



Algoritmos y Programación

Programación con
Restricciones

Bloque 2: Algoritmos y
Computabilidad

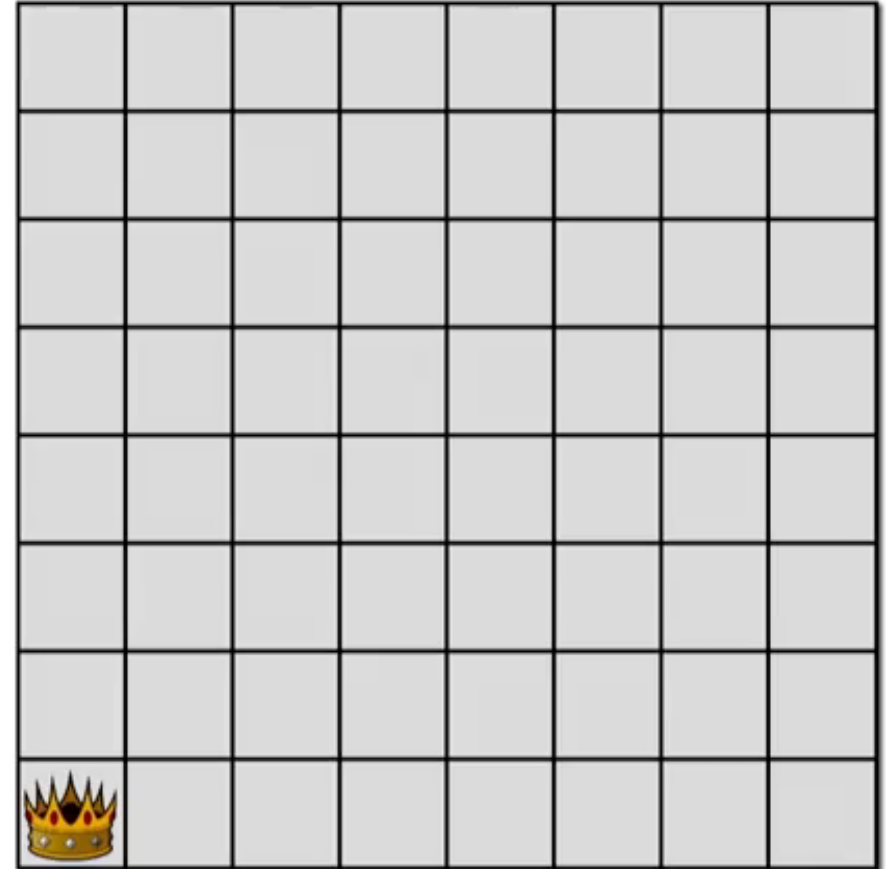
Programación con restricciones

- Jaffar y Lassez. 1987
- Problemas combinatorios y de optimización
- Devuelve resultados exactos (no es heurístico)
- Modelado de Problemas
- Desarrollo de Resolutores
- El paradigma computacional consiste básicamente en:

Reducir el conjunto de valores que una variable puede tomar en base a las restricciones establecidas

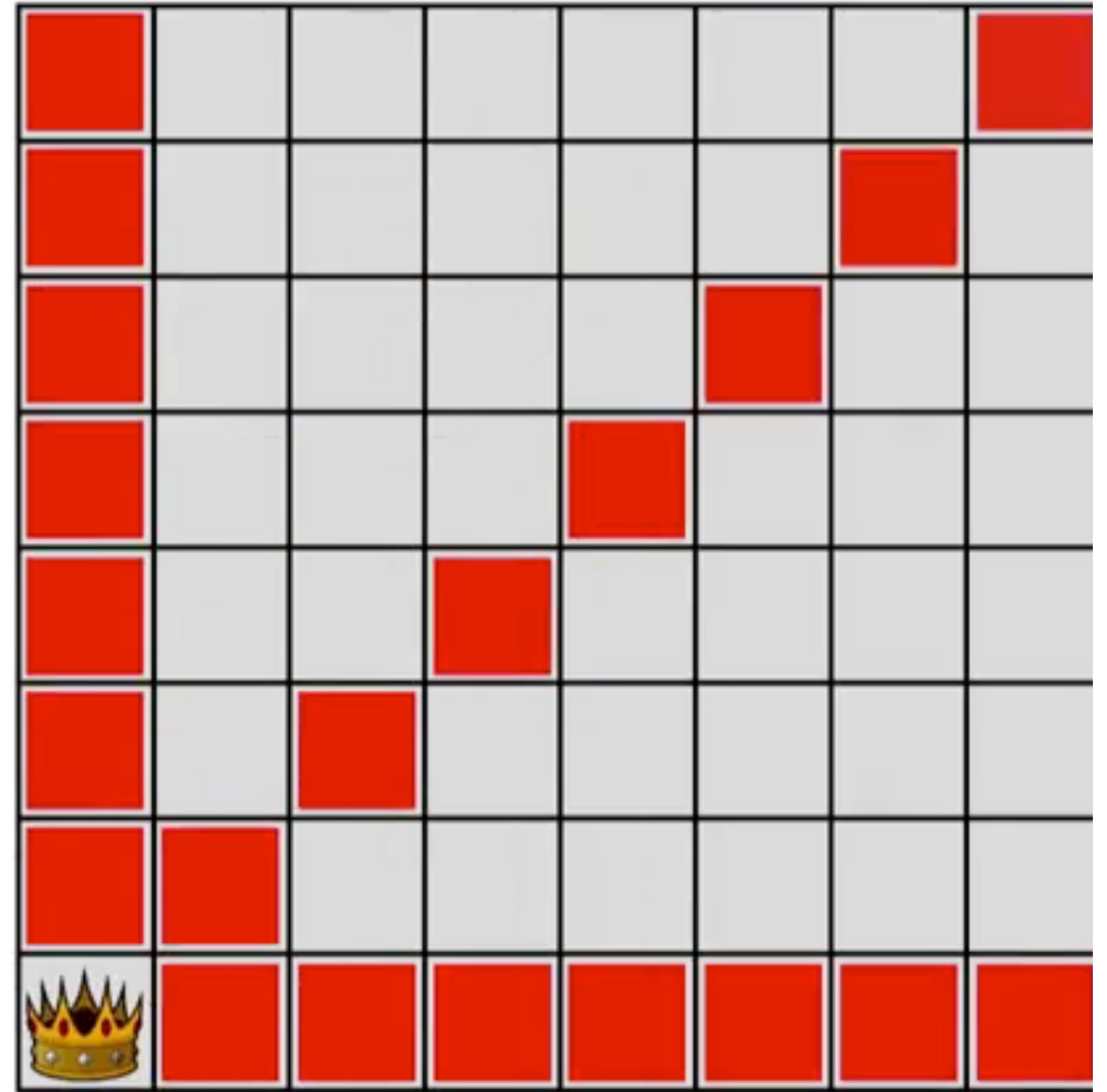
Problema de n reinas

- Colocar 8 reinas en un tablero de 8x8 de tal forma que no se encuentran 2 de ellas en la misma fila, misma columna o misma diagonal
- Generar todas las formas posibles de situar las reinas en el tablero, comprobando si se ha encontrado una solución implicaría recorrer un espacio de búsqueda de $\binom{n^2}{n}$ que para el caso de $n=8$ supondría 4.426.165.368 posibilidades.



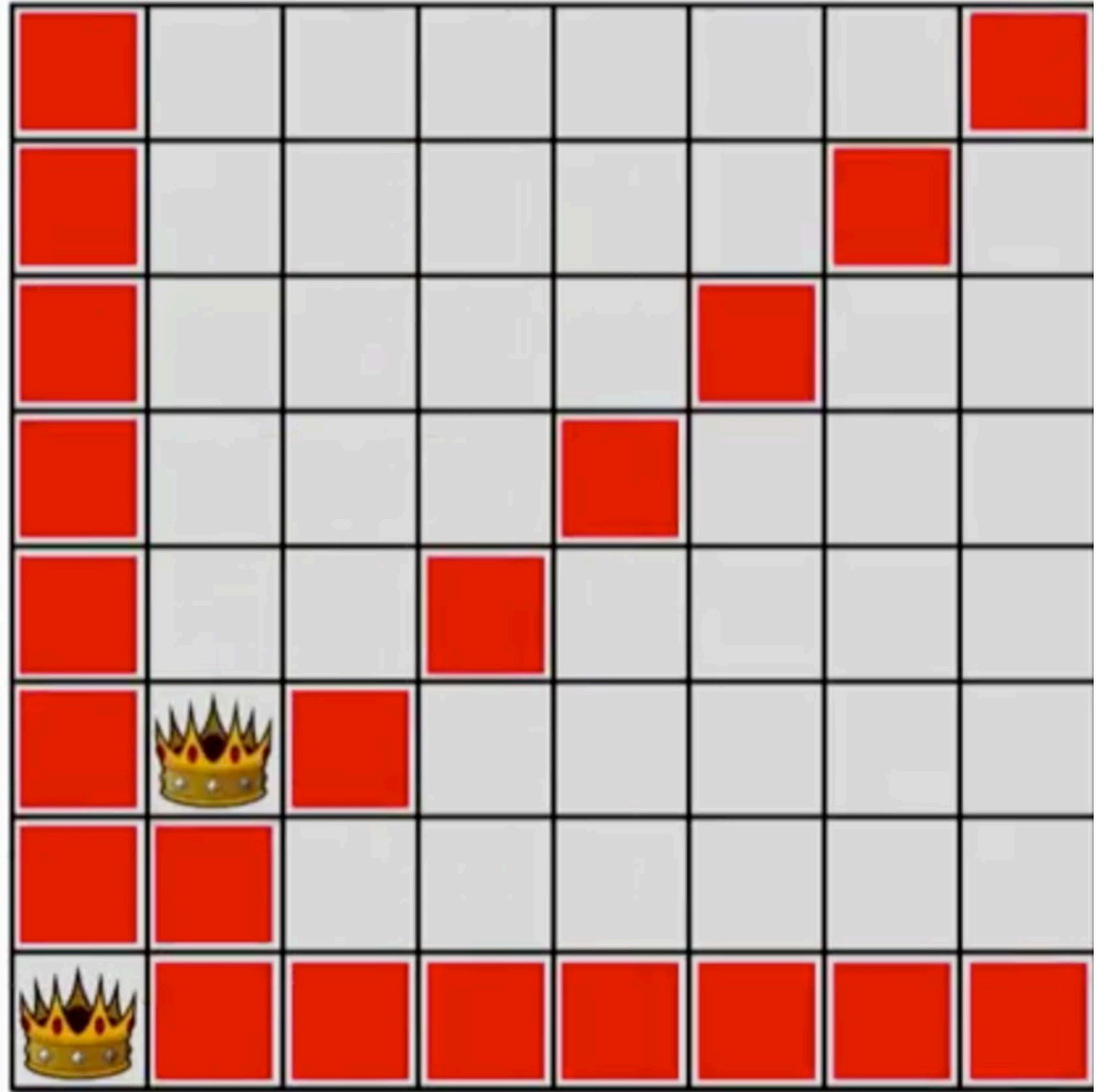
Problema de n reinas

- El objetivo es reducir al máximo el espacio de búsqueda, en base a las restricciones establecidas



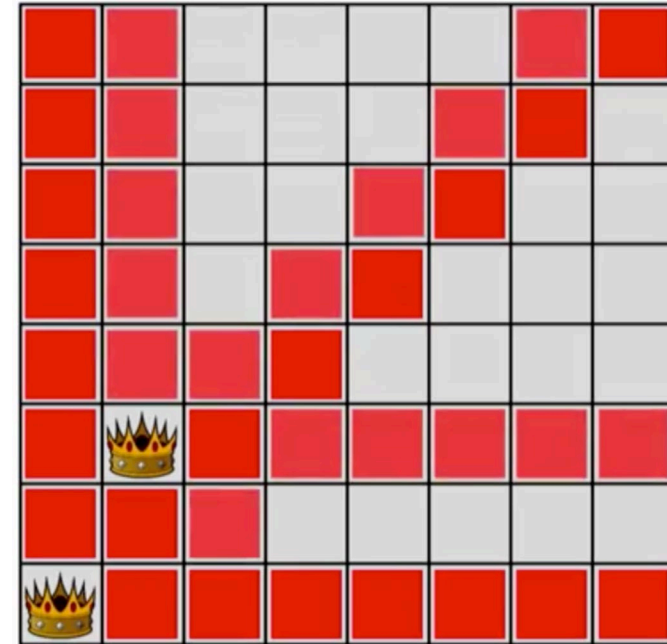
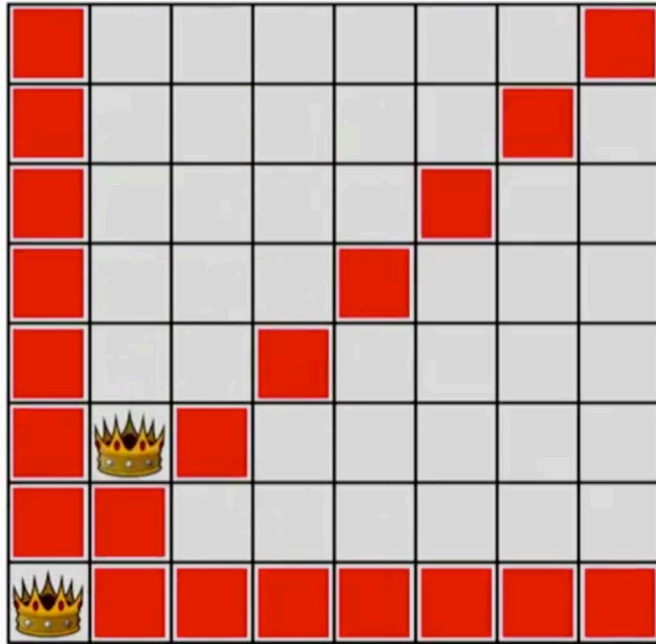
Problema de n reinas

- Ahora podemos colocar una nueva reina



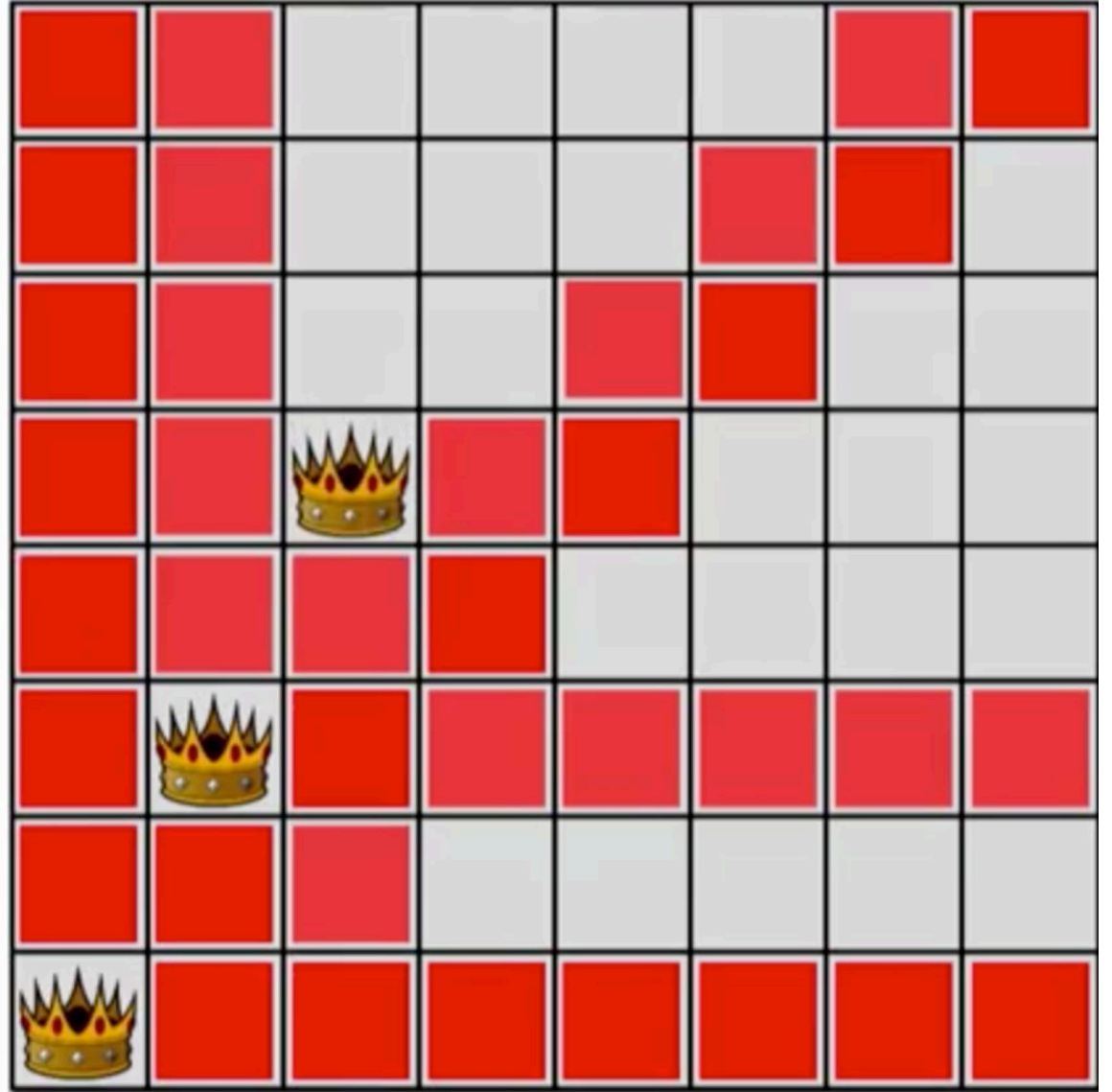
Problema de n reinas

- Continuamos reduciendo el espacio de búsqueda



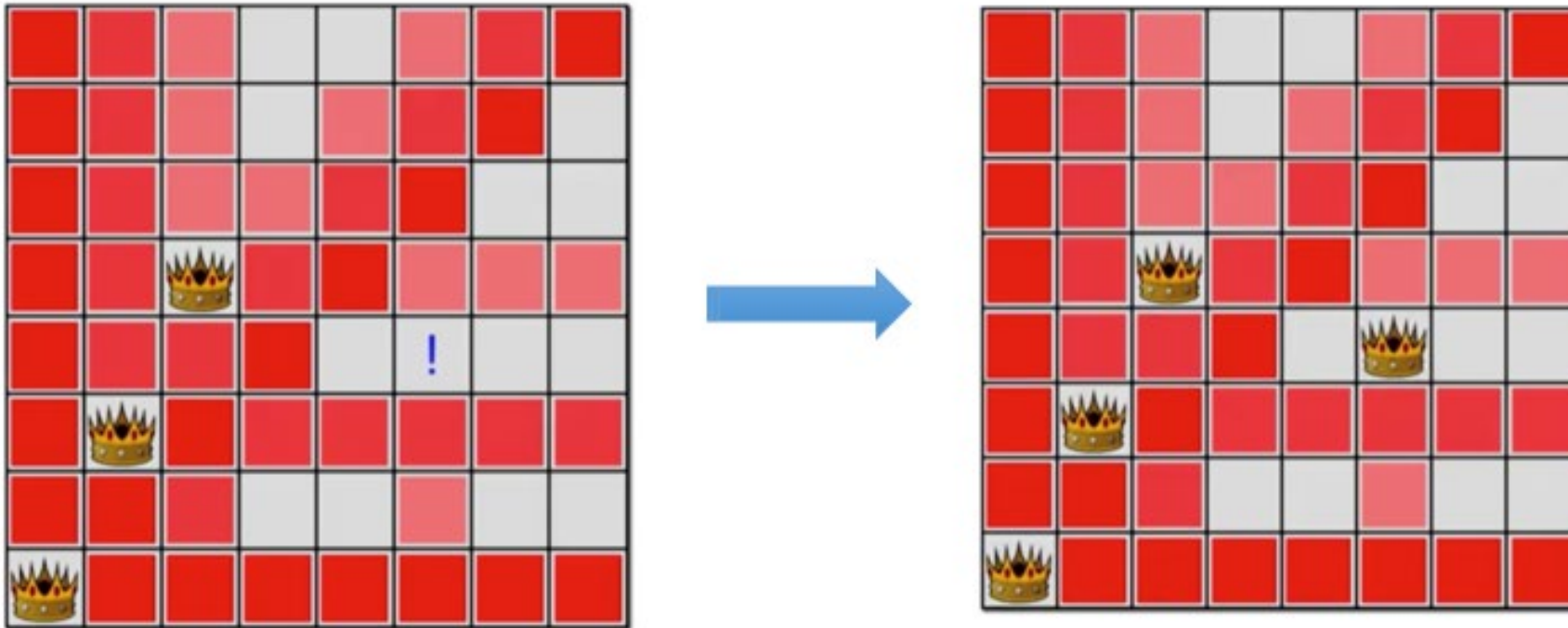
Problema de n reinas

- Colocamos la tercera reina



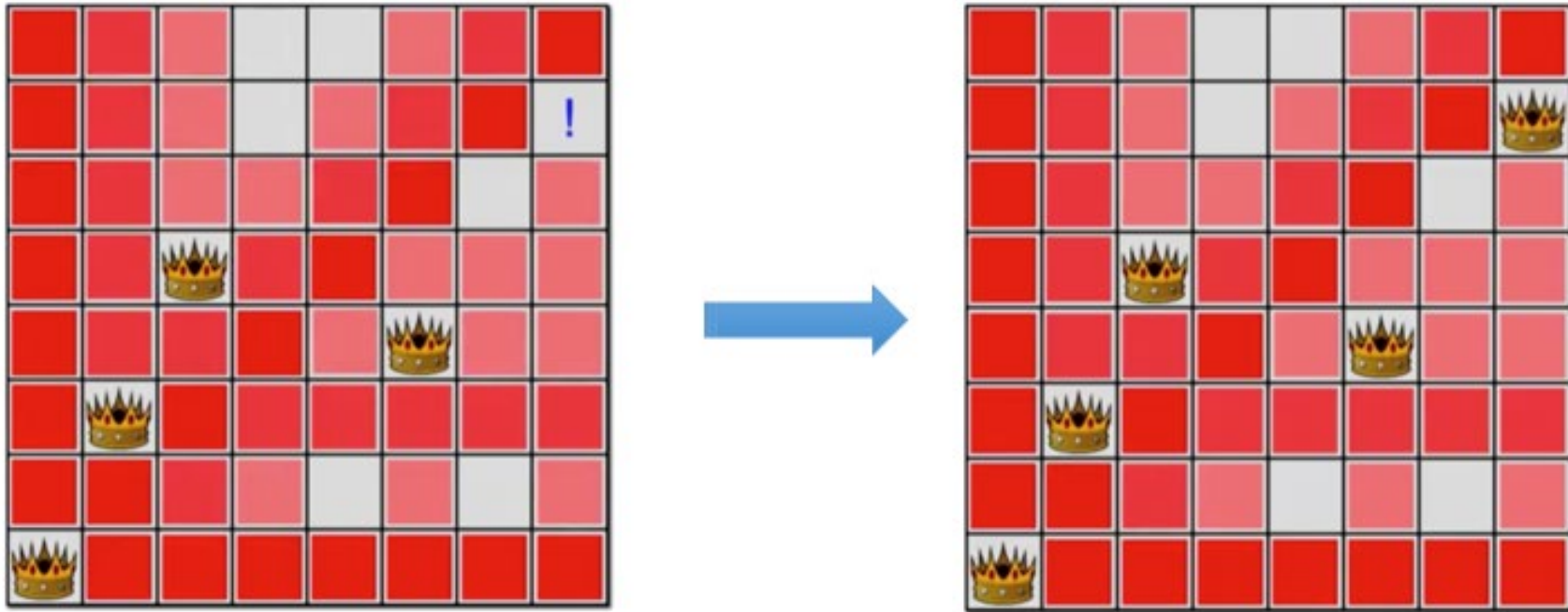
Problema de n reinas

- Observamos que sólo hay un hueco factible donde colocar la cuarta reina



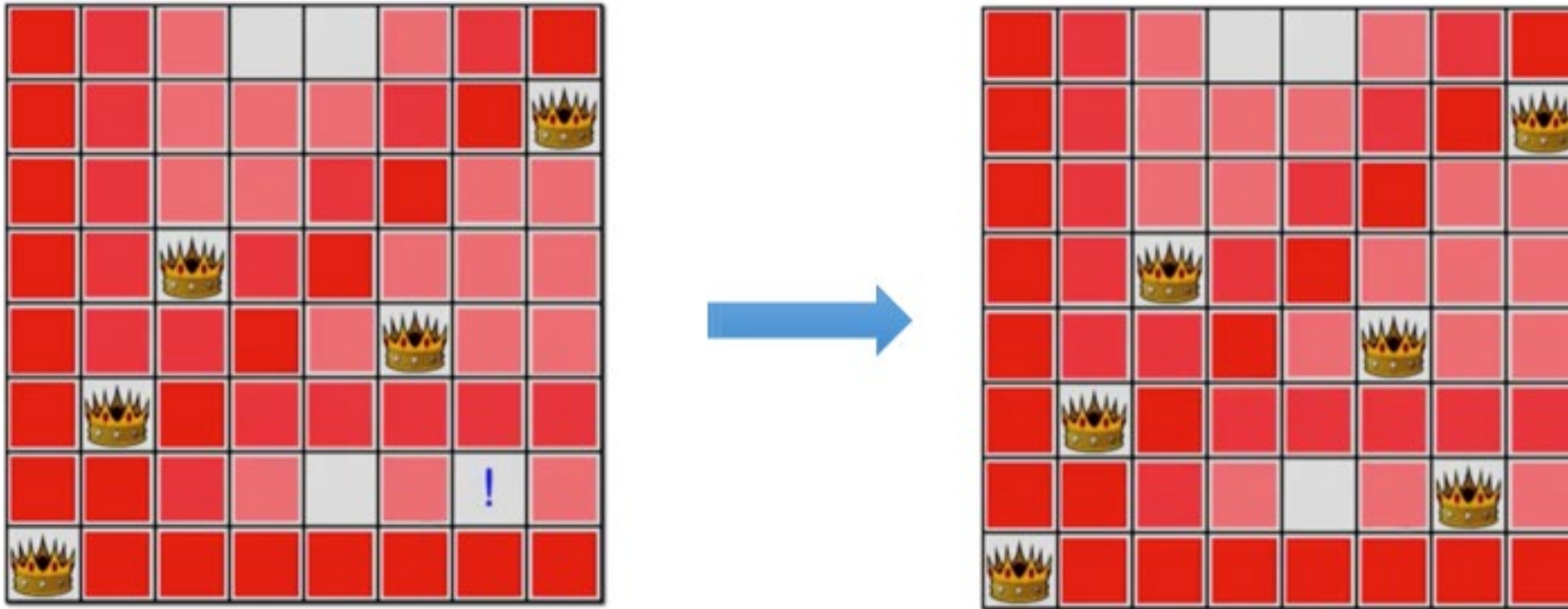
Problema de n reinas

- Lo mismo ocurre con la quinta reina



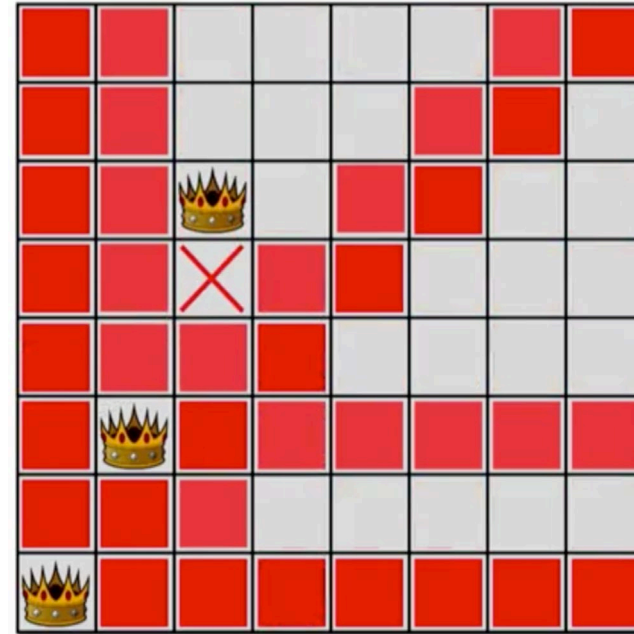
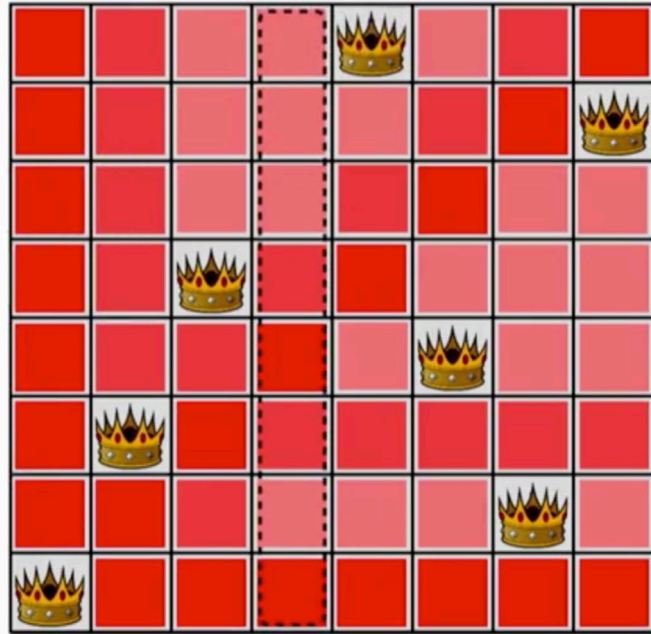
Problema de n reinas

- Al colocar la sexta reina, podemos ver que en esta configuración no se puede conseguir una solución factible

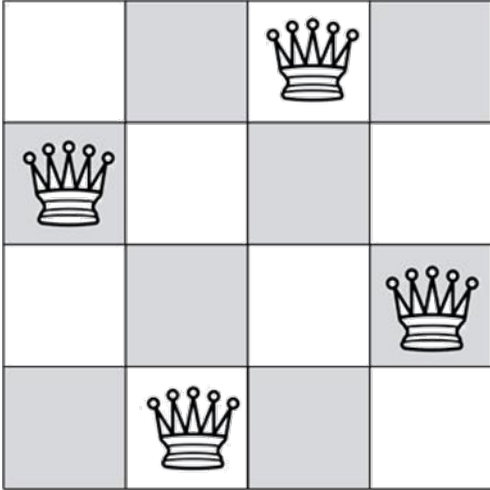


Problema de n reinas

- Al colocar la sexta reina, podemos ver que en esta configuración no se puede conseguir una solución factible. Por tanto, ... hacemos “backtrack”



Algoritmos para búsquedas



Generate and Test
(GT)

Backtracking
(un paso atrás)

Variantes de backtracking

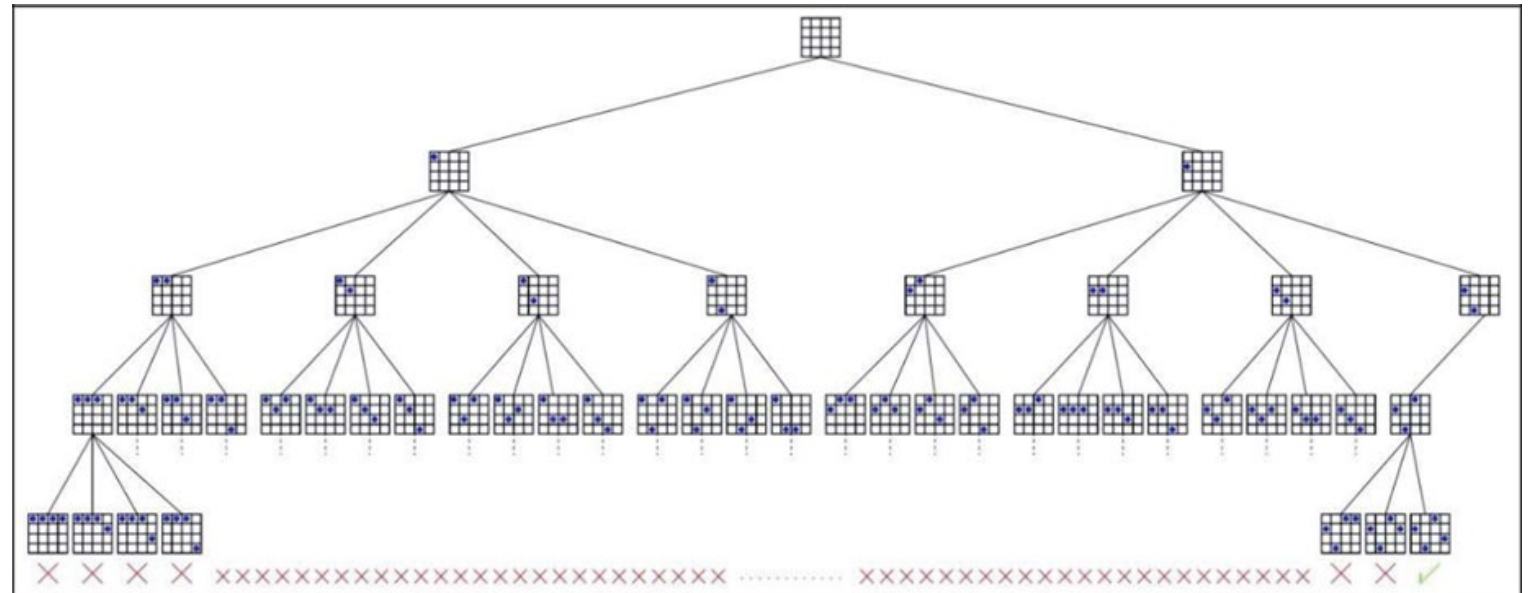
Esquemas look
Back

- Backjumping

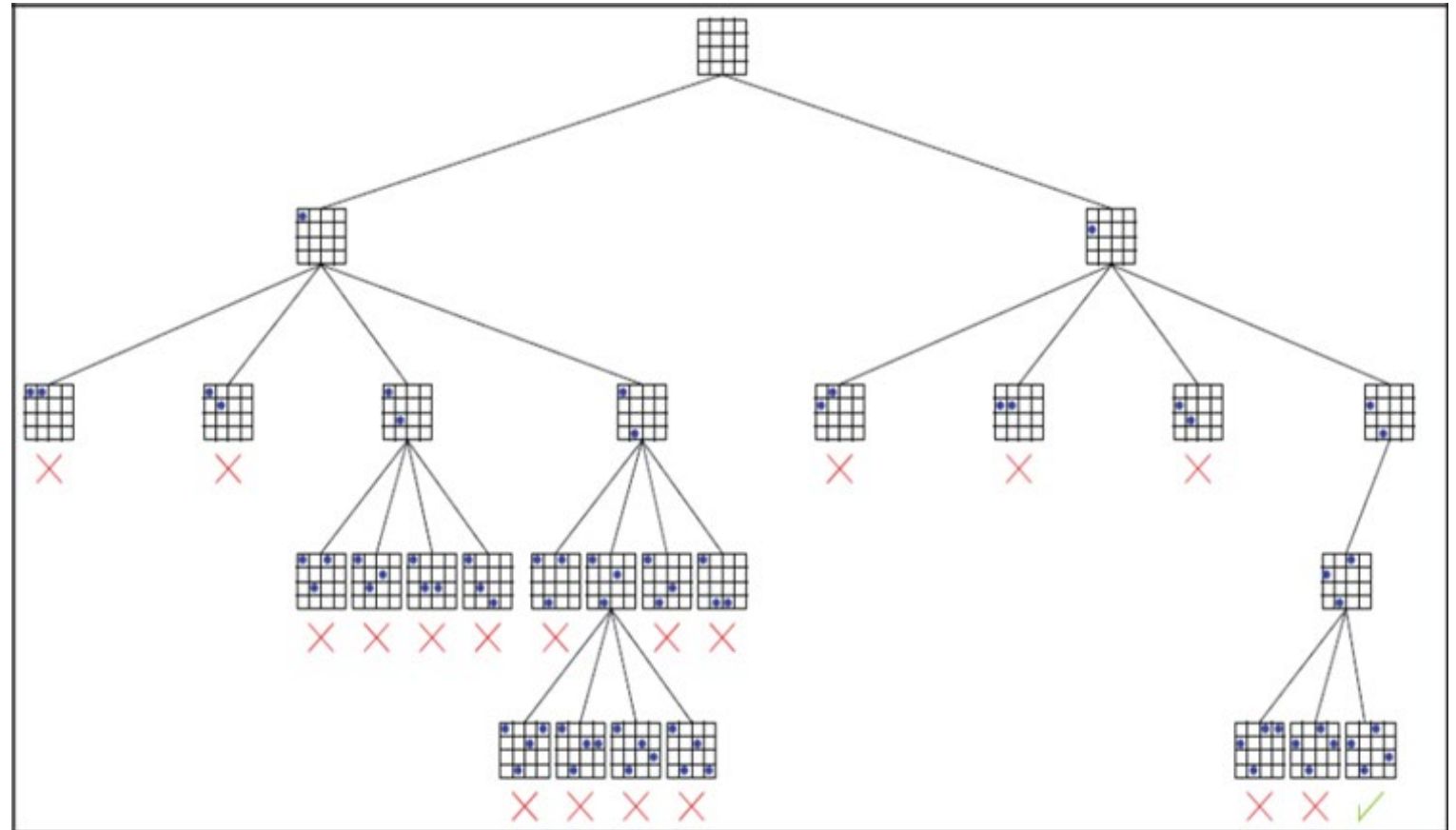
Esquemas look
Ahead

- Forward Checking
- Maintaining Arc Consistency

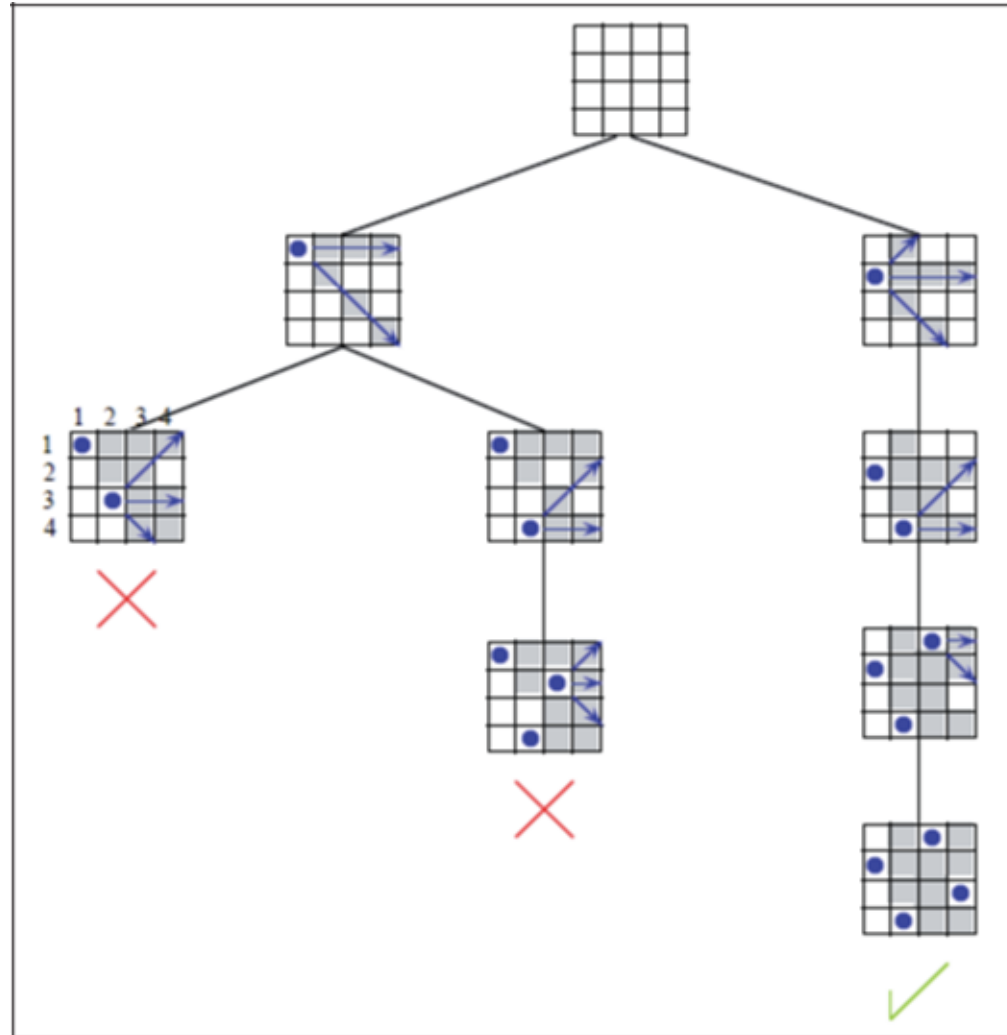
Generate and Test



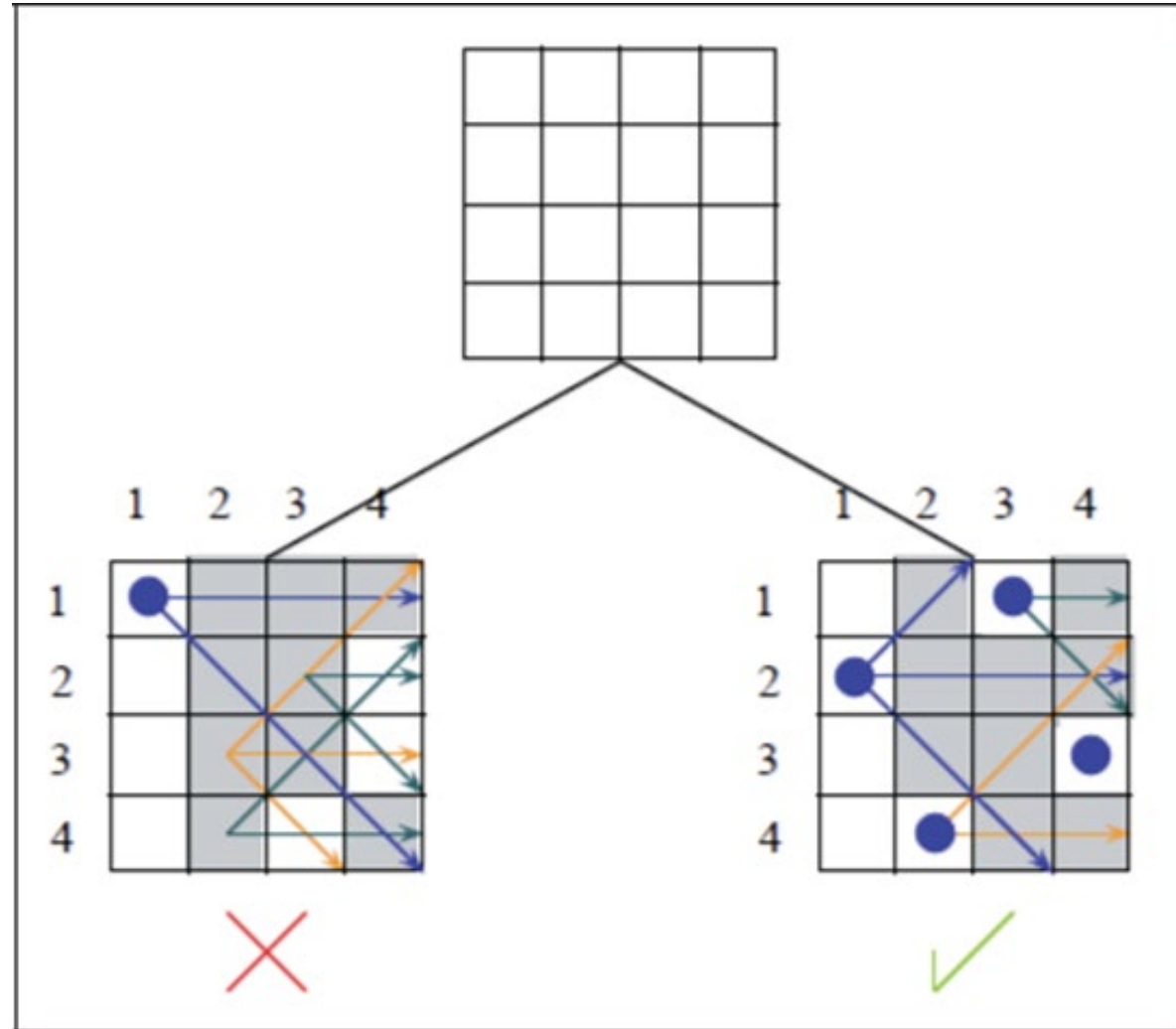
Backtracking



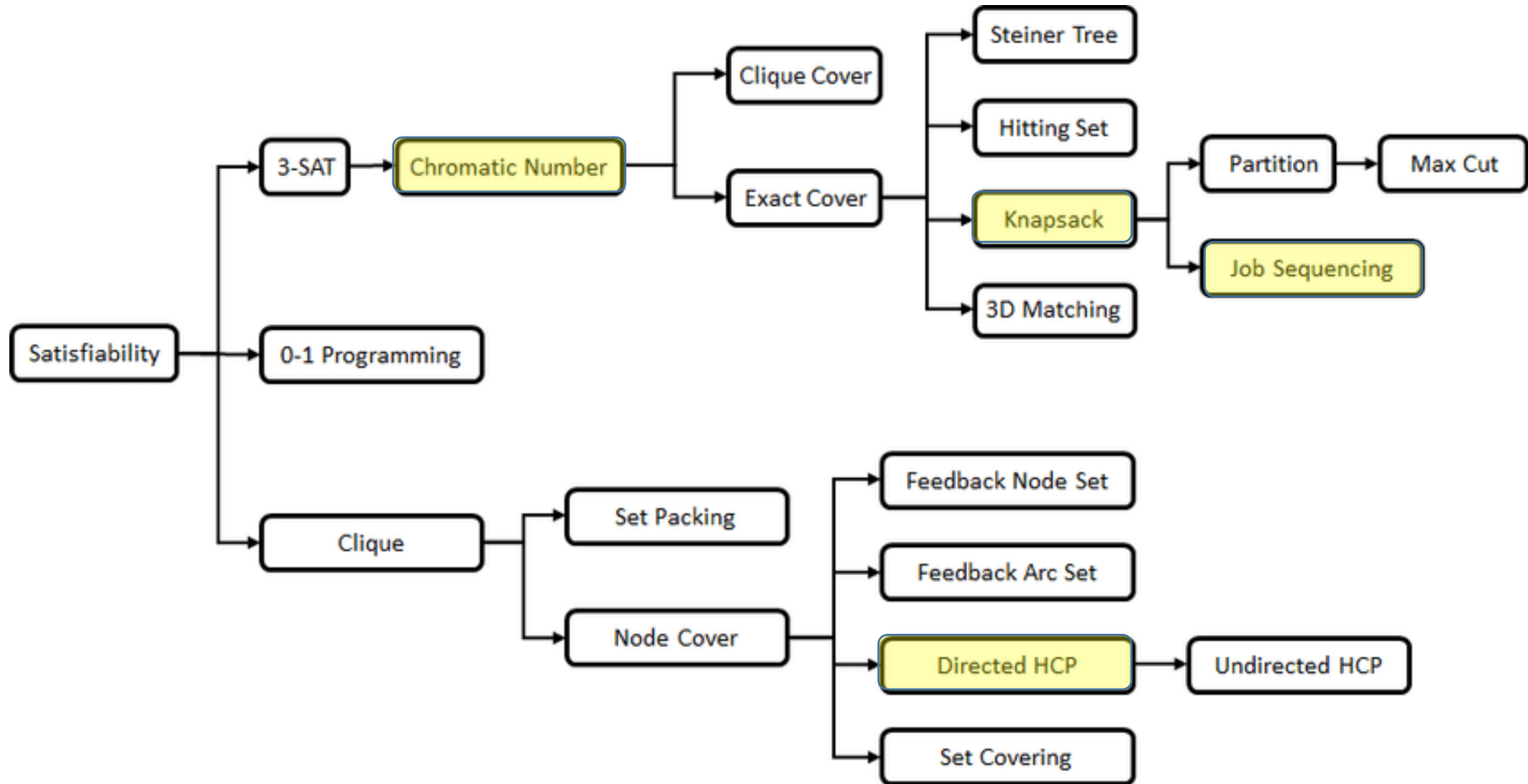
Forward Checking

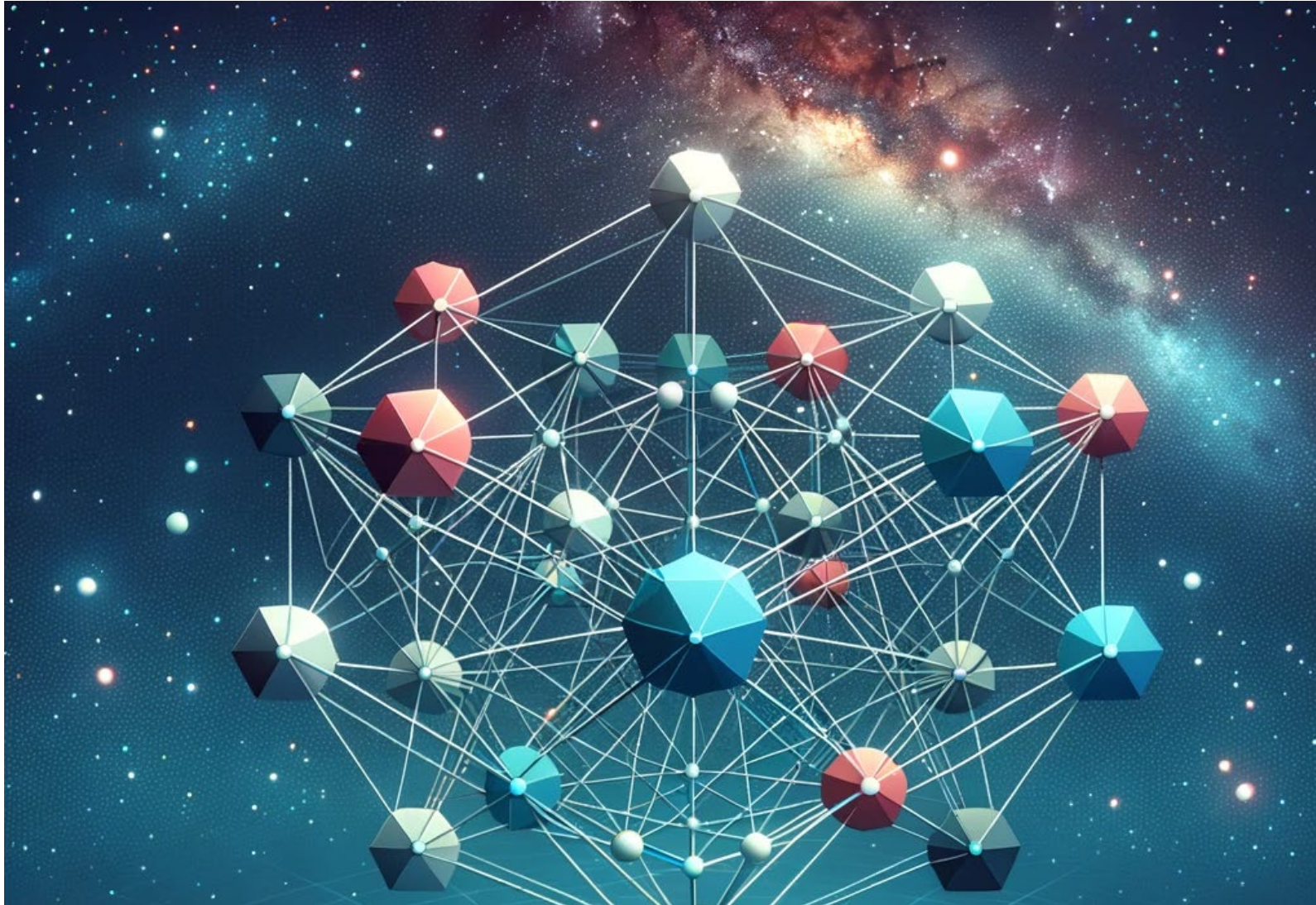


Maintaining Arc Consistency



21 problemas de Karp

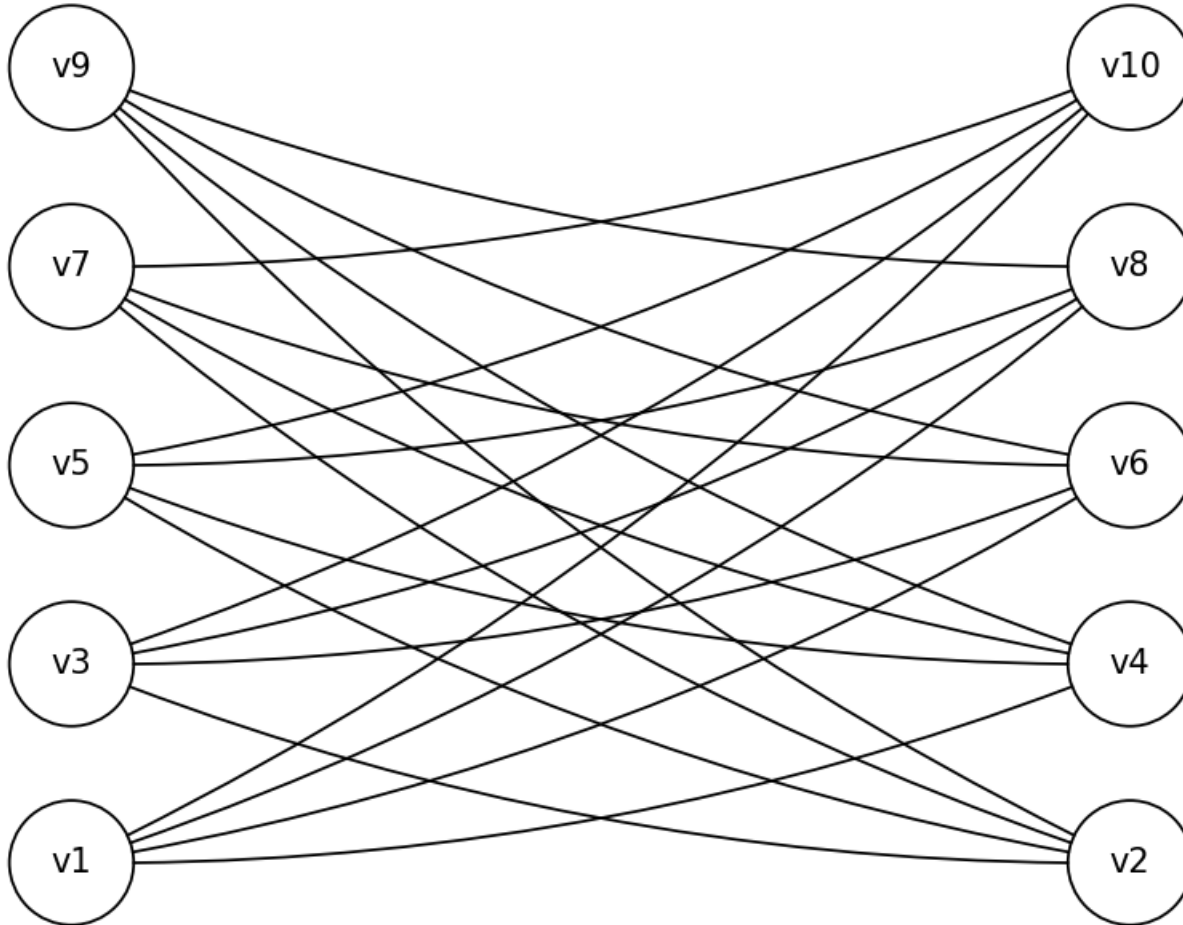




Número cromático

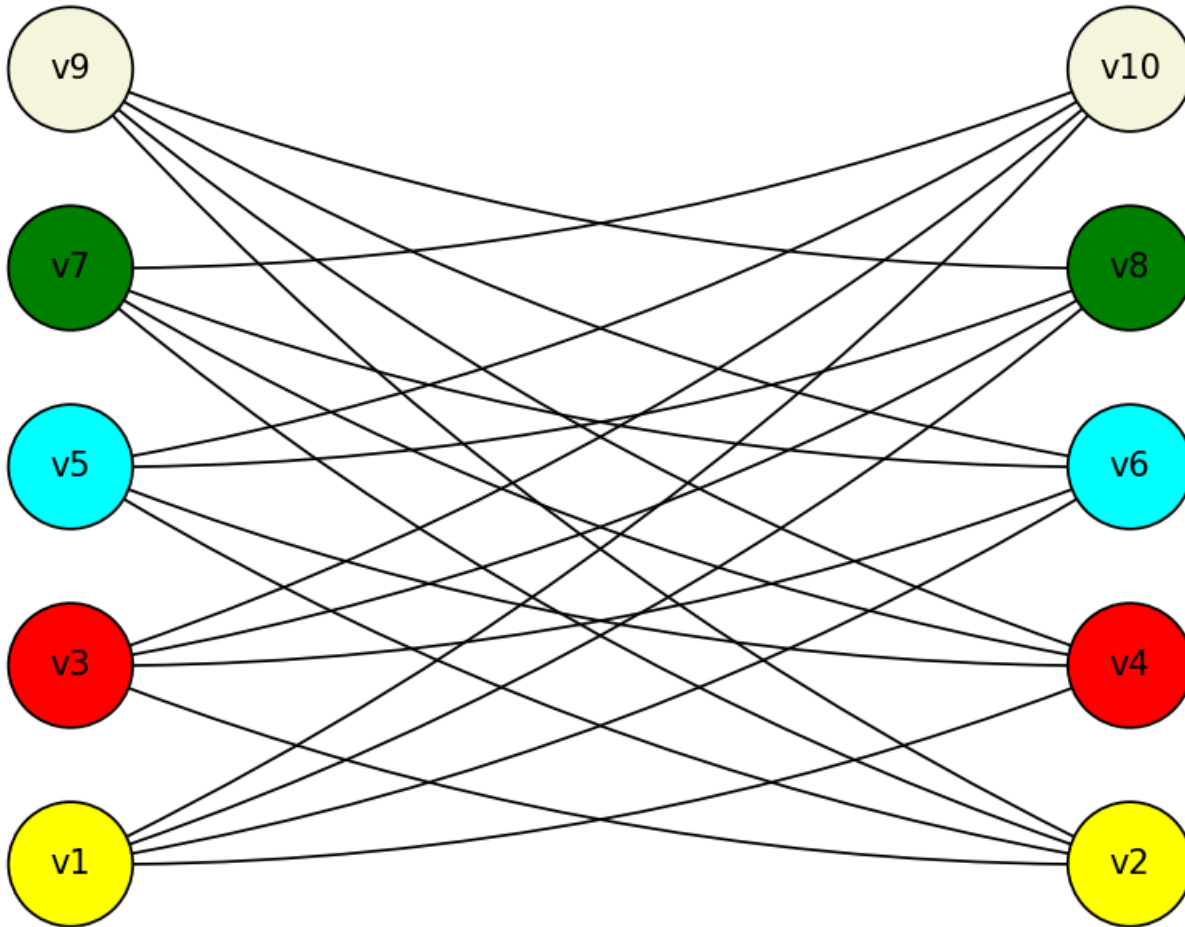
Coloreado de un grafo

Estrategia Greedy (Voraz)



- Depende del orden de visita de los nodos
- Se empieza asignando el primer color al primer nodo
- Mientras se pueda aplicar ese color se sigue avanzando en el orden de los nodos
- Cuando haya un conflicto se incrementa el color asignado

Estrategia Greedy (Voraz)



Orden:

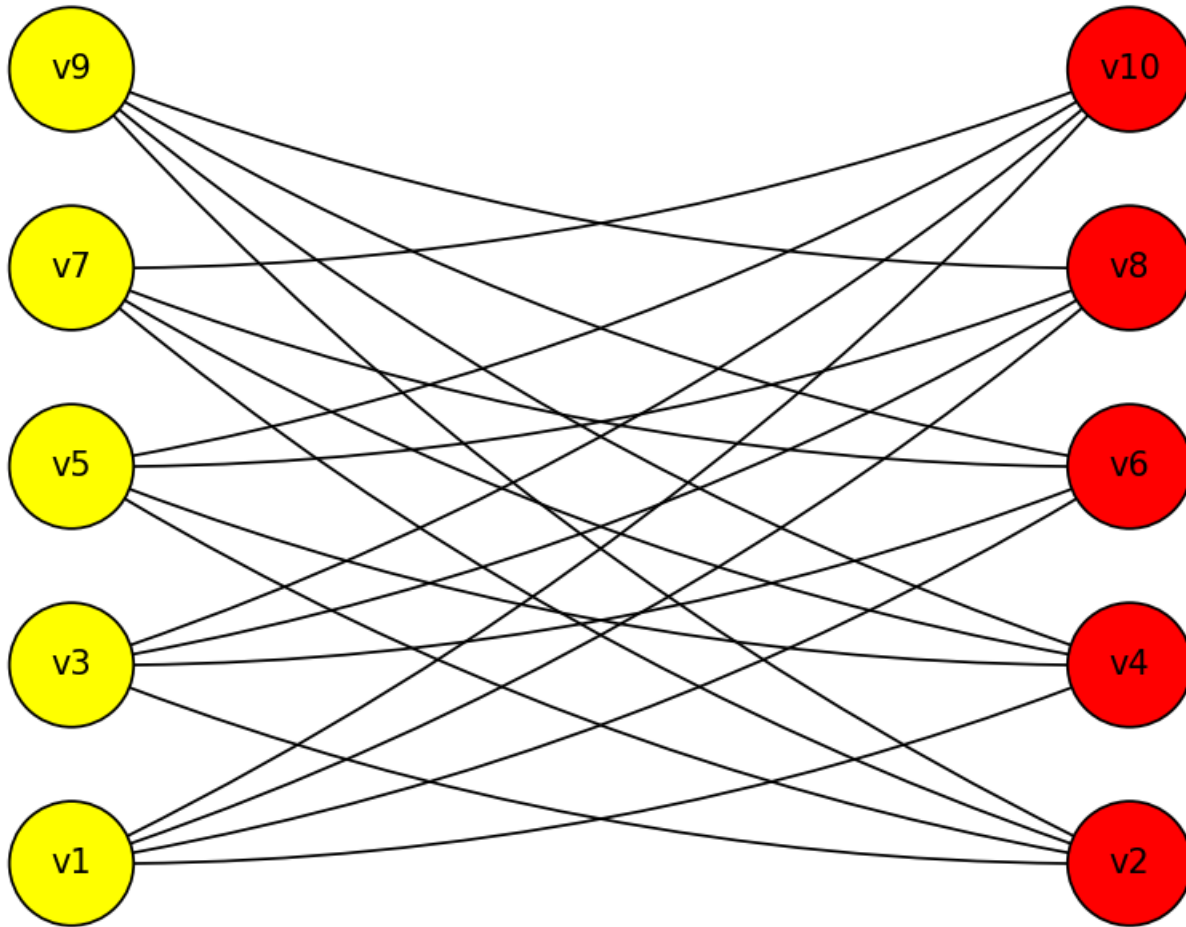
['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8', 'v9', 'v10']

input = 1 2 3 4 5 6 7 8 9 10

output =[0, 0, 1, 1, 2, 2, 3, 3, 4, 4]

Harían falta 5 colores

Estrategia Greedy (Voraz)



Orden:

['v1', 'v3', 'v5', 'v7', 'v9', 'v2', 'v4', 'v6', 'v8', 'v10']

input = 1 3 5 7 9 2 4 6 8 10

output = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Número cromático: 2 colores

Algoritmo DSatur

- Abreviatura de Degree Saturation
- Propuesto por Brélaz (1979)
- Similar a Greedy, la ordenación de los vértices la hace el algoritmo en función de la saturación de los vértices y en caso de igualdad del **grado de cada vértice** (número de aristas, número de restricciones en los que participa)
- **Grado de saturación** de un vértice v , $\text{sat}(v)$: número de colores diferentes asignados a los vértices adyacentes
- El siguiente vértice en ser coloreado es el de máxima saturación, en caso de que haya varios vértices con la misma saturación, se elige el que presente un mayor grado

Algoritmo DSatur

1. Inicialización:

- Todos los vértices del grafo son inicialmente no coloreados.
- El grado de saturación de cada vértice es 0.

2. Selección del primer vértice:

- Se selecciona el vértice con el mayor grado (número de aristas) y se le asigna el color 1.

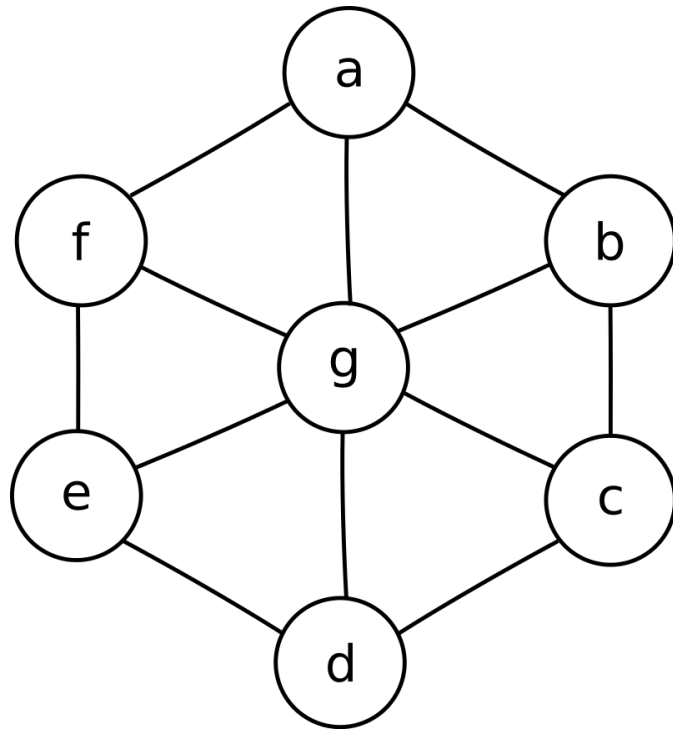
3. Proceso iterativo:

- En cada iteración, se selecciona el vértice con el mayor grado de saturación (número de colores diferentes en sus vértices adyacentes).
- Si hay empate en el grado de saturación, se selecciona el vértice con el mayor grado.
- Se asigna al vértice seleccionado el color más bajo que no haya sido utilizado por sus vértices adyacentes.
- Se actualiza el grado de saturación de los vértices adyacentes del vértice coloreado.

4. Terminación:

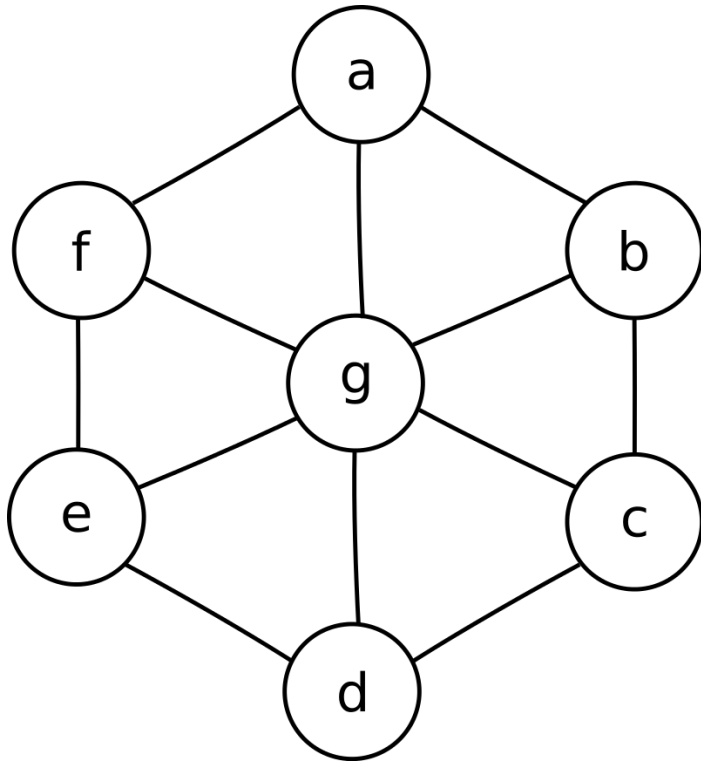
- El proceso se repite hasta que todos los vértices hayan sido coloreados.

Ejemplo, colorear el siguiente grafo con dSatur



En caso de empate en la heurística dSatur,
utilizar orden lexicográfico (orden alfabético)

Solución



1. Vertex g (color 1)
2. Vertex a (color 2)
3. Vertex b (color 3)
4. Vertex c (color 2)
5. Vertex d (color 3)
6. Vertex e (color 2)
7. Vertex f (color 3)

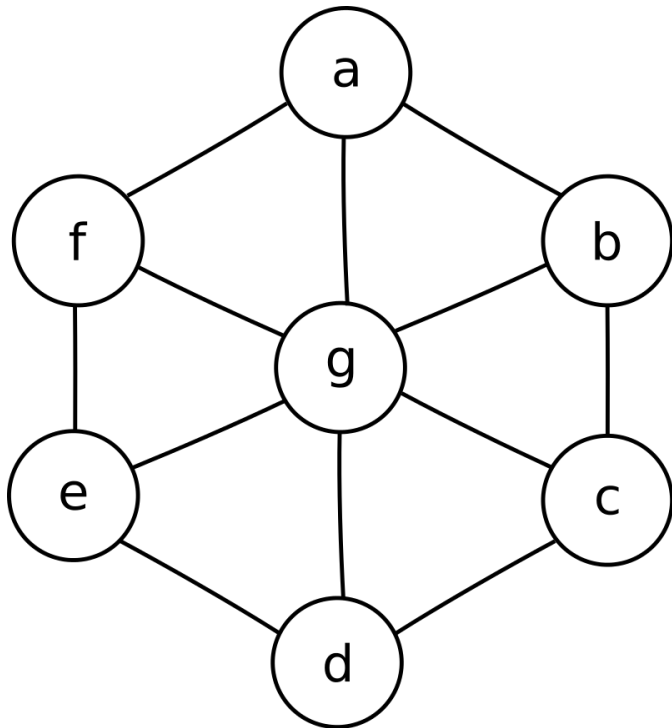
$$\mathcal{S} = \{\{g\}, \{a, c, e\}, \{b, d, f\}\}$$

<https://en.wikipedia.org/wiki/DSatur>

Algoritmo RLF

- Abreviatura de “Recursive Largest First”
- Propuesto por Leighton (1979)
- Colorea lo máximo que pueda el grafo con un color. En lugar de iterar por los vértices asignando colores, itera por colores, asignando a cada color el máximo número de vértices que pueda.
- Cuando ha terminado con un color, elimina del grafo los vértices asignados, y repite el proceso con otro color.

Ejemplo, colorear el siguiente grafo con RLF




1. Vertex *g* (color 1)
2. Vertex *a*, *c*, and then *e* (color 2)
3. Vertex *b*, *d*, and then *f* (color 3)

Comparación empírica

- Lewis 2015 (valores para 50 grafos generados aleatoriamente)

n	Voraz	DSatur	RLF
100	21.14 ± 0.95	18.48 ± 0.81	17.44 ± 0.61
500	72.54 ± 1.33	65.18 ± 1.06	61.04 ± 0.78
1000	126.64 ± 1.21	115.44 ± 1.23	108.74 ± 0.90
1500	176.20 ± 1.58	162.46 ± 1.42	153.44 ± 0.86
2000	224.18 ± 1.90	208.18 ± 1.02	196.88 ± 1.10

RLF mostró consistentemente los valores más bajos en todos los tamaños de grafos, en este ejemplo es el algoritmo más eficiente de los tres presentados para colorear estos grafos aleatorios.



Problema colorear un mapa

- En este problema se trata de colorear un mapa de forma que 2 territorios adyacentes no reciban el mismo color
 - Elegir las variables de decisión
 - Expresar las restricciones en términos de estas variables
- Variable de decisión: color que asignamos a cada territorio
- Dominio de la variable: 4 colores diferentes

Problema colorear un mapa

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
  color[Belgium] ≠ color[France];  
  color[Belgium] ≠ color[Germany];  
  color[Belgium] ≠ color[Netherlands];  
  color[Belgium] ≠ color[Luxembourg];  
  color[Denmark] ≠ color[Germany];  
  color[France] ≠ color[Germany];  
  color[France] ≠ color[Luxembourg];  
  color[Germany] ≠ color[Netherlands];  
  color[Germany] ≠ color[Luxembourg];  
}
```



Problema colorear un mapa



minizinc

```
enum Countries = {Belgium, Denmark, France, Germany, Netherlands, Luxembourg};
enum Colors = {black, yellow, red, blue};

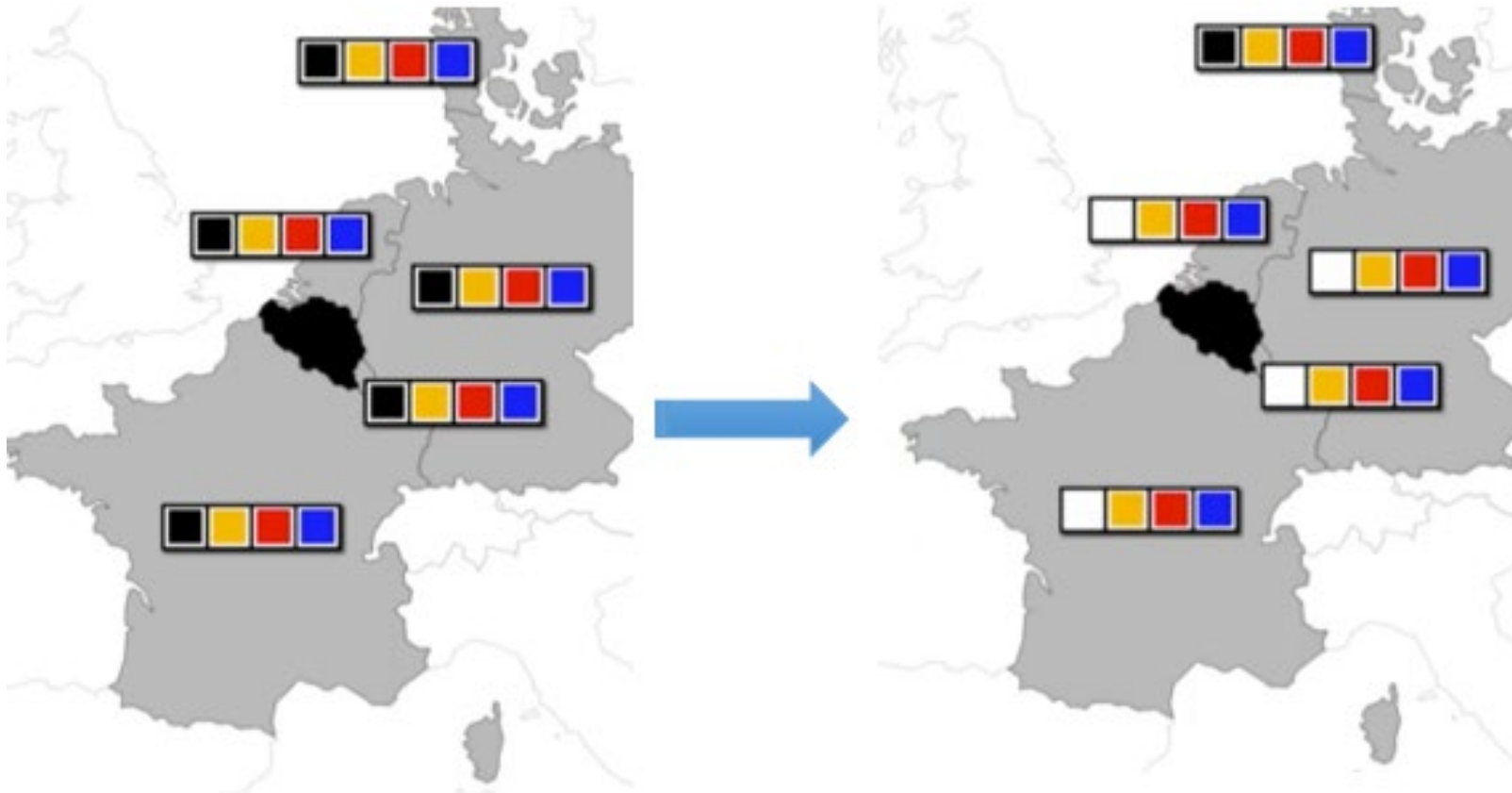
array[Countries] of var Colors: color;

constraint color[Belgium] != color[France];
constraint color[Belgium] != color[Germany];
constraint color[Belgium] != color[Netherlands];
constraint color[Belgium] != color[Luxembourg];
constraint color[Denmark] != color[Germany];
constraint color[France] != color[Germany];
constraint color[France] != color[Luxembourg];
constraint color[Germany] != color[Netherlands];
constraint color[Germany] != color[Luxembourg];

solve satisfy;
```

Problema colorear un mapa

Supongamos que asignamos el color negro a Bélgica:



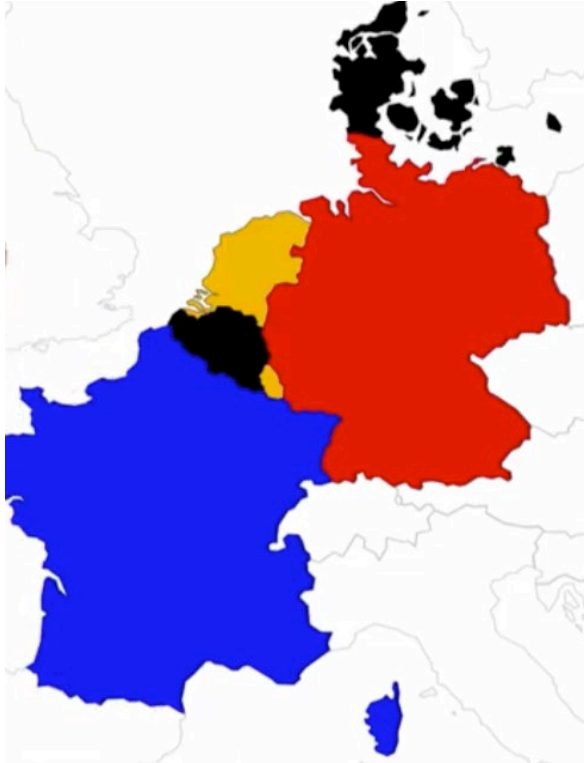
Problema colorear un mapa

Comprobamos las restricciones donde aparece Bélgica, y sustituimos la variable por su valor:

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
    black ≠ color[France];  
    black ≠ color[Germany];  
    black ≠ color[Netherlands];  
    black ≠ color[Luxembourg];  
    color[Denmark] ≠ color[Germany];  
    color[France] ≠ color[Germany];  
    color[France] ≠ color[Luxembourg];  
    color[Germany] ≠ color[Netherlands];  
    color[Germany] ≠ color[Luxembourg];  
}
```

Problema colorear un mapa

Continuando con este proceso, obtenemos como solución:



Branch & Prune

Branch: Descomponemos el problema en subproblemas más pequeños

Prune: Reducimos el espacio de búsqueda lo máximo posible, eliminando los valores no factibles del dominio de las variables

Minizinc

- Estructura básica
 1. **include** *⟨filename⟩*;
 2. Declaración de variables
 1. **Parámetros**: float/range of float, int/range of int, string, bool, ann
 2. **Variables de decisión**: float/range of float, int/range of int, bool
 3. *⟨type inst expr⟩*: *⟨variable⟩* [= *⟨expression⟩*];
 3. Asignación de valores a variables: *⟨variable⟩* = *⟨expression⟩*;
 4. Restricciones: **constraint** *⟨Boolean expression⟩*;
 5. solve satisfy;
solve maximize *⟨arithmetic expression⟩*;
solve minimize *⟨arithmetic expression⟩*;
 6. **output** [*⟨string expression⟩*, ▪ ▪ ▪ , *⟨string expression⟩*];

Minizinc

- Enum, Arrays, Sets
 - **enum** Colors = {black, yellow, red, blue};
 - [var] enum-name: var-name
- **array**[0..4] of var Colors: colors;
- **set of int**: size = 0..4;
- **array**[0..3,size] **of var float**: t;

Minizinc

- Hello World - hello.mzn
 - Construir un modelo que muestre en pantalla “Hello World”
 - Ejecutar desde el IDE y desde consola

Minizinc:

- **output**
- **show**



Minizinc

- Input and Output - inout.mzn
 - Construir un modelo inout.mzn que defina un parámetro entero y lo muestre en pantalla
 - Desde consola: `minizinc inout.mzn -D'n = 23;'`
 - Desde el IDE debería saltar un pop up

Minizinc

- **Variable de decisión**
 - Construir un modelo `xvar.mzn` con una variable de decisión en el rango 1..10 y que muestre todos sus valores
 - Desde consola: `minizinc xvar.mzn -a`
 - Desde el IDE:
 - Tab de configuración
 - User-defined behavior
 - “Stop after this many solutions (0 means “all”)

Solver configuration:

Gecode 6.1.0 [built-in]

Hide configuration editor Show project explorer

Configuration

Gecode 6.1.0 [built-in]

Clone Reset to defaults ☒ default built-in configuration

Solving

Solver: Gecode 6.1.0

Time limit: 0 seconds (disabled)

☐ Default behavior

Optimization problems: print all intermediate solutions
Satisfaction problems: stop after first solution

☒ User-defined behavior

☐ Print intermediate solutions (for optimization problems)

Stop after this many solutions (0 means "all"): 1 (for satisfaction problems)

Minizinc

- Optimización
 - Construir un modelo `xoptima.mzn` con una variable de decision, x , en el rango $0..10$ con la restricción de que x debe ser divisible por 4, la salida debe ser aquella que minimice el valor $(x - 7)^2$
 - Desde consola:
 - `minizinc xoptima.mzn -a`, devuelve todas las soluciones
 - `miniznc xoptima.mzn`: devuelve el óptimo
 - Desde el IDE:
 - Tab de configuración
 - User-defined behavior
 - “Stop after this many solutions (0 means “all”)

Minizinc:

- **mod**
- **solve minimize ...**

Minizinc

- Arrays
 - Construir un modelo array.mzn:
 - Parámetro n , que define la longitud del array
 - El array contiene variables de decisión con valores entre 0..9
 - Restricción, la suma de los valores del array es igual al producto de los valores del array
 - La salida debe ser el array resultante
 - Añadir la restricción de que el orden de los valores del array es creciente
 - $x[1] \leq x[2] \dots \leq x[n]$

Minizinc:

- **sum**
- **product**
- **forall**
- **solve satisfy**

Minizinc

- Secuencia
 - Construir un modelo `sequence.mzn`:
 - Parámetro n , que define la longitud del array
 - El array contiene variables de decisión con valores entre 0..3
 - Restricción, el primer valor es 0, el ultimo vale 3, y la suma de 2 números adyacentes en el array es como mucho 3
 - Restricción, el valor en posiciones divisibles por 3 tiene que ser mayor o igual a 2.
 - Maximizar la suma de los valores del array
 - La salida debe ser *suma = array de valores*
 - Por ejemplo, **$6 = [0,1,2,0,3]$**
 - Testear desde consola, con valores de n entre 3 y 9

Minizinc:

- **`forall (... where ...)`**

Ejercicio propuesto

- Reclutar un ejército
 - Presupuesto máximo 10000\$
 - Soldados de 4 (F,L,Z,J) pueblos distintos
 - Fuerza:6. Coste: 13\$. Máximo 1000 soldados
 - Fuerza:10. Coste 21\$. Máximo 400 soldados
 - Fuerza: 8. Coste 17\$. Máximo 500 soldados
 - Fuerza:40. Coste 100\$. Máximo 150 soldados
 - Maximizar la fuerza del ejercito

Minizinc: **solve maximize**

Ejercicio propuesto

- Contar el número de soldados, se sabe que el número está entre 100 y 500.
 - Se ordenan en columnas de 5 soldados, y sobran 2
 - Se ordenan en columnas de 7 soldados y sobran 2
 - Se ordenan en columnas de 12 soldados, y sobra 1
 - ¿Cuántos soldados hay?
- Variable de decisión:
 - Army
- 3 restricciones

Minizinc:

- **solve satisfy**
- **mod**

Ejercicio propuesto

- Plantear un modelo de la mochila.
 - Parámetros:
 - Crear un set of int de ITEMS
 - 2 Arrays: value y weight
 - Variable de decisión:
 - Array: taken (indica el número de veces que un ítem se mete en la mochila)

Ejemplo:

capacity=10;
value=[45,48,35];
weight=[5,8,3];

taken = [1, 0, 1]
Total Value = 80

Ejercicio propuesto

- Plantear un modelo de la mochila, donde los objetos pueden incluirse en la mochila 2 veces.
 - Parámetros:
 - Crear un set of int de ITEMS
 - 2 Arrays: value y weight
 - Variable de decisión:
 - Array: taken (indica el número de veces que un ítem se mete en la mochila)

Ejemplo:

capacity=10;	}	taken = [2, 0, 0] Total Value = 90
value=[45,48,35];		
weight=[5,8,3];		