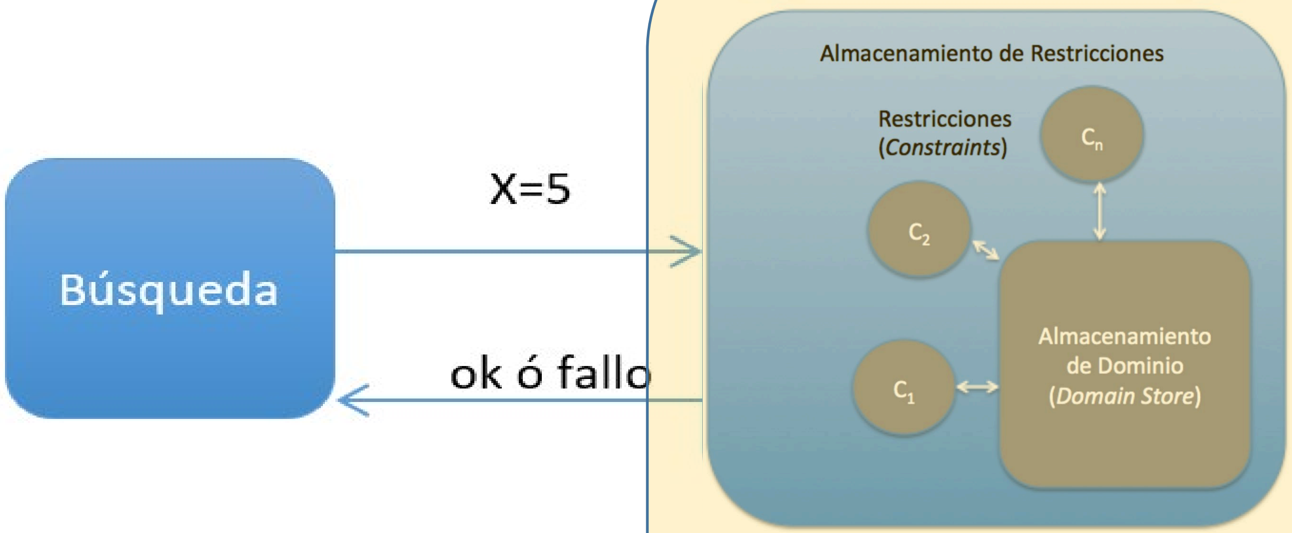
Algoritmos y Programación

Programación con Restricciones 6.1

Almacenamiento de Dominio



Test de factibilidad

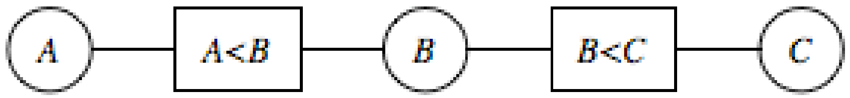
$$\exists\, v_1 \in D_1, \dots, v_n \in D_n:$$
$$c(x_1 = v_1, \dots, x_n = v_n) = \mathit{true}$$

Reducir el espacio de búsqueda

Red de restricciones

- Cada variable viene representada por un nodo de tipo círculo
- Cada restricción viene representada por un nodo de tipo rectángulo
- Para cada variable y para cada restricción existe un arco que los une
- Al grafo obtenido se le llama red de restricciones
- Por ejemplo, A<B y B<C generaría el grafo de la derecha, con los arcos que se muestran

Red de Restricciones



Arcos

⟨A,A<B⟩

⟨B,B>A⟩

⟨B,B<C⟩

⟨C,C>B⟩

Consistencia de arco. Algoritmo AC-3

Paso 1) Convertir cada restricción binaria en 2 arcos,

por ejemplo:

La restricción $A \neq B$ pasa a ser $A \neq B$ y $B \neq A$. A la parte izquierda del arco lo vamos a llamar x_i y a la parte derecha x_j

Paso 2) Añadir todos los arcos a una lista de trabajo

Paso 3) Repetir hasta que la lista de trabajo esté vacía

Coger un arco (x_i, x_j)de la lista de trabajo y comprobar que es factible

Paso 3.1) Para cada valor de x_i, debe haber algún valor de x_j que cumpla la restricción

Paso 3.2) Eliminar los valores inconsistentes de x_i

Paso 3.3) Si el dominio de valores de x_i ha cambiado, añadir a la lista de trabajo todos los arcos de la forma (x_k, x_i), si alguno de los arcos ya está en la lista de trabajo no hay que volver añadirlo

Ejemplo

Ejemplo:
Supongamos:

$A = \{1,2,3\}$
$B = \{1,2,3\}$
$C = \{1,2,3\}$

con las restricciones:

$A > B$

$B = C$

Paso 1) Convertir cada restricción binaria en 2 arcos

Arcos

$A > B \rightarrow \boldsymbol{A > B},\ \boldsymbol{B < A}$
$B = C \rightarrow \boldsymbol{B = C}, \boldsymbol{C = B}$

Paso 2) Añadir todos los arcos a una lista de trabajo

$A > B$
$B < A$
$B = C$
$C = B$

Ejemplo

Ejemplo:
Supongamos:

$A = \{1,2,3\}$
$B = \{1,2,3\}$
$C = \{1,2,3\}$

con las restricciones:

$A > B$

$B = C$

$A > B$
$B < A$
$B = C$
$C = B$

Paso 3) Empezamos con el primer arco de la lista de trabajo $A > B$

Para que el arco sea consistente $A = \{\cancel{1},2,3\}$

$\cancel{A > B}$
$B < A$
$B = C$
$C = B$

Ejemplo

Ejemplo:
Supongamos:

$A = \{\cancel{1},2,3\}$
$B = \{1,2,3\}$
$C = \{1,2,3\}$

con las restricciones:

$A > B$

$B = C$

$A > B$
$B < A$
$B = C$
$C = B$

Paso 3) B<A

Para que el arco sea consistente $B = \{1,2,\cancel{3}\}$.

$\cancel{A > B}$
$\cancel{B < A}$
$B = C$
$C = B$
$A > B$

Ejemplo

Ejemplo:

Supongamos:

$$A = \{\cancel{1},2,3\}$$
$$B = \{1,2,\cancel{3}\}$$
$$C = \{1,2,3\}$$

con las restricciones:

$$A > B$$

$$B = C$$

$A > B$
$B < A$
$B = C$
$C = B$

Paso 3) B=C

$\cancel{A > B}$
$\cancel{B < A}$
$\cancel{B = C}$
$C = B$
$A > B$

Ejemplo

Ejemplo:

Supongamos:

$A = \{\cancel{1},2,3\}$

$B = \{1,2,\cancel{3}\}$

$C = \{1,2,3\}$

con las restricciones:

$A > B$

$B = C$

$A > B$
$B < A$
$B = C$
$C = B$

Paso 3) C=B

$C = \{1,2,\cancel{3}\}$

$\cancel{A > B}$
$\cancel{B < A}$
$\cancel{B = C}$
$\cancel{C = B}$
$A > B$
$B = C$

Ejemplo

Ejemplo:

Supongamos:

$A = \{\cancel{1},2,3\}$

$B = \{1,2,\cancel{3}\}$

$\underline{C = \{1,2,\cancel{3}\}}$

con las restricciones:

$A > B$

$B = C$

$A > B$
$B < A$
$B = C$
$C = B$

Paso 3) A>B

$\cancel{A > B}$
$\cancel{B < A}$
$\cancel{B = C}$
$\cancel{C = B}$
$\cancel{A > B}$
$B = C$

Ejemplo

Ejemplo:
Supongamos:

$A = \{\cancel{1},2,3\}$
$B = \{1,2,\cancel{3}\}$
$C = \{1,2,3\}$

con las restricciones:

$A > B$

$B = C$

$A > B$
$B < A$
$B = C$
$C = B$

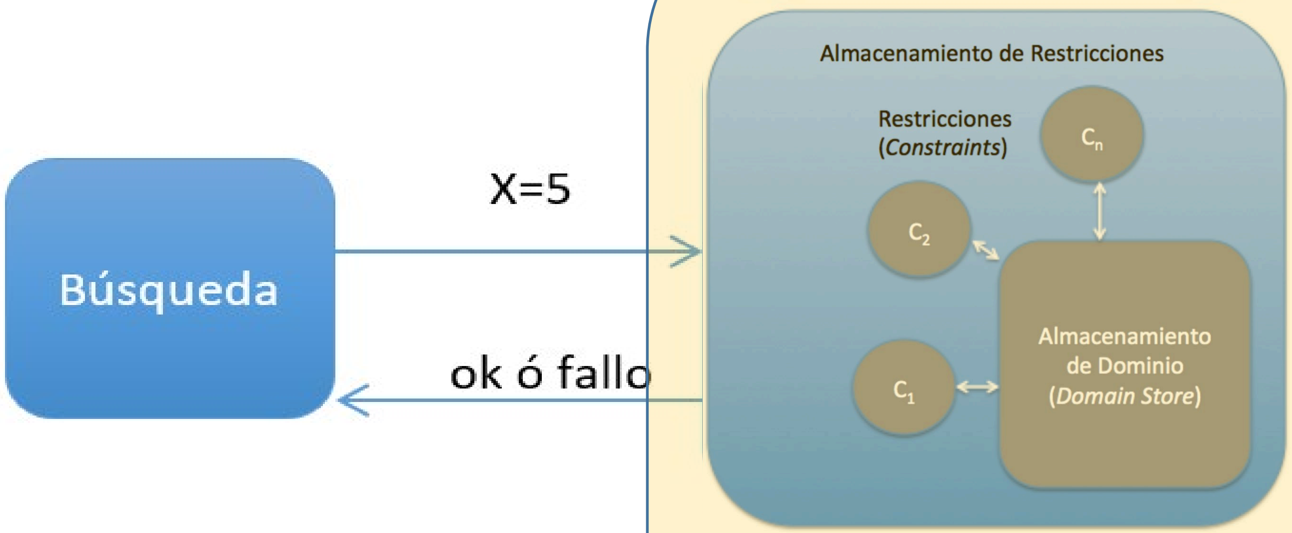
Paso 3) B=C

Finalmente probamos con B=C que tampoco modifica los dominios de las variables concluyendo que:

$A = \{2,3\}$
$B = \{1,2\}$
$C = \{1,2\}$

es arco consistente o dominio consistente.

Subsistema de búsqueda



Test de factibilidad

$$\exists\, v_1 \in D_1, \dots, v_n \in D_n:$$
$$c(x_1 = v_1, \dots, x_n = v_n) = true$$

Reducir el espacio de búsqueda

Subsistema de búsqueda

- Heurísticas de selección de variables
  - Orden de entrada (minizinc: *input_order*)
  - Dominios con valores más pequeños (minizinc: *smallest*)
  - Dominios con valores más grandes (minizinc: *largest*)
  - Dominio con cardinalidad menor (minizinc: *first_fail*)
  - Dominio con cardinalidad menor y en caso de empate mayor cantidad de restricciones (minizinc: *most_constrained*)
- Heurísticas de selección de valores

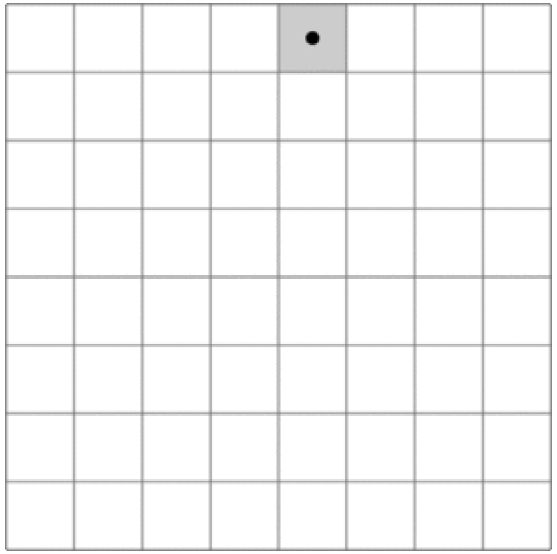
https://www.minizinc.org/doc-2.4.2/en/lib-annotations.html#search-annotations

Problema del Caballo

- Utilizar un caballo para visitar, una sola vez, todas las casillas de un tablero de ajedrez

Solución de Euler

42	57	44	9	40	21	46	7
55	10	41	58	45	8	39	20
12	43	56	61	22	59	6	47
63	54	11	30	25	28	19	38
32	13	62	27	60	23	48	5
53	64	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	52	15	34	3	50	17	36



Posible modelo:
Caballo de Euler

- En este modelo tenemos:
  - La variable de decisión *jump* que para cada posición del tablero nos da la siguiente posición a donde va a saltar el caballo:
    - *var{int} jump[i in Board] in Knightmoves(i)*
  - Todos los valores de la variable *jump* tienen que formar un circuito: *circuit(jump)*
  - *Knightmoves(i):* Función que da las reglas para el salto del caballo

```
function set{int} Knightmoves(int i) {
   set{int} S;
   if (i % 8 == 1)
      S = {i-15,i-6,i+10,i+17};
   else if (i % 8 == 2)
      S = {i-17,i-15,i-6,i+10,i+15,i+17};
   else if (i % 8 == 7)
      S = {i-17,i-15,i-10,i+6,i+15,i+17};
   else if (i % 8 == 0)
      S = {i-17,i-10,i+6,i+15};
   else
      S = {i-17,i-15,i-10,i-6,i+6,i+10,i+15,i+17};
   return filter(v in S) (v >= 1 && v <= 64);
}
```

```
range Board = 1..64;
var{int} jump[i in Board] in Knightmoves(i);
solve {
   circuit(jump);
}
```

circuit
minizinc

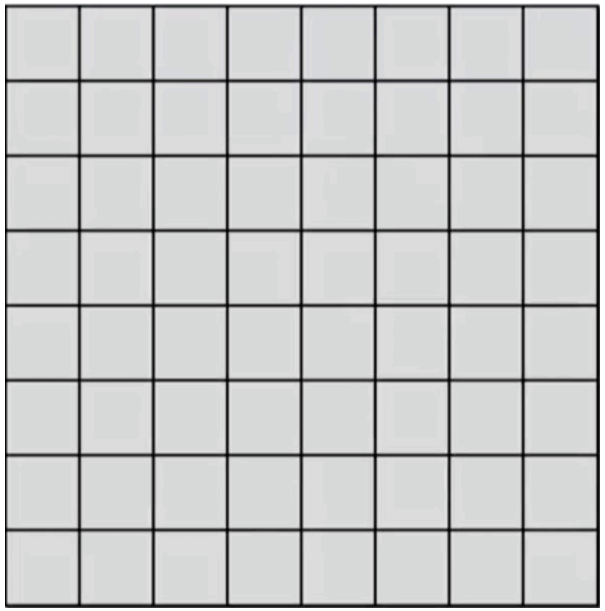
Documentación de minizinc:

circuit

- **circuit[array[int] of var int: x)**
  Constraints the elements of x to define a circuit where x[i] = j mean that j is the successor of i.

Caballo de Euler

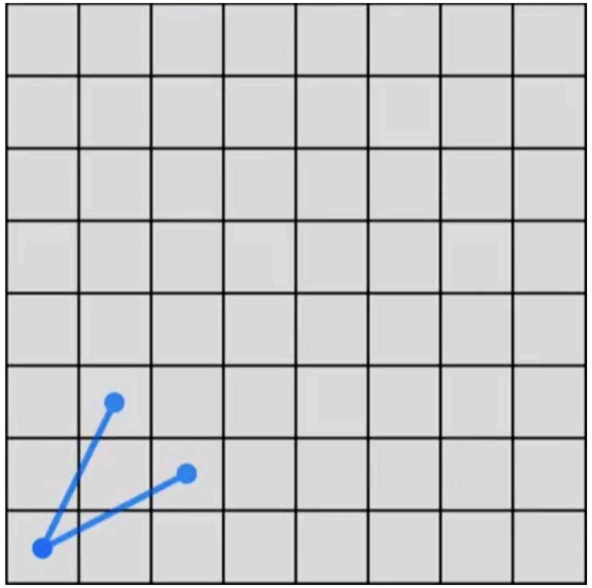
- Principio de primer fallo



¿Dónde deberíamos empezar?

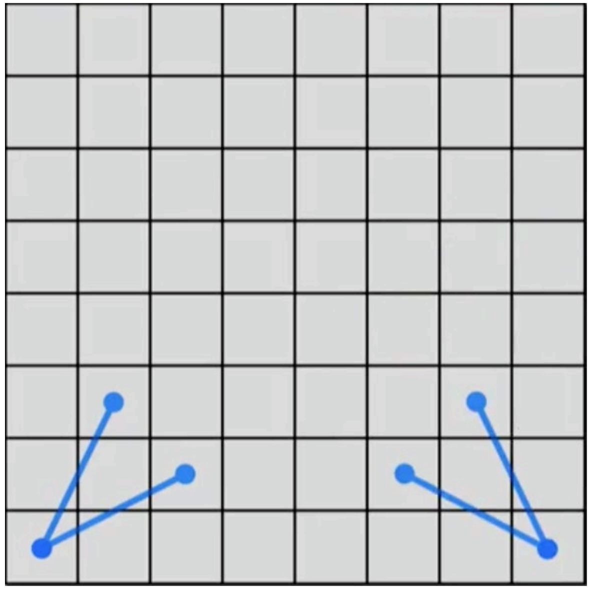
¿Desde dónde tendría un caballo más dificultades para saltar?

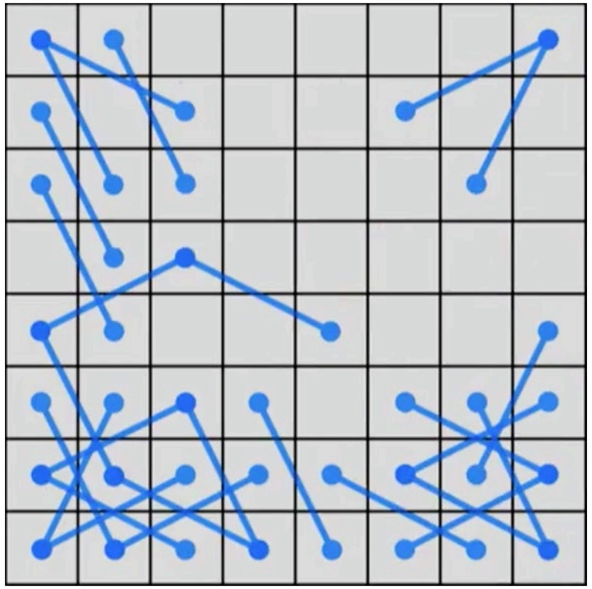
Caballo de Euler

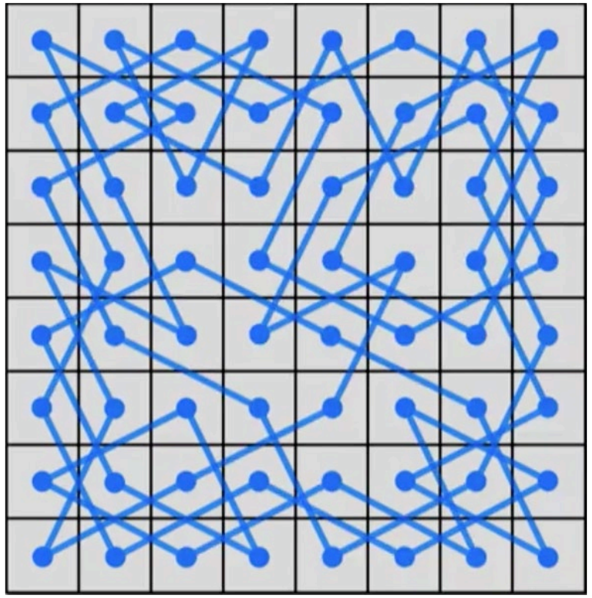


Volvemos a aplicar el mismo principio. ¿Cuál sería la siguiente casilla a explorar?

Caballo de Euler



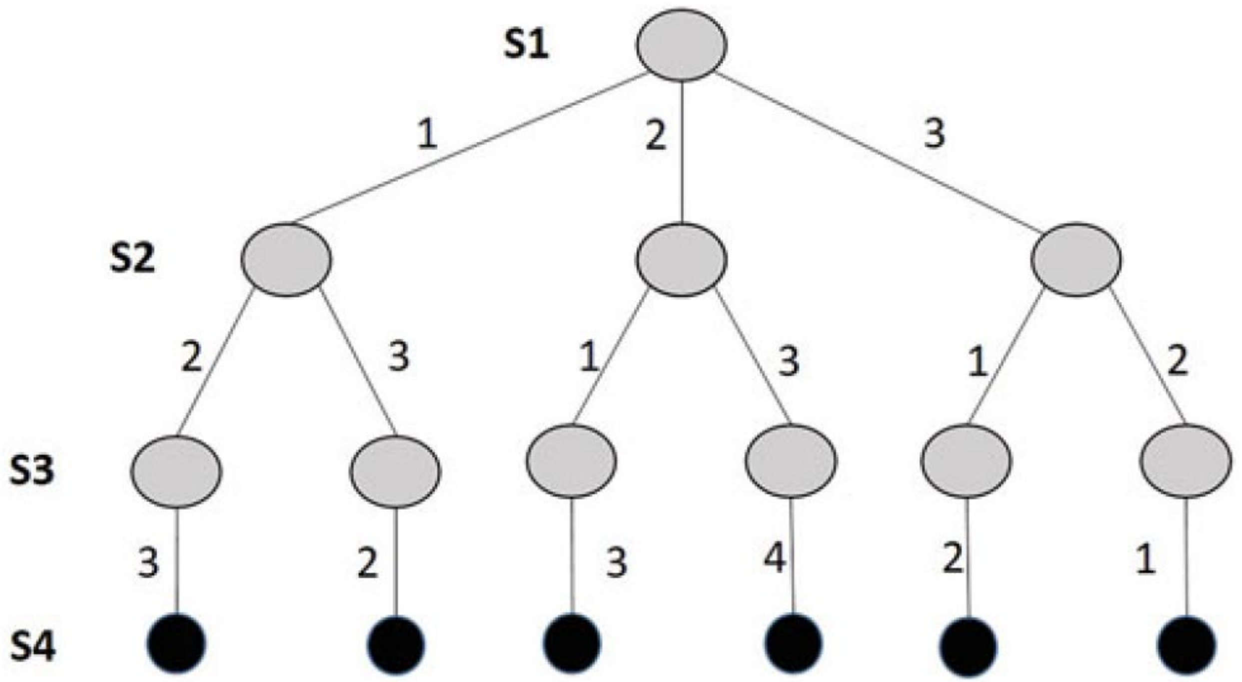




Ejemplo

- Supongamos que tenemos 4 variables de decisión y una restricción *alldifferent*:

$$S1 \in 1..3,\ S2 \in 1..3,\ S3 \in 1..3,\ S4 \in 1..2$$

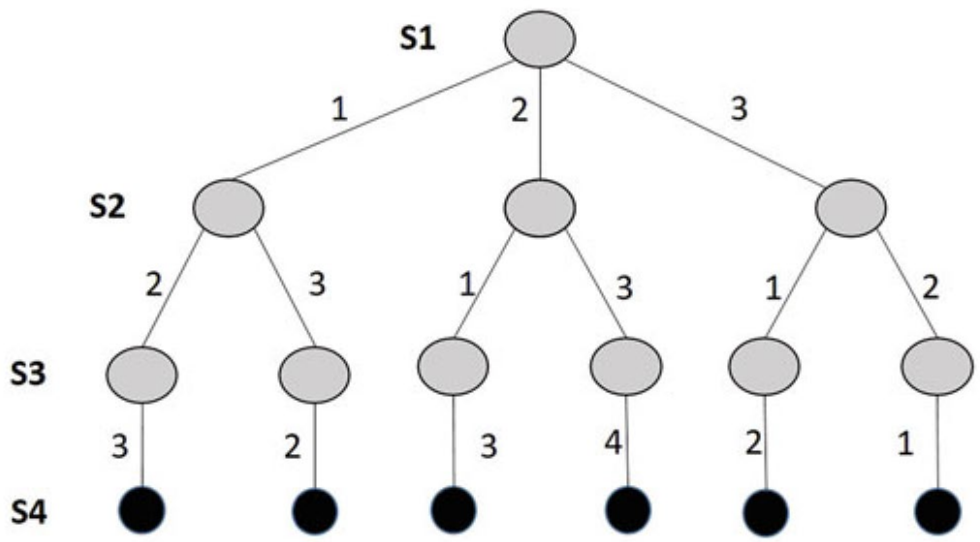


Árbol de fallo

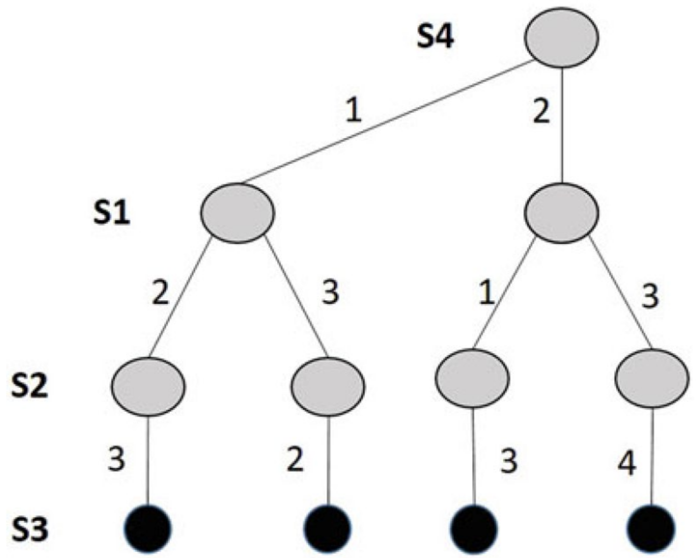
Ejemplo

- Supongamos que tenemos 4 variables de decisión y una restricción *alldifferent*:

$$S1 \in 1..3,\ S2 \in 1..3,\ S3 \in 1..3,\ S4 \in 1..2$$

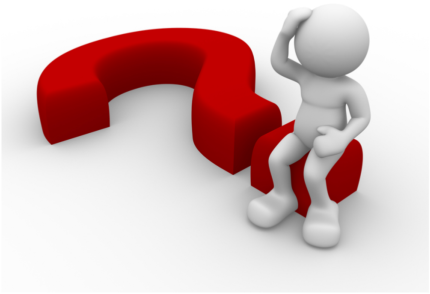


Árbol de fallo



First Fail

Ejemplo, coloreado de mapas



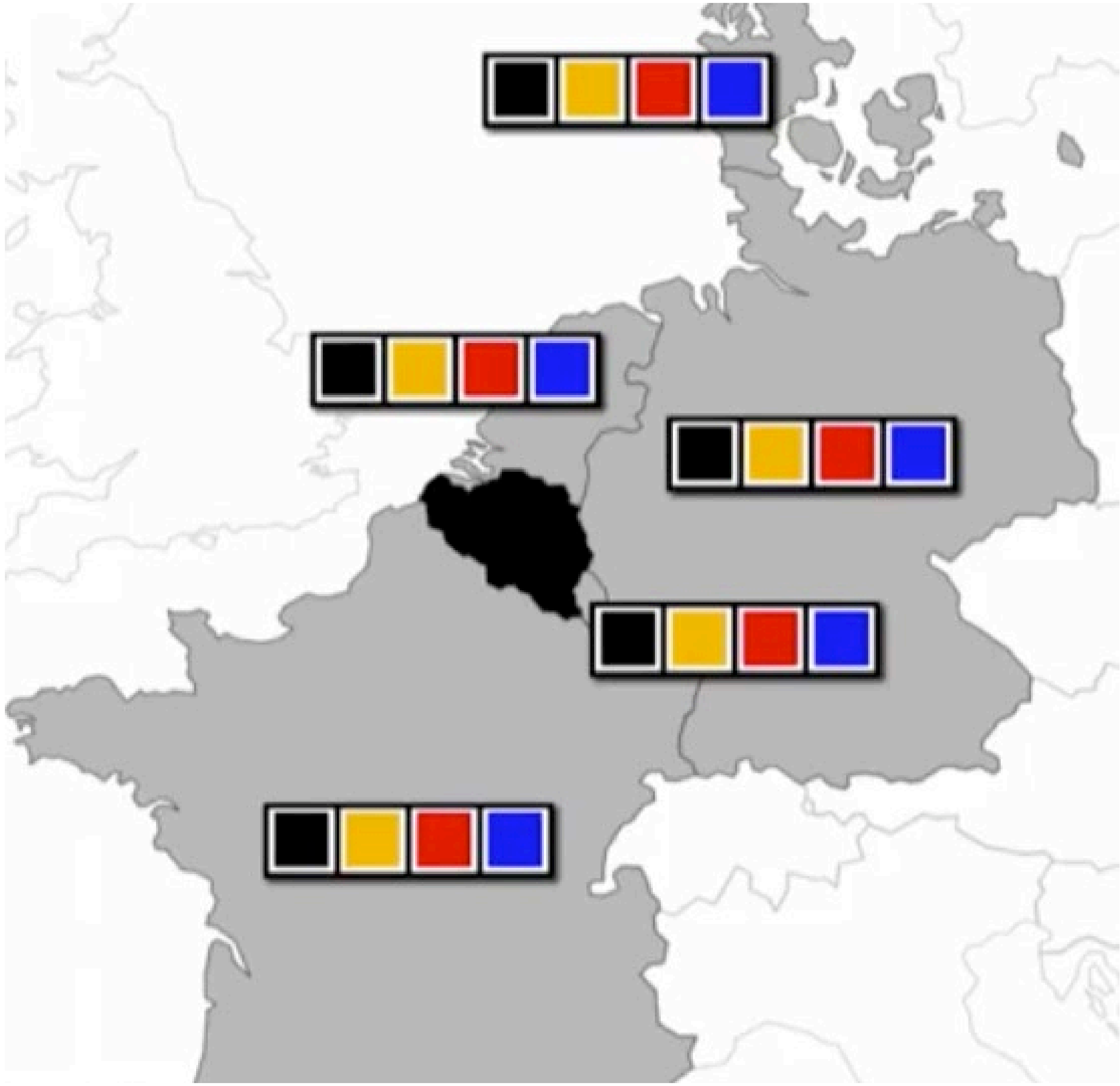
¿Empezaríamos por Dinamarca o por Bélgica?

Coloreado de mapas

- Como Dinamarca sólo tiene frontera con Alemania, estaríamos empezando por el que tiene menor probabilidad de fallar.
- Por esta razón, sería mejor empezar con Bélgica (mayor cantidad de restricciones)

minizinc: *most_constrained*

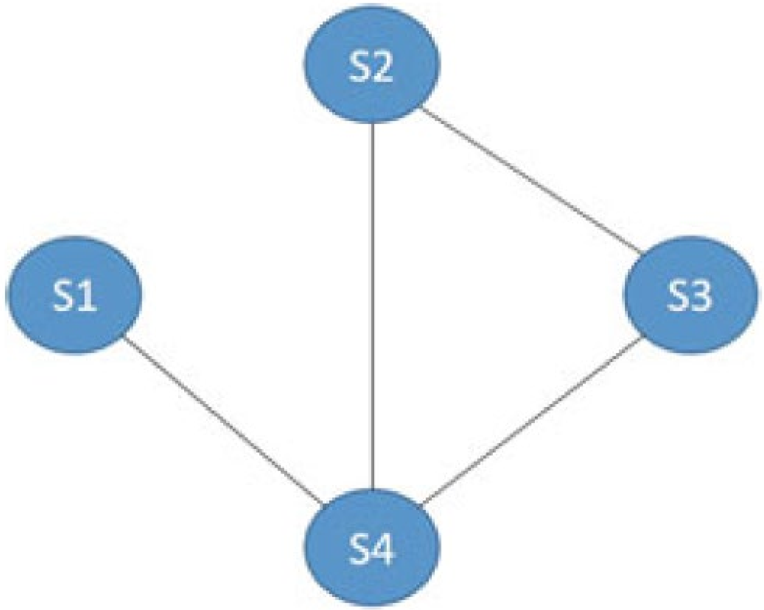
Igualdad de dominio

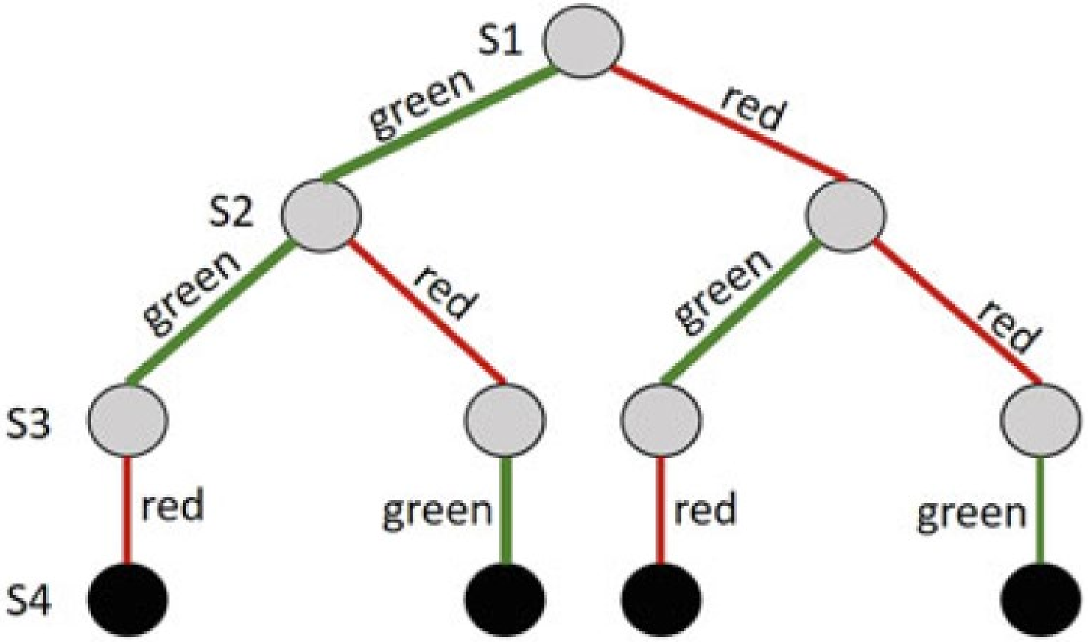


Ejemplo

- $S1, S2, S3, S4 \in \{red, green\}$

$$S4 \neq S1,\ S4 \neq S2,\ S4 \neq S3,\ S3 \neq S2$$





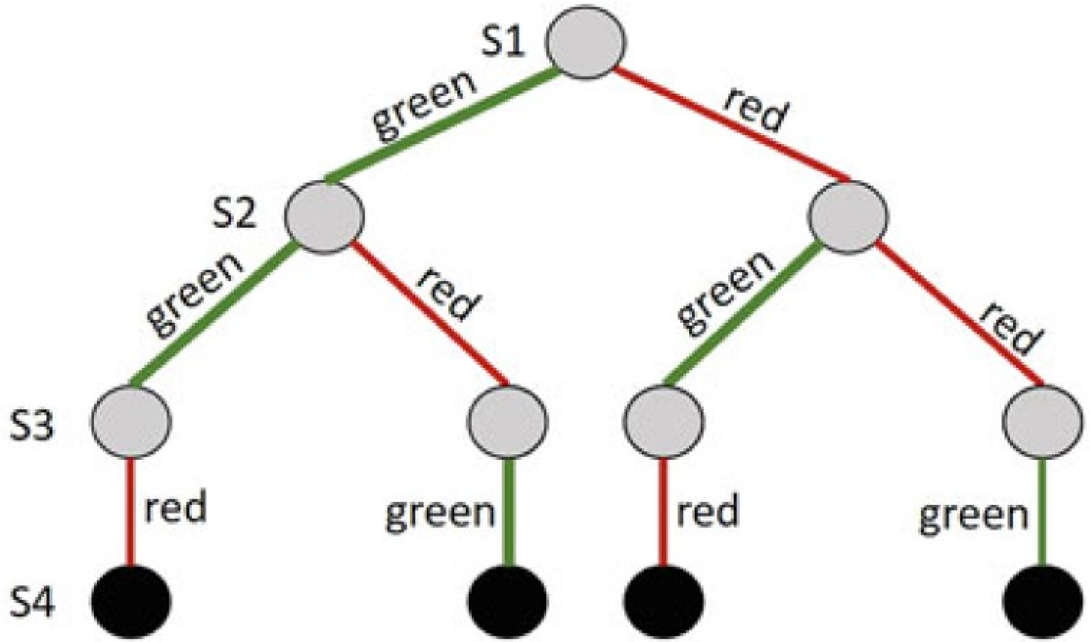
Árbol de fallo

Ejemplo

- $S1, S2, S3, S4 \in \{red, green\}$

$$S4 \neq S1,\ S4 \neq S2,\ S4 \neq S3,\ S3 \neq S2$$

```
enum col = {red,green} ;
array [1..4] of var col: S ;

constraint S[4] != S[1] ;
constraint S[4] != S[2] ;
constraint S[4] != S[3] ;
constraint S[3] != S[2] ;

solve
      :: int_search(S,input_order, indomain)
      satisfy ;
```

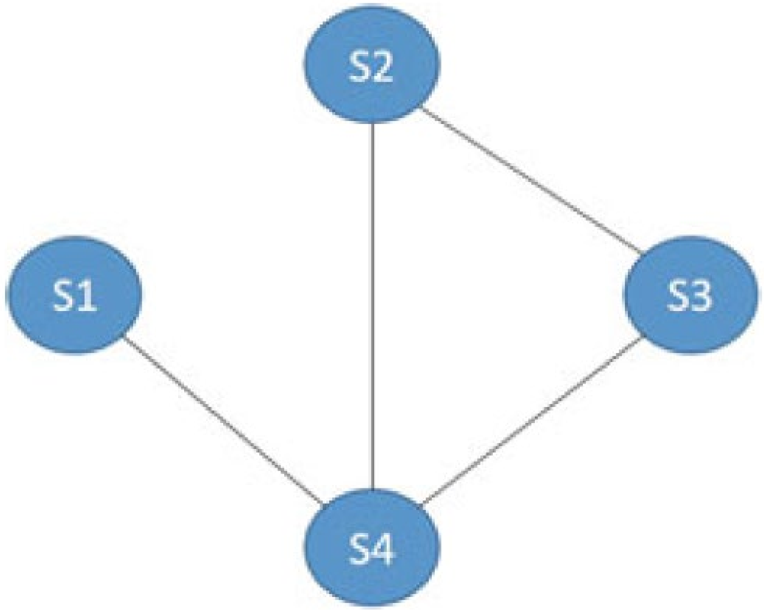


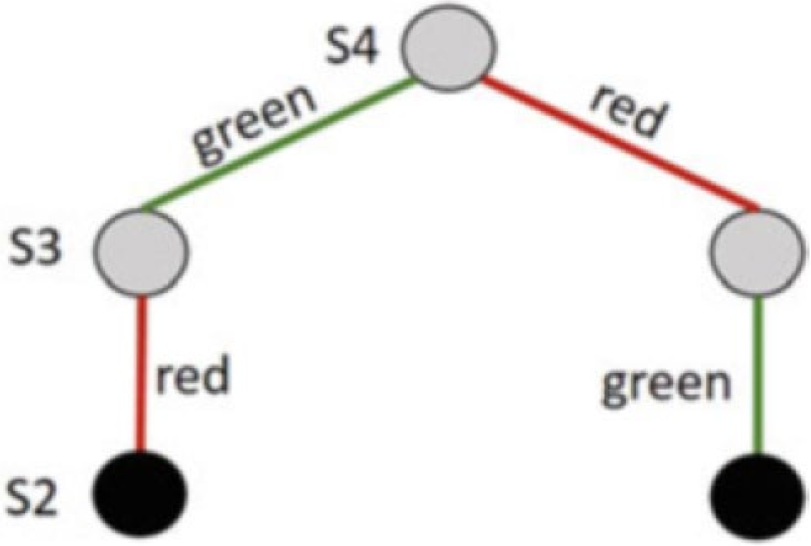
Árbol de fallo

Ejemplo

- $S1, S2, S3, S4 \in \{red, green\}$

$$S4 \neq S1,\ S4 \neq S2,\ S4 \neq S3,\ S3 \neq S2$$





Reverse(S)

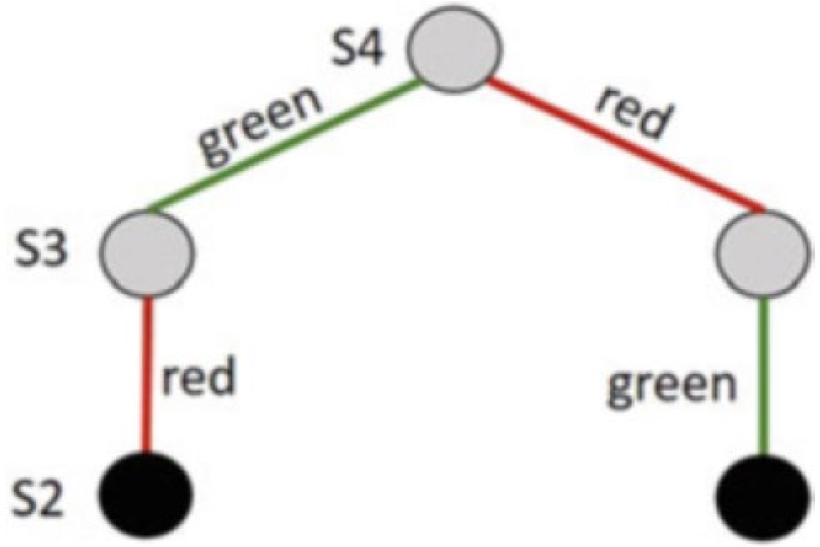
Ejemplo

- $S1, S2, S3, S4 \in \{red, green\}$

$$S4 \neq S1,\ S4 \neq S2,\ S4 \neq S3,\ S3 \neq S2$$

```
solve
     :: int_search(reverse(S),input_order,indomain)
     satisfy ;

solve
     :: int_search([S[4],S[3],S[2],S[1]],input_order,indomain)
     satisfy ;
```

Forma automática: first fail, most_constrained



Árbol de fallo

Subsistema de búsqueda

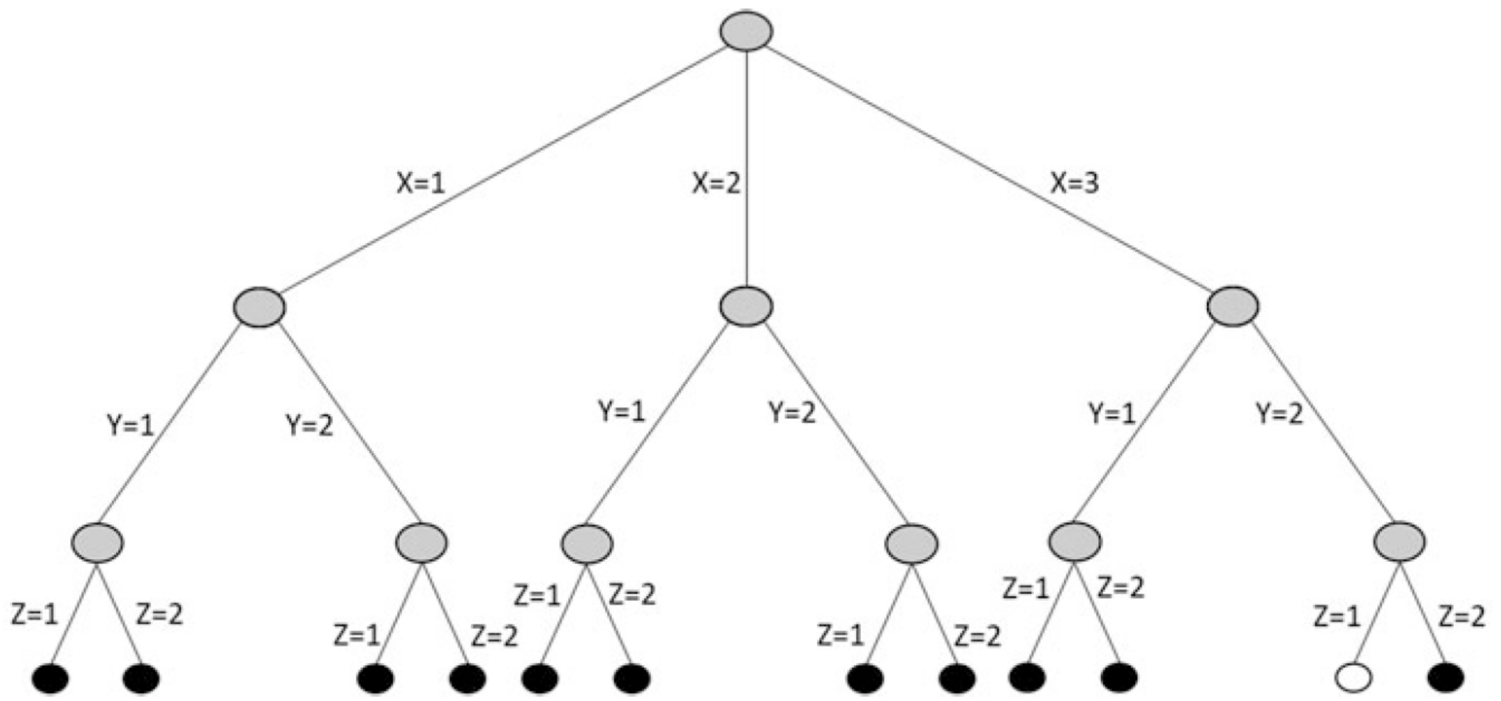
- Heurísticas de selección de variables
  - Orden de entrada (minizinc: *input_order*)
  - Dominios con valores más pequeños (minizinc: *smallest*)
  - Dominios con valores más grandes (minizinc: *largest*)
  - Principio de primer fallo (minizinc: *first_fail*)

- **Heurísticas de selección de valores**
  - Indomain_min, max, median, random
  - Indomain Split

https://www.minizinc.org/doc-2.4.2/en/lib-annotations.html#search-annotations

Ejemplo

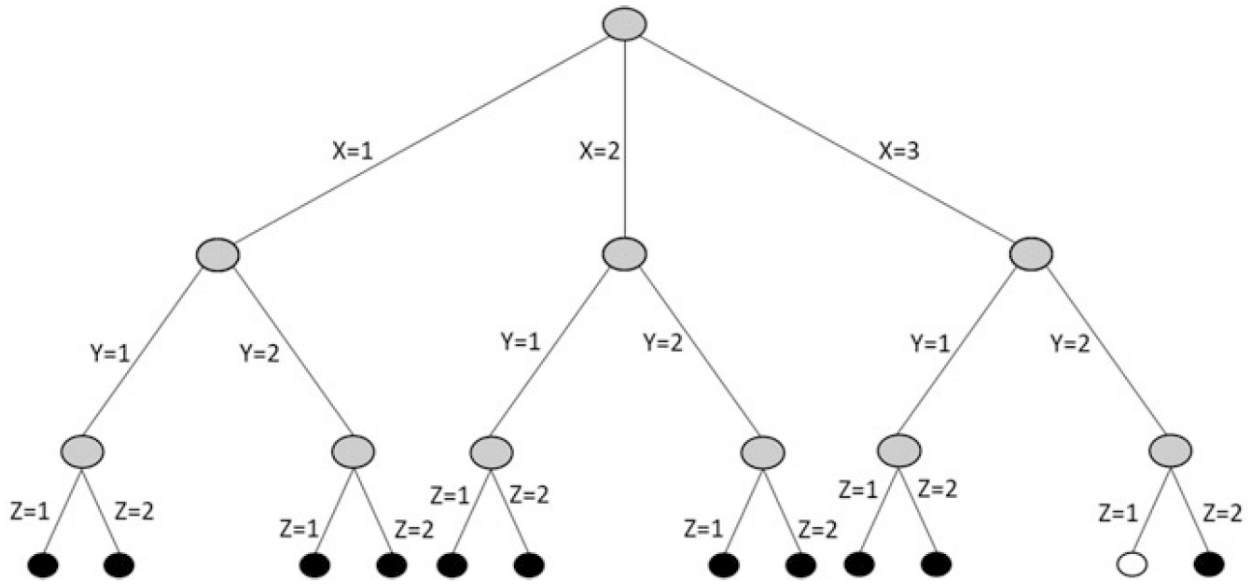
- Supongamos 3 variables de decisión: $X \in \{1,2,3\}, Y \in \{1,2\}, Z \in \{1,2\}$
- Restricción $Y \neq Z$.   Queremos maximizar $X + Y + Z$



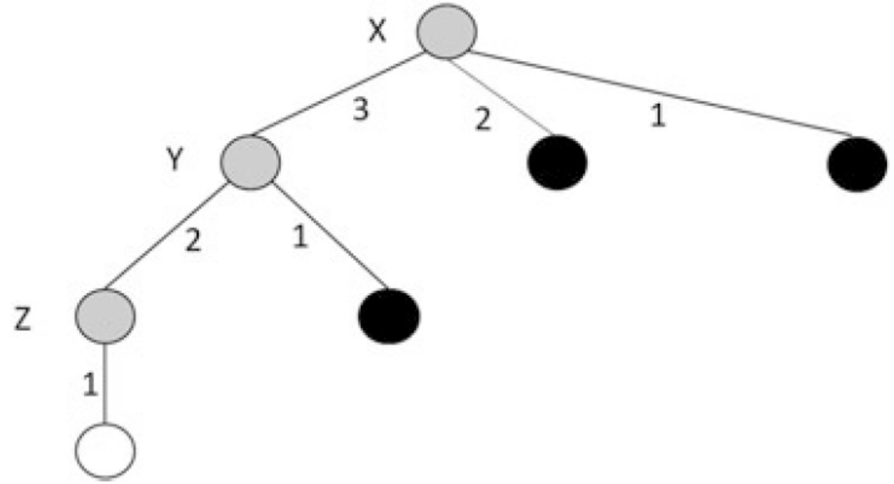
Indomain_min

Ejemplo

- Supongamos 3 variables de decisión:$X \in \{1,2,3\}, Y \in \{1,2\}, Z \in \{1,2\}$
- Restricción $Y \neq Z$.   Queremos **maximizar** $X + Y + Z$



Branch and Bound con DFS ***Indomain_min***



Indomain_max

Ejemplo de las 8 reinas con minizinc

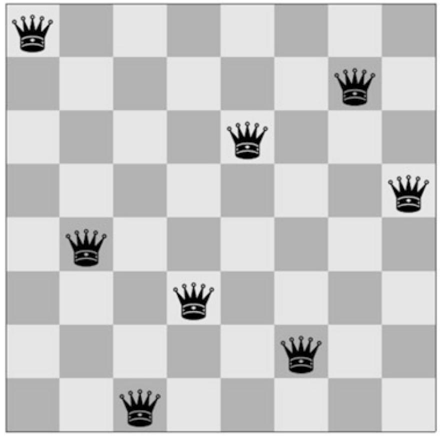
```
include "globals.mzn";

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

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

```
solve :: int_search(row,
                    first_fail,
                    indomain_min)
                    satisfy;
```

Combinación de heurísticas de búsqueda

Q4,Q5,Q3,Q6,Q2,Q7,Q1,Q8

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



Search steps	Naive	(A) Middle-out Var	(B) First fail	(A+B)	(A+B)+ Middle-out Val
8-queens	10	0	17	0	5
16-queens	542	28	4	0	7
32-queens	Timeout	Timeout	9	1	15
64-queens	Timeout	Timeout	365	3	2
128-queens	Timeout	Timeout	Timeout	Timeout	0

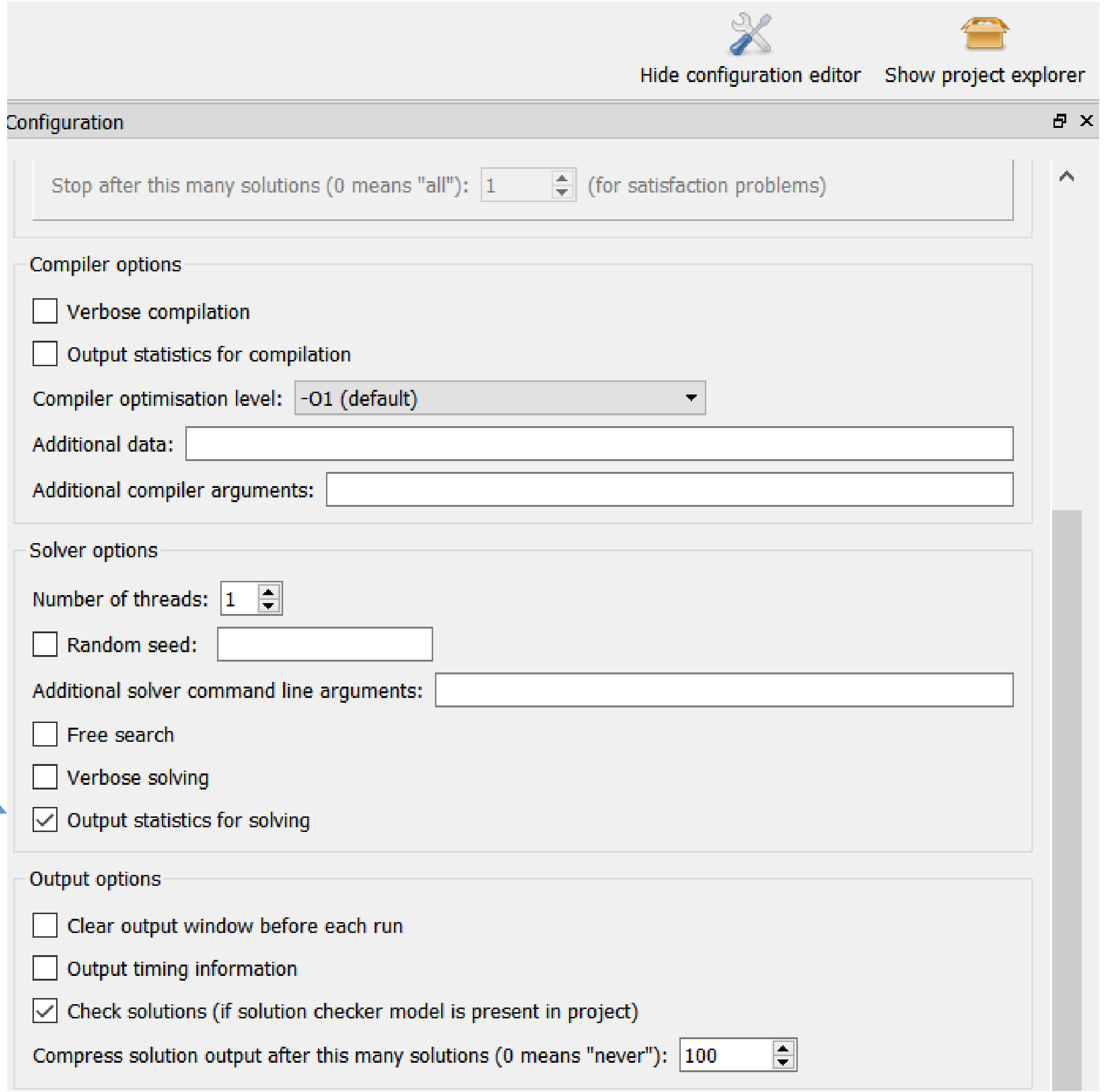
- Naive: orden de variables: *Q*1,*Q*2, . . . , *Q*8, orden de valores: 1,2,…8
- A: orden de variables: Las variables se cogen del centro hacia los lados, orden de valores: 1,2…8
- B: orden de variables: First fail, orden de valores: 1,2, …8
- A+B: Igual a B, cuando 2 variables tienen la misma cardinalidad coge la más cercana al centro
- A+B+middle-out-val: Igual a (A+B) pero el orden de valores sigue la fila más cercana al centro del tablero, 4,5,3,6,2,7,1,8

Ejemplo de sintaxis en minizinc

```
%Variables de decisión
…
array[POS] of var int: difference;
```

```
%ann: varsel = input_order;
%ann: varsel = first_fail;
ann: varsel = smallest;
%ann: varsel = largest;
ann: valsel = indomain_min;
%ann: valsel = indomain_max;
%ann: valsel = indomain_median;
%ann: valsel = indomain_random;


solve:: int_search(difference,
                   varsel,
                   valsel,)
  maximize obj;
```

Activación de las estadísticas en minizinc



Output

```
----------
==========
%%%mzn-stat initTime=0.001
%%%mzn-stat solveTime=0.274
%%%mzn-stat solutions=2
%%%mzn-stat variables=69
%%%mzn-stat propagators=67
%%%mzn-stat propagations=1250488
%%%mzn-stat nodes=140363
%%%mzn-stat failures=70176
%%%mzn-stat restarts=0
%%%mzn-stat peakDepth=22
%%%mzn-stat-end
Finished in 419msec
```

Resumen

- Ejemplo: Problema de las *n* reinas
- Ejemplo: Coloreado de mapas
- Series mágicas
- Sudoku
- Ruptura de simetrías
- Restricciones redundantes
- Modelos duales
- Restricciones globales
- Subsistema de búsqueda.
- Estrategias de selección de variable
- Estrategias de selección de valores
- Caballo de Euler

VPL: TSP con trayecto parcial mínimo

Dado un problema de TSP (circuito hamiltoniano), a diferencia del problema clásico donde el objetivo es minimizar la distancia total del recorrido, queremos en este caso que el objetivo sea **minimizar** la distancia mayor entre cada par de ciudades, es decir, queremos minimizar la distancia del máximo trayecto parcial.

Parámetros

```
include "globals.mzn";
int: numCities;       %número de ciudades
set of int: City = 1..numCities;
int: maxAllowedEdge;     % máxima distancia permitida de un trayecto del recorrido

% distancia entre ciudades
% -1 significa que no hay conexión directa
array[City,City] of int: distance;
```

Ejemplo

-1 => no hay camino entre las ciudades

```
maxAllowedEdge = 600;
numCities = 15;
distance = [|0,-1,250,-1,-1,473,-1,172,-1,372,360,414,-1,-1,243
 |-1,0,-1,-1,99,161,284,-1,-1,-1,446,431,478,262,457
 |250,-1,0,-1,-1,-1,573,408,697,159,281,332,-1,-1,219
 |-1,-1,-1,0,-1,-1,296,-1,400,481,392,338,172,449,-1
 |-1,99,-1,-1,0,247,196,-1,-1,-1,410,384,380,181,446
 |473,161,-1,-1,247,0,391,519,-1,-1,448,453,-1,422,416
 |-1,284,573,296,196,391,0,-1,670,480,293,247,224,233,375
 |172,-1,408,-1,-1,519,-1,0,-1,542,524,576,-1,-1,407
 |-1,-1,697,400,-1,-1,670,-1,0,545,605,577,567,-1,-1
 |372,-1,159,481,-1,-1,480,542,545,0,201,234,-1,-1,216
 |360,446,281,392,410,448,293,524,605,201,0,56,445,-1,118
 |414,431,332,338,384,453,247,576,577,234,56,0,390,478,171
 |-1,478,-1,172,380,-1,224,-1,567,-1,445,390,0,295,-1
 |-1,262,-1,449,181,422,233,-1,-1,-1,-1,478,295,0,-1
 |243,457,219,-1,446,416,375,407,-1,216,118,171,-1,-1,0|];
```

Resultado esperado para el ejemplo

```
succ = array1d(1..15, [8, 7, 10, 11, 6, 14, 15, 3, 4, 9, 12, 13, 5, 2, 1]);
maxEdge = 545;
----------
==========
Finished in 192msec
```