Algoritmos y Programación

Programación con Restricciones 4

Hoy veremos

- Modelar con conjuntos
- Restricciones globales
- Ruptura de simetría
  - Simetría de variable
  - Simetría de valor

Modelo básico, mochila 0-1

```
int: n;
set of int: ITEMS = 1..n;

int: capacity;

array[ITEMS] of int: value;
 array[ITEMS] of int: weight;

array[ITEMS] of var 0..1: taken;

constraint (sum(i in ITEMS)(weight[i]*taken[i]))<=capacity;

solve maximize sum(i in ITEMS)(value[i]*taken[i]);
```

Modelo mochila n-ítems

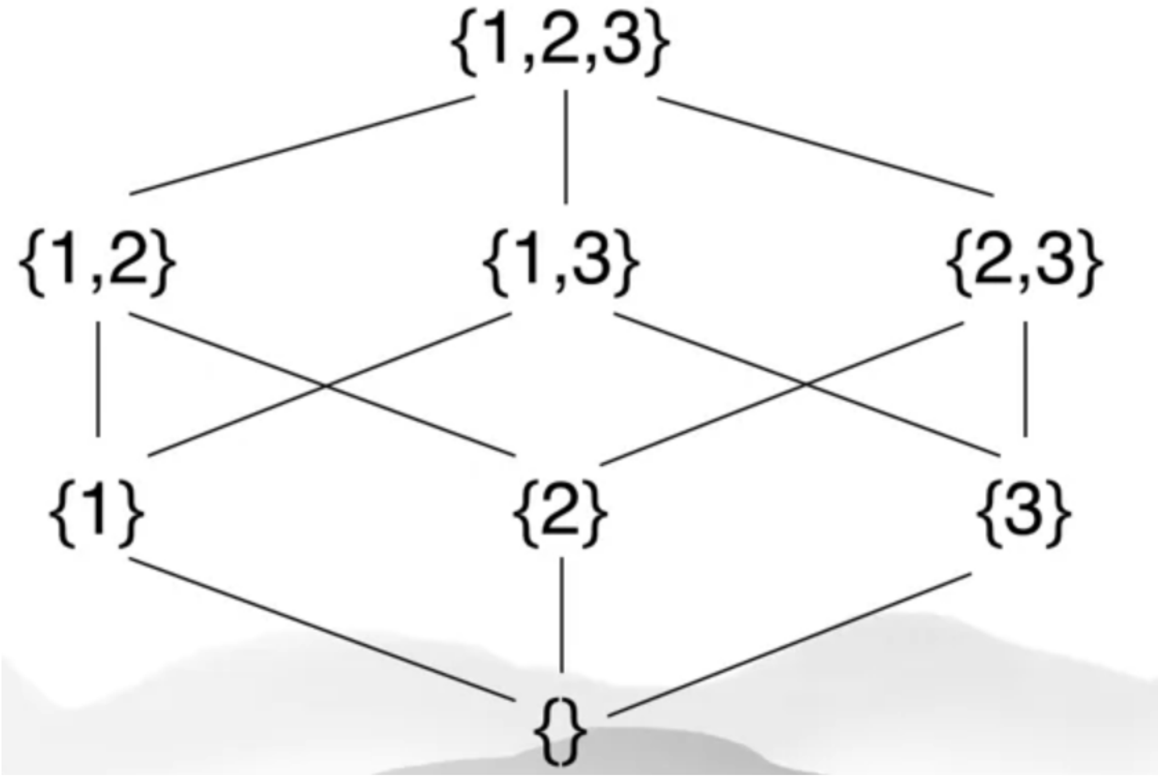
- Plantear un modelo de la mochila, donde los objetos pueden incluirse en la mochila más de una vez
  - Parámetros:
    - Crear un set of int de ITEM
    - 2 Arrays: value y weight
  - Variable de decisión:
    - **Opción 1) Array**: taken
    - **Opción 2) Set**: taken

Solución

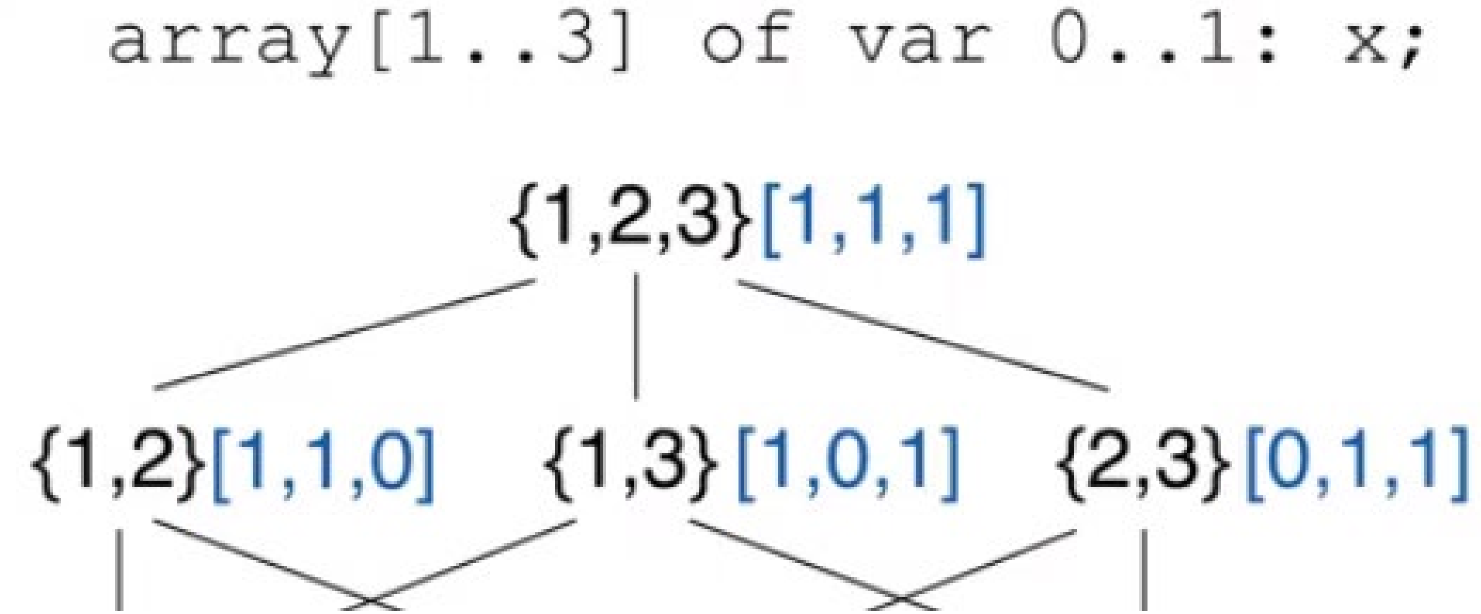
```
int: n;
set of int: ITEMS = 1..n;

int: capacity;

array[ITEMS] of int: value;
 array[ITEMS] of int: weight;

array[ITEMS] of var int: taken;
constraint forall(i in ITEMS)(taken[i]>=0);

constraint (sum(i in ITEMS)(weight[i]*taken[i]))<=capacity;

solve maximize sum(i in ITEMS)(value[i]*taken[i]);
```

Set

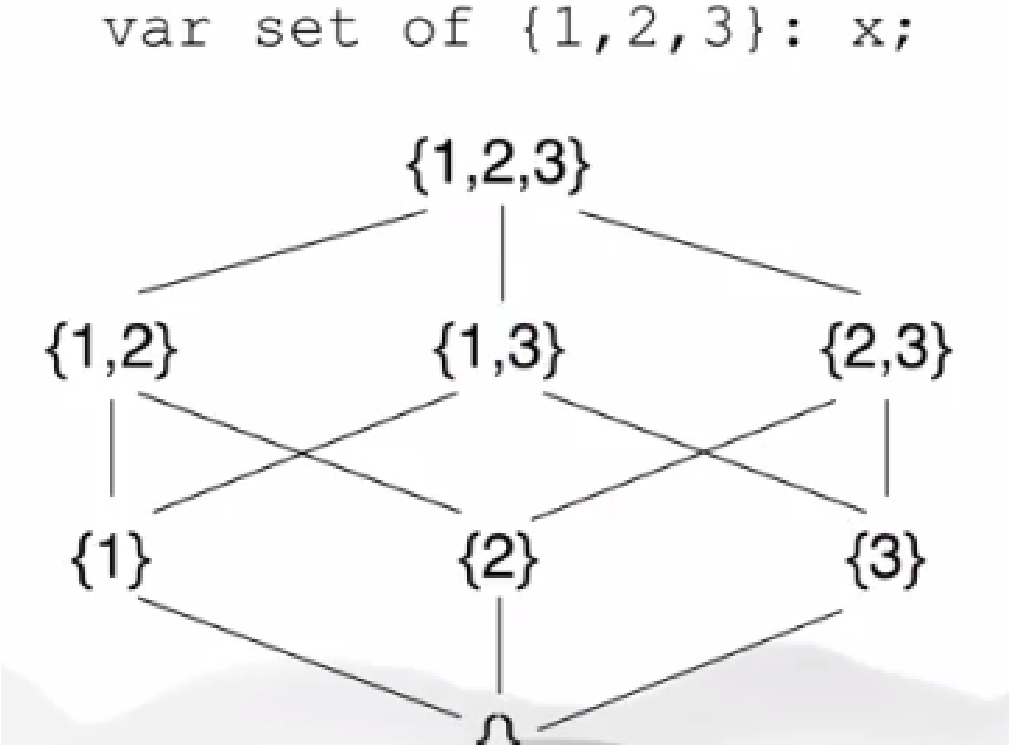
```
var set of {1,2,3}: x;
```



Set vs. Array

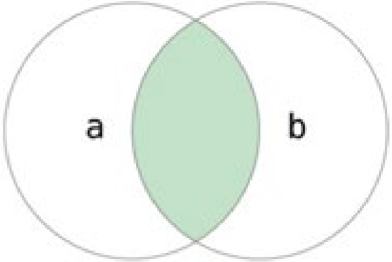


- Operaciones con Set: **in, subset, superset, intersect, unión, card, diff, symdiff**

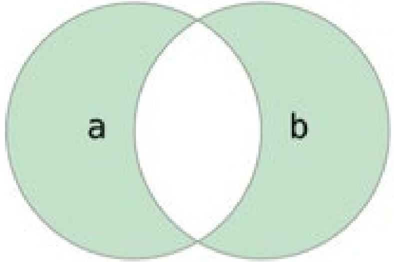


Set

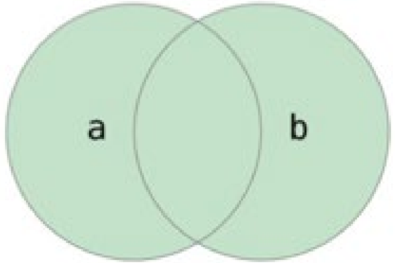
a *intersect* b



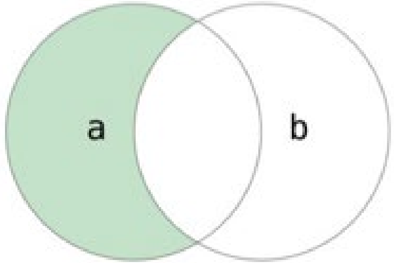
a *symdiff* b



a *union* b



a *diff* b



Por ejemplo, {1,2,5,6} symdiff {2,3,4,5} = {1,3,4,6}

knapsack 0-1 con set

```
int: n;
set of int: ITEMS = 1..n;

int: capacity;
array[ITEMS] of int: value;
array[ITEMS] of int: weight;
%array[ITEMS] of var 0..1: taken;

var set of ITEMS: taken;

constraint (sum(i in ITEMS)(weight[i]*(i in taken))<=capacity;
solve maximize sum(i in ITEMS)(value[i]*(i in taken);
```

knapsack 0-1 con set

```
int: n;
set of int: ITEMS = 1..n;

int: capacity;
array[ITEMS] of int: value;
array[ITEMS] of int: weight;
%array[ITEMS] of var 0..1: taken;
 var set of ITEMS: taken;
```

Recorremos todos los ítems

```
constraint (sum(i in ITEMS)(weight[i]*(i in taken))<=capacity;
solve maximize sum(i in ITEMS)(value[i]*(i in taken));
```

Recorremos todos los ítems

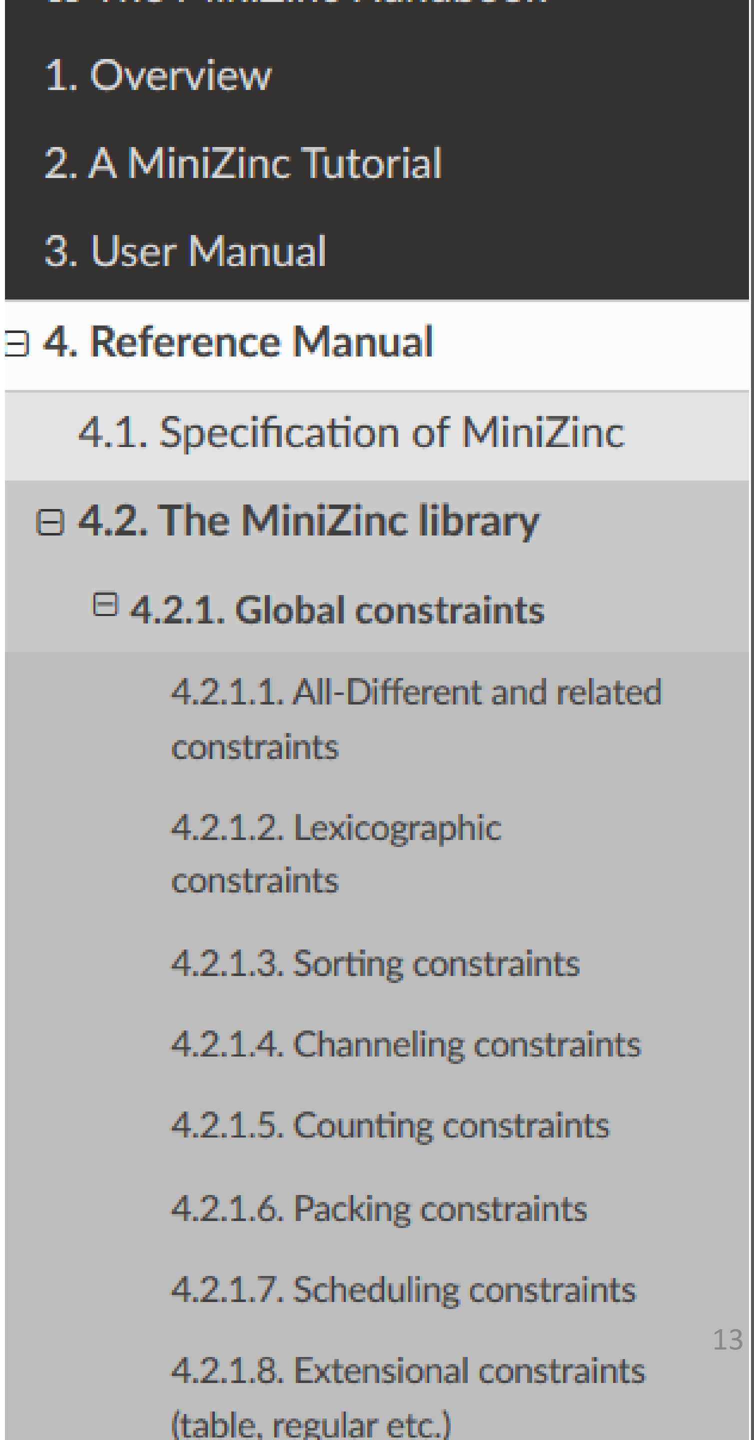
knapsack 0-1 con set. Modelo más conciso

```
int: n;
set of int: ITEMS = 1..n;

int: capacity;
array[ITEMS] of int: value;
array[ITEMS] of int: weight;

 var set of ITEMS: taken;


constraint sum(i in taken)(weight[i])<=capacity;
solve maximize sum(i in taken)(value[i]);
```

Restricciones globales

- Permiten capturar las subestructuras combinatorias
- Existen más de 100 tipos de restricciones globales
- Ejemplos: *alldifferent*, *inverse*, *table*, *circuit*, …

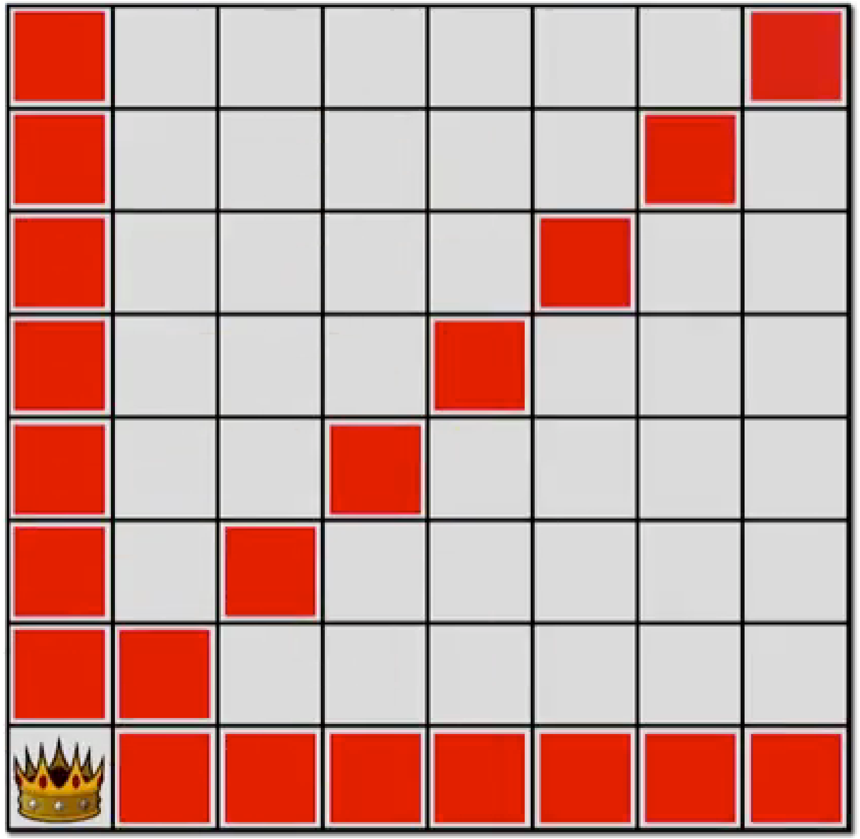
https://www.minizinc.org/doc-2.2.3/en/lib-globals.html



Restricciones globales

```
row[1] ≠ row[2];
...
row[1] ≠ row[8];

row[1] ≠ row[2] + 1;
...
row[1] ≠ row[8] + 7;

row[1] ≠ row[2] - 1;
...
row[1] ≠ row[8] - 7;
```

```
int: n = 8;

set of int: R = 1..n;

array [R] of var R: row;

constraint forall(i in R, j in i+1..n)(
                row[i] != row[j]  /\
                row[i] != row[j] + (j-i) /\
                row[i] != row[j] - (j-i)
                );

solve satisfy;
```



https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas

Restricciones globales

- ***alldifferent(x_1, …, x_n):*** Especifica que todos los valores x_1, …, x_n, deben ser diferentes

```
int: n = 8;

set of int: R = 1..n;

array [R] of var R: row;

constraint forall(i in R, j in i+1..n)(
               row[i] != row[j]  /\
               row[i] != row[j] + (j-i) /\
               row[i] != row[j] - (j-i)
               );

solve satisfy;
```

```
include "globals.mzn";

set of int: R = 1..8;
array[R] of var R: row;

constraint alldifferent(row);
constraint all_different([row[i]+i | i in R]);
constraint all_different([row[i]-i | i in R]);

solve satisfy;
```

Restricciones globales

- Cuando añadimos una restricción al sistema, recordemos que hacemos 2 cosas:

  - **Test de factibilidad**
    - $\exists\ v_1 \in D_1, \dots, v_n \in D_n$:
    - $c(x_1 = v_1, \dots, x_n = v_n) = \ true$

  - Reducir el espacio de búsqueda

Restricciones globales

- Pongamos un ejemplo:
  - restricción ***alldifferent(x₁***,…, *x₃)*
  - $x_1 \in [1..2], \ldots, x_3 \in [1..2]$
-
  - $D_1 = \{1,2\}, D_2 = \{1,2\}, D_3 = \{1,2\}$

- ***alldifferent*** nos dará directamente que no es factible. Principio del palomar (https://es.wikipedia.org/wiki/Principio_del_palomar)

-

Restricciones globales

- Sin embargo, con la expresión:

$$x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$$

- Si utilizamos el mismo test que utilizamos para *alldifferent:*
  - $(x_1 = 1, x_2 = 2) = true$
  - $(x_1 = 1, x_3 = 2) = true$
  - $(x_3 = 1, x_1 = 2) = true$

  (recordemos que las restricciones son independientes)

Restricciones globales: Poda

- Supongamos ahora que añadimos un valor al dominio D_3:
  - restricción *alldifferent(x_1,…, x_3)*
  - $x_1 \in [1..2], x_2 \in [1..2]\ , x_3 \in [1..3]$

$$D_1 = \{1,2\}, D_2 = \{1,2\}, D_3 = \{1,2,3\}$$

- Para encontrar una solución, aplicando el principio del palomar, se deduce fácilmente (poda) que $x_3 \neq 1\ y\ x_3 \neq 2$ y por tanto $x_3 = 3$.

- Sin embargo, si hubiésemos utilizado las restricciones $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$, no habríamos podido hacer ninguna poda, ya que no se puede detectar que, para que exista una solución, $x_3 \neq 1\ y\ x_3 \neq 2$.

Restricciones globales

- Resumiendo:

- Las Restricciones globales permiten encontrar combinaciones no factibles con antelación
- Llevan a espacios de búsquedas más pequeños y eficientes
- El subsistema de almacenamiento de dominio es capaz de tener en cuenta simultáneamente todos los dominios involucrados en la restricción
- mejoran la poda

Restricciones de tabla

- Supongamos que tenemos 3 variables de decisión:
-
- $x \in \{1,2\}$
- $y \in \{1,2\}$
- $z \in \{3,4,5\}$
-
- El número total de combinaciones es el producto cartesiano de las 3 variables:
- $|\{1,2\}| \cdot |\{1,2\}| \cdot |\{1,2,3\}|$
-
- en total, 12 posibilidades.

Restricciones de tabla

Las restricciones de tabla, construyen una tabla con todas las posibilidades factibles, supongamos por ejemplo que las soluciones factibles son:

$$x \in \{1,2\}$$
$$y \in \{1,2\}$$
$$z \in \{3,4,5\}$$

Restricciones de Tabla	X	Y	Z
Combinación 1	1	1	5
Combinación 2	1	2	4
Combinación 3	2	2	3
Combinación 4	1	2	3

Si en este momento nos llega que $Z \neq 5$, eliminamos la combinación 1, y observamos que Y sólo puede tomar el valor 2

·Table constraint

The `table` constraint enforces that the tuple of variables takes a value from a set of tuples. Since there are no tuples in MiniZinc this is encoded using arrays. The usage of `table` has one of the forms

```
table(array[int] of var bool: x, array[int, int] of bool: t)
table(array[int] of var int:  x, array[int, int] of int:  t)
```

Ejemplo

MEAL ≡ [DOWNLOAD]

```
% Planning a balanced meal
include "table.mzn";
int: min_energy;
int: min_protein;
int: max_salt;
int: max_fat;
set of FOOD: desserts;
set of FOOD: mains;
set of FOOD: sides;
enum FEATURE = { name, energy, protein, salt, fat, cost};
enum FOOD;
array[FOOD,FEATURE] of int: dd; % food database
```

```
set of int: FEATURES = 1..6;
int: name = 1; int: energy = 2; int: protein = 3;
int: salt = 4; int: fat = 5; int: cost = 6;
```

Ejemplo, valores de parámetros

MEAL.DZN ≡ [DOWNLOAD]

```
FOODS = { icecream, banana, chocolatecake, lasagna,
          steak, rice, chips, brocolli, beans} ;

dd = [| icecream,      1200,  50,  10, 120,  400     % icecream
      | banana,         800, 120,   5,  20,  120     % banana
      | chocolatecake, 2500, 400,  20, 100,  600     % chocolate cake
      | lasagna,       3000, 200, 100, 250,  450     % lasagna
      | steak,         1800, 800,  50, 100, 1200     % steak
      | rice,          1200,  50,   5,  20,  100     % rice
      | chips,         2000,  50, 200, 200,  250     % chips
      | brocolli,       700, 100,  10,  10,  125     % brocolli
      | beans,         1900, 250,  60,  90,  150 |]; % beans

min_energy = 3300;
min_protein = 500;
max_salt = 180;
max_fat = 320;
desserts = { icecream, banana, chocolotecake };
mains = { lasagna, steak, rice };
sides = { chips, brocolli, beans };
```

Ejemplo

```
array[FEATURE] of var int: main;
array[FEATURE] of var int: side;
array[FEATURE] of var int: dessert;
var int: budget;
```

Ejemplo

```
constraint main[name] in mains;
constraint side[name] in sides;
constraint dessert[name] in desserts;
constraint table(main, dd);
constraint table(side, dd);
constraint table(dessert, dd);
constraint main[energy] + side[energy] + dessert[energy] >=min_energy;
constraint main[protein]+side[protein]+dessert[protein] >=min_protein;
constraint main[salt] + side[salt] + dessert[salt] <= max_salt;
constraint main[fat] + side[fat] + dessert[fat] <= max_fat;
constraint budget = main[cost] + side[cost] + dessert[cost];

solve minimize budget;
```

Ruptura de Simetría

- Añade una restricción que fuerza a que toda solución factible sea mejor que la anterior
- Siempre lleva al óptimo global
- Puede haber una explosión combinatoria
- Ruptura de simetría
  - Objetivo: explorar árboles de búsqueda más pequeños
  - En general, si una solución no es factible, no lo será ninguna de las configuraciones simétricas
  - Tipos de simetría
    - Simetría de variable
    - Simetría de valores
  - Posible solución: imponer algún tipo de orden

Simetría de valor

- Ejemplo, coloreado de grafo

```
color = array1d(0..3, [2, 1, 2, 2]);
----------
color = array1d(0..3, [1, 2, 1, 1]);
----------
==========
```

- value_precede

```
predicate value_precede(int: s, int: t, array [int] of var int: x)
```

Requires that `s` precede `t` in the array `x`.

Precedence means that if any element of `x` is equal to `t`, then another element of `x` with a lower index is equal to `s`.

```
array[0..NUM_NODES-1] of var 1..NUM_NODES: color;

constraint forall(n in 1..NUM_NODES-1)
    (value_precede(n, n+1, color));
```

Ejemplo

- Supongamos una matriz de dimensión *vxb* de variables binarias 0/1, que cumple con 3 restricciones:
  1. El número de 1 por filas es *r*.
  2. El número de 1 por columnas es *k*.
  3. El producto escalar de cada par de filas es *l* (se intersectan en *l* puntos)

- De forma genérica la entrada es ***(v,b,r,k,l),*** aparece en la teoría de diseño combinatorio como Diseño de Bloques Incompletos Balanceados (***BIBDs Balanced Incomplete Block Designs***), y se utiliza para el diseño de experimentos.

Producto escalar

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

(3,3,2,2,1)

1	1	0
0	1	1
1	0	1

http://www.austinmohr.com/14spring299/KJH_BIBD_Paper.pdf

Ejemplo

- “Los diseños de bloques son diseños combinatorios de un tipo especial. Este área es una de las partes más antiguas de la combinatoria, como en el problema de la colegiala de Kirkman propuesto en 1850”

https://es.wikipedia.org/wiki/Combinatoria

Problema de la colegiala de *Kirkman*:

Quince jóvenes estudiantes salen de paseo todos los días de la semana, de lunes a domingo, de forma ordenada, formando cinco filas de tres estudiantes cada una, ¿cómo organizarlas todos los días de la semana para que ningún par de alumnas compartan fila más de un día?

BIBD *(v,b,r,k,l)*

1. El número de 1 por filas es *r*.
2. El número de 1 por columnas es *k*.
3. El producto escalar de cada par de filas es *l* (se intersectan en *l* puntos)

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

```
range Rows = 1..v;
range Cols = 1..b;
var{int} m[Rows,Cols] in 0..1;
solve {
   forall(i in Rows)
      sum(y in Cols) m[i,y] = r;
   forall(j in Cols)
      sum(x in Rows) m[x,j] = k;
   forall(i in Rows,j in Rows: j > i)
      sum(x in Cols) (m[i,x] & m[j,x]) = 1;
}
```

Simetría de variable

- Intercambiando cualquier par de filas, también nos da una solución válida

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

(7,7,3,3,1)

1	0	1	0	1	0	0
0	1	1	0	0	1	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

Simetría de variable

- Posible solución: imponer un orden lexicográfico

El orden lexicográfico podemos obtenerlo del valor binario de la configuración, por ejemplo: si *a=0110010* y *b=1010100* entonces $a \leq b$.

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

Matriz ordenada lexicográficamente

(7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	1

No aparecen simetrías por filas

Simetría de variable

- Intercambiando cualquier par de columnas, también nos da una solución válida

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

(7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	0	0	0	0	1	1
0	1	0	0	1	0	1
1	1	0	1	0	0	0
0	0	0	1	1	1	0

Simetría de variable

- Haciendo lo mismo por columnas:

(7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	1

(7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1

- No aparecen simetrías por filas
- No aparecen simetrías por columnas

BIBD *(v,b,r,k,l).* Simetría de variable

1. El número de 1 por filas es *r*.
2. El número de 1 por columnas es *k*.
3. El producto escalar de cada par de filas es *l* (se intersectan en *l* puntos)

(7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1

```
range Rows = 1..v;
range Cols = 1..b;
var{int} m[Rows,Cols] in 0..1;
solve {
   forall(i in Rows)
      sum(y in Cols) m[i,y] = r;
   forall(j in Cols)
      sum(x in Rows) m[x,j] = k;
   forall(i in Rows,j in Rows: j > i)
      sum(x in Cols) (m[i,x] & m[j,x]) = 1;
   forall(i in 1..v-1)
      lexleq(all(j in Cols) m[i,j],all(j in Cols) m[i+1,j]);
   forall(j in 1..b-1)
      lexleq(all(i in Rows) m[i,j],all(i in Rows) m[i,j+1]);
}
```

Para cada par de filas (*i* e *i+1*), añadimos una restricción lexicográfica, y lo mismo con las columnas (*j* y *j+1*)