



UNIVERSIDAD  
DE GRANADA



# Técnicas de los Sistemas Inteligentes

Grado en Ingeniería Informática

**Curso 2020-21. Seminario 2**

**Satisfacción de Restricciones**

Jesús Giráldez Crú y Pablo Mesejo Santiago

**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**

**<http://decsai.ugr.es>**

En la práctica anterior hemos trabajado “implícitamente” con restricciones. Por ejemplo:

- No pasar cerca de un enemigo
- Si gemas < 10 → Buscar gema
- Si gemas ≥ 10 → Ir al portal
- No pasar por debajo de una roca (\*)
- etc.



(\*) En el juego original, para desplazarse por el mapa hay que cavar túneles. Cuando se realiza una excavación bajo una roca, ésta cae sobre el avatar y lo mata, terminando así el juego. Por tanto, en el juego original hay que evitar este tipo de movimientos.

La **satisfacción de restricciones** es una de las **áreas clásicas de IA**

Otras áreas clásicas de IA podrían ser los problemas de *búsqueda*, los problemas de *planificación* o los problemas con *incertidumbre*, entre otros

Es común encontrar estos problemas en **numerosas aplicaciones de IA**, a menudo como parte de otros problemas más complejos: razonamiento espacial-temporal, controladores de sensores, concurrencia y sincronización, coherencia geométrica (visión por computador), consistencia de bases de datos, alineación de secuencias (bioinformática), planificación de producción, asignación de recursos (*scheduling*), diseño y verificación de circuitos electrónicos, seguridad criptográfica, etc.

Por ello, **resolverlos eficientemente** es una parte crucial en todas estas aplicaciones.

Un **problema de satisfacción de restricciones** (del inglés, **CSP**, constraint satisfaction problem) se suele expresar como una **tupla** con tres elementos:

- $X$  es un conjunto de **variables**
- $D$  es un conjunto de **dominios** para  $X$
- $C$  es un conjunto de **restricciones** sobre  $X$

Ejemplo: tenemos dos números enteros positivos de forma que uno es mayor que el otro. ¿Cómo se podría expresar este problema como un CSP?

Un **problema de satisfacción de restricciones** (del inglés, **CSP**, constraint satisfaction problem) se suele expresar como una **tupla** con tres elementos:

- $X$  es un conjunto de **variables**
- $D$  es un conjunto de **dominios** para  $X$
- $C$  es un conjunto de **restricciones** sobre  $X$

Ejemplo: tenemos dos números enteros positivos de forma que uno es mayor que el otro. ¿Cómo se podría expresar este problema como un CSP?

### Solución 1:

- $X = \{ x, y \}$
- $D = \{ \mathbb{Z}, \mathbb{Z} \}$
- $C = \{ x > 0, y > 0, x > y \}$

Es un AND !

### Solución 2:

- $X = \{ x, y \}$
- $D = \{ [1, \infty), [1, \infty) \}$
- $C = \{ x > y \}$

Dominios y restricciones son conceptos relacionados; los dominios se pueden acotar con restricciones

Un **problema de optimización de restricciones** (del inglés, **COP**) se suele expresar como una **tupla** con cuatro elementos:

- $\langle X, D, C \rangle$  como en los CSP
- $f$  es una **función de coste** sobre  $X$  a **minimizar/maximizar**

Ejemplo: tenemos dos números enteros positivos de forma que uno es mayor que el otro, y la suma de ambos es **mínima**.  
¿Cómo se podría expresar este problema como un COP?

**Solución:**

- $X = \{ x, y \}$
- $D = \{ \mathbb{Z}, \mathbb{Z} \}$
- $C = \{ x > 0, y > 0, x > y \}$
- $f = x + y$  (minimizar)

El **problema de coloreado de mapas** es otro ejemplo de problema de satisfacción de restricciones:

- **Variables:** las regiones del mapa
- **Dominios:** los posibles colores (comunes para todas las regiones)
- **Restricciones:** dos regiones limítrofes no pueden tener el mismo color

$$X = \{WA, NT, SA, Q, NSW, V, T\}$$

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

$$C1: WA \neq NT$$

$$C2: WA \neq SA$$

$$C3: NT \neq SA$$

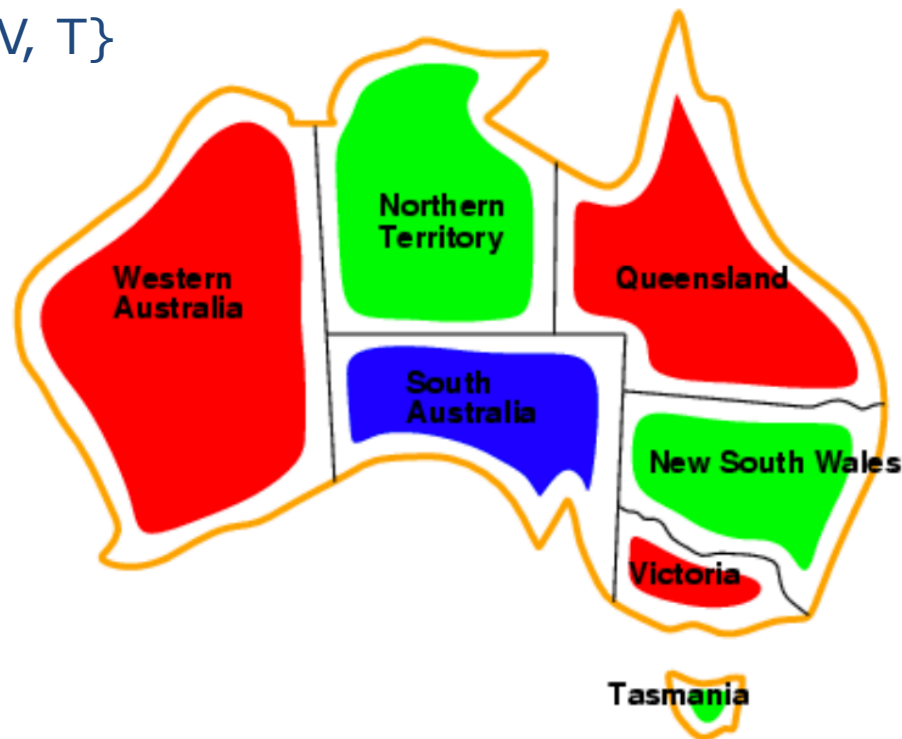
$$C4: NT \neq Q$$

$$C5: SA \neq Q$$

$$C6: SA \neq NSW$$

$$C7: SA \neq V$$

$$C8: Q \neq NSW$$

$$C9: NSW \neq V$$




Dada una ruta y un conjunto de casillas prohibidas, ¿cómo se podría definir este problema como un CSP?

**Pregunta:** ¿La ruta y las casillas prohibidas son variables? ¿Por qué?





Dada una ruta y un conjunto de casillas prohibidas, ¿cómo se podría definir este problema como un CSP?

**Pregunta:** ¿La ruta y las casillas prohibidas son variables? ¿Por qué?

## INPUT

R: conjunto de casillas de la ruta

P: conjunto de casillas prohibidas

NOTA: R y P no son variables, sino "constantes" (vienen dadas y son fijas para un time-step concreto)

## DEFINICIÓN DEL PROBLEMA

X: casillas del grid

D: {0,1}

C1:  $x = 1, \forall x \in X \cap R$

C2:  $x = 0, \forall x \in X \cap P$

Asigna el valor 1 a las casillas de la ruta

Asigna el valor 0 a las casillas prohibidas

## SOLUCIÓN

- Si la ruta no pasa por ninguna casilla prohibida, se satisfacen todas las restricciones (todas las casillas con un único valor)
- Si la ruta pasa por alguna casilla prohibida, esa casilla tendrá tanto valor "0" como valor "1" (lo cual viola alguna de las restricciones anteriores)

Existen muchas técnicas para resolver problemas de satisfacción de restricciones:

- **Constraint Programming (CP): Paradigma general** para resolver (cualquier) CSP. Las técnicas de resolución se basan principalmente en *búsqueda y propagación de restricciones* y/o técnicas de *búsqueda local* (estocástica).
- **Satisfiability (SAT)** En problemas que admiten la codificación binaria, suele ser un caso particular de CSP donde todas las variables tienen un **dominio binario** (true/false). SAT es el primer problema **NP-completo**. Se usan técnicas de resolución similares a las de CP, pero adaptadas a estos dominios específicos, que suelen ser más eficientes que esas técnicas “generales” de CP cuando el problema admite una codificación eficiente en dominios binarios (es decir, el problema de forma natural tiene variables Booleanas)
- **Linear Programming (LP): Sistema de inecuaciones lineales:**  

$$a \cdot x_1 + b \cdot x_2 + c \cdot x_3 + \dots + k \cdot x_n \geq C$$

Existe una interpretación geométrica del sistema que permite conocer si éste tiene solución.  
 → Algoritmo **SIMPLEX**

Existen muchas técnicas para resolver problemas de satisfacción de restricciones:

- **Pseudo-Boolean Constraints (PB)**: Caso particular de LP donde las variables tienen dominio binario. Es un área muy estudiada teóricamente (complejidad computacional) pero no existen implementaciones competitivas para este modelo
- **SAT-Modulo Theory (SMT)**: Llamadas iterativas a un SAT solver para problemas que requieren mayor expresividad que la lógica proposicional (=SAT). Ejemplo: **lógica de predicados** r orden.
- **Answer Set Programming (ASP)**: Método de programación declarativa basado en búsqueda, que garantiza terminación (planteado como alternativa a Prolog, que no la garantiza). Muy usado en problemas de representación de conocimiento (*knowledge representation*)

Hay tres grandes diferencias entre todas estas técnicas:

- **Codificación** del problema:
    - Dado un problema, ¿**cómo lo represento**?
      - ¿Qué variables hacen falta? ¿Qué restricciones se necesitan?
    - Una **representación correcta** es aquella capaz de encontrar TODAS las soluciones y rechazar TODO lo que no es solución
  - Algoritmos para **resolver el problema**
    - Dada una instancia, ¿cómo se resuelve?
    - En general todos aplican (alguna variante de algún método de) **búsqueda**, pero existen muchos métodos (búsqueda en profundidad, búsqueda local, etc.)
  - **Heurística** usada por el algoritmo de búsqueda
    - ¿Por dónde buscar?
- La heurística es normalmente considerada como uno de los componentes de cada algoritmo de resolución

Resolver CSP es **NP-completo** ...

... ¿y eso qué significa?

Si tengo un problema con “n” variables, **cualquier algoritmo** requiere **en el peor caso** “exp(n)” pasos para resolverlo. Es decir, el algoritmo tiene una **complejidad  $O(\exp(n))$**

Pero no siempre estamos *en el peor caso*

Por eso, la **codificación**, los **algoritmos de resolución** (y las **heurísticas**) usadas para resolver un determinado problema pueden **afectar considerablemente su resolución**. Veamos un ejemplo.

## Pigeon-Hole Principle: $\text{PHP}(n,k)$

$n$  pigeons:



$k$  holes:

1	2	3	...	k
---	---	---	-----	---

Asignar cada paloma a un palomar de forma que todo palomar tiene como máximo una paloma asignada

Es fácil ver que si  $n \leq k$ , entonces se puede satisfacer el problema (da igual con qué asignación). En cambio si  $n > k$ , entonces el problema es insatisfactible



Veamos dos **CODIFICACIONES** de  $PHP(n,k)$  para cualquier  $n,k > 0$

**Variables** (en ambas codificaciones):

- $n \cdot k$  variables Booleanas:
- $x(i,j)$  representa que la paloma "i" está en el palomar "j"

### Codificación SAT:

- Toda paloma está en al menos un palomar:  
 $x(i,1) \vee x(i,2) \vee \dots \vee x(i,k)$  . para toda paloma "i"
- Toda paloma está como máximo en un palomar, es decir, si una paloma está en un palomar, no está en otro  
 $x(i,j) \rightarrow !x(i,j')$  . para toda paloma "i" y palomares "  $j \neq j'$  "

### Codificación PB:

- Toda paloma está en al menos un palomar:  
 $x(i,1) + x(i,2) + \dots + x(i,k) \geq 1$  . para toda paloma "i"
- Todo palomar tiene como máximo una paloma:  
 $x(1,j) + x(2,j) + \dots + x(n,j) \leq 1$  . para todo palomar "j"

¿Cuánto se tarda en resolver estas dos fórmulas?



	PHP ( n+1 , n )	
n	SAT	PB
6	0.005277 s	0.004187 s
7	0.055557 s	0.005532 s
8	1.088930 s	0.006287 s
9	6.283010 s	0.007432 s
10	<b>99.242400 s</b>	0.008916 s
...		
100	?	0.128000 s
200	?	1.052000 s
300	?	3.760000 s
400	?	9.160000 s
500	?	<b>18.468000 s</b>

Se sabe que la codificación **SAT** es  **$O(\exp(n))$**  mientras que la codificación **PB** es  **$O(n)$**

¿Y **cómo elegir** la codificación y el algoritmo de resolución adecuados para un determinado problema?

**No existe una respuesta “universal”**, cada problema es un mundo...

Actualmente, existe una amplia comunidad científica trabajando en problemas de satisfacción de restricciones, tanto para mejorar la eficiencia de los algoritmos de resolución, incluyendo las heurísticas, así como para mejorar las codificaciones de los problemas

Sin embargo, existen **técnicas generales** con las que podemos resolver razonablemente bien un gran número de CSP...

## ¿Qué veremos en esta práctica?

**Codificar** problemas usando un lenguaje de CP

- Los resolveremos usando CP solvers (y sus correspondientes heurísticas) como "caja negra" (*black box*)
- Una de las claves está en encontrar una **buena representación del problema** (variables) que nos permitan fácilmente expresar las restricciones
- Además, una buena representación nos permitirá que el CP solver sea capaz de resolver el problema **más eficientemente**

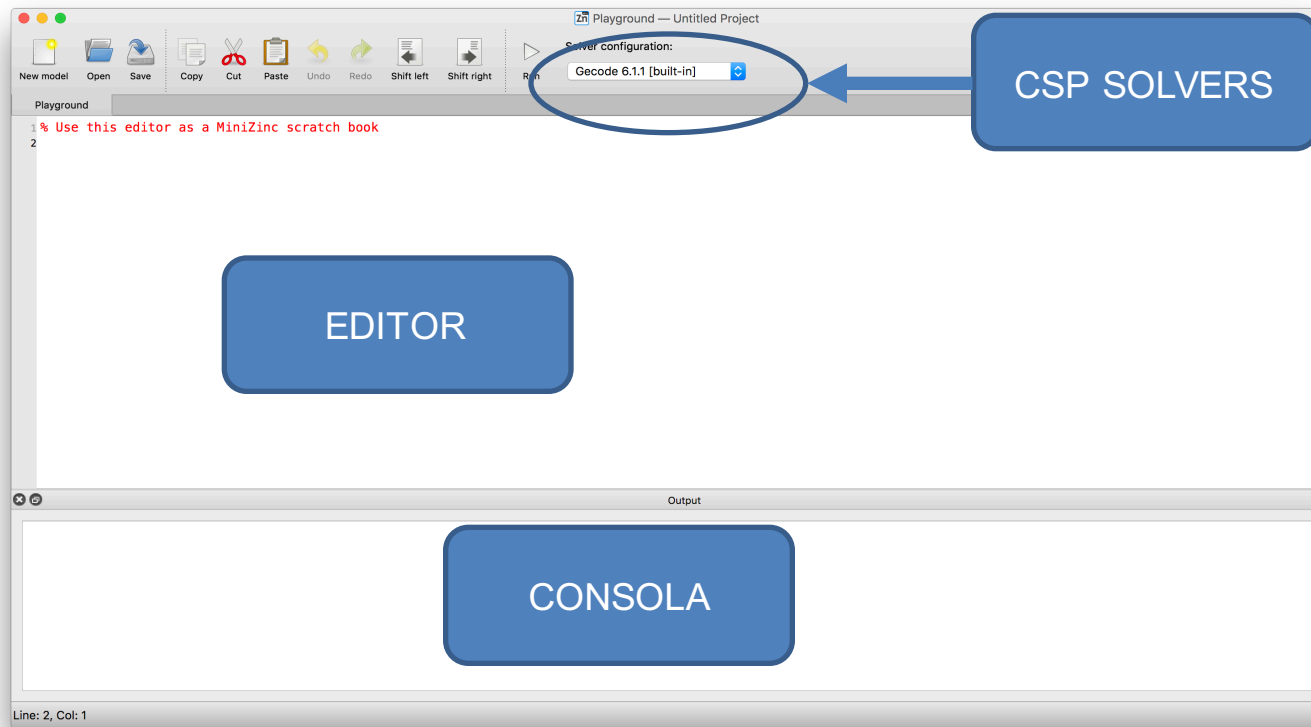
## ¿Qué **NO** veremos en esta práctica?

**Algoritmos** de resolución (búsqueda) y **heurísticas** como "caja blanca" (*white box*)

- En CSP/COP **complejos**, estos conocimientos ayudan a elegir la codificación y algoritmos de resolución más eficientes

En esta práctica usaremos **MiniZinc**: <https://www.minizinc.org/>

MiniZinc es un **lenguaje de modelado de restricciones** gratuito y open-source. Su distribución es multiplataforma (Win, Linux, OSX) e incluye un IDE con un editor de CSP y una batería de CSP solvers (ampliable) para resolverlos



Tutorial de MiniZinc en <https://www.minizinc.org/doc-2.5.3/en/index.html>

La distribución de MiniZinc se puede encontrar en:  
<https://www.minizinc.org/software.html>

**Instalaremos** la distribución apropiada a nuestro SO (se recomienda instalar el “Bundled Binary Package”)

Crearemos nuestro **primer problema CSP**:

**% Definición de variables:**

var int: x;

var int: y;

**% Definición de restricciones:**

constraint x > 0;

constraint y > 0;

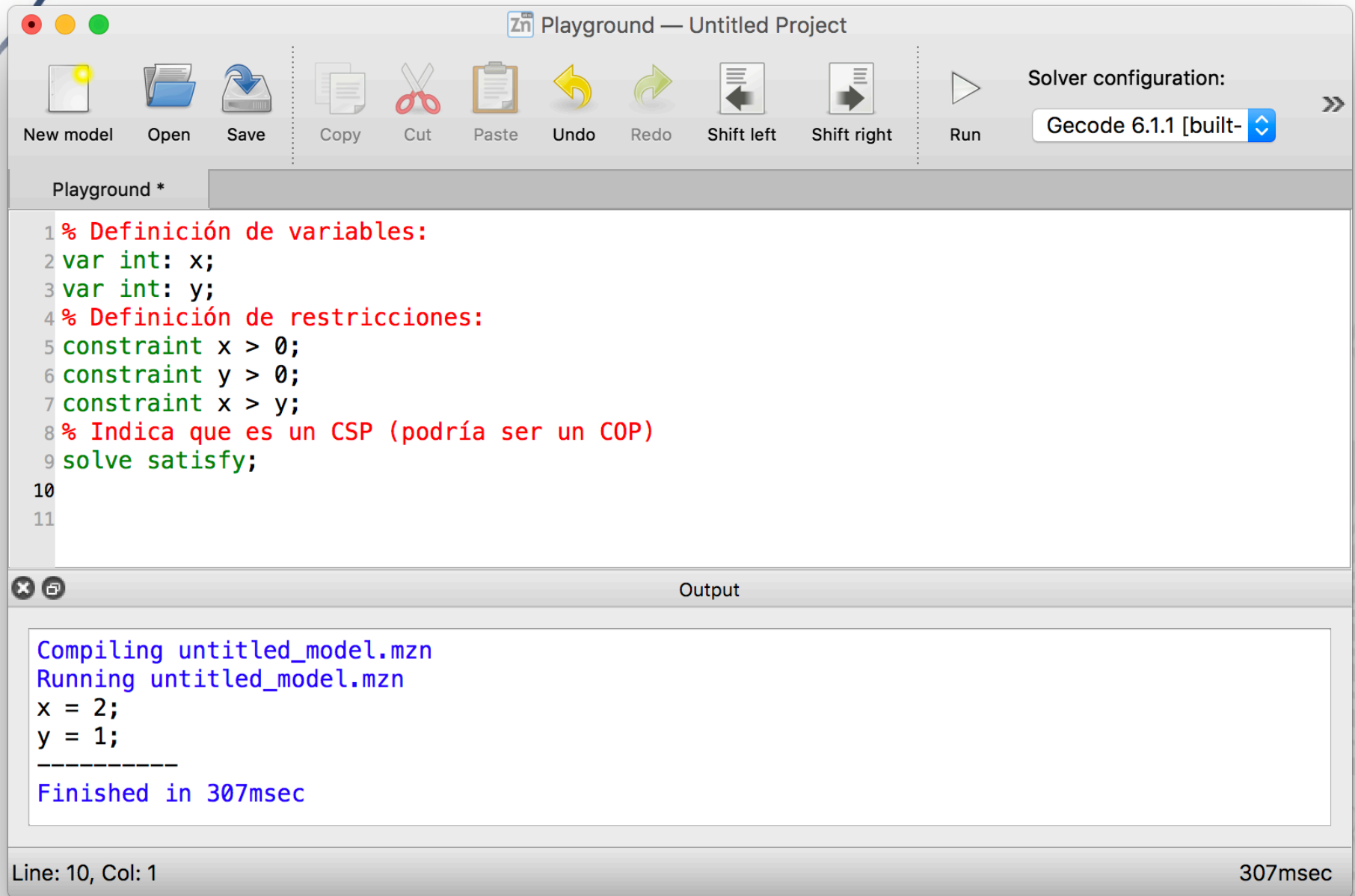
constraint x > y;

**% Indica que es un CSP (podría ser un COP)**

solve satisfy;

p.ej. solve minimize x + y;

Ejecutamos con el botón “**Run**” (podemos usar el CSP solver por defecto, normalmente Gecode), y ver el resultado en la **Consola**



The screenshot shows the MiniZinc Playground interface. The title bar reads "Playground — Untitled Project". The toolbar includes icons for "New model", "Open", "Save", "Copy", "Cut", "Paste", "Undo", "Redo", "Shift left", "Shift right", and a "Run" button. To the right of the toolbar, the "Solver configuration:" section shows "Gecode 6.1.1 [built-...]" with a dropdown arrow.

The main editor area contains the following MiniZinc code:

```

1 % Definición de variables:
2 var int: x;
3 var int: y;
4 % Definición de restricciones:
5 constraint x > 0;
6 constraint y > 0;
7 constraint x > y;
8 % Indica que es un CSP (podría ser un COP)
9 solve satisfy;
10
11

```

Below the editor is the "Output" pane, which displays the compilation and solving process:

```

Compiling untitled_model.mzn
Running untitled_model.mzn
x = 2;
y = 1;
-----
Finished in 307msec

```

At the bottom left, the status bar shows "Line: 10, Col: 1". At the bottom right, the execution time "307msec" is displayed.

## % ESTRUCTURA GENERAL DE UN CSP EN MINIZINC

### % 1. Definición de variables y constantes:

```
var int : x;
```

```
var -1..1 : y; ← Enteros en [-1, 1]
```

```
int : z = 6;
```

### % 2. Definición de restricciones:

```
constraint x > 0 /\ y > 0;
```

```
constraint x > y;
```

```
constraint x > z;
```

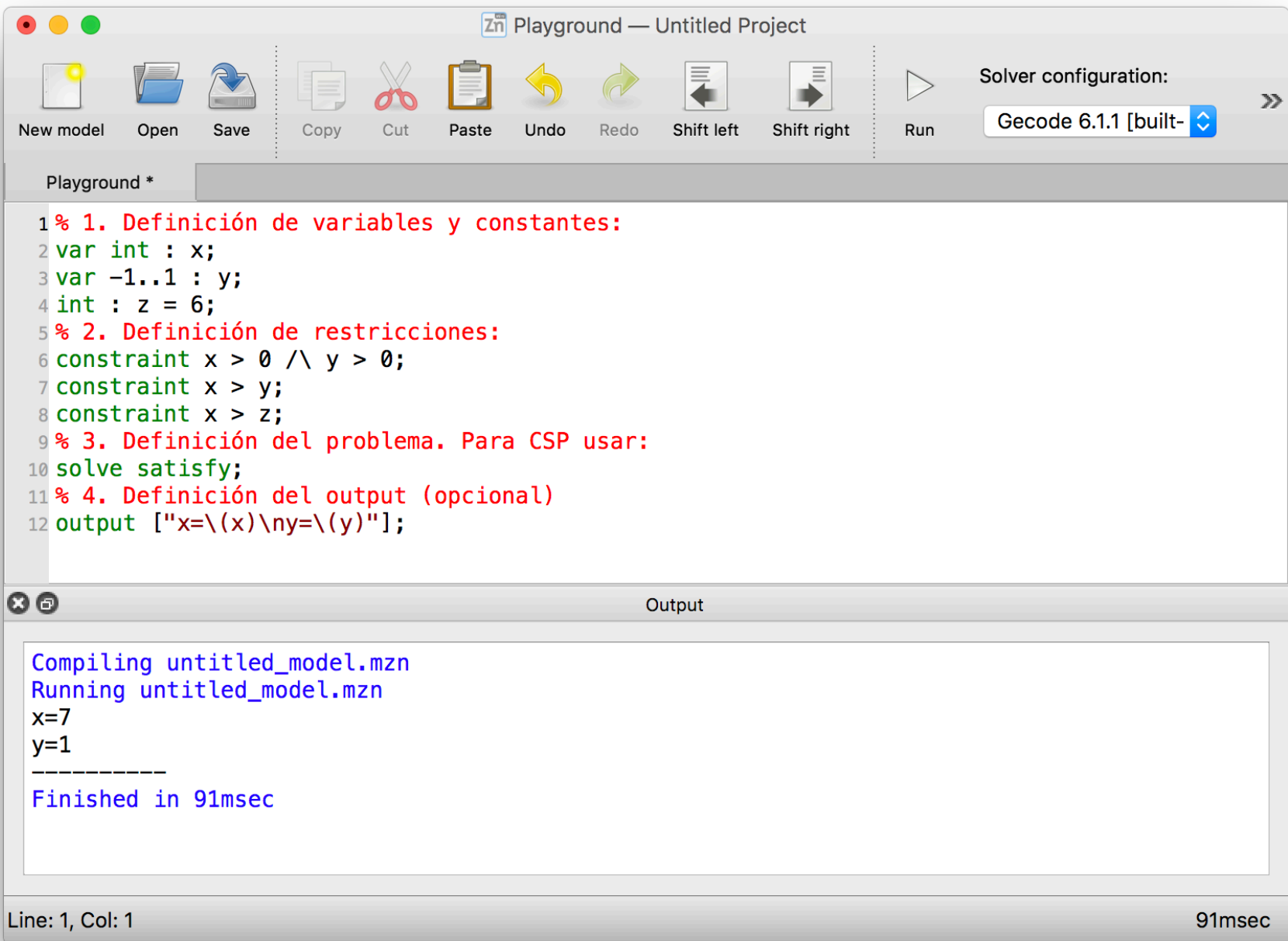
### % 3. Definición del problema. Para CSP usar:

```
solve satisfy;
```

### % 4. Definición del output (opcional)

```
output ["x=\(x)\n y=\(y)"];
```





The screenshot shows the MiniZinc Playground interface. The top toolbar includes buttons for New model, Open, Save, Copy, Cut, Paste, Undo, Redo, Shift left, Shift right, Run, and Solver configuration. The Solver configuration is set to Gecode 6.1.1 [built-].

The main editor displays the following MiniZinc code:

```

1 % 1. Definición de variables y constantes:
2 var int : x;
3 var -1..1 : y;
4 int : z = 6;
5 % 2. Definición de restricciones:
6 constraint x > 0 /\ y > 0;
7 constraint x > y;
8 constraint x > z;
9 % 3. Definición del problema. Para CSP usar:
10 solve satisfy;
11 % 4. Definición del output (opcional)
12 output ["x=\(x)\ny=\(y)"];
  
```

The Output window shows the following results:

```

Compiling untitled_model.mzn
Running untitled_model.mzn
x=7
y=1
-----
Finished in 91msec
  
```

The status bar at the bottom indicates "Line: 1, Col: 1" and "91msec".

% Las variables son aquellas para las que el CSP solver va a buscar valores dentro de sus dominios y respetando las restricciones

% La definición de toda variable comienza por "var"

% seguida por su dominio y su nombre o identificador:

```
var int : x;
```

```
var -1..1 : y;
```

% Un dominio puede ser:

% \* Un tipo primitivo (int, float o bool)

% \* Un intervalo (para int: "a .. b"; para float: "a.x .. b.y", p.ej. 0.0 .. 10.0)

% \* Un conjunto previamente definido ("set of"):

```
set of int: values1 = -1..1;
```

```
var values1: y2;
```

```
set of int: values2 = {-1,0,1};
```

```
var values2: y3;
```

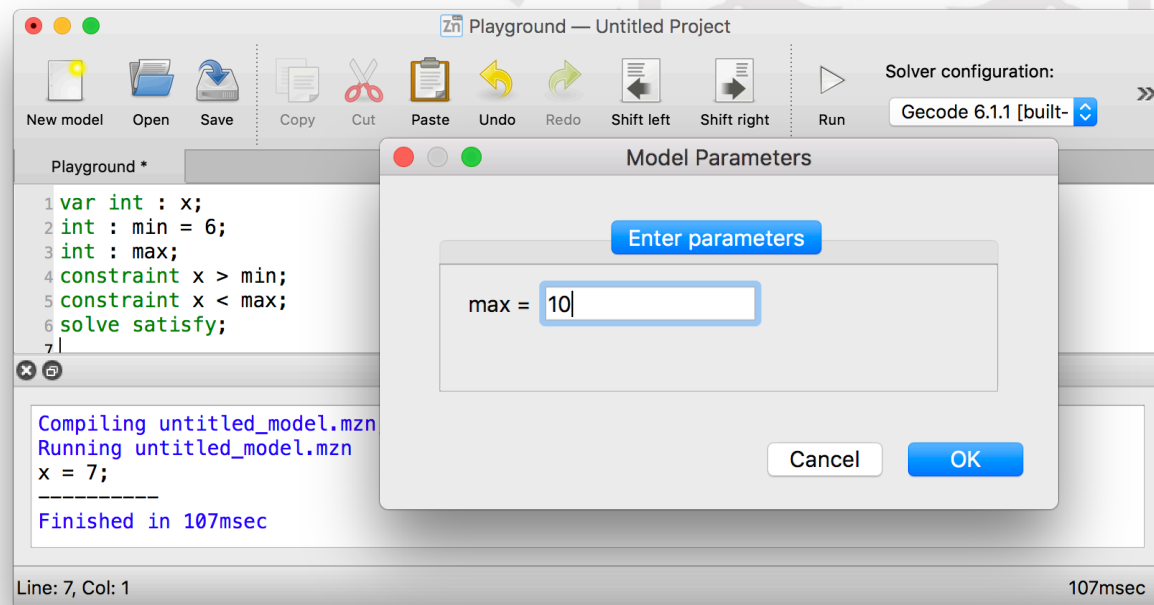
% Las constantes son datos del problema que participan en las restricciones del problema pero cuyos valores son fijos

% Se definen igual que las variables pero sin la palabra reservada "var". También se puede declarar su valor:

```
int : z1;
int : z2 = 6;
```

% Si no se define su valor, éste será preguntado antes de comenzar la ejecución. Ejemplo:

```
var int : x;
int : min = 6;
int : max;
constraint x > min;
constraint x < max;
solve satisfy;
```



% La definición de una restricción comienza por "constraint"  
 % Se pueden usar los operadores lógicos/aritméticos habituales:  
 %  $\leq$  ,  $<$  ,  $=$  ,  $\neq$  ,  $>$  ,  $\geq$   
 % + , - , \* . División entera "a div b". Resto "a mod b"  
 % AND: " $\wedge$ " OR: " $\vee$ " IF: " $\rightarrow$ " IFF: " $\leftrightarrow$ "


% Ejemplo:












var 1..3: WA; var 1..3: NT; var 1..3: SA; var 1..3: Q;  
 var 1..3: NSW; var 1..3: V; var 1..3: T;

constraint WA  $\neq$  NT  $\wedge$  WA  $\neq$  SA ;  
 constraint NT  $\neq$  SA  $\wedge$  NT  $\neq$  Q ;  
 constraint Q  $\neq$  NSW ;  
 constraint NSW  $\neq$  V ;  
 constraint SA  $\neq$  Q ;  
 constraint SA  $\neq$  NSW  $\wedge$  SA  $\neq$  V ;

solve satisfy;



 Playground — Untitled Project

Solver configuration: Gecode 6.1.1 [built-]

Run

Playground \*

```

1 var 1..3: WA; var 1..3: NT; var 1..3: SA; var 1..3: Q;
2 var 1..3: NSW; var 1..3: V; var 1..3: T;
3
4 constraint WA != NT /\ WA != SA ;
5 constraint NT != SA /\ NT != Q ;
6 constraint Q != NSW ;
7 constraint NSW != V ;
8 constraint SA != Q /\ SA != NSW /\ SA != V ;
9
10 solve satisfy;
11
    
```


Output

```

Compiling untitled_model.mzn
Running untitled_model.mzn
WA = 3;
NT = 2;
SA = 1;
Q = 3;
NSW = 2;
V = 3;
T = 1;
-----
Finished in 99msec
    
```

Line: 11, Col: 1

99msec



% Una forma muy útil de definir variables es mediante *arrays*, los cuales son a menudo el input de algunas restricciones globales

% Un vector (unidimensional) se define como:

```
array [<set-of-int>] of var <type>: <identifier>;
```

% donde <set-of-int> identifica las posiciones del *array* y puede ser cualquier conjunto de enteros (también negativos). Ejemplo:

```
array [1..7] of var 1..3: regiones;
```

```
constraint regiones[1] != regiones[2] /\ regiones[1] != regiones[3];
```

```
constraint regiones[2] != regiones[3] /\ regiones[2] != regiones[4];
```

```
constraint regiones[4] != regiones[5];
```

```
constraint regiones[5] != regiones[6] ;
```

```
constraint regiones[3] != regiones[4] /\ regiones[3] != regiones[5]
/\ regiones[3] != regiones[6] ;
```

```
solve satisfy;
```

```
output ["region \ (i) = \ (regiones[i]) \ n" | i in 1..7];
```

% El conjunto de posiciones puede ser cualquiera (pero cuidado cuando se accede a él!):

```
array [1..7] of var 1..3: regiones;
    output ["\\(regiones[i]) " | i in 1..7];
array [10..16] of var 1..3: regiones;
    output ["\\(regiones[i]) " | i in 10..16];
array [-10..-4] of var 1..3: regiones;
    output ["\\(regiones[i]) " | i in -10..-4];
```

% Para evitarlo se pueden definir "sets"

```
set of int: POS = 1..7;
set of int: VALUES = 1..3;
array [POS] of var VALUES: regiones;
output ["\\(regiones[i]) " | i in POS];
```



% También se pueden definir arrays de constantes

% (sin la palabra "var"):

set of int: POS = 1..7;

set of int: VALUES = 1..3;

array [POS] of var VALUES: regiones;

**array[POS] of string: strRegiones =**

**["WA","NT", "SA", "Q", "NSW", "V", "T"];**

constraint regiones[1] != regiones[2] /\ regiones[1] != regiones[3] ;

constraint regiones[2] != regiones[3] /\ regiones[2] != regiones[4] ;

constraint regiones[4] != regiones[5] ;

constraint regiones[5] != regiones[6] ;

constraint regiones[3] != regiones[4] ;

constraint regiones[3] != regiones[5] /\ regiones[3] != regiones[6] ;

output ["\\(**strRegiones[i]**) = \\\(regiones[i])\\n" | i in POS];

% Para definir un array multidimensional únicamente tenemos que  
% definir los tamaños de cada dimension:

```
set of int: XPOS = 1..5;
```

```
set of int: YPOS = 1..3;
```

```
array [XPOS,YPOS] of var 0..9: grid;
```

% Grid de 5 filas, 3 columnas, con valores del 0 al 9:

grid	y=1	y=2	y=3
x=1	var 0..9	var 0..9	var 0..9
x=2	var 0..9	var 0..9	var 0..9
x=3	var 0..9	var 0..9	var 0..9
x=4	var 0..9	var 0..9	var 0..9
x=5	var 0..9	var 0..9	var 0..9

- El formateo de salida (**output**) debe ser una **lista de strings**
- Algunas funciones útiles para mostrar el valor de las variables: **show**, **show\_int**, **show\_float**:

```
output["\n(S)\n"]
output[show(S)]
```

Expresiones equivalentes

```
output["t=\n(t)\n"]
output["t=" ++ show(t) ++ "\n"]
```

Expresiones equivalentes

- fix** se emplea en el output para convertir una variable en una constante. Es necesario en ocasiones, por ejemplo, cuando se emplea un if en el output:

```
output
[ "Encargado: " ] ++
[ if fix(encargado[i]) == 1 then "\n(nombres[i])" else "" endif | i in 1..n ]
;
```

<https://www.minizinc.org/doc-2.5.3/en/modelling.html> (sección Output and Strings)

% A menudo, es común usar restricciones globales, definidas por el usuario o predefinidas en librerías. Las más comunes son:

```
include "globals.mzn";
```

```
constraint all_different(<array>);
```

```
constraint forall(<set>)(<restricción>);
```

% Por ejemplo:

```
array[0..9] of var 0..9: digitos;
```

```
constraint all_different(digitos);
```

```
constraint forall(i,j in 0..9)(i==j ∨ digitos[i] != digitos[j]);
```

$$a \rightarrow b \equiv \neg a \vee b$$

$$i \neq j \rightarrow \text{digitos}[i] \neq \text{digitos}[j]$$

% Los array multidimensionales, si queremos, los podemos transformar en unidimensionales:

```
array[1..3,1..3] of var 1..9: grid;
```

```
constraint all_different( [ grid[i,j] | i,j in 1..3 ] );
```

```
constraint all_different( grid );
```

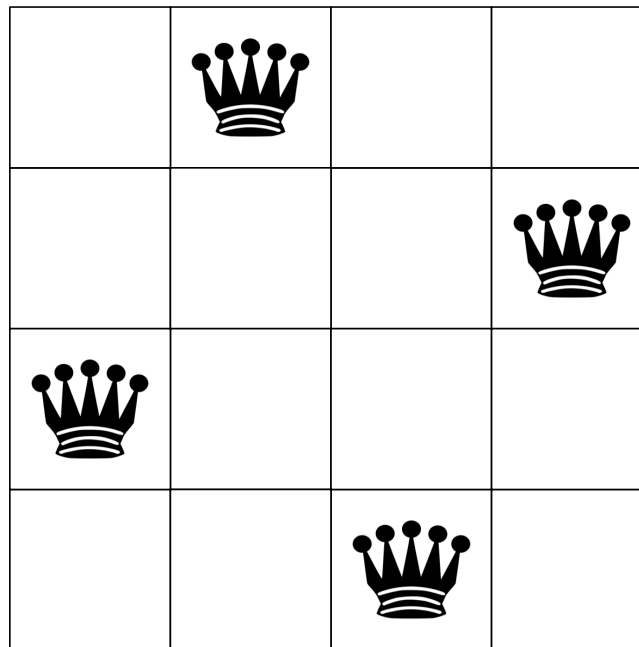
Ambos all\_different producen el mismo resultado!

% También se pueden usar predicados definidos por el usuario

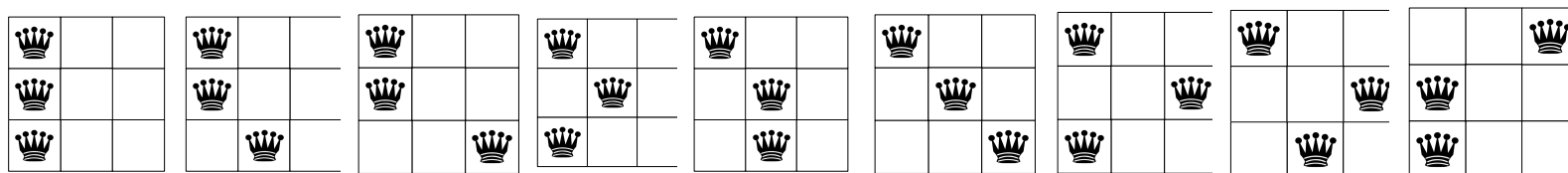
# Problema de las N-Reinas

**Problema:** Posicionar N reinas en un tablero NxN de forma que ningún par de reinas se ataque mutuamente.

Ejemplo: 4-reinas



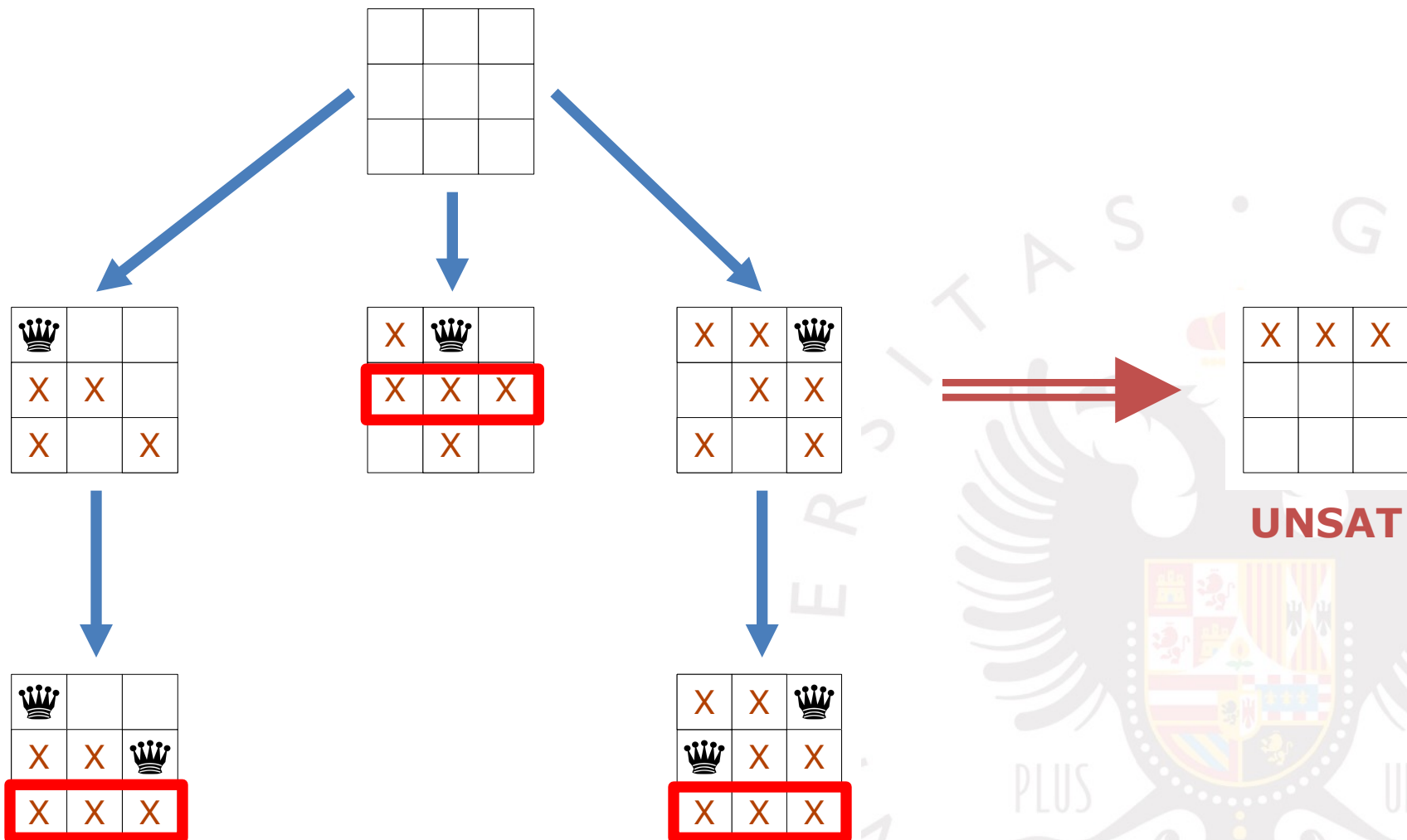
# Problema de las N-Reinas



N	#configuraciones
3	27
4	256
8	16.777.216
16	18.446.744.073.709.551.616
...	...

No se pueden explorar todas ellas por fuerza bruta.  
Hace falta **búsqueda!**

# Problema de las N-Reinas









% Ejemplo: N-Reinas:

include "globals.mzn";

int: n;

array [1..n] of var 1..n: reinas;

	X=1	X=2	X=3	X=4
Y=1				
Y=2				
Y=3				
Y=4				

reinas = [2,4,1,3]

% Ataques en misma fila: no hay (cada reina ya está en filas

% distintas gracias al modelado del problema)

constraint alldifferent(reinas);

% Ataques en columnas

constraint alldifferent([reinas[i] + i | i in 1..n]); % Ataques en diag. princ.

constraint alldifferent([reinas[i] - i | i in 1..n]); % Ataques en diag. sec.

solve satisfy;

output [ if fix(reinas[j]) == i then "Q" else "." endif ++

if j == n then "\n" else "" endif | i,j in 1..n];

### Para las **diagonales**:

Si una reina en la fila  $i$  está en la posición  $Q_i$ , entonces en la fila  $j=i+x$  la reina NO puede estar en la posición  $Q_i+x$ . Generalizando tenemos:

$|Q_j - Q_i| \neq |i - j|$  (para cualquier par  $i,j$ )

Eliminando los valores absolutos tenemos:

$Q_j - j \neq Q_i - i$  (diagonal principal)

$Q_j + j \neq Q_i + i$  (diagonal secundaria)

$i=1$		$Q_i=2$		
$j=i+x$ $=1+2$				$Q_j=2+2$

$4-3 \neq 2-1$  ✗ diagonal principal

$4+3 \neq 2+1$  ✓ diagonal secundaria

% Estas restricciones globales se pueden combinar con otras  
% funciones típicas (suma, contar, etc...):

```
sum(<POS>) ( <array> )
```

```
count(<array>,<value>)
```

```
...
```

% Ejemplo:

```
include "globals.mzn";
```

% Array con 10 posiciones con números del 0 al 20

```
array[0..9] of var 0..20: myarray;
```

% All different:

```
constraint forall(i in 0..20)(count(myarray,i)<=1);
```

% La suma de la primera mitad igual a la suma de la segunda mitad

```
constraint sum(i in 0..4)(myarray[i]) ==  
           sum(i in 5..9)(myarray[i]);
```

```
solve satisfy;
```

# Problema del cuadrado mágico

**Problema:** Dado un tablero  $N \times N$ , posicionar los números entre 1 y  $N^2$  de forma que todos ellos sean distintos y que todas las sumas por filas, columnas y diagonales principales sean iguales (esta suma se conoce como “suma mágica”).

Ejemplo: Cuadrado mágico de tamaño 4  
(suma mágica = 34)

9	6	11	8
5	10	7	12
4	3	14	13
16	15	2	1

```
include "globals.mzn";
int: n;
set of int: S = 1..n; % size
set of int : N = 1 .. n*n; % numbers
array[S, S] of var N: magic;
var int: ms; % magic sum
```

También valdría  
alldifferent(magic)

% Todos los números distintos

```
constraint alldifferent( [ magic[fila,col] | fila,col in S ] );
```

% Suma mágica en filas y columnas

```
constraint forall(fila in S)(sum(col in S) (magic[fila,col]) == ms);
```

```
constraint forall(col in S) (sum(fila in S) (magic[fila,col]) == ms);
```

% Suma mágica en diagonales

```
constraint sum(fila in S)(magic[fila,fila]) == ms;
```

```
constraint sum(fila in S) (magic[fila, n-fila+1]) == ms;
```

```
solve satisfy;
```

## % Ejemplo: Sudoku

```
include "globals.mzn";
```

```
int: S = 3; int: N = S * S;
```

```
set of int: RangoPuzzle = 1..N; set of int: RangoCuadrado = 1..S;
```

```
array[RangoPuzzle, RangoPuzzle] of var RangoPuzzle: puzzle;
```

## % Todos diferentes por filas

```
constraint forall (i in RangoPuzzle)
```

```
    ( alldifferent( [ puzzle[i,j] | j in RangoPuzzle ] ) );
```

## % Todos diferentes por columnas

```
constraint forall (j in RangoPuzzle)
```

```
    ( alldifferent( [ puzzle[i,j] | i in RangoPuzzle ] ) );
```

## % (continua en la siguiente diapositiva...)

```
% int: S = 3; int: N = S * S;
```

```
% set of int: RangoPuzzle = 1..N; set of int: RangoCuadrado = 1..S;
```

```
%Todos diferentes en cada cuadrado
```

```
constraint forall (a, o in RangoCuadrado)
```

```
( alldifferent( [ puzzle[S*(a-1)+x , S*(o-1)+y]
                  | x, y in RangoCuadrado ] ) );
```

Cada cuadrado 3x3 se puede ver como una "celda" con coordenadas a,o en un grid 3x3 (con a,o en 1..3)

Las 9 celdas de ese cuadrado son:

Filas:  $S*(a-1)+x$

Columnas:  $S*(o-1)+y$   
(con x,y en 1..3)

```
solve satisfy;
```

```
output [ "\((puzzle[i,j]) " ++
```

```
if i == N then "\n" else "" endif | j,i in 1..N];
```



- **Los solvers que se van a emplear para desarrollar la práctica y corregirla son Gecode 6.1.1 o Chuffed 0.10.4 (o versiones siguientes).**
  - Se trata de solvers completos. Es decir, dada una codificación adecuada, deberían dar la solución.
- No hay que confundir que la solución sea correcta o no (es decir, satisfaga todas las restricciones) con que la codificación en sí misma sea correcta y completa.
  - Por correcta se entiende que la codificación proporciona soluciones que satisfacen todas las restricciones.
  - Por completa se entiende que cualquier solución es admitida como válida por la codificación.

Nosotros debemos generar soluciones correctas y completas (si

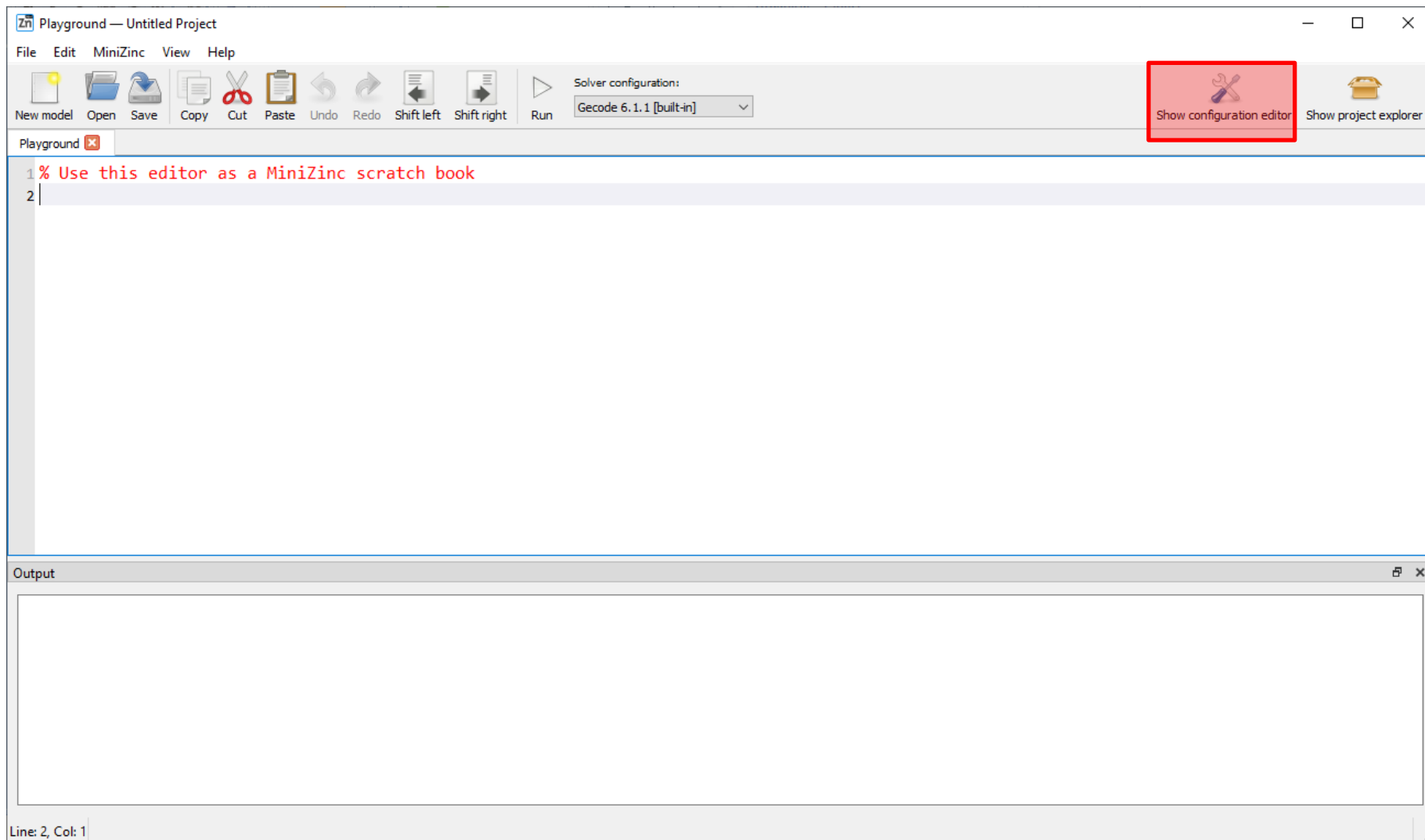
- Ejemplo: encontrar un número en  $[1,3]$ .
  - Las soluciones  $x=1$ ,  $x=2$  y  $x=3$  son correctas. Cualquier otra no.
  - La codificación  $x>0 \ \&\& \ x<4$  es correcta y completa.
  - La codificación  $x>0$  no es correcta (podría dar como solución  $x=4$ )
  - La codificación  $x==1 \ || \ x==3$  no es completa (nunca puede dar la solución  $x=2$ )

- **Tres heurísticas para enfrentarse a la práctica**

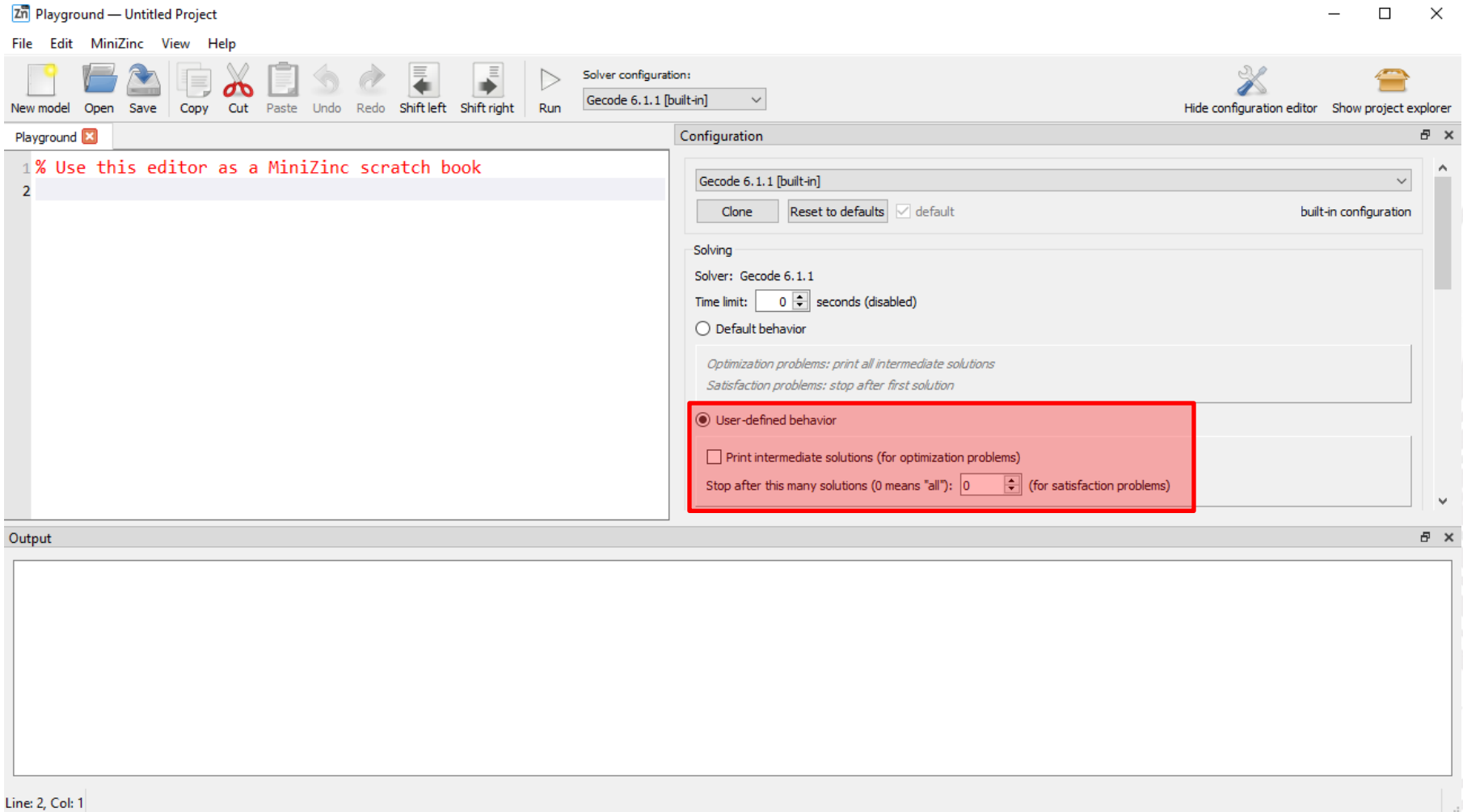
1. Si se usan **demasiadas variables y restricciones** podemos estar intentando resolver un problema de un modo demasiado complicado. Puede que eso sea lo que haga que no se resuelva de modo adecuado el problema en un tiempo razonable.
2. Estamos ante problemas pequeños. De modo que, una vez completamente codificados, se deberían **resolver en 1-3 segundos**. No obstante, en ocasiones, cuando están codificados parcialmente pueden tardar unos 30-60 segundos en devolver la primera solución.
3. En términos generales, todo lo que sea que un problema no se resuelva en el tiempo indicado anteriormente (1-3 segundos) o que no funcione **igual de bien con cualquier solver** es para dudar de si lo que se ha hecho es correcto.

Utilizar operadores unarios, más que binarios; y binarios mejor que ternarios. Encontrar u

- MiniZinc permite **imprimir todas las soluciones de una codificación**. Esto permite **verificar que una codificación es correcta y completa**.



- "Show configuration editor" - "User-defined behavior" - "Stop after this many solutions (0 means 'all')".



The screenshot shows the MiniZinc Playground interface. The main editor on the left contains two lines of text:   
1 % Use this editor as a MiniZinc scratch book   
2   
The Configuration panel on the right shows the following settings:   
- Solver configuration: Gecode 6.1.1 [built-in]   
- Solving: Gecode 6.1.1   
- Time limit: 0 seconds (disabled)   
- Behavior: User-defined behavior (selected)   
- Print intermediate solutions (for optimization problems): unchecked   
- Stop after this many solutions (0 means "all"): 0 (for satisfaction problems)   
The 'User-defined behavior' section is highlighted with a red box. The status bar at the bottom indicates 'Line: 2, Col: 1'.

- La práctica consiste en realizar la codificación de 10 problemas de satisfacción de restricciones (ver enunciado de la práctica)
- Cada ejercicio tiene una puntuación máxima de un punto, que se debe obtener para que la solución sea **completa** (las soluciones que no son completas se consideran incorrectas). Los ejercicios deben ejecutarse en menor de un segundo y tienen una longitud máxima de 1000 caracteres.
- **No es necesario elaborar ninguna memoria, pero el código sí debe estar adecuadamente comentado.**
- **IMPORTANTE: la salida de cada ejercicio se debe formatear como se indica en el enunciado!**
- La entrega consistirá en un **fichero ZIP** que contenga los 10 ficheros correspondientes al **código MiniZinc** de cada ejercicio
- Los ejercicios que tengan **errores sintácticos en MiniZinc** automáticamente califican con **0 (cero) puntos**.

**Fecha de entrega: 16 de mayo de 2021 23:59**



UNIVERSIDAD  
DE GRANADA



# Técnicas de los Sistemas Inteligentes

Grado en Ingeniería Informática

**Curso 2020-21. Seminario 2**

**Satisfacción de Restricciones**

Jesús Giráldez Crú y Pablo Mesejo Santiago

**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**

**<http://decsai.ugr.es>**