

# Tema 5: Problemas de satisfacción de restricciones

José Luis Ruiz Reina  
José Antonio Alonso  
Franciso J. Martín Mateos  
María José Hidalgo

Departamento de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla

Inteligencia Artificial I, 2011

# Índice

PSRs: representación

Backtracking

Consistencia de arcos

Reparación heurística

# Problema de Satisfacción de Restricciones

- Un *problema de satisfacción de restricciones* (PSR) viene definido por los siguientes elementos:
  - Un conjunto finito de *variables*  $X_1, \dots, X_n$
  - Un conjunto finito de *dominios*  $D_i$  asociados a cada variable  $X_i$ , especificando los posibles valores que puede tomar
  - Un conjunto finito de *restricciones*  $C_1, \dots, C_m$ , que definen una serie de propiedades que deben verificar los valores asignados a las variables
- Una solución al problema es una asignación de valores a las variables  $\{X_1 = v_1, \dots, X_n = v_n\}$  tal que  $v_i \in D_i$  y se verifican las restricciones
- Esta formulación permite una representación simple del problema, y el uso de heurísticas de propósito general, *independientes* del problema

# Tipos de Problemas de Satisfacción de Restricciones

- Clasificación según el tipo de restricciones:
  - Restricciones de obligación (hard constraints).
  - Restricciones de preferencia (soft constraints).
- Clasificación según los dominios:
  - Dominios discretos (finitos o infinitos).
  - Dominios continuos.
- Clasificación según el número de variables implicadas en las restricciones:
  - Restricciones binarias.
  - Restricciones múltiples.

## Ejemplo: N reinas

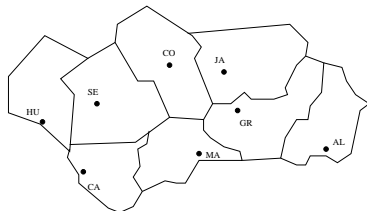
- Situar  $N$  reinas en un tablero de ajedrez de tamaño  $N \times N$  de forma que no se den jaque mutuamente.
- Variables:  $V_1, \dots, V_N$
- Dominios:  $D_i = \{1, \dots, N\}$
- Restricciones:
  - Jaque horizontal:  $V_i \neq V_j$
  - Jaque diagonal:  $|V_i - V_j| \neq |i - j|$
- Ejemplo ( $V_1 = 1, V_2 = 3$ ):

R				
	R			

- Problema con dominios finitos y restricciones binarias (de obligación).

## Ejemplo: coloreado de mapas

- Usando tres colores (rojo, azul, verde) colorear el mapa de Andalucía:



- Variables: Huelva, Cádiz, Sevilla, Córdoba, Málaga, Jaen, Granada, Almería
- Dominios: {**rojo**, **azul**, **verde**}
- Restricciones:  $P \neq Q$ , para cada par de provincias vecinas  $P$  y  $Q$
- Problema con dominios finitos y restricciones binarias (de obligación).

## Ejemplo: satisfacibilidad proposicional

- Problema SAT (para cláusulas):
  - Dado un conjunto de variables proposicionales  $L = \{p_1, \dots, p_n\}$  y un conjunto de cláusulas proposicionales  $\{C_1, \dots, C_m\}$  formadas con los símbolos de  $L$ , determinar si existe una asignación de valores de verdad a los símbolos de  $L$  de manera que se satisfagan todas las cláusulas
- Variables:  $\{p_1, \dots, p_n\}$
- Dominios:  $\{T, F\}$  (*booleanos*)
- Restricciones:  $\{C_1, \dots, C_m\}$
- Problema con dominios finitos y restricciones múltiples (de obligación).
- Problema NP-completo
  - Por tanto, en el peor caso, no es posible resolver PSRs con dominios finitos en tiempo polinomial (si  $P \neq NP$ ).

## Ejemplo: criptoaritmética

- Asignar valores del 1 al 8 a cada letra de manera que:

$$\begin{array}{r}
 \text{(C1 C2 C3)} \\
 \text{I D E A} \\
 \text{I D E A} \quad + \\
 \hline
 \text{M E N T E}
 \end{array}$$

- Variables:  $I, D, E, A, M, N, T, C_1, C_2, C_3$
- Dominios:  $\{1, \dots, 8\}$  para  $I, D, E, A, M, N, T$  y  $\{0, 1\}$  para  $C_1, C_2, C_3$
- Restricciones:
  - Primera suma:  $2 * A = (10 * C_3) + E$
  - Segunda suma:  $(2 * E) + C_3 = (10 * C_2) + T$
  - Tercera suma:  $(2 * D) + C_2 = (10 * C_1) + N$
  - Cuarta suma:  $(2 * I) + C_1 = (10 * M) + E$
- Problema con dominios finitos y restricciones múltiples (de obligación).

## Ejemplo: asignación de tareas

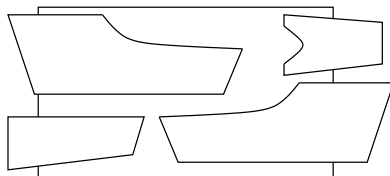
- Asignar tareas a empleados de acuerdo con su capacidad para desarrollarlas
- Tabla de capacidades ( $\mathbf{C}_i$ ):

	$T_1$	$T_2$	$T_3$
$E_1$	1	3	2
$E_2$	3	2	1
$E_3$	2	3	1

- Variables:  $E_i$
- Dominios:  $D_i = \{T_1, \dots, T_n\}$
- Restricciones:
  - Obtener la mejor  $\sum_{i=1}^n \mathbf{C}_i$
- Problema con dominios finitos y restricciones múltiples (de preferencia).

## Ejemplo: planificación de corte

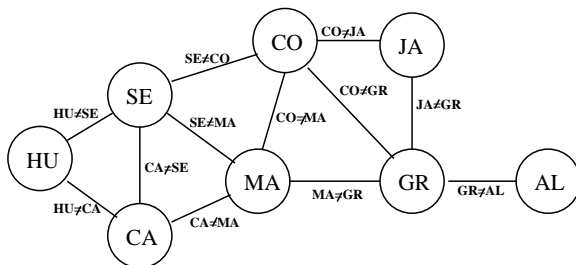
- Encontrar la manera de situar patrones de corte en una pieza de cartón.
- Variables:  $P_1, P_2, P_3, P_4$  (piezas)
- Dominios: Coordenadas en el plano.
- Restricciones:
  - Las piezas no deben superponerse.



- Problema con dominios continuos y restricciones binarias (de obligación).

# Problemas de Satisfacción de Restricciones

- En este tema, trataremos PSRs:
  - Con dominios finitos
  - Con restricciones binarias de obligación
- Todo problema con restricciones múltiples se puede reducir a uno equivalente con restricciones binarias
- Representación en forma de grafo de un PSR:



# Implementación de la representación de un PSR

- Variables: estructura con nombre + dominio
  - Funciones de acceso: **PSR-VAR-NOMBRE** y **PSR-VAR-DOMINIO**
- La variable global **\*VARIABLES\*** contiene a todas las variables del PSR
- Restricciones: estructura con los nombres de las variables involucradas + función
  - Funciones de acceso: **PSR-RESTR-VARIABLES** y **PSR-RESTR-FUNCION**
- La variable global **\*RESTRICCIONES\*** contiene todas las restricciones del PSR

# PSR como problema de espacio de estados

- Estados:
  - Asignaciones parciales (y *consistentes* con las restricciones) de valores a variables:  
 $\{X_{i_1} = v_{i_1}, \dots, X_{i_k} = v_{i_k}\}$
  - Estado inicial:  $\{\}$
  - Estados finales: asignaciones completas
- Operadores:
  - Asignar un valor (de su dominio) a una variable no asignada en el estado
  - Aplicabilidad: la asignación resultante tiene que ser consistente
- Soluciones mediante búsqueda en profundidad
  - Puesto que las soluciones están a una profundidad fija (igual al número de variables), profundidad es preferible a anchura

# Simplificaciones en la búsqueda

- El orden en el que se aplican los operadores es irrelevante para la solución final (*conmutatividad*)
  - Por tanto, los operadores pueden reducirse a considerar las posibles asignaciones *a una única variable* no asignada
- El camino hacia la solución no es importante
  - No necesitamos la estructura de nodo
  - No necesitamos una representación explícita para los operadores
- No es posible repetir un estado durante la búsqueda:
  - No es necesario realizar comprobaciones para evitar repeticiones
  - Es decir, no es necesario tener una lista **CERRADOS** con las asignaciones parciales ya analizadas

# Búsqueda en profundidad para PSR: implementación

- Estados: representados como listas de asociación
- Usaremos una función que elige la nueva variable a asignar (de entre las no asignadas) y otra función que va a ordenar los valores del dominio de la variable seleccionada:

- `SELECCIONA-VARIABLE(ESTADO)`
- `ORDENA-VALORES(DOMINIO)`

- Función que calcula los sucesores:

`FUNCION PSR-SUCESORES(ESTADO)`

1. Hacer `VARIABLE` igual a `SELECCIONA-VARIABLE(ESTADO)`

Hacer `DOMINIO-ORD` igual

`ORDENA-VALORES(PSR-VAR-DOMINIO(VARIABLE))`

Hacer `SUCESORES` igual a vacío

2. Para cada `VALOR` en `DOMINIO-ORD`,

Hacer `NUEVO-ESTADO` igual a la asignación obtenida  
ampliando `ESTADO` con `VARIABLE=VALOR`

Si `NUEVO-ESTADO` es consistente con `*RESTRICCIONES*`,  
añadir `NUEVO-ESTADO` a `SUCESORES`

3. Devolver `SUCESORES`

# Búsqueda en profundidad para PSR: implementación

**FUNCION PSR-BUSQUEDA-EN-PROFUNDIDAD()**

1. Hacer ABIERTOS igual a la lista formada por la asignación vacía
2. Mientras que ABIERTOS no esté vacía,
  - 2.1 Hacer ACTUAL el primer estado de ABIERTOS
  - 2.2 Hacer ABIERTOS el resto de ABIERTOS
  - 2.3 Si ACTUAL es una asignación completa
    - 2.4.1 devolver ACTUAL y terminar.
    - 2.4.2 en caso contrario,
      - 2.4.2.1 Hacer NUEVOS-SUCESORES igual a PSR-SUCESORES(ACTUAL)
      - 2.4.2.2 Hacer ABIERTOS igual al resultado de incluir NUEVOS-SUCESORES al principio de ABIERTOS
3. Devolver FALLO.

## Algoritmo de *backtracking*

- El algoritmo de búsqueda en profundidad anterior se suele llamar algoritmo de *backtracking*, aunque usualmente, se presenta de manera recursiva:

**FUNCION PSR-BACKTRACKING()**

1. Devolver PSR-BACKTRACKING-REC({})

**FUNCION PSR-BACKTRACKING-REC(ESTADO)**

1. Si ESTADO es una asignación completa, devolver ESTADO y terminar

2. Hacer VARIABLE igual a SELECCIONA-VARIABLE(ESTADO)  
Hacer DOMINIO-ORD  
igual ORDENA-VALORES(PSR-VAR-DOMINIO(VARIABLE))

3. Para cada VALOR en DOMINIO-ORD,  
3.1 Hacer NUEVO-ESTADO igual a la asignación  
obtenida ampliando ESTADO con VARIABLE=VALOR  
3.2 Si NUEVO-ESTADO  
es consistente con \*RESTRICCIONES\*:  
3.2.1 Hacer RESULTADO igual a  
PSR-BACKTRACKING-REC(NUEVO-ESTADO)  
3.2.2 Si RESULTADO no es FALLO,  
devolver RESULTADO y terminar

3. Devolver FALLO

## Heurísticas en el algoritmo de *backtracking*

- En principio, *Backtracking* realiza una búsqueda ciega
  - Búsqueda no informada, ineficiente en la práctica
- Sin embargo, es posible dotar al algoritmo de cierta heurística que mejora considerablemente su rendimiento
  - Estas heurísticas son de propósito general
  - Independientes del problema
  - Sacan partido de la estructura especial de los PSR
- Posibles mejoras heurísticas:
  - Selección de nueva variable a asignar
  - Orden de asignación de valores a la variable elegida
  - Propagación de información a través de las restricciones
  - Vuelta atrás “inteligente” en caso de fallo

## Selección de la siguiente variable a asignar

- Incorporar heurística en la definición de **SELECCIONA-VARIABLE( ESTADO )**
- Heurística MRV:
  - seleccionar la variable con el menor número de valores en su dominio consistente con la asignación parcial actual
  - Ejemplo: con la asignación (parcial)

**{ Huelva = Rojo, Sevilla = Azul, Malaga = Rojo }**

la siguiente variable a asignar sería **Cadiz** o **Cordoba** (en ambos casos sólo hay un valor en sus dominios consistente con la asignación)

- Otra heurística: variable que aparece en el mayor número de restricciones (*grado*)
  - Usada para desempatar MRV

# Ordenación de valores de la variable seleccionada

- Definición adecuada para la función  
**ORDENA-VALORES (DOMINIO)**
  - Considerar antes los que eliminen menos posibles valores de las variables por asignar

## Comprobación hacia adelante (*forward checking*)

- Para cada estado, mantener información sobre los posibles valores para las variables que quedan por asignar:
  - Cada vez que se asigna un nuevo valor a una variable, quitar del dominio de las variables por asignar, aquellos valores que no sean consistentes con el nuevo valor asignado
- Cuestiones técnicas:
  - Cada nodo del árbol debe contener el estado junto la lista de valores posibles en las variables por asignar
  - Muy fácil de usar en conjunción con la heurística MRV

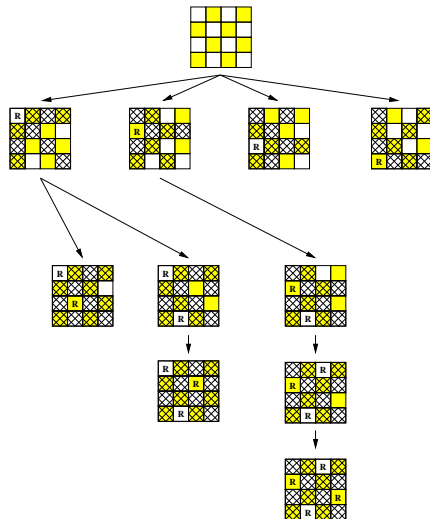
## Ejemplo de (*forward checking*) en el coloreado de mapas

- Ejemplo (seleccionando variables en orden alfabético):

```
{AL=R}           --> eliminar R del dominio de GR
{AL=R,CA=R}     --> eliminar R de los dominios de HU, SE y MA
{AL=R,CA=R,CO=A} --> eliminar A de los dominios de MA, SE y JA
{AL=R,CA=R,CO=A,GR=V} --> eliminar V de los dominios de JA y MA
```

- En este momento, **Malaga** no tiene ningún valor posible.  
Por tanto, no es necesario seguir explorando en el árbol
- Es decir, se ha detectado inconsistencia sin necesidad de asignar valores a **Huelva** y **Jaen**, y para cualquier extensión de la asignación parcial construida hasta el momento

# Ejemplo: *Backtracking+forward checking* en 4-reinas



# Propagación de restricciones

- La comprobación hacia adelante es un caso particular de *propagación de restricciones*:
  - Propagar las implicaciones de las restricciones sobre una variable sobre otras variables
  - existen técnicas más completas para propagar restricciones que la comprobación hacia adelante
  - la propagación debe ser rápida: completitud vs rapidez
- La *consistencia de arcos* proporciona un método con un buen compromiso entre eficiencia y completitud

## Consistencia de arcos

- En un PSR, por *arco* entendemos un *arco dirigido* en el grafo que lo representa
  - Equivalentemente, un arco es una restricción en la que hay una variable distinguida
  - Notación:  $\mathbf{B} > \underline{\mathbf{E}}, \underline{\mathbf{CO}} \neq \mathbf{SE}, |\mathbf{V}_i - \mathbf{V}_j| \neq |i - j|$
- Arco *consistente* respecto a un conjunto de dominios asociados a un conjunto de variables:
  - Para cualquier valor del dominio asociado a la variable distinguida del arco, *existen* valores en los dominios de las restantes variables que satisfacen la restricción del arco
  - Ejemplo: si  $\mathbf{SE}$  y  $\mathbf{CO}$  tienen ambas asignadas como dominio  $\{\mathbf{rojo}, \mathbf{verde}\}$ , el arco  $\underline{\mathbf{CO}} \neq \mathbf{SE}$  es consistente
  - Ejemplo: si el dominio de  $\mathbf{SE}$  es  $\{\mathbf{rojo}\}$  y el de  $\mathbf{CO}$  es  $\{\mathbf{rojo}, \mathbf{verde}\}$ , el arco  $\underline{\mathbf{CO}} \neq \mathbf{SE}$  no es consistente. **Nota:** el arco puede hacerse consistente eliminando  $\mathbf{rojo}$  del dominio de  $\mathbf{CO}$

# Idea intuitiva de AC-3 (consistencia de arcos)

- **Objetivo:**
  - a partir de los dominios iniciales de las variables
  - devolver un conjunto de dominios actualizado tal que todos los arcos del problema sean consistentes
- **Actualización de dominios:**
  - Si un arco es inconsistente, podemos hacerlo consistente eliminando del dominio de la variable distinguida aquellos valores para los que no existen valores en los dominios de las restantes variables que satisfagan la restricción

# Idea intuitiva de AC-3 (consistencia de arcos)

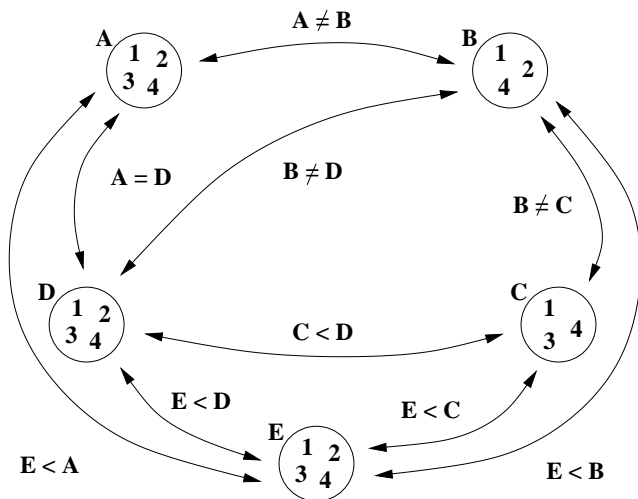
- Revisión de arcos:
  - si el dominio de una variable se actualiza, es necesario revisar la consistencia de los arcos en los que aparece la variable como variable no distinguida
- Criterio de parada:
  - Todos los arcos son consistentes respecto a los dominios de las variables
  - O algún dominio queda vacío (inconsistencia)

## Consistencia de arcos (ejemplo)

- Planificación de las acciones de un robot (Poole, pag. 149): Un robot necesita planificar cinco actividades (A, B, C, D y E), donde cada actividad ha de comenzar en un momento en el tiempo (1, 2, 3, o 4) y dura exactamente una unidad de tiempo. Restricciones:
  - La actividad B no puede realizarse en el momento número 3.
  - La actividad C no puede realizarse en el momento número 2.
  - Las actividades A y B no pueden realizarse simultáneamente.
  - Las actividades B y C no pueden realizarse simultáneamente.
  - La actividad C ha de realizarse antes de la D.
  - Las actividades B y D no pueden realizarse simultáneamente.
  - Las actividades A y D han de realizarse simultáneamente.
  - La actividad E ha de ser la primera.

## Consistencia de arcos (ejemplo)

- Representación como grafo:





## Implementación del algoritmo AC-3

- Arcos: estructura con los campos variable y restricción
  - Funciones de acceso: **ARCO-VARIABLE** y **ARCO-RESTRICCIÓN**
- Generación de todos los arcos de un PSR:

```
FUNCION ARCOS()
```

```
1. Hacer ARC igual a vacío
```

```
2. Para cada RESTRICCIÓN es *RESTRICCIONES*
```

```
2.1 Para cada VARIABLE en PSR-RESTR-VARIABLES(RESTRICCIÓN)
    Añadir a ARC el arco formado por VARIABLE+RESTRICCIÓN
```

```
3. Devolver ARC
```

- Cálculo de dominio actualizado:

```
FUNCION ACTUALIZA-DOMINIO(VARIABLE, RESTRICCIÓN, VARIABLES)
```

```
1. Hacer DOMINIO-ACTUAL igual al dominio de VARIABLE en VARIABLES
   Hacer NUEVO-DOMINIO igual a vacío
```

```
2. Para cada VALOR en DOMINIO-ACTUAL hacer
```

```
2.1 Si para ese VALOR de VARIABLE existe una asignación a las
    restantes variables de RESTRICCIÓN que satisfaga RESTRICCIÓN,
    incluir VALOR en NUEVO-DOMINIO
```

```
3. Devolver NUEVO-DOMINIO
```

# Implementación del algoritmo AC-3

**FUNCION AC-3(VARIABLES)**

1. Hacer RED igual a ARCOS()
2. Mientras RED no sea vacío
  - 2.1 Hacer ACTUAL el primero de RED y RED el resto de RED
  - 2.2 Hacer VARIABLE igual a ARCO-VARIABLE(ACTUAL) y RESTRICCION igual ARCO-RESTRICCION(ACTUAL)
  - 2.3 Hacer DOMINIO-ACTUAL igual al dominio de VARIABLE en VARIABLES
  - 2.4 Hacer DOMINIO-NUEVO igual a ACTUALIZA-DOMINIO(VARIABLE,RESTRICCION,VARIABLES)
  - 2.5 Si DOMINIO-NUEVO y DOMINIO-ACTUAL son distintos, entonces
    - 2.5.1 Actualizar (destructivamente) el dominio de ACTUAL en VARIABLES con NUEVO-DOMINIO
    - 2.5.2 Incluir en RED todos los arcos de ARCOS() en los que aparezca VARIABLE y ésta NO sea la variable distinguida del arco
  - 2.6 Devolver VARIABLES

- Entrada: una lista de variables+dominios
- Salida: la lista de entrada, con los dominios actualizados (destructivamente) de manera que todos los arcos del PSR son consistentes

## Propiedades de AC-3

- Complejidad en tiempo  $O(n^2 d^3)$  (en el caso de restricciones binarias), donde:
  - $n$ : número de variables
  - $d$ : máximo número de valores de un dominio
- La complejidad polinomial nos indica que sólo con AC-3 no es posible resolver un PSR
  - Pero puede combinarse con algún tipo de búsqueda
- Dos aproximaciones:
  - Incorporar en el algoritmo de *backtracking*, como un paso de *propagación de restricciones* después de cada asignación
  - Intercalar AC-3 con búsqueda

# Búsqueda de soluciones mediante consistencia de arcos

- Posibles resultados tras aplicar AC-3:
  - Existe un dominio vacío: no hay solución.
  - Todos los dominios son unitarios: hay una solución.
  - No hay dominios vacíos y al menos uno no es unitario: ¿soluciones?
- Idea: romper un dominio no vacío en subdominios y aplicar AC-3 a los estados resultantes
  - Esto se puede hacer de manera sistemática (búsqueda)
  - Hasta que se obtiene un estado con todos los dominios unitarios (solución)
  - O bien hasta que se detecta inconsistencia en todas las alternativas (no hay solución)

# Búsqueda de soluciones mediante consistencia de arcos

`FUNCION BUSQUEDA-AC3()`

1. Hacer `ABIERTOS` igual a una lista cuyo único elemento es una copia de `*VARIABLES*`
2. Mientras que `ABIERTOS` no esté vacía,
  - 2.1 Hacer `ACTUAL` igual a `AC3(PRIMERO(ABIERTOS))`
  - 2.2 Hacer `ABIERTOS` el resto de `ABIERTOS`
  - 2.3 Si no existen dominios vacíos en `ACTUAL`,
    - 2.3.1 Si `ACTUAL` tiene todos los dominios unitarios, devolver `ACTUAL` y terminar
    - 2.3.2 Si no, calcular los sucesores de `ACTUAL` y añadirlos a la lista de `ABIERTOS`
3. Devolver `FALLO`.

- Detalles a concretar de la implementación:
  - Sucesores: escoger un dominio no unitario y dividirlo de alguna manera (se puede incorporar heurística para la división y para el orden de sucesores)
  - Gestión de la cola de abiertos: anchura o profundidad

# Algoritmos de mejora iterativa para PSRs

- Planteamiento como un problema de búsqueda local:
  - Estados: asignaciones completas (consistentes o inconsistentes)
  - Estado inicial escogido aleatoriamente
  - Estados finales: soluciones al PSR
  - Generación de sucesor: elegir una variable y cambiar el valor que tiene asignado (usando cierta heurística y aleatoriedad)

# Generación de sucesor mediante la heurística de *mínimos conflictos*

- Variable elegida para cambiar su valor:
  - Aquella (distinta de la última modificada) que participa en más restricciones NO satisfechas en el estado (los empates se deshacen aleatoriamente)
- Nuevo valor elegido:
  - El valor (distinto del que tenía) que produce el menor número de restricciones incumplidas
  - Los empates se deshacen aleatoriamente

# Implementación de reparación heurística con mínimos conflictos

- Nodo de búsqueda: estructura con la asignación actual y la última variable modificada

**FUNCION MIN-CONFLICTOS(MAX-ITERACIONES)**

1. Hacer ACTUAL igual a un nodo con una asignación generada aleatoriamente.
2. Para I desde 1 hasta MAX-ITERACIONES hacer
  - 2.1 Si la asignación de ACTUAL es una solución, devolverla y terminar.
  - 2.2 Hacer VARIABLE igual a una variable (distinta de la última modificada) escogida aleatoriamente de entre aquellas que participan en el mayor número de restricciones incumplidas.
  - 2.3 Hacer SUCESORES igual a la lista de nodos obtenidos cambiando en la asignación ACTUAL el valor de VARIABLE por cada uno de los posibles valores del dominio de VARIABLE en \*VARIABLES\* (excepto el correspondiente al valor que tenía en ACTUAL)
  - 2.4 Hacer ACTUAL el nodo de SUCESORES escogido aleatoriamente de entre aquellos cuya asignación incumple el menor número de restricciones
3. Devolver FALLO

# Reparación heurística: ejemplo con N-reinas

V1		V2		V3		V4	
valor	conflictos	valor	conflictos	valor	conflictos	valor	conflictos
1	3	1	3	1	3	1	3
1	2	1	2	4	0	1	2
1	2	2	1	4	0	1	1
3	1	2	1	4	0	1	0
3	0	1	1	4	0	1	1
3	0	1	0	4	0	2	0

# Propiedades del algoritmo de reparación heurística

- No es completo
- Complejidad en tiempo: lineal en el número de iteraciones
- Aleatoriedad: en la asignación inicial y al deshacer empates
- Reparación heurística tiene muy buenos resultados en algunos tipos de PSRs:
  - PSRs cuyas soluciones están distribuidas uniformemente a lo largo del espacio de estados
  - PSRs en los que las restricciones pueden cambiar dinámicamente

## Bibliografía

- Russell, S. y Norvig, P. *Inteligencia Artificial (un enfoque moderno)* (Pearson Educación, 2004). Segunda edición
  - Cap. 5: “Problemas de Satisfacción de Restricciones”.
- Russell, S. y Norvig, P. *Artificial Intelligence (A Modern Approach)* (Prentice–Hall, 2010). Third Edition
  - Cap. 6: “Constraint Satisfaction Problems”.
- Nilsson, N.J. *Inteligencia artificial (una nueva síntesis)* (McGraw–Hill, 2000)
  - Cap. 11 “Métodos alternativos de búsqueda y otras aplicaciones”.
- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
  - Cap. 4.7: “Constraint Satisfaction Problems”.