



Universidad de
Oviedo



Técnicas de Inteligencia Artificial para la Optimización y Programación de Recursos

Programación con Restricciones

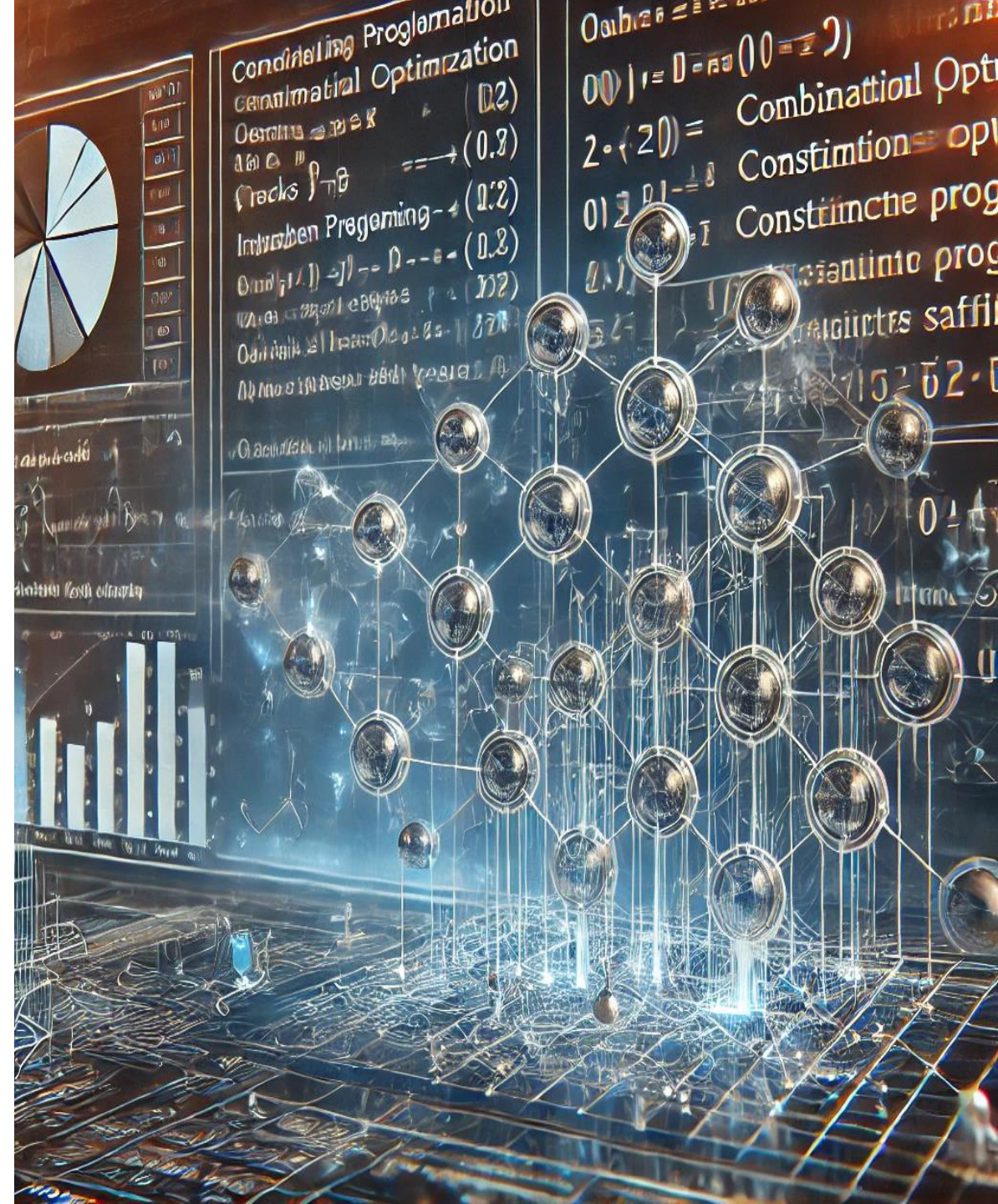
Carlos Mencía Cascallana

Raúl Mencía Cascallana

{menciacarlos, menciaraul}@uniovi.es

Ciencia de la Computación e Inteligencia Artificial

Departamento de Informática



Contenidos

- 1. Problemas de satisfacción de restricciones y optimización**
2. El lenguaje MiniZinc
3. Modelado de problemas de scheduling

Contenidos

1. Problemas de satisfacción de restricciones y optimización

- **Introducción y conceptos básicos**
- Ejemplos de modelos de CSPs/CSOPs
- Resolución

2. El lenguaje MiniZinc

3. Modelado de problemas de scheduling

Sudoku

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Completar la cuadrícula con números del 1 al 9, de modo que el mismo número no se repita en ninguna fila, columna o bloque.

Sudoku

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Completar la cuadrícula con números del 1 al 9, de modo que el mismo número no se repita en ninguna **fila**, columna o bloque.

Sudoku

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Completar la cuadrícula con números del 1 al 9, de modo que el mismo número no se repita en ninguna fila, **columna** o bloque.

Sudoku

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Completar la cuadrícula con números del 1 al 9, de modo que el mismo número no se repita en ninguna fila, columna o **bloque**.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Inicialmente tenemos 9 opciones en cada celda.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Si nos fijamos en la primera fila, podemos descartar los valores 2 y 5 en todas las celdas, ya que no se puede repetir el mismo número en una misma fila.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Si nos fijamos en la primera fila, podemos descartar los valores 2 y 5 en todas las celdas, ya que no se puede repetir el mismo número en una misma fila.

Los eliminamos.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Repetimos el proceso anterior en cada fila, columna y bloque, y el número de opciones se reduce considerablemente.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Podemos reducir las opciones aún más.

En el bloque superior derecho, observamos que el valor 1 solamente está presente en una celda. Lo mismo ocurre con el valor 2.

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Podemos reducir las opciones aún más.

En el bloque superior derecho, observamos que el valor 1 solamente está presente en una celda. Lo mismo ocurre con el valor 2.

Fijamos dichos valores y los eliminamos de las opciones de las celdas de su misma fila y columna.

Sudoku

<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>2</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>5</div>	<div>9</div>	<div>1</div>	<div>1 2 3 4 5 6 7 8 9</div>
<div>1 2 3 4 5 6 7 8 9</div>	<div>9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>7</div>	<div>3</div>	<div>2</div>
<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>2</div>	<div>7</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>6</div>	<div>1 2 3 4 5 6 7 8 9</div>
<div>2</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>4</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>9</div>
<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>9</div>	<div>7</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>6</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>
<div>6</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1</div>
<div>1 2 3 4 5 6 7 8 9</div>	<div>8</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>4</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>6</div>
<div>4</div>	<div>6</div>	<div>3</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>8</div>	<div>1 2 3 4 5 6 7 8 9</div>
<div>1 2 3 4 5 6 7 8 9</div>	<div>2</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>6</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>8</div>	<div>1 2 3 4 5 6 7 8 9</div>	<div>4</div>	<div>1 2 3 4 5 6 7 8 9</div>

Repetimos el mismo procedimiento en todas las filas, columnas y bloques hasta que no sea posible descartar más opciones.

Este proceso se llama **propagación de restricciones**

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	7	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9

Búsqueda

- Seleccionamos una celda.
- Partimos en subproblemas (valor 1, valor 8).

			2		5	9	1	
	9					7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

valor 1 / \ valor 8

Sudoku

Valor 1

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	7	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9

			2		5	9	1	
	9					7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

valor 1 / \ valor 8

			2		5	9	1	
	9		1			7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

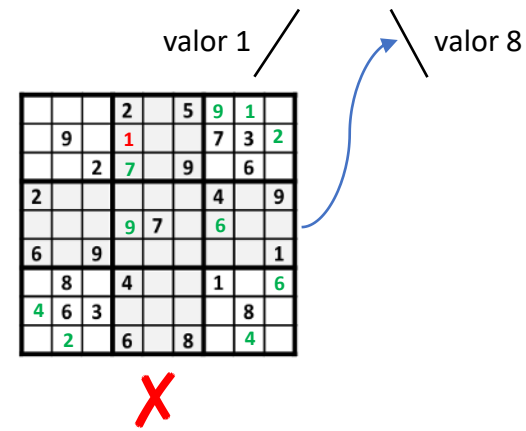
Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	7	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9

Valor 1

- Tras la propagación de restricciones, se agotan las opciones en una celda (**no hay solución**).
- Probamos con el otro valor.

			2		5	9	1	
	9					7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	



Sudoku

Valor 8

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	7	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9

			2		5	9	1	
	9					7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

valor 1

valor 8

			2		5	9	1	
	9		1			7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

			2		5	9	1	
	9		8			7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

X

Sudoku

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	5	9	1	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	3	2
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	2	7	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9
2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
6	1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1
1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	6
4	6	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	6	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9

Valor 8

- Propagamos restricciones (no hay inconsistencias).
- Seguimos buscando a partir de esta configuración.

			2		5	9	1	
	9					7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

valor 1

valor 8

			2		5	9	1	
	9		1			7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

X

			2		5	9	1	
	9		8			7	3	2
		2	7		9		6	
2						4		9
			9	7		6		
6		9						1
	8		4			1		6
4	6	3					8	
	2		6		8		4	

...

Sudoku

3	7	8	2	6	5	9	1	4
5	9	6	8	1	4	7	3	2
1	4	2	7	3	9	5	6	8
2	1	7	3	8	6	4	5	9
8	5	4	9	7	1	6	2	3
6	3	9	5	4	2	8	7	1
7	8	5	4	2	3	1	9	6
4	6	3	1	9	7	2	8	5
9	2	1	6	5	8	3	4	7

Solución

Sudoku

X_{11}	X_{12}	X_{13}	X_{14} 2	X_{15}	X_{16} 5	X_{17}	X_{18}	X_{19}
X_{21}	X_{22} 9	X_{23}	X_{24}	X_{25}	X_{26}	X_{27} 7	X_{28} 3	X_{29}
X_{31}	X_{32}	X_{33} 2	X_{34}	X_{35}	X_{36} 9	X_{37}	X_{38} 6	X_{39}
X_{41} 2	X_{42}	X_{43}	X_{44}	X_{45}	X_{46}	X_{47} 4	X_{48}	X_{49} 9
X_{51}	X_{52}	X_{53}	X_{54}	X_{55} 7	X_{56}	X_{57}	X_{58}	X_{59}
X_{61} 6	X_{62}	X_{63} 9	X_{64}	X_{65}	X_{66}	X_{67}	X_{68}	X_{69} 1
X_{71}	X_{72} 8	X_{73}	X_{74} 4	X_{75}	X_{76}	X_{77} 1	X_{78}	X_{79}
X_{81}	X_{82} 6	X_{83} 3	X_{84}	X_{85}	X_{86}	X_{87}	X_{88} 8	X_{89}
X_{91}	X_{92}	X_{93}	X_{94} 6	X_{95}	X_{96} 8	X_{97}	X_{98}	X_{99}

Sudoku es un **problema de satisfacción de restricciones**

Variables de decisión: X_{ij} con $1 \leq i, j \leq 9$ (una para cada celda)

Dominios: cada variable X_{ij} debe tomar un valor en $\{1, \dots, 9\}$

Restricciones:

■ Filas:

$\text{alldifferent}(X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19})$
 \dots
 $\text{alldifferent}(X_{91}, X_{92}, X_{93}, X_{94}, X_{95}, X_{96}, X_{97}, X_{98}, X_{99})$

■ Columnas:

$\text{alldifferent}(X_{11}, X_{21}, X_{31}, X_{41}, X_{51}, X_{61}, X_{71}, X_{81}, X_{91})$
 \dots
 $\text{alldifferent}(X_{19}, X_{29}, X_{39}, X_{49}, X_{59}, X_{69}, X_{79}, X_{89}, X_{99})$

■ Bloques:

$\text{alldifferent}(X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{23}, X_{31}, X_{32}, X_{33})$
 \dots
 $\text{alldifferent}(X_{77}, X_{78}, X_{79}, X_{87}, X_{88}, X_{89}, X_{97}, X_{98}, X_{99})$

■ Pistas iniciales:

$X_{14} = 2$ $X_{16} = 5$ $X_{22} = 9$ \dots $X_{94} = 6$ $X_{96} = 8$

Sudoku

X ₁₁	X ₁₂	X ₁₃	X ₁₄ 2	X ₁₅	X ₁₆ 5	X ₁₇	X ₁₈	X ₁₉
X ₂₁	X ₂₂ 9	X ₂₃	X ₂₄	X ₂₅	X ₂₆	X ₂₇ 7	X ₂₈ 3	X ₂₉
X ₃₁	X ₃₂	X ₃₃ 2	X ₃₄	X ₃₅	X ₃₆ 9	X ₃₇	X ₃₈ 6	X ₃₉
X ₄₁ 2	X ₄₂	X ₄₃	X ₄₄	X ₄₅	X ₄₆	X ₄₇ 4	X ₄₈	X ₄₉ 9
X ₅₁	X ₅₂	X ₅₃	X ₅₄	X ₅₅ 7	X ₅₆	X ₅₇	X ₅₈	X ₅₉
X ₆₁ 6	X ₆₂	X ₆₃ 9	X ₆₄	X ₆₅	X ₆₆	X ₆₇	X ₆₈	X ₆₉ 1
X ₇₁	X ₇₂ 8	X ₇₃	X ₇₄ 4	X ₇₅	X ₇₆	X ₇₇ 1	X ₇₈	X ₇₉
X ₈₁	X ₈₂ 6	X ₈₃ 3	X ₈₄	X ₈₅	X ₈₆	X ₈₇	X ₈₈ 8	X ₈₉
X ₉₁	X ₉₂	X ₉₃	X ₉₄ 6	X ₉₅	X ₉₆ 8	X ₉₇	X ₉₈	X ₉₉

Resolución de CSPs

- Búsqueda + propagación de restricciones
- Técnicas de *propósito general* (sirven para resolver cualquier problema modelado como un CSP).
- Razonar en términos de las restricciones permite reducir el espacio de búsqueda significativamente.
- Flexibilidad en la búsqueda, mediante heurísticos de selección de variables/valores.

Problemas de Satisfacción de Restricciones

Un problema de satisfacción de restricciones (**CSP**) es una tripla $P = (X, D, C)$, donde:

$X = \{X_1, X_2, \dots, X_n\}$ es un conjunto finito de **variables**

$D = \{D_1, D_2, \dots, D_n\}$ especifica el **dominio** de cada variable

- El dominio $D_i = \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ es el conjunto de valores posibles para X_i
- También emplearemos $dom(X_i)$ para referirnos a D_i

$C = \{C_1, C_2, \dots, C_m\}$ es un conjunto de **restricciones** sobre las variables de X

El objetivo es asignar un valor $v_i \in D_i$ a cada variable $X_i \in X$ de modo que se satisfagan todas las restricciones de C .

CSP: *Constraint Satisfaction Problem*

Variables / Dominios

Variables

- Enteras, reales, booleanas, conjuntos, tuplas, etc.
- Emplearemos principalmente variables enteras.

Dominios

- Discretos/continuos, finitos/infinitos.
- Consideraremos dominios discretos finitos.
- Ejemplos:

$\{2, 6, 9\}$

$\{\text{rojo, verde, azul}\}$

$[1,3] = 1..3 = \{1,...,3\} = \{1, 2, 3\}$

Restricciones

Una **restricción** restringe los valores que las variables pueden tomar al mismo tiempo

Formalmente: $C_i \in \mathcal{C}$ es un par $C_i = (S_i, R_i)$

- $S_i = \{x_{i1}, x_{i2}, \dots, x_{ik}\} \subseteq X$ son las variables de C_i (**ámbito**)
- $R_i \subseteq D_{i1} \times D_{i2} \times \dots \times D_{ik}$ son las tuplas que satisfacen C_i (**relación**)
- La **aridad** de C_i es el número de variables de su ámbito, es decir $|S_i|$

Representación

- Extensional $(\{x, y\}, \{(1,2), (1,3), (2,3)\}) \quad (x, y) \in \{(1,2), (1,3), (2,3)\}$
- Intensional $(\{x, y\}, x < y) \quad x < y$

Una tupla $\tau \in D_{i1} \times D_{i2} \times \dots \times D_{ik}$ **satisface** C_i si $\tau \in R_i$ e **infringe** C_i si $\tau \notin R_i$

Restricciones

Las restricciones pueden ser

- Aritméticas

$$x + 1 < y \quad z \neq y * x \quad x \bmod 2 = 0 \quad \sum_{i=1}^n x_i < 25$$

- Lógicas

$$x = 2 \vee x < y \quad x \neq z \wedge x < y \quad x = 3 \rightarrow x < 2 * y$$

- Explícitas (tuplas de valores posibles)

$$(x, y) \in \{(0,0), (1,1), (2,2)\} \quad (x, y, z) \in \{(3,1,7), (2,5,0)\}$$

Restricciones

Dependiendo de su **aridad**, una restricción puede ser:

- Unaria si contiene una variable ($x * x > 9$)
- Binaria si contiene dos variables ($x + y < 5$)
- Ternaria si contiene tres variables ($x + y = z$)
- ...

Restricciones globales

- Aridad no fija, con un significado concreto
- Facilitan el modelado e incorporan métodos específicos de propagación
- Ejemplo: *alldifferent*(X_1, \dots, X_k) impone que las variables X_1, \dots, X_k toman valores diferentes

Asignaciones

Una **asignación** (o instanciación) A asocia a una o varias variables un valor de su dominio

Representaciones

- $A = \{x = 1, y = 5, z = 9\}$
- $A = (1, 5, 9)$ si establecemos un orden entre las variables (x, y, z)

Asignación completa / parcial

- Completa si asigna un valor a **todas** las variables de un problema
- Parcial si asigna un valor a solo algunas variables de un problema

Asignación **consistente** / **inconsistente**

- Consistente si satisface todas las restricciones del problema
- Inconsistente si infringe alguna restricción del problema

Solución

Solución de un CSP P : asignación completa y consistente

- Si existe al menos una solución: CSP consistente, satisfacible o factible
- En caso contrario: CSP inconsistente o insatisfacible

Ejemplos

$$X = \{x, y, z\}$$

$$D: D_x = \{1, 2, 3, 4\}, D_y = \{2, 3, 5\}, D_z = \{1, 4\} \quad \text{Consistente} \\ \{x = 4, y = 3, z = 1\}$$

$$C = \{x \neq y, x \geq y * z\}$$

$$X = \{x, y, z\}$$

$$D: D_x = D_y = D_z = \{1, 2\} \quad \text{Inconsistente}$$

$$C = \{x \neq y, x \neq z, y \neq z\}$$

Problemas de Optimización (CSOPs)

Un problema de satisfacción y optimización de restricciones (**CSOP**) es una cuádrupla $P = (X, D, C, f)$, donde:

$X = \{X_1, X_2, \dots, X_n\}$ es un conjunto finito de **variables**

$D = \{D_1, D_2, \dots, D_n\}$ especifica el **dominio** de cada variable

$C = \{C_1, C_2, \dots, C_m\}$ es un conjunto de **restricciones** sobre las variables de X

f es una **función objetivo** definida sobre (un subconjunto de) las variables de X .

El objetivo es asignar un valor $v_i \in D_i$ a cada variable $X_i \in X$ de modo que se satisfagan todas las restricciones en C y **se optimice (maximice o minimice) la función objetivo**

CSOP: *Constraint Satisfaction and Optimization Problem*

También COP: *Constraint Optimization Problem*

Problemas de Optimización (CSOPs)

Solución factible: asignación que satisface todas las restricciones

Solución óptima: solución factible que optimiza la función objetivo

Ejemplo

$$\mathbf{X} = \{x, y, z\}$$

$$\mathbf{D}: D_x = \{1, 2, 3, 4\}, D_y = \{2, 3, 5\}, D_z = \{1, 4\}$$

$$\mathbf{C} = \{C_1, C_2\} \text{ con } C_1: x \neq y, C_2: x \geq y * z$$

Objetivo: minimizar $f(x, y, z) = x + y$

Escribiremos $f(\mathbf{X}) = x + y$

Soluciones factibles:

$$S_1 = \{x = 4, y = 3, z = 1\}, f(S_1) = 7$$

$$S_2 = \{x = 4, y = 2, z = 1\}, f(S_2) = 6$$

$$S_3 = \{x = 3, y = 2, z = 1\}, f(S_3) = 5 \text{ (óptima)}$$

Contenidos


1. Problemas de satisfacción de restricciones y optimización

- Introducción y conceptos básicos
- **Ejemplos de modelos de CSPs/CSOPs**
- Resolución

2. El lenguaje MiniZinc

3. Modelado de problemas de scheduling

Modelado

- El primer paso es entender el problema
 - Datos de entrada, ¿qué se tiene que calcular?, ¿cómo son las soluciones?, etc.
- Modelar / Formular / Representar el problema como un CSP/COP
 - Variables de decisión, dominios 
 - Restricciones
 - Función objetivo y criterio de optimización (COPs)
- **Corrección:** soluciones del CSP/COP \Leftrightarrow soluciones del problema
- **Eficiencia:** el CSP/COP debe poder resolverse lo más rápido posible (**avanzado**)

Problemas

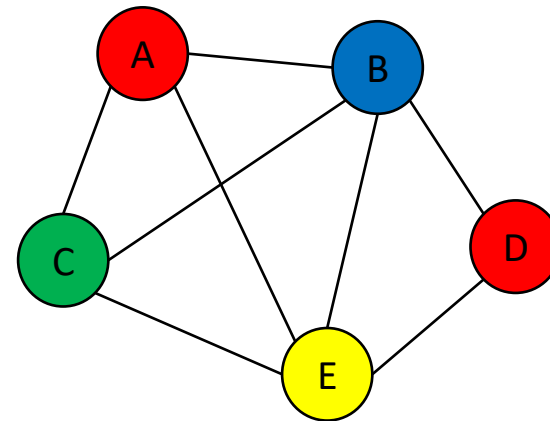
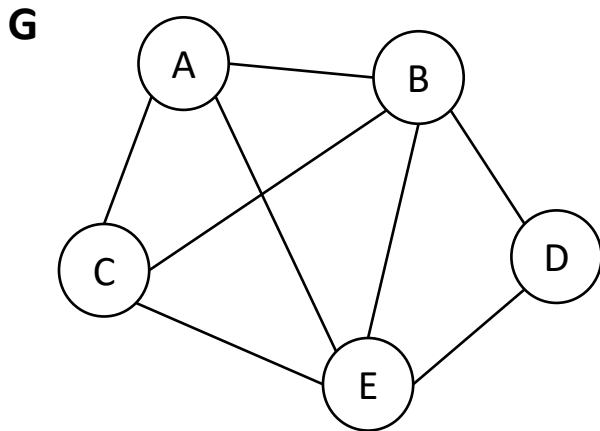
- Coloreado de grafos
- El problema de las N-reinas
- El problema de la mochila (0-1)
- Planificación de tareas en una máquina

Problemas

- **Coloreado de grafos**
- El problema de las N-reinas
- El problema de la mochila (0-1)
- Planificación de tareas en una máquina

Coloreado de grafos

Un grafo no dirigido $G = (V, E)$ es **k-coloreable** si se puede asignar un color a cada vértice, de modo que cada par de vértices adyacentes tengan colores distintos y se empleen como máximo k colores.



G es 4-coloreable
(no es 3-coloreable)

Coloreado de grafos

Dado un grafo no dirigido $G = (V, E)$ y un número natural $k \geq 1$. Determinar si G es k -coloreable. En caso afirmativo, colorearlo empleando como máximo k colores.

Modelo CSP

- Variables: $\mathbf{X} = \{X_v \mid v \in V\}$
- Dominios: $\text{dom}(X_v) = \{1, \dots, k\}$, para todo $X_v \in \mathbf{X}$
- Restricciones:
 - Para cada arco $\{i, j\} \in E$: $X_i \neq X_j$

Una variable asociada a cada vértice (color)

Con esta definición, garantizamos que no se utilicen más de k colores

Cada arco da lugar a una restricción, imponiendo que dos vértices no tengan el mismo color si son adyacentes

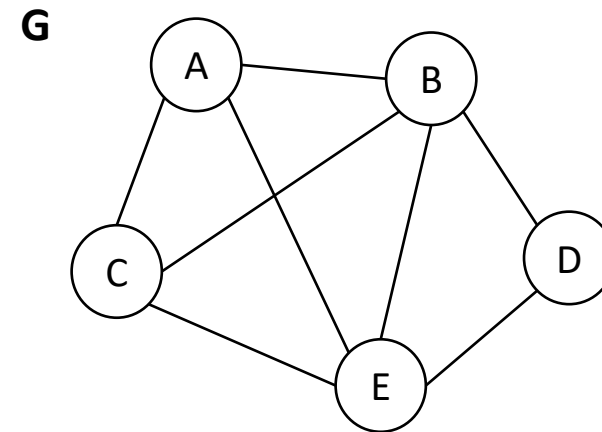
Coloreado de grafos

Dado un grafo no dirigido $G = (V, E)$ y un número natural $k \geq 1$. Determinar si G es k -coloreable. En caso afirmativo, colorearlo empleando como máximo k colores.

Datos de entrada: $G, k = 4$

- $\mathbf{X} = \{X_A, X_B, X_C, X_D, X_E\}$
- $\text{dom}(X_v) = \{1, 2, 3, 4\}$, para todo $X_v \in \mathbf{X}$
- Restricciones:

$X_A \neq X_B$	$X_A \neq X_C$	$X_A \neq X_E$	$X_B \neq X_C$
$X_B \neq X_D$	$X_B \neq X_E$	$X_C \neq X_E$	$X_D \neq X_E$



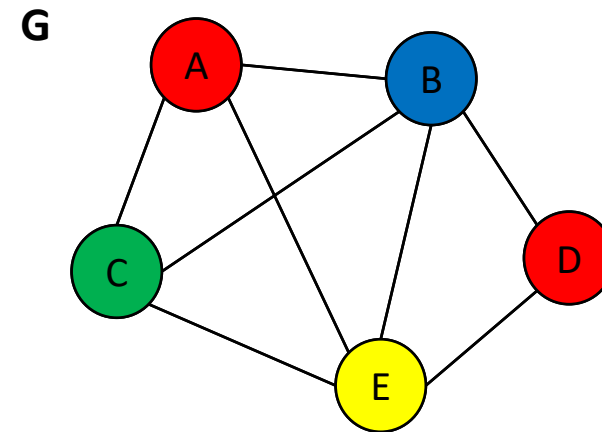
Coloreado de grafos

Dado un grafo no dirigido $G = (V, E)$ y un número natural $k \geq 1$. Determinar si G es k -coloreable. En caso afirmativo, colorearlo empleando como máximo k colores.

Datos de entrada: $G, k = 4$

- $\mathbf{X} = \{X_A, X_B, X_C, X_D, X_E\}$
- $\text{dom}(X_v) = \{1, 2, 3, 4\}$, para todo $X_v \in \mathbf{X}$
- Restricciones:

$X_A \neq X_B$	$X_A \neq X_C$	$X_A \neq X_E$	$X_B \neq X_C$
$X_B \neq X_D$	$X_B \neq X_E$	$X_C \neq X_E$	$X_D \neq X_E$



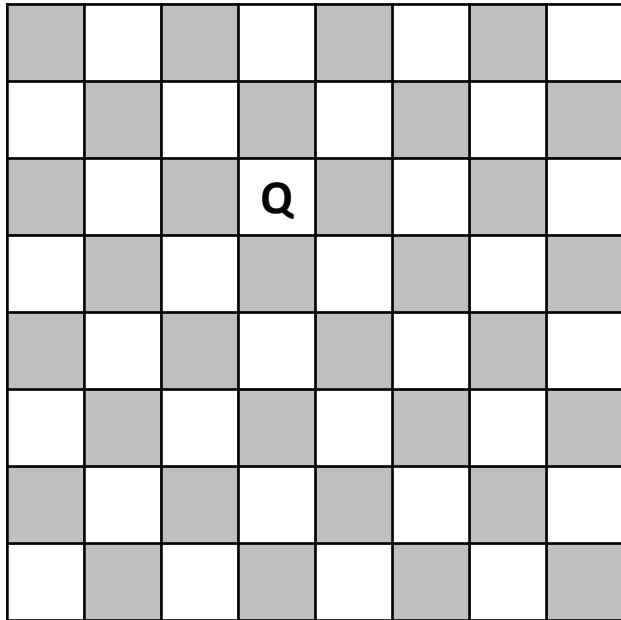
$$\{ X_A = 1, X_B = 2, X_C = 3, X_D = 1, X_E = 4 \}$$

Problemas

- Coloreado de grafos
- **El problema de las N-reinas**
- El problema de la mochila (0-1)
- Planificación de tareas en una máquina

El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.



El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.

	X		X		X		
		X	X	X			
X	X	X	Q	X	X	X	X
		X	X	X			
	X		X		X		
X			X			X	
			X				X
			X				

Posiciones atacadas

- Fila
- Columna
- Diagonal
- Antidiagonal

El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.

	Q						
						Q	
		Q					
					Q		
							Q
				Q			
Q							
			Q				

Una posible solución

El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.

Modelo CSP

- Variables: $\mathbf{X} = \{X_i \mid i \in \{1, \dots, N\}\}$
- Dominios: $\text{dom}(X_i) = \{1, \dots, N\}$, para todo $Q_i \in \mathbf{X}$
- Restricciones:
 - Columna: $X_i \neq X_j$, con $1 \leq i < j \leq N$
 - Diagonal: $X_i - i \neq X_j - j$, con $1 \leq i < j \leq N$
 - Antidiagonal: $X_i + i \neq X_j + j$, con $1 \leq i < j \leq N$

Una variable para cada reina. X_i es la columna en la que irá la reina de la fila i . Esto garantiza que cada reina se coloque en una fila diferente.

En principio, X_i se puede colocar en cualquiera de las N columnas

Con $1 \leq i < j \leq N$ consideramos todos los posibles pares $\{i, j\}$, donde $i \neq j$ sin repetir ninguno

El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.

Datos de entrada: N = 4

- Variables: $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$
- Dominios: $\text{dom}(X_i) = \{1, \dots, 4\}$, para todo $X_i \in \mathbf{X}$

- Restricciones:

- Columnas:

$X_1 \neq X_2$	$X_1 \neq X_3$	$X_1 \neq X_4$
$X_2 \neq X_3$	$X_2 \neq X_4$	$X_3 \neq X_4$

- Diagonal:

$X_1 - 1 \neq X_2 - 2$	$X_1 - 1 \neq X_3 - 3$	$X_1 - 1 \neq X_4 - 4$
$X_2 - 2 \neq X_3 - 3$	$X_2 - 2 \neq X_4 - 4$	$X_3 - 3 \neq X_4 - 4$

- Antidiagonal:

$X_1 + 1 \neq X_2 + 2$	$X_1 + 1 \neq X_3 + 3$	$X_1 + 1 \neq X_4 + 4$
$X_2 + 2 \neq X_3 + 3$	$X_2 + 2 \neq X_4 + 4$	$X_3 + 3 \neq X_4 + 4$

	1	2	3	4
X_1				
X_2				
X_3				
X_4				

El problema de las N-reinas

Colocar N reinas en un tablero de NxN de manera que no se ataquen entre sí.

Datos de entrada: N = 4

- Variables: $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$
- Dominios: $\text{dom}(X_i) = \{1, \dots, 4\}$, para todo $X_i \in \mathbf{X}$

- Restricciones:

- Columnas:

$X_1 \neq X_2$	$X_1 \neq X_3$	$X_1 \neq X_4$
$X_2 \neq X_3$	$X_2 \neq X_4$	$X_3 \neq X_4$

- Diagonal:

$X_1 - 1 \neq X_2 - 2$	$X_1 - 1 \neq X_3 - 3$	$X_1 - 1 \neq X_4 - 4$
$X_2 - 2 \neq X_3 - 3$	$X_2 - 2 \neq X_4 - 4$	$X_3 - 3 \neq X_4 - 4$

- Antidiagonal:

$X_1 + 1 \neq X_2 + 2$	$X_1 + 1 \neq X_3 + 3$	$X_1 + 1 \neq X_4 + 4$
$X_2 + 2 \neq X_3 + 3$	$X_2 + 2 \neq X_4 + 4$	$X_3 + 3 \neq X_4 + 4$

	1	2	3	4
X_1		Q		
X_2				Q
X_3	Q			
X_4			Q	

$$\{X_1 = 2, X_2 = 4, X_3 = 1, X_4 = 3\}$$

Problemas

- Coloreado de grafos
- El problema de las N-reinas
- **El problema de la mochila (0-1)**
- Planificación de tareas en una máquina

El problema de la mochila (0-1)

Se tiene una mochila con capacidad para llevar un **peso máximo W** y un conjunto de n **objetos** $O = \{O_1, \dots, O_n\}$. Cada objeto O_i tiene un peso $w_i > 0$ y un valor $v_i > 0$. Seleccionar qué objetos llevar en la mochila de modo que no se exceda su capacidad y se **maximice el valor total de los objetos** seleccionados.

Ejemplo: $W = 20$, 4 objetos:

O_i	w_i	v_i
O_1	8	10
O_2	4	8
O_3	10	12
O_4	7	9

$\{O_1, O_2, O_3\}$ **no es una solución** (excede W)





$\{O_3, O_4\}$ es una solución, con valor total 21

$\{O_1, O_2, O_4\}$ es una **solución óptima**, con valor total 27

El problema de la mochila (0-1)

Se tiene una mochila con capacidad para llevar un peso máximo W y un conjunto de n objetos $O = \{O_1, \dots, O_n\}$. Cada objeto O_i tiene un peso $w_i > 0$ y un valor $v_i > 0$. Seleccionar qué objetos llevar en la mochila de modo que no se exceda su capacidad y se maximice el valor total de los objetos seleccionados.

Modelo COP

- Variables: $X = \{S_i \mid i \in \{1, \dots, n\}\}$ 
- Dominios: $dom(S_i) = \{0, 1\}$, para todo $S_i \in X$ 
- Restricciones:
 - Capacidad: $\sum_{i=1}^n w_i * S_i \leq W$ 
- Objetivo: maximizar $f(X)$
 - $f(X) = \sum_{i=1}^n v_i * S_i$ 

Una variable S_i asociada a cada objeto O_i

$S_i = 1$: objeto O_i seleccionado
 $S_i = 0$: objeto O_i no seleccionado

El peso de los objetos seleccionados no puede exceder W

Maximizamos el valor total de los objetos seleccionados

El problema de la mochila (0-1)

Datos de entrada: ($W = 20$, ver tabla)

- Variables: $\mathbf{X} = \{S_1, S_2, S_3, S_4\}$
- Dominios: $\text{dom}(S_i) = \{0, 1\}$, para todo $S_i \in \mathbf{X}$
- Restricciones:
 - Capacidad:
$$8 * S_1 + 4 * S_2 + 10 * S_3 + 7 * S_4 \leq 20$$
- Objetivo: maximizar $f(\mathbf{X})$
 - $f(\mathbf{X}) = 10 * S_1 + 8 * S_2 + 12 * S_3 + 9 * S_4$

O_i	w_i	v_i
O_1	8	10
O_2	4	8
O_3	10	12
O_4	7	9

Solución óptima: $\{O_1, O_2, O_4\}$

$$S = \{S_1 = 1, S_2 = 1, S_3 = 0, S_4 = 1\}$$

$$f(S) = 27$$

Problemas

- Coloreado de grafos
- El problema de las N-reinas
- El problema de la mochila (0-1)
- **Planificación de tareas en una máquina**

Planificación de tareas en una máquina

Se requiere planificar un conjunto de n tareas $T = \{T_1, \dots, T_n\}$ en una máquina. Cada tarea T_i tiene los siguientes valores (todos son enteros):

- Instante de inicio más temprano posible $r_i \geq 0$ (cabeza)
- Tiempo de procesamiento $p_i > 0$
- Instante máximo de fin (deseado) $d_i \geq 0$ (deadline)

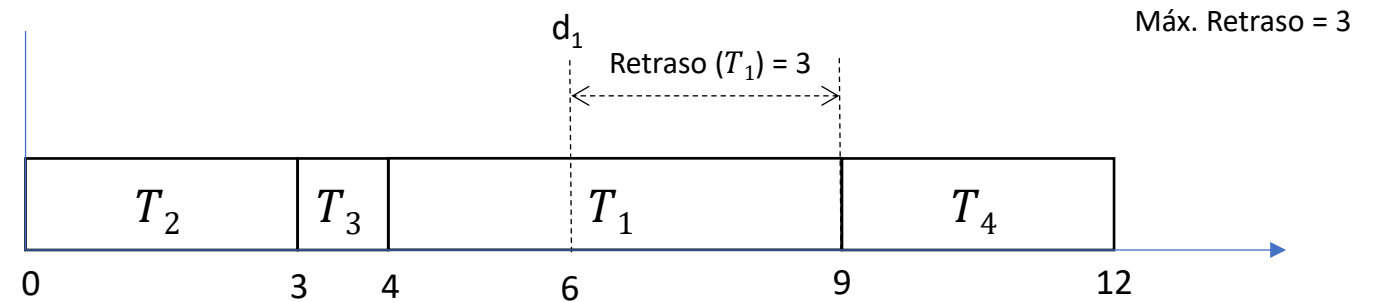
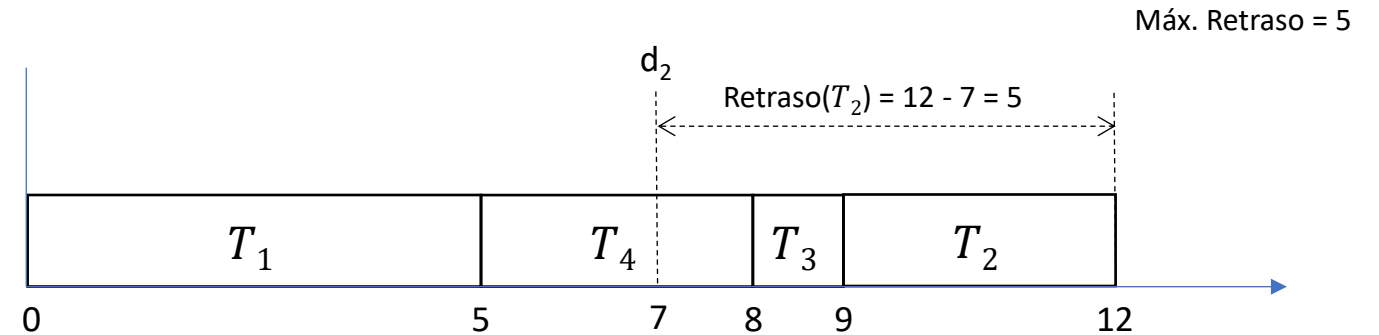
Una **planificación** factible es una **asignación de un tiempo de inicio st_i a cada tarea T_i** , de modo que se procese sin interrupción y su ejecución no se solape en el tiempo con la de ninguna otra tarea.

El objetivo es calcular una planificación factible que **minimice el retraso máximo incurrido por cualquiera de las tareas**.

Planificación de tareas en una máquina

Ejemplo con 4 tareas

T_i	r_i	p_i	d_i
T_1	0	5	6
T_2	0	3	7
T_3	3	1	5
T_4	5	3	9








Solución óptima

Planificación de tareas en una máquina

Se requiere planificar n tareas $T = \{T_1, \dots, T_n\}$ en una máquina. Cada tarea T_i tiene los valores r_i , p_i y d_i . Una planificación factible es una asignación de un tiempo de inicio st_i a cada tarea T_i , de modo que se procese sin interrupción y su ejecución no se solape en el tiempo con la de ninguna otra tarea. El objetivo es calcular una planificación factible que minimice el retraso máximo incurrido por cualquiera de las tareas.

Modelo COP

- Variables: $X = \{st_i \mid i \in \{1, \dots, n\}\}$ 
 - Dominios: $dom(st_i) = \{0, \dots, H\}$, $st_i \in X$ 
 - Restricciones:
 - $st_i \geq r_i$, $i \in \{1, \dots, n\}$ 
 - $(st_i + p_i \leq st_j) \vee (st_j + p_j \leq st_i)$, $1 \leq i < j \leq n$ 
 - Objetivo: minimizar $f(X)$
 - $f(X) = \max_{i \in \{1, \dots, n\}} (st_i + p_i - d_i)$ 
- Una variable st_i para cada tarea T_i (tiempo de inicio)
- Por ahora consideramos 0 como tiempo de inicio más temprano de las tareas, y un H suficientemente grande como tiempo más tardío (Ej: $H = \max_i(r_i) + \sum_{i=1}^n p_i$)
- La tarea T_i no se puede planificar antes del instante r_i
- No solapamiento: dadas T_i y T_j , T_i finaliza antes de que T_j empiece, o T_j finaliza antes del comienzo de T_i
- Minimizamos el máximo retraso: $st_i + p_i - d_i$ es el retraso de la tarea T_i en la planificación

Planificación de tareas en una máquina

Datos de entrada: (n = 4, ver tabla)

- Variables: $X = \{st_1, st_2, st_3, st_4\}$.
- Dominios: $dom(st_i) = \{0, \dots, 17\}, st_i \in X$.

Restricciones:

- $st_i \geq r_i, i \in \{1, \dots, n\}$.

$$H = 5 + (5 + 3 + 1 + 3) = 17$$

$$st_1 \geq 0$$

$$st_2 \geq 0$$

$$st_3 \geq 3$$

$$st_4 \geq 5$$

- $(st_i + p_i \leq st_j) \vee (st_j + p_j \leq st_i), 1 \leq i < j \leq n$

$$(st_1 + 5 \leq st_2) \vee (st_2 + 3 \leq st_1)$$

$$(st_1 + 5 \leq st_3) \vee (st_3 + 1 \leq st_1)$$

$$(st_1 + 5 \leq st_4) \vee (st_4 + 3 \leq st_1)$$

$$(st_2 + 3 \leq st_3) \vee (st_3 + 1 \leq st_2)$$

$$(st_2 + 3 \leq st_4) \vee (st_4 + 3 \leq st_2)$$

$$(st_3 + 1 \leq st_4) \vee (st_4 + 3 \leq st_3)$$

- Objetivo: minimizar $f(X) = \max_{i \in \{1, \dots, n\}} (st_i + p_i - d_i)$

$$f(X) = \max(st_1 + 5 - 6, \quad st_2 + 3 - 7, \quad st_3 + 1 - 5, \quad st_4 + 3 - 9)$$

T_i	r_i	p_i	d_i
T_1	0	5	6
T_2	0	3	7
T_3	3	1	5
T_4	5	3	9

Planificación de tareas en una máquina

Datos de entrada: (n = 4, ver tabla)

- Variables: $X = \{st_1, st_2, st_3, st_4\}$
- Dominios: $dom(st_i) = \{0, \dots, 17\}, st_i \in X$
- Restricciones:
 - $st_i \geq r_i, i \in \{1, \dots, n\}$

$st_1 \geq 0$	$st_2 \geq 0$	$st_3 \geq 3$	$st_4 \geq 5$
---------------	---------------	---------------	---------------

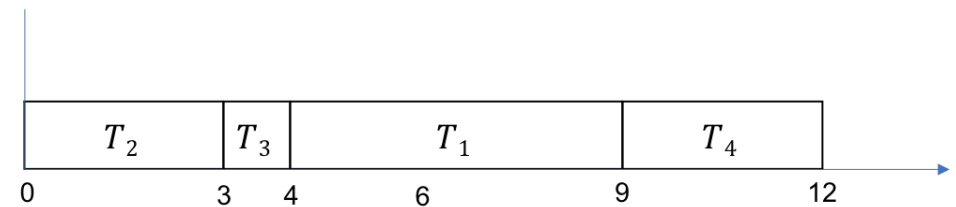
- $(st_i + p_i \leq st_j) \vee (st_j + p_j \leq st_i), 1 \leq i < j \leq n$

$(st_1 + 5 \leq st_2) \vee (st_2 + 3 \leq st_1)$	$(st_1 + 5 \leq st_3) \vee (st_3 + 1 \leq st_1)$	$(st_1 + 5 \leq st_4) \vee (st_4 + 3 \leq st_1)$
$(st_2 + 3 \leq st_3) \vee (st_3 + 1 \leq st_2)$	$(st_2 + 3 \leq st_4) \vee (st_4 + 3 \leq st_2)$	$(st_3 + 1 \leq st_4) \vee (st_4 + 3 \leq st_3)$

- Objetivo: minimizar $f(X) = \max_{i \in \{1, \dots, n\}} (st_i + p_i - d_i)$

$$f(X) = \max(st_1 + 5 - 6, \quad st_2 + 3 - 7, \quad st_3 + 1 - 5, \quad st_4 + 3 - 9)$$

Solución



$$S = \{st_1 = 4, st_2 = 0, st_3 = 3, st_4 = 9\}$$

$$f(S) = 3$$

Prácticas avanzadas de modelado

- Especificar dominios ajustados
- Emplear restricciones globales cuando sea conveniente
- Añadir restricciones implicadas
- Combinar distintas vistas en un único modelo
- Eliminar simetrías

	q						
						q	
		q					
					q		
							q
				q			
q							
			q				

				σ			
	σ						
			σ				
					σ		
							σ
		σ					
σ							
						σ	

				σ			
							σ
			σ				
σ							
		σ					
					σ		
	σ						
						σ	

Contenidos

1. Problemas de satisfacción de restricciones y optimización

- Introducción y conceptos básicos
- Ejemplos de modelos de CSPs/CSOPs
- **Resolución**

2. El lenguaje MiniZinc

3. Modelado de problemas de scheduling

Resolución de CSPs / CSOPs

Búsqueda: explorar de un modo sistemático todas las opciones para encontrar una solución del problema (o demostrar que no existe)

- Coste **exponencial** / Algoritmos **completos**

Propagación de restricciones: Eliminar valores inconsistentes de los dominios de las variables

- Coste **polinomial** / Algoritmos **incompletos**

Búsqueda + propagación de restricciones

- Distintas posibilidades de **combinación**
- Equilibrio entre la efectividad de la propagación de restricciones y su coste computacional

CSPs binarios

CSP binario: todas sus restricciones tienen **aridad 2**

A continuación, nos centraremos en CSPs binarios

- **Teorema:** Todo CSP se puede transformar en un CSP binario equivalente (que representa el mismo conjunto de soluciones).
- Los conceptos / métodos para CSPs binarios pueden **generalizarse** a CSPs arbitrarios.

Asumiremos que entre cada dos variables $\{X_i, X_j\}$ hay a lo sumo una restricción, que denotaremos por $C_{\{i,j\}}$.

Grafo de restricciones

Un CSP binario $P = (X, D, C)$ se puede representar mediante un **grafo de restricciones** $G = (V, E)$:

- Cada **vértice** $v \in V$ representa una **variable** $X_i \in X$.
- Cada **arco** $\{v_i, v_j\} \in E$, representa una **restricción** $C_{\{i,j\}} \in C$ (definida sobre X_i y X_j).

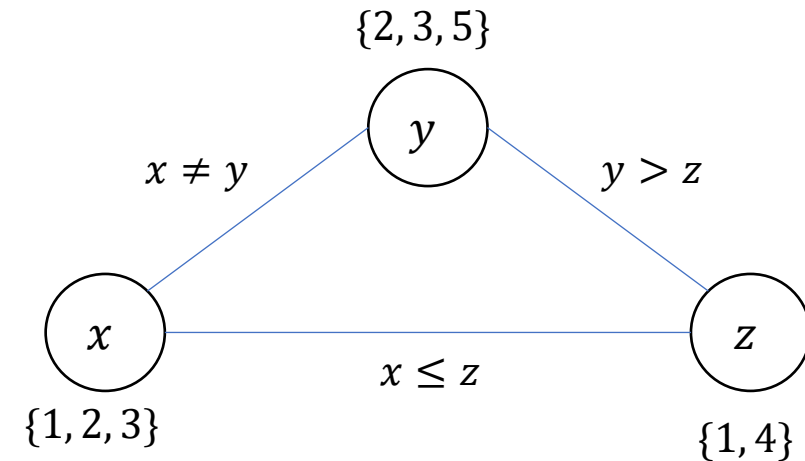
Ejemplo

$P = (X, D, C)$, donde

$X = \{x, y, z\}$

D : $D_x = \{1, 2, 3\}$, $D_y = \{2, 3, 5\}$, $D_z = \{1, 4\}$

$C = \{x \neq y, x \leq z, y > z\}$

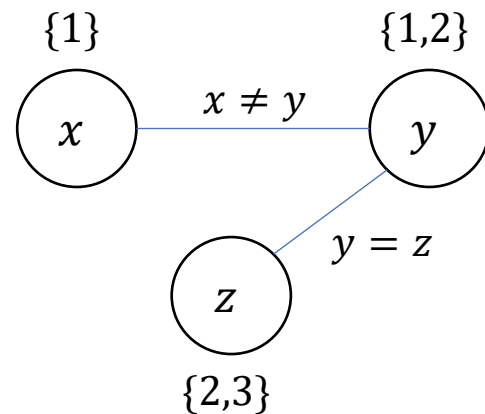


Propagación de restricciones

Dado un CSP $P = (X, D, C)$, una variable $X_i \in X$ y un valor $a \in \text{dom}(X_i)$

- $P[X_i = a]$ denota el CSP obtenido a partir de P , reemplazando X_i por a .
- El valor $a \in \text{dom}(X_i)$ es **consistente** si $P[X_i = a]$ tiene solución, e **inconsistente** si no.

Ejemplo



$P[x = 1]$ tiene solución: $\{x = 1, y = 2, z = 2\} \Rightarrow$ El valor $1 \in D_x$ es consistente

$P[y = 1]$ no tiene solución \Rightarrow El valor $1 \in D_y$ es inconsistente

$P[y = 2]$ tiene solución: $\{x = 1, y = 2, z = 2\} \Rightarrow$ El valor $2 \in D_y$ es consistente

$P[z = 2]$ tiene solución: $\{x = 1, y = 2, z = 2\} \Rightarrow$ El valor $2 \in D_z$ es consistente

$P[z = 3]$ no tiene solución \Rightarrow El valor $3 \in D_z$ es inconsistente

Propagación de restricciones

Los valores inconsistentes se pueden eliminar sin perder ninguna solución del problema.

Determinar si un valor es inconsistente es un problema **NP-completo** (en general), pero en algunos casos se puede detectar fácilmente.

Los algoritmos de **filtrado**, o **propagación de restricciones** detectan y eliminan valores inconsistentes eficientemente.

- Dicho de otro modo, hacen explícitas restricciones implícitas en el problema.

Consistencia local

Los algoritmos de filtrado se basan en el concepto de **consistencia local**

- Consistencia a **nivel de subestructuras del problema** (p. ej., restricciones).

Una **propiedad de consistencia local** permite eliminar valores inconsistentes que no cumplen dicha propiedad

Forzar una propiedad de consistencia local:

- Eliminar valores inconsistentes hasta que se cumpla la propiedad.
- Idealmente: proceso eficiente (complejidad polinomial).

Consistencia local

Propiedades de consistencia local

- Consistencia de nodos (nodo-consistencia)
- Consistencia de arcos (arco-consistencia)
- Consistencia de caminos (camino-consistencia)
- k -consistencia
- ...

Consistencia de nodos

Consistencia a nivel de restricciones unarias.

- Una variable $X_i \in X$ es **nodo-consistente** sii cada valor $a \in D_i$, satisface todas las **restricciones unarias** sobre X_i .
- Un CSP es nodo-consistente sii **todas** sus variables son nodo-consistentes.

Consistencia de arcos

Dada la restricción $C_{\{i,j\}}$ entre las variables X_i y X_j

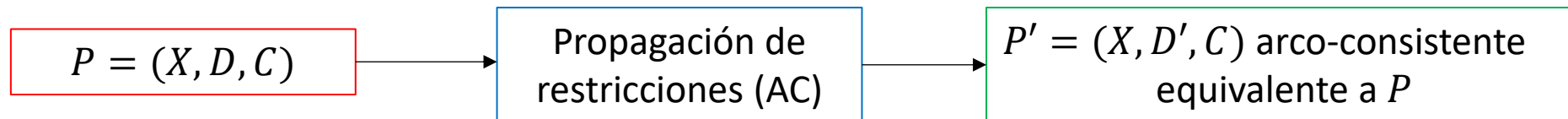
- El **valor** $a \in D_i$ es **arco-consistente** respecto a la variable X_j sii **existe** un valor $b \in D_j$ tal que $\{X_i = a, X_j = b\}$ satisface la restricción $C_{\{i,j\}}$.
 - Si existe, decimos que b es un **sopORTE** de a en D_j .
- El **arco** (X_i, X_j) es **arco-consistente** sii cada valor $a \in D_i$, es arco-consistente respecto a X_j .
- La **restricción** $C_{\{i,j\}}$ entre las variables X_i y X_j es **arco-consistente** sii los dos arcos (X_i, X_j) y (X_j, X_i) son arco-consistentes.
- Un **CSP** es **arco-consistente** sii todas sus restricciones son arco-consistentes.

Consistencia de arcos

Filtrado

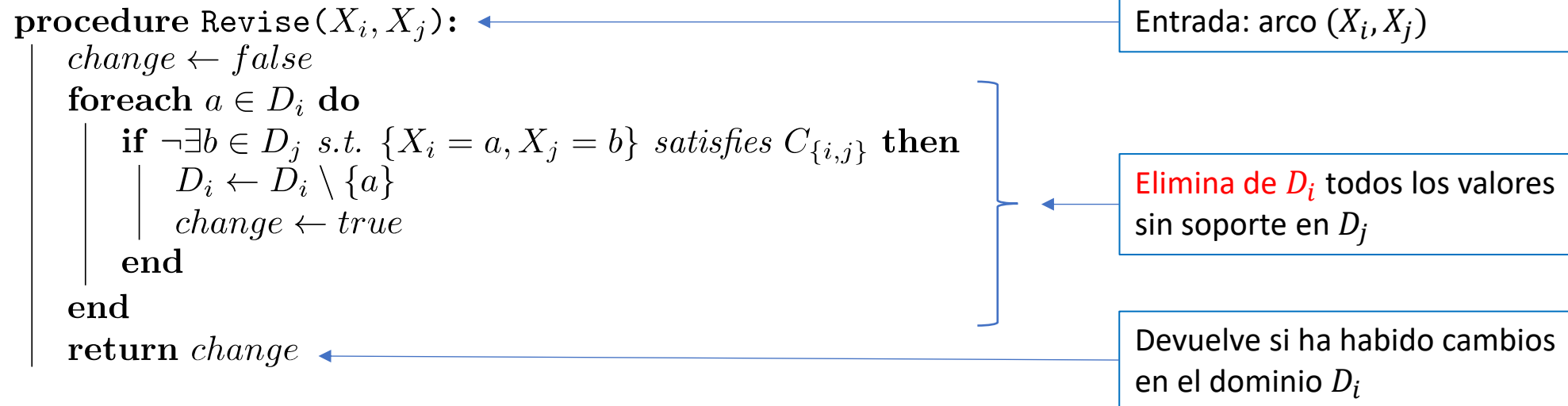
Podemos eliminar valores arco-inconsistentes de los dominios de las variables sin perder ninguna solución del problema

Existen diversos algoritmos para forzar la arco-consistencia en CSPs binarios: **AC-1**, **AC-3**, AC-4, ...



Consistencia de arcos Filtrado

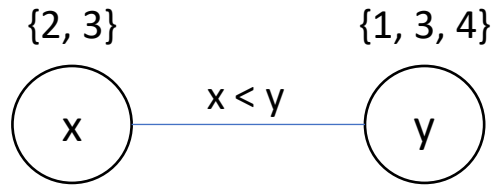
Procedimiento *Revise*: forzar arco-consistencia de un arco (*revisar* un arco)



Complejidad: $O(d^2)$, con $d = \max_i \{|D_i|\}$

Procedimiento *Revise*

Ejemplo



Revise(x, y): eliminamos valores de D_x que no tienen soporte en D_y

- $x = 2$: tiene soporte en D_y (3 y 4)
- $x = 3$: tiene soporte en D_y (4)

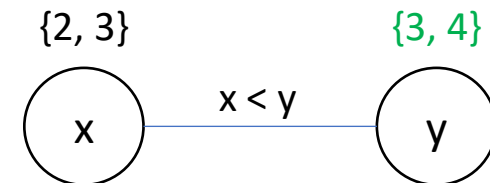
No eliminamos ningún valor de D_x

Revise(y, x): eliminamos valores de D_y que no tienen soporte en D_x

- $y = 1$: no tiene soporte en D_x
- $y = 3$: tiene soporte en D_x (2)
- $y = 4$: tiene soporte en D_x (2 y 3)

Eliminamos el valor 1 de D_y

Resultado:



Algoritmo AC-3

Q es un conjunto (sin repeticiones) de arcos tratado como una cola (normalmente FIFO)

procedure AC-3(X, D, C):

$Q \leftarrow \{(X_i, X_j), (X_j, X_i) \mid C_{\{i,j\}} \in C\}$

while $Q \neq \emptyset$ **do**

$(X_i, X_j) = \text{ExtractFrom}(Q)$

if Revise(X_i, X_j) **then**

$Q \leftarrow Q \cup \{(X_k, X_i) \mid C_{\{k,i\}} \in C, k \neq j\}$

end

end

Entrada: CSP $P = (X, D, C)$

Q se inicializa con todos los arcos (en ambas direcciones)

Extrae el primer arco (X_i, X_j) y lo elimina de Q

Si Revise(X_i, X_j) elimina algún valor de D_i , se añaden a Q todos los arcos que llegan a X_i , excepto (X_j, X_i)

Condiciones de **terminación**

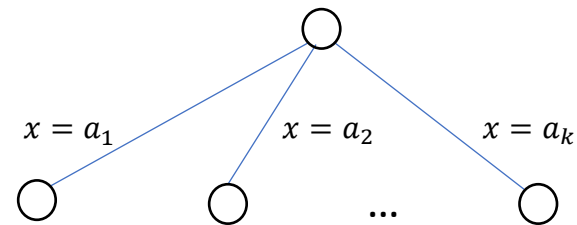
- Al comienzo de una iteración Q está vacía: el CSP ya es **arco-consistente**
- Tras Revise(X_i, X_j) el dominio de X_i se queda vacío: el CSP es **inconsistente**

Resolución mediante búsqueda

Conmutatividad del problema: al llegar a una solución, no importa el orden en que se asignaron valores a las variables

Árbol de búsqueda

- Raíz: asignación vacía
- En cada nodo se considera **una única variable no instanciada**, y se abren tantas ramas como valores haya en su dominio, instanciándola al valor correspondiente en cada una



Variable x
 $D_x = \{a_1, a_2, \dots, a_k\}$

- Tamaño: $O(d^n)$, con factor de ramificación $d = \max_i \{|D_i|\}$ y profundidad máxima $n = |X|$

Backtracking (BT)

Cada nodo del árbol de búsqueda representa una **asignación parcial**

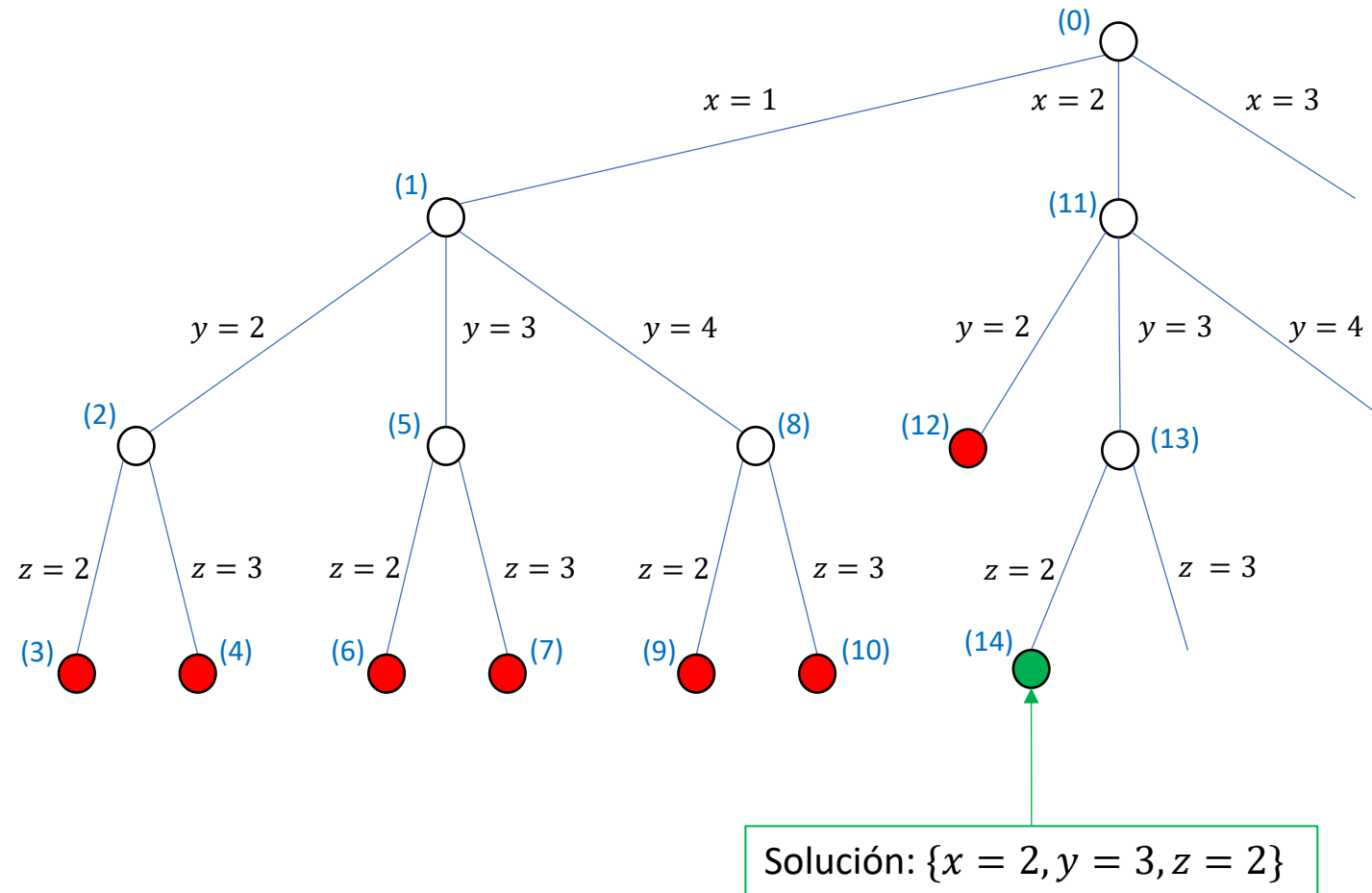
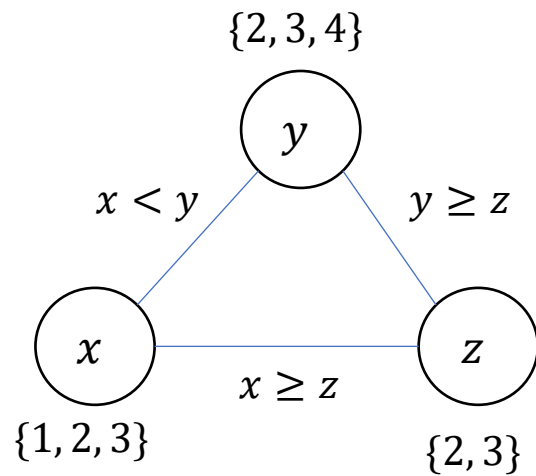
Objetivo: extender la asignación parcial a una **solución** (si existe)

Esquema general

- Seleccionar una variable no asignada previamente
- Asignarle un valor de su dominio
- Si lo anterior no infringe ninguna restricción entre las variables asignadas previamente y la última, repetir el proceso con otra variable no asignada (comprobación ***hacia atrás***)
- En caso contrario, **retroceder**: eliminar la última decisión (se descartan todas las asignaciones que extienden esta asignación parcial)

Backtracking (BT)

Ejemplo



Backtracking + Propagación de Restricciones

Backtracking solo comprueba la consistencia respecto a las variables instanciadas previamente en la rama (comprobación *hacia atrás*)

Es **incapaz de anticipar conflictos** con variables futuras

- Necesita desarrollar una parte (potencialmente grande) del subárbol para detectarlos

Combinación con propagación de restricciones (comprobación *hacia delante*)

- Aplicar antes de y/o durante la búsqueda
- Reducción potencial del tamaño del espacio de búsqueda
- **Compromiso:** Efectividad vs. coste computacional

Backtracking + Propagación de Restricciones

Forzar distintos niveles de consistencia:

- Nodo-consistencia (**eficiente**)
- Arco-consistencia (**parcial o totalmente**)
- Niveles superiores (**coste elevado**)

Compromiso: Efectividad vs. coste

Dos algoritmos:

- **Forward Checking (FC)**: fuerza la arco-consistencia entre la última variable asignada y las variables futuras
- **Maintaining Arc Consistency (MAC)**: fuerza la arco-consistencia de todo el (sub)problema

Heurísticos de ordenación de variables y valores

Criterios para elegir la variable en cada nodo y el orden en que considerar cada valor de su dominio.

Diversos tipos

- Deterministas / Aleatorizados
- Estáticos / Dinámicos
- Propósito general / Dependientes de aplicación
- ...

Nos centraremos principalmente en heurísticos deterministas, dinámicos y de propósito general

El objetivo es **minimizar el número de nodos visitados**

- Cálculo eficiente: no deben suponer demasiado coste computacional.
- Suelen funcionar bien en la práctica, aunque **no ofrecen garantías**.

Heurísticos de ordenación de variables

Determinan la siguiente variable a instanciar en cada nodo del árbol de búsqueda, entre aquellas no asignadas.

Ordenación estática de variables

- Se establece un orden fijo entre las variables antes de iniciar la búsqueda.
- Este orden se repite en todos los niveles del árbol de búsqueda.
- Puede ser útil cuando no se propagan restricciones (ej: Backtracking simple).

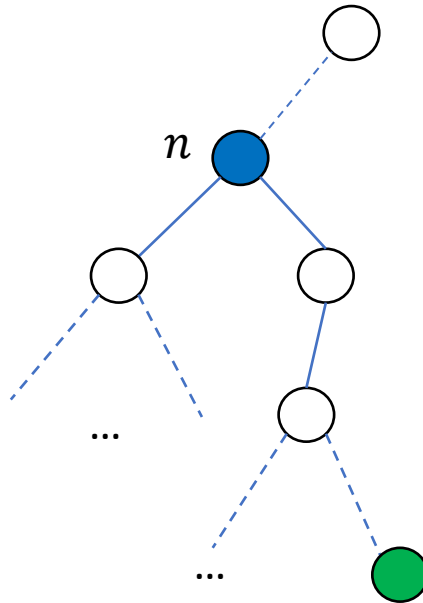
Ordenación dinámica de variables

- El orden de selección de variables puede cambiar de forma dinámica durante la búsqueda.
- Tiene en cuenta la situación del subproblema en cada nodo, incluyendo las variables futuras.
- Adecuada junto con **propagación de restricciones** durante la búsqueda (FC, MAC, ...).

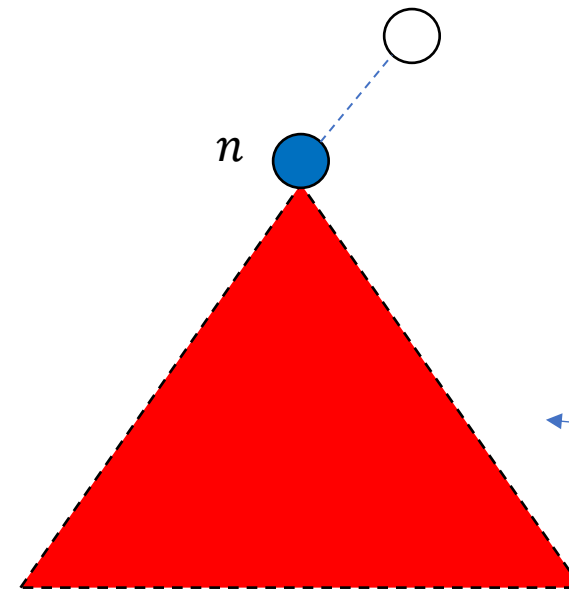
Heurísticos de ordenación de variables

Ordenación dinámica

Dado un nodo n del árbol de búsqueda: ¿qué variable seleccionar en n ?



Situación 1: **hay solución** desde n



Caso más frecuente
(durante la búsqueda)

Situación 2: **no hay solución** desde n

Heurísticos de ordenación de variables

Ordenación dinámica

Asumimos la **segunda situación**

- Si no hay solución a partir del nodo, nos interesa detectarlo lo antes posible.
- Así, podemos retroceder y continuar la búsqueda por otra rama.

Principio fail-first (fallar primero)

- Elegir la variable que, aparentemente, antes nos conduce a una situación sin salida.
- En otras palabras: seleccionar la variable *más restringida*.

"To succeed, try first where you are most likely to fail"

Heurísticos de ordenación de variables

Ordenación dinámica

Heurístico minimum-remaining-values (MRV)

- Seleccionar la variable no instanciada con el menor número de valores restantes en su dominio.
- También conocido como heurístico **dom** o fail-first.

Heurístico maximum-degree (MD)

- Grado (estático) de una variable: número de restricciones en las que está involucrada.
- Grado (dinámico): número de restricciones en las que está involucrada con variables no asignadas.
- Seleccionar la variable con mayor grado (dinámico).
- También conocido como heurístico **deg**.

Combinación de ambos heurísticos

- Heurístico **dom+deg**: emplear dom resolviendo empates con deg.
- Otras opciones: dom/deg, dom/wdeg ...

Heurísticos de ordenación de valores

Una vez se selecciona una variable, cada valor de su dominio abre una nueva rama en el árbol de búsqueda.

- En este punto, hay de **decidir en qué orden explorar cada valor**.
- Si a partir del nodo actual se puede llegar a una solución, interesa encontrarla lo antes posible

Principio succeed-first (tener éxito primero)

- Elegir el valor con el que, aparentemente, se tenga más posibilidades de llegar a una solución
- Asumiendo que tal solución existe (situación 1 mencionada anteriormente)
- En otras palabras: elegir el valor *menos restrictivo*

Heurísticos de ordenación de valores

Heurístico least-constraining-value (LCV) o valor menos restrictivo

- Explorar primero el valor que sea consistente con el mayor número de elementos de los dominios de las variables no asignadas relacionadas con la variable seleccionada.

Cálculo

- Suponemos que en el nodo actual se ha seleccionado la variable s
- Dado un valor $a \in \text{dom}(s)$, para cada variable no asignada u relacionada con s , se cuenta el número de valores $b \in \text{dom}(u)$ tal que $\{s = a, u = b\}$ no infringe ninguna restricción.
- Se asocia al valor $a \in \text{dom}(s)$ el número total de valores compatibles en todas las variables
- El heurístico LCV calcula lo anterior para todos los valores del dominio de s , y los **ordena de mayor a menor en función dicho número total**.

Contenidos

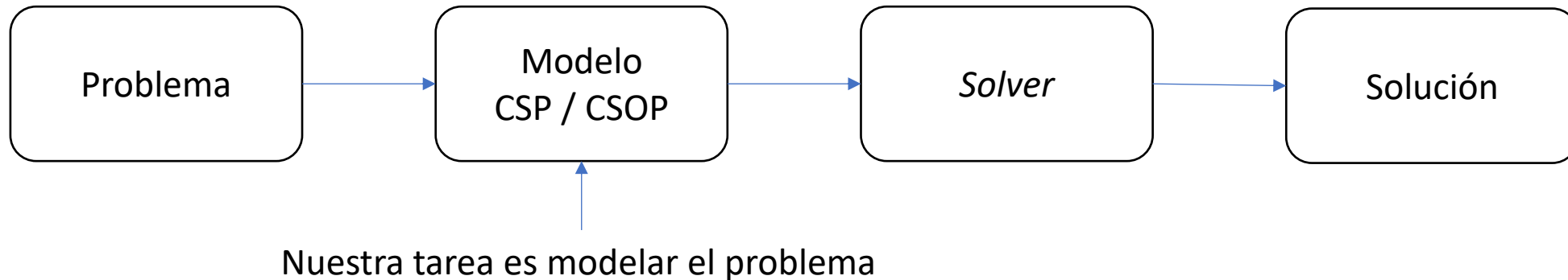
1. Problemas de satisfacción de restricciones y optimización
- 2. El lenguaje MiniZinc**
3. Modelado de problemas de scheduling

Contenidos

1. Problemas de satisfacción de restricciones y optimización
- 2. El lenguaje MiniZinc**
 - Introducción y elementos básicos
 - Arrays
 - Aspectos avanzados
3. Modelado de problemas de scheduling

Programación con Restricciones

La **programación con restricciones** (*constraint programming*) es un paradigma de programación **declarativa**.



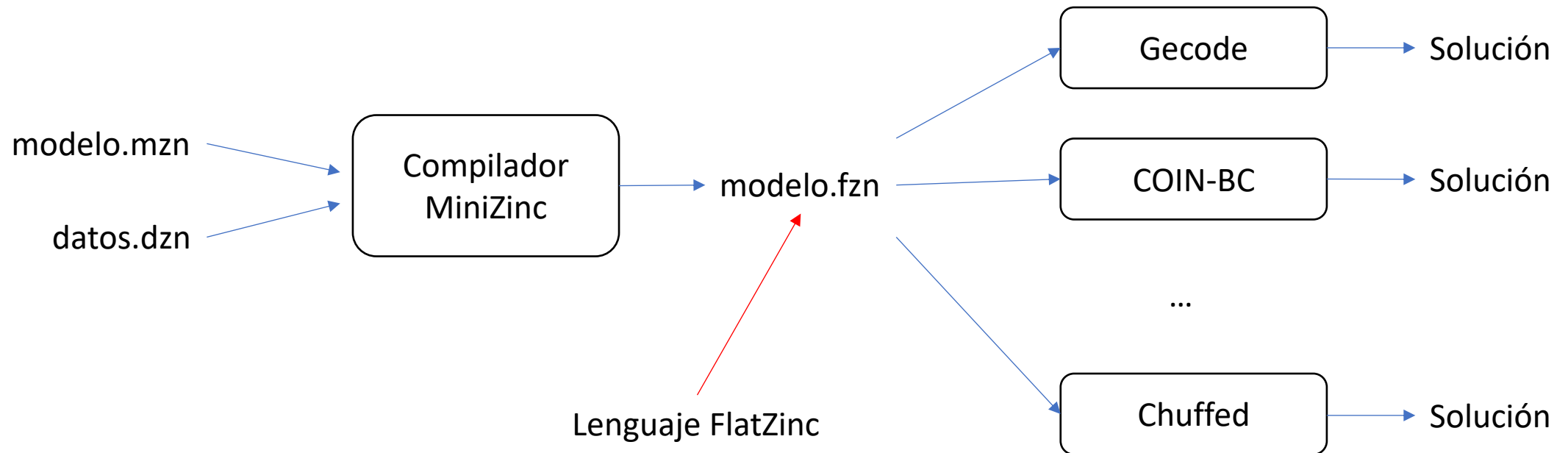
MiniZinc

- Lenguaje de alto nivel de programación con restricciones.
- Desarrollado y mantenido en la Monash University y en la University of Melbourne (Australia).
- Disponible en www.minizinc.org
 - Versiones para Windows, macOS y Linux.
 - Código abierto.
 - Tutoriales y documentación.
- Instalación sencilla
 - Instrucciones en la página web.



MiniZinc

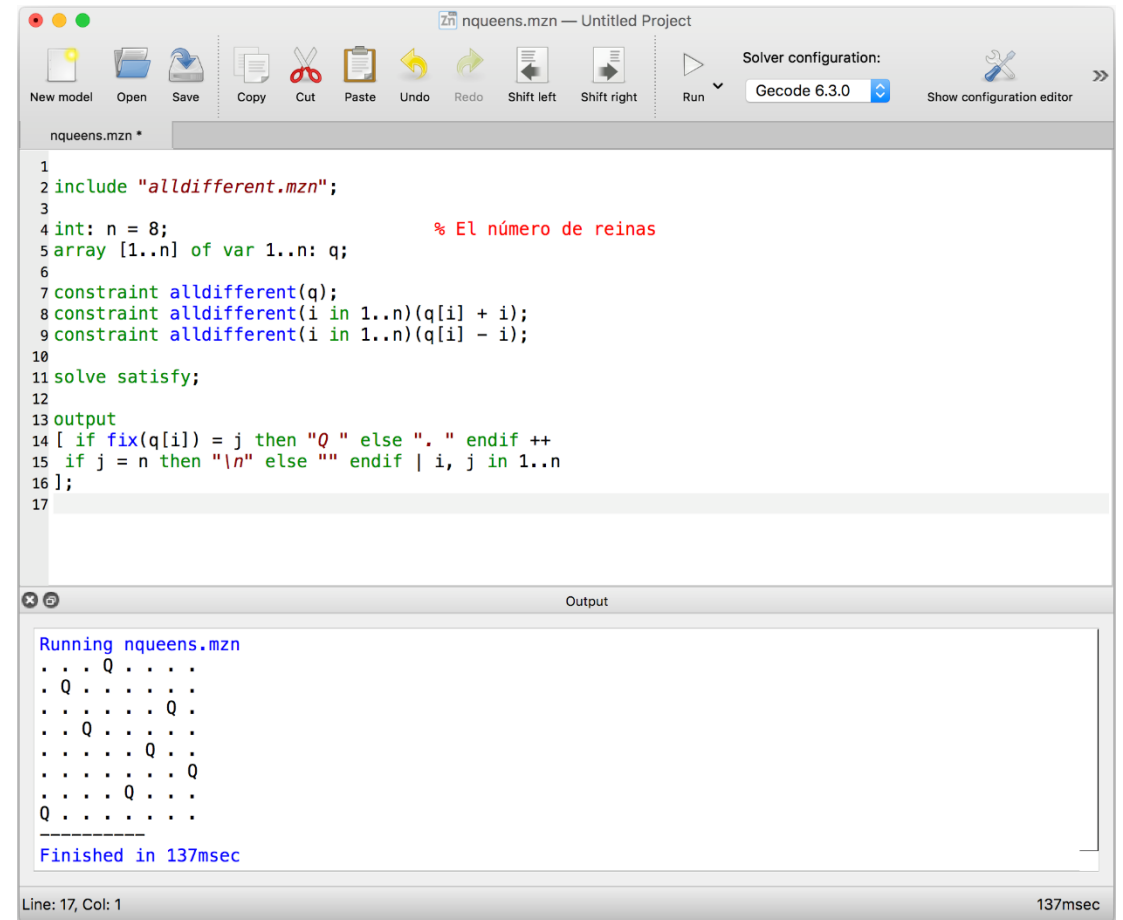
- Compatible con muchos *solvers*



MiniZinc

Se puede ejecutar desde la terminal de comandos o empleando el **IDE**

Interfaz de Python (en desarrollo)



The screenshot displays the MiniZinc IDE interface. The top toolbar includes icons for 'New model', 'Open', 'Save', 'Copy', 'Cut', 'Paste', 'Undo', 'Redo', 'Shift left', 'Shift right', 'Run', and 'Show configuration editor'. The 'Run' button is highlighted, and the solver configuration is set to 'Gecode 6.3.0'. The main editor window shows the code for 'nqueens.mzn', which defines a constraint satisfaction problem for the N-Queens puzzle. The code includes comments in Spanish, such as '% El número de reinas'. The bottom output window shows the execution results, including the solution for the 8-Queens problem and the execution time of 137msec.

```
1
2 include "alldifferent.mzn";
3
4 int: n = 8;                                % El número de reinas
5 array [1..n] of var 1..n: q;
6
7 constraint alldifferent(q);
8 constraint alldifferent(i in 1..n)(q[i] + i);
9 constraint alldifferent(i in 1..n)(q[i] - i);
10
11 solve satisfy;
12
13 output
14 [ if fix(q[i]) = j then "Q " else ". " endif ++
15  if j = n then "\n" else "" endif | i, j in 1..n
16 ];
17
```

Running nqueens.mzn

```
. . . Q . . . .
. Q . . . . .
. . . . . Q .
. . Q . . . .
. . . . . Q .
. . . . . Q .
. . . . . Q .
Q . . . . .
```

Finished in 137msec

Line: 17, Col: 1 137msec

MiniZinc

MiniZinc incorpora un gran número de elementos

- Diversos tipos de variables: enteras, reales, booleanas, conjuntos, ...
- Un catálogo muy amplio de restricciones globales (como alldifferent)
- Anotaciones para indicar distintas posibilidades en la búsqueda
- Etc.

Estudiaremos solo unas partes del lenguaje

Primer ejemplo

Pasos a seguir

1. Abrir el archivo *01_primer_ejemplo.mzn*
2. Observar el programa (y su sintaxis)
 - Parámetros enteros y conjuntos
 - Variables de decisión (*var*), y sus dominios
 - Restricciones (*constraint*)
 - Resolución (*solve*)
3. Ejecutar el programa (*Run*) y observar la salida
4. Descomentar la *restricción extra* y ejecutar el programa. ¿Qué ocurre?
5. Volver comentar la restricción extra.
6. Cambiar *solve satisfy* por las otras dos opciones (una a una). Ejecutar el programa. ¿Qué ocurre?

```
% 01_primer_ejemplo.mzn

% Parámetros / Datos
int: n = 4; % n es un entero

set of int: Dx = 1..n; % Dx es el conjunto {1,...,n}
set of int: Dy = {2,3,5}; % Dy es el conjunto {2,3,5}

% Variables de decisión
var Dx: x; % x es una variable con dominio {1,...,n}
%var 1..n: x: % alternativa
var Dy: y; % y es una variable con dominio {2,3,5}
var {1,4}: z; % z tiene dominio {1,4}

% Restricciones
constraint x != y;
constraint x >= y * z;
%constraint x < y-z; % restricción extra

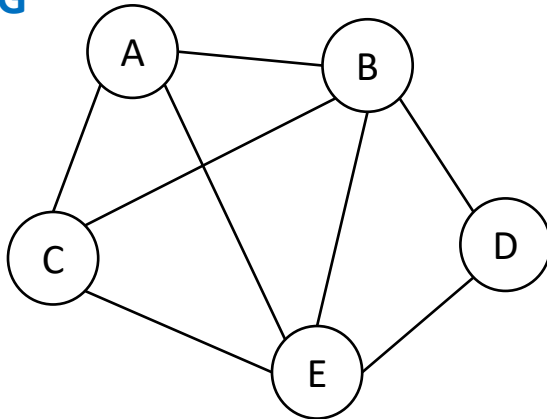
% Resolución
solve satisfy;
%solve minimize x+y;
%solve maximize x+y;
```

Coloreado de grafos

Dado un grafo no dirigido $G = (V, E)$ y un número $k \geq 1$, determinar si G es k -coloreable. En caso afirmativo colorearlo empleando no más de k colores.

Crear un modelo para la siguiente instancia del problema:

$k = 4$ **G**



Modelo CSP (general)

- Variables: $\mathbf{X} = \{X_v \mid v \in V\}$
- Dominios: $\text{dom}(X_v) = \{1, \dots, k\}$, para todo $X_v \in \mathbf{X}$
- Restricciones:
 - Para cada arco $\{i, j\} \in E$: $X_i \neq X_j$

Coloreado de grafos

Pasos a seguir

1. Abrir el archivo *02_coloreado_basico.mzn*
2. El modelo tiene un parámetro k (con el valor 4).
3. Definir un conjunto de enteros de nombre COLORES con los elementos $\{1, \dots, k\}$.
4. Definir una variable para cada vértice del grafo.
5. Definir una restricción para cada arco del grafo.
6. Resolverlo (*solve satisfy*) y analizar la solución.

```
% 02_coloreado_basico.mzn
```

```
% Datos
```

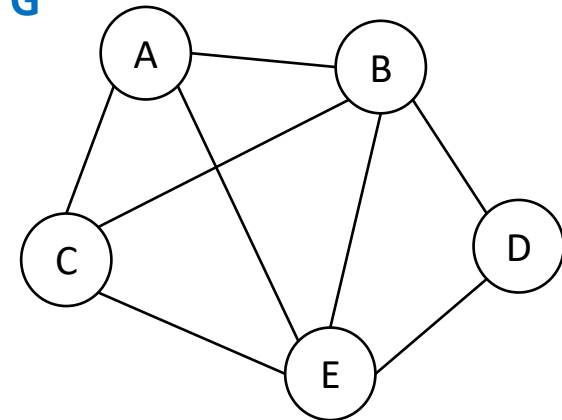
```
int: k = 4; % k es el número de colores
```

```
% Variables de decisión (una por cada variable)
```

```
% Restricciones
```

```
% Resolución
```

$k = 4$ **G**



Coloreado de grafos

Introducir datos en el modelo

El modelo tiene un parámetro $k = 4$.

Sería deseable poder introducir distintos valores sin necesidad de cambiar el modelo.

Para ello, sustituimos `int: k = 4;` por `int: k;` (lo dejamos libre).

Dos opciones:

1. Ejecutamos el programa: aparece una ventana pidiéndonos un valor para k (por teclado).
2. Proporcionamos el valor en un **archivo de datos**: Abrimos en MiniZinc IDE el archivo `02_k3.dzn`. Ejecutamos el programa y seleccionamos este archivo como entrada.

Probar con varios valores con varios valores de k .

```
% 02_coloreado_basico.mzn
```

```
% Datos
```

```
int: k = 4; % k es el número de colores
```

```
% Variables de decisión (una por cada variable)
```

```
% Restricciones
```

```
% Resolución
```

```
% 02_k3.dzn
```

```
k = 3;
```

Coloreado de grafos

The screenshot shows the Z3 GUI interface. The main window displays a file named `coloreado_basico.mzn` with the following code:

```
1 % coloreado_basico.mzn
2
3 % Datos
4 int: k;
5
6 set of int: COLORES = 1..k;
7
8 % Variables de decisión
9 var COLORES: A;
10 var COLORES: B;
11 var COLORES: C;
12 var COLORES: D;
13 var COLORES: E;
14
15 % Restricciones
16 constraint A != B;
17 constraint A != C;
18 constraint A != E;
19 constraint B != C;
20 constraint B != D;
21 constraint B != E;
22 constraint C != E;
23 constraint D != E;
24
25
26 % Resolución
27 solve satisfy;
28
```

The right-hand pane shows the **Configuration** editor. The **Solver configuration** is set to `Gecode 6.3.0 *`. The **Solver** is `Gecode 6.3.0`. The **Options** section includes:

- ☒ Maintain these options across solver configurations
- Solving**
 - ☐ Time limit: 0,000s
 - ☐ Default behavior
 - ☒ User defined behavior
- Optimization problems**
 - ☒ Print intermediate solutions
 - ☒ Stop after this many optimal solutions (uncheck for all): 1
- Satisfaction problems**
 - ☐ Stop after this many solutions (uncheck for all): 1

The status bar at the bottom indicates `Line: 24, Col: 1` and `115msec`.

Enumerar todas las soluciones
(solo problemas sin optimización)

Abrir "Show configuration editor"

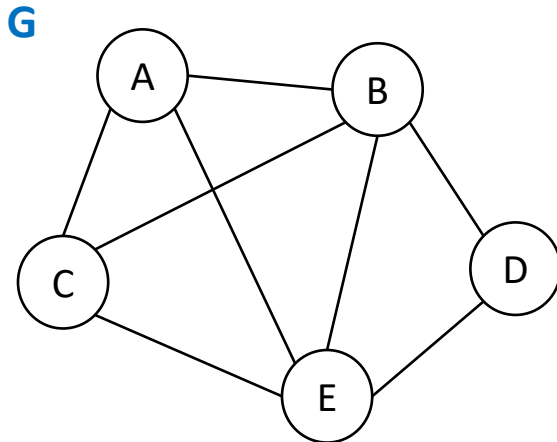
Marcar

Desmarcar

Coloreado de grafos (optimización)

Dado un grafo no dirigido $G = (V, E)$. Dar un coloreado de G que utilice el mínimo número de colores posible (**número cromático**).

Crear un modelo para la siguiente instancia del problema:



Modelo CSOP

- Variables: $\mathbf{X} = \{X_v, v \text{ en } V\}$
- Dominios: $\text{dom}(X_v) = \{1, \dots, |V|\}$, para todo $X_v \in \mathbf{X}$
- Restricciones:
 - Para cada arco $\{i, j\} \in E$: $X_i \neq X_j$
- **Objetivo:** minimizar $f(\mathbf{X}) = \max(\mathbf{X})$

Coloreado de grafos (optimización)

Pasos a seguir

1. Abrir el archivo *03_coloreado_basico_opt.mzn*
2. El modelo tiene un parámetro *n* (número de vértices del grafo).
3. En este caso definimos $\text{COLORES} = \{1, \dots, n\}$ (en general no sabemos cuántos harán falta, pero *n* sería suficiente).
4. Hay las mismas variables y restricciones que teníamos antes.
5. Resolución: minimizamos el valor máximo de las 5 variables (el argumento es un array de variables – estudiaremos arrays en la siguiente sesión).
6. Resolverlo y analizar la solución.
7. Probar a quitar o añadir arcos al grafo.

```
% 03_coloreado_basico_opt.mzn

% Parámetros / Datos
int: n = 5; % tenemos n=5 vértices
set of int: COLORES = 1..n; % n colores posibles

% Variables de decisión
var COLORES: A; var COLORES: B;
var COLORES: C; var COLORES: D;
var COLORES: E;

% Restricciones
constraint A != B; constraint A != C;
constraint A != E; constraint B != C;
constraint B != D; constraint B != E;
constraint C != E; constraint D != E;

% Resolución
solve minimize max([A,B,C,D,E]);
```

MiniZinc

Tipos enumerados (`enum`):

- Conjunto de elementos donde a cada uno se le da un nombre específico al que nos podemos referir posteriormente.
- En la práctica, estos elementos se convierten en números enteros: 1, 2, ...
- Ejemplo: `enum COLOR = {rojo, verde, azul};`

Operadores lógicos

- Podemos crear restricciones complejas mediante operadores lógicos.
- Disyunción (`\ /`), conjunción (`/ \`), negación (`not`), implicación (`->`), doble implicación (`<->`).
- Ejemplo: `constraint x < 5 -> y = z;`

Ejemplo

Pasos a seguir

1. Abrir el archivo *04_semaforos.mzn*
2. El problema es encontrar un color de luz para cada semáforo de modo que se cumplan todas las restricciones.
3. Observar el programa (y su sintaxis)
 - Tipo enumerado COLOR ({rojo, verde, ambar}).
 - Variables de decisión (con dominio COLOR).
 - Restricciones entre los tres semáforos.
4. Ejecutar el programa (**Run**) y observar la salida.

```
% 04_semaforos.mzn
```

```
% Parámetros / Datos
```

```
enum COLOR = {verde, rojo, ambar}; % tres colores
```

```
% Variables de decisión
```

```
var COLOR: S1; % S1: color del primer semáforo
```

```
var COLOR: S2; % S2: color del segundo semáforo
```

```
var COLOR: S3; % S3: color del tercer semáforo
```

```
% Restricciones
```

```
% S1 está en verde o S3 está en rojo (o ambas cosas)
```

```
constraint S1 = verde ∨ S3 = rojo;
```

```
% No es posible que S1 y S2 estén en rojo al mismo tiempo
```

```
constraint not(S1 = rojo ∧ S2 = rojo);
```

```
% Si S2 está en verde, entonces S3 está en rojo
```

```
constraint S2 = verde -> S3 = rojo;
```

```
% S1 está en ámbar si y solo si S2 está en verde
```

```
constraint S1 = ambar <-> S2 = verde;
```

```
% Resolución
```

```
solve satisfy;
```

Ejercicio

En un restaurante se ofrece un **menú** en el que se puede elegir entre las siguientes opciones:

- Entrante: calamares, croquetas o ensalada
- Plato principal: entrecot, lubina o menestra
- Postre: fruta o helado

Un cliente (muy exigente) llega al restaurante con las siguientes **restricciones**:

1. El entrante debe ser croquetas o el postre helado (o ambas cosas).
2. Si el entrante es ensalada, el plato principal tiene que ser lubina.
3. Si el entrante es croquetas, el plato principal es entrecot o menestra.
4. El postre es helado si y solo si el plato principal es entrecot.
5. No puede ser que el entrante sea calamares y el plato principal entrecot.

¿Existe alguna posibilidad de menú para el cliente?

Ejercicio

Pasos a seguir

1. Abrir el archivo `05_menu.mzn`
2. Completar el programa añadiendo las restricciones
3. Ejecutar el programa (**Run**) para determinar si hay alguna solución

```
% 05_menu.mzn
```

```
% Parámetros / Datos
```

```
% Un tipo enumerado para cada plato (ENTRANTES, PRINCIPALES, POSTRES)
```

```
enum ENTRANTES = {calamares, croquetas, ensalada};
```

```
enum PRINCIPALES = {entrecot, lubina, menestra};
```

```
enum POSTRES = {fruta, helado};
```

```
% Variables de decisión
```

```
var ENTRANTES: Entrante; % Entrante será uno de los entrantes
```

```
var PRINCIPALES: Principal; % Principal será un plato principal
```

```
var POSTRES: Postre; % Postre será uno de los postres
```

```
% Restricciones
```

```
% El entrante es croquetas o el postre helado (o las dos cosas)
```

```
% Si el entrante es ensalada, el plato principal es lubina
```

```
% Si el entrante es croquetas, el plato principal es entrecot o menestra
```

```
% El postre es helado si y solo si el plato principal es entrecot
```

```
% No puede ser que el entrante sea calamares y el plato principal entrecot
```

```
% Resolución
```

```
solve satisfy;
```

Contenidos

1. Problemas de satisfacción de restricciones y optimización
- 2. El lenguaje MiniZinc**
 - Introducción y elementos básicos
 - **Arrays**
 - Aspectos avanzados
3. Modelado de problemas de scheduling

Arrays en MiniZinc

- Cuando construimos un modelo en MiniZinc, nos interesa poder resolver **cualquier instancia** de un problema
- No conocemos a priori el tamaño de la entrada (por ejemplo, el número de vértices y arcos en un grafo), ni los datos
- Solución: emplear **arrays**
 - Para los datos de entrada
 - Para las variables de decisión

Arrays en MiniZinc

Ejemplo

Pasos a seguir

1. Abrir el archivo *01_arrays_ejemplo.mzn*
2. Observar el programa (y su sintaxis)
 - Declaración de un array de datos
 - Declaración de un array de variables de decisión
 - Indexación de los arrays
 - Restricciones `forall`
3. Ejecutar el programa y observar la salida.
4. Dejar *libres* `n` y `a`:
 - `int: n`
 - `array[1..n] of int: a;`
5. Abrir el archivo *01_arrays_instancia_1.dzn* y ejecutar el programa proporcionándole dicha entrada.
6. Probar también con *01_arrays_instancia_2.dzn*

```
% 01_arrays_ejemplo.mzn
```

```
% Parámetros / Datos
```

```
int: n = 5; % Tendremos 5 elementos
```

```
% array de n enteros
```

```
% se indexa desde 1: a[1], a[2],..., a[n]
```

```
array[1..n] of int: a = [5, 12, 8, 9, 10];
```

```
% Variables de decisión
```

```
% array de variables de decisión con dominios {0,...,25}
```

```
array[1..n] of var 0..25: Res;
```

```
% Restricciones
```

```
% El primer elemento de Res debe ser mayor que el último
```

```
constraint Res[1] > Res[n];
```

```
% La suma de los elementos 2 y 3 debe ser igual al penúltimo
```

```
constraint Res[2] + Res[3] = Res[n-1];
```

```
% Res[i] debe ser mayor o igual que a[i], para todo i
```

```
constraint forall(i in 1..n)(Res[i] >= a[i]);
```

```
% Todos los elementos deben de ser diferentes entre sí
```

```
constraint forall(i,j in 1..n where i < j)(Res[i] != Res[j]);
```

```
% Resolución
```

```
solve satisfy;
```

```
% Descomentar para ver el efecto
```

```
% output["Resultado: " ++ show(Res)];
```


Arrays en MiniZinc

forall

`forall` expresa la **conjunción** de varias restricciones (impone que se cumplan **todas**)

`forall(i in 1..3) (Res[i] > 5)` **equivalente a** `Res[1] > 5 /\ Res[2] > 5 /\ Res[3] > 5`

Podemos indicar condiciones mediante `where`

- Todas las variables en posiciones pares del array son mayores que 10

```
constraint forall(i in 1..n where i mod 2 = 0) (Res[i] > 10);
```

- Todas las variables del array son distintas dos a dos

```
constraint forall(i,j in 1..n where i < j) (Res[i] != Res[j]);
```

```
constraint forall(i in 1..(n-1)) (  
    forall(j in (i+1)..n) (  
        Res[i] != Res[j]  
    )  
);
```

Arrays en MiniZinc

`exists`

`exists` expresa la **disyunción** de varias restricciones (impone que se cumpla **al menos una**)

`exists(i in 1..3) (Res[i] > 5)` **equivalente a** `Res[1] > 5 \/ Res[2] > 5 \/ Res[3] > 5`

Podemos indicar condiciones mediante `where`

- Alguna variable en una posición par del array es mayor que 10

```
constraint exists(i in 1..n where i mod 2 = 0) (Res[i] > 10);
```

- Al menos dos variables del array son distintas entre sí

```
constraint exists(i,j in 1..n where i < j) (Res[i] != Res[j]);
```

Arrays en MiniZinc

Funciones de agregación

sum

- La suma de las variables del array es mayor que 50

```
constraint sum(i in 1..n) (Res[i]) > 50;
```

```
constraint sum(Res) > 50;
```

```
constraint sum([Res[i] | i in 1..n]) > 50;
```

También podemos emplear
comprensiones de arrays

- La suma de las variables en posiciones pares del array es mayor que 50

```
constraint sum(i in 1..n where i mod 2 = 0) (Res[i]) > 50;
```

```
constraint sum([Res[i] | i in 1..n where i mod 2 = 0]) > 50;
```

Arrays en MiniZinc

Funciones de agregación

max y min

- El valor máximo/mínimo de cualquier variable del array es menor que 30

```
constraint max(i in 1..n) (Res[i]) < 30;
```

```
constraint max(Res) < 30;
```

```
constraint min(i in 1..n) (Res[i]) < 30;
```

```
constraint min(Res) < 30;
```

product

- El producto de las variables en la primera mitad del array es igual a 100

```
constraint product(i in 1..(n div 2)) (Res[i]) = 100;
```

Arrays en MiniZinc

Funciones de agregación

count

- El valor 3 se asigna exactamente a 2 variables del array

```
constraint count(Res, 3) = 2;
```

```
constraint count(i in 1..n) (Res[i] = 3) = 2;
```

Res[i] = 3 se evalúa a verdadero o falso
count cuenta las veces que es verdadero

Con sum

```
constraint sum(i in 1..n) (Res[i] = 3) = 2;
```

Res[i] = 3 se evalúa a verdadero o falso,
y se convierte automáticamente a 1 o 0

```
constraint sum(i in 1..n) (bool2int(Res[i]) = 3) = 2;
```

Aquí hacemos la conversión explícitamente

El problema de la mochila (0-1)

Se tiene una mochila con capacidad para llevar un **peso máximo W** y un conjunto de n **objetos** $O = \{O_1, \dots, O_n\}$. Cada objeto O_i tiene un peso $w_i > 0$ y un valor $v_i > 0$. Seleccionar qué objetos llevar en la mochila de modo que no se exceda su capacidad y se **maximice el valor total de los objetos** seleccionados.

Ejemplo: $W = 20$, 4 objetos:

O_i	w_i	v_i
O_1	8	10
O_2	4	8
O_3	10	12
O_4	7	9

$\{O_1, O_2, O_3\}$ **no es una solución** (excede W)





$\{O_3, O_4\}$ es una solución, con valor total 21

$\{O_1, O_2, O_4\}$ es una **solución óptima**, con valor total 27

El problema de la mochila (0-1)

Se tiene una mochila con capacidad para llevar un peso máximo W y un conjunto de n objetos $O = \{O_1, \dots, O_n\}$. Cada objeto O_i tiene un peso $w_i > 0$ y un valor $v_i > 0$. Seleccionar qué objetos llevar en la mochila de modo que no se exceda su capacidad y se maximice el valor total de los objetos seleccionados.

Modelo CSOP

- Variables: $X = \{S_i \mid i \in \{1, \dots, n\}\}$ 
 - Dominios: $dom(S_i) = \{0, 1\}$, para todo $S_i \in X$ 
 - Restricciones:
 - Capacidad: $\sum_{i=1}^n w_i * S_i \leq W$ 
 - Objetivo: maximizar $f(X)$
 - $f(X) = \sum_{i=1}^n v_i * S_i$ 
- Una variable S_i asociada a cada objeto O_i
- $S_i = 1$: objeto O_i seleccionado
 $S_i = 0$: objeto O_i no seleccionado
- El peso de los objetos seleccionados no puede exceder W
- Maximizamos el valor total de los objetos seleccionados

El problema de la mochila (0-1)

Pasos a seguir (1)

1. Abrir el archivo *02_mochila_0-1.mzn*
2. Observar los datos de entrada: *n* es el número de objetos, *W* es la capacidad máxima de la mochila y los arrays *peso* y *valor* contienen dicha información para cada objeto.
3. El modelo tiene una variable de decisión de nombre *ValorTotal*, que será igual a la suma del valor de los objetos seleccionados.
4. Añadir un array con las variables de decisión restantes: una por cada objeto, indicando si se selecciona o no (dominio 0..1).
5. Completar el modelo añadiendo las restricciones según se indica en los comentarios.
6. Indicar que se maximice *ValorTotal* en la resolución.
7. Ejecutar el programa y observar la salida.

```
% 02_mochila_0-1.mzn

% Parámetros / Datos
int: n = 4; % número de objetos
int: W = 20; % capacidad máxima de la mochila
array[1..n] of int: peso = [8,4,10,7]; % peso de los objetos
array[1..n] of int: valor = [10,8,12,9]; % valor de los objetos

% Variables de decisión
% ValorTotal (función objetivo)
% Dominio: desde 0 a una cota superior del valor óptimo
var 0..sum(valor): ValorTotal; % (observar dominio)

% Añadir array de variables con dominio 0..1 (una por objeto)
% Llamarlo Sel

% Restricciones
% El peso total de objetos seleccionados no excede la capacidad

% ValorTotal es la suma del valor de los objetos seleccionados

% Resolución: maximizar ValorTotal

output["Selección: " ++ show(Sel) ++ "\n" ++
       "Valor total: " ++ show(ValorTotal)];
```


El problema de la mochila (0-1)

Pasos a seguir (2)

1. Dejar libres los datos de entrada.
2. Estos están en el archivo de datos de nombre *02_mochila_0-1_i1.dzn*
3. Crear un nuevo archivo “New model”, y especificar en él los datos para la instancia de la derecha.
4. Guardarlo como un archivo de tipo .dzn (MiniZic data) con el nombre *02_mochila_0-1_i2.dzn*
5. Ejecutar el programa con dicha entrada y observar la salida.

$W = 30$, 6 objetos:

O_i	w_i	v_i
O_1	4	7
O_2	9	10
O_3	11	13
O_4	15	14
O_5	6	5
O_6	5	6

Arrays multidimensionales

- MiniZinc permite el uso de **arrays multidimensionales**
- En muchos modelos se emplean matrices de variables, por lo que veremos cómo emplear arrays de **dos dimensiones**
- Los presentaremos mediante un ejemplo (archivo *03_arrays_2d.mzn*).
 - Abrir dicho archivo y observarlo en detalle.
 - Prestar atención a la sintaxis y a las explicaciones que se incluyen en los comentarios.
 - Ejecutar el programa y ver el resultado.
 - En las transparencias que siguen, se comentan algunos de los aspectos más importantes.

Arrays 2D

Parámetros y datos

```
int: n = 3; % Tendremos n filas  
int: m = 2; % Tendremos m columnas  
  
% Matriz de enteros (n filas x m columnas)  
array[1..n,1..m] of int: matriz = [| 1, 2, | 3, 4, | 5, 6|];
```

Array de enteros de n x m (3 filas y 2 columnas)
(Las filas van de 1 a n y las columnas de 1 a m)

Empleamos | (barra vertical) al principio y al final del array, y también para separar las distintas filas

Definimos este array 2D

	c1	c2
f1	1	2
f2	3	4
f3	5	6

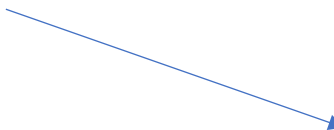
Arrays 2D

Variables de decisión

% Matriz (n x m) de variables con dominio 1..6
array[1..n,1..m] of var 1..6: MRes;

Array n x m (3 x 2) de variables con dominio 1..6
(Las filas van de 1 a n y las columnas de 1 a m)

Definimos este array 2D de variables



	c1	c2
f1	MRes[1, 1]	MRes[1, 2]
f2	MRes[2, 1]	MRes[2, 2]
f3	MRes[3, 1]	MRes[3, 2]

Arrays 2D

Restricciones

% El elemento (1,1) de MRes es mayor que el (1,2)
`constraint MRes[1,1] > MRes[1,2];`


% En la última columna, el primer elemento es menor que el último
`constraint MRes[1,m] < MRes[n,m];`

% MRes[i,j] es distinto de matriz[i,j], para todo i y j
`constraint forall(i in 1..n, j in 1..m)(MRes[i,j] != matriz[i,j]);`


% La suma de los elementos de la primera fila es igual 10
`constraint sum(j in 1..m)(MRes[1,j]) = 10;`

% La suma de los elementos de la primera columna es igual a 12
`constraint sum(i in 1..n)(MRes[i,1]) = 12;`


Para referirnos al elemento (i,j), es decir, fila i y columna j, empleamos `MRes[i,j]`




El índice i recorre las filas (1..n) y el j las columnas (1..m)



Fijamos la fila a 1 y recorremos las columnas con j (1..m)



Fijamos la columna a 1 y recorremos las filas con i (1..n)



Contenidos

1. Problemas de satisfacción de restricciones y optimización
- 2. El lenguaje MiniZinc**
 - Introducción y elementos básicos
 - Arrays
 - **Aspectos avanzados**
3. Modelado de problemas de scheduling

Cuadrado latino

Un **cuadrado latino** es una matriz de tamaño $n \times n$ en la que cada elemento es un número de 1 a n y tal que cada uno de ellos aparece exactamente una vez en cada fila y en cada columna.

2	1	3
1	3	2
3	2	1

$n = 3$

3	1	2	4	5
1	2	5	3	4
2	5	4	1	3
4	3	1	5	2
5	4	3	2	1

$n = 5$

Cuadrado latino

Primer modelo

Pasos a seguir

1. Abrir el archivo *01_cuadrado_latino.mzn*
2. Observar los datos de entrada: n es el número de filas y columnas del cuadrado.
3. El modelo tiene una matriz de $n \times n$ variables de decisión, de nombre `Cuadrado`. Cada variable tiene el dominio $1 \dots n$.
4. Las restricciones se imponen dos a dos mediante el uso de `forall`. Primero para las filas y después para las columnas.
5. Configurar Gecode para limitar el tiempo a 20 segundos y para que muestre estadísticas de resolución (ver la siguiente transparencia).
6. Ejecutar el programa y observar la salida. Ver hasta qué valor de n se puede resolver dentro del tiempo límite.

```
% 01_cuadrado_latino.mzn

% Parámetros / Datos
int: n = 3; % el cuadrado será de nxn

% Variables de decisión
% Matriz de nxn variables con dominio 1..n
array[1..n,1..n] of var 1..n: Cuadrado;

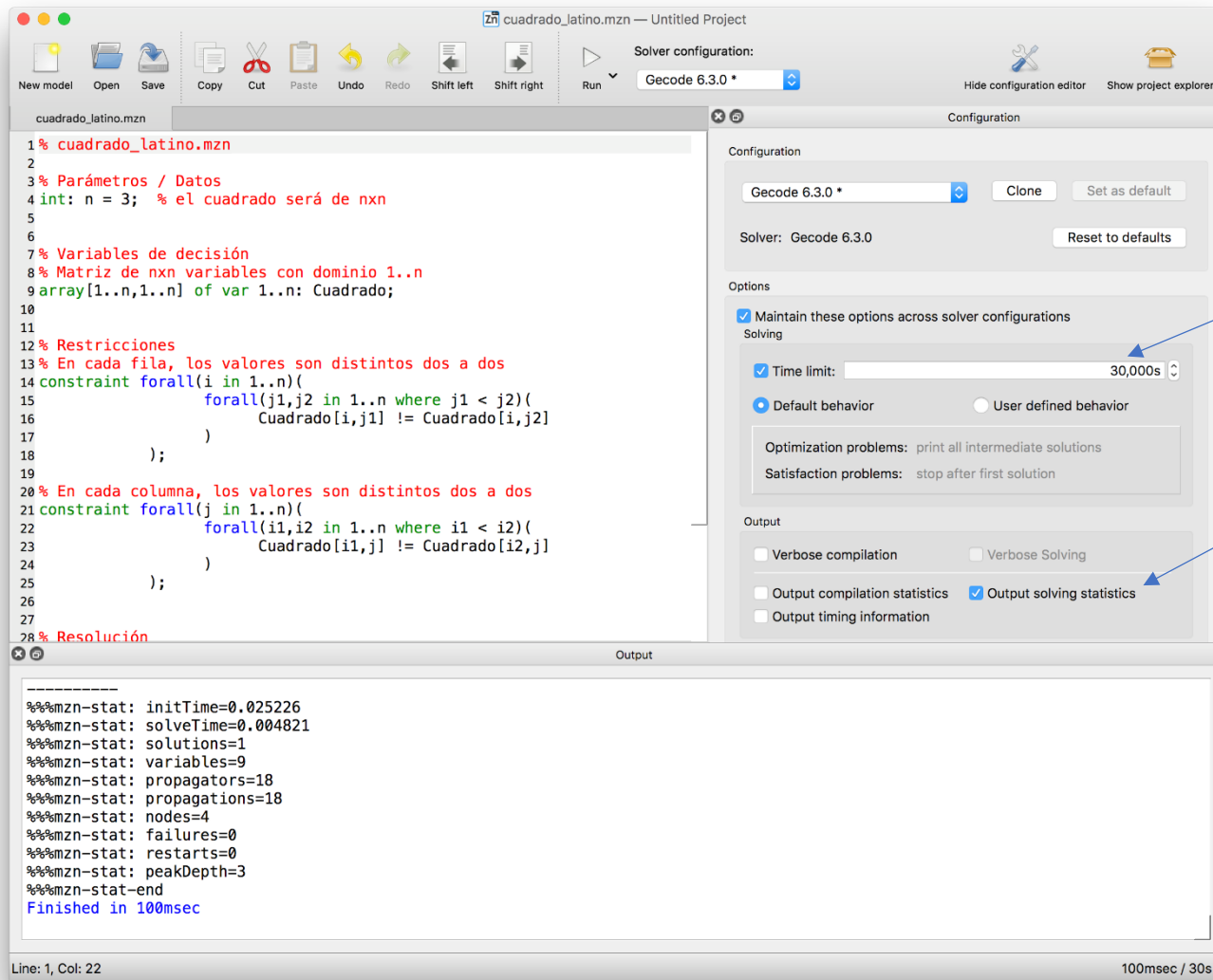
% Restricciones
% En cada fila, los valores son distintos dos a dos
constraint forall(i in 1..n)(
    forall(j1,j2 in 1..n where j1 < j2)(
        Cuadrado[i,j1] != Cuadrado[i,j2]
    )
);

% En cada columna, los valores son distintos dos a dos
constraint forall(j in 1..n)(
    forall(i1,i2 in 1..n where i1 < i2)(
        Cuadrado[i1,j] != Cuadrado[i2,j]
    )
);

% Resolución
solve satisfy;

% Salida
output[show(Cuadrado[i,j]) ++
    if j = n then "\n" else " " endif | i,j in 1..n];
```


Configuración de Gecode



Marcar la casilla “Time limit” e introducir el tiempo límite

Marcar la casilla “Output solving statistics”

Estadísticas:

- `solveTime`: tiempo de resolución (segundos)
- `propagations`: número de propagaciones
- `nodes`: nodos visitados
- `failures`: número de *retrocesos*
- `restarts`: número de reinicios
- `peakDepth`: profundidad máxima alcanzada

La restricción global *alldifferent*

La **restricción global *alldifferent*** permite imponer que todas las variables de un array tomen valores diferentes entre sí.

- Normalmente es más eficiente que imponer las restricciones dos a dos.
- Para emplearla es necesario añadir al inicio: `include "alldifferent.mzn";`

Ejemplos

- Todas las variables del array *A* (indexado en $1..n$) son diferentes dos a dos

```
constraint alldifferent([A[i] | i in 1..n]);
```

- Las variables del array *A* (indexado en $1..n$) con índice par son diferentes entre sí

```
constraint alldifferent([A[i] | i in 1..n where i mod 2 = 0]);
```

Cuadrado latino

Modelo con *alldifferent*

Pasos a seguir (1)

1. Abrir el archivo
`02_cuadrado_latino_alldifferent.mzn`
2. Completar las restricciones empleando *alldifferent* (observar que se ha incluido `alldifferent.mzn`).
3. Configurar Gecode para limitar el tiempo a 30 segundos y para que muestre estadísticas de resolución.
4. Ejecutar el programa variando el valor de n y observar la salida. En este punto, emplear solamente `solve satisfy`.

```
% 02_cuadrado_latino_alldifferent.mzn

include "alldifferent.mzn"; % incluimos alldifferent.mzn
% Parámetros / Datos
int: n = 3; % el cuadrado será de nxn

% Variables de decisión
% Matriz de nxn variables con dominio 1..n
array[1..n,1..n] of var 1..n: Cuadrado;

% Restricciones
% En cada fila, los valores son distintos dos a dos

% En cada columna, los valores son distintos dos a dos

% Resolución
solve satisfy;

% Salida
output[show(Cuadrado[i,j]) ++
  if j = n then "\n" else " " endif | i,j in 1..n];
```

Contenidos

1. Problemas de satisfacción de restricciones y optimización
2. El lenguaje MiniZinc
3. **Modelado de problemas de scheduling**

Modelado de problemas de scheduling

Veremos distintos elementos y herramientas para modelar problemas de scheduling en el marco de los CSPs y CSOPs

En general:

- Las tareas / actividades tienen asociadas **variables de decisión** (p.ej. tiempo de inicio, ...).
- Los recursos dan lugar a **restricciones** (p.ej., imposibilidad de procesar dos tareas a la vez, ...).
- Puede haber restricciones sobre una o varias tareas independientemente de los recursos (p.ej. restricciones de precedencia, ...).
- Es una buena práctica ajustar los dominios de las variables inicialmente, con cuidado de no perder soluciones relevantes (p.ej. mediante el cálculo de cotas de la función objetivo).

Tareas / Actividades

Las tareas suelen tener una **duración** o **tiempo de procesamiento** (p_i)

Variables de decisión:

start_i: Tiempo en el que comienza la ejecución de la tarea i .

end_i: Tiempo en el que termina la ejecución de la tarea i .

Tareas no interrumpibles

$$\text{start}_i + p_i = \text{end}_i$$

Dominios de las variables:

start_i: $[0, H - p_i]$

end_i: $[p_i, H]$



H es el **horizonte de planificación**

El tiempo de fin de todas las tareas no puede exceder H

Cabezas y deadlines

En ocasiones, una tarea puede llegar al sistema en un instante dado y/o debe terminar de procesarse antes de un tiempo límite determinado.

Dada una tarea i :

- **Cabeza** o *release date* (r_i): tiempo de inicio más temprano posible.
- **Deadline** o *due date* (d_i): tiempo de fin más tardío posible.

Las deadlines pueden ser restricciones **duras** o **blandas** (empleadas en la función objetivo)

Se pueden modelar mediante **restricciones unarias**:

$$\text{start}_i \geq r_i$$

$$\text{start}_i \leq d_i - p_i$$

$$\text{end}_i \geq r_i + p_i$$

$$\text{end}_i \leq d_i$$

También se puede **acotar el dominio** de las variables start_i y end_i :

$$\text{dom}(\text{start}_i) = \{r_i, \dots, d_i - p_i\}$$

$$\text{dom}(\text{end}_i) = \{r_i + p_i, \dots, d_i\}$$

Optimización

Habitualmente no sólo nos interesa satisfacer restricciones (CSPs), sino que además hay algún criterio o **función objetivo a optimizar** (CSOPs)

Métricas típicas:

- Makespan: máximo tiempo de fin de cualquier tarea.
- Tiempo de flujo total: suma del tiempo de fin de las tareas.
- Tardiness máximo: retraso máximo de cualquier tarea respecto a su deadline (blanda).
- Tardiness total: suma del retraso de todas las tareas respecto a su deadline (blanda).
- Número de *tardy jobs*: Número de tareas que terminan después de su deadline (blanda).
- ...

También son comunes versiones **ponderadas** de las funciones objetivo de tipo suma

Restricciones de precedencia

Relaciones de precedencia:

$A \rightarrow B$: la relación de precedencia entre tareas más común es la que impone que la tarea B no puede empezar a procesarse hasta que la tarea A termine.

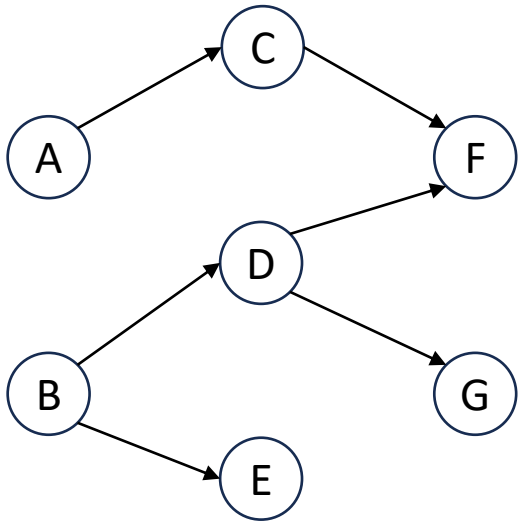
$$\mathbf{start_A + p_A \leq start_B} \quad (\text{equivalentemente } \mathbf{end_A \leq start_B})$$

Existen otros tipos de relaciones de precedencia.

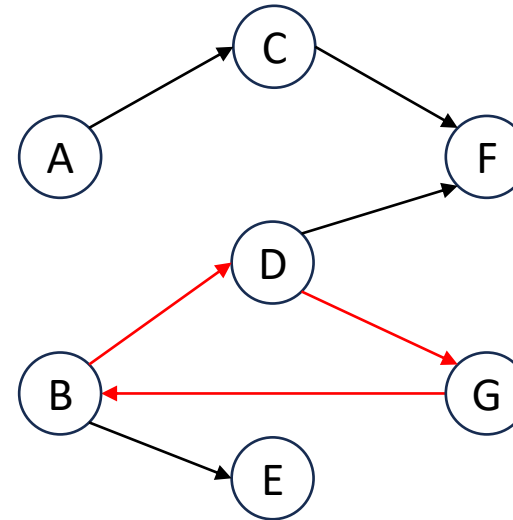
Restricciones de precedencia

Grafo de precedencia

Las relaciones de precedencia suelen expresarse mediante de grafos de precedencia



Los vértices representan tareas y los arcos relaciones de precedencia



Si hay ciclos en el grafo, el problema
no es satisfacible

Recursos

Comúnmente, las **tareas compiten por recursos**

En una fábrica, se puede necesitar una misma herramienta para más de una tarea

Al hacer un calendario de exámenes, no puede ser que un profesor esté en más de un examen al mismo tiempo

Si en un taller hay 5 trabajadores, y cada uno de ellos sólo puede desarrollar una tarea en cada momento, como mucho se pueden desarrollar 5 tareas a la vez

...

Tipos de recursos

Unarios

Acumulativos

Recursos unarios

Un **recurso unario** sólo puede estar ocupado por una tarea en cada momento

Esto significa que se debe imponer que, de todas las tareas que hacen uso del mismo recurso, haya a lo sumo una ejecutándose en cada momento

Opciones para modelado:

- Disyunciones

- Restricción global disjunctive

Recursos unarios - Modelado

Se puede modelar por medio de **disyunciones**

Si las tareas A y B necesitan ocupar un mismo recurso durante su ejecución, o bien A se ejecuta antes que B o bien B se ejecuta antes que A:

$$\text{start}_A + p_A \leq \text{start}_B \vee \text{start}_B + p_B \leq \text{start}_A$$

Se necesita una “cláusula” por cada par de tareas que utilicen una máquina

En total, si tenemos n tareas que compiten por un recurso, tendremos $(n^2-n)/2$ cláusulas.

Recursos unarios - Modelado

Restricción global *disjunctive*

También conocida como no_overlap.

Cuenta con sus propios mecanismos de propagación de restricciones, que la hacen una opción muy eficiente.

Se utiliza una restricción disjunctive por cada recurso unario.

Si A y B necesitan el mismo recurso: $disjunctive([start_A, start_B], [p_A, p_B])$

Restricción global *disjunctive* - Sintaxis

```
disjunctive(start, duracion)
```

start: Array de variables de decisión start (una por cada tarea que haga uso del recurso)

duracion: Array con las duraciones de las tareas que hacen uso del recurso.

Recursos unarios (1)

Pasos a seguir

1. Abrir el archivo *01_disjunctive_1.mzn*
2. Observar (y tratar de entender) el modelo
3. Completar el modelo empleando la restricción global *disjunctive*
 1. Incluir “disjunctive.mzn”.
 2. Imponer una restricción *disjunctive* para representar el recurso M2.

```
% 01_disjunctive_1.mzn
```

```
% H: Horizonte de planificación
```

```
int: H = 250;
```

```
% pi: Duración de la tarea i
```

```
int: pA = 25; int: pB = 22; int: pC = 50;
```

```
int: pD = 37; int: pE = 13; int: pF = 20; int: pG = 45;
```

```
% starti: Tiempo de inicio de la tarea i
```

```
var 0..H-pA: startA;
```

```
var 0..H-pB: startB;
```

```
var 0..H-pC: startC;
```

```
var 0..H-pD: startD;
```

```
var 0..H-pE: startE;
```

```
var 0..H-pF: startF;
```

```
var 0..H-pG: startG;
```

```
% Tenemos 2 recursos unarios, M1 y M2
```

```
% - Las tareas C, D y G necesitan hacer uso de M1 durante su ejecución
```

```
constraint startC + pC <= startD ∨ startD + pD <= startC;
```

```
constraint startD + pD <= startG ∨ startG + pG <= startD;
```

```
constraint startC + pC <= startG ∨ startG + pG <= startC;
```

```
% - Las tareas A, B, E y F necesitan hacer uso de M2 durante su ejecución
```

```
% COMPLETAR
```

```
% CSP
```

```
solve satisfy;
```


Recursos unarios (2)

Pasos a seguir

1. Abrir el archivo *02_disjunctive_2.mzn*
2. Observar (y tratar de entender) el modelo
3. Completar el modelo empleando la restricción global *disjunctive* para modelar los recursos.
4. Ejecutar el modelo con las instancias *02_d_20_5.dzn* y *02_d_100_10.dzn*.

```
% 02_disjunctive_2.mzn
```

```
include "disjunctive.mzn";
```

```
% H: Horizonte de planificación
```

```
int: H;
```

```
% n_tareas: Número de tareas
```

```
int: n_tareas;
```

```
% n_recursos: Número de recursos
```

```
int: n_recursos;
```

```
% p: Array de duraciones
```

```
array[1..n_tareas] of int: p;
```

```
% r: Array de recursos requeridos por cada tarea
```

```
array[1..n_tareas] of int: r;
```

```
% start: array de variables de decisión start
```

```
array[1..n_tareas] of var 0..H: start;
```

```
% end: array de variables de decisión end
```

```
array[1..n_tareas] of var 0..H: end;
```

```
% Hay que relacionar start y end (start[i] + p[i] = end[i])
```

```
constraint forall(i in 1..n_tareas)(start[i] + p[i] = end[i]);
```

```
% Tan solo puede haber una tarea de cada recurso ejecutándose en cada momento
```

```
% COMPLETAR
```

```
% Objetivo: minimización del makespan
```

```
solve minimize max(i in 1..n_tareas)(start[i]+p[i]);
```

Recursos acumulativos

Los *recursos acumulativos* pueden procesar más de una tarea en cada momento, siempre y cuando no se exceda su *capacidad*.

Cada tarea que haga uso de un recurso acumulativo tiene un consumo asociado a ella, y este se resta de la capacidad del recurso durante todo su tiempo de procesamiento.

Tareas no interrumpibles.

Se pueden modelar con la restricción global *cumulative*.

Restricción global *cumulative* - Sintaxis

```
cumulative(start, duracion, consumo, capacidad)
```

start: Array de variables de decisión start (una por cada tarea que haga uso del recurso)

duracion: Array con las duraciones de las tareas que hacen uso del recurso.

consumo: Array con el consumo de cada tarea que hace uso del recurso.

capacidad: Capacidad del recurso (entero)