Razonamiento con Restricciones

Tutorial IBERAMIA 2002

Javier Larrosa
Dep. LSI, UPC, Barcelona

Pedro Meseguer
IIIA, CSIC, Bellaterra

Esquema Global

1. Introducción
- Definiciones
- Ejemplos

2. Métodos de Resolución
- Búsqueda
- Inferencia
- Métodos híbridos

3. Modelización
- Primal / dual
- Restricciones globales
- Programación con restricciones

4. Restricciones Blandas
- Modelos
- Algoritmos

Esquema

Introducción

- Definiciones
- Ejemplos

Metodos de resolución

- Búsqueda:
  - Generar y testear
  - Backtracking
  - Backjumping
- Inferencia
  - Consistencia de arcos
  - Consistencia de caminos
  - K-consistencia

  } *incompleta*
  - Problemas libres de backtracking
  - Consistencia adaptativa

  } *completa*
- Algoritmos híbridos
  - Forward Checking
  - Really Full Lookahead
  - Maintaining Arc Consistency
  - Heurísticas

Satisfacción de Restricciones

Definición: CSP o *constraint satisfaction problem* , definido por una terna (***X***, ***D***, ***C***)

$\boldsymbol{X} = \{ X_1, X_2, \ldots, X_n \}$ variables

$\boldsymbol{D} = \{ D_1, D_2, \ldots, D_n \}$ dominios

$\boldsymbol{C} = \{ C_1, C_2, \ldots, C_r \}$ restricciones

Dada una restricción C_i ,

$\text{var}(C_i) = \{X_{i1},..., X_{ik}\}$ relaciona k variables (restricción k-aria)

$\text{rel}(C_i) \subseteq D_{i1} \times D_{i2} \times \ldots \times D_{ik}$ tuplas de valores permitidos

Solución: asignación de valores a variables satisfaciendo todas las restricciones

Complejidad:

- NP-completo
- algoritmos de coste exponencial (caso peor)

Interés de los CSP

Relevancia:

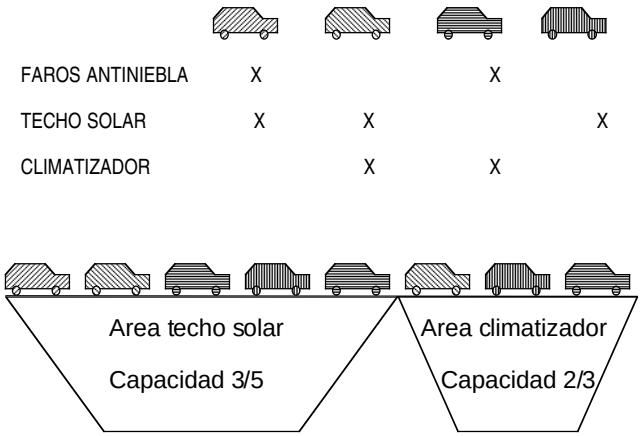
- Problemas reales como CSPs:
  - *Car sequencing problem*
  - Asignación de recursos (*scheduling*)
  - Diseño de bloques (*BIBDs*)
- Para la IA:
  - Restricciones: formal general de representación del conocimiento
  - Satisfacción de restricciones: razonamiento automático
  - Ejemplos:
    - » SAT
    - » razonamiento temporal
    - » razonamiento basado en modelos

Especialización:

- tipo de dominios:
  - discretos / continuos
  - finitos / infinitos
- tipo de restricciones:
  - binarias / n-arias

Car sequencing problem

Cadena de montaje de coches



Formulación:

- variables: *n* coches a producir
- dominios: modelos de coche
- restricciones: capacidad de las áreas

Características:

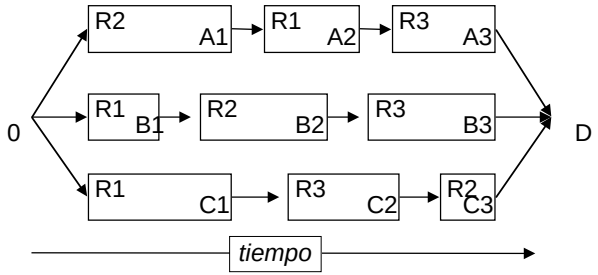
- CSP no binario, discreto y finito

Asignación de recursos

Job-shop scheduling

- *n* jobs, cada uno con *m* operaciones,
- *m* recursos, cada operación necesita un recurso de forma exclusiva durante un tiempo
- precedencia entre las operaciones de cada job,

¿Se pueden realizar los *n* jobs en tiempo D?



Formulación:

- variables: operaciones
- dominios: tiempos de inicio de cada operacion
- restricciones:
  - precedencia entre las operaciones de un job
  - exclusividad de cada recurso en el tiempo

Características:

- CSP binario, discreto y finito (acotado por D)

Restricciones binarias

CSP binario:

$\{ X_1, X_2, \ldots, X_n \}$ variables

$\{ D_1, D_2, \ldots, D_n \}$ dominios discretos y finitos

$\{ R_{ij} \}$ restricciones binarias

$var(R_{ij}) = \{X_i , X_j\}$

$rel(R_{ij})$ = {valores permitidos para X_i y X_j}

R_{ij} simétrica

Solución: asignación de valores a variables satisfaciendo todas las restricciones.

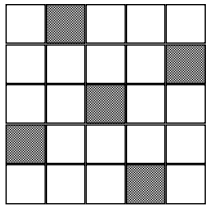
Generalidad: todo problema n-ario se puede formular como un problema binario [Rossi et al, 90]

Ejemplos:

- Coloreado de grafos
- Satisfacibilidad booleana (SAT)
- N-reinas, crucigramas, criptoaritmética

N-reinas

Definición: posicionar n reinas en un tablero de ajedrez $n \times n$, de forma que no se ataquen.



$n = 5$

Formulación: 1 reina por fila

- variables: reinas, X_i reina en la fila i-ésima
- dominios: columnas posibles $\{1, 2, \ldots, n\}$
- restricciones: no colocar dos reinas en
  - la misma columna
  - la misma diagonal

  $rel(R_{ij}) = \{(a,b) \mid a \neq b \wedge |i-j| \neq |a-b|\}$

Características:

- CSP binario, discreto y finito
- existe un método de solución constructivo

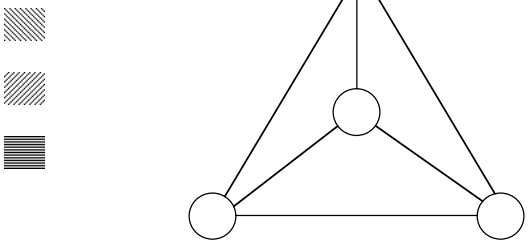
Coloreado de grafos

Definición: Dado un grafo,

- n nodos
- m colores,

asignar un color a cada nodo de forma que no haya dos nodos adyacentes con el mismo color.

Colores



Formulación:

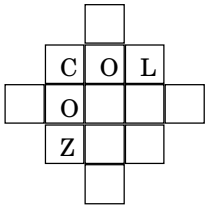
- variables: nodos
- dominios: colores posibles
- restricciones: $\neq$ nodos adyacentes

Características:

- CSP binario, discreto y finito

Generación de crucigramas

Definición: Dada una rejilla y un diccionario, construir un crucigrama legal.



4 palabras de 1 letra
4 palabras de 3 letras
2 palabras de 5 letras

Formulación:

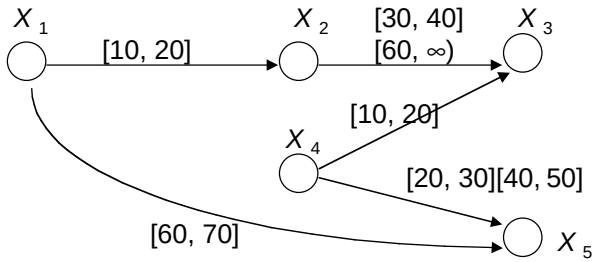
- variables: grupo de casillas para una palabra (*slots*)
- dominios: palabras del diccionario con la longitud adecuada
- restricciones: misma letra en la intersección de dos palabras

Características:

- CSP binario, discreto y finito (dominios grandes)

Restricciones temporales

Definición: dado un conjunto de sucesos que ocurren en intervalos temporales con ciertas relaciones, encontrar una asignación temporal consistente.



Formulación:

- variables: sucesos
- dominios: intervalo temporal para cada suceso
- restricciones: distancia temporal permitida entre sucesos; relaciones temporales *antes*, *después*, *solapado*, etc.

Características:

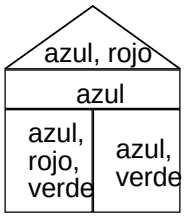
- CSP binario, contínuo e infinito

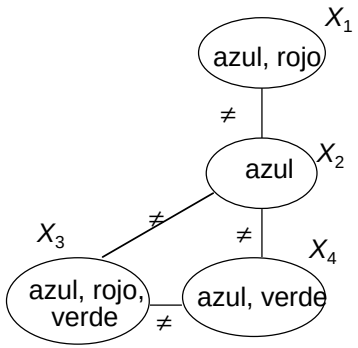
Grafo de restricciones

Grafo de restricciones:

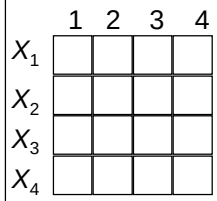
- $\{X_i\}$ nodos
- $\{D_i\}$ dominios en los nodos
- $\{R_{ij}\}$ arcos etiquetados

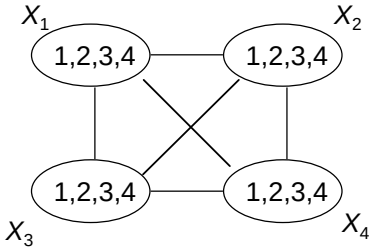
Coloreado de mapas





4-reinas





Esquema

Introducción

- Definiciones
- Ejemplos

Metodos de resolución

- Búsqueda: ←
  - Generar y testear
  - Backtracking
  - Backjumping
- Inferencia
  - Consistencia de arcos
  - Consistencia de caminos
  - K-consistencia

  } *incompleta*
  - Problemas libres de backtracking
  - Consistencia adaptativa

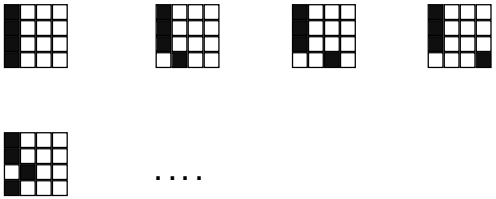
  } *completa*
- Algoritmos híbridos
  - Forward Checking
  - Really Full Lookahead
  - Maintaining Arc Consistency
  - Heurísticas

Generar y testear

Algoritmo: generación y test de todas las asigaciones totales posibles.

1. Generar una asignación total.
2. Comprobar si es solución. Si es, stop, sino ir a 1

Ejemplo: 4-reinas



. . . .

Eficiencia:

- es muy poco eficiente
- genera muchas asignaciones que violan la misma restricción

Búsqueda con backtracking

Estrategia:

- Construir una *solución parcial* : asignación parcial que satisface las restricciones de las variables involucradas.
- Extender la solución parcial, incluyendo una variable cada vez hasta una solución total
- Si no se puede extender $\rightarrow$ backtracking
  - cronológico: se elimina la última decisión
  - no cronológico: se elimina una decisión anterior

Tipos de variables:

- *pasadas* $\in$ solución parcial, tienen valor asignado
- *futuras* $\notin$ solución parcial, no tienen valor asignado
- *actual*, variable en curso

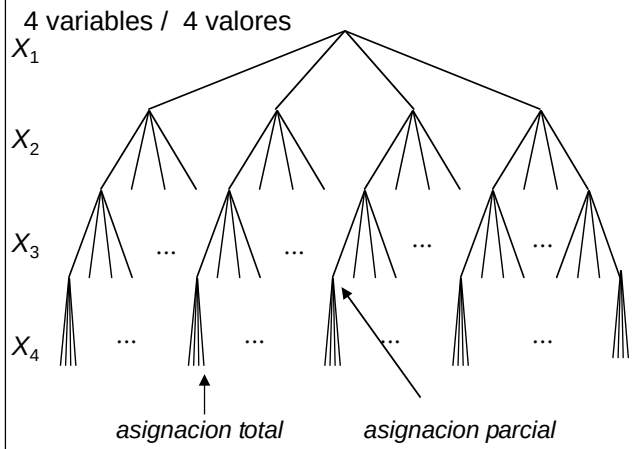
Tipos de restricciones:

- C_P restricciones entre pasadas
- C_{PF} restricciones entre pasadas y futuras
- C_F restricciones entre futuras
- C_X restricciones que involucran var X

Árbol de búsqueda

Espacio de estados: representable por un árbol

- raíz: asignación vacía
- a cada nivel asociamos una variable
- sucesores de un nodo: todos los valores de la variable asociada a ese nivel



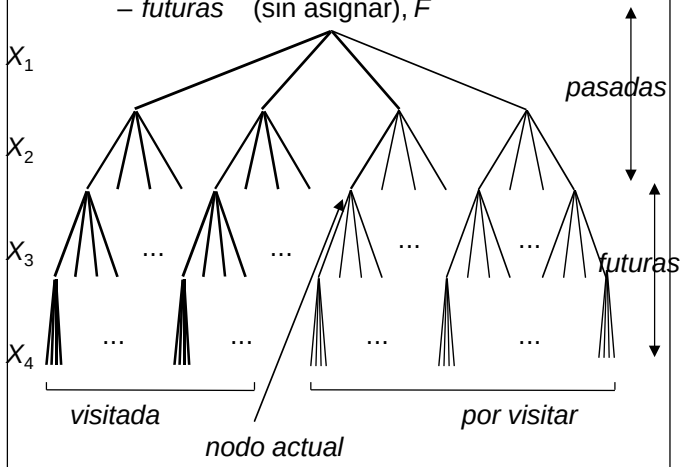
Arbol de búsqueda:

- contiene *todos* los estados
- recorrido *exhaustivo* $\rightarrow$ método *completo*

Recorrido del árbol de búsqueda

Recorrido primero en profundidad: *preorden*

- en cada nivel se asigna una nueva variable
  - variable *actual*
- variables
  - *pasadas* (asignadas), *P*
  - *futuras* (sin asignar), *F*



Si nodo actual (*P*) inconsistente:

- subárbol sucesor *no* contiene soluciones
- no se visita $\rightarrow$ *se poda*
- ventaja con respecto a *generar y testear*

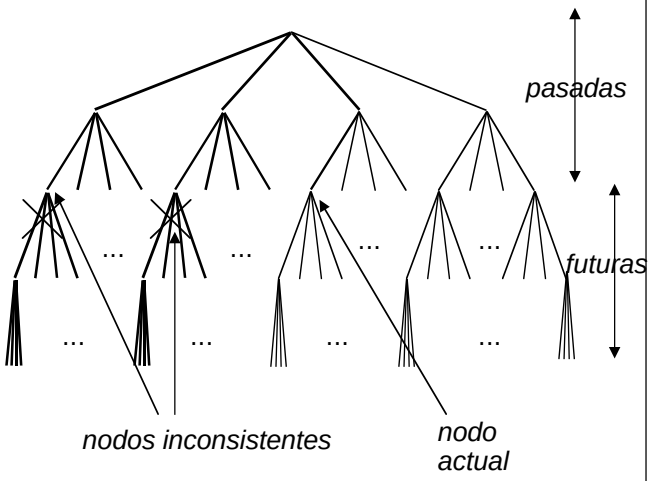
Backtracking

BT = DFS + consistencia(*P*)

DFS: búsqueda primero en profundidad

consistencia(*P*): Si *P* es consistente, continua
sino, backtracking

consistencia(*P*): es suficiente con comprobar que *actual* es consistente con variables anteriores



BT recursivo

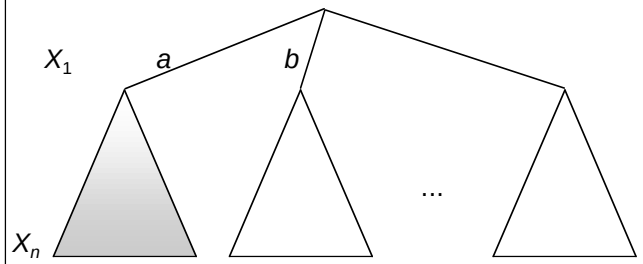
```
funcion bt(x variable): booleano
    para todo a∈D(x) hacer
        x ← a
        si test(x,pasadas)
            pasadas ← pasadas + {x}
            si bt(siguiente(x)) retorna CIERTO
            sino pasadas ← pasadas - {x}
    retorna FALSO
```

```
funcion test(x variable,P conjunto):booleano
    para todo y∈P hacer
        si (val(x),val(y))∉ rel(R_xy) retorna FALSO
    retorna CIERTO
```

Problemas del backtracking

Visión local: la variable actual sólo se compara con las variables pasadas

Situación: (1, *a*) incompatible con (*n* , _)

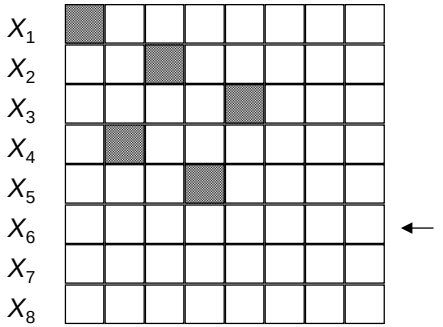


BT: recorrerá el subárbol $\{X_2, ..., X_n\}$ (1/m del árbol total) para darse cuenta de que *a* no esta en la solución.

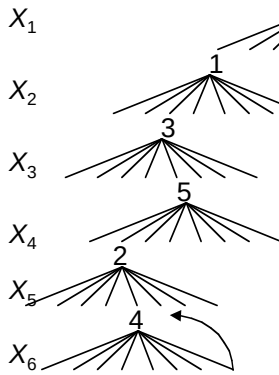
Alternativas:

- detectar que *a* es incompatible con D_n
  - -- antes de la búsqueda
  - -- al asignar *a*
- backtracking no cronológico:
  - -- saltar directamente de X_n a X_1

Backtracking cronológico

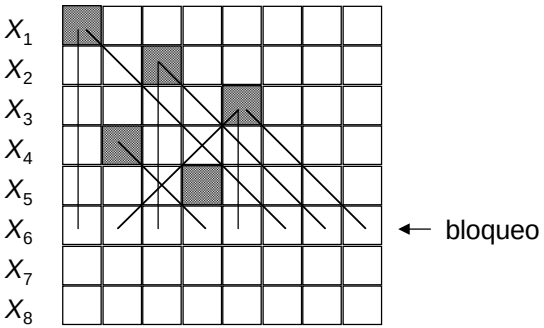


Arbol de búsqueda:



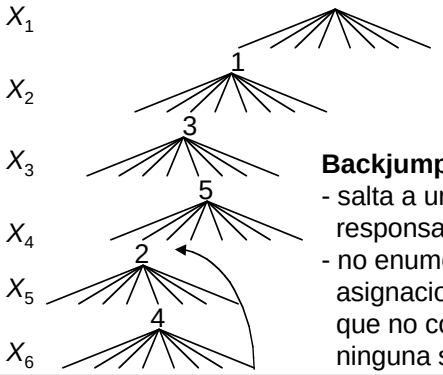
BT: enumera exhaustivamente todas las asignaciones parciales consistentes

Backjumping



Observación:

- cambiar el valor de X_5 no elimina el bloqueo
- backtracking sobre una variable anterior: X_4



Backjumping:

- salta a una variable responsable del bloqueo
- no enumera aquellas asignaciones parciales que no conducen a ninguna solución

Backjumping dirigido por conflictos

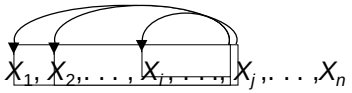
Conjunto conflicto de X_j :

- var pasadas incompatibles con algun valor de X_j

X_1									{} conjunto conflicto
X_2	1	1							{1}
X_3	1	2	1	2					{1, 2}
X_4	1								{1}
X_5	1	4	2						{1, 2, 4}
X_6	1	3	2	4	3	1	2	3	{1, 2, 3, 4}
X_7									
X_8									

Proceso :

- backjumping (X_j) = max conjunto conflicto(X_j)



- tras backjump de X_j a X_i ,
  conjunto conflicto (X_i) = conjunto conflicto (X_i) $\cup$ {conjunto conflicto (X_j) - X_i }
- se transpasan a X_i los conflictos de X_j con variables anteriores a X_i

Esquema

Introducción

- Definiciones
- Ejemplos

Metodos de resolución

- Búsqueda:
  - Generar y testear
  - Backtracking
  - Backjumping

- Inferencia ←
  - Consistencia de arcos
  - Consistencia de caminos } *incompleta*
  - K-consistencia
  - Problemas libres de backtracking } *completa*
  - Consistencia adaptativa

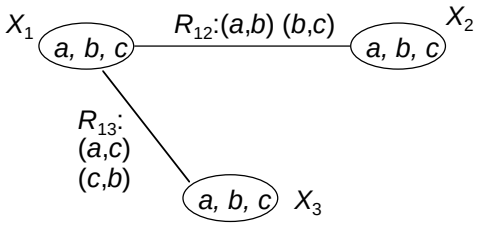
- Algoritmos híbridos
  - Forward Checking
  - Really Full Lookahead
  - Maintaining Arc Consistency
  - Heurísticas

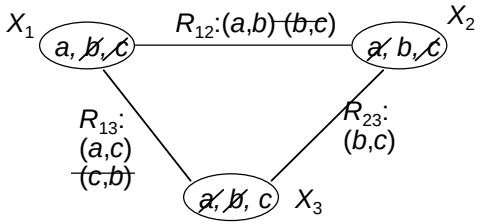
Propagación de restricciones

Restricciones:

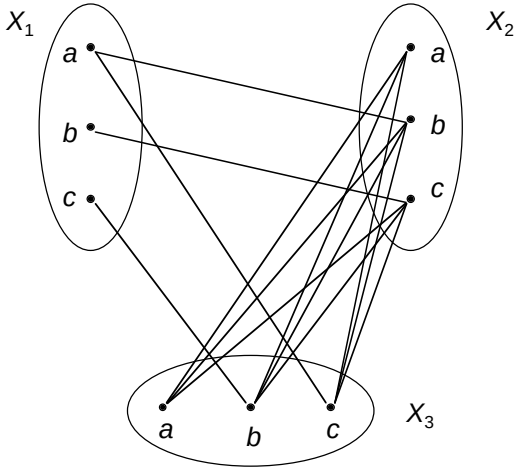
- explícitas: definición del problema
- implícitas: inducidas por la acción de las explícitas

Propagación: hace explícitas las restricciones implícitas



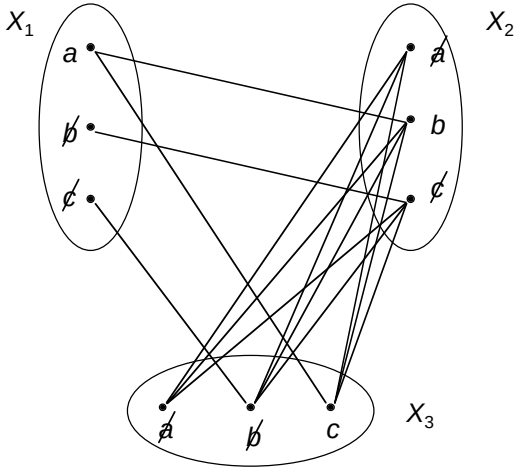


Microestructura



valores compatibles

Microestructura



valores compatibles

no se pueden extender
a una solución global

Reducción de problemas

Propagación:

- P genera un $P' = P$ + restricciones implícitas
- P' es equivalente a P, $SOL(P) = SOL(P')$
- P se *reduce* a P'

Total:

- *consistencia global*
- sintetiza una única restricción global

  tuplas permitidas = *soluciones*
- coste temporal exponencial

Parcial:

- *consistencia local*
- P' es más fácil de resolver que P

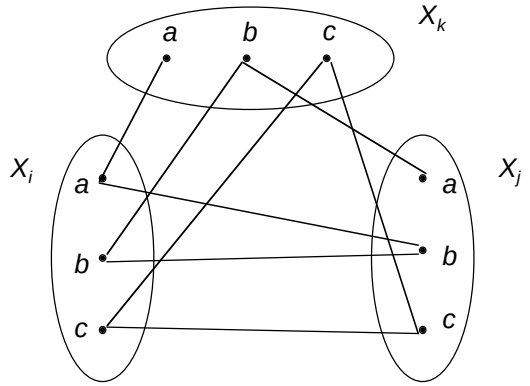
  |espacio estados P| > |espacio de estados P' |
- si $D_i = \emptyset$, final, no hay solución
- coste polinómico
- preproceso antes de búsqueda con backtracking
- intercalado con búsqueda con backtracking

Consistencia de arcos

Una restricción R_{ij} es **arco consistente direccional** (de i a j) ssi para todo valor $a \in D_i$ existe $b \in D_j$ tal que $(a, b) \in rel(R_{ij})$

Una restricción R_{ij} es **arco consistente** si es arco consistente direccional en los dos sentidos

Un problema es **arco consistente** ssi todas sus restricciones lo son.

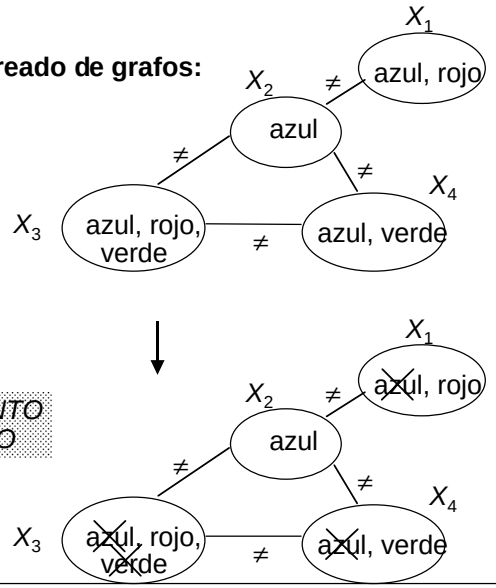


Filtrado por consistencia de arcos

Idea: si para $a \in D_i$ no existe $b \in D_j$ tal que $(a, b) \in rel(R_{ij})$ se puede eliminar a de D_i porque a no estará en ninguna solución.

Filtrado de dominios por consistencia de arcos: se eliminan los valores responsables de la arco inconsistencia.

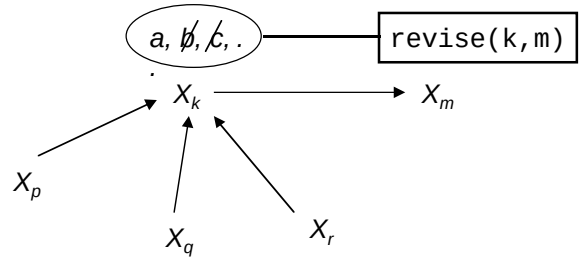
Coloreado de grafos:



AC-1: Algoritmo consistencia de arcos

```
funcion revise (i,j variable): booleano;
  delete ← FALSO
  para cada a∈Di hacer
    si no hay b∈Dj tq.(a,b)∈rel(Rij)
      Di ← Di - a
      delete ← CIERTO
  retorna delete



procedimiento AC-1 ()
  Q ←{(i,j)|(i,j)∈ arcos(G), i≠j}
  /* Rij aporta dos arcos a Q (i,j) y (j,i)*/
  repetir
    cambio ← FALSO
    para cada (i,j)∈Q hacer
      cambio ← revise(i,j) o cambio
  hasta no (cambio)


/* variables globales

     G: grafo de restriccioens
    {Di}: dominios de las variables     */
```

AC-3: Algoritmo consistencia de arcos



revise(k,m) borra b y c de D_k

¿Qué arcos hay que revisitar? Aquellos que han dejado de ser arco consistentes por el borrado de b y c.

(k,_): no, si era arco consistente,
lo sigue siendo tras el borrado
(_,k): si, pueden convertirse en arco
inconsistentes por el borrado

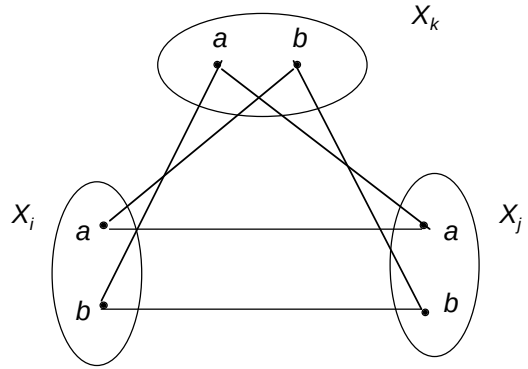
```
procedimiento AC-3 (G)
  Q ← {(i,j)|(i,j)∈ arcos(G), i≠j}
  mientras Q≠∅ hacer
    selecciona y borra un arco (k,m) de Q
    si revise(k,m) entonces
      Q:=Q ∪ {(i,k)|(i,k)∈ arcos(G),i≠k,i≠m}
```

Consistencia de caminos

- Un par de valores ((*i*, *a*) (*j, b*)), tal que (*a, b*)$\in$ *rel*(R_{ij}), es **camino consistente** ssi para todo X_k , $i\neq k$, $j\neq k$, existe $c\in D_k$ tal que,

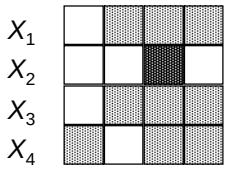
$$(a, c)\in rel(R_{ik}) \quad \text{y} \quad (c, b)\in rel(R_{kj})$$

- Un par de variables (X_i , X_j) es **camino consistente** ssi todo par de valores (*a, b*)$\in$ *rel*(R_{ij}) es camino consistente.
- Un problema *P* es **camino consistente** ssi todo par de variables es camino consistente.



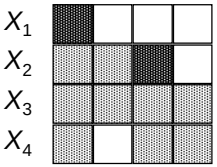
N-reinas y consistencia local

N-REINAS: para N=4 es *arco consistente*



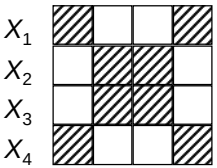
Para cualquier valor de X_i existe un valor de X_j consistente (cada reina restringe 3 posiciones)

N-REINAS: para N=4 es *no es camino consistente*



No hay valor posible para X_3

Consistencia de caminos ↓



Soluciones:

Representación matricial

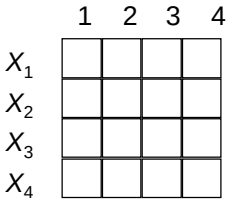
$R_{ij} \subseteq D_i \times D_j$; definición conjuntista

Dominios discretos y finitos:

- ordenar valores
- representación matricial de la restricción

$$r_{ij,\,ab} = 1 \quad \text{si } (a, b) \in R_{ij}$$
$$0 \quad \text{si } (a, b) \notin R_{ij}$$

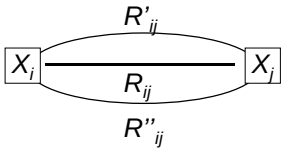
4-REINAS



$$r_{12} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Operaciones con restricciones

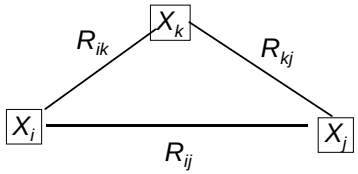
Intersección de restricciones:



$R_{ij} = R'_{ij} \cap R''_{ij}$ $\quad r_{ij,\,kl} = r'_{ij,\,kl} \wedge r''_{ij,\,kl}$

$r_{ij} = r'_{ij}\ \&\ r''_{ij}$

Composición de restricciones:

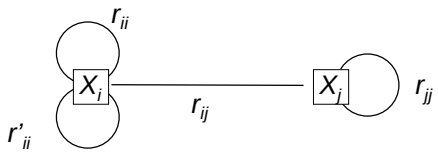


$rel(R_{ij}) = \{(a, b) | c\_D_k, (a, c) \in rel(R_{ik},),\ (c, b) \in rel(R_{kj})\}$

$r_{ij} = r_{ik} \cdot r_{kj}$ $\quad r_{ij,\,pq} = \vee_{t=1,..,\,m} (r_{ik,\,pt} \wedge r_{kj,\,tq})$

Consistencia local y operadores

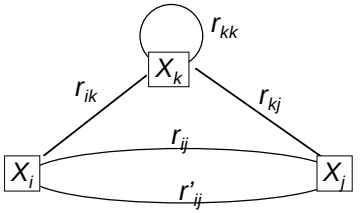
Consistencia de arcos: $r'_{ii} = r_{ii}\ \&\ r_{ij} \cdot r_{jj} \cdot r_{ji}$



- genera restricciones unitarias (filtra dominios)
- se alcanza cuando $r'_{ii} = r_{ii}$

Consistencia de caminos:

$r'_{ij} = r_{ij}\ \&\ r_{ik} \cdot r_{kk} \cdot r_{kj}$



- genera restricciones binarias
- se alcanza cuando $r'_{ij} = r_{ij}$

PC-1: Algoritmo consistencia caminos

```
procedimiento PC-1 ()
  Y^n ← {r_ij};
  repetir
    Y^0 ← Y^n;
    para k ← 1 hasta n hacer
      para i ← i hasta n hacer
        para j ← 1 hasta n hacer
          Y_ij^k ← Y_ij^k-1 & Y_ik^k-1 * Y_kk^k-1 * Y_kj^k-1;
  hasta Y^n=Y^0;
  {r_ij} ← Y^n;



/* variables globales

    r_ij: representaciones matriciales
          de las restricciones            */
```

PC-2: Algoritmo consistencia caminos

```
funcion revise (i,j,k variable)
  Z ← Y_ij & Y_ik * Y_kk * Y_kj
  si Z=Y_ij retorna FALSO
  sino
    Y_ij ← Z
    retorna CIERTO


funcion related_paths (i,k,j variable)
  S_a ← {(i,j,m)|(i≤m≤n),m≠j} ∪
        {(m,i,j)|(1≤m≤j),m≠i} ∪
        {(j,i,m)|(j<m≤n)} ∪
        {(m,j,i)|(1≤m≤i)}
  S_b ← {(p,i,m)|(1≤p≤m), (1≤m≤n),
                ¬(p=i=m), ¬(p=m=k)}
  si i<j retorna S_a
  sino retorna S_b


procedimiento PC-2 ()
  Q:={(i,k,j)|(i≤j), ¬(i=k=j)}
  mientras Q≠Ø hacer
    selecciona y borra camino (i,k,j) de Q
    si revise(i,k,j)
      Q ←  Q ∪ related_paths(i,k,j);
```

K-Consistencia

K-Consistencia:

- dado un subconjunto de *k*-1 variables asignadas $\{X_1, X_2, \ldots, X_{k-1}\}$ consistente;
- para cualquier X_k existe $d \in D_k$ tal que $\{X_1, X_2, \ldots, X_k\}$ es consistente.

K-Consistencia: *generalización*

2-consistencia: *consistencia de arcos*

3-consistencia: *consistencia de caminos*

K= *n*, propagación total

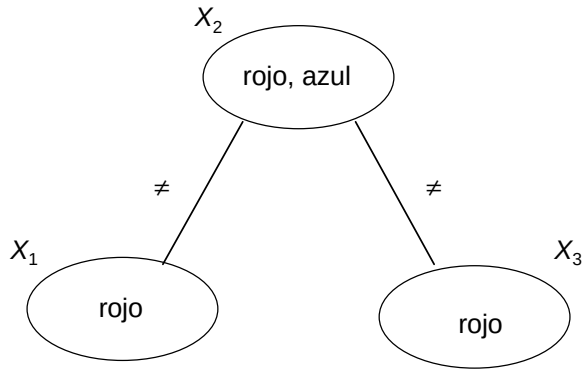
K-Consistencia fuerte:

J-consistente, para $1 \leq J \leq K$

Algoritmos para K-consistencia fuerte:

- Freuder 82, Cooper 89
- Complejidad: O(exp *K*)

K-consistencia: Ejemplo



- K-consistencia no implica K-consistencia fuerte
- Ejemplo:
  - Es 3-consistente: para cualquier par de dos variables con valores consistentes, existe un valor consistente para la tercera
  - No es 2-consistente: arco X2 - X1

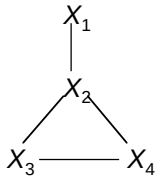
Problemas libres de backtracking

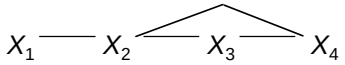
Orden de variables: { X_1, X_2,. . . , X_n}

Anchura de un nodo: # arcos a nodos anteriores

Anchura de una ordenación: max$_i$ {anchura X_i }

Anchura de un grafo: min anchura ordenaciones





anchura = 2

TEOREMA: Dado un orden de variables con anchura K, el problema se puede resolver sin backtracking si el nivel de consistencia fuerte es mayor que K. [Freuder 82]

Algoritmos:

- Mínima anchura de un grafo: O(n^2)
- K-consistencia: O(exp k).

Añade arcos extras, aumenta la anchura
No añade arcos para anchura 1
Los árboles tienen anchura 1

→ *Estructura de árbol = libre de backtracking, tras consistencia de arcos*

Consistencia Direccional

Consistencia Direccional:

- grafo ordenado, {X_1, X_2,.. , X_n},
- consistencia local siguiendo el orden:
  - consistencia de arcos direccional:
    de X_i a X_j, $i < j$
  - consistencia de caminos direccional:
    (X_i, X_j), camino de X_i, a X_j siguiendo orden

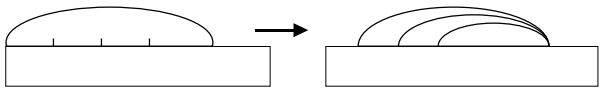
TEOREMA: un grafo de anchura 1 y consistencia de arcos direccional: es libre de backtracking. [Dechter y Pearl 88]

Algoritmos:

DAC: óptimo, O(nm^2), [Dechter y Pearl 88]
DPC: O(n^3m^3), [Dechter y Pearl 88]

Grafos regulares de anchura 2:

No cambian su anchura tras DPC
Detección: O(n) [Arnborg 85]



Consistencia Adaptativa

Consistencia adaptativa:

- grafo ordenado, $\{X_1, X_2, .. , X_n\}$,
- anchura de X_i es w_i
- consistencia direccional $w_i + 1$ en X_i

(consistencia adaptada a la anchura).

TEOREMA: un grafo ordenado con consistencia adaptativa es libre de backtracking.

[Dechter y Pearl 88]

Algoritmo:

- A-C: [Dechter y Pearl 88]

  $O(n \exp(W^* + 1))$ W^*: anchuraresultante

  Añade arcos extras, aumenta la anchura

- Cota superior: [Tarjan y Yannakakis 84]

  Calculable en $O(n + e)$

Consistencia Adaptativa

Problema P, variable X, C_X: restricciones sobre X

IDEA:

- Sustituir C_X por $\underline{c}$
- $\underline{c}$ resume el efecto de C_X sobre P
- $\underline{c}$ no menciona X

entonces X se puede eliminar

eliminación de variable

PROCESO:

problemas $P \rightarrow P' \rightarrow P'' \rightarrow \ldots \rightarrow P^{(n-1}$

variables $n \quad n-1 \quad n-2 \quad 1$

SOLUCION:

- $S^{(n-1}$ solución de $P^{(n-1}$

  ↓

- $S^{(n-2}$ solución de $P^{(n-2}$ a partir de $S^{(n-1}$

  ↓

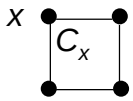
- . . .

  ↓

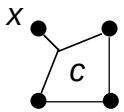
- S solución de P a partir de S'

Eliminación de Variable

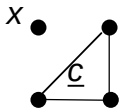
Para eliminar var *X*:



- *Join* todas las restricciones $C_x \rightarrow c$
- Sustituir C_x por c



- Proyectar la variable x fuera $c \rightarrow \underline{c}$
- Sustituir c por $\underline{c}$

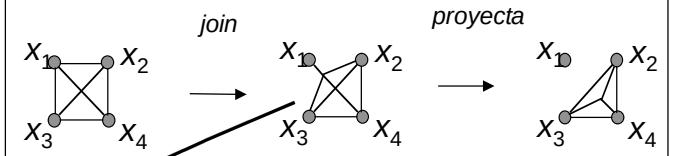


- Si existe c' $var(c') = var(\underline{c})$, $\underline{c} \leftarrow \underline{c} \cap c'$

Para obtener la solución:

- Variables se procesan en orden inverso
- Se asigna a X un valor consistente con variables anteriores, y con restricciones intermedias totalmente asignadas

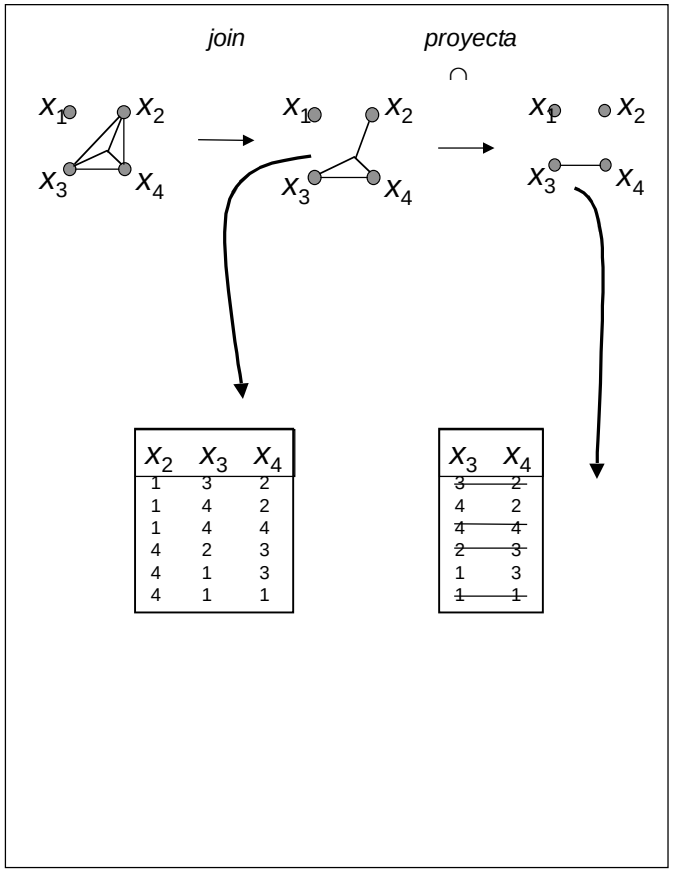
Ejemplo: 4-reinas (x_1)



x_1	x_2	x_3	x_4
1	3	2	2
1	3	2	3
1	3	4	2
1	3	4	3
1	4	2	2
1	4	2	3
1	4	4	2
1	4	4	3
2	4	1	1
2	4	1	3
2	4	1	4
2	4	3	1
2	4	3	3
2	4	3	4

x_1	x_2	x_3	x_4
3	1	4	4
3	1	4	2
3	1	4	1
3	1	2	4
3	1	2	2
3	1	2	1
4	2	3	3
4	2	3	2
4	2	1	3
4	2	1	2
4	1	3	3
4	1	3	2
4	1	1	3
4	1	1	2

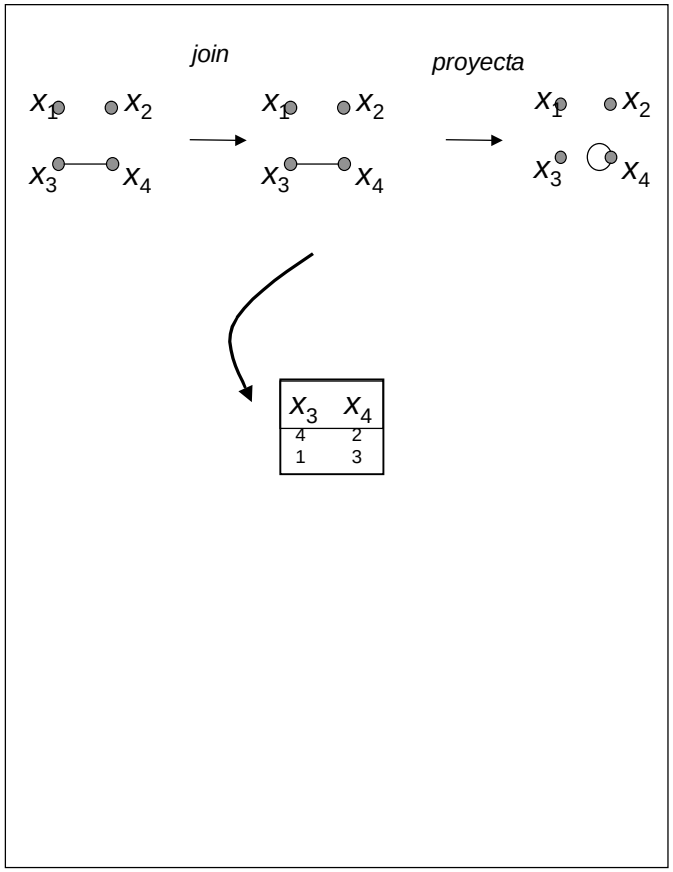
Ejemplo: 4-reinas (x_2)



x_2	x_3	x_4
1	3	2
1	4	2
1	4	4
4	2	3
4	1	3
4	1	1

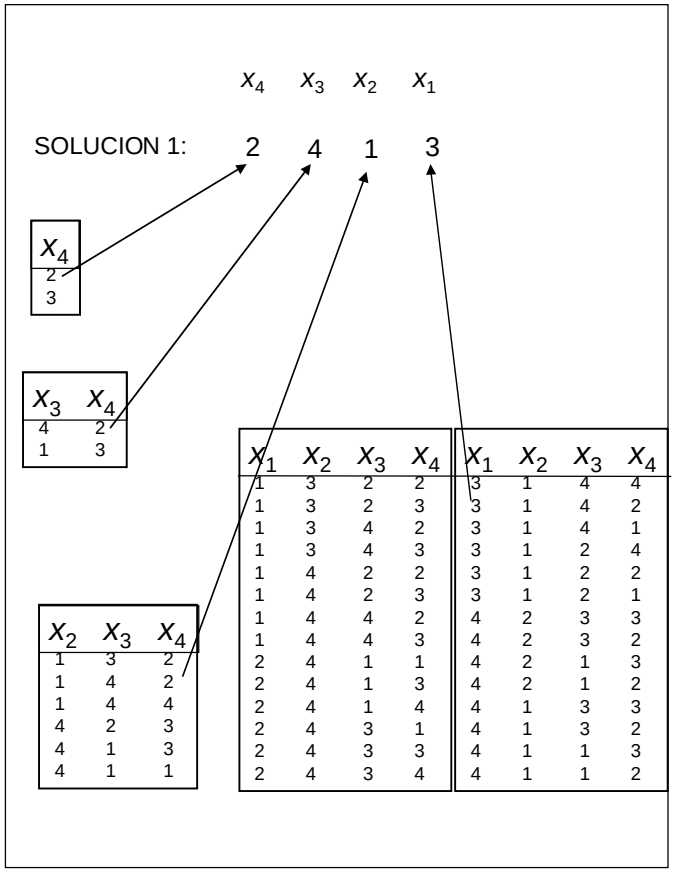
x_3	x_4
~~3~~	~~2~~
4	2
~~4~~	~~4~~
~~2~~	~~3~~
1	3
~~1~~	~~1~~

Ejemplo: 4-reinas (x_3)

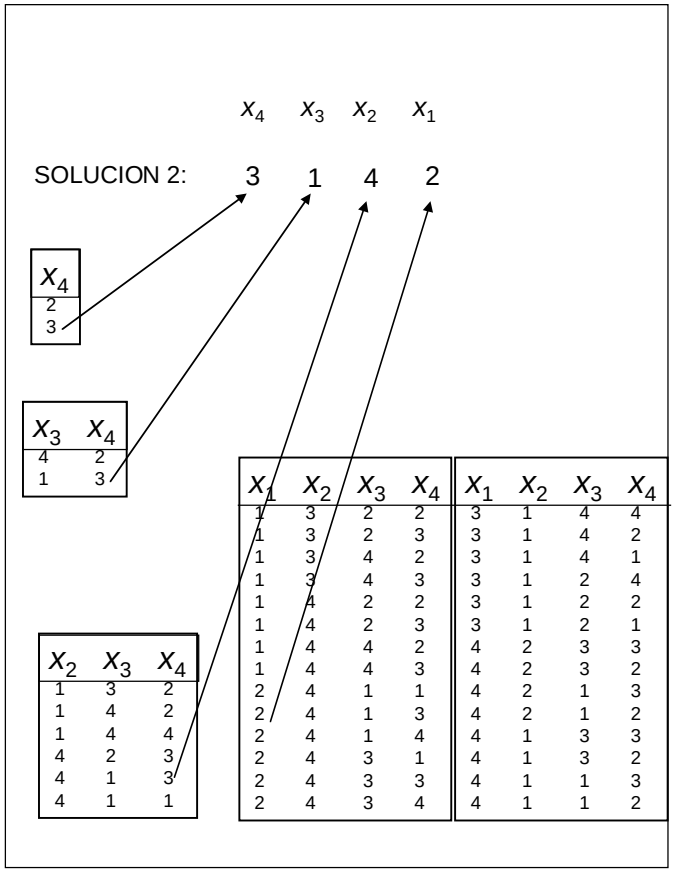


x_3	x_4
4	2
1	3

Ejemplo: Todas las soluciones 4-reinas



Ejemplo: Todas las soluciones 4-reinas



Esquema

Introducción
- Definiciones
- Ejemplos

Metodos de resolución
- Búsqueda:
  - Generar y testear
  - Backtracking
  - Backjumping
- Inferencia
  - Consistencia de arcos
  - Consistencia de caminos
  - K-consistencia

  } *incompleta*
  - Problemas libres de backtracking
  - Consistencia adaptativa

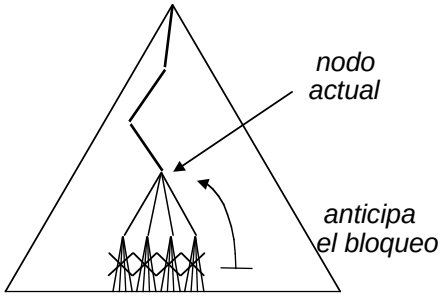
  } *completa*
- Algoritmos híbridos ←
  - Forward Checking
  - Really Full Lookahead
  - Maintaining Arc Consistency
  - Heurísticas

Algoritmos Híbridos

Combinación: búsqueda + propagación

Búsqueda: BT (DFS)

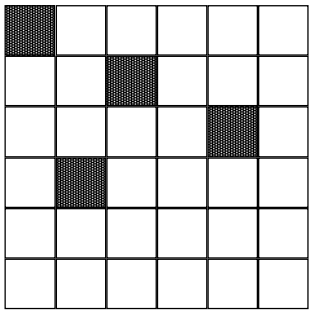
Propagación: en cada nodo
- propaga el efecto de la asignación actual
- bloqueo: rama sin soluciones
- objetivo: *anticipar* la detección de bloqueos
- si bloqueo → backtracking



Compromiso:
- \+ evito visitar un número multiplicativo de nodos
- \- coste de la anticipación en cada nodo

6 reinas

¿Hay solución en los descendientes de este nodo?



6 reinas

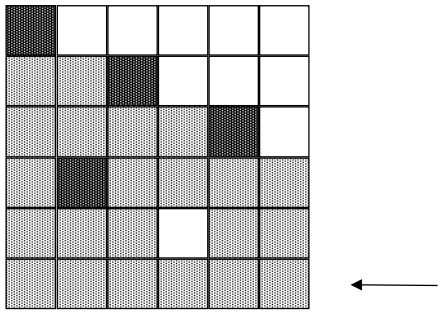
No hay solución. Ultima variable sin valores factibles.

Anticipación:

eliminar columnas y diagonales prohibidas

→ eliminar valores de dominios futuros

bloqueo ⟷ dominio vacío



Esquema de anticipación

1. Búsqueda:

1. Variable actual X_i $D_i = \{a, b, c\}$
2. Asignación $X_i \leftarrow a \Leftrightarrow D_i = \{a\}$

2. (Consistencia entre pasadas)

3. Tras asignar ($\Leftrightarrow$ reducción de dominio)

- consistencia local
  - parte / todo el problema
  - nivel de consistencia
- ***si algún dominio futuro vacío***
  - bloqueo
  - backtracking
- sino, se continúa la búsqueda

Compromiso coste / beneficio

- coste, beneficio $\uparrow$ con
  - tamaño parte localmente consistente
  - nivel de consistencia
- crecimiento desigual
- compromiso óptimo:
  - depende del problema
  - en general, consistencia de arcos

Forward checking

FC = BT + AC(C_{PF})

AC(C_{PF}) : eliminar en dominios futuros valores incompatibles con las variables pasadas

FC = BT + AC(C_{PF})

BT = DFS + consistencia(P)

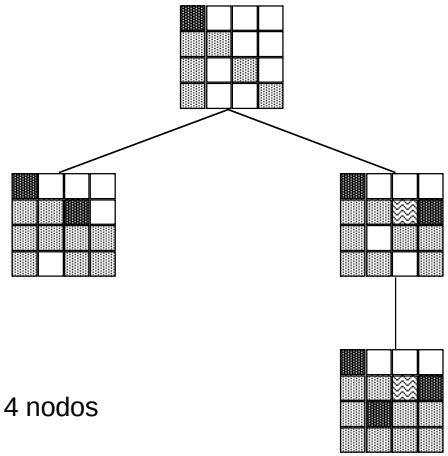
FC = DFS + consistencia(P) **+ AC**(C_{PF})

AC(C_{PF}) $\longrightarrow$ **consistencia**(P)

- tras **AC**(C_{PF}) en dominios de F sólo hay valores consistentes con P
- cuando se tome una nueva variable de F, esta tendrá todos sus valores consistentes con P

FC = DFS + AC(C_{PF})

FC: Ejemplo



4 nodos

valor asignado

valor podado

valor usado sin éxito

Forward checking

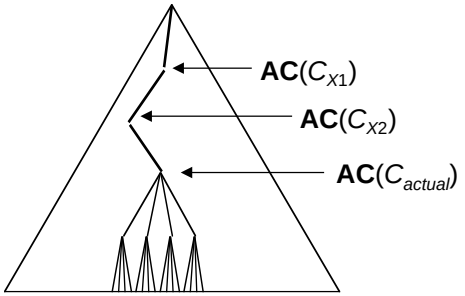
FC = DFS + AC(C_{PF})

$P = \{X_1, X_2, \ldots, X_i\}$

AC(C_{PF}) = **AC**(C_{X1}) + **AC**(C_{X2}) + ... + **AC**(C_{Xi})

AC(C_{PF})

- en cada nodo **AC**(C_{actual})
- acumular resultados a lo largo de la rama actual:
  - **AC**(C_{actual}) poda dominos futuros
  - sucesores de *actual* heredan dominios futuros podados

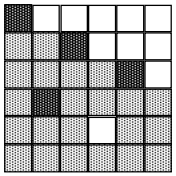


Forward checking

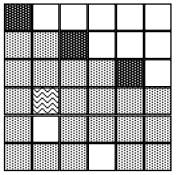
FC = DFS + AC(C_{actual})

AC(C_{actual}): efecto lateral

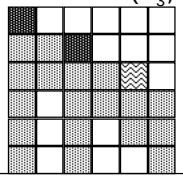
- poda en dominios futuros
- backtracking: restaurar valores podados tras la asignación de la variable *actual*



restaura(X_4,2)



restaura(X_3,5)



Algoritmo FC

```
funcion FC(i variable): booleano
| para cada a∈factibles[i] hacer
| | Xi ← a
| | si i=N solución retorna CIERTO
| | sino
| | | si forward(i,a)
| | | | si FC(i+1) retorna CIERTO
| | | restaurar(i)
| retorna FALSO


funcion forward(i variable,a valor): booleano
| para toda j=i+1 hasta N hacer
| | vacio ← CIERTO
| | para cada b∈factibles[j] hacer
| | | si (a,b)∈Rij vacio ← FALSO
| | | sino eliminar b de factible[j]
| | |      añadir b a podado[j]
| | si vacio retorna FALSO
| retorna CIERTO



procedimiento restaura(i variable)
| para toda j=i+1 hasta N hacer
| | para todo b∈podado[j] hacer
| | | si Xi resposable filtrado b
| | | |  eliminar b de podado[j]
| | | |  anadir b a factible[j]
```

Really full lookahead

RFL = FC + AC(F)

RFL = DFS + AC(C_{actual}) **+ AC**(C_F)

AC(C_F):

- poda extra en dominios futuros
- después de **AC**(C_{actual})

RFL / FC:

- mayor nivel de poda:

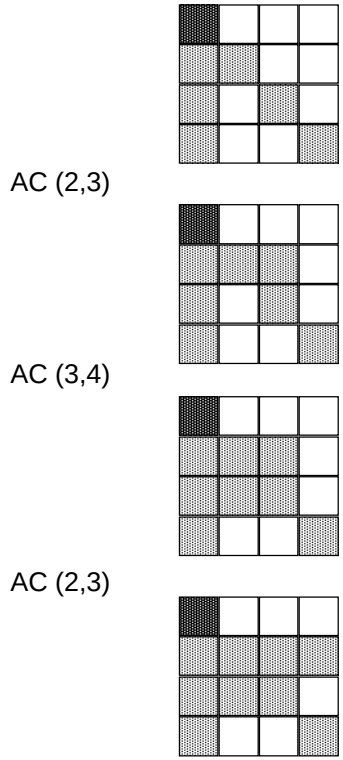
  poda(**RFL**) ≥ poda(**FC**)

- coste extra:

  AC(C_F)

- adecuado para
  - restricciones duras
  - problemas esparsos

RFL: Ejemplo



Traza AC, 1 solo nodo

Algoritmo RFL

```
funcion RFL(i variable): booleano
  para cada a∈factible[i] hacer
    X_i ← a
    si i=N retorna CIERTO
    sino
      si ac-forward(i,a)
        si RFL(i+1) retorna CIERTO
      restaura(i)

funcion ac-forward(i variable,a valor): bool
  retorna forward(i,a) AND AC(F)

funcion AC (F conjunto): booleano   /* AC3 */
  Q ← {(i,j)|i,j∈F,(i,j)∈ arcos(G),i≠j}
  mientras Q≠Ø hacer
    selecciona y borra un arco (k,m) de Q
    si revise(k,m) entonces
      si factibles[i]=Ø retorna FALSO
      sino Q ← Q ∪ {(i,k)|i,k∈F,
                    (i,k)∈ arcos(G),i≠k,i≠m}
  retorna CIERTO

funcion revise (i,j variable): booleano
  delete ← FALSO
  para cada a∈factibles[i] hacer
    si no hay b∈factibles[j] tq.(a,b)∈rel(R_ij)
      borra a de factibles[i]
      delete ← CIERTO
  retorna delete


/*  funcion forward
    procedimiento restaura    como en FC   */
```

Maintaining arc consistency

C_{antes} restricciones **antes** de asignar *actual*

$C_{despues}$ restricciones **después** de asignar *actual*

$$\mathbf{MAC = DFS + AC}(C_{antes}) \mathbf{+ AC}(C_{despues})$$

MAC: en cada nodo

- **antes** de la asignación, convierte el subproblema actual en arco consistente,

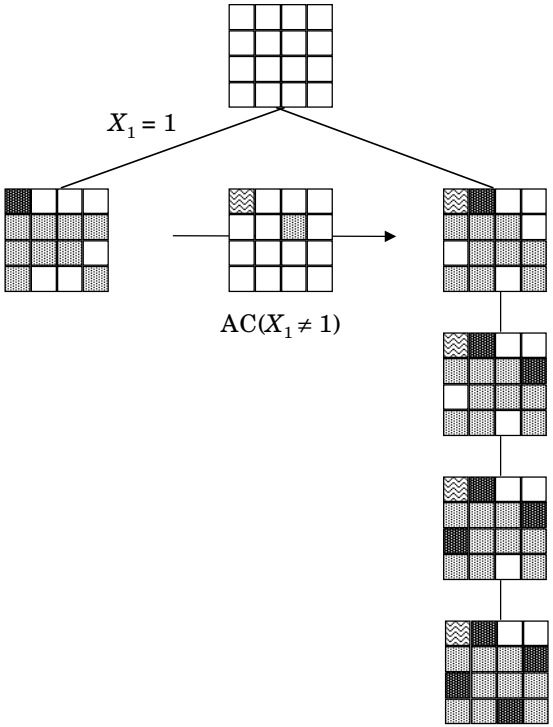
$$\mathbf{AC}(C_{antes})$$

- **después** de la asignación, como **RLF**,

$$\mathbf{AC}(C_{despues}) = \mathbf{AC}(C_{actual}) + \mathbf{AC}(C_F)$$

$$\mathbf{MAC = DFS + AC}(C_{antes}) \mathbf{+ AC}(C_{actual}) + \mathbf{AC}(C_F)$$

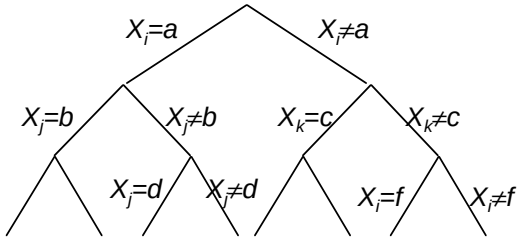
MAC: Ejemplo



MAC: árbol de búsqueda

Arbol de búsqueda:

- binario
- en cada nivel
  - una variable X_i
  - dos opciones: a, $\neg a$
- se puede cambiar de variable sin agotar los valores
- en cada nodo, arco consistencia del subproblema actual

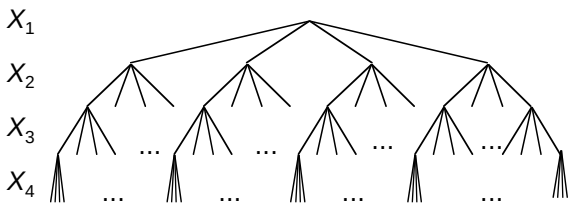


Algoritmo MAC

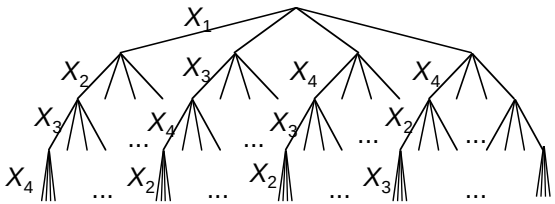
```
funcion MAC(i variable): booleano
  mientras factibles[i]≠Ø hacer
    a ← extrae(factibles[i])
    X_i ← a                          /* X_i en P */
    si i=N retorna CIERTO
    sino
      si ac-forward(i,a)
         si MAC(i+1) retorna CIERTO
      restaura(i)
      añadir a a usados[i]
      X_i ← Ø                        /* X_i en F */
      si ¬refutar(i,a) salir bucle
  para cada a∈usados[i] hacer
    restaura_refutar(i,a)
    anadir a factibles[i]
  retorna FALSO

funcion refutar(i variable,a valor): booleano
  retorna AC(F)

procedimiento restaura_refutar(i variable,
                               a valor)
  para toda j=i+1 hasta N hacer
    para todo b∈podados[j] hacer
      si refutacion de (i,a) resp. filtrado b
        eliminar b de podados[j]
        añadir b a factibles[j]

/*  funcion ac-forward
    funcion AC
    procedimiento restaura

    Iguales a los de RFL        */
```

Ordenación de variables

Ordenación estática de variables: cada nivel del árbol de búsqueda se asocia con una variable



- No es necesaria para que el árbol de búsqueda sea exhaustivo.
- Es necesario que cada nodo se asocie con una variable para la generación de sucesores.



- Diferentes variables en el mismo nivel: **ordenación dinámica de variables**

Heurística: selección de variable

En nodo *q* ¿qué variable asignar a continuación?

1. Hay solución en *descendientes*(*q*)
   – cualquier variable es adecuada

2. No hay solución en *descendientes*(*q*)
   – asignar la variable que antes descubre que no hay solución

En general, ¿qué situación es más frecuente?

- Salvo problemas triviales, situación 2
- Mayor esfuerzo del algoritmo: salir de subproblemas sin solución

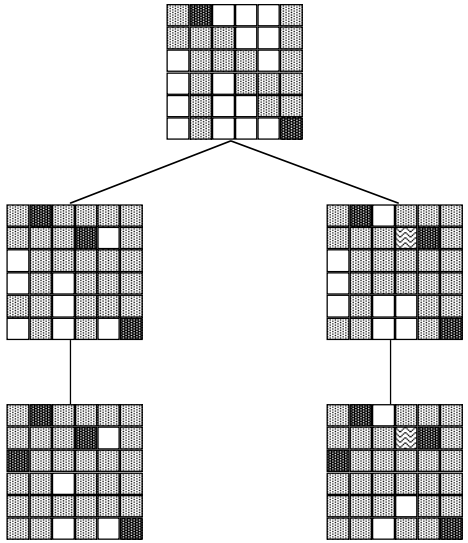
Suponemos situación 2:

asignar primero aquella variable que, aparentemente, antes nos conduce a un fallo (principio *fail-first*)

Heurística dominios mínimos:

asignar primero la variable con menor número de valores factibles

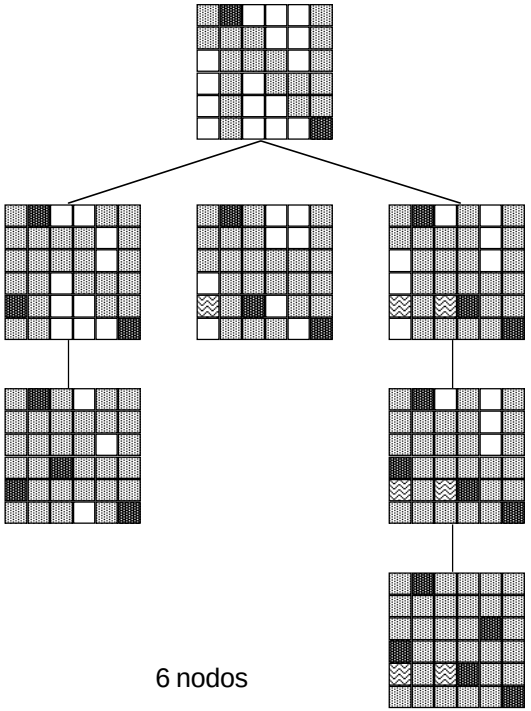
Ejemplo: 6 reinas



4 nodos

para detectar que no hay solución con (1,2) (6,6)

Ejemplo



6 nodos

para detectar que no hay solución con (1,2) (6,6)

Heurística: selección de valor

En nodo *q* ¿qué valor asignar a continuación?

1. Hay solución en *descendientes*(*q*)
   – un valor que mantenga la resolubilidad del nodo sucesor

2. No hay solución en *descendientes*(*q*)
   – cualquier valor es adecuado

Suponemos situación 1:

asignar primero aquella valor que, aparentemente, antes nos conduce al éxito (principio *success-first*)

Heurística anticipación valores:

asignar primero el valor que es consistente con mayor número de valores factibles del resto de variables

Coste heurísticas

Compromiso coste / beneficio:

- coste: suma de costes de cálculo en cada nodo
- beneficio en todo el árbol
- sale a cuenta si coste < beneficio

Dominios mínimos y anticipación:

- Dominios mínimos: exige calcular el número de valores factibles
- Algoritmos de anticipación: calculan los valores factibles
- Anticipación obtiene dominios mínimos *sin coste* adicional

Evaluación de las heurísticas:

- Evaluación empírica
- Esfuerzo computacional
  - constraint checks / tiempo CPU

Resultados empíricos, 20 reinas, 1ª solución

BT	25.428.842 cc
FC	2.398.022 cc
FC+dom min	4.144 cc