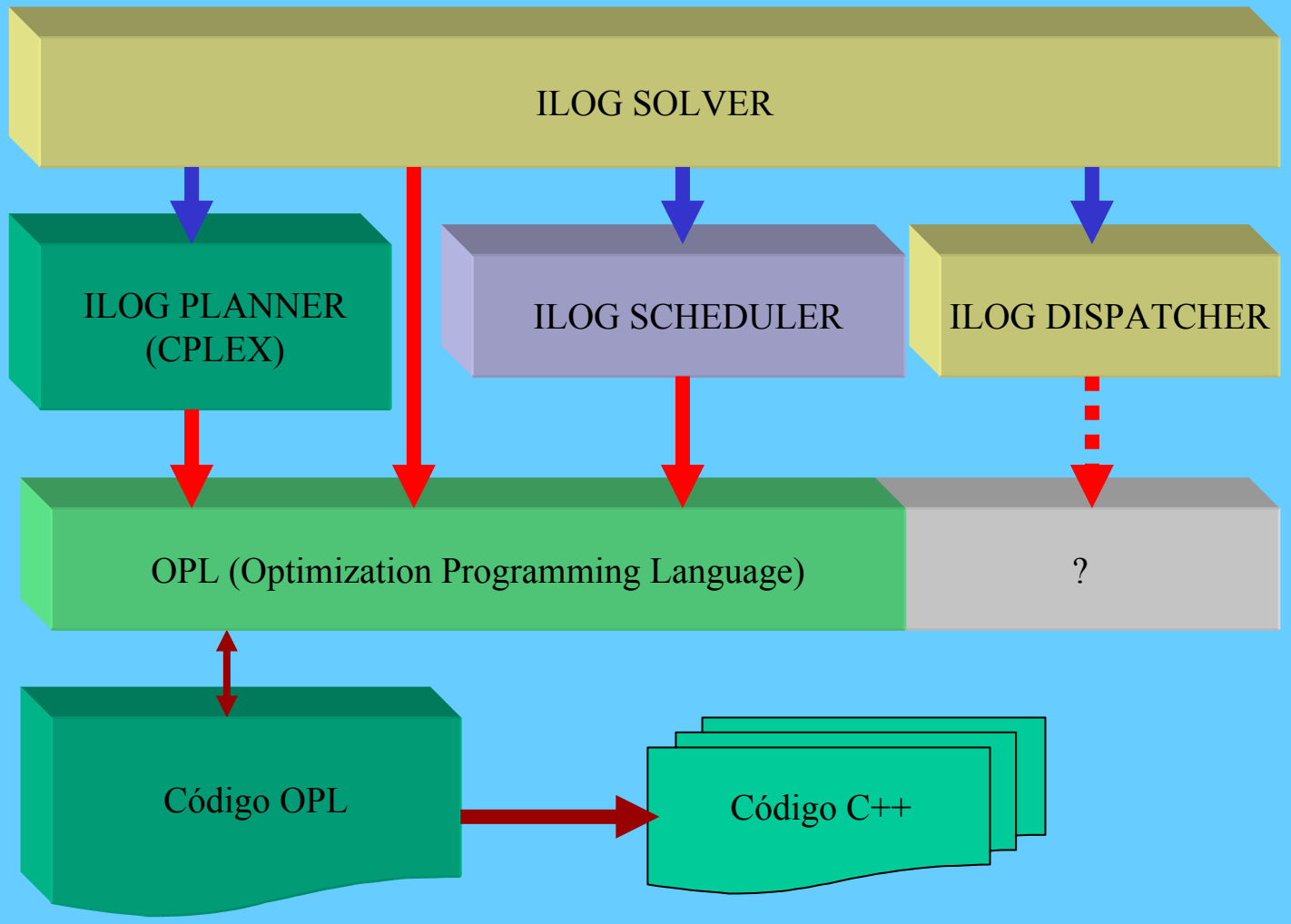
PROGRAMACION CON RESTRICCIONES Y OPTIMIZACION (OPL)
(Introducción al curso de doctorado)

Programa:

1. Programación lineal, entera y mixta en OPL
2. Conexión con bases de datos relacionales
3. Programación con restricciones en OPL
4. Modelos de Planificación (*scheduling*) en OPL
5. OPLScript y Generación de código C++ para modelos OPL.

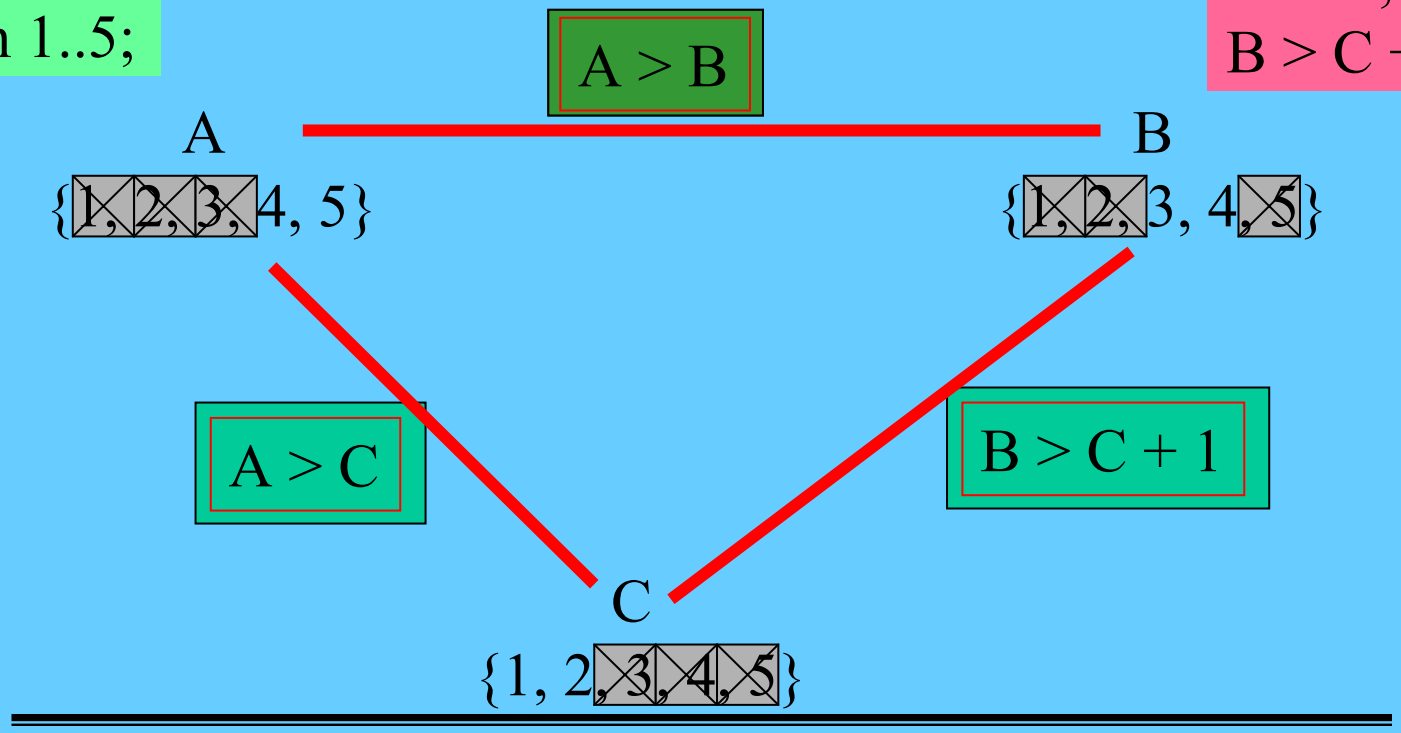
OPTIMIZACION ILOG



Algorítmo de propagación de consistencia

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

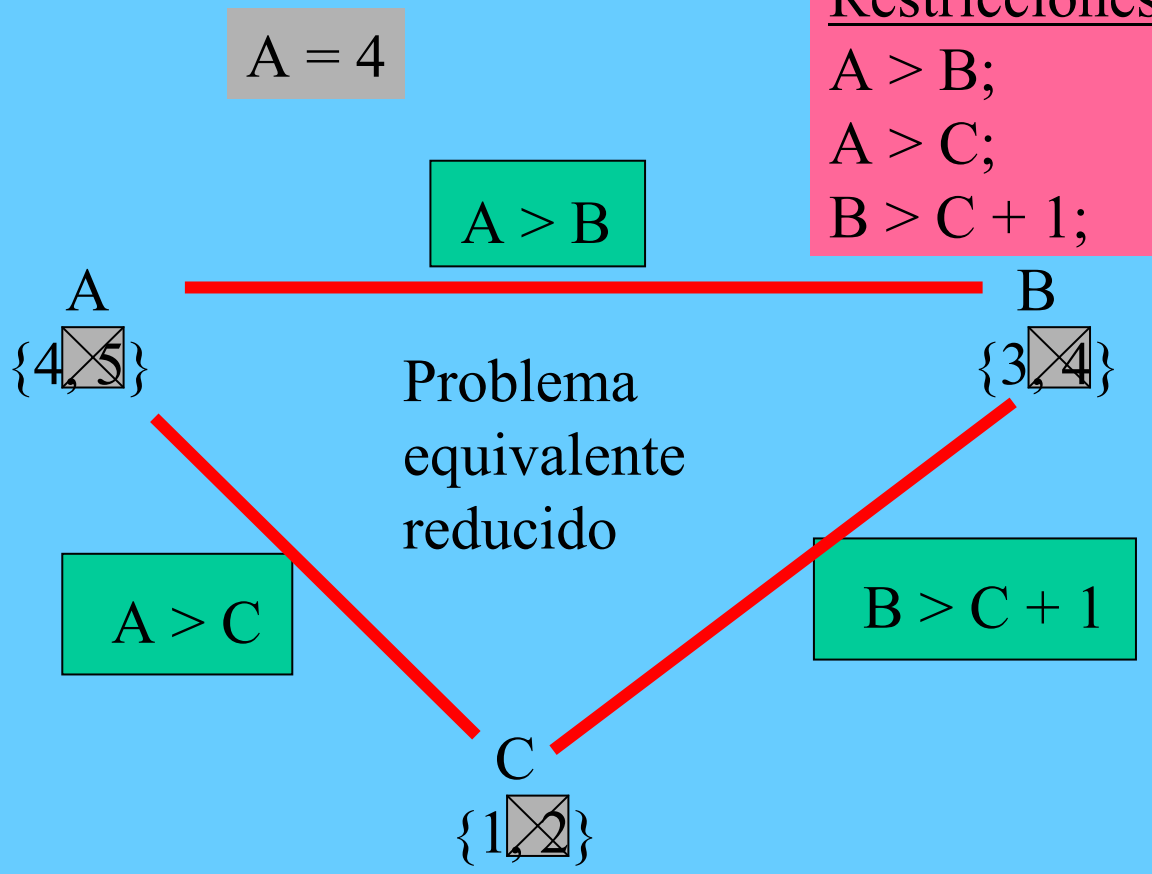
Restricciones
A > B;
A > C;
B > C + 1;



Proceso de búsqueda

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

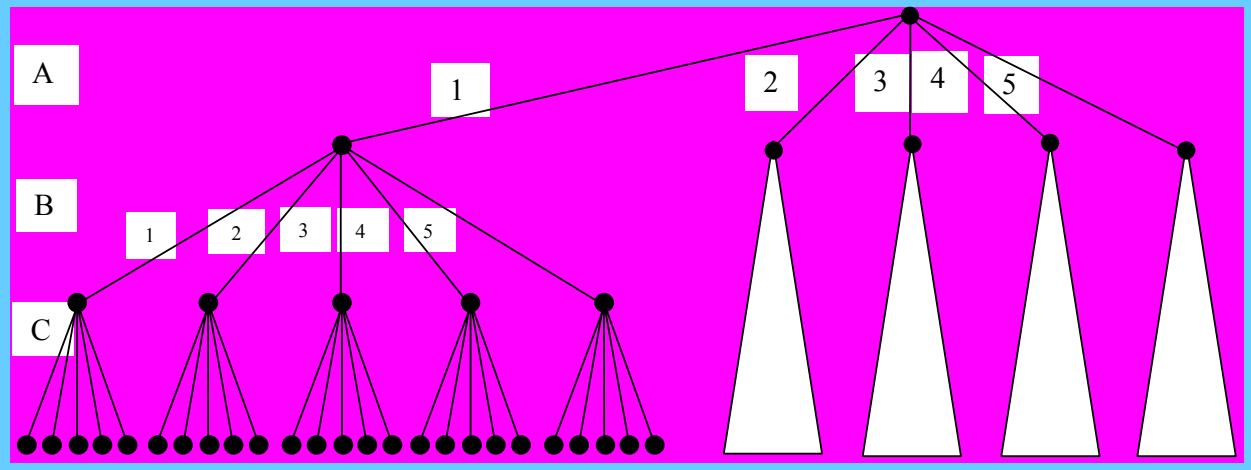
Restricciones
A > B;
A > C;
B > C + 1;



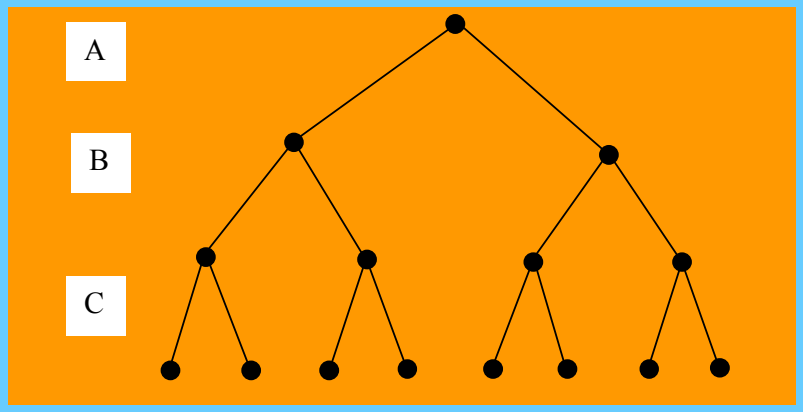
Solución
A = 4;
B = 3;
C = 1;

Espacios de búsqueda antes y después de la propagación

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



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



ILOG PLANNER
(CPLEX)

Programación lineal

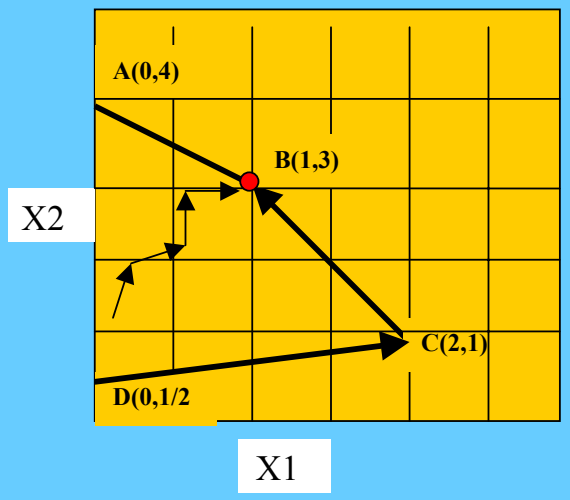
Algoritmo del Simplex
(Dantzing 1947)

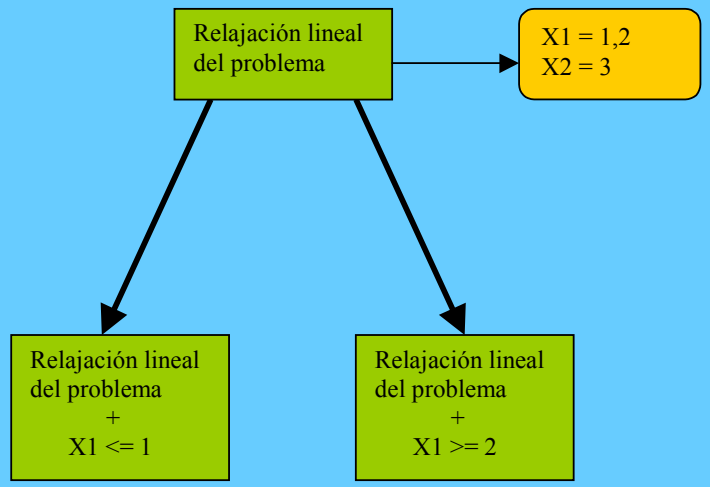
Algoritmo de barrera
(punto interior Karmarkar 1984)

$$\text{Maximizar } 3x_1 + 2x_2$$
$$x_1 + x_2 \leq 4$$
$$2x_1 + x_2 \leq 5$$
$$-x_1 + 4x_2 \geq 2$$

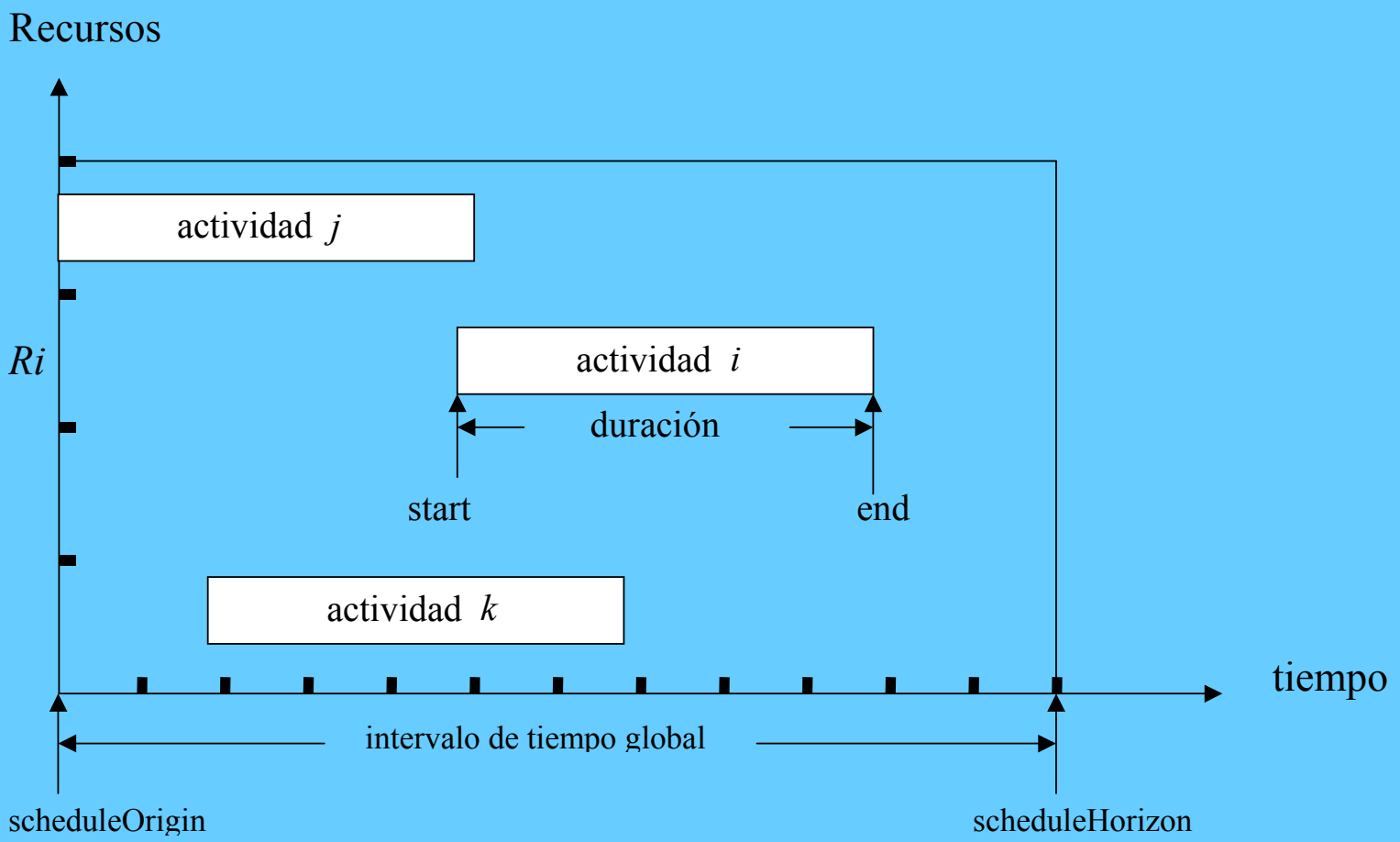
Programación entera/mixta

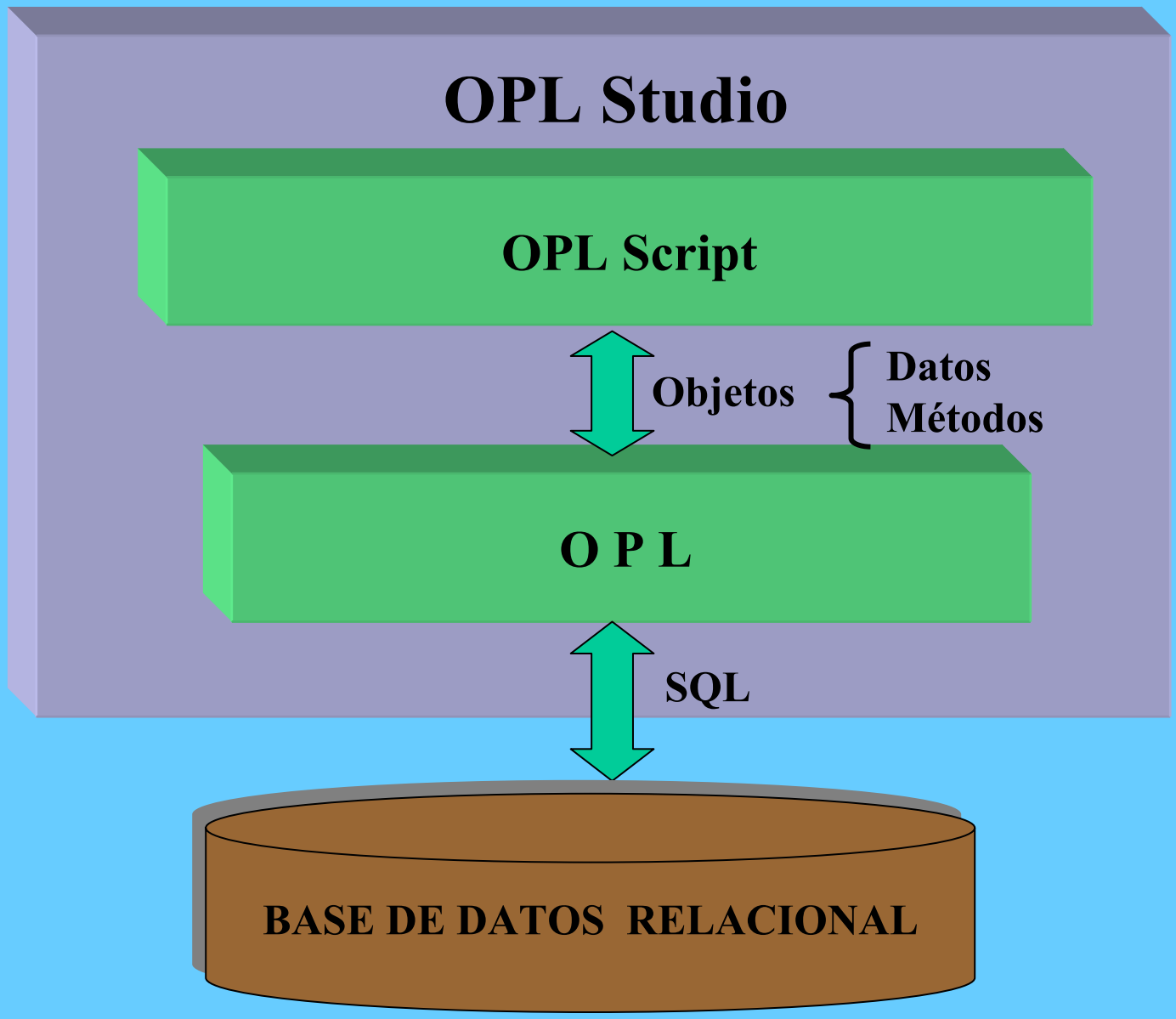
Algoritmo de ramificación y acotación
(branch&bound)





ILOG SCHEDULER




OPL Studio
OPL Script
Objetos
Datos
Métodos
O P L
SQL
BASE DE DATOS RELACIONAL


Programación Lógica (LP) →
Programación Lógica con Restricciones: CLP(X)
Unificación
Restricciones sobre reales: CLP(R)
Restricciones sobre dominios finitos: CLP(FD)
Restricciones sobre conjuntos
Inteligencia Artificial (AI)
Resolución de restricciones (CSP)
(arco consistencia+retroceso)
Investigación Operativa
Programación lineal y entera
+
AMPL
Programación Orientada a Objetos
CLP en C++ (ILOG SOLVER)
(biblioteca de clases para resolución de restricciones)
Programación con restricciones
OPL = Programación lineal/entera (AMPL)
(simplex, barrera)
+
Programación con restricciones (DF)
(propagación de consistencia)

Programación lineal

Minimizar (Maximizar) $\sum_{j=1}^{n} c_j x_j$

Sujeto a $\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1,2,\ldots,m$

$x_j \geq 0, \quad x_j \in \Re \quad j = 1,2,\ldots,n$

Ejemplo

Maximizar $3x_1 + 2x_2$

Sujeto a

$$x_1 + x_2 \leq 4$$
$$2x_1 + x_2 \leq 5$$
$$-x_1 + 4x_2 \geq 2$$
$$x_1, x_2 \geq 0 \qquad x_1, x_2 \in \Re$$

Programación lineal

Ejemplo

Planificación de la producción 1

Productos	beneficio	Componentes		
		nitrógeno	hidrógeno	cloro
amoniaco (NH3)	40	1*N	3*H	0*Cl
cloruro_amónico (NH4Cl)		50	1*N	4*H
		50	180	40

demanda c
componentes para
1*CL cada prod

stock[Componentes]

Programación lineal

Ejemplo

Maximizar 40 * amoniaco + 50 * cloruro_amonico

Sujeto a amoniaco + cloruro_amonico <= 50

3 * amoniaco + 4 * cloruro_amonico <= 180

cloruro_amonico <= 40;

amoniaco >= 0

cloruro_amonico >= 0

Primera versión OPL

p1.mod

```
var float+ amoniaco;
var float+ cloruro_amonico;

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

Optimal Solution with Objective Value: 2300.000000

amoniaco = 20.000000
cloruro_amonico = 30.000000

Segunda versión OPL

p2.mod

```
enum  Productos {amoniaco,  cloruro_amonico};
var  float+  produccion[Productos];

maximize
  40 * produccion[amoniaco] + 50 * produccion[cloruro_amonico]

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: 2300.000000

produccion[amoniaco] = 20.000000
produccion[cloruro_amonico] = 30.000000

Tercera versión OPL

p3.mod

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

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

var float+ produccion[Productos];

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

Optimal Solution with Objective Value: 2300.000000

produccion[amoniaco] = 20.000000
produccion[cloruro_amonico] = 30.000000

Cuarta versión (modelo)

Separación *modelo del problema* - *instancia de datos* dentro de un proyecto

p4.mod

```
enum  Productos …;
enum Componentes …;

float+ demanda[Productos, Componentes] = …;
float+ beneficio[Productos] = …;
float+ stock[Componentes] = …;

var float+ produccion[Productos];


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

Cuarta versión (datos)

p4.dat

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

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

Optimal Solution with Objective Value: 2300.000000

produccion[amoniaco] = 20.000000
produccion[cloruro_amonico] = 30.000000

Tipos de datos en OPL

Tipos básicos

Enteros (int)
Reales (float)
Enumerados (enum)

Tipos estructurados

Rangos (range)
Arrays
Registros
Conjuntos

Tipos enteros

Inicialización mediante un valor

int a = 25;

declara un entero de valor 25

int+ a = 25;

declara un entero no negativo de valor 25

Inicialización mediante una expresión

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

declara un entero de valor 9

Inicialización mediante una consulta al usuario

int numReinas << “Numero de reinas: “;

Tipos reales

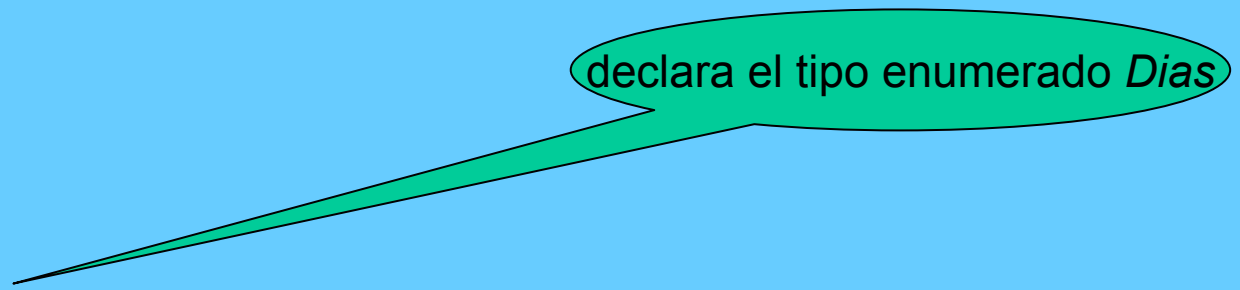
float f = 3.2;

declara un real de valor 3.2

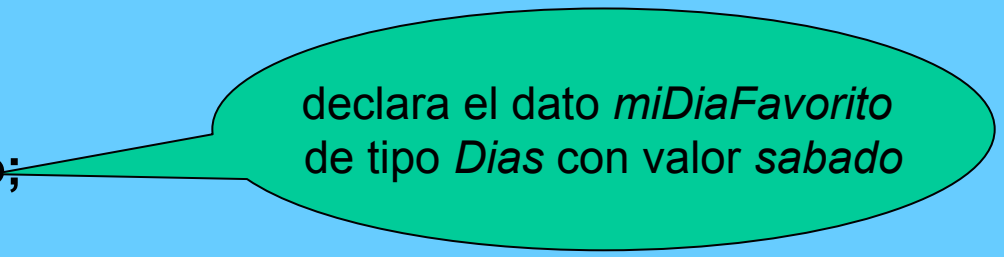
float+ f = 4.0;

declara un real no negativo de valor 4.0

Tipos enumerados




```
enum Dias { lunes, martes, miercoles, jueves, viernes, sabado, domingo
```




```
Dias miDiaFavorito = sabado;
```

Rangos de enteros

Valores en los límites

range Filas 1..10;

declara el rango 1..10

Filas r = 8;

declara un dato *r* de tipo (variando en) *Filas* con valor *8*

Filas v[1..3] = [1,2,3];

Declara el array *v* variando en *Filas*con valor [1,2,3]

Expresiones en los límites

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

Arrays

Tipos base:

enteros (int)

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

reales (float)

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

enumerados (enum)

Dias d [1..2] = [lunes, martes];

Arrays

Indices:

rango de enteros

explícito

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

implícito

range R 1..4;

int a[R] = [10, 20, 30, 40];

define a[1], a[2], etc. con valores 10, 20, etc.

define a[lunes], a[martes], etc. con valores 10, 20, etc.

tipo enumerado

int a [Dias] = [10, 20, 30, 40, 50, 60, 70];

Arrays (índices)

conjuntos finitos de tipos arbitrarios

```
struct 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 10, 20 y 30

EJEMPLO

```
enum Tarea = {t1, t2, t3, t4, t5, t6, t7, t8, t9};
enum Recurso = {r1, r2, r3, r4};

{Tarea} re = {t1, t2, t9};
{Tarea} r[Recurso] = [{t1, t2, t3}, {t5, t6}, {t1, t7, t8}, {t2, t8, t9}];

struct asigna {
        Tarea  uno;
        Tarea  dos;
};
{asigna}  asignas = {<t1, t2>, <t2, t3>, <t5, t6>};

{asigna}  asignar[Recurso] = [{<t1, t2>, <t5, t6>}, {<t2, t3>, <t8, t9
                              <t7, t8>}, {<t4, t5>}, {<t5, t6>, <t6, t3>}]
```

Arrays multidimensionales

OPL soporta la definición de arrays con más de una dimens
y la mezcla de tipos en los índices

```
int a[1..2, 1..3] = …;

int a[Dias, 1..3] = …;
```

Array multidimensional frente a array unidimensional de registros

define valores enteros 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 sea necesario, recogiendose este hecho con un valor 0.

Array bidimensional

enum Alacenes {a1, a2, a3, a4, a5};
enum Clientes {c1, c2, c3, c4};
int transporte[Almacenes, Clientes] = [[2,0,0,5], [0,0,0,0], [0,0,0,0], [0,0,0,8], [0,0,0,

Array unidimensional de registros

enum Almacenes {a1, a2, a3, a4, a5};
enum Clientes {c1, c2, c3, c4};
struct Ruta {
Almacenes a;
Clientes c;
};
{Ruta} rutas = {<a1, c1>, <a1, c4>, <a4, c4>, <a5, c4>} ;
int transporte[rutas] = [2, 5, 8, 9];

define valores enteros de *transporte* sólo para la combinación de almacenes y clientes definidos en el conjunto de estructuras rutas

Planificación de la producción 2

Productos	MatPrimas		demanda	costoInterno	costoExterno
	m1	m2			
	consumo				
p1	0.5	0.2	100	0.6	0.8
p2	0.4	0.4	200	0.8	0.9
p3	0.3	0.6	300	0.3	0.4
	20	40			
disponibilidad					

Solución 1: uso de *arrays* para los datos de los productos

p5.mod

```
// DECLARACION DE DATOS
enum Productos ...;
enum MatPrimas ...;

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

// VARIABLES  DE  DECISION

var float+ prodInterna[Productos];
var float+ compraExterna[Productos];
```

p5.mod

```
// FUNCION  DE  OPTIMO
minimize
   sum(p in Productos) (costeInterno[p]*prodInterna[p] +
                        costeExterno[p]*compraExterna[p])
// RESTRICCIONES
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];
};
```

Planificación de la producción 2

p5.dat

```
// productos elaborados por la compañia
Productos       =       { p1 p2 p3 };

// materias primas utilizadas en la elaboración de los producto
MatPrimas       =       { m1 m2 };

// consumo de paterias primas
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 ];
```

Planificación de la producción 2

Solución 2: uso de *records* para los datos de los productos

p6.dat

```
// productos elaborados por la compañia
Productos =  { p1, p2, p3 };

// materias primas utilizadas en la elaboración de los productos
MatPrimas = { m1, m2 };

// datos de cada uno de los productos: demanda,
// coste de producción interna, coste de producción externa y
// consumo de paterias primas
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 de cada materia prima
disponibilidad = [ 20, 40 ];
```

Solución 2: uso de *records* para los datos de los productos

p6.mod

```
// DECLARACION DE DATOS
enum Productos ...;
enum MatPrimas ...;

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

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

Planificación de la producción 2

Solución 2: uso de *records* para los datos de los productos

p6.mod

```
// VARIABLES  DE  DECISION

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

// FUNCION  DE  OPTIMO

minimize
   sum(p in Productos)
            (producto[p].costeInterno*prodInterna[p] +
           producto[p].costeExterno*compraExterna[p])
```

Solución 2: uso de *records* para los datos de los productos

p6.mod

```
// RESTRICCIONES

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;
};
```

Planificación de la producción 2

Solución 2: uso de *records* para los datos de los productos

Optimal Solution with Objective Value: 372.0000

prodInterna[p1] = 40.0000
prodInterna[p2] = 0.0000
prodInterna[p3] = 0.0000

compraExterna[p1] = 60.0000
compraExterna[p2] = 200.0000
compraExterna[p3] = 300.0000

Parámetros formales

p in S

p = parámetro formal
S = conjunto del que toma valores

S = un rango de enteros

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

S = un tipo enumerado

enum Productos…;
float+ coste[Productos] = …;
float+ maxCoste = (*p in Productos*) coste[p];

Parámetros formales

S = un conjunto finito

```
enum Ciudades…;
struct Conexion {
        Ciudades origen;
        Ciudades destino;
};

{Conexion} conexiones = …;
float+ coste[conexiones] = …;
float+ maxCoste  max ( r  in  conexiones) coste[r];
```

Parámetros formales

p in S : condición

```
forall ( i in 1..8)
        forall ( j in 1..8 : i < j )
                reina[i] <> reina[j];
```

Combinación de parámetros

p, q in S : condición

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

que es equivalente a:

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

Parámetros formales

p in S, q in T : condición ó p in S & q in T : condició

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

que son equivalentes equivalentes entre sí y a:

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

Parámetros formales

ordered p, q in S equivalente a: p, q in S : p < q

Las siguientes expresiones con parámetros formales son equivalentes:

```
forall ( i  in  1..8)
        forall ( j  in  1..8 : i < j )
                reina[i] <> reina[j];

forall (i, j  in  1..8 : i < j )
        reina[i] <> reina[j];

forall (i  in  1..8 & j  in  1..8 : i < j )
        reina[i] <> reina[j];

forall (i  in  1..8 , j  in  1..8 : i < j )
        reina[i] <> reina[j];

forall (ordered i, j  in  1..8 )
        reina[i] <> reina[j];
```

Parámetros formales

Parámetros estructurados

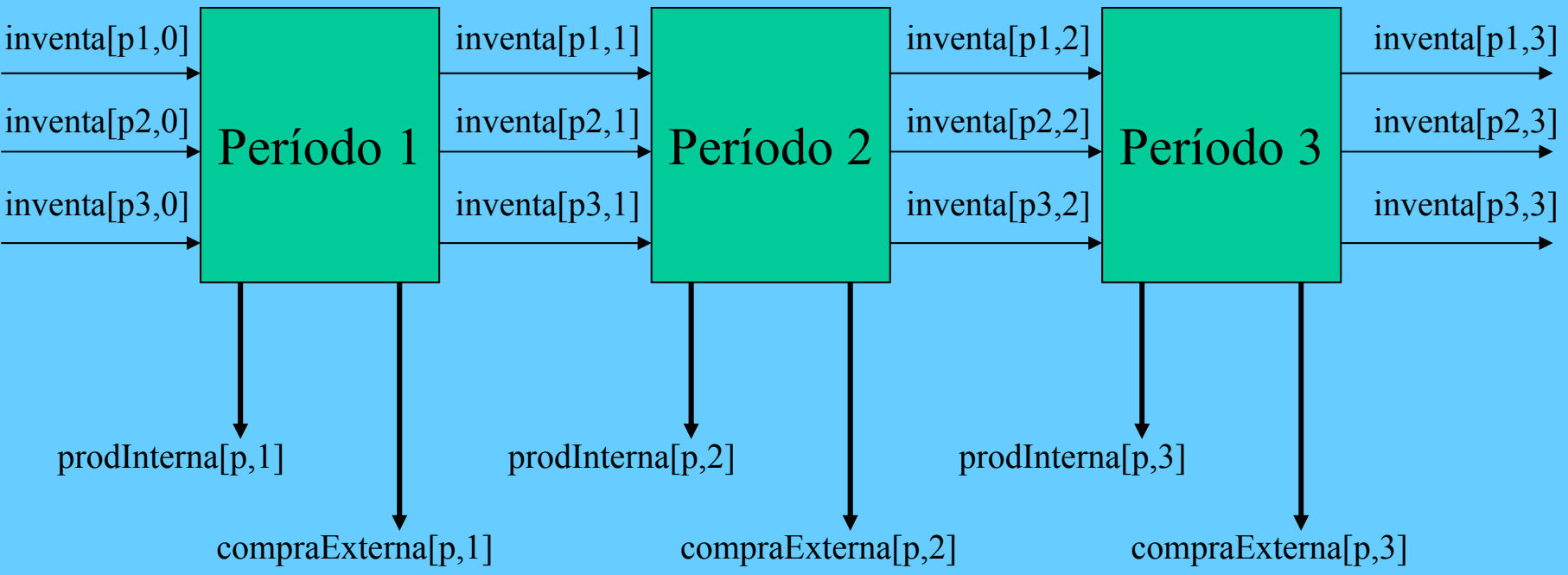
Si *p* es un parámetro estructurado *<p1, p2, …>* y se utiliza la expresión *p in S*, cu
haya que hacer referencia a un campo de *p* habrá que utilizar la expresión *p. cam*
En cambio si utilizamos la expresión *<p1, p2, …> in S*, podemos utilizar directamen
el nombre del parámetro componente.

```
Enum Tareas…;
Struct Precedencia {
        Tareas primera;
        Tareas segunda;
};
{Precedencia} Prec = …;
int duracion[Tareas] = …;
var int start[Tareas] in 0..maxTiempo;

solve {
        forall ( p in Prec)
                start[p.segunda] >= start[p.primera] + duracion[p.priemera];
};
```

Las dos últimas líneas se pueden escribir de la siguiente manera:

```
        forall ( <b, a> in Prec)
                start[a] >= start[b] + duracion[b];
```

Producción multi-período



Producción multi-período

mulprod.dat

```
Productos =  { p1, p2, p3 };
MatPrimas = { m1, rm2 };
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 ];
```

Producción multi-período

mulprod.mod

```
// DECLARACION DE DATOS
enum Productos ...;                    // enumeración de los productos
enum MatPrimas ...;                    // enumeración de las materias primas
int nuPeriodos = ...;                  // número de  períodos de producción
range Periodos 1..nuPeriodos;          // rango de períodos de producción

struct Plan {                          // estructura para la presentación de datos
   float+ interno;
   float+ externo;
   float+ inventario;};

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
```

Producción multi-período

mulprod.mod

```
// VARIABLES DE DECISION

// Producción interna de cada producto en cda período
var float+ prodInterna[Productos,Periodos];
// Compra externa de cada producto en cada período
var float+ compraExterna[Productos,Periodos];
// Inventario de cada producto en cada período
var float+ inventa[Productos,0..nuPeriodos];

// 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])
```

Producción multi-período

mulprod.mod

```
// 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]; };

// PRESENTACION
Plan plan[p in Productos, t in Periodos] =
< prodInterna[p,t], compraExterna[p,t], inventa[p,t] >;
display plan;
```

Optimal Solution with Objective Value: 457.0000

plan[p1,1] = <interno:10.0000,externo:0.0000,inventario:0.0000>
plan[p1,2] = <interno:0.0000,externo:100.0000,inventario:0.0000>
plan[p1,3] = <interno:0.0000,externo:50.0000,inventario:0.0000>
plan[p2,1] = <interno:0.0000,externo:20.0000,inventario:0.0000>
plan[p2,2] = <interno:0.0000,externo:200.0000,inventario:0.0000>
plan[p2,3] = <interno:0.0000,externo:100.0000,inventario:0.0000>
plan[p3,1] = <interno:50.0000,externo:0.0000,inventario:0.0000>
plan[p3,2] = <interno:66.6667,externo:33.3333,inventario:0.0000>
plan[p3,3] = <interno:66.6667,externo:33.3333,inventario:0.0000>

Programación entera

Minimizar (Maximizar) $\sum_{j=1}^{n} c_j x_j$

Sujeto a $\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1,2,\ldots,m$

$x_j \geq 0, \quad x_j \in \mathrm{Z} \quad j = 1,2,\ldots,n$

Ejemplo

Maximizar $3x_1 + 2x_2$

Sujeto a

$$x_1 + x_2 \leq 4$$
$$2x_1 + x_2 \leq 5$$
$$-x_1 + 4x_2 \geq 2$$
$$x_1, x_2 \geq 0 \qquad x_1, x_2 \in \mathrm{Z}$$

Programación entera

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 q
tiene una determinada capacidad para cada atributo (peso máxi
volumen máximo, etc.) de manera tal que se maximice el valor de
objetos seleccionados.

Programación entera

Ejemplo (modelo)

mochila..mod

```
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[a];
```

Programación entera

mochila..mod

```
var 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];
```

Programación entera

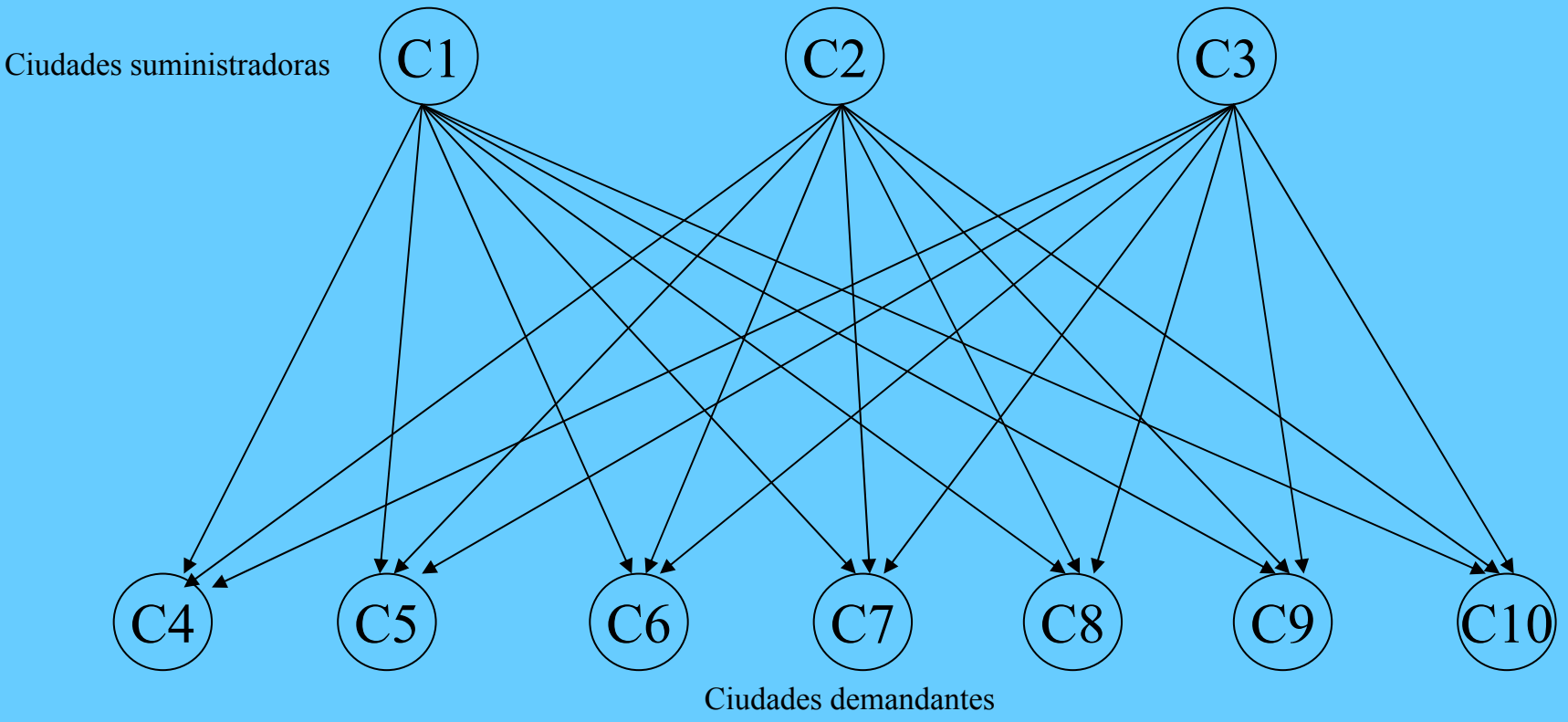
mochila..dat

```
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]
                    ];
```

Problema propuesto 1 (programación lineal)

Transporte de productos entre ciudades a coste mínimo

Productos suministrados: p1 p2 p3
Cada ciudad suministradora dispone de una cantidad fija de cada producto
Cada ciudad demandante requiere una cantidad fija de cada producto
El costo del suministro depende del producto y del trayecto
Existe un límite único en la capacidad de transporte para todos los suministros



Problema propuesto 1 (datos)

transp1.dat

```
Ciudades = { C1 C2 C3 C4  C5  C6  C7  C8  C9  C10 };
Productos   = { p1 p2 p3 };
limite    = 625; // límite de la capacidad de transporte

suministro = #[
        <p1 C1>: 400
        <p2 C1>: 800
        <p3 C1>: 200
        <p1 C2>: 700
        <p2 C2>: 1600
        <p3 C2>: 300
        <p1 C3>: 800
        <p2 C3>: 1800
        <p3 C3>: 300
        ]#;

demanda = #[
        <p1 C4>: 300
        <p2 C4>: 500
        <p3 C4>: 100
        <p1 C5>: 300
        <p2 C5>: 750
        <p3 C5>: 100
        <p1 C6>: 100
        <p2 C6>: 400
        <p3 C6>: 0
        <p1 C7>: 75
        <p2 C7>: 250
        <p3 C7>: 50
        <p1 C8>: 650
        <p2 C8>: 950
        <p3 C8>: 200
        <p1 C9>: 225
        <p2 C9>: 850
        <p3 C9>: 100
        <p1 C10>: 250
        <p2 C10>: 500
        <p3 C10>: 250
        ]#;
```

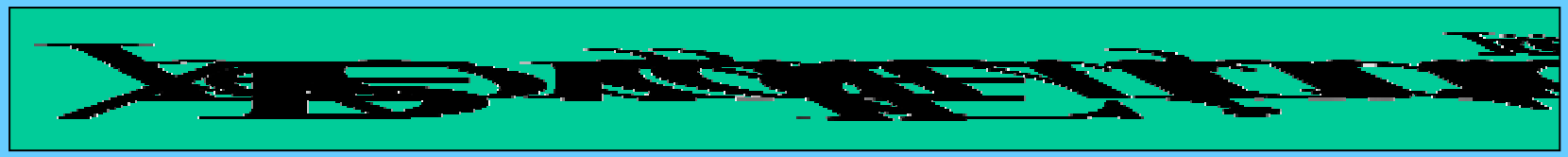
transp1.dat

Problema propuesto 2 (datos)

```
Rutas = {
    <p1 C1 C4>,
    <p1 C1 C5>,
    <p1 C1 C6>,
    <p1 C1 C7>,
    <p1 C1 C8>,
    <p1 C1 C9>,
    <p1 C1 C10>,
    <p1 C2 C4>,
    <p1 C2 C5>,
    <p1 C2 C6>,
    <p1 C2 C7>,
    <p1 C2 C8>,
    <p1 C2 C9>,
    <p1 C2 C10>,
    <p1 C3 C4>,
    <p1 C3 C5>,
    <p1 C3 C6>,
    <p1 C3 C7>,
    <p1 C3 C8>,
    <p1 C3 C9>,
    <p1 C3 C10>,
    <p2 C1 C4>,
    <p2 C1 C5>,
    <p2 C1 C6>,
    <p2 C1 C7>,
    <p2 C1 C8>,
    <p2 C1 C9>,
    <p2 C1 C10>,
    <p2 C2 C4>,
    <p2 C2 C5>,
    <p2 C2 C6>,
    <p2 C2 C7>,
    <p2 C2 C8>,
    <p2 C2 C9>,
    <p2 C2 C10>,
    <p2 C3 C4>,
    <p2 C3 C5>,
    <p2 C3 C6>,
    <p2 C3 C7>,
    <p2 C3 C8>,
    <p2 C3 C9>,
    <p2 C3 C10>,
    <p3 C1 C4>,
    <p3 C1 C5>,
    <p3 C1 C6>,
    <p3 C1 C7>,
    <p3 C1 C8>,
    <p3 C1 C9>,
    <p3 C1 C10>,
    <p3 C2 C4>,
    <p3 C2 C5>,
    <p3 C2 C6>,
    <p3 C2 C7>,
    <p3 C2 C8>,
    <p3 C2 C9>,
    <p3 C2 C10>,
    <p3 C3 C4>,
    <p3 C3 C5>,
    <p3 C3 C6>,
    <p3 C3 C7>,
    <p3 C3 C8>,
    <p3 C3 C9>,
    <p3 C3 C10> };

coste = [ 30, 10,   8, 10, 11, 71, 6,  22, 7, 10, 7, 21, 82, 13, 19, 11, 12, 10, 25,   83, 15,
          39, 14, 11, 14, 16, 82, 8,  27, 9, 12, 9, 26, 95, 17, 24, 14, 17, 13, 28,   99, 20,
          41, 15, 12, 16, 17, 86, 8,  29, 9, 13, 9, 28, 99, 18, 26, 14, 17, 13, 31, 104, 20 ];
```

Problema propuesto (planteamiento)

Variables de decisión



Función de óptimo

$$minimizar \sum_{p \in Productos, o,d \in Ciudades} coste_{p,o,d} * transporte_{p,o,d}$$

Problema propuesto (planteamiento)

Restricciones

Para cada producto y cada ciudad, la suma de lo transportado al resto de ciudades debe ser igual a la cantidad de producto suministrado por la ciudad:

$$\forall p \in \mathrm{Pr}\,oductos, o \in Ciudades, \sum_{d \in Ciudades} transporte_{p,o,d} = suministro_{p,o}$$

Para cada producto y cada ciudad, la suma de lo transportado desde el resto de ciudades debe ser igual a la cantidad de producto demanado por la ciudad:

$$\forall p \in \mathrm{Pr}\,oductos, d \in Ciudades, \sum_{o \in Ciudades} transporte_{p,o,d} = demanda_{p,d}$$

La cantidad de producto en cada transporte entre dos ciudades no puede superar el valor limite:

$$\forall o, d \in Ciudades, \sum_{p \in \mathrm{Pr}\,oductos} transporte_{p,o,d} = limite$$

Problema propuesto 2 (programación lineal)

Optimización del beneficio

Materias primas

Crudos	Crudo1	Crudo2	Crudo3
disponibilidad	5000 barriles	5000 barriles	5000 barriles
precio	45$ / barril	35$ / barril	25$ / barril
octano	12	6	8
plomo	0.5	2	3

MEZCLA
maxProduccion = 14000 barriles /día
costProduccion = 4$ / barril

Productos elaborados

	super	regular	diesel
demanda	3000 barriles	2000 barriles	1000 barriles
precio	70$ / barril	60$ / barril	50$ / barril
octano	>= 10	>= 8	>=6
plomo	<= 1	<= 2	<= 1

Problema propuesto 2 (datos)

transp1.dat

```
Gasolinas = { super regular diesel };
Crudos = { Crudo1 Crudo2 Crudo3 };

gas = [
  #<demanda:3000 precio:70 octano:10 plomo:1 >#
  #<demanda:2000 precio:60 octano:8 plomo:2 >#
  #<demanda:1000 precio:50 octano:6 plomo:1 >#
    ];

crudo = [
  #<disponibilidad:5000 precio:45 octano:12 plomo:0.5 >#
  #<disponibilidad:5000 precio:35 octano:6  plomo:2 >#
  #<disponibilidad:5000 precio:25 octano:8  plomo:3 >#
    ];

maxProduccion = 14000;
costProd = 4;
```

Problema propuesto 3 (programación entera)

Una empresa debe contratar trabajadores de un conjunto {1, 2, ...,32} para construir un edificio. La construcción implica una serie de tareas t1, t2,...,t15 específicas cuya realización requiere una cualificación. Cada trabajador está cualificado para un subconjunto de tareas y tiene un coste de contratación.

Problema:

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.

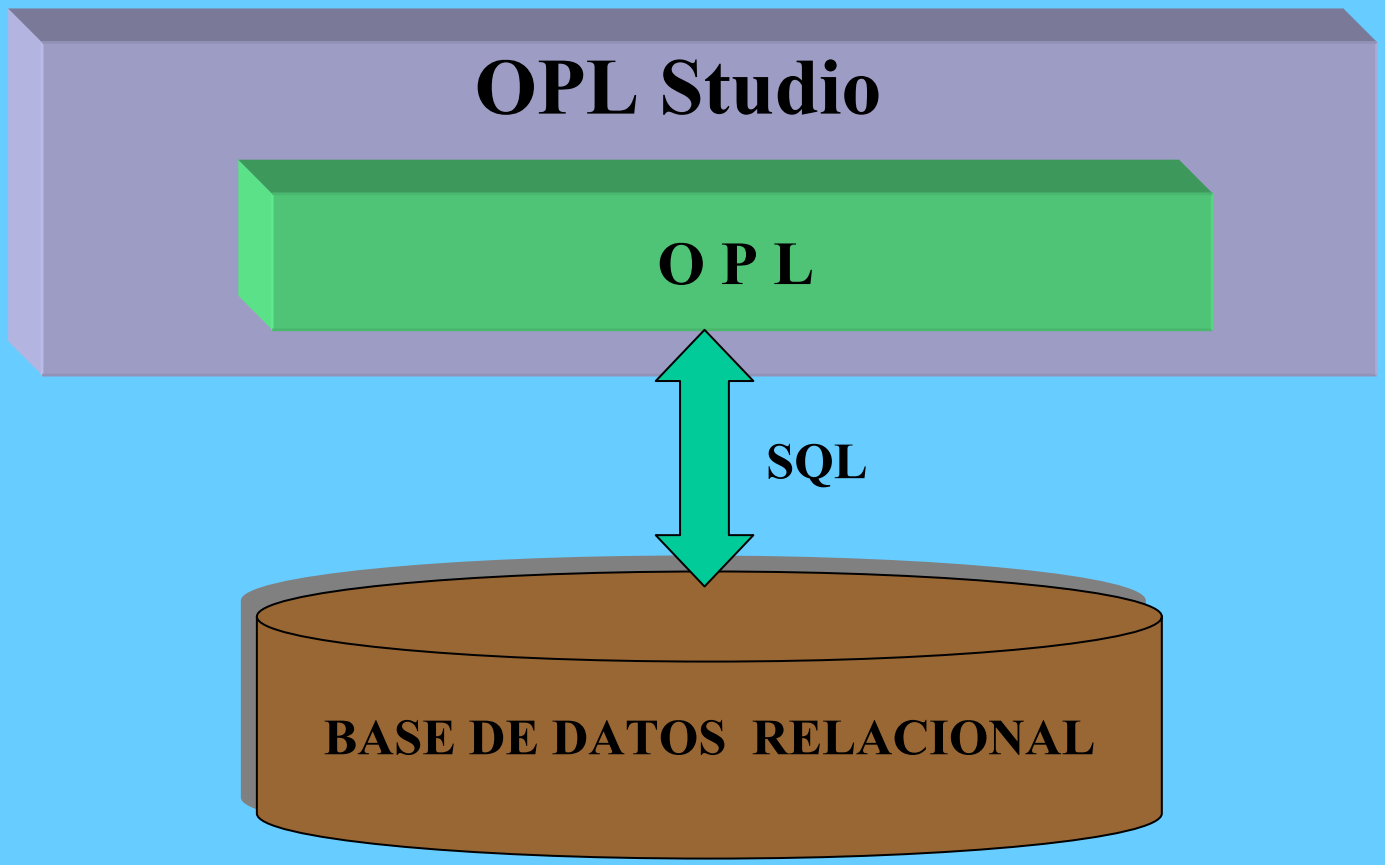
recubrimiento.dat

```
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 }
                           { 2 12 15 19 21 23 27 29 30 31 32 }
                           { 3 10 19 24 26 30 32 }
                           { 4 21 25 28 32 }
                           { 5 11 16 22 23 27 31 }
                           { 6 20 24 26 30 32 }
                           { 7 12 17 25 30 31 }
                           {  8 17 20 22 23  }
                           {  9 13 14  26 29 30 31 }
                           {  10 21 25 31 32 }
                           {  14 15 18 23 24 27 30 32 }
                           {  18 19 22 24 26 29 31 }
                           {  11 20 25 28 30 32 }
                           {  16 19 23 31 }
                           {  9 18 26 28 31 32 }                    ];
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 ];
```

Conexión de OPL con BDs

En esta sesión trataremos los siguientes temas:

- Establecimiento de una conexión con una BD relacional desde OPL Studio
- Lectura de relaciones de la BD
- Creación de nuevas relaciones (tablas)
- Escritura sobre relaciones de la BD


OPL Studio
O P L
SQL
BASE DE DATOS RELACIONAL

BDs conectables a OPL

OPL Studio 2.1 puede conectarse con las siguientes BDs:

- Sobre Windows NT, Windows 95/98
  - ODBC (Open Data Base Connectivity) 3.0
  - Oracle V7.3, V8
  - Sybase 11.0
  - Microsoft SQL Server 6.5
- Sobre UNIX
  - Oracle V7.3, V8

Ejemplo: *produc*

produc.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 ];
```

BD correspondiente a *produc.dat*

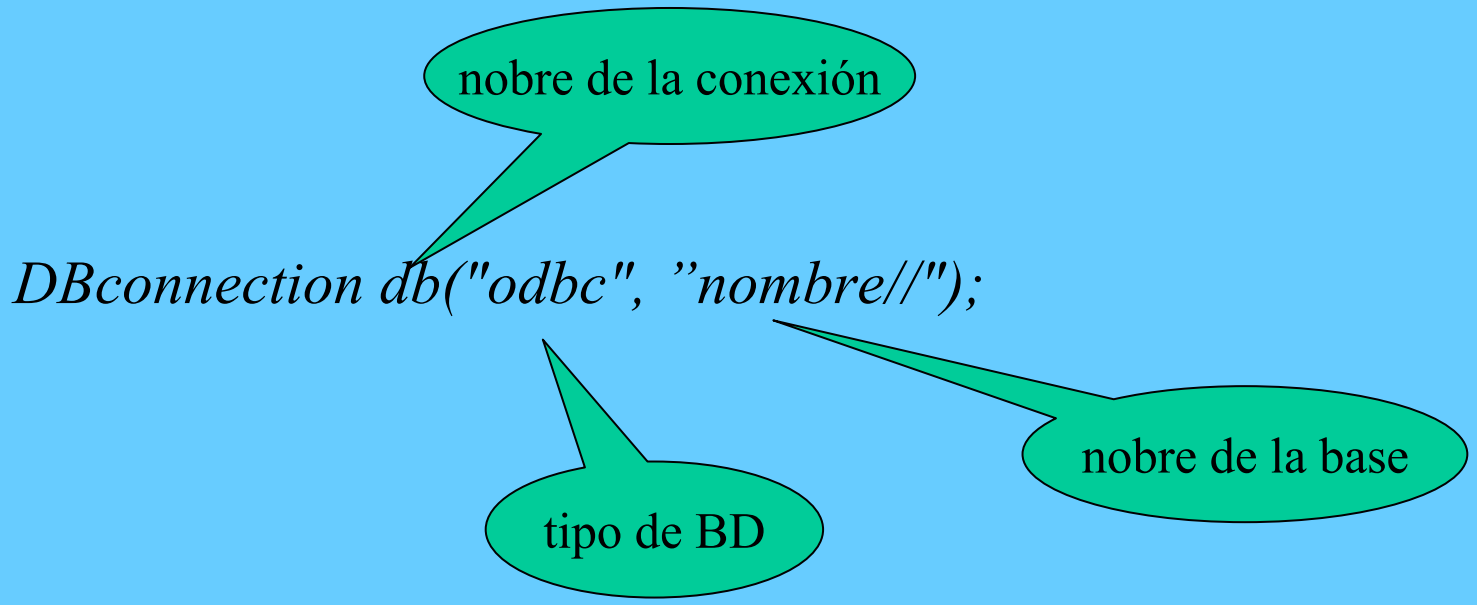
consum	produ	matp	consu
	p1	m1	0,5
	p1	m2	0,2
	p2	m1	0,4
	p2	m2	0,4
	p3	m1	0,3
	p3	m2	0,6

dispon	matp	disp
	m1	20
	m2	40

product	produ	demand	costint	costext
	p1	100	0,6	0,8
	p2	200	0,8	0,9
	p3	300	0,3	0,4

Conexión con ODBC

Sentencia de conexión:



Ejemplo:

DBconnection db("odbc", "base//");

Lectura de relaciones de una BD

Relación (tabla) a leer

consum	produ	matp	consu
	p1	m1	0,5
	p1	m2	0,2
	p2	m1	0,4
	p2	m2	0,4
	p3	m1	0,3
	p3	m2	0,6

Estructura de los elementos

```
struct r_consum {
   string  produ;
   string  matp;
   float   consu;
};
```

Conjunto soporte de la relación

```
{r_consum} t_consum from DBread(db,
    "SELECT DISTINCT produ, matp,consu
       FROM consum");
```

Transformación en array

Estructura del índice

```
struct r_iconsu {
  string produ;
  string matp;
};
```

Conjunto de índices

```
{r_iconsu} iconsu = {<produ,matp>
      |<produ,matp,consu> in t_consum};
```

Array índexado

```
float+ consumo[iconsu];
initialize
   forall ( d  in t_consum)
      consumo[<d.produ, d.matp>] = d.consu;
```

Lectura de la relación *dispon*

Estructura de los elementos

Relación (tabla)

dispon	matp	disp
	m1	20
	m2	40

```
struct r_dispon {
  string  matp;
  float  disp;
};

{r_dispon} t_dispon from DBread(db,
   "SELECT DISTINCT matp,disp
      FROM dispon");

float+ disponibilidad[MatPrimas];
initialize
   forall ( d  in t_dispon)
      disponibilidad[d.matp] = d.disp;
```

Conjunto de lectura

Array *disponibilidad*

Lectura de la relación *product*

product	produ	demand	costint	costext
	p1	100	0,6	0,8
	p2	200	0,8	0,9
	p3	300	0,3	0,4

Relación (tabla)

```
struct r_product {
  string produ;
  float  demand;
  float costint;
  float costext;
};
```

Estructura de los elementos

Conjunto de lectura

```
{r_product} t_product from DBread(db,
   "SELECT DISTINCT produ, demand, costint, costext
      FROM product");
```

Definición e inicialización de arrays

```
float+ demanda[Productos];
initialize
   forall ( d  in t_product)
      demanda[d.produ] = d.demand;

float+ costeInterno[Productos];
initialize
   forall ( d  in t_product)
      costeInterno[d.produ] = d.costint;

float+ costeExterno[Productos];
initialize
   forall ( d  in t_product)
      costeExterno[d.produ] = d.costext;
```

Array *demanda*

Array *costeInterno*

Array *costeExterno*

Creación y eliminación de relaciones

```
struct r_Resultados {
  string produ;
  float proint;
  float proext;
};
      .
      .
      .
DBexecute(db,"create table Resultados(
   producto string,
   produccionInterna float,
   compraExterna float)");


DBexecute(db,"drop table Resultados");
```

Escritura de relaciones

```
{r_Resultados} t_Resultados =
   {<productos,prodInterna[productos],
     compraExterna[productos]>|
      productos in Productos};


DBupdate(db,"insert into Resultados(producto,
                    produccionInterna,compraExterna)
   values(?,?,?)")(t_Resultados);
```

Problema propuesto

- Crear una base de datos *base1* en Access con los datos del archivo *transp1.dat* con la siguiente estructura de tablas:
  - demanda(producto, ciudad, cantidad)
  - suministro(producto, ciudad, cantidad)
  - coste(producto, ciudadO, ciudadD, dolares)

- Modificar el modelo *transp1.mod* de manera que tome los datos de la base de datos *base1* y escriba el resultado en la misma base.

Problema de satisfacción de restricciones

(DOMINIOS FINITOS)

$$X = \{ X_1 , X_2 , \dots , X_n \}$$

conjunto de variables que toma valores de los respectivos dominios finitos del conjunto

$$D = \{ D(X_1),\ D(X_2) , \dots , D(X_n) \}$$

Una **asignación** a las variables $X_1 , X_2, \dots , X_n$ es una *n*-tupla de valores

$$(d_1, d_2, \dots , d_n) \text{ con } d_i \in D(X_i) \quad i = 1,...n.$$

Una **restricción** R(V) con $V \subseteq X$, es un subconjunto del producto cartesiano de los dominios:

$$R(X_1, X_2, \dots , X_m) \subseteq D(X_1) \ x\ D(X_2) \ x \dots x\ D(X_m).$$

Una asignación $(d_1, d_2, \dots , d_m)$ **satisface** la restricción R si:

$$R(d_1 , d_2, \dots , d_m) \in R(X_1 , X_2, \dots , X_m)$$

Una **restricción es satisfacible** si existe al menos una asignación que la satisface

Problema de satisfacción de restricciones

(DOMINIOS FINITOS)

Ejemplo

Variables: X, Y, Z

Dominios: D(X) = {1, 2, 5}, D(Y) = {1, 2, 3, 4, 5}, D(Z) = {1, 2, 5, 7}

Restricción: R(X, Y, Z) = $X < Y \wedge Y < Z$

Asignaciones: (1, 1, 1) insatisfacible

(1, 2, 5) satisfacible

Problema de satisfacción de restricciones (representación)

PSR

$D(X) = [1..9]$
$D(Y) = [1..9]$
$D(Z) = [1..9]$
$D(T) = [1..9]$
$D(W) = [1..9]$

$X > Y + Z + 6$
$Y < T + Z$
$W \neq Y$
$X = W + T$
$X < Z$

PSR en OPL

```
var int X in 1..9;
var int Y in 1..9;
var int Z in 1..9;
var int T in 1..9;
var int W in 1..9;

solve {
  X > Y + Z + 6;
  Y < T + Z;
  W <> Y;
  X = W + T;
  X <> Z;
};
```

Soluciones del PSR

Solution [1]
X = 9
Y = 1
Z = 1
T = 1
W = 8

Solution [2]
X = 9
Y = 1
Z = 1
T = 2
W = 7

Solution [3]
X = 9
Y = 1
Z = 1
T = 3
W = 6

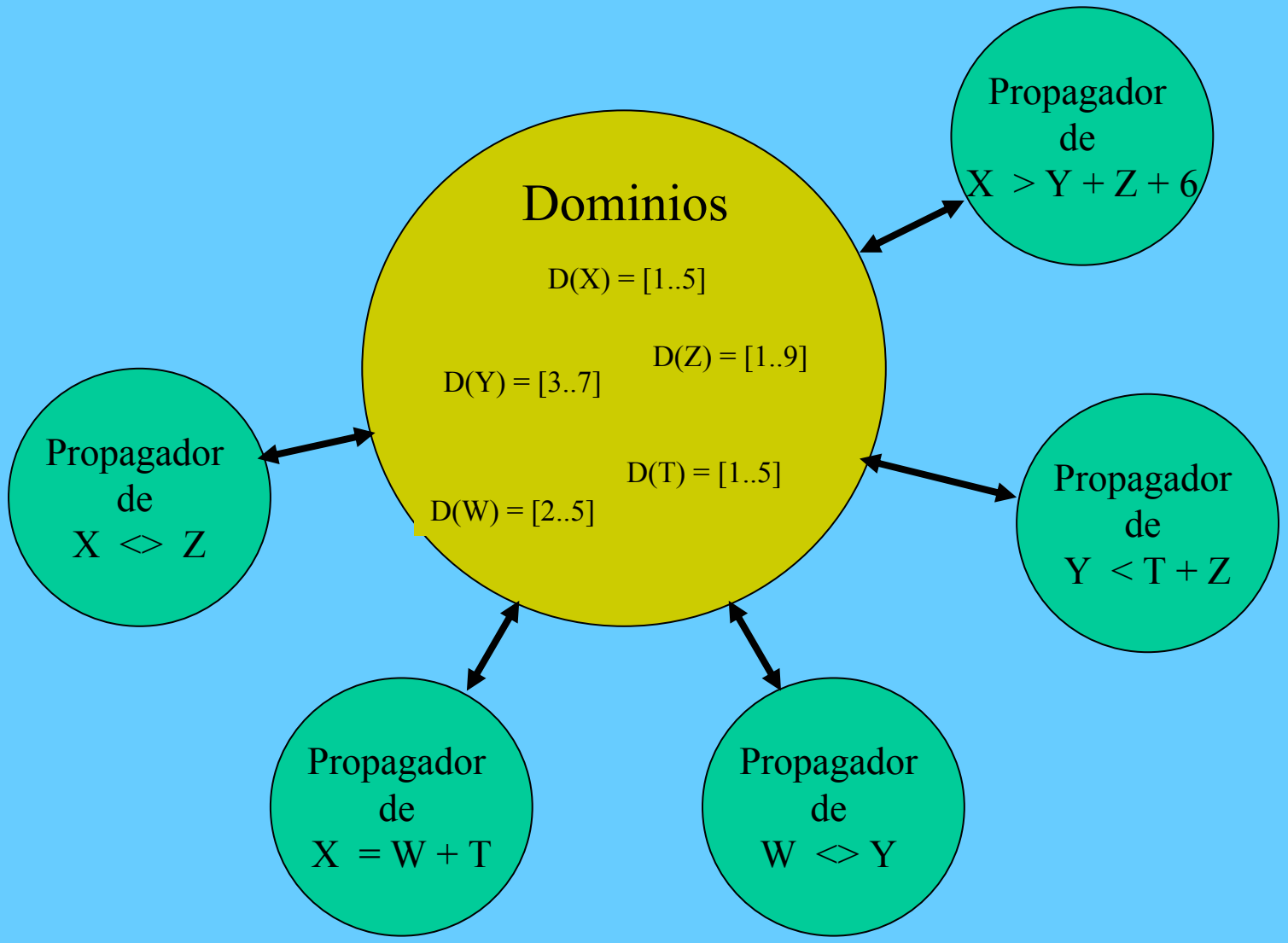
Solution [4]
X = 9
Y = 1
Z = 1
T = 4
W = 5

Solution [5]
X = 9
Y = 1
Z = 1
T = 5
W = 4

Solution [6]
X = 9
Y = 1
Z = 1
T = 6
W = 3

Solution [7]
X = 9
Y = 1
Z = 1
T = 7
W = 2

Arquitectura del resolutor de restricciones de dominios finitos



Arco consistencia

- Una restricción primitiva *r* es **arco_consistente** con dominio *D* si |*vars(r)*| 2 o *vars(r) = {x,y}* y para cada *d* in *D(x)* existe *e* en *D(y)* tal que $\{x \mapsto d, y \mapsto e\}$ es una solución de r, y análogamente para *y*

- Un PSR es arco consistente si todas sus restricciones primitivas son arco_consitentes

Consistencia de límites

- Una restricción primitiva *r* es **límite_consistente** con dominio *D* si para cada variable *x* en *vars(r)* existen números reales *d1, ..., dk* para el resto de variables *x1, ..., xk* tal que $\{x \mapsto \min(D, x), x1 \mapsto d1, \ldots xk \mapsto dk\}$ es una solución de *r,* y análogamente para $\{x \mapsto \max(D, x)\}$

- Un PSR aritmético es límite_consistente si lo son todas sus restricciones primitivas

Ejemplos de consistencia de límites

$$X = 3Y + 5Z$$

$$D(X) = [2..7], D(Y) = [0..2], D(Z) = [-1..2]$$

No es límite_consistente, ya que para *Z=2 (max(D(Z))* no existe solución de *X-3Y=10 en D(X) y D(Y)*

En cambio si es *límite_consistente* el siguiente dominio:

$$D(X) = [2..7], D(Y) = [0..2], D(Z) = [0..1]$$

Obtención de la consistencia de límites

- Dado un dominio actual *D* se modifican los puntos extremos de las variables para que resulte **límite_consistente**

- **los propagadores o reglas de propagación** se encargan de esta tarea

Reglas de propagación (propagadores)

Consideremos la restricción primitiva $X = Y + Z$ que es equivalente a las tres formas

$$X = Y + Z \quad Y = X - Z \quad Z = X - Y$$

Razonando sobre los valores mínimo y máximo obtenemos las siguientes reglas de propagación:

$$X \geq \min(D,Y) + \min(D,Z) \quad X \leq \max(D,Y) + \max(D,Z)$$
$$Y \geq \min(D,X) - \max(D,Z) \quad Y \leq \max(D,X) - \min(D,Z)$$
$$Z \geq \min(D,X) - \max(D,Y) \quad Z \leq \max(D,X) - \min(D,Y)$$

Reglas de propagación (propagadores)

$$X = Y + Z$$

$$D(X) = [4..8], D(Y) = [0..3], D(Z) = [2..2]$$

Las reglas de propagación determinan que:

$$(0 + 2 =)\ \ 2 \leq X \leq 5\ \ (= 3 + 2)$$

$$(4 - 2 =)\ \ 2 \leq Y \leq 6\ \ (= 8 - 2)$$

$$(4 - 3 =)\ \ 1 \leq Z \leq 8\ \ (= 8 - 0)$$

Por lo que los dominios pueden reducirse a:

$$D(X) = [4..5], D(Y) = [2..3], D(Z) = [2..2]$$

Reglas de propagación (propagadores)

$$4W + 3P + 2C \leq 9$$

$$W \leq \frac{9}{4} - \frac{3}{4}\min(D,P) - \frac{2}{4}\min(D,C)$$

$$P \leq \frac{9}{3} - \frac{4}{3}\min(D,W) - \frac{2}{3}\min(D,C)$$

$$C \leq \frac{9}{2} - \frac{4}{2}\min(D,W) - \frac{3}{2}\min(D,P)$$

Si el dominio inicial es:

$$D(W) = [0..9], D(P) = [0..9], D(C) = [0..9]$$

Determinamos que $W \leq \frac{9}{4},\ P \leq \frac{9}{3},\ C \leq \frac{9}{2}$

y el nuevo dominio será:

$$D(W) = [0..2], D(P) = [0..3], D(C) = [0..4]$$

Disecuaciones $Y \neq Z$

Las reglas de las disecuaciones proporcionan una propagación débil. Sólo hay propagación cuando un miembro toma un valor fijo e igual al mínimo o máximo del otro miembro.

$D(Y) = [2..4], D(Z) = [2..3]$ sin propagación

$D(Y) = [2..4], D(Z) = [3..3]$ sin propagación

$D(Y) = [2..4], D(Z) = [2..2]$ propagación $D(Y) = [3..4], D(Z) = [2..2]$

Multiplicación

$$X = Y \times Z$$

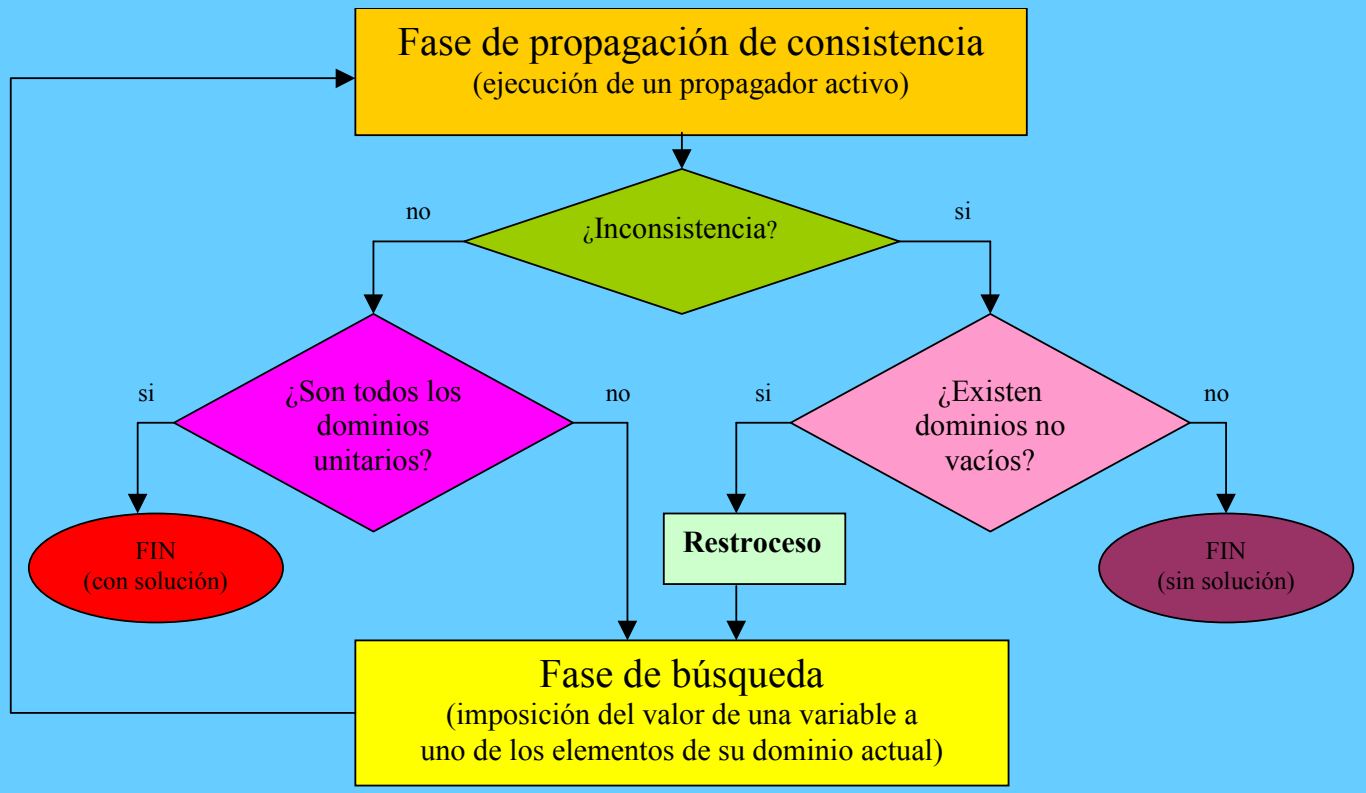
Si todas las variables son positivas el propagador sería:

$$X \geq \min(D,Y) \times \min(D,Z) \qquad X \leq \max(D,Y) \times \max(D,Z)$$
$$Y \geq \min(D,X) / \max(D,Z) \qquad Y \leq \max(D,X) / \min(D,Z)$$
$$Z \geq \min(D,X) / \max(D,Y) \qquad Z \leq \max(D,X) / \min(D,Y)$$

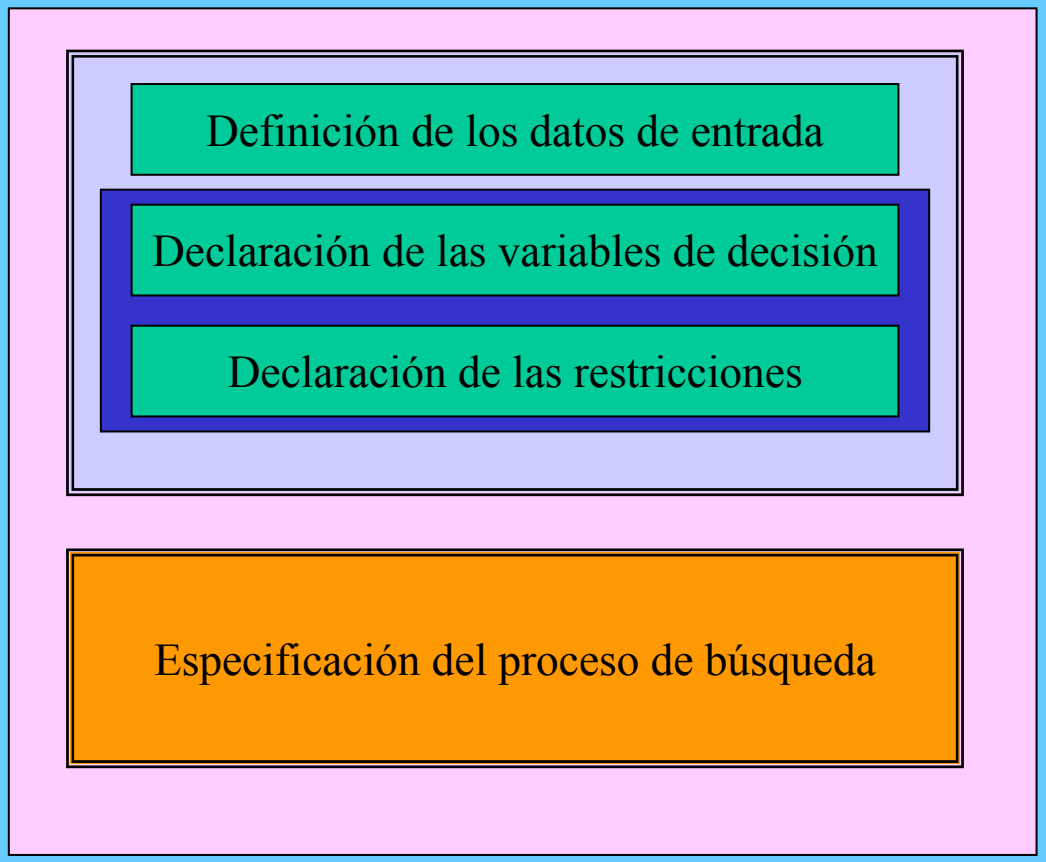
Ejemplo: $D(X) = [4..8], D(Y) = [1..2], D(Z) = [1..3]$

se convierte en: $D(X) = [4..6], D(Y) = [2..2], D(Z) = [2..3]$

Algorítmo general del resolutor



Estructura de un programa de restricciones (DF)



Declaración de restricciones discretas en OPL

- Restricciones básicas
- Combinación lógica de restricciones
- Restricciones de orden superior
- Restricciones con variables en los índices
- Restricciones globales
- Predicados
- Restricciones de planificación (scheduling)

Restricciones básicas

Construidas con:

- datos discretos
- variables discretas
- operadores aritméticos
- funciones

Ejemplo:

forall(f,g in Frecuencias

abs(frec[f] -frec[g]) > 16

Combinación lógica de restricciones

- Utilizan los operadores lógicos tradicionales para conectar diferentes restricciones discretas

Ejemplo:

forall(<a, b> in disyunciones)

inicio[t] >= inicio[a] + duracion[a] \/ inicio[a] >= inicio[b] + duracion[b]

Restricciones de orden superior

- Toda restricción discreta tiene asociada una variable discreta implícita con dominio *[0,1]*
- La variable se instancia a *1* cuando la restricción se hace cierta
- La variable se instancia a *0* cuando la restricción se hace falsa

Ejemplo:

```
forall( i in Rango)
        s[i] = sum(j in Rango) (s[j] = i)
```

Restricciones con variables en los índices

```
enum Mujeres ...;
enum Hombres ...;
.
.
.
var Mujeres esposa [Hombres];
var Hombres   marido [Mujeres];
.
.
.
forall(h in Hombres)  marido [esposa[h] ] = h;


forall(m in Mujeres)  esposa [marido[m] ] = m;
```

Restricciones globales

alldifferent

- tiene como argumento un array de variables discretas
- se satisface cuando todos los elementos del array tienen valores diferentes

Ejemplo:

```
var int a[1..5] in 1..5;
solve{
  alldifferent(a);
  forall(i in 1..4)a[i]<a[i+1]
};
```

Restricciones globales

circuit

- tiene como argumento un array de variables enteras v_1, v_2, v_3, ..., v_n
- el rango de las variables es 1..n
- se satisface cuando la secuencia $(1, v_1)$, (v_1, v_2), ..., (v_{n-1}, v_n), (v_n, v_{n+1}) es un circuito Hamiltoniano

Ejemplo:

```
var int a[1..5] in 1..6;
solve{
  circuit(a);
};
```

Solution [1]	Solution [2]	Solution [3]	Solution [4]	
a[1] = 2	a[1] = 2	a[1] = 2	a[1] = 2	
a[2] = 3	a[2] = 3	a[2] = 4	a[2] = 4	
a[3] = 4	a[3] = 5	a[3] = 1	a[3] = 5	...
a[4] = 5	a[4] = 1	a[4] = 5	a[4] = 3	
a[5] = 1	a[5] = 4	a[5] = 3	a[5] = 1	

Ejemplo 1

Encontrar un número de ocho dígitos que sea un cuadrado perfecto y que siga siendolo cuando se coloca un "1" delante del dígito más significativo.

perfcuad.mod

```
var int n in 10000000..99999999;
var int x in 0..10000;
var int y in 0..20000;
solve{
        n = x * x;
        100000000 + n = y * y;
};
```

Solution [1]	Solution [2]
n = 23765625	n = 56250000
x = 4875	x = 7500
y = 11125	y = 12500

PROGRAMACION CON RESTRICCIONES

Ejemplo 2

Ubicación sin amenaza de *n* reinas en un tablero de ajedrez de *n* * *n*.

reinas.mod

```
int n << "número de reinas: ";
range Dominio 1..n;

var Dominio reinas[Dominio];
solve {
        forall (ordered i, j  in Dominio) {
                reinas[i] <> reinas[j];
                reinas[i] - reinas[j] <> j - i;
                reinas[i] - reinas[j] <> i - j;}
};
```

Solution [1]	Solution [2]	Solution [3]	Solution [4]	Solution [5]	Solution [6]	Solution [7]
reinas[1] = 1	reinas[1] = 1	reinas[1] = 2	reinas[1] = 2	reinas[1] = 3	reinas[1] = 3	reinas[1] = 4
reinas[2] = 3	reinas[2] = 4	reinas[2] = 4	reinas[2] = 5	reinas[2] = 1	reinas[2] = 5	reinas[2] = 1
reinas[3] = 5	reinas[3] = 2	reinas[3] = 1	reinas[3] = 3	reinas[3] = 4	reinas[3] = 2	reinas[3] = 3
reinas[4] = 2	reinas[4] = 5	reinas[4] = 3	reinas[4] = 1	reinas[4] = 2	reinas[4] = 4	reinas[4] = 5
reinas[5] = 4	reinas[5] = 3	reinas[5] = 5	reinas[5] = 4	reinas[5] = 5	reinas[5] = 1	reinas[5] = 2

...

PROGRAMACION CON RESTRICCIONES

Encontrar una secuencia de números $S = (s0, s1, \ldots, sn-1)$ tal que *si* representa el número de ocurrencias de *i* en *S* (series mágicas).

magica.mod

```
int n << "Número de variables: ";
range Rango 0..n-1;
range Dominio 0..n;

var Dominio s[Rango];
solve {
        forall (y in Rango)
                s[y] = sum(j in Rango) (s[j] = y);
};
```

Solution [1]	Solution [1]
s[0] = 2	s[0] = 3
s[1] = 1	s[1] = 2
s[2] = 2	s[2] = 1
s[3] = 0	s[3] = 1
s[4] = 0	s[4] = 0
	s[5] = 0
	s[6] = 0

Ejemplo 4

Matrimonios estables

Emparejar un conjunto de Hombres con conjunto de Mujeres de manera que constituyan matrimonios estables.

Cada hombre ordena el conjunto de las Mujeres en función de sus preferencias, asignándole 1 a la de máxima preferencia, 2 a la siguiente, y así sucesivamente.

Cada mujer ordena el conjunto de los Hombres en función de sus preferencias, asignándole 1 al de máxima preferencia, 2 al siguiente, y así sucesivamente.

Por definición, un matrimonio entre el hombre *h* y la mujer *m* es estable si cumple las dos condiciones siguientes:

1. Si *h* prefiere más a otra mujer *m2* que a su esposa *m*, entonces *m2* prefiere más a su marido que a *h*.
2. Si *m* prefiere más a otro hombre *h2* que a su marido *h*, entonces h2 prefiere más a su esposa que a *m*.

PROGRAMACION CON RESTRICCIONES

Ejemplo 4

matrimonios.mod

```
enum Mujeres ...;
enum Hombres ...;

int preferenciasMujeres[Mujeres,Hombres] = ...;
int preferenciasHombres[Hombres,Mujeres] = ...;

var Mujeres esposa[Hombres];
var Hombres   marido[Mujeres];

solve {
  forall(h in Hombres)  marido[esposa[h]] = h;

  forall(m in Mujeres)  esposa[marido[m]] = m;

  forall(h in Hombres & m2 in Mujeres)
      preferenciasHombres[h,m2] < preferenciasHombres[h,esposa[h]] =>
        preferenciasMujeres[m2,marido[m2]] < preferenciasMujeres[m2,h];

  forall(m in Mujeres & h2 in Hombres)
      preferenciasMujeres[m,h2] < preferenciasMujeres[m,marido[m]] =>
        preferenciasHombres[h2,esposa[h2]] < preferenciasHombres[h2,m];
};
```

PROGRAMACION CON RESTRICCIONES

Ejemplo 4

matrimonios.mod

```
data {
Hombres = {Ricardo, Jaime, Juan, Hugo, Gregorio};
Mujeres = {Elena, Teresa, Luisa, Sarah, Carmen};

preferenciasMujeres =
  #[ Elena:          #[Ricardo:1, Jaime:2, Juan:4, Hugo:3, Gregorio:5 ]#,
     Teresa:         #[Ricardo:3, Jaime:5, Juan:1, Hugo:2, Gregorio:4 ]#,
     Luisa:#[Ricardo:5, Jaime:4, Juan:2, Hugo:1, Gregorio:3 ]#,
     Sarah:          #[Ricardo:1, Jaime:3, Juan:5, Hugo:4, Gregorio:2 ]#,
     Carmen:         #[Ricardo:4, Jaime:2, Juan:3, Hugo:5, Gregorio:1 ]#  ]#;

preferenciasHombres =
  #[ Ricardo:        #[Elena:5, Teresa:1, Luisa:2, Sarah:4, Carmen:3 ]#,
     Jaime  :        #[Elena:4, Teresa:1, Luisa:3, Sarah:2, Carmen:5 ]#,
     Juan   :        #[Elena:5, Teresa:3, Luisa:2, Sarah:4, Carmen:1 ]#,
     Hugo   :        #[Elena:1, Teresa:5, Luisa:4, Sarah:3, Carmen:2 ]#,
     Gregorio   :    #[Elena:4, Teresa:3, Luisa:2, Sarah:1, Carmen:5 ]#  ]#;
      };
```

PROGRAMACION CON RESTRICCIONES

Ejemplo 4

Solution [1]

esposa[Ricardo] = Teresa
esposa[Jaime] = Elena
esposa[Juan] = Carmen
esposa[Hugo] = Luisa
esposa[Gregorio] = Sarah

marido[Elena] = Jaime
marido[Teresa] = Ricardo
marido[Luisa] = Hugo
marido[Sarah] = Gregorio
marido[Carmen] = Juan

Solution [2]

esposa[Ricardo] = Teresa
esposa[Jaime] = Luisa
esposa[Juan] = Carmen
esposa[Hugo] = Elena
esposa[Gregorio] = Sarah

marido[Elena] = Hugo
marido[Teresa] = Ricardo
marido[Luisa] = Jaime
marido[Sarah] = Gregorio
marido[Carmen] = Juan

Solution [3]

esposa[Ricardo] = Sarah
esposa[Jaime] = Elena
esposa[Juan] = Teresa
esposa[Hugo] = Luisa
esposa[Gregorio] = Carmen

marido[Elena] = Jaime
marido[Teresa] = Juan
marido[Luisa] = Hugo
marido[Sarah] = Ricardo
marido[Carmen] = Gregorio

Especificación del proceso de búsqueda en OPL

Definición del árbol de búsqueda

- Implícita (por omisión)
- Explícita

Definición de la estrategia de búsqueda

- Depth-First Search (DFSearch)
- Best-First search
- Limited Discrepancy Search (LDSearch)
- Depth-Bounded Discrepancy search (DDSearch)
- Interleaved Depth-First search (IDFSearch)

Definición explícita del árbol de búsqueda


```
try  <elecciones> endtry;
```

ejemplo:

```
var  int  x in 1..5;
solve  { x <> 1; }
search {
        try   x = 1 | x = 2 | x = 3 | x = 4 | x = 5 endtry;
       };
```

Secuencia de sentencias try

Ejemplo:

```
var  int  x in 0..1;
var  int  y in 0..1;
solve  { x + y <= 1; }
search {
        try  x = 1 | x = 0   endtry;
        try  y = 1 | y = 0   endtry;
       };
```

Definición explícita del árbol de búsqueda

tryall (<parametros formales>) <elecciones> ;

Es la forma iterativa de try

Ejemplo:

tryall(i in 1..5) x = i;

es equivalente a:

try x = 1 | x = 2 | x = 3 | x = 4 | x = 5 endtry

Definición explícita del árbol de búsqueda

Admite la especificación del orden de las alternativas:

Ejemplos:

```
tryall(i in 1..5 ordered by increasing i) x = i;

tryall(i in 1..5 ordered by decreasing i) x = i;
```

Puede incluir una condición:

Ejemplo:

```
tryall(i in 1..10 : i mod 3 = 0) x = i;
```

Definición explícita del árbol de búsqueda

Cuantificador universal

```
forall  (<parametros formales>) <elecciones> ;
```

Ejemplo:

```
var int queen[1..4] in 1..4;
solve {
        forall(ordered i, j in 1..4)
                {
                    Queen[i] <> queen[j];
                    Queen[i] + i <> quee[j] + j;
                    Queen[i] - i <> queen[j] - j;
                }   };
```

```
search {
        forall(i in 1..4)
                tryall(v in 1..4)
                        queen[i] = v;  };
```

```
search {
        try   queen[1] = 1 |queen[1] = 2 |queen[1] = 3 |queen[1] = 4;
        try   queen[2] = 1 |queen[2] = 2 |queen[2] = 3 |queen[2] = 4;
        try   queen[3] = 1 |queen[3] = 2 |queen[3] = 3 |queen[3] = 4;
        try   queen[4] = 1 |queen[4] = 2 |queen[4] = 3 |queen[4] = 4;
    };
```

Definición explícita del árbol de búsqueda

Ordenación dinámica

```
search {
        forall(i in 1..8 ordered by increasinig dsize(queen[i]))
                tryall(v in 1..8)
        queen[i] = v;
};
```

Ordenación por tuplas de expresiones

```
search {
  forall(i in 1..8 ordered by increasinig <dsize(queen[i]), dmin(queen[i])>))
                tryall(v in 1..8)
                        queen[i] = v;
};
```

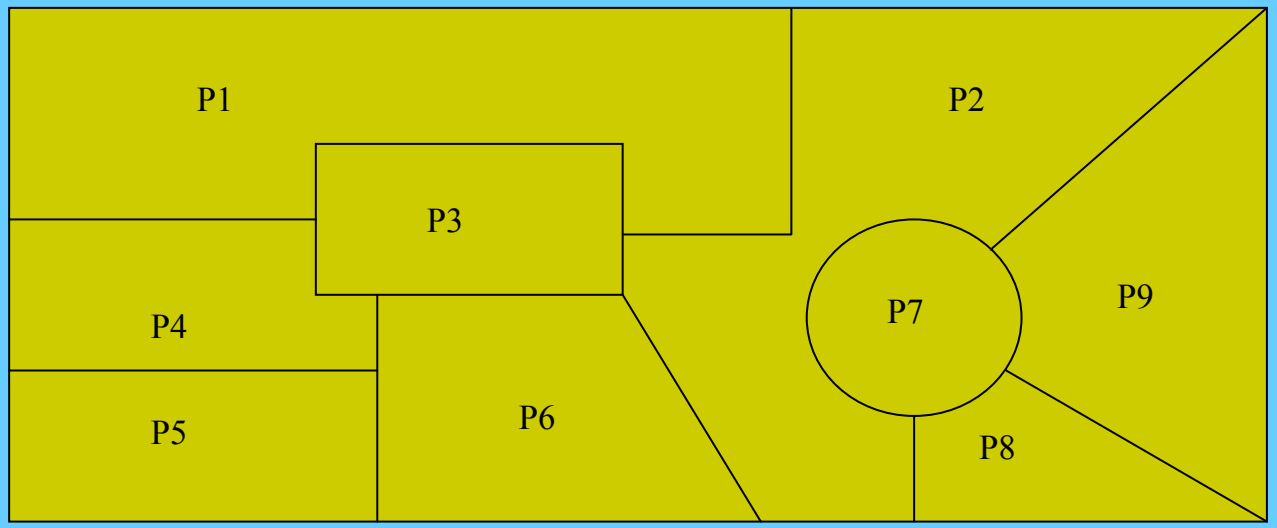
Puede incluir una condición:

```
search {
  forall(i in 1..10 : i mod 2 = 0)
                tryall(v in 1..10)
                        queen[i] = v;
```

Problema propuesto 1

Coloreado de un mapa

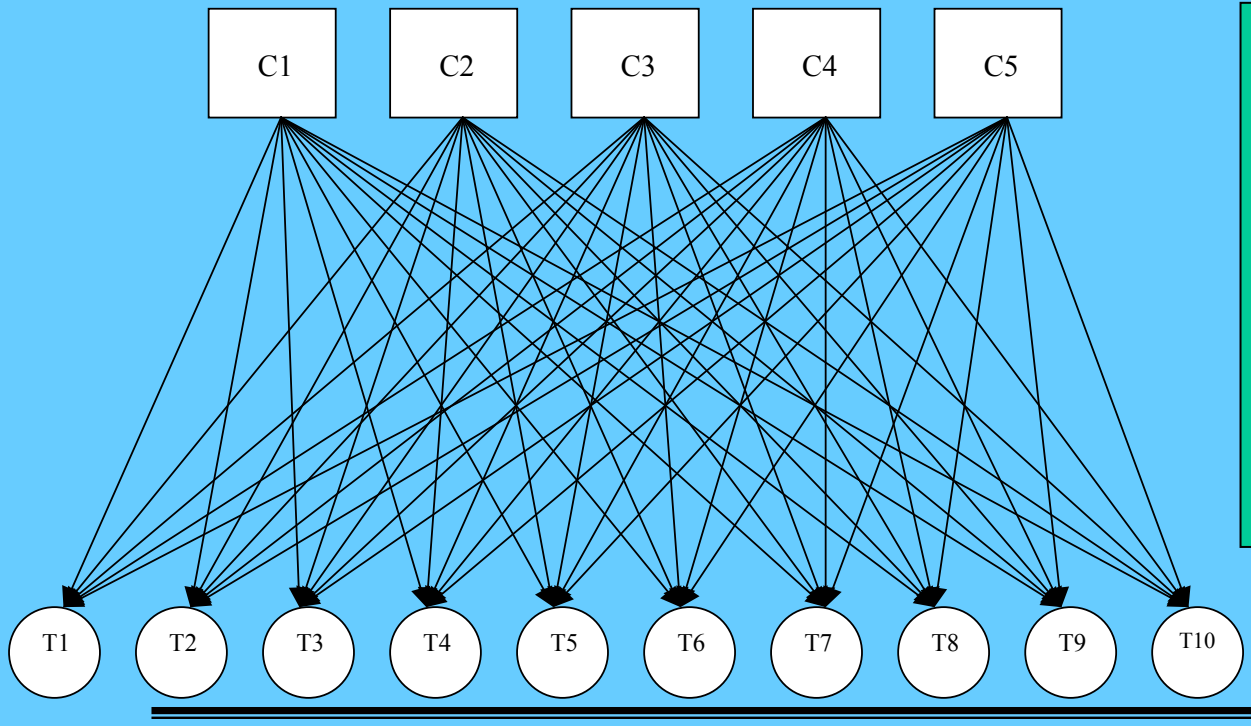
Colorear cada uno de los 9 paises del mapa de la figura con uno de los cuatro siguientes colores: azul, rojo, amarillo y gris, de manera que no tengan el mismo color dos paises que compartan frontera



Problema propuesto 2

Ubicación de almacenes

Ubicar cinco almacenes en cinco ciudades para suministrar a 10 tiendas. Cada almacén tiene un costo fijo de mantenimiento de 30 y una capacidad (número de tiendas) de suministro dada en la tabla. Cada tienda puede ser suministrada por un único almacén, y el costo del suministro depende de la ciudad de ubicación según los datos de la tabla. Se trata de determinar qué almacenes se construyen, en qué ciudades, y a qué tiendas deben suministrar de manera que se minimice el costo total.



ciudades		C1	C2	C3	C4	C5
capacidad		01	04	02	01	03
tiendas	T1	20	24	11	25	30
	T2	28	27	82	83	74
	T3	74	97	71	96	70
	T4	02	55	73	69	61
	T5	46	96	59	83	04
	T6	42	22	29	67	59
	T7	01	05	73	59	56
	T8	10	73	13	43	96
	T9	93	35	63	85	46
	T10	47	65	55	71	95

Planificación temporal y asignación de recursos (scheduli

Proceso de asignar recursos a actividades y ubicar éstas en el tiempo

Planificación temporal pura

Ubica actividades en el tiempo conocidos los recursos demandados po cada actividad.

Ejemplo: problemas de asignación de tareas a máquinas (job-shop)

Asignación de recursos pura

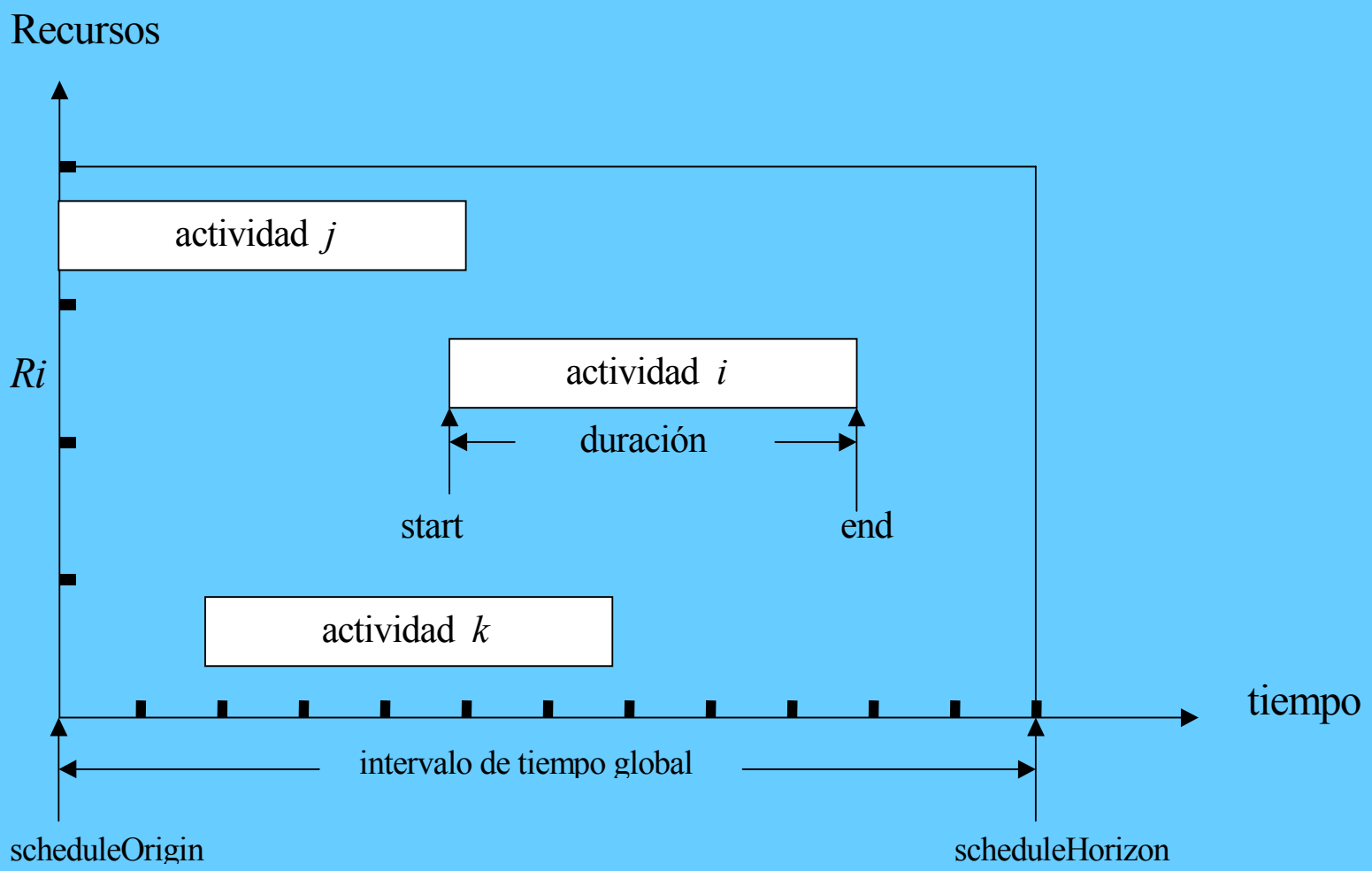
Asigna recursos a actividades conocidos los intervalos de ejecución de actividades y garantizando que en ningún momento se sobrepase la disponibilidad de recursos.

Ejemplo: asignación de personal a los vuelos de una compañía aérea.

Planificación temporal y asignación de recursos en general

Existe libertad para decidir tanto los tiempos de realización de las activi como los recursos que se ponen a disposición de dichas actividades.

Planificación temporal y asignación de recursos



Intervalo de tiempo global

Para la especificar el intervalo de tiempo global se utilizan las dos siguientes declaracio

scheduleOrigin = <Expr> define el extremo inferior y cerrado del intervalo
scheduleHorizon = <Expr> define el extremo superior y abierto del mismo

Ejemplos:

scheduleOrigin = 0;
Declara el instante 0 como el extremo cerrado inicial del intervalo de tiempo global

scheduleHorizon = 365;
Declara el instante 365 como el extremo abierto final del plan temporal.

En conjunto las dos declaraciones anteriores establecen *[0, 365)*
(cerrado a la izquierda y abierto a la derecha)

scheduleHorizon = sum (t in Tareas) duracion[t];
En este caso el extremo final (valor máximo del extremo derecho del intervalo global) se extrae de los datos del problema, es decir, de la duración de las tareas.

Actividades

Una actividad es un objeto que contiene tres elementos de **datos** ligados por una **rest**

Los **datos** son tres variables de dominios finitos:

el tiempo de inicio	*start in scheduleOrigin.. scheduleHorizon*
el tiempo de finalización	*end in scheduleOrigin.. scheduleHorizon*
la duración	*duration*

La duración puede ser:

- un valor constante expresado en la declaración de la actividad
- una variable entera explícitamente declarada
- una variable entera que toma valores del intervalo *[0, scheduleHorizon-scheduleOrigin]*

La **restricción** impone que en todo momento:

tiempo de finalización = tiempo de inicio + duración.

end = start + duration

Planificación temporal y asignación de recursos
(tipos de datos)

Ejemplos de declaración de actividades:

Activity carpinteria(10);
Declara la actividad carpintería de duración constante 10

Var int duracionCarpinteria in 8 .. 10;
Activity carpinteria(duracionCarpinteria);
Declara la actividad carpintería cuya duración es una variable entera con dominio 8 ..

Activity carpinteria;
Declara la actividad carpinteria cuya duración es una variable entera con dominio[0, scheduleHorizon-scheduleOrigin)

Activity tareas[t in 1 .. 10](duration[t]);
Declara un array de 10 actividades cuyas duraciones respectivas son *duration[1],…, duration[10]*

Planificación temporal y asignación de recursos
(tipos de datos)

Actividades interrumpibles

Son actividades que pueden ser interrumpidas y reasumidas durante su duración

Ejemplos:

Activity fontaneria breakable;
Declara la actividad fontanería como interrumpible

Activity tareas[t in 1 .. 10] (duration)[t]) breakable;
Declara un array de 10 actividaes interrumpibles, *tareas[1], tareas[2],* ... ,de duraciones respectivas *duration[1],…, duration[10]*

Activity tareas[t in 1 .. 10] (duration[t]) breakable if t in ConjuntoInterrumpible;
Declara un array de 10 actividades de las que sólo son interrumpibles las que pertenecen al ConjuntoInterrumpible

Referenciación de los elementos de una actividad

Los elementos de una actividad
- el tiempo de inicialización (start)
- el de finalización (end)
- la duración (duration)

pueden ser accedidos como los campos de una estructura:
- *actividad.start*
- *actividad.end*
- *actividad.duration*

Planificación temporal y asignación de recursos
(tipos de datos)

Recursos

Los recursos son utilizados por las actividades en su ejecución. La cantidad de un determinado o determinados recursos que demanda una actividad será impuesto por las restricciones de recursos que estudiaremos posteriormente.

En este apartado nos ocuparemos tan sólo de la declaración de los diferentes tipos de recursos:

- unitarios
- discretos
- reservas
- alternativos

Planificación temporal y asignación de recursos
(tipos de datos)

Recursos unitarios

Son recursos que no puede ser compartido por más de una actividad. Se utilizan para modelar problemas en los que aparecen recursos individuales no divisibles, por ejemplo, en problemas de planificación de tareas en máquinas (job-shop).

Ejemplos:

UnaryResource grua;
Declara el recurso unitario de nombre grúa

UnaryResource maquinas[1 .. 10];
Declara el array de 10 recursos unitarios *maquinas[1],…, maquinas[10]*

Planificación temporal y asignación de recursos
(tipos de datos)

Recursos discretos

Se trata de un recurso disponible en múltiplos de una unidad de manera que todas las unidades son equivalentes e intercambiables desde el punto de vista de la aplicación.

Ejemplos:

DiscreteResource presupuesto(30000);
Declara el recurso presupuesto con una capacidad de 30.000

DiscreteResource rec[t in 1..10](cap[t]);
Declara el array de 10 recursos discretos *rec[1],…,rec[10]* con capacidades respectivas *cap[1],…cap[10]*

DiscreteResource res[1..10](3);
Declara el array de 10 recursos discretos *rec[1],…,rec[10]* todos ellos con capacidad 10

Planificación temporal y asignación de recursos
(tipos de datos)

Reservas

Es un tipo de recurso que no sólo puede ser consumido o demandado por una actividad sino que además el mismo recurso puede ser proporcionado o producido por otra actividad.

Ejemplos:

Reservoir tanque1(1000);
Declara un recurso de reserva con una capacidad máxima de 1.000 y una capacidad inicial de 0

Reservoir tanque2(1000, 100);
Declara un recurso de reserva con una capacidad máxima de 1.000 y una capacidad inicial de 100.

Planificación temporal y asignación de recursos
(tipos de datos)

Recursos alternativos

Los recursos alternativos, a diferencian de los discretos, garantizan que se mantienen a lo largo de la duración de la actividad sin ser intercambiados.

Ejemplos:

UnaryResource horno[1..10];
AlternativeResource s(horno);
Declara que *s* es el conjunto de recursos unitarios *horno[1],…, horno[10]*

Planificación temporal y asignación de recursos
(restricciones)

Restricción de precedencia *(precedes)*

Establece ordenes de precedencia entre las actividades

Ejemplo:

actividad1 precedes actividad2;
Declara que la *actividad1* debe preceder a la *actividad2*.

Sería equivalente a declarar:

actividad2.start >= actividad1.end

aunque la primera produce mejor visualización de los resultados

Planificación temporal y asignación de recursos
(restricciones)

Restricciones sobre recursos unitarios

requires: liga un recurso unitario con una actividad:
sintaxis: Actividad *requires* RecursoUnitario

Los recursos unitarios son interrumpibles, y para especificar las interrupciones existen otras
tres restricciones, una para periódicas y dos para no periódicas.

Interrupción periódica
sintaxis: periodicBreak(RecursoUnitario, Inicio, Duración, Periodicidad)

Interrupciones no periódicas.
La primera expresa los extremos inicial y final de la interrupción del recurso
sintaxis: *break(RecursoUnario, ExtremoInicialInterrupción, ExtremoFinalInterrupción)*

La segunda expresa el extremo inicial de la interrupción y su duración:
sintaxis: *breakOnDuration(RecursoUnario, ExtremoInicialInterrupción, DuraciónInterrupción)*

Planificación temporal y asignación de recursos
(restricciones)

Ejemplos de restricciones sobre recursos unitarios:

UnaryResource grua;
excavacion requires grua;
Especifica que la actividad excavación requiere el recurso unitario grúa durante su ejecu

periodicBreak(grua, 5, 2, 7);
Especifica que el recurso unitario grúa (previamente declarado) tiene una interrupción c
7 días, que la primera interrupción es en el día 5, y que la duración de la interrupción es

break(grua, 10, 12);
Especifica que el recurso unitario grúa tiene una interrupción en el intervalo [10,12]

breakOnDuration(grua, 10, 2);
Otra alternativa para la misma especificación anterior

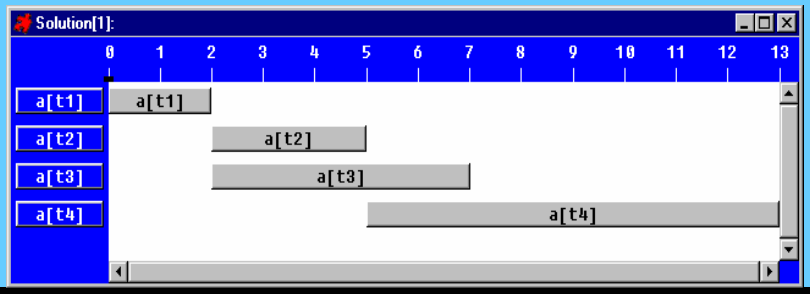
Actividades con precedencias

tarea1.mod

```
enum tarea  {t1, t2, t3, t4};
int duracion[tarea] = [2, 3, 5, 8];
struct precedencia{tarea anterior; tarea posterior;};
{precedencia} precedencias = {<t1, t2>, <t1, t3>, <t2, t4>};

scheduleHorizon = 20;

Activity a[t in tarea](duracion[t]);

minimize
        a[t4].start
subject to
{
        forall(t in precedencias)
                a[t.anterior] precedes a[t.posterior];
};
```

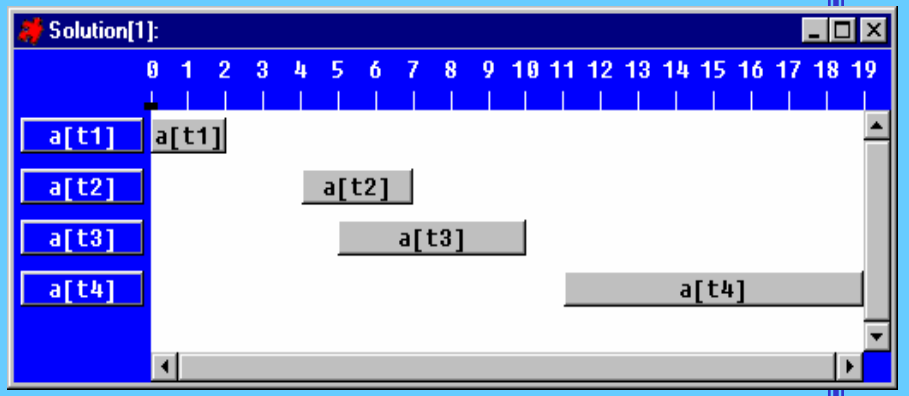
Optimal Solution with Objective Value: 5

a[t1] = [0 -- 2 --> 2]
a[t2] = [2 -- 3 --> 5]
a[t3] = [2 -- 5 --> 7]
a[t4] = [5 -- 8 --> 13]



Actividades con precedencias y distancias


```
enum tarea  {t1, t2, t3, t4};
int duracion[tarea] = [2, 3, 5, 8];
struct precedencia{tarea anterior;
                   tarea posterior;
                   int distancia; };
{precedencia} precedencias = {<t1, t2, 2>, <t1, t3, 3>, <t2, t4, 4>};

scheduleHorizon = 20;

Activity a[t in tarea](duracion[t]);

minimize    a[t4].start
subject to
{
        forall(t in precedencias)
                a[t.anterior].end + t.distancia <= a[t.posterior].start;
};
```



Optimal Solution with Objective Value: 11
a[t1] = [0 -- 2 --> 2]
a[t2] = [4 -- 3 --> 7]
a[t3] = [5 -- 5 --> 10]
a[t4] = [11 -- 8 --> 19]

Actividades con precedencias y recursos unitarios

tarea3.mod

```
enum tarea  {t1, t2, t3, t4};
int duracion[tarea] = [2, 3, 5, 8];
struct precedencia{tarea anterior; tarea posterior;};
{precedencia} precedencias = {<t1, t2>, <t1, t3>, <t2, t4>};
enum recurso {r1, r2};
{tarea} uso[recurso] = [{t1, t4}, {t2, t3}];

scheduleHorizon = 20;
Activity a[t in tarea](duracion[t]);
UnaryResource rec[recurso];

minimize
  a[t4].start

subject to
{
  forall(t in precedencias)
    a[t.anterior] precedes a[t.posterior];

  forall(r in recurso)
    forall(t in uso[r])
      a[t] requires rec[r];
};
```

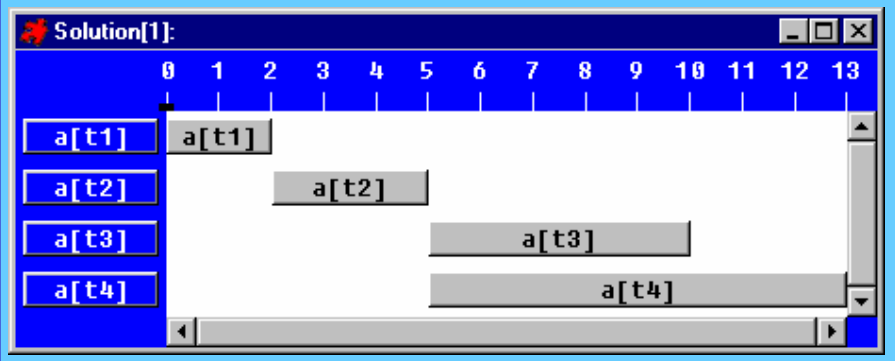
Actividades con precedencias y recursos unitarios (resultados)

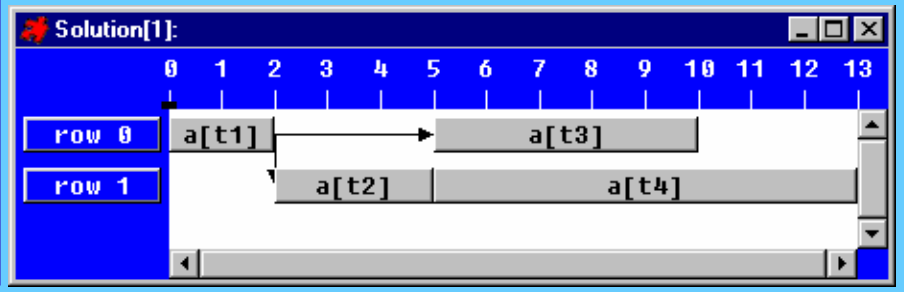
Optimal Solution with Objective Value: 5

a[t1] = [0 -- 2 --> 2]
a[t2] = [2 -- 3 --> 5]
a[t3] = [5 -- 5 --> 10]
a[t4] = [5 -- 8 --> 13]

rec[r1] = Unary Resource
  required by a[t4] over [5,13] in capacity 1
  required by a[t1] over [0,2] in capacity 1

rec[r2] = Unary Resource
  required by a[t3] over [5,10] in capacity 1
  required by a[t2] over [2,5] in capacity 1





Actividades interrumpibles con precedencias y recursos unitarios

tarea4.mod

```
enum tarea  {t1, t2, t3, t4};
int duracion[tarea] = [2, 3, 5, 8];
struct precedencia{tarea anterior; tarea posterior;};
{precedencia} precedencias = {<t1, t2>, <t1, t3>, <t2, t4>};

enum recurso {r1, r2};
{tarea} uso[recurso] = [{t1, t4}, {t2, t3}];

{tarea} tareasNoInterrumpibles = {t1};

scheduleHorizon = 20;

Activity a[t in tarea](duracion[t]) breakable  if t not in tareasNoInterrumpibles;

UnaryResource rec[recurso];
```

Actividades interrumpibles con precedencias y recursos unitarios

tarea4.mod

```
minimize
        a[t4].start
subject to
{
        forall(r in recurso)
                periodicBreak(rec[r], 4, 2, 20); //no funciona break(rec[r], 4,2);

        forall(t in precedencias)
                a[t.anterior] precedes a[t.posterior];

        forall(r in recurso)
                forall(t in uso[r])
                        a[t] requires rec[r];
};
```

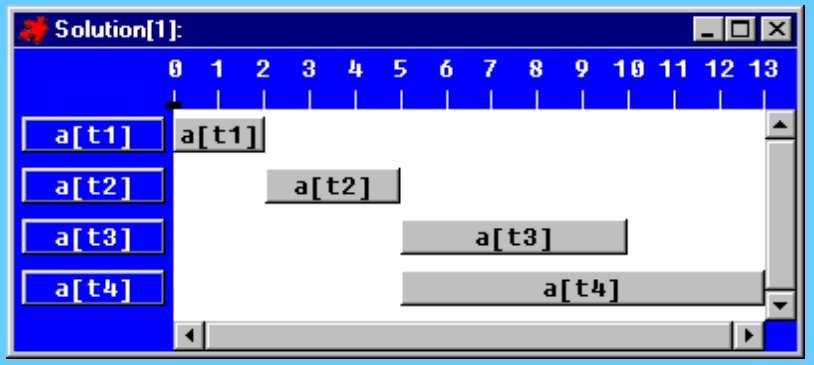
Actividades interrumpibles con precedencias y recursos unitarios
(solución)

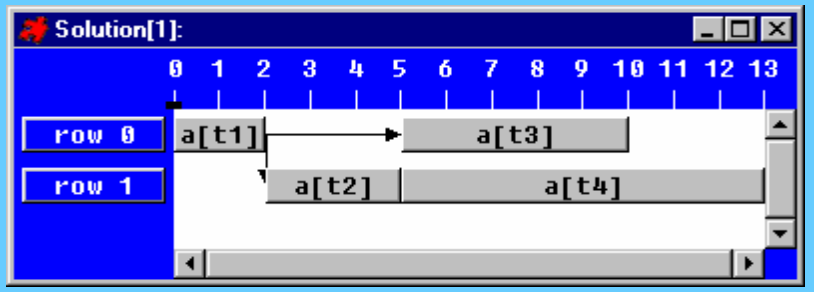
Optimal Solution with Objective Value: 7

a[t1] = [0 -- 2 --> 2]
a[t2] = [2 -- (3) 5 --> 7]
a[t3] = [7 -- (5) 5 --> 12]
a[t4] = [7 -- (8) 8 --> 15]

rec[r1] = Unary Resource
  required by a[t4] over [7,15] in capacity 1
  required by a[t1] over [0,2] in capacity 1

rec[r2] = Unary Resource
  required by a[t3] over [7,12] in capacity 1
  required by a[t2] over [2,7] in capacity 1





Planificación temporal y asignación de recursos
(restricciones)

Restricciones sobre recursos discretos

Uso del recurso
La expresión general para asignar una Cantidad de Recurso discreto a una Actividad es la siguiente:

Actividad requires(Cantidad) Recurso

El recurso es utilizado por la Actividad, y tan pronto como finaliza,la Cantidad de Recurso es devuelta, quedando disponible para otra actividad.

Consumo del recurso
Si lo que se quiere es modelar un recurso que se consume cuando es utilizado por u actividad debe utilizarse la siguiente expresión general:

Actividad consumes (Cantidad) Recurso

Planificación temporal y asignación de recursos
(restricciones)

Grado de propagación

Los algoritmos de propagación internos de OPL aseguran que en ningún instante de tie
la cantidad asignada de recurso sobrepasa la capacidad total declarada. Estos algoritm
pueden utilizar tres grados de propagación (poda de valores de los dominios):

Nivel por defecto

Nivel disjunctive

DiscreteResource Recurso(Capacidad) using disjunctive

Nivel edgeFinder

DiscreteResource Recurso(Capacidad) using edgeFinder

El nivel de propagación adecuada depende fuertemente de la aplicación.

Planificación temporal y asignación de recursos
(restricciones)

Variación de la capacidad

La capacidad de un recurso discreto puede variar con el tiempo, generalizando el conce
de interrupción de los recursos unitarios.

1. Se puede imponer un **límite superior al uso del recurso** con la siguiente expresión:

CapacityMax(RecursoDiscreto, ExtremoInicial, ExtremoFinal, CapacidadMáxima)

Con ella se especifica que la capacidad de *RecursoDiscreto* requerida por las actividade
el intervalo *[ExtremoInicial, ExtremoFinal)* es como máximo *CapacidadMáxima.*

2. Se puede imponer un **límite inferior al uso del recurso** con la siguiente expresión:

CapacityMin (RecursoDiscreto, ExtremoInicial, ExtremoFinal, CapacidadMínima)

Con ella se especifica que la capacidad de *RecursoDiscreto* requerida por las actividade
en el intervalo *[ExtremoInicial, ExtremoFinal)* es como mínimo *capacidadMínima.*

Planificación temporal y asignación de recursos

(restricciones)

Ejemplos de restricciones sobre recursos discretos

cimientacion requires(2) martillos;
Especifica que la actividad cimentación requiere dos martillos.

DiscreteResource martillos(3) using disjuntive
Especifica que el nivel de propagación del recurso discreto martillo con capacidad máxima 3 es *disjunctive*

DiscreteResource martillos(3) using edgeFinder
Especifica que el nivel de propagación del recurso discreto martillo con capacidad máxima 3 es *edgeFinder*

a rquires(2) r;
Especifica que la actividad a requiere el recurso r en una cantidad de 2 unidades, y quedará disponible para otras actividades tan pronto finalice la actividad a.

a consumes(2) r;
Especifica que la actividad a requiere el recurso r en una cantidad de 2 unidades, y esta cantidad queda definitivamente consumida para el resto de las actividades.

Planificación temporal y asignación de recursos
(restricciones)

Restricciones sobre recursos reservas

Actividad requires(Cantidad) Recurso

Actividad consumes (Cantidad) Recurso

Actividad provides(Cantidad) Recurso

Actividad produces (Cantidad) Recurso

Planificación temporal y asignación de recursos
(restricciones)

Ejemplos de restricciones sobre recursos reservas

a requires(2) fontaneria;
Especifica que la actividad a requiere dos unidades del recurso fontanería para su ejec
y que estas unidades quedarán a disposición de las restantes actividades tan pronto co
finalice a actividad a.

a consumes(2) fontaneria;
Especifica que la actividad a rquiere dos unidades del recurso fontanería para su ejecu
y que estas unidades quedarán consumidas para las restantes actividades.

b provides(2) fontaneria;
Especifica que la actividad b proporciona dos unidades del recurso fontanería durante
su ejecución.

b produces(2) fontaneria;
Especifica que la actividad b produce dos unidades del recurso fontanería desde su
instante de finalización hasta el final de la planificación *(scheduleHorizon).*

Planificación de la construcción de una casa en tiempo mínimo

Actividades	duración	precedencias
albañilería	7	{}
carpintería	3	{albañilería}
tejados	1	{carpintería}
fontanería	8	{albañilería}
techos	3	{albañilería}
ventanas	1	{tejados}
fachada	2	{fontanería, tejados}
jardín	1	{tejados, fontanería}
pintura	2	{techos}
traslado	1	{ventanas, fachada, jardín, pintura}

Actividad	**esprecede a**
albañilería	{carpintería, fontanería, techos}
carpintería	{tejados}
tejados	{ventanas, fachada, jardín}
fontanería	{fachada, jardín}
techos	{pintura}
ventanas	{traslado}
fachada	{traslado}
jardín	{traslado}
pintura	{traslado}

Planificación de la construcción de una casa

casa.mod

```
enum Tarea ...;
struct Precedencia {
  Tarea anterior;
  Tarea posterior;
};
int duracion[Tarea] = ...;
{Precedencia} precedencias = ...;
int maxDuracion = sum(t in Tarea) duracion[t];
scheduleHorizon = maxDuracion;
Activity a[t in Tarea](duracion[t]);
minimize
  a[traslado].start
subject to {
    forall( t in precedencias )
        a[t.anterior] precedes a[t.posterior];
};
```

Planificación de la construcción de una casa

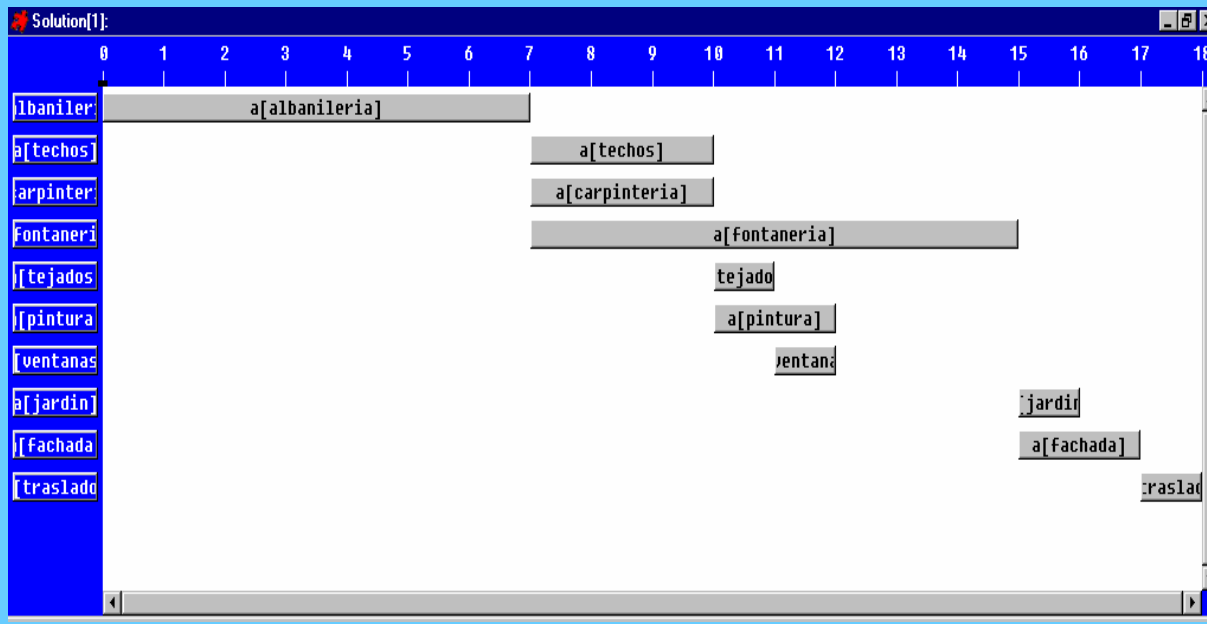
casa.dat

```
Tarea =  {albanileria, carpinteria, tejados, fontaneria, techos, ventanas, fachada, jardin, pintura, traslado};

duracion =          #[          albanileria:        7,
                                carpinteria:        3,
                                tejados:            1,
                                fontaneria:         8,
                                techos:             3,
                                ventanas:           1,
                                fachada:            2,
                                jardin:             1,
                                pintura:            2,
                                traslado:           1           ]#;
precedencias =      {           <albanileria,carpinteria>,
                                <albanileria,fontaneria>,
                                <albanileria,techos>,
                                <carpinteria,tejados>,
                                <tejados,ventanas>,
                                <tejados,fachada>,
                                <tejados,jardin>,
                                <fontaneria,fachada>,
                                <fontaneria,jardin>,
                                <techos,pintura>,
                                <ventanas,traslado>,
                                <fachada,traslado>,
                                <jardin,traslado>,
                                <pintura,traslado>          };
```

Planificación de la construcción de una casa (resuñtados)

Optimal Solution with Objective Value: 17

a[albanileria] = [0 -- 7 --> 7]
a[carpinteria] = [7 -- 3 --> 10]
a[tejados] = [10 -- 1 --> 11]
a[fontaneria] = [7 -- 8 --> 15]
a[techos] = [7 -- 3 --> 10]
a[ventanas] = [11 -- 1 --> 12]
a[fachada] = [15 -- 2 --> 17]
a[jardin] = [15 -- 1 --> 16]
a[pintura] = [10 -- 2 --> 12]
a[traslado] = [17 -- 1 --> 18]



Planificación de la construcción de una casa con presupuesto

Al problema anterior se añade la siguiente restricción de presupuesto:

Cada tarea cuesta una cantidad proporcional a su duración de 1.000 por día, es decir, el presupuesto global será de 29.000 (1.000 x suma de las duraciones de todas las tareas), de la cual sólo 20.000 está disponible al comienzo del proyecto, de la cantidad restantes 9.000 se dispondrá 15 días más tarde.

Planificación de la construcción de una casa con presupuesto

casa1.mod

```
enum Tarea ...;
struct Precedencia { Tarea anterior; Tarea posterior;};
int duracion[Tarea] = ...;
{Precedencia} precedencias = ...;
int maxDuracion = sum(t in Tarea) duracion[t];
scheduleHorizon = maxDuracion;
Activity a[t in Tarea](duracion[t]);
DiscreteResource presupuesto(29000);
minimize
  a[traslado].start
subject to {
  forall( t in precedencias )
     a[t.anterior] precedes a[t.posterior];

  forall(t in Tarea)
    a[t] requires(1000*duracion[t]) presupuesto;

capacityMax(presupuesto,0,15,20000);
};
```

Problema propuesto 1

Planificación de trabajos en máquinas

Planificar un número de trabajos (6) sobre un conjunto de máquinas (6) teniendo en cuenta que cada trabajo se compone de una secuencia de tareas (6) y que cada tarea requiere una máquina

```
int numMaquinas = ...;
range Maquinas 1..numMaquinas;
int numTrabajos = ...;
range Trabajos 1..numTrabajos;
int numTareas = ...;
range Tareas 1..numTareas;

Maquinas uso[Trabajos,Tareas] = ...;



int+ duracion[Trabajos,Tareas] = ...;
```

trabajo.dat

```
numMaquinas = 6;
numTrabajos = 6;
numTareas = 6;

uso = [
      [ 3, 1, 2, 4, 6, 5],
      [ 2, 3, 5, 6, 1, 4],
      [ 3, 4, 6, 1, 2, 5],
      [ 2, 1, 3, 4, 5, 6],
      [ 3, 2, 5, 6, 1, 4],
      [ 2, 4, 6, 1, 5, 3]
            ];
duracion = [
      [ 1,  3,  6,  7,  3,  6],
      [ 8,  5, 10, 10, 10,  4],
      [ 5,  4,  8,  9,  1,  7],
      [ 5,  5,  5,  3,  8,  9],
      [ 9,  3,  5,  4,  3,  1],
      [ 3,  3,  9, 10,  4,  1]
            ];
```

Problema propuesto 2

Carga de un barco

Planificar en el menor tiempo posible las 34 actividades que requiere la carga de un barco sujetas a las relaciones de precedencia recogidas en *conjuntoPrecedencias,* las duraciones recogidas en *duracion* y sabiendo que cada una de ellas requiere un único recurso discreto de capacidad 8 en las cantidades recogidas en *demanda*

cbarco.dat

```
capacidad = 8;
numTareas = 34;

duracion = [3, 4, 4, 6, 5, 2, 3, 4, 3, 2, 3, 2, 1, 5, 2, 3, 2,
            2, 1, 1, 1, 2, 4, 5, 2, 1, 1, 2, 1, 3, 2, 1, 2, 2 ];

demanda = [4, 4, 3, 4, 5, 5, 4, 3, 4, 8, 4, 5, 4, 3, 3, 3, 6,
           7, 4, 4, 4, 4, 7, 8, 8, 3, 3, 6, 8, 3, 3, 3, 3, 3 ];

conjuntoPrecedencias = {
< 1,  2>, < 1,  4>, < 2,  3>, < 3,  5>, < 3,  7>, < 4,  5>,
< 5,  6>, < 6,  8>, < 7,  8>, < 8,  9>, < 9, 10>, < 9, 14>,
<10, 11>, <10, 12>,<11, 13>, <12, 13>, <13, 15>, <13, 16>,
<14, 15>, <15, 18>, <16, 17>, <17, 18>, <18, 19>, <18, 20>,
<18, 21>, <19, 23>, <20, 23>, <21, 22>, <22, 23>, <23, 24>,
<24, 25>, <25, 26>, <25, 30>, <25, 31>, <25, 32>, <26, 27>,
<27, 28>, <28, 29>, <30, 28>, <31, 28>, <32, 33>, <33, 34>
};
```

```
int capacidad = ...;
int numTareas = ...;
range Tareas 1..numTareas;
int duracion[Tareas] = ...;
int demanda[Tareas] = ...;

struct Precedencias {
  int anterior;
  int posterior;
};

{Precedencias} conjuntoPrecedencias = ...;
```

OPL Script

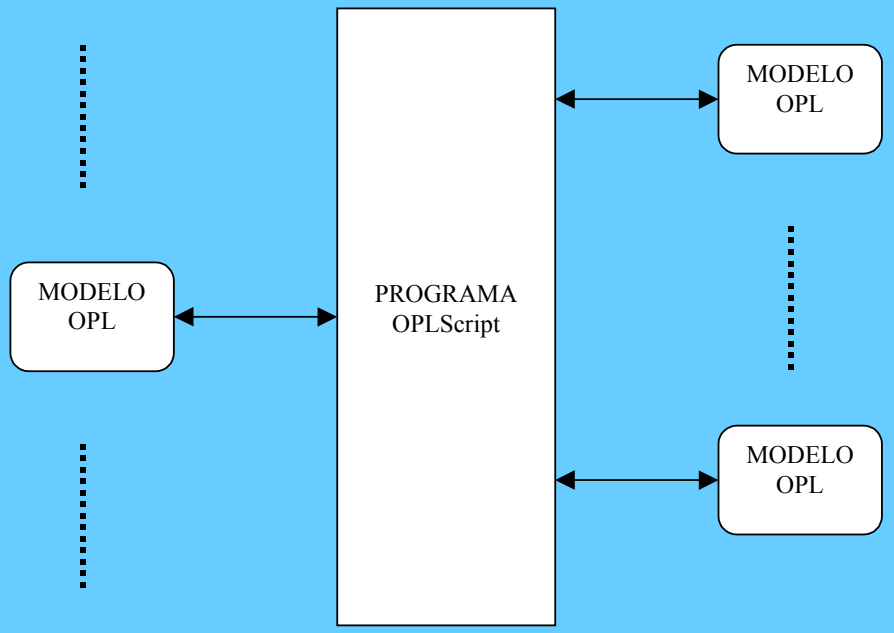
OPLScript es un lenguaje que permite controlar la ejecución de uno o varios modelos OPL contemplandolos como objetos, es decir, permitiendo:

1) acceder a sus estructuras de datos

a) leyendo y/o modificando valores constantes del modelo

b) leyendo valores de instanciación de las variables de decisión

2) ejecutar métodos predefinidos sobre los modelos OPL



OPLSCript es un lenguaje con estilo imperativo que ofrece las mismas estructuras de datos
de alto nivel que OPL, ampliadas con otras nuevas que cubren necesidades específicas de estas aplicaciones (por ejemplo, arrays abiertos)

OPL Script

Ejemplo 1:
Programa OPLSscript que ejecuta el modelo de ubicación de n reinas (Sreinas.mod) para n = 5,6,7,8 y visualiza los resultados por consola

Declaración del modelo OPLScript *m* inicializado con el modelo OPL *Sreinas.mod* en modo de edición (editMode), es decir, permitiendo modificar sus datos (en este caso el parámetro *n* de *Sreinas.mod*)

Sreinas.osc

```
Model m("Sreinas.mod") editMode;

forall(i in 5..8){
  m.n := i;
  if m.nextSolution() then
  cout << " n-reinas con n = " << m.n << endl;
  forall(i in 1..m.n){
    cout << "reinas[" << i << "] = ";
    cout << m.reinas[i] << endl;
  }
  cout << endl;
  m.reset();
}
```

Método predefinido que devuelve el valor *1* si el modelo tiene una nueva solución, *0* en caso contrario

Acceso al dato *n* del objeto *m*

Actualiza la constante *n* de *m* con el valor de *i*

Método predefinido que reinicializa el modelo *m*

OPL Script

Sreinas.mod

```
int n = ...;
range Dominio 1..n;

var Dominio reinas[Dominio];
solve {
   forall (ordered i, j  in Dominio) {
      reinas[i] <> reinas[j];
      reinas[i] - reinas[j] <> j - i;
      reinas[i] - reinas[j] <> i - j;
   }
};

data {
n = 5;
};
```

OPL Script

Ejemplo 2:
Programa OPLSscript que ejecuta el modelo de producción multiperíodo (Smulprod.mod) para valores de la disponibilidad de la materia prima m1= 20 (valor original en el modelo), 21, 22, 23 y visualiza los resultados respectivos de la función de óptimo y el número de iteraciones realizadas.

Smulprod.osc

```
Model produce("Smulprod.mod","Smulprod.dat") editMode;
import enum componentes produce.MatPrimas;
int+ capm1 := produce.disponibilidad[m1];

forall(i in 1..4) {
  produce.disponibilidad[m1] := capm1;
  produce.solve();
  cout << "Funcion objetivo: " << produce.objectiveValue() << endl;
  cout << "Iteraciones: " << produce.getNumberOfIterations() << endl;
  produce.reset();
  capm1 := capm1 + 1;
}
```

Importa el tipo enumerado MatPrimas del modelo produce y lo renombra a componentes para utilizar en el script

OPL Script

Smulprod.mod

```
// DECLARACION DE DATOS

enum Productos ...;
enum MatPrimas ...;
int nuPeriodos = ...;
range Periodos 1..nuPeriodos;

struct Plan {float+ interno; float+ externo;  float+ inventario;};

float+ consumo[MatPrimas,Productos] = ...;
int+ disponibilidad[MatPrimas] = ...;
float+ demanda[Productos,Periodos] = ...;
float+ costInterno[Productos] = ...;
float+ costExterno[Productos]  = ...;
float+ inventario[Productos]  = ...;
float+ costInventario[Productos]  = ...;

// VARIABLES DE DECISION

var float+ prodInterna[Productos,Periodos];
var float+ compraExterna[Productos,Periodos];
var float+ inventa[Productos,0..nuPeriodos];
```

OPL Script

Smulprod.mod

```
// 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];
};
```

OPL Script

Smulprod.dat

```
Productos =  { p1, p2, p3 };
MatPrimas = { m1, rm2 };
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 ];
```

OPL Script

Declaraciones de datos

I) OPLScript dispone de las mismas estructuras de datos que OPL con las siguientes diferencias sintácticas:

- las declaraciones utilizan el signo de asignación := en lugar de = (ejemplo, **int a := 1;**)

- Las declaraciones de conjuntos utilizan la palabra reservada *setof* en lugar de llaves (ejemplo, **setof(int) s := {1,2,3};**)

II) OPLScript dispone de las siguientes nuevas estructuras:

- Arrays abiertos

- archivos (files)

- bases (basis)

- declaraciones de importación

OPL Script

Metodos predefinidos de los modelos

Método	Semántica
solve()	resuelve el modelo
nextSolution()	devuelve la siguiente solución del modelo
reset()	pasa el modelo a su estado inicial
restore()	restaura la última solución encontrada por el modelo
objectiveValue()	devuelve el valor de óptimo de un modelo de optimización
setBasis(Basis)	especifica una base inicial en modelos de programación lineal
setFailLimit(Int)	especifica un límite del número de fallos del modelo
setOrLimit(Int)	especifica un límite del número de puntos de elección del modelo
setTimeLimit(Int)	especifica un límite del tiempo de ejecución (CPU) del modelo
getNumberOfFails()	devuelve el número de fallos habidos en la ejecución del modelo
getNumberOfVariables()	devuelve el número de variables del modelo
getNumberOfConstraints()	devuelve el número de restricciones del modelo
getNumberOfChoicePoints()	devuelve el número de puntos de elección del modelo
getNumberOfIterations()	devuelve el número de iteraciones en la ejecución de un modelo LP
getTime()	devuelve el tiempo en segundos de CPU
getBasis()	devuelve la base de la solución de un modelo LP

OPL Script

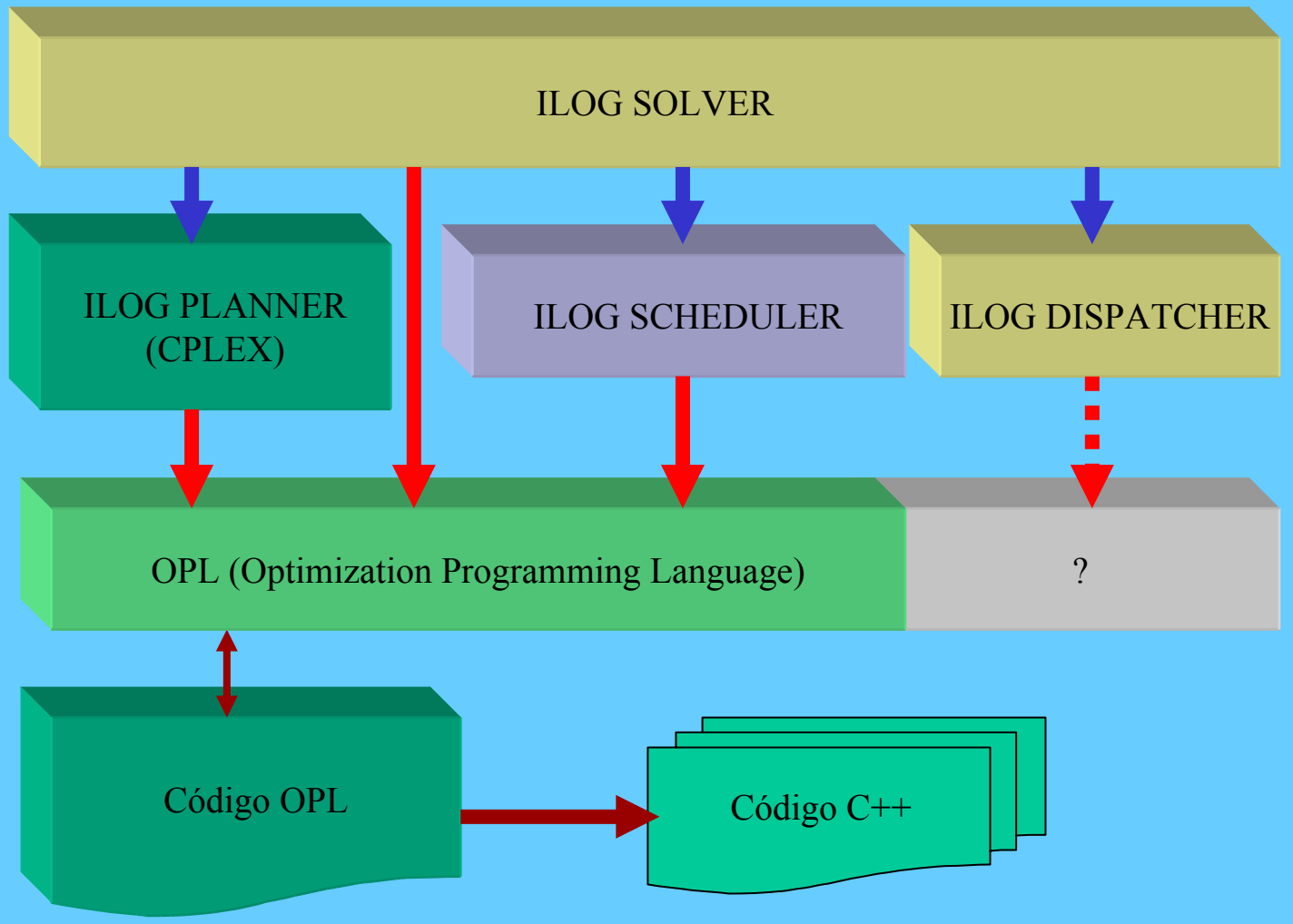
Problema propuesto

Escribir un programa en OPLScript que ejecute alguno de los modelos codificados en prácticas anteriores para diferentes valores de alguno de sus parámetros y visualice los resultados por consola

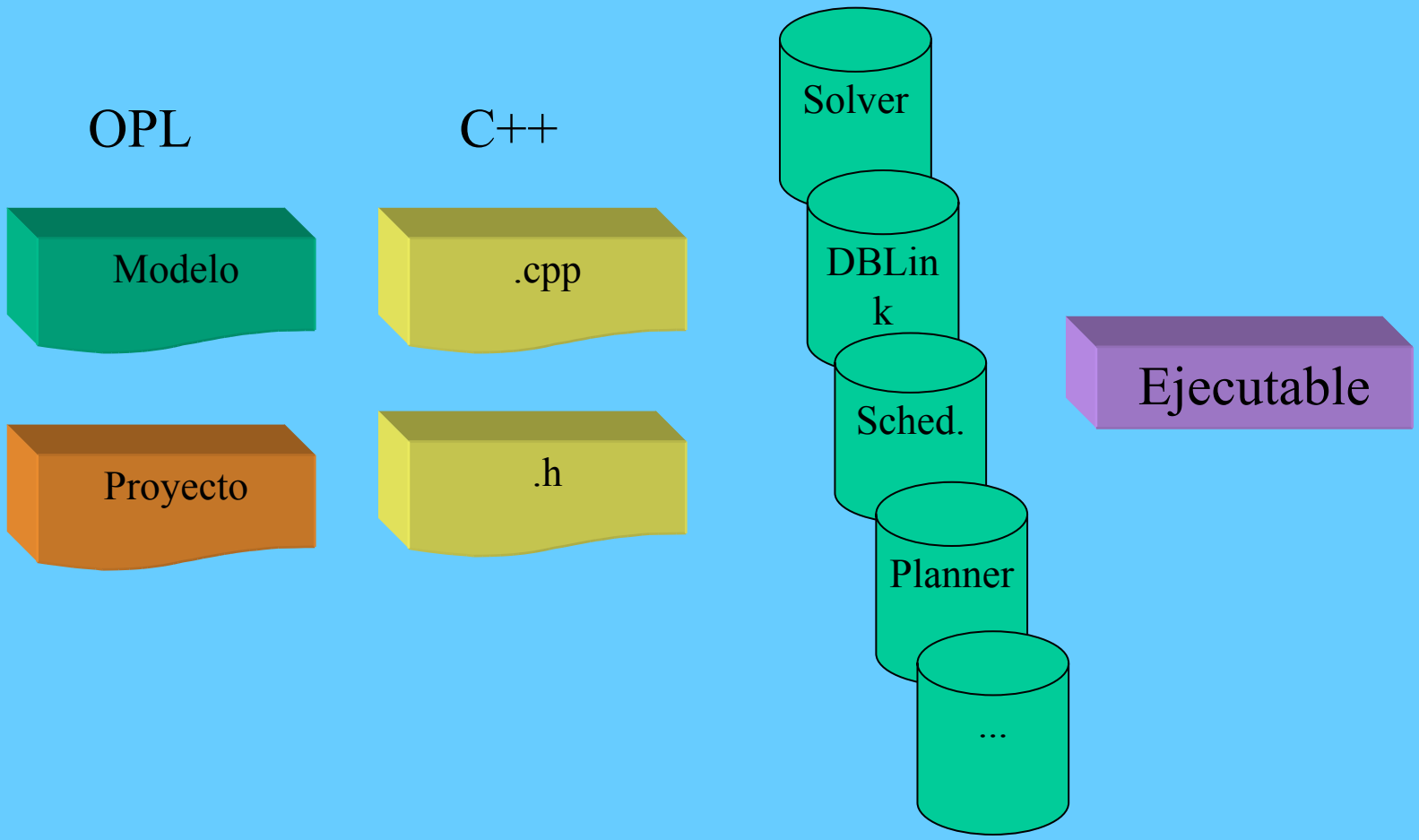
Generación de código C++

1. Visión general
2. Procedimiento de generación de código
3. Elementos en el entorno de programación orientada a objetos
4. Aplicaciones

Visión general



Procedimiento de generación de código



Programa OPL (Proyecto/Modelo)

```
enum Productos ...;
enum MatPrimas ...;

struct DatosProductos {
  float+ demanda;
  float+ costoInterno;
  float+ costoExterno;
  float+ consumo[MatPrimas];
};

DatosProductos product[Productos] = ...;
float+ disponibilidad[MatPrimas] = ...;

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

minimize
  sum(p in Productos)
(product[p].costoInterno*prodInterna[p] +
                product[p].costoExterno*compraExterna[p])
subject to {
  forall(r in MatPrimas)
    sum(p in Productos) product[p].consumo[r] *
prodInterna[p] <= disponibilidad[r];
```

Programa C++ (.cpp/.h)

```
int main(int argc,char* argv[])
{
  int i = 1;
  try {
    IloSolver_productos solver;
    if (solver.nextSolution()) {
      cout << "solución[" << i << "]" << endl;
      i++;
      cout << endl;
    }
    solver.end();
  }
```

```
class IloEnumValue_Productos: public
IloEnumValue {
public:
  IloEnumValue_Productos() {}
  IloEnumValue_Productos(IloSolver* s) :
IloEnumValue(s) {}
  IloEnumValue_Productos(const
IloEnumValue_Productos& o) :
IloEnumValue(o) {}
  IlcInt operator==(IloEnumValue_Productos
```

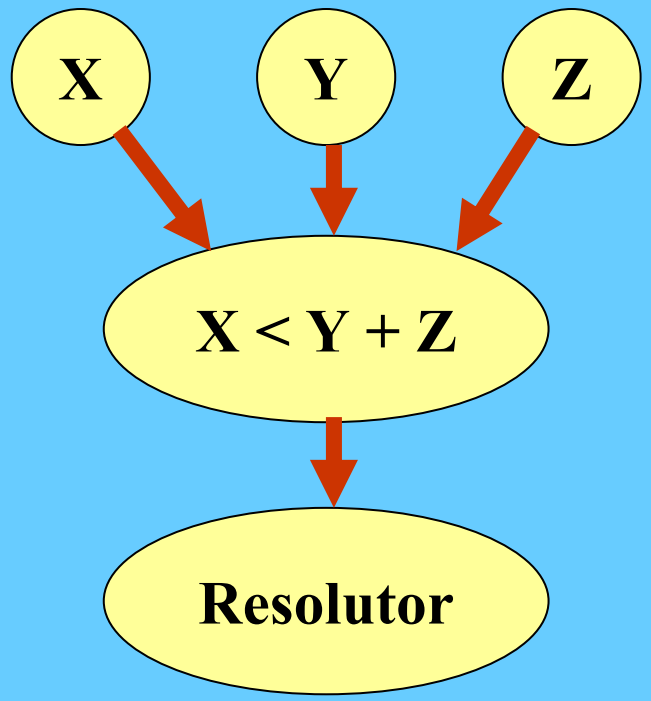
Elementos en el entorno de programación orientada a objetos

Objetos y sus relaciones

Variables de decisión

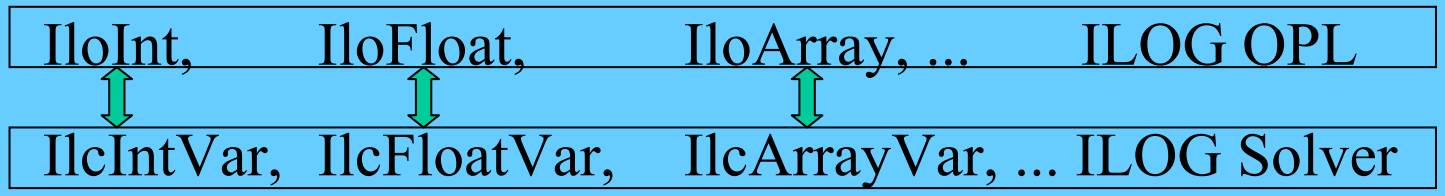
Restricciones

Resolutor



Variables de decisión

Clases



IloInt_*nombre*

Métodos

IloSolver_*solver*::get_*nombre*() ILOG OPL

IlcIntVar::getValue ILOG Solver

Restricciones

Clases

IloConstraint (?)	ILOG OPL
IlcConstraint	ILOG Solver

Métodos

IloSolver_nombre::add	ILOG OPL
IlcManager::add	ILOG Solver

Resolutor

Clases

IloSolver_*nombre*	ILOG OPL
IlcManager	ILOG Solver

Métodos

IloSolver_*nombre*::add	ILOG OPL
IlcManager::add	ILOG Solver
IloSolver_*nombre*:: printInformation	ILOG OPL
IlcManager:: printInformation	ILOG Solver

Aplicaciones

- Visualización de la solución
- Operaciones con modelos OPL (OPL Script)
  - Modificación de datos

    (Ej: parámetros de un modelo - generados en el programa, leídos de un archivo de texto, leídos por consola,...)
  - Adición de restricciones

    (Ej: ajuste de modelos)

Visualización de la solución (1/3)


```
int main(int argc,char* argv[])
{
  int i = 1;
  try {
    IloSolver_productos solver;
    if (solver.nextSolution()) {
      cout << "solución[" << i << "]" << endl;
      MostrarSolucion(solver);
      i++;
      cout << endl;
    }
    solver.end();
  }
  catch (...) {
  }
  return 0;
}
```

Visualización de la solución (2/3)

```
enum Productos ...;
```

```
var float+ prodInterna[Productos];
var float+ compraExterna[Productos];
```

```
void MostrarSolucion(IloSolver_productos solver) {
  IloEnumIterator_Productos ite(solver.get_Productos());
  IloArray_prodInterna prodInterna = solver.get_prodInterna();
  IloArray_compraExterna compraExterna = solver.get_compraExterna();
  while (ite.ok()) {
    IloEnumValue_Productos p = *ite;
    cout << "prodInterna[" << p << "]=" << prodInterna[p] << "\t";
    cout << "compraExterna[" << p << "]=" << compraExterna[p] << endl
<< flush;
    ++ite;
  }
  solver.printInformation();
}
```

Visualización de la solución (3/3)


```
solución[1]
prodInterna[p1]=40.000000   compraExterna[p1]=60.000000
prodInterna[p2]=0.000000    compraExterna[p2]=200.000000
prodInterna[p3]=0.000000    compraExterna[p3]=300.000000
Number of fails                 : 0
Number of choice points         : 0
Number of variables             : 15
Number of constraints           : 9
Reversible stack (bytes)        : 4044
Solver heap (bytes)             : 136204
Solver global heap (bytes)      : 4044
And stack (bytes)               : 4044
Or stack (bytes)                : 4044
Search Stack (bytes)            : 4044
Constraint queue (bytes)        : 4112
Total memory used (bytes)       : 160536
Elapsed time since creation     : 0.030000
```

Operaciones con modelos (1/2)


```
int n = ...;
var int reinas[1..n] in 1..n;
solve
      forall(ordered i,j in 1..n) {
        reinas[i] <> reinas[j];
        reinas[i] + i <> reinas[j] + j;
        reinas[i] - i <> reinas[j] - j
      };

data {
   n = 5;
};
```

Operaciones con modelos (2/2)


```
int main(int argc,char* argv[])
{
  int i = 1;
  try {
    IloSolver_reinas solver;
    if (solver.nextSolution()) {
      cout << "solución[" << i << "]"
      << endl;
      i++;
      cout << endl;
    }
    solver.end();
  }
  catch (...) {
  }
  return 0;
}
```

```
int main(int argc,char* argv[])
{
  IloSolver_reinas solver(IloEdit);
  try {
    for (int i = 5; i <= 8; i++) {
      IloInt n = solver.get_n();
      n = i;
      if (solver.nextSolution()) {
        MostrarSolucion(solver);
      }
      solver.reset(IloEdit);
    }
    solver.end();
  }
  catch (...) {
  }
  return 0;
}
```