

La résolution vue comme un problème de satisfaction de contraintes

CSP

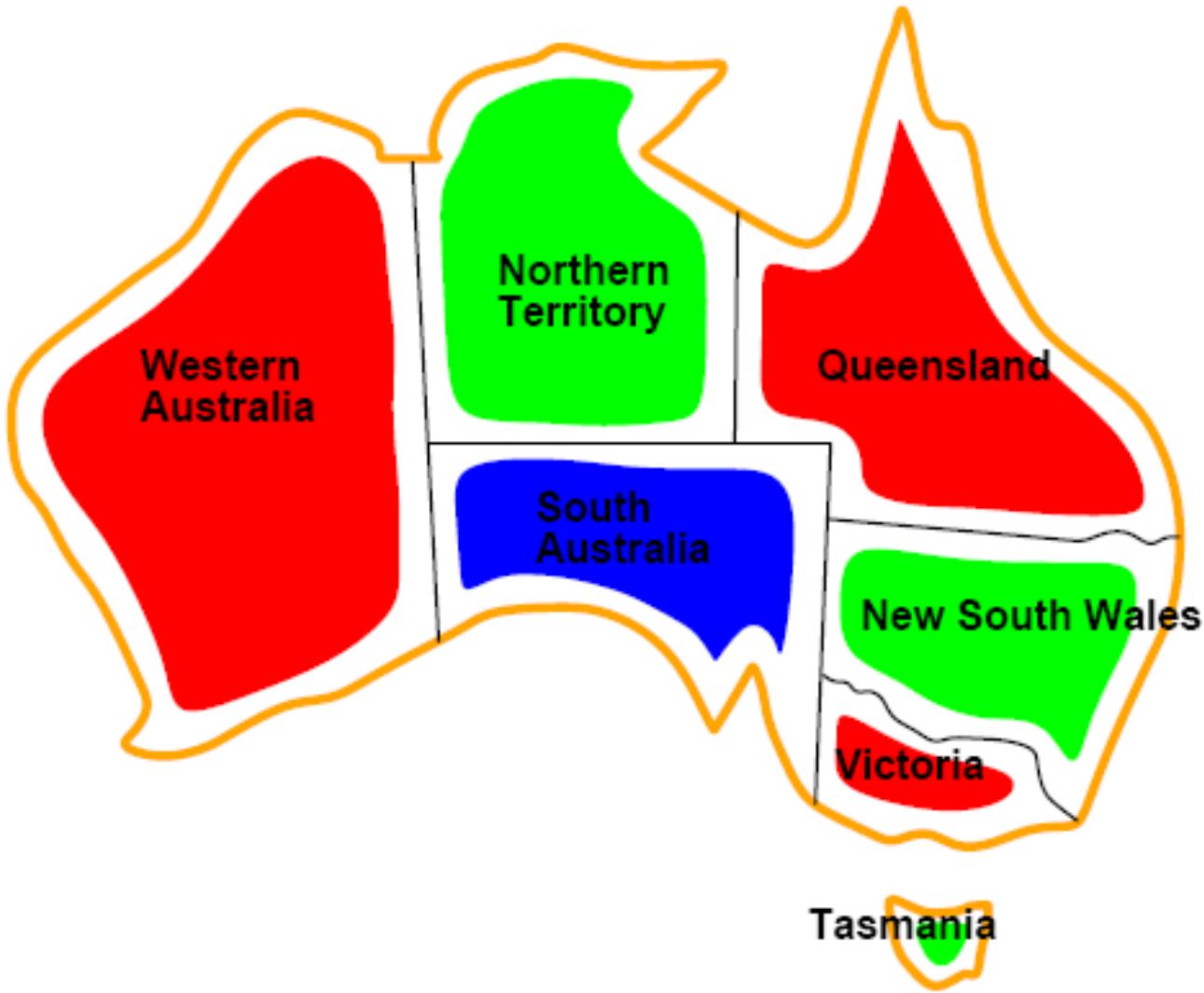
Les CSP : Constraint Satisfaction Problem

- Un CSP est un type de problème particulier
 - Les états sont définis par les **valeurs** d'un ensemble de **variables**
 - » Les variables sont caractérisées par un **domaine** de valeurs possibles associé à chaque variable
 - Les actions sont des **assignations** de valeurs aux variables et/ou des **retraits** de valeurs possibles aux domaines
 - La fonction de test de but est un ensemble de **contraintes** auquel les valeurs des variables doivent satisfaire

Exemple : problème de coloration de carte



Une solution



CSP vs. Problème Standard

- Les CSP définissent une structure particulière pour les états : **un ensemble de variables partiellement valuées**, et une structure particulière pour la fonction but : **un ensemble de contraintes**
- Les algorithmes de recherche d'une solution peuvent tirer parti de cette structuration
 - On passe d'une vision exploration d'un ensemble d'états : d'un état initial à un état but
 - À une vision **résolution pas à pas** d'un problème : de l'étape où aucune variable n'est assignée à l'étape où toutes les variables sont assignées
- Les techniques de résolution dépendent de la nature des domaines et contraintes

Réseau de contraintes

- Un réseau de contraintes est un triplet (X, D, C)
 - $X = \{x_1, x_2, \dots, x_n\}$ est un ensemble fini de variables
 - D , le domaine de X , est une fonction qui associe à chaque variable x_i un ensemble fini de valeurs $D(x_i)$ (le domaine de la variable x_i).
 - $C = \{c_1, \dots, c_m\}$ est un ensemble de contraintes
- Chaque contrainte c_i est une relation (*i.e.* un ensemble de tuples de valeurs)
 - définie sur un sous-ensemble ordonné des variables de X , appelé sa portée et notée* $P(c_i) = \langle x_{i1}, \dots, x_{ik} \rangle$ où k est l'arité.
 - c_i désigne donc l'ensemble des tuples de valeurs autorisés par la contrainte (généralement $c_i \subseteq D(x_{i1}) \times \dots \times D(x_{ik})$)

* Nous nous autoriserons à utiliser $P(c_i)$ comme un ensemble et comme une séquence (sans répétition) de variables

Attribution Partielle et Complète

- Une **Attribution** A sur un sous-ensemble X' des variables de X est une application qui associe à chaque variable x de X' une valeur de son domaine D(x)
 - si $X' \subset X$ on parle **d'Attribution partielle**
 - si $X' = X$ on parle **d'Attribution complète**
- On note :
 - **var(A)** le domaine de A (les variables ayant une image par A)
 - **A[t]** l'application de A aux variables de t (t étant un tuple de variables distinctes de var(A))

Consistance et Solution

- Une assignation A **viole une contrainte c** ssi
 1. $P(c) \subseteq \text{var}(A)$ (toutes les variables de c ont une image par A)
 2. $A[P(c)] \notin c$ (le tuple de valeur défini par A pour les variables de c n'est pas autorisé)
- A est **localement consistante** si A ne viole aucune contrainte de C
- Une **solution** est une assignation complète qui ne viole aucune contrainte de C
 - L'ensemble des solutions d'un réseau de contraintes $R=(X,D,C)$ est noté $\text{Sol}(R)$

Problème de satisfaction de contraintes

- Un CSP est le problème défini par :
 - Instance : un réseau de contraintes $R=(X,D,C)$
 - Question : $\text{Sol}(R) \neq \emptyset$ (R a-t-il une solution ?)
- CSP est NP-complet

Variables, Domaines, Contraintes

- Propriété : Tout CSP n-aire peut être ramené à un CSP binaire
 - Unaire : ?
 - $1 < a < 5 \rightarrow ?$

Variables, Domaines, Contraintes

- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
 - Unaire : restriction de domaine
 - $1 < a < 5 \rightarrow D(a) = \{2,3,4\}$

Variables, Domaines, Contraintes

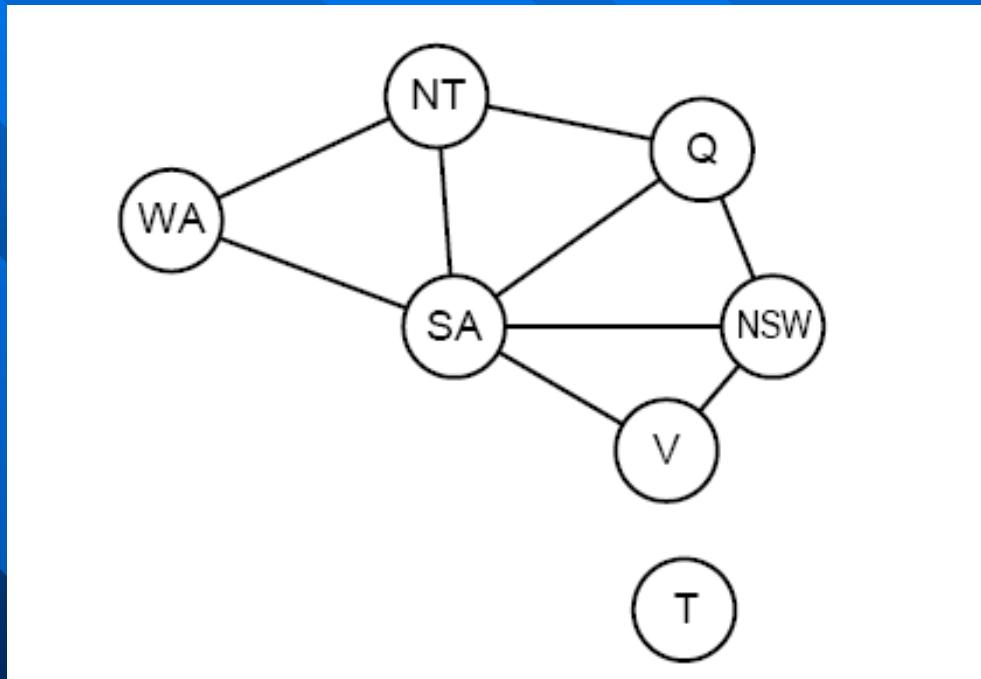
- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
 - Unaire : restriction de domaine
 - $1 < a < 5 \rightarrow D(a) = \{2,3,4\}$
 - N-aire avec $n > 2$: ?
 - $a+b=c \rightarrow ?$

Variables, Domaines, Contraintes

- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
 - Unaire : restriction de domaine
 - $1 < a < 5 \rightarrow D(a) = \{2, 3, 4\}$
 - N-aire avec $n > 2$: introduction de nouvelles variables dont les domaines sont des tuples
 - $a+b=c \rightarrow$
 - $X = \langle a, b, c, ab \rangle$
 - $D(ab) = D(a) \times D(b)$
 - $C = \{a = \text{Proj}_1(ab), b = \text{Proj}_2(ab), \text{Proj}_1(ab) + \text{Proj}_2(ab) = c\}$

Représentation graphique d'un CSP binaire

- Représentation
 - Les variables X → des sommets
 - Les contraintes C → des arêtes



- Les algorithmes de recherche peuvent alors tirer parti de la structure du graphe
 - Ex. : T est un pb indépendant, SA est très constraint

Représentation d'un CSP n-aire

- Un **multi-hypergraphe** dont les sommets sont les variables et les arêtes sont les contraintes que l'on représente par son biparti d'incidence :
 - des **sommets variables**
 - des **sommets contraintes**
 - des **arêtes** indiquant la participation d'une variable à une contrainte
- On peut étiqueter
 - les sommets variables par leur domaine
 - les sommets contraintes par leurs tuples
 - les arêtes adjacentes à une contrainte par la position de la variable dans la contrainte

Exemples :

- Les puzzles arithmétiques codés

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

- Les n-reines

- Placer **n** reines sur un échiquier **n**×**n** de telle sorte qu'aucune ne soit en prise

- Le sudoku

Résolution d'un CSP : algo naïf

- L'état initial est l'état dans lequel aucune variable n'est assignée : $A = \emptyset$
- Une action consiste à choisir une variable non assignée et à lui associer une valeur de son domaine : $A \leftarrow A \cup \{(x_i, v)\}$ où $v \in D(x_i)$
 - Le facteur de branchement est borné par la somme des tailles des domaines
 - La profondeur de l'arbre de recherche est naturellement bornée par le nombre de variables
- Lorsque toutes les variables sont assignées, la fonction de test de but vérifie si les contraintes sont satisfaites ou pas
 - A est-il une solution de R ?

On peut donc utiliser un algorithme de recherche en profondeur $\rightarrow O((\sum_i |D_i|)^{|X|})$

Propriétés de l'espace de recherche

- L'ordre d'assignation des variables n'affecte pas la solution
 - Quelque soit le chemin emprunté, une même assignation des variables définit un seul état
 - Donc inutile de tester plusieurs chemins
- Lors du déroulement de la recherche en profondeur on choisit **un ordre d'assignation** des variables
 - La complexité de la profondeur devient $O(\Pi_i |D_i|)$
 - L'arbre de recherche dépendra de l'ordre choisi

Propriétés de l'espace de recherche

- Certaines branches de l'arbre de recherche peuvent être coupées en tirant parti du fait que le test de but est un ensemble de contraintes
 - Si une assignation partielle viole une contrainte, il est inutile d'étendre cette assignation car elle ne conduira qu'à des « échecs ».
 - Exemple :
 - » $A = \{(WA, Red), (NT, Red)\}$ ne pourra jamais être prolongée en un A' satisfaisant car elle viole la contrainte $c = (WA \neq NT)$

Résolution d'un CSP : Backtracking algorithm

- L'algorithme prend en entrée
 - Un réseau de contraintes
 - Une **assignation partielle** localement consistante
- On teste à chaque extension de l'assignation partielle courante si elle ne viole pas de contraintes
 - Si viol, on « **backtrack** » immédiatement : on revient sur les choix précédents de valeur
 - Si non, on continue à développer la branche

Backtracking algorithm

[Golomb and Baumert, 1965]

BacktrackingSearch(Network R=(X,D,C)) = BT({}, R);

Fonction BT(Assignation A, Network R) : Booléen

Début

si |A| = |R.X| alors

afficher a;

retourner true;

finsi;

 x \leftarrow ChoixVariableNonAssignée(R.X, A);

pour tout v \in Tri(R.D(x)) faire

si Consistant(A \cup {(x,v)}, R.C) alors

si BT(A \cup {(x,v)}, R) alors retourner true; finsi;

finsi;

finpour;

retourner false;

Fin

Exemple



Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

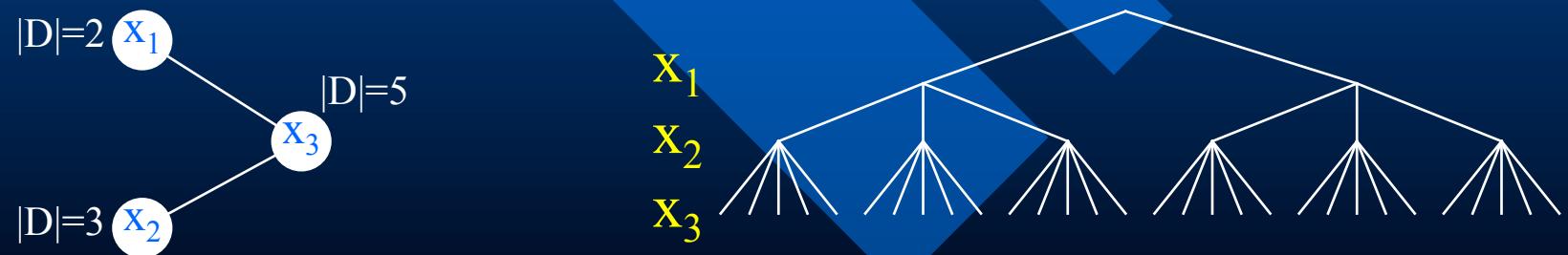
Améliorations du Backtrack

■ Les pistes générales

- Comment choisir l'ordre des variables
 - » Fonction ChoixVariableNonAssignée
- Comment choisir l'ordre des valeurs
 - » Fonction Tri
- Comment détecter les incohérences au plus tôt
 - » Fonction Consistant
- Etude des propriétés structurelles des réseaux
 - » Techniques de Décomposition en : composantes connexes, structure arborescente...

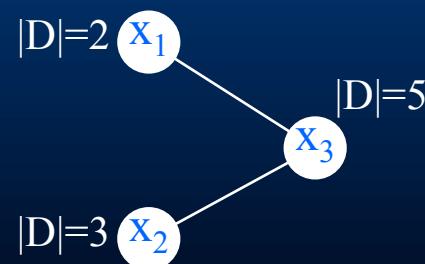
Heuristiques d'ordonnancement des variables

- L'idée est de calculer avant l'exécution du BT un ordre « pertinent » d'assignation des variables
 - Ces heuristiques sont qualifiées de **statique** car l'ordre n'est pas remis en cause au cours du BT.
 - **lex** : variables considérées dans l'ordre lexicographique
 - **rand** : variables considérées dans un ordre aléatoire
- L'ordre d'assignation des variables déterminé par l'heuristique modifie l'**arbre de recherche développé**.
 - L'objectif est de limiter le nombre de nœuds explorés (i.e. la taille de l'arbre de recherche développé)



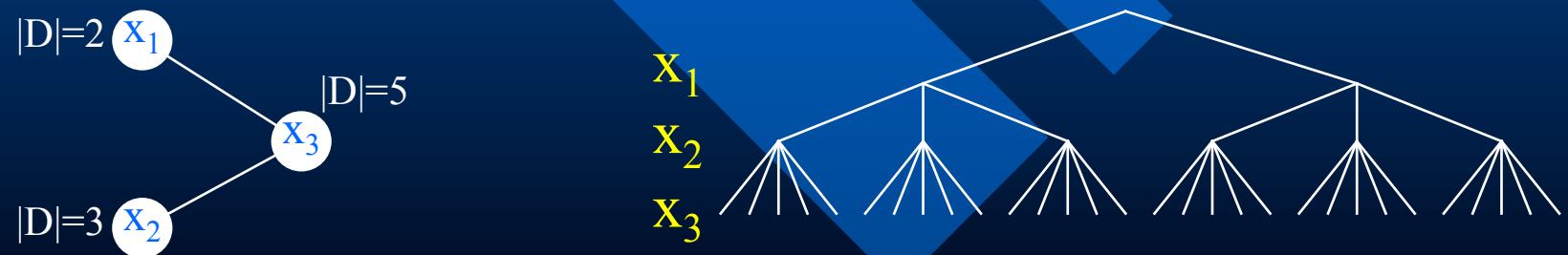
Heuristiques d'ordonnancement des variables

- L'idée est de calculer avant l'exécution du BT un ordre « pertinent » d'assignation des variables
 - Ces heuristiques sont qualifiées de **statique** car l'ordre n'est pas remis en cause au cours du BT.
 - **lex** : variables considérées dans l'ordre lexicographique
 - **rand** : variables considérées dans un ordre aléatoire
- L'ordre d'assignation des variables déterminé par l'heuristique modifie l'**arbre de recherche développé**.
 - L'objectif est de limiter le nombre de nœuds explorés (i.e. la taille de l'arbre de recherche développé)



Heuristiques statiques

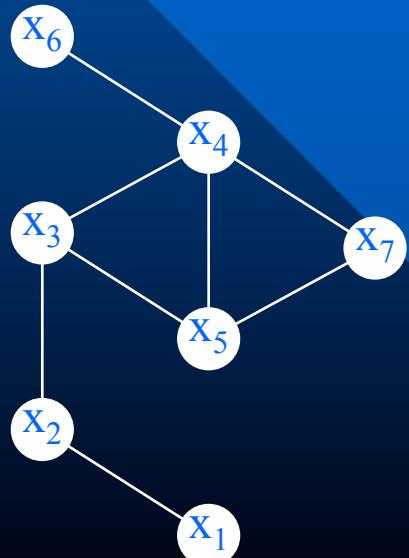
- 1^{ière} idée : choisir la variable ayant le plus petit domaine (heuristique **min-dom**)
 - L'idée étant de minimiser le degré de branchement (plus de chances d'avoir un échec rapidement)
 - Problème : bien souvent les variables d'un problème ont toutes la même cardinalité.
- 2^{ième} idée : exploiter la structure du graphe de contrainte



Heuristique statique : min-width

[Freuder, 1982]

- Ordonne les variables de la dernière à la première en sélectionnant à chaque étape une variable de degré (i.e. nombre de contraintes portant sur la variable) min dans le (sous)graphe de contrainte réduit aux variables non encore sélectionnées (*et à leurs contraintes incidentes*)



x₇
x₅ x₇
x₄ x₅ x₇
x₃ x₇
x₆
x₂ x₆
x₁ x₆

Heuristique statique : max-deg

[Dechter & Meiri, 1989]

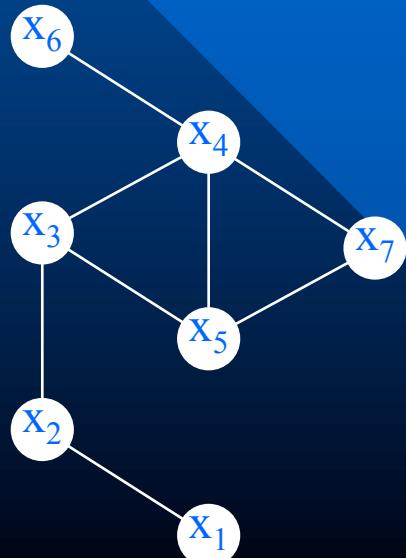
- Ordonne dans l'ordre décroissant des degrés (*i.e.* *nombre de contraintes portant sur la variable*)



Heuristique statique : max-card

[Dechter & Meiri, 1989]

- Ordonne en sélectionnant une variable qui est connectée au plus grand nombre de variables déjà sélectionnées (i.e. plus grand nombre de variables déjà sélectionnées qui partagent au moins une contrainte avec la variable)



x₁ x₂ x₃ x₄ x₅ x₆ x₇
x₂
x₃
x₄ x₅
x₅
x₇
x₆

Heuristiques statiques

- On peut combiner les heuristiques
 - Exemple : max-card combinée à max-degree quand égalité pour max-card



Ordonnancement des valeurs (définie pour des contraintes binaires)

- L'idée est de choisir une valeur qui a plus de chances de conduire à une solution
 - Une valeur qui laisse le plus de couples autorisés restants dans les contraintes où participe la variable
 - » Par exemple : Soit $C(x)$ l'ensemble des contraintes contenant x , on choisit pour x la valeur y dont $\prod_{c \in C(x)} |c_{x=y}|$ est maximal
 - » Exemple : $D(x)=\{a,b,c,d,e\}$
 - Pour a $\rightarrow 1*3 = 3$
 - Pour b $\rightarrow 2*2 = 4$
 - Pour c $\rightarrow 2*1 = 2$
 - Pour d $\rightarrow 0*1 = 0$
 - Pour e $\rightarrow 0*0 = 0$

Ordre : **b,a,c** et on peut voir qu'il est inutile de tester *c* et *d*

x	z
a	o
a	n
b	p
c	o
b	n
a	p
c	1
b	3
a	1
c	2
b	1
d	o

x	y
a	o
a	n
b	p
c	1
b	3
a	1
c	2
b	1

Ordonnancement des valeurs (définie pour des contraintes binaires)

- L'idée est de choisir une valeur qui a plus de chances de conduire à une solution
 - Une valeur qui laisse le plus de couples autorisés restant dans les contraintes où participe la variable
 - » Par exemple : Soit $C(x)$ l'ensemble des contraintes contenant x , on choisit pour x la valeur v dont $\prod_{c \in C(x)} (|c_{x=v}|)$ est maximal
- L'ordre d'affectation des valeurs n'influe pas sur la structure de l'arbre
 - Il permute seulement l'ordre des branches
- Le choix d'un ordre ne présente aucun intérêt si on cherche toutes les solutions.

Détection des incohérences

- L'enjeu est de détecter des incohérences dans les valeurs des variables au plus tôt pour éviter des assignations inutiles
- L'idée est de supprimer des valeurs impossibles des domaines des variables afin de diminuer l'espace de recherche
- Cette suppression peut-être envisagée :
 - en **amont du BT** pour réduire les domaines à considérer
 - après chaque assignation pour réduire les domaines des variables restant à assigner
 - » on parle de **domaines dynamiques** : *ils évoluent au cours du BT*

Détection des incohérences

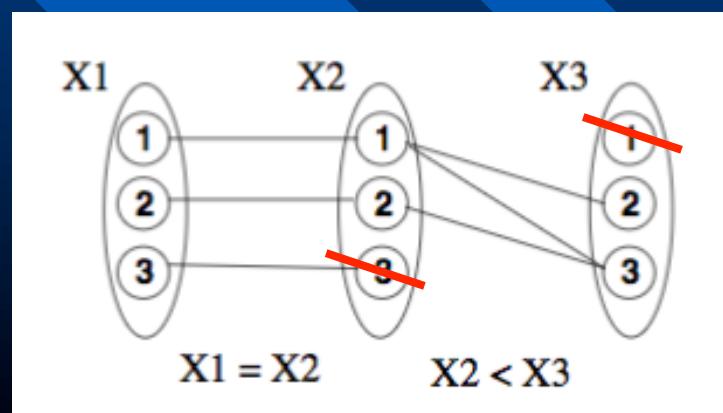
- Définition de propriétés de consistance que les domaines des variables doivent respecter
 - » Ex : une valeur du domaine d'une variable doit avoir au moins une occurrence dans chacune des contraintes où elle apparaît
 - A partir de telles propriétés, on cherche à calculer une réduction des domaines qui vérifie cette propriété et garantit que les solutions du CSP sont conservées
 - » Ex : soit $C(x)$ l'ensemble des contraintes contenant x :
$$\text{Dréduit}(x) = D(x) \cap \bigcap_{c \in C(x)} (c|_x)$$
- Définition opérationnelle de règles de réduction de domaine attachées aux contraintes
 - » Ex : on peut associer à la contrainte $x \neq y$ la règle :
***si* v est assigné à x alors $D(y) \leftarrow D(y) - \{v\}$**

Arc-Consistance

■ Une propriété plus forte de consistance :

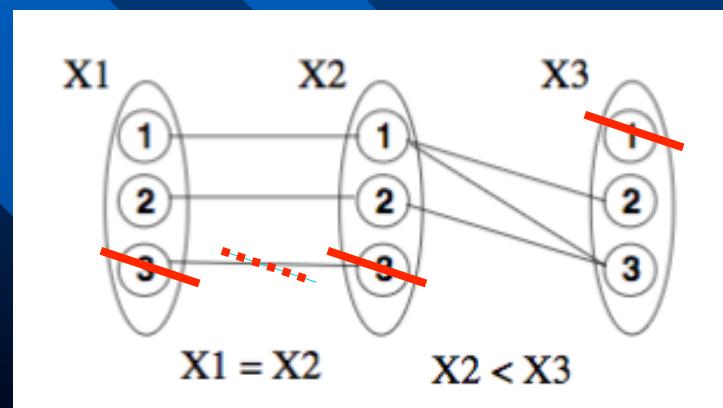
- Un réseau (X, D, C) est dit arc-consistant sur D si toutes les contraintes de C sont arc-consistantes sur D
- Une contrainte c est arc-consistante sur D si toutes les valeurs de toutes les variables de $P(c)$ ont un tuple support dans c
- Un tuple support d'une valeur v pour une variable x d'une contrainte c est un tuple t tel que $t|_x = v$ et t valide
 - » Un tuple d'une contrainte est dit valide s'il appartient au sous-ensemble du produit cartésien des domaines des variables de la contrainte

■ Exemple : Non Arc-Consistant



Arc-Consistance

- On restaure l'arc-consistance en supprimant :
 - des domaines les valeurs sans support
 - des contraintes les tuples qui n'appartiennent plus au produit cartésien des domaines des variables de la contrainte
 - » cette seconde suppression n'est pas réalisée en pratique !
 - La fermeture arc-consistance d'un réseau (X, D, C) consiste à calculer le plus grand domaine $D_{AC} \subseteq D$ tel que (X, D_{AC}, C) est arc-consistant
 - Cette fermeture est unique
 - Exemple :
- Arc-Consistant



AC3 : un algo pour tester/restaurer l'arc-consistance [Mackworth, 1977]

- Une fonction *Révise* qui filtre le domaine d'une variable x relativement à une contrainte c en vérifiant que chaque valeur v du domaine de cette variable possède un tuple t support dans la contrainte
 - t appartient au produit cartésien des domaines des variables de c
 - t contient pour x la valeur v
- La fonction *AC* maintient un ensemble Q de couples var/cont (x,c) tels que $x \in P(c)$ qu'elle considère un à un :
 - *Révise* est appelée
 - si le $D(x)$ est modifié alors l'ensemble des couples (x',c') tels que c' porte sur x et x' (avec $c' \neq c$ et $x' \neq x$) est ajouté à Q pour être (re)considérer.

AC3 : un algo pour tester/restaurer l'arc-consistance [Mackworth, 1977]

Fonction Revise (Variable x, Constraint c, Network R) : Booléen

Début

modif \leftarrow false ;

pour tout v \in R.D(x) faire

si il n'existe pas t \in c tel que $t|_x = v$ et t valide* pour R.D alors

 supprimer v de R.D(x) ;

 modif \leftarrow true ;

finsi;

finpour;

retourner modif ;

Fin;

* valide : chaque valeur du tuple est dans le domaine courant de la variable correspondante

AC3 : un algo pour tester/restaurer l'arc-consistance

[Mackworth, 1977]

Fonction AC3 (Network R) : Booléen

Début

```
Q ← {(x,c) | c ∈ R.C et x ∈ P(c)};  
tant que Q ≠ {} faire  
    retirer (x,c) de Q;  
    si Revise(x,c,R) alors  
        si R.D(x)={} alors retourner false ;  
        sinon  
            Q ← Q ∪ {(x',c') | c' ∈ R.C, x ∈ P(c'), x' ∈ P(c'), c'≠c, x'≠x};  
        finsi;  
    finsi;  
fintantque;  
retourner true;
```

Fin;

Les méthodes prospectives

- Les choix faits dans les assignations précédentes peuvent produire une assignation partielle
 - localement consistante
 - mais ne permettant pas d'obtenir une solution
 - dans ce cas BT testera tous les choix de complétiions avant de revenir sur l'assignation précédente

 - Les méthodes prospectives mettent en œuvre une étape de **propagation de contraintes** à chaque nœud de l'arbre de recherche
 - Motivation : découvrir des inconsistances plus rapidement
 - Principe : éliminer des valeurs des domaines
- Domaines dynamiques



Le Forward Checking

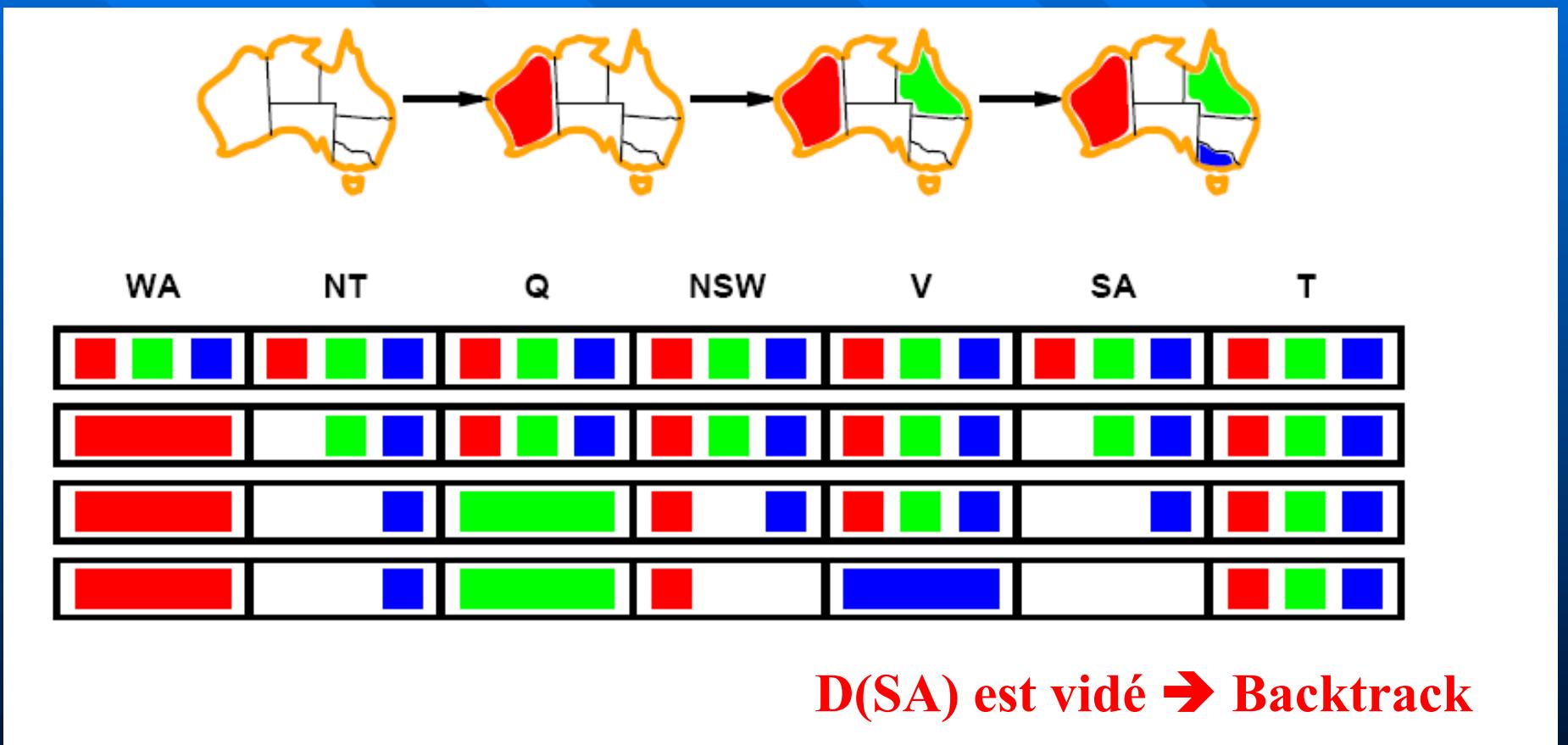
binaire [Haralick & Elliott, 1980]

n-aire [Van Hentenryck, 1989]

- L'idée consiste lors de l'assignation d'une variable **x** à supprimer des domaines des variables non encore assignées les valeurs qui violent une contrainte
 - On supprime des domaines des variables **y** non assignées et reliées à **x** par une contrainte **c** - *dont toutes les variables sauf y sont assignées* - les valeurs qui engendrent un viol de **c**
 - Si un domaine devient vide alors « backtracking » immédiat et restauration des domaines

Remarque : il n'est alors plus utile de vérifier la consistance d'une assignation

Le forward checking par l'exemple



Forward Checking

Fonction FC(Assignation A, Network R) : Booléen

Début

si |A| = |R.X| alors

afficher a;

retourner true;

finsi;

 x \leftarrow ChoixVariableNonAssignée(R.X, A);

 D_{old} \leftarrow D ;

pour tout v \in Tri(R.D(x)) faire

si Propage(x, v, A) alors

si FC(A \cup {(x,v)}, R) alors retourner true;

finsi;

 D \leftarrow D_{old} ;

finpour;

retourner false;

Fin

Forward Checking

Fonction Propage(Variable x, Valeur v, Assignation A) : Booléen

Accès en modification : Network R

Début

```
pour tout c ∈ R.C tel que x ∈ P(c) et |P(c)-(var(A) ∪ {x})|=1 faire
    pour tout w ∈ R.D(y) où y est l'unique variable de P(c)-(var(A) ∪ {x}) faire
        si il n'existe pas t ∈ c tel que t|x=v et t|y=w alors
            supprimer w de R.D(y) ;
        finsi ;
    finpour;
    si R.D(y)=∅ alors retourner false ;
    finpour ;
    retourner true ;
Fin ;
```

Application

WA

NT

NSW

Q

V

SA

T

R G B

WA	NT	Q	NSW	V	SA	T
RGB						

NT	SA
GB	GB

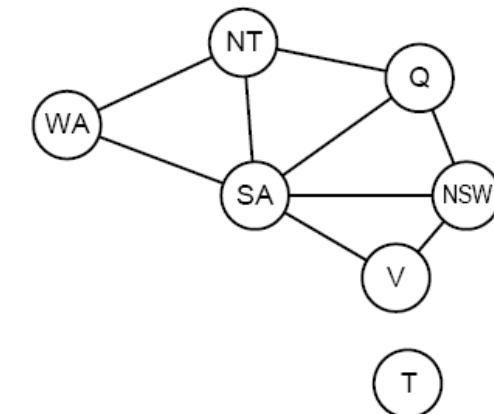
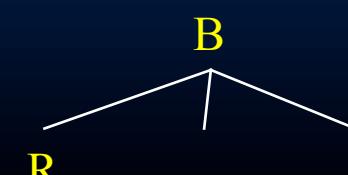
Q	SA
RB	B

Q	V	SA
B	GB	B

Q	V	SA
RB	RB	B

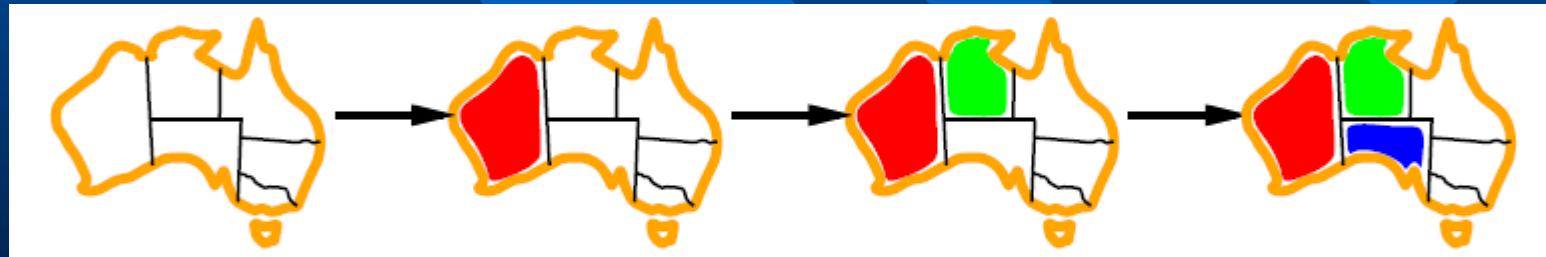
SA
Backtrack

SA
B



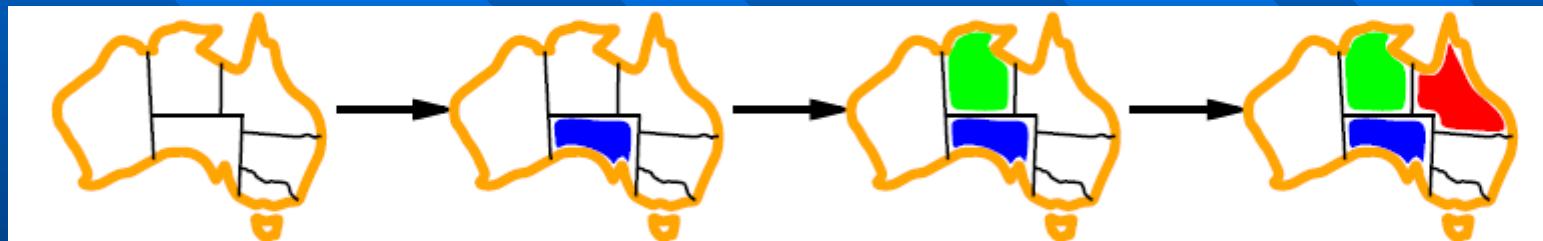
Les idées d'heuristiques pour les domaines dynamiques

- **dom** [Haralick & Elliott, 1980] : choix de la variable qui a le plus petit nombre de valeurs restantes dans son **domaine courant**



Les idées d'heuristiques pour les domaines dynamiques

- dom+deg : dom est en cas d'égalité deg (plus grand nombre de contraintes portant sur elle)



- dom/deg : choix de la variable qui a le plus petit ratio taille du domaine courant sur nombre de contraintes portant sur elle
- Variantes avec
 - ddeg : (dynamic) degré en enlevant du graphe les variables assignées
 - wdeg : (weighted) degré pondéré par le poids des contraintes ou le poids est dynamiquement calculé en comptant le nombre de viols provoqué par la contrainte