

Inteligencia Artificial

Problemas de satisfacción de restricciones

Primavera 2007

profesor: Luigi Ceccaroni

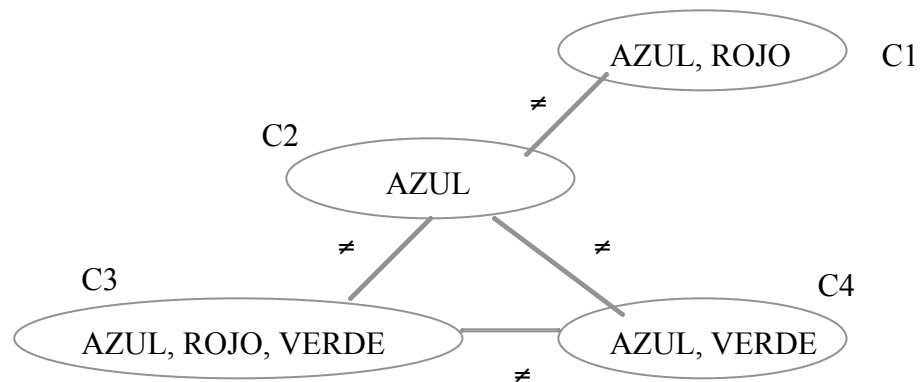
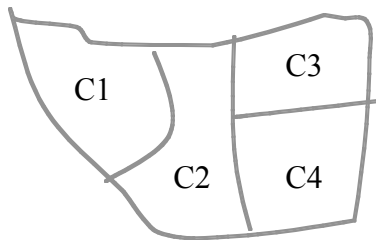


Problemas de satisfacción de restricciones (PSRs)

- Componentes del *estado* = **grafo de restricciones**:
 - Variables
 - Dominios (valores posibles para las variables)
 - Restricciones (binarias) entre las variables
- Objetivo: encontrar un estado (una asignación **completa** de valores a las variables) que satisfice las restricciones.
- Ejemplos:
 - colorear mapas, crucigramas, n-reinas
 - asignación/distribución/ubicación de recursos:
distribución de tareas de fabricación, ubicación de gasolineras de hidrogeno y antenas de telefonía

Representación

- Problema: colorear mapa
- Ejemplo de estado:
 - Variables (n) = etiquetas de nodos
 - Dominios = contenido de nodos
 - Restricciones = arcos dirigidos y etiquetados entre nodos



Estado inicial

Representación

- A cada paso hay una asignación de variable.
- Las soluciones deben ser asignaciones completas y por lo tanto, en el árbol de búsqueda, aparecen a profundidad n .
- El árbol se extiende sólo a profundidad n .
- Los algoritmos de búsqueda primero en profundidad son populares para PSRs.
- El camino que alcanza una solución es irrelevante.
- La clase más simple de PSR implica **variables discretas y dominios finitos**: problemas de coloreo de mapas, de n -reinas...

Dominios finitos

- Si el tamaño máximo del dominio de cualquier variable es d , entonces el número de posibles asignaciones completas es $O(d^n)$, exponencial en el número de variables.
- Los PSRs con **dominio finito** incluyen a los PSRs *booleanos*, cuyas variables pueden ser *verdaderas* o *falsas*.
- En el caso peor, no se pueden resolver los PSRs con dominios finitos en menos de un tiempo exponencial.
- En la mayoría de aplicaciones prácticas, sin embargo, los algoritmos para PSR resuelven problemas órdenes de magnitud más grandes que los resolubles con algoritmos de búsqueda no informada.

Dominios infinitos

- Las variables discretas pueden tener **dominios infinitos** (p.e., el conjunto de números enteros o de cadenas).
- Con dominios infinitos, no es posible describir restricciones enumerando todas las combinaciones permitidas de valores.
- Se debe usar un **lenguaje de restricción**:
 - $\text{Comienzo-Trabajo}_1 + 5 \leq \text{Comienzo-Trabajo}_3$
- Existen algoritmos para **restricciones lineales** sobre variables enteras.
- No existe un algoritmo general para **restricciones no lineales** sobre variables enteras.
- En algunos casos, se pueden reducir los problemas de dominio infinito a dominio finito simplemente acotando los valores de todas las variables, p.e.:
 - fijando límites temporales para el comienzo de los trabajos.

Dominios continuos

- Los PSRs con **dominios continuos** son muy comunes en el mundo real.
- La categoría más conocida son los problemas de **programación lineal**: las restricciones deben ser desigualdades lineales que forman una región *convexa*.
- Los problemas de programación lineal pueden resolverse en tiempo *polinomial* en el número de variables.

Algoritmos

- Basados en búsqueda
 - complejidad:
 - tiempo: $O(\exp(n))$
 - espacio: $O(n)$
- Basados en programación dinámica
 - complejidad:
 - tiempo: $O(\exp(w^*+1))$
 - espacio: $O(\exp(w^*))$
- Donde n es el número de variables y w^* es la anchura inducida del grafo de restricciones dado un orden.

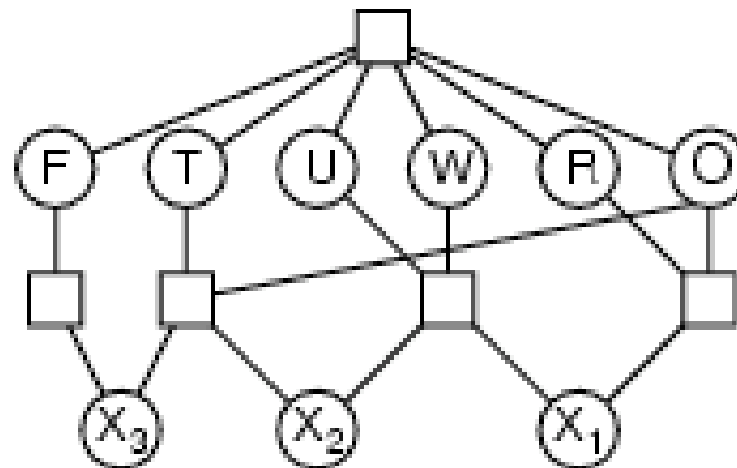
Algoritmos basados en búsqueda

- Formulación *incremental* como en un problema de búsqueda estándar:
 - **Estado inicial:** la asignación vacía { }, en que todas las variables no están asignadas.
 - **Función de sucesor:** asignación de un valor a cualquier variable no asignada, a condición de que no suponga conflicto con variables ya asignadas.
 - **Test objetivo:** la asignación actual es completa.
 - **Costo del camino:** un costo constante (p.e., 1) para cada paso.
- Formulación *completa* de estados: cada estado es una asignación completa que satisface o no las restricciones. Los métodos de búsqueda local trabajan bien para esta formulación.
- Búsqueda heurística
 - Búsqueda en profundidad con vuelta atrás cronológica
- Propagación de restricciones
 - Antes de la búsqueda
 - Durante la búsqueda

Restricciones

- El tipo más simple es la **restricción *unaria***, que restringe los valores de una sola variable.
- Una **restricción binaria** relaciona dos variables.
- Las restricciones de orden alto implican tres o más variables. Un ejemplo son los puzzles *cripto-aritméticos*.

$$\begin{array}{r} \text{ T W O} \\ + \text{ T W O} \\ \hline \text{ F O U R} \end{array}$$



Puzzles cripto-aritméticos

- Variables: $F T U W R O X_1 X_2 X_3$
- Dominio: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Restricciones:
 - *TodasDif* (F, T, U, W, R, O)
 - $O + O = R + 10 \cdot X_1$
 - $X_1 + W + W = U + 10 \cdot X_2$
 - $X_2 + T + T = O + 10 \cdot X_3$
 - $X_3 = F, T \neq 0, F \neq 0$

Búsqueda en profundidad con vuelta atrás cronológica

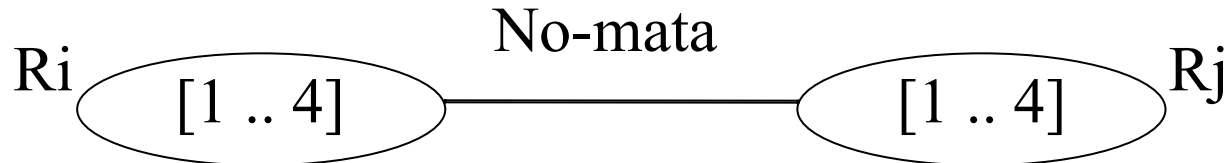
- Búsqueda en profundidad sobre las variables
- Asignar valor por estrategia exhaustiva
 - Comprobar restricciones tras cada posible asignación
 - Si no se satisfacen para ningún valor, vuelta atrás sobre la última asignación válida
- Tipos de variables:
 - pasadas
 - actual
 - futuras

Algoritmo de vuelta atrás cronológica

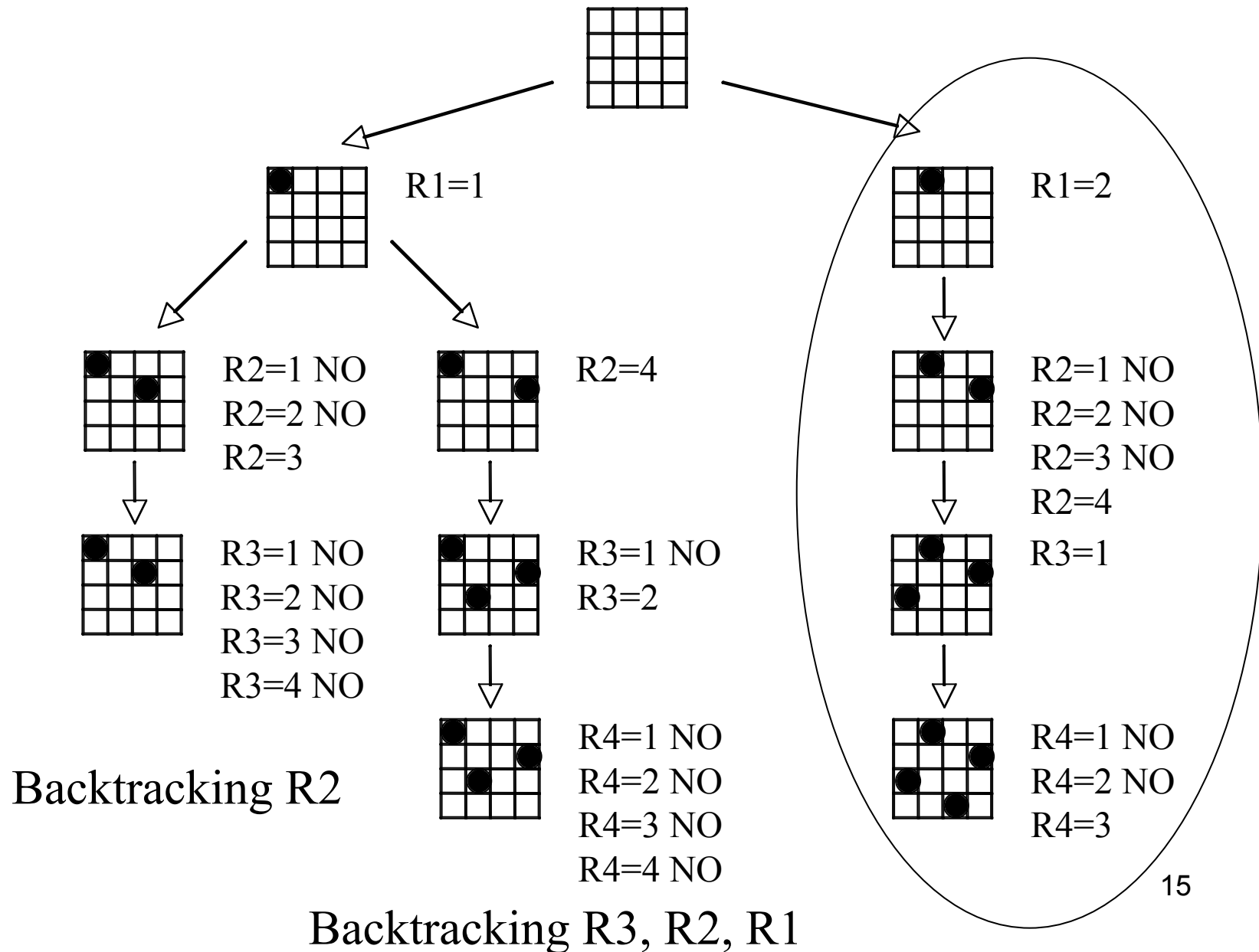
- 1. Asignar a cada variable el valor indefinido. Pila vacía.
- 2. **Seleccionar una variable** futura como variable actual.
- **Si hay**, borrar de futuras y empilar (top = variable actual),
- **sino**, la asignación actual es SOLUCIÓN.
- 3. **Seleccionar un valor** no usado para la variable actual.
- **Si hay**, marcar el valor como usado,
- **sino**, asignar valor indefinido a la variable actual,
- marcar todos sus valores como no usados,
- desempilar la variable y añadirla a futuras,
- si pila vacía, NO HAY SOLUCIÓN, sino, ir a 3.
- 4. **Comprobar las restricciones** entre las variables pasadas y la actual.
- **Si satisfechas**, ir a 2, sino ir a 3.
- (Es posible usar heurísticas para seleccionar variables (2.) y valores (3.))

Ejemplo: 4-reinas

- Colocar 4 reinas, **una en cada fila** de un tablero 4x4, sin que se maten
 - Variables: R_1, \dots, R_4 (reinas)
 - Dominios: $[1 .. 4]$ para cada R_i (columna)
 - Restricciones: R_i no-mata R_j
- Grafo:

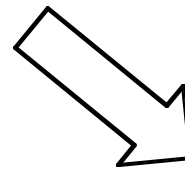
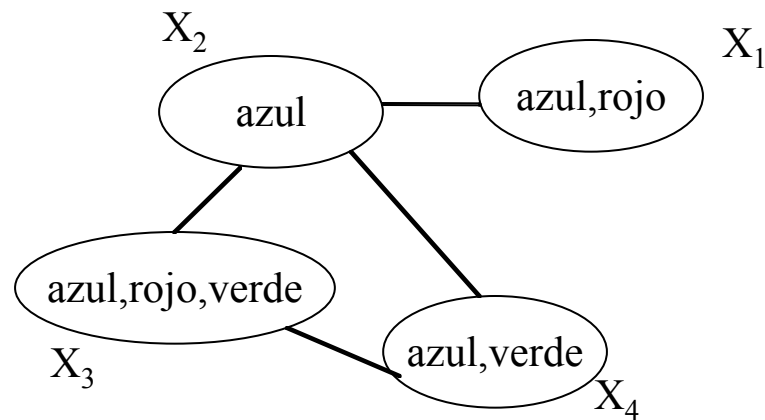


Ejemplo: 4-reinas



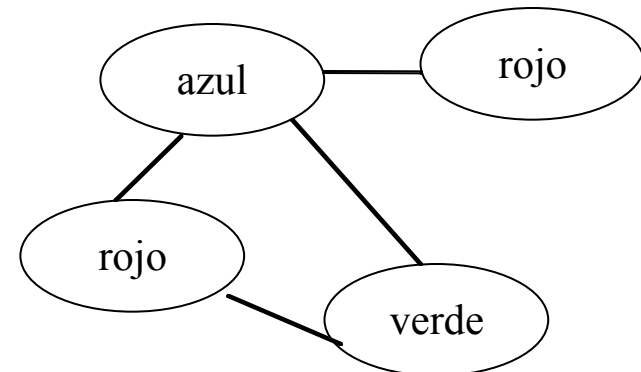
Propagación de restricciones

Un conjunto de restricciones puede inducir otras (que estaban implícitas).
La **propagación de restricciones (PR)** es el proceso de hacerlas explícitas



El papel de la PR es disminuir el espacio de búsqueda. Se puede realizar la propagación:

- 1) como preproceso: eliminar zonas del espacio donde no hay soluciones (*arc consistency*);
- 2) durante el proceso: podar el espacio a medida que la búsqueda progresa (*forward checking*).



Propagación de restricciones

Cada ciclo tiene dos partes:

1) Se propagan las restricciones:

- Posible utilización de reglas de inferencia
- Tener en cuenta que las restricciones no tienen por qué ser independientes (Muchas restricciones implican a varias variables, una variable participa en muchas restricciones.)

2) Se analiza el resultado:

2.1 Solución encontrada

2.2 Solución imposible

2.3 Seguir buscando: proceso heurístico de búsqueda

Propiedades sobre grafos de restricciones

- Se pueden definir propiedades sobre los grafos de restricciones que permiten reducir el espacio de búsqueda.
- *k-consistency*: poda de valores que no sean posibles para un grupo de k variables:
 - *Arc consistency* (*2-consistency*): Eliminamos valores imposibles para parejas de variables.
 - *Path consistency* (*3-consistency*): Eliminamos valores imposibles para ternas de variables.
 - ...
- Comenzar con un grafo *k-consistent* (2, 3, ...) reduce el número de vueltas atrás.

PR: preproceso

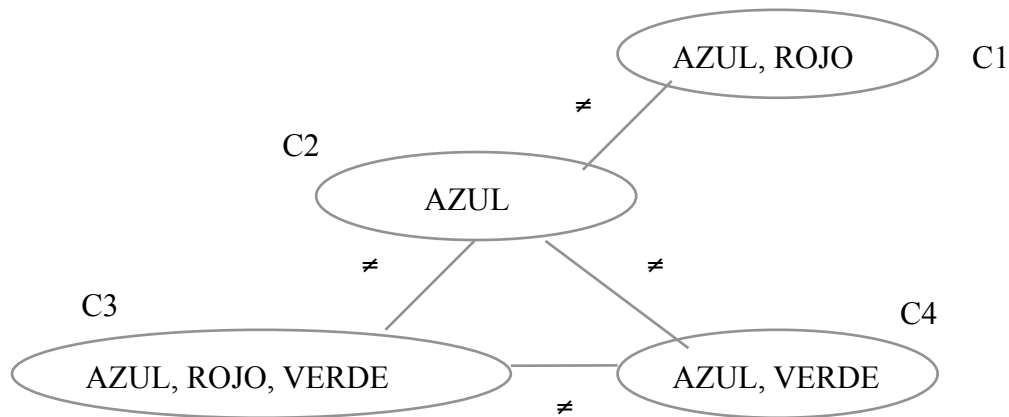
Un PSR es arco-consistente si para cada par de variables (X_i, X_j) y para cualquier valor v_k de D_i existe un valor v_l de D_j tal que se satisfacen las restricciones. Es decir se trata de que los valores posibles de X_i sean consistentes con la restricción asociada al arco.

Lo que realmente pretendemos es que todas las variables sean arco consistentes para todos los arcos que inciden en ellas. Es decir que los dominios actuales de cada variable sean consistentes con todas las restricciones.

Si un PSR no es arco-consistente se le puede convertir mediante el siguiente algoritmo:

- 1 Añadir a lista_arcos los arcos dirigidos del grafo de restricciones
- 2 mientras no vacia (lista_arcos) hacer
 - 2.1 seleccionar un arco (X_i, X_j) de lista_arcos, eliminarlo
 - 2.2 si existe un valor v_k de D_i tal que no existe un valor v_l de D_j compatible se elimina v_k de D_i . Se añaden a lista_arcos aquellos arcos de G que terminan en X_i excepto el inverso del analizado (es decir se vuelve a analizar la consistencia de arcos cuyo destino ha podido cambiar)

Ejemplo: Colorear Mapa



Lista inicial:

(C1,C2), (C2,C1), (C2,C3), (C3,C2),
(C2,C4), (C4,C2), (C3,C4), (C4,C3)

(C1,C2): eliminar AZUL

(C2,C1): ok

(C2,C3): ok

(C3,C2): eliminar AZUL

(C2,C4): ok

(C4,C2): eliminar AZUL

(C3,C4): eliminar VERDE

añadir (C2,C3)

(C4,C3): ok

(C2,C3): ok

Propagación durante la búsqueda (*forward checking*)

- Modificación del algoritmo de búsqueda en profundidad con vuelta atrás cronológica
- **Anticipación:** detectar cuanto antes caminos sin solución y podarlos.
 - Asignar un valor y consultar las restricciones sobre las variables futuras con arco desde la actual
 - Se eliminan valores no compatibles de los dominios correspondientes a dichas variables futuras
- Equivale a hacer arco-consistente la variable actual con las futuras en cada paso
- La eficiencia dependerá del problema (incrementamos el coste de cada iteración)

Algoritmo *forward checking*

1. Asignar a cada variable el valor indefinido. Pila vacía.
2. **Seleccionar una variable** futura como variable actual
Si hay, borrar de futuras y empilar (top = variable actual)
sino, la asignación actual es SOLUCIÓN
3. **Seleccionar un valor** no usado para la variable actual
Si hay, marcar el valor como usado
sino, asignar valor indefinido a la variable actual
 marcar todos sus valores como no usados
 desempilar la variable y añadirla a futuras
 si pila vacía, NO HAY SOLUCIÓN, sino, ir a 3
4. **Propagar las restricciones** a las variables futuras
 si el dominio de una de ellas resulta vacío, ir a 3

Ejemplo: 4-reinas

R1=1? Propagamos: R2=3,4; R3=2,4; R4=2,3 \Rightarrow **R1=1**

R2=3? Propagamos: R3= \emptyset

R2=4? Propagamos: R3=2; R4=3 \Rightarrow **R2=4**

R3=2? Propagamos: R4= \emptyset

Ningún otro valor para R3. Backtracking a R2

Ningún otro valor para R2. Backtracking a R1

R1=2? Propagamos: R2=4; R3=1,3; R4=1,3,4 \Rightarrow **R1=2**

R2=4? Propagamos: R3=1; R4=1,3 \Rightarrow **R2=4**

R3=1? Propagamos: R4=3 \Rightarrow **R3=1**

R4=3? No hay propagaciones \Rightarrow **R4=3**

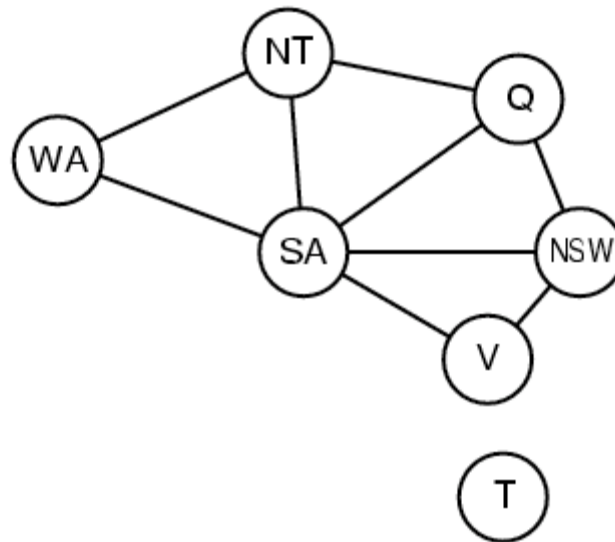
Heurísticas adicionales

- La búsqueda con vuelta atrás puede mejorarse.
- Reordenación de las variables:
 - Antes de la búsqueda (primero las mas restrictivas)
 - Durante la búsqueda
 - Variable con mas restricciones
 - Variable con menos valores
- La reordenación de variables puede reducir el tiempo de búsqueda varios ordenes de magnitud en ciertos problemas.

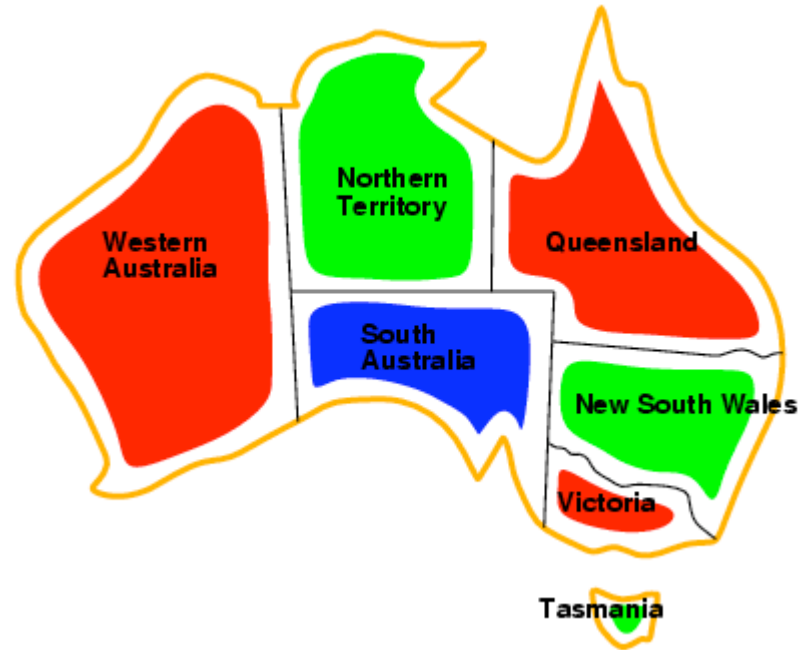
Grafos de restricciones: ejemplo



Grafos de restricciones



Grafos de restricciones



Forward checking

- **Idea:**
 - Keep track of remaining legal values for unassigned variables
 - Terminate search when any variable has no legal values



Forward checking

- **Idea:**
 - Keep track of remaining legal values for unassigned variables
 - Terminate search when any variable has no legal values



Forward checking

- **Idea:**
 - Keep track of remaining legal values for unassigned variables
 - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■
■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■
■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■	■ ■ ■



Forward checking

- **Idea:**
 - Keep track of remaining legal values for unassigned variables
 - Terminate search when any variable has no legal values

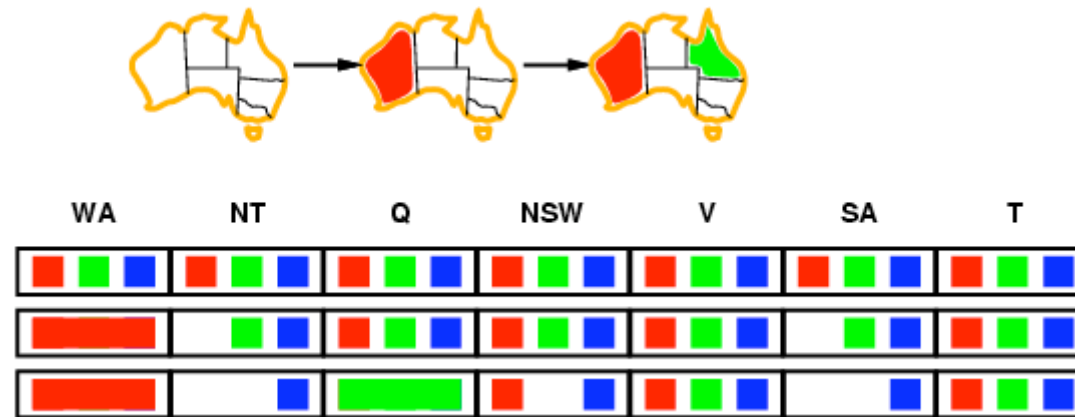


	WA	NT	Q	NSW	V	SA	T
WA	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue
NT	Red	Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Green, Blue	Red, Green, Blue
Q	Red	Blue	Green	Red, Blue	Red, Green, Blue	Blue	Red, Green, Blue
V	Red	Blue	Green	Red	Blue		Red, Green, Blue



Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

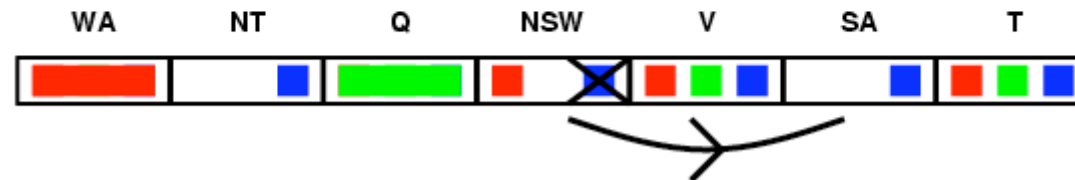
Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \ Y$ is consistent iff
 - for **every** value x of X there is **some** allowed y



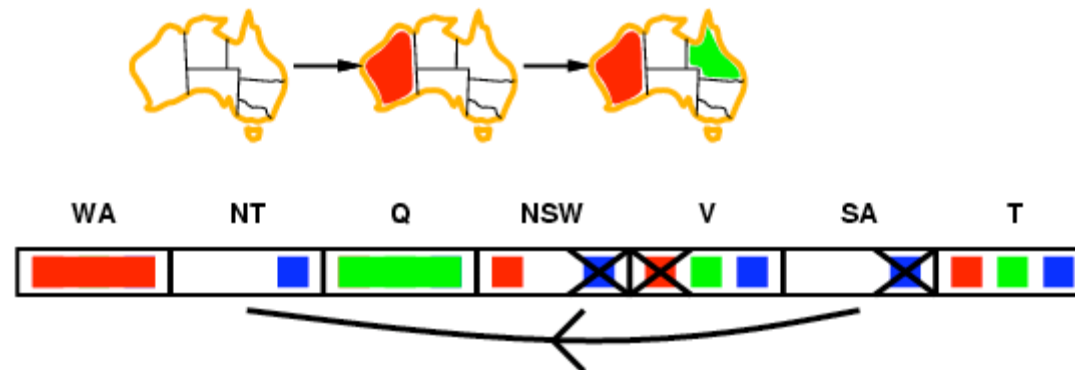
Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \ Y$ is consistent iff
 - for **every** value x of X there is **some** allowed y



Arc consistency

- Simplest form of propagation makes each arc **consistent**
- X Y is consistent iff
 - for **every** value x of X there is **some** allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment