccion[p] <= stock[c];
};
```

p4.dat

```
Productos = {"amoniaco", "cloruro_amonico"};
Componentes = {"nitrogeno", "hidrogeno", "oxigeno"};

demanda = [[1, 3, 0], [1, 4, 1]];
beneficio = [40, 50];
stock = [50, 180, 40];
```

Optimal Solution with Objective Value: 2300

produccion[amoniaco] = 20
produccion[cloruro_amonico] = 30

Estructura general de un programa OPL

En un programa OPL existen 4 zonas principales de información: Datos, Variables de decisión, Función de Coste y Restricciones:

Datos

Básicos
- Enteros
- Rango de enteros (utilizados como rango del índice de un array o como rango de variación de una variable entera)
- Reales
- Rango de reales (utilizados como rango de variación de una variable real)
- Cadenas de caracteres

Compuestos
- Tuples
- Arrays
- Conjuntos

Variables de decisión

Enteras
Reales

Función de coste

maximize o minimize *Expresión lineal de variables de decisión* (con datos constantes);

Restricciones

subject to
{
Expresión lineal de variables de decisión (<=, ==, <=) *Expresión lineal de variables de decisión*
}

En las siguientes transparencias estudiaremos con ejemplos los recursos expresivos de cada zona.

Datos básicos en OPL

Enteros

Inicialización mediante un valor

```
int a = 25;
```

Declara un número entero de valor 25

Inicialización mediante una expresión

```
int n = 3;
int dimension = n*n;
```

Declara un número entero de valor 9

Rangos de enteros

Valores en los límites

```
range Filas = 1..10;
```

Declara el rango de enteros que va de 1 a 10

Expresiones en los límites

```
int n = 8;
range Filas = n+1..2*n+1;
```

Declara el rango de enterosque va de 9 a 17

Reales

```
float  f = 3.2;
```

Declara un número real de valor 3.2

Rangos de reales

```
range rf = f1.0..f10.0;
```

Declara el rango de reales que va de 1 a 10

Cadenas (sring)

```
string s = "lunes";
```

Declara un cadena de caracteres (string) s de valor `"lunes";`

Datos compuestos en OPL

Conjuntos

```
{string} Componentes = {"nitrogeno", "hidrogeno", "oxigeno"};
```

Declara un conjunto de 3 strings

Tuples

```
tuple Ruta {
   string a;
   string c;
   };

Ruta t = <"a","b">;
```

Declara Ruta como un tipo *tuple* con dos componentes de tipo string: a y b. El tipo *tuple* es análogo al tipo *record* de los lenguajes de programación imperativos. Permite construir datos compuestos por otros datos (campos) de diferentes tipos

Declara el *tuple* t con valor <"a","b"> de tipo Ruta

Conjunto de tuples

```
{Ruta} rutas = {<"a1", "c1">, <"a1", "c4">, <"a4", "c4">, <"a5", "c4">} ;
```

Declara un conjunto de 4 *tuples* de tipo Ruta

Arrays

Enteros indexados con un rango definido

```
range R= 1..4;
int a[R] = [10, 20, 30, 40];
```

Declara un vector a de 4 enteros indexados por un rango de enteros, y los inicializa con los valores a[1]=10,…a[4]=40

Enteros indexados con un rango explícito

```
int e[1..4] = [10, 20, 30, 40];
```

Declara un vector e de 4 enteros indexados por un rango de enteros, y los inicializa con los valores e[1]=10,…e[4]=40

Enteros indexados con un conjunto de *strings*

```
int a [Componentes] = [10, 20, 30];
```

Declara un vector a de 3 enteros indexados por un conjunto de strings, y los inicializa con los valores a["nitrogeno"]=10, a["hidrogeno"]= 20, a["oxigeno"]=30.

Reales indexados con un rango explícito

```
float f [1..4] = [1.2, 2.3, 3.4, 4.5]
```

Declara un vector f de 4 reales indexados por un rango de enteros, y los inicializa con los valores f[1]=1.2,…f[4]=4.5

Array de string indexado con un rango explícito

```
string  s [1..2] = ["nitrogeno", "hidrogeno"];
```

Define s[1]= "nitrogeno" y s[2] = "hidrogeno"

Entero indexado con un tuple

```
tuple Arco {
              int origen;
              int destino;
             };
{Arco} Arcos = {<1, 2>, <1, 4>, <1, 5>};
int a[Arcos] = [10, 20, 30];
```

Define a[<1,2>], a[<1,4>] y a[<1,5>] con valores respectivos 10, 20 y 30

***Array* multidimensional frente a *array* unidimensional de registros**

En bastantes ocasiones es más económico definir *arrays* unidimensionales indexados sobre un *tuple* que *arrays* bidimensionales:

```
{string} Almacenes = {"a1", "a2", "a3", "a4", "a5"};
{string} Clientes = {"c1", "c2", "c3", "c4"};
int transporte1[Almacenes][Clientes] = [[2,0,0,5], [0,0,0,0], [0,0,0,0], [0,0,0,8], [0,0,0,9] ];
```

Un *array* bidimensional define valores de transporte para todas las combinaciones de almacenes y clientes, es decir, transporte["a1","c1"], transporte["a1","c2"] etc., aunque para muchas de ellas no exista conexión, recogiéndose este hecho con un valor 0.

```
tuple Ruta {
   string a;
   string c;
   };
{Ruta} rutas = {<"a1", "c1">, <"a1", "c4">, <"a4", "c4">, <"a5", "c4">} ;
int transporte2[rutas] = [2, 5, 8, 9];
```

Un array con índice tipo *tuple* permite definir valores de *transporte* sólo para la combinación de almacenes y clientes definidos en el conjunto de rutas

Inicialización externa de arrays

Los arrays se pueden inicializar con datos externos, declarándose en el archivo .mod e inicializándose en el archivo .dat:

```
// archivo .mod
int a[1..2][1..3] = ...;

// archivo .dat
a = [[10, 20, 30],[40, 50, 60]];
```

También se pueden inicializar especificando los pares (índice, valor). En este caso se deben utilizar los delimitadores #[y]# en lugar de [y].

```
//archivo .mod
int a[Dias] = ...;

/archivo .dat
a = #[
      "Lunes": 1,
      "Martes": 2,
      "Miercoles": 3,
      "Jueves": 4,
      "Viernes": 5,
      "Sabado": 6,
      "Domingo": 7
      ];#
```

El orden de los pares puede ser arbitrario, y las dos formas de inicialización se pueden combinar:

```
// archivo .mod
int a[1..2][1..3] = ...;

// archivo .dat
a = #[
      2: [40, 50, 60],
      1: [10, 20, 30]
     ]#;
```

Inicialización externa de tuples

Los tuples también se pueden inicializar con datos externos, declarándose en el archivo .mod e inicializándose en el archivo .dat. Se pueden inicializar dando la lista de los valores de los diferentes campos, o listando los pares (campo, valor). En este caso, como ocurre con los arrays, los delimitadores < y > se sustituyen por #< y ># y no importa el ordenamiento de los pares. Por ejemplo:

```
// archive  .mod
tuple punto
{
  int x;
  int y;
}
punto p1=...;
punto p2=...;

// archive  .dat
p1=#<y:1,x:2>#;
p2=<2,1>;
```

Inicialización externa de conjuntos

Los conjuntos de tuples también se pueden inicializar con datos externos, declarándose en el archivo .mod e inicializándose en el archivo .dat. Por ejemplo:

```
// archivo .mod
tuple Precedencia
{
  int antes;
  int despues;
}
{Precedencia} precedencias = ...;


// archivo .dat
precedencias = {<1,2>, <1,3>, <3,4>};
```

Variables de decisión

Las variables de decisión en OPL (CPLEX) sólo pueden ser de tipo entero o de tipo real (toman valores enteros o reales). Sus declaraciones respectivas son:

dvar int *nombre_de_la_variable_entera* in *rango_entero_de variación*

dvar float *nombre_de_la_variable_real* [in *rango_entero_de variación*]

El rango en las variables de decisión de tipo *float* es opcional

La no negatividad se puede especificar explícitamente con el signo + como subfijo del nombre del tipo:

dvar int+ *nombre_de_la_variable_entera* in *rango_entero_de variación*
dvar float+ *nombre_de_la_variable_real* [in *rango_entero_de variación*]

Las variables de decisión pueden indexarse de la misma manera que los datos, es decir, con rangos de enteros (implícitos o explícitos), conjuntos de *strings*, conjuntos de *tuples*, etc.

```
{string} dias = {"Lunes","Martes","Miercoles","Jueves","Viernes","Sabado","Domingo"};
{string} barcos= {"B1","B2","B3"};

dvar float+ var1[dias];
dvar int var2[barcos][dias] in 0..1;
```

Define 7 variables reales no negativas var1["Lunes"],…, var1["Domingo"]

Define 21 variables enteras var2["B1"]["Lunes"],…, var2["B3"]["Domingo"]

Las dimensiones de um array pueden ser de diferente tipo:

```
dvar int var3[1..10][dias] in 0..1;
```

Define 70 variables enteras var3[1]["Lunes"],…, var3[10]["Domingo"]

Función de coste

Una función de coste será una expresión lineal de variables de decisión:

*Expresión_lineal_de_variables_de_decisión := c1*var1 + c2*var2 + …*

Donde c1, c2,… son datos constantes (enteros o reales) y var1, var2,… variables de decisión (también enteras o reales)

La expresión lineal de variables y constantes indexadas se puede simplificar utilizando el siguiente operador de agregación :

sum(i in *rango_de_variación_de_i)*

El operador sum es equivalente al símbolo matemático del sumatorio. La expresión i in *rango_de_variación_de_i* se denomina parámetro formal y su sintaxis la veremos en la siguiente transparencia. Un ejemplo de expresión lineal indexada utilizada como función de coste de minimización sería la siguiente:

```
minimize  sum (i in dias) const1[i]*var1[i] + sum (p in productos) const2[p]*var2[p]
```

Restricciones

Una restricción tiene la forma general:

Expresión_lineal_de_variables_de_decisión_1 (<= ó == ó <=) Expresión_lineal_de_variables_de_decisión_2

Donde *Expresión_lineal_de_variables_de_decisión* tiene la misma sintaxis que en la función de coste. En particular una constante, incluida el 0, puede ser una *Expresión_lineal_de_variables_de_decisión.*

Para simplificar la expresión de restricciones se pueden incluir dentro de construcciones iterativas forall (i in *rango_de_variación_de_i)* siempre que el parámetro formal i aparezca como índice en las variables de decisión y/o constantes de las expresiones lineales. Por ejemplo, la expresión iterativa:

```
subject to
{
    forall (i in 1..3)const1[i]*var[i] >= const2[i];
}
```

Es equivalente a las siguientes tres restricciones:

```
subject to
{
    const1[1]*var[1] >= const2[1];
    const1[2]*var[2] >= const2[2];
    const1[3]*var[3] >= const2[3];
}
```

Parámetros formales

Los parámetros formales juegan un papel fundamental en OPL. Se utilizan con operadores de agregación, conjuntos y sentencias `forall`.

El parámetro formal más simple tiene la forma:

p in S

donde *p* es el parámetro formal y *S* el conjunto del cual *p* toma sus valores.

El conjunto *S* puede ser:

- Un rango de enteros

```
int n=6;
int s == sum(i in 1..n) i*i;
```

- Un conjunto de `strings`

```
{string} Productos ={"coche", "camion"};
float costo[Productos] =[12000,10000];
float maxCosto = max(p in Productos) costo[p];
```

- Un conjunto de tuples

```
{string} Ciudades = {"Paris", "Londres", "Berlin"};
tuple Conexion
{
   string orig;
   string dest;
}
{Conexion} conexiones ={<"Paris","Berlin">,<"Paris", "Londres">};
float costo[connections] = [ 1000, 2000 ];
float maxCosto= max(r in conexiones ) costo[r];
```

Parámetros formales con filtro

Si se necesita filtrar algunos valores del rango de un parámetro formal se pueden utilizar condiciones:

p in S : condicion

que asigna a p todos los elementos de S que cumplen la condición. Por ejemplo:

```
int n=8;
dvar int a[1..n][1..n];
subject to
{
 forall(i in 1..8)
   forall(j in 1..8: i < j)
     a[i][j] >= 0;
}
```

La restricción `a[i][j] >= 0` es modelada para todo `i` y `j` tal que $1 \le i < j \le 8$.

Se pueden combinar parámetros para producir una expresión más compacta. Por ejemplo, la dos declaraciones siguientes son equivalentes:

```
int s = sum(i,j in 1..n: i < j) i*j;
int s = sum(i in 1..n) sum(j in 1..n: i < j) i*j;
```

También son equivalentes las siguientes declaraciones:

```
int s = sum(i in 1..n, j in 1..m) i*j;
int s = sum(i in 1..n) sum(j in 1..m) i*j;
```

Estos parámetros también pueden someterse a condiciones de filtro. Por ejemplo la dos declaraciones siguientes son equivalentes:

```
forall(i,j in 1..n : i < j) a[i][j] >= 0;
forall(i in 1..n, j in 1..n : i<j) a[i][j] >= 0;
```

La siguiente es una expressión equivalente y aún más compacta:

```
forall(ordered i,j in 1..n)a[i][j] >= 0;
```

Ejemplo: planificación de la producción 2

Una compañía fabrica tres productos p1, p2 y p3 utilizando dos materias primas m1 y m2. En la siguiente tabla aparece el consumo de materias primas para fabricar cada producto (*consumo),* la *disponibilidad total de materias primas,* la *demanda* prevista, el *coste interno* de producción para cada producto, y el *coste externo* de cada producto para cuando la producción interna no sea suficiente y haya que recurrir a la compra externa de los productos. Hay que determinar la producción interna y externa de cada producto de manera que se minimice el costo total de la producción.

Componentes	Productos consumo			Disponibilidad de materias primas
	p1	p2	p3	
m1	0.5	0.4	0.3	20
m2	0.2	0.4	0.6	40
demanda	100	200	300	
coste interno	0.6	0.8	0.3	
coste externo	0.8	0.9	0.4	

Datos

$costInterno_p$
$costExterna_p$
$costeInventario_p$
$consumo_{rp}$
$disponibilidad_r$
$demanda_p$

$r \in MatPrimas$
$p \in Productos$

Variables de decisión

$prodInterna_p$
$prodExterna_p$

$$Minimizar\ z = \sum_{p \in Productos} costeInterno_p \cdot prodInterna_p + costExterna_p \cdot prodExterna_p$$

$$sujeto\ a: \sum_{p \in Productos} consumo_{pr} \cdot prodInterna_p \leq disponibilidad_r \quad \forall r \in MatPrimas$$

$$prodInterna_p + compraExterna_p \geq demanda_p \quad \forall p \in Productos$$

Ejemplo: Planificación de la producción 2 (Solución 1: uso de *arrays* para los datos de los productos)

.mod

```
{string} Productos =...;
{string} MatPrimas =...;

float consumo[Productos][MatPrimas] = ...;
float disponibilidad[MatPrimas] = ...;
float demanda[Productos] = ...;
float costeInterno[Productos] = ...;
float costeExterno[Productos]  = ...;

dvar float+ prodInterna[Productos];
dvar float+ compraExterna[Productos];

minimize
   sum(p in Productos) (costeInterno[p]*prodInterna[p] +
                        costeExterno[p]*compraExterna[p]);
subject to {
   forall(r in MatPrimas)
      sum(p in Productos) consumo[p][r] * prodInterna[p]  <= disponibilidad[r];

   forall(p in Productos)
      prodInterna[p] + compraExterna[p] >= demanda[p];
};
```

.dat

```
Productos =  { "p1", "p2", "p3" };
MatPrimas = { "m1", "m2" };

consumo = [ [0.5, 0.2],
            [0.4, 0.4],
            [0.3, 0.6]
          ];

disponibilidad = [ 20, 40 ];
demanda    = [ 100, 200, 300 ];
costeInterno   = [ 0.6, 0.8, 0.3 ];
costeExterno   = [ 0.8, 0.9, 0.4 ];
```

```
solution (optimal) with objective 372
prodInterna     = [40     0     0];
compraExterna = [60   200   300];
```

Ejemplo: Planificación de la producción 2 (Solución 2: uso de *tuples* para los datos de los productos)

.mod

```
{string} Productos =...;
{string} MatPrimas =...;

tuple DatosProd {
   float demanda;
   float costeInterno;
   float costeExterno;
   float consumo[MatPrimas];
};

DatosProd producto[Productos] = ...;
float disponibilidad[MatPrimas] = ...;

dvar float+ prodInterna[Productos];
dvar float+ compraExterna[Productos];

minimize
   sum(p in Productos) (producto[p].costeInterno*prodInterna[p] +
                        producto[p].costeExterno*compraExterna[p]);
subject to {
   forall(r in MatPrimas)
      sum(p in Productos)
        producto[p].consumo[r] * prodInterna[p] <= disponibilidad[r];

   forall(p in Productos)
      prodInterna[p] + compraExterna[p] >= producto[p].demanda;
```

.dat

```
Productos =  { "p1", "p2", "p3" };

MatPrimas = { "m1", "m2" };

producto = #[
      p1 : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >
      p2 : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >
      p3 : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >]#;

disponibilidad = [ 20, 40 ];
```

```
solution (optimal) with objective 372
prodInterna     = [40   0   0];
compraExterna = [60 200 300];
```

Ejemplo: Problema de la mochila con objetos de múltiples atributos

Objetos con múltiples atributos (por ejemplo, peso, volumen, etc.) y un valor (por ejemplo, euros) deben ubicarse en una mochila que tiene una determinada capacidad para cada atributo (peso máximo, volumen máximo, etc.) de manera tal que se maximice el valor de los objetos seleccionados.

$$Datos:$$

$$valor_i = valor\ del\ objeto\ i$$

$$capacidad\_objeto_{ji} = capacidad\ del\ objeto\ i\ en\ el\ atributo\ j$$

$$no = número\ de\ objetos$$

$$na = número\ de\ atributos$$

$$Variables de decisión:$$

$$seleccionar_i = número\ de\ objetos\ de\ tipo\ i\ seleccionados$$

$$Maximizar\ z = \sum_{i=1}^{no} valor_i \cdot seleccionar_i$$

$$sujeto\ a:$$

$$\sum_{i=1}^{no} capacidad\_objeto_{ji} \cdot seleccionar_i \quad \forall j = 1...na$$

```
int numObjetos =...;
int numAtributos =...;

range Objetos = 1..numObjetos;
range Atributos = 1..numAtributos;
int capacidad_mochila[Atributos] =...;
int valor[Objetos] =...;
int capacidad_objetos[Atributos][Objetos] =...;

int maxValor = max(a in Atributos) capacidad_mochila[a];

dvar int seleccionar[Objetos] in 0..maxValor;

maximize
   sum(o in Objetos) valor[o]*seleccionar[o];

subject to
{
   forall(a in Atributos)
        sum(o in Objetos) capacidad_objetos[a][o]*seleccionar[o]
 <= capacidad_mochila[a];
```

```
Optimal Solution with Objective Value: 261922
seleccionar[1] = 0
seleccionar[2] = 0
seleccionar[3] = 0
seleccionar[4] = 154
seleccionar[5] = 0
seleccionar[6] = 0
seleccionar[7] = 0
seleccionar[8] = 913
seleccionar[9] = 333
seleccionar[10] = 0
seleccionar[11] = 6499
seleccionar[12] = 1180
```

```
numObjetos = 12;
numAtributos = 7;
capacidad_mochila=[18209, 7692, 1333, 924, 26638, 61188, 13360];
valor = [96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81];
capacidad_objetos =
[
   [19,    1,  10,   1,    1,   14,  152, 11,   1,   1, 1, 1],
   [ 0,    4,  53,   0,    0,   80,    0,  4,   5,   0, 0, 0],
   [ 4,  660,   3,   0,   30,    0,    3,  0,   4,  90, 0, 0],
   [ 7,    0,  18,   6,  770,  330,    7,  0,   0,   6, 0, 0],
   [ 0,   20,   0,   4,   52,    3,    0,  0,   0,   5, 4, 0],
   [ 0,    0,  40,  70,    4,   63,    0,  0,  60,   0, 4, 0],
   [ 0,   32,   0,   0,    0,    5,    0,  3,   0, 660, 0, 9]
];
```

Ejemplo: Producción multi-período

Una compañía fabrica tres productos p1, p2 y p3 utilizando dos materias primas m1 y m2. El consumo de materias primas para cada producto aparece en la tabla *consumo*, así como la disponibilidad total de materias primas. La previsión de demanda para tres períodos (meses) de producción aparece en la tabla *demanda*. Los costes de producción internos para cada período aparecen en la tabla "coste". También aparece en esta tabla el coste externo de cada producto para cuando la producción interna no sea suficiente y haya que recurrir a la compra externa de los productos. El coste de los inventarios (almacenes) también aparece en esta tabla. Hay que determinar la producción interna y externa de cada producto así como los inventarios de cada período de manera que se minimice el costo.

consumo	p1	p2	p3	disponibilidad
m1	0.5	0.4	0.3	20
m2	0.2	0.4	0.6	40

demanda	t =1	t =2	t =3
p1	10	100	50
p2	20	200	100
p3	50	100	100

coste	p1	p2	p3
interno	0.4	0.6	0.1
externo	0.8	0.9	0.4
inventario	0.1	0.2	0.1

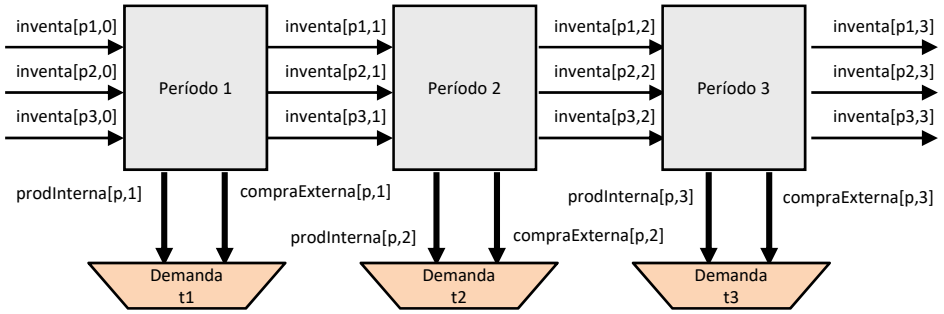
Datos

$costInterno_p$
$costExterna_p$
$costeInventario_p$
$consumo_{rp}$
$disponibilidad_r$
$demanda_{pt}$
$inventa_{pt}$
$inventario_p$

Variables de decisión

$prodInterna_{pt}$
$prodExterna_{pt}$
$inventario_{pt}$

$r \in matPrimas$
$p \in Productos;\ t = 0,1,2,3$



$$Minimizar\ z = \sum_{t=1}^{3} \sum_{p \in Productos} costInterno_p \cdot prodInterna_{pt} + costExterna_p \cdot prodExterna_{pt} + costeInventario_p \cdot inventario_{pt}$$

$$sujeto\ a: \sum_{p \in Productos} consumo_{rp} \cdot prodInterna_{pt} \le disponibilidad_r \quad \forall r,t \quad r \in matPrimas;\ t = 1,2,3$$

$$inventa_{pt-1} + prodInterna_{pt} + compraExterna_{pt} = demanda_{pt} + inventa_{pt} \forall p,t \quad p \in Productos;\ t = 1,2,3$$

$$inventa_{p0} = inventario_p \quad \forall p \in Productos$$

Producción multi-período: Modelo OPL

```
// .mod

// DECLARACION DE DATOS
{string} Productos =...;                    // enumeración de los productos
{string} MatPrimas =...;                    // enumeración de las materias primas
int nuPeriodos = ...;                       // número de  períodos de producción
range Periodos = 1..nuPeriodos;             // rango de períodos de producción
float consumo[MatPrimas][Productos] = ...; // materias primas  por producto
float disponibilidad[MatPrimas] = ...;      // disponibilidad de  materias primas
float demanda[Productos][Periodos] = ...;   // demanda de productos por período
float costInterno[Productos] = ...;         // coste interno de los productos
float costExterno[Productos]  = ...;        // coste externo de los productos
float inventario[Productos]  = ...;         // inventario inicial de cada producto
float costInventario[Productos]  = ...;     // coste almacenamiento de productos


// VARIABLES DE DECISION
dvar float+ prodInterna[Productos][Periodos];      // Producción interna de cada producto en cada período
dvar float+ compraExterna[Productos][Periodos];    // Compra externa de cada producto en cada período
dvar float+ inventa[Productos][0..nuPeriodos];     // Inventario de cada producto en cada período

// FUNCION DE OPTIMO
minimize
   sum(p in Productos, t in Periodos)
      (costInterno[p]*prodInterna[p][t]   +
       costExterno[p]*compraExterna[p][t] +
       costInventario[p]*inventa[p][t]);

// RESTRICCIONES
subject to {
   forall(r in MatPrimas, t in Periodos)
      sum(p in Productos)consumo[r][p] * prodInterna[p][t]
      <= disponibilidad[r];

   forall(p in Productos, t in Periodos)
      inventa[p][t-1] + prodInterna[p][t] + compraExterna[p][t]
      == demanda[p][t] + inventa[p][t];

   forall(p in Productos)
      inventa[p][0] == inventario[p]; };
```

```
// .dat
Productos =  { "p1", "p2", "p3"
};
MatPrimas = { "m1", "m2" };
nuPeriodos = 3;

consumo = [
           [ 0.5, 0.4, 0.3 ],
           [ 0.2, 0.4, 0.6 ]
          ];
disponibilidad = [ 20, 40 ];
demanda = [
           [ 10 100 50 ],
           [ 20 200 100 ],
           [ 50 100 100 ]
          ];
inventario = [ 0 0 0 ];
costInventario = [ 0.1 0.2 0.1];
costInterno = [ 0.4, 0.6, 0.1 ];
costExterno  = [ 0.8, 0.9, 0.4 ];
```

```
solution (optimal) with objective 457

prodInterna = [ [10  0      0    ]
                [ 0  0      0    ]
                [50 66.667 66.667]
              ];
compraExterna =[
                [ 0 100      50    ]
                [20 200     100    ]
                [ 0  33.333  33.333]
              ];
inventa = [
           [0 0 0 0]
           [0 0 0 0]
           [0 0 0 0]
         ];
```

Conexión de OPL con una hoja de cálculo (excel)

Establecimiento de la conexión

La sentencia:

```
// archivo  .dat
SheetConnection sheet("gasolina.xls");
```

Establece una conexión del programa OPL con una hoja de cálculo denominada *gasolina.xls.* En la especificación del archivo .xls se pueden utilizar caminos relativos al directorio del archivo *.dat*.

Lectura desde una hoja conectada

Los rangos de la hoja de cálculo pueden leerse sobre *arrays* unidimensionales o multidimensionales o sobre conjuntos. Por ejemplo:

```
// archivo .mod
{string} Gasolinas = ...;
tuple TipoGasolina
{
   float demanda;
   float precio;
   float octanaje;
   float plomo;
}
TipoGasolina gas[Gasolinas] = ...;

// archivo .dat
SheetConnection sheet("gasolina.xls");
Gasolinas from SheetRead(sheet,"gas!A2:A4");
gas from SheetRead(sheet,"gas!B2:E4");
```

Define *Gasolinas* como un conjunto de string

Define *TipoGasolina* como un tuple con 4 componentes de tipo float

Define *gas* como un array índexado sobre el conjunto de string *Gasolinas* y cuyos elementos son tuples de *TipoGasolina*

	A	B	C	D	E
1					
2	g1	5	7	10	13
3	g2	6	8	11	14
4	g3	7	9	12	15
5					

Archivo*: gasolina.xls*
Hoja*: gas*

Establece la conexión con el archivo excel gasolina.xsl

Instancia *Gasolinas* con los string del rango A2:A4 de la hoja *gas* del archivo *gasolina.xls*

Instancia *gas* con los float del rango B2:E4 de la hoja *gas* del archivo *gasolina.xls.* *gas[“g1”]=<5,7,10,13>, gas[“g2”]=<6,8,11,14>, gas[“g3”]=<7,9,12,15>*

Escritura en una hoja de cálculo

La salida de resultados sobre una hoja de cálculo conectada se realiza con sentencias del tipo:

```
a to SheetWrite(sheet,"RESULTADO!A2:A4");
b to SheetWrite(sheet,"RESULTADO!B2:D4");
```

Lleva el array a al rango A2:A4 de la hoja RESULTADO

Lleva el array b al rango A2:A4 de la hoja RESULTADO

En este caso OPL abre la hoja de cálculo en modo lectura-escritura, y la acción puede fallar si otro proceso está ya utilizando la misma hoja de cálculo. Los tipos utilizados son los mismos que en la lectura. Las celdas se rellenan de izquierda a derecha y de arriba a abajo.

Ejemplo: lectura de los datos del modelo p4.mod desde una hoja excel

p4.dat

```
Productos = {"amoniaco", "cloruro_amonico"};

Componentes = {"nitrogeno", "hidrogeno", "oxigeno"};

demanda = [[1, 3, 0], [1, 4, 1]];

beneficio = [40, 50];

stock = [50, 180, 40];
```

p4.dat

```
// Conexión con la hoja de cálculo
SheetConnection sheet("produccion.xls");



// Lectura de datos
Productos from SheetRead(sheet,"Hoja1!A10:A11");

Componentes from SheetRead(sheet,"Hoja1!A4:A6");

demanda from SheetRead(sheet,"Hoja1!B4:C6");

beneficio from SheetRead(sheet,"Hoja1!B10:B11");

stock from SheetRead(sheet,"Hoja1!D4:D6");


// Escritura de resultados

produccion to SheetWrite(sheet,"Hoja1!D10:D11");
```

El archivo .xls deberá estar en el mismo directorio que el .dat. En caso contrario habrá que especificar el camino de ubicación que puede ser relativo a la ubicación de .dat

produccion.xls

	A	B	C	D	E
1					
2					
3	Componentes	demanda		stock	
4	nitrogeno	1	1	50	
5	hidrogeno	3	4	180	
6	oxigeno	0	1	40	
7					
8					
9	Productos	beneficio		produccion	
10	amoniaco	40		20	
11	cloruro_amonico	50		30	
12					
13					
14					
15					

Ejercicio 1

Una empresa debe contratar trabajadores de un conjunto {1, 2, ...,32} para construir un edificio. El trabajo implica una serie de tareas t1, t2,...,t15 específicas cuya realización requiere cualificación. Cada trabajador está cualificado para un subconjunto de tareas y tiene un coste de contratación. Determinar el subconjunto de trabajadores que hay que contratar de manera que reúna todas las cualificaciones necesarias para la construcción del edificio minimizando el coste.

```
nuTrabajadores = 32;
Tareas = {t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15 };
cualificacion =
[
   { 1  9 19  22  25  28  31 }              // trabajadores cualificados para la tarea t1
   { 2 12 15 19 21 23 27 29 30 31 32 }      // trabajadores cualificados para la tarea t2
   { 3 10 19 24 26 30 32 }                  // trabajadores cualificados para la tarea t3
   { 4 21 25 28 32 }                        // trabajadores cualificados para la tarea t4
   { 5 11 16 22 23 27 31 }                  // trabajadores cualificados para la tarea t5
   { 6 20 24 26 30 32 }                     // trabajadores cualificados para la tarea t6
   { 7 12 17 25 30 31 }                     // trabajadores cualificados para la tarea t7
   { 8 17 20 22 23  }                       // trabajadores cualificados para la tarea t8
   { 9 13 14 26 29 30 31 }                  // trabajadores cualificados para la tarea t9
   {10 21 25 31 32 }                        // trabajadores cualificados para la tarea t10
   {14 15 18 23 24 27 30 32 }               // trabajadores cualificados para la tarea t10
   {18 19 22 24 26 29 31 }                  // trabajadores cualificados para la tarea t12
   {11 20 25 28 30 32 }                     // trabajadores cualificados para la tarea t13
   {16 19 23 31 }                           // trabajadores cualificados para la tarea t14
   {9 18 26 28 31 32 }                      // trabajadores cualificados para la tarea t15
 ];
coste = [ 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 6 6 6 7 8 9 ]; //coste de cada trabajador
```

Ejercicio 2

Ubicar los datos del problema de la **producción multi-período** en una hoja excel y modificar el código OPL del archivo *.dat* para que la lectura de datos y escritura de resultados se haga desde el dicha hoja.