

# Intelligence artificielle

## Satisfaction de contraintes

---

Elise Bonzon

`elise.bonzon@u-paris.fr`

LIPADE - Université de Paris

<http://www.math-info.univ-paris5.fr/~bonzon/>

1. Exemples de CSP
2. Recherche en arrière pour les CSPs (backtracking search)
3. Structure des problèmes
4. Conclusion

## Exemples de CSP

---

# Problèmes de satisfaction de contraintes (CSP)

- Problèmes de recherche “classiques” :
  - Un **état** est une “boite noire”
  - N'importe quelle structure de données qui contient un test pour le but, une fonction d'évaluation, une fonction successeur
- **CSP** :
  - Un **état** est défini par un ensemble de **variables**  $X_i$ , dont les **valeurs** appartiennent au **domaine**  $D_i$
  - Le **test pour le but** est un ensemble de **contraintes** qui spécifient les combinaisons autorisées pour les valeurs sur des sous-ensembles de variables
- Exemple simple d'un **langage formel de représentation**
- Permet d'utiliser des algorithmes généraux plus efficaces que les algorithmes de recherche standards

## Exemple : coloriage de carte



- **Variables** :  $WA, NT, SA, Q, NSW, V, T$
- **Domaines** :  $D_i = \{\text{rouge}, \text{vert}, \text{bleu}\}$
- **Contraintes** : régions adjacentes de couleurs différentes

## Exemple : coloriage de carte



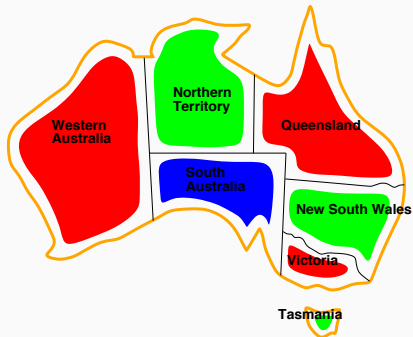
- **Variables** :  $WA, NT, SA, Q, NSW, V, T$
- **Domaines** :  $D_i = \{\text{rouge, vert, bleu}\}$
- **Contraintes** : régions adjacentes de couleurs différentes
  - Par exemple,  $WA \neq NT$  (si le langage le permet)

## Exemple : coloriage de carte



- **Variables** :  $WA, NT, SA, Q, NSW, V, T$
- **Domaines** :  $D_i = \{\text{rouge}, \text{vert}, \text{bleu}\}$
- **Contraintes** : régions adjacentes de couleurs différentes
  - Par exemple,  $WA \neq NT$  (si le langage le permet)
  - Ou  $(WA, NT) \in \{(\text{rouge}, \text{vert}), (\text{rouge}, \text{bleu}), (\text{vert}, \text{rouge}), \dots\}$

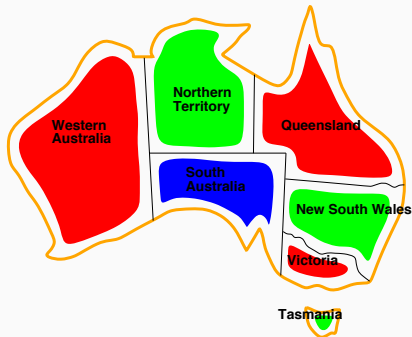
## Exemple : coloriage de carte



- Les **solutions** sont des affectations qui satisfont toutes les contraintes



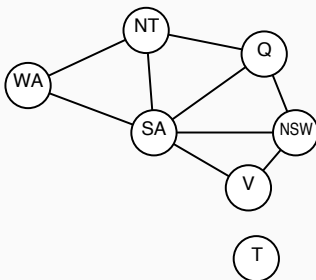
## Exemple : coloriage de carte



- Les **solutions** sont des affectations qui satisfont toutes les contraintes
- Par exemple,  $\{WA = \text{rouge}, NT = \text{vert}, Q = \text{rouge}, NSW = \text{vert}, V = \text{rouge}, SA = \text{bleu}, T = \text{vert}\}$

# Graphe de contraintes

- **CSP binaires** : chaque contrainte lie au maximum deux variables
- **Graphe de contraintes** : les nœuds sont des variables, les arcs représentent les contraintes



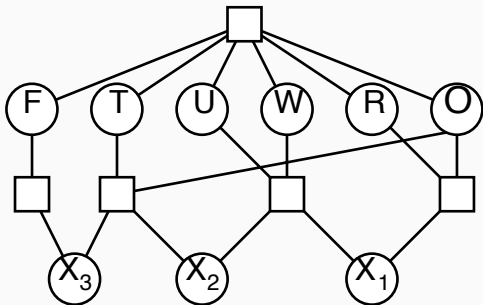
- Les algorithmes CSP utilisent les graphes de contraintes
- Permet d'accélérer la recherche : par exemple, colorier la Tasmanie est un sous-problème indépendant

- Variables **discrètes**
  - **Domaines finis** : si de taille  $d$ , il y a  $O(d^n)$  affectations complètes
    - Par exemple, CSPs booléens
  - **Domaines infinis** (entiers, caractères...)
    - Par exemple, mise en place d'un planning, avec date de début/de fin pour chaque tâche
    - Nécessite un **langage de contraintes**. Eg  $StartJob_1 + 5 \leq StartJob_5$
    - Si les contraintes sont **linéaires**, le problème est soluble
    - Si les contraintes sont **non linéaires**, problème indécidable
- Variable **continues**
  - Par exemple, temps de début/fin pour les observations du télescope de Hubble
  - Contraintes linéaires solubles en temps polynomial en utilisant des méthodes de programmation linéaire

- **Contraintes unaires**, ne concernent qu'une seule variable
    - Par exemple,  $SA \neq vert$
  - **Contraintes binaires**, concernent une paire de variables
    - Par exemple,  $SA \neq WA$
  - **Contraintes d'ordre plus élevé**, concernent 3 variables ou plus
    - Par exemple, contraintes sur les puzzles cryptarithmiques
  - **Préférences** (ou contraintes souples)
    - Par exemple, *rouge* est mieux que *vert*
    - Souvent représentable par un coût associé à chaque affectation de variable
- ⇒ Problèmes d'optimisation de variables

## Exemple : puzzle cryptarithmétique

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



- **Variables** :  $F, T, U, W, R, O, X_1, X_2, X_3$
- **Domaines** :  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- **Contraintes** :
  - $\text{Alldiff}(F, T, U, W, R, O)$
  - $O + O = R + 10X_1$
  - $X_1 + W + W = U + 10X_2$
  - ...

# Problèmes CSPs du monde réel

- Problèmes d'affectation (eg. *qui enseigne quel cours ?*)
- Problèmes d'emploi du temps
- Configuration de matériels
- Planification pour les transports
- Planification dans les usines
- Allocation de salles
- ...
- **Note** : beaucoup de problèmes du monde réel impliquent des variables à valeurs réelles

# Formulation de la recherche standard (recherche incrémentale)

- Les états sont définis par les valeurs des variables déjà affectées
- **Etat initial** : un ensemble d'affectations vides  $\{\}$
- **Fonction successeur** : attribuer une valeur à une variable non encore affectée, de façon cohérente (vis à vis des contraintes) à l'affectation actuelle
- **Test du but** : toutes les variables sont affectées

# Formulation de la recherche standard (recherche incrémentale)

- Cet algorithme de recherche marche pour tous les CSPs
- Chaque solution apparait à une profondeur de  $n$  s'il y a  $n$  variables
  - Utiliser la recherche en profondeur d'abord
- $n$  : nombre de variables ;  $d$  : taille du domaine des variables ;  $b$  : facteur de branchement
- $b = (n - p)d$  à profondeur  $p$ 
  - $n!d^n$  feuilles
  - alors qu'il n'y a que  $d^n$  affectations possible !!



## Recherche en arrière pour les CSPs (backtracking search)

---

# Backtracking search

- L'affectation des variables est **commutative**
  - L'ordre dans lequel on affecte les variables n'a pas d'importance
  - $WA = \text{rouge}$  puis  $NT = \text{vert}$  est la même chose que  $NT = \text{vert}$  puis  $WA = \text{rouge}$
- Il n'y a donc besoin de ne considérer **qu'une seule variable** par profondeur de l'arbre de recherche
  - $b = d$ , et donc  $d^n$  feuilles
- Recherche en profondeur d'abord avec l'affectation d'une variable à la fois est appelée **recherche par retour arrière** (**backtracking search**)
- Algorithme de recherche basique pour les CSPs
- Permet de résoudre le problème des  $n$  reines pour  $n \sim 25$

# Algorithme de recherche par retour arrière

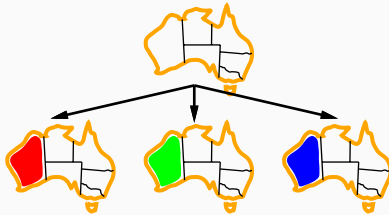
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

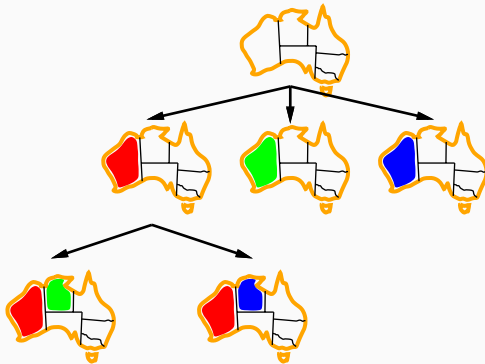
# Exemple



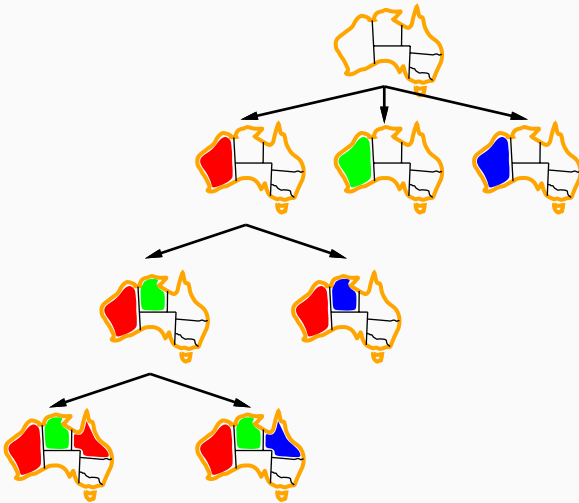
# Exemple



# Exemple



# Exemple



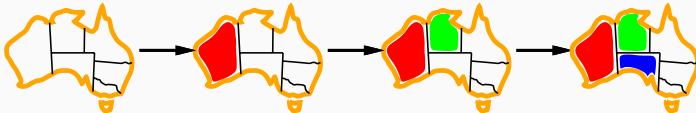
1. Comment choisir la variable à affecter ensuite ?  
(Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables ?  
(Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt ?
4. Comment tirer avantage de la structure du problème ?



1. Comment choisir la variable à affecter ensuite ?  
(Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables ?  
(Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt ?
4. Comment tirer avantage de la structure du problème ?

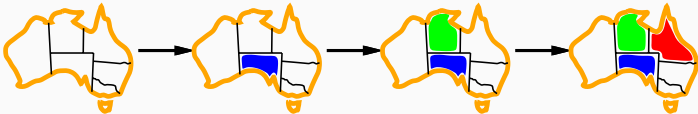
# Valeurs minimum restantes (MRV)

- Heuristique des **valeurs minimum restantes (MRV)**  
⇒ choisir une des variables ayant le moins de valeur “légale” possible



# Heuristique du degré

- Si plusieurs variables ne peuvent pas être départagées par l'heuristique MRV
- **Heuristique du degré**
  - ⇒ choisir la variable qui a le plus de contraintes à respecter parmi les variables restantes

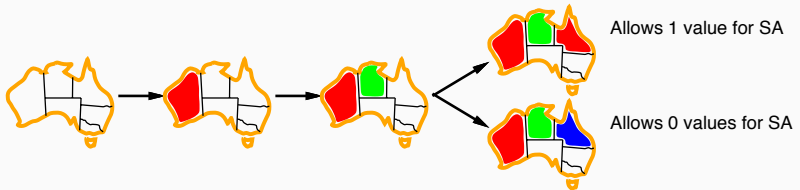


# Améliorer l'efficacité de la recherche par backtrack

1. Comment choisir la variable à affecter ensuite ?  
(Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables ?  
(Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt ?
4. Comment tirer avantage de la structure du problème ?

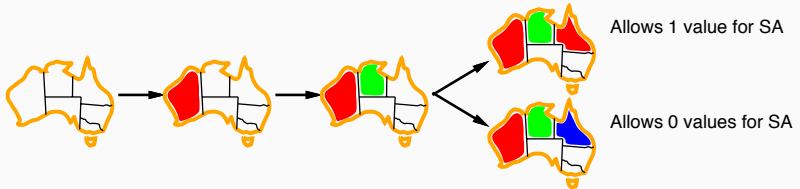
# Valeur la moins contraignante

- Etant donné une variable, choisir celle qui a la valeur la moins contraignante
  - ⇒ la variable qui empêche le moins d'affectations possibles sur les variables restantes



# Valeur la moins contraignante

- Etant donné une variable, choisir celle qui a la valeur la moins contraignante
  - ⇒ la variable qui empêche le moins d'affectations possibles sur les variables restantes



- Combiner ces heuristiques permet de résoudre le problème des  $n$  reines, avec  $n = 1000$

1. Comment choisir la variable à affecter ensuite ?  
(Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables ?  
(Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt ?
4. Comment tirer avantage de la structure du problème ?

# Vérification en avant

- **Idée** : garder en mémoire les valeurs autorisée pour les variables qu'il reste à affecter
- Arrête la recherche lorsqu'une variable n'a plus de valeur "légal" possible



WA

NT

Q

NSW

V

SA

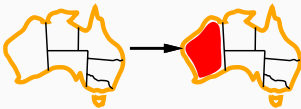
T





# Vérification en avant

- **Idée** : garder en mémoire les valeurs autorisée pour les variables qu'il reste à affecter
- Arrête la recherche lorsqu'une variable n'a plus de valeur "légal" possible



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Vérification en avant

- **Idée** : garder en mémoire les valeurs autorisée pour les variables qu'il reste à affecter
- Arrête la recherche lorsqu'une variable n'a plus de valeur "légal" possible



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Vérification en avant

- **Idée** : garder en mémoire les valeurs autorisée pour les variables qu'il reste à affecter
- Arrête la recherche lorsqu'une variable n'a plus de valeur "légal" possible



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Propagation de contraintes

- La vérification en avant permet de propager l'information des variables affectées aux variables non affectées, mais ne permet pas de détecter tous les échecs

# Propagation de contraintes

- La vérification en avant permet de propager l'information des variables affectées aux variables non affectées, mais ne permet pas de détecter tous les échecs



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Propagation de contraintes

- La vérification en avant permet de propager l'information des variables affectées aux variables non affectées, mais ne permet pas de détecter tous les échecs



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

- NT* et *SA* ne peuvent pas être tous les deux bleus !
- La **propagation de contraintes** permet de vérifier les contraintes localement

# Consistence des arcs

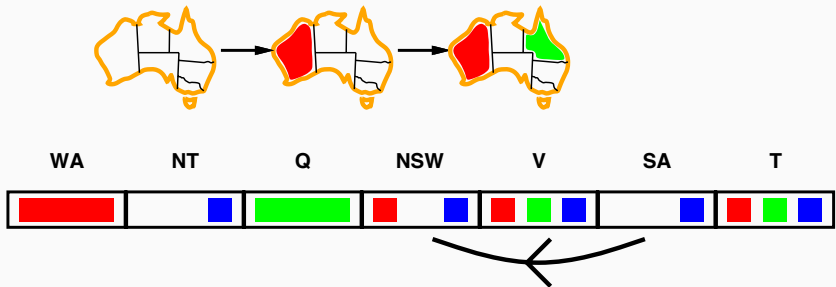
- La forme la plus simple de propagation est de rendre les arcs **consistents**
- $X \rightarrow Y$  est consistant ssi pour **toute** valeur  $x$  de  $X$ , il y a **au moins un**  $y$  autorisé



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Consistence des arcs

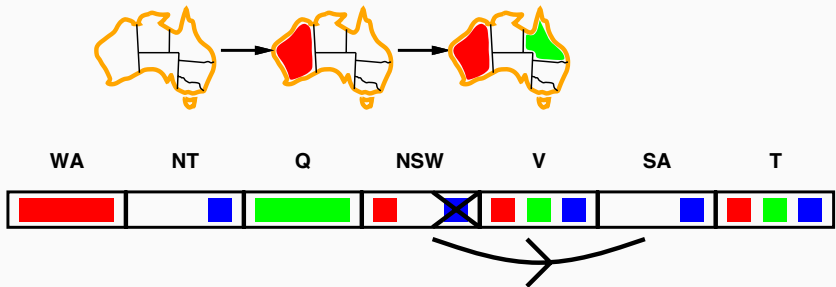
- La forme la plus simple de propagation est de rendre les arcs **consistents**
- $X \rightarrow Y$  est consistant ssi pour **toute** valeur  $x$  de  $X$ , il y a **au moins un**  $y$  autorisé





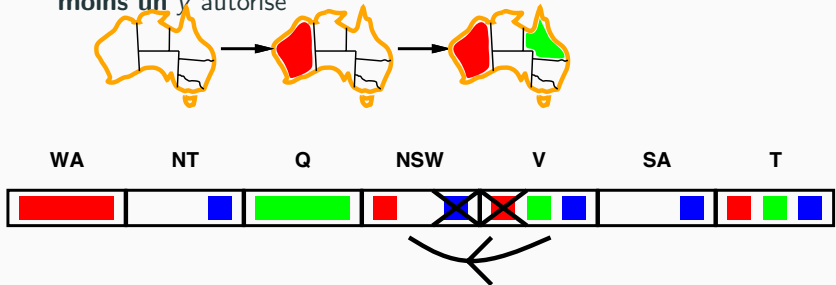
# Consistence des arcs

- La forme la plus simple de propagation est de rendre les arcs **consistents**
- $X \rightarrow Y$  est consistant ssi pour **toute** valeur  $x$  de  $X$ , il y a **au moins un**  $y$  autorisé



# Consistence des arcs

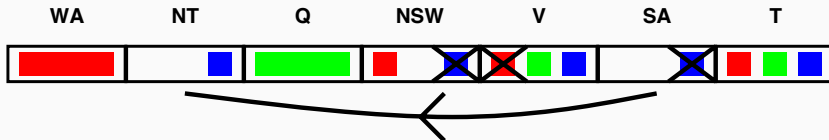
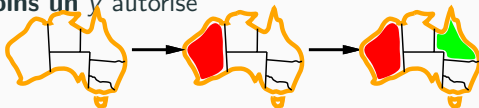
- La forme la plus simple de propagation est de rendre les arcs **consistents**
- $X \rightarrow Y$  est consistant ssi pour **toute** valeur  $x$  de  $X$ , il y a **au moins un**  $y$  autorisé



- Si  $X$  perd une valeur, les voisins de  $X$  doivent être revérifiés

# Consistence des arcs

- La forme la plus simple de propagation est de rendre les arcs **consistents**
- $X \rightarrow Y$  est consistant ssi pour **toute** valeur  $x$  de  $X$ , il y a **au moins un**  $y$  autorisé



- Si  $X$  perd une valeur, les voisins de  $X$  doivent être revérifiés
- Repère un échec avant la vérification en avant
- Peut être lancé comme un pré-processeur ou après chaque affectation

# Algorithme de vérification de consistance d'arcs

**function AC-3**(*cs*) **returns** the CSP, possibly with reduced domains

**inputs:** *cs*, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$

**local variables:** *queue*, a queue of arcs, initially all the arcs in *cs*

**while** *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$

**if** REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) **then**

**for each**  $X_k$  **in** NEIGHBORS[ $X_i$ ] **do**

            add  $(X_k, X_i)$  to *queue*

---

**function REMOVE-INCONSISTENT-VALUES**( $X_i, X_j$ ) **returns** true iff succeeds

*removed*  $\leftarrow$  false

**for each**  $x$  **in** DOMAIN[ $X_i$ ] **do**

**if** no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$

**then** delete  $x$  from DOMAIN[ $X_i$ ]; *removed*  $\leftarrow$  true

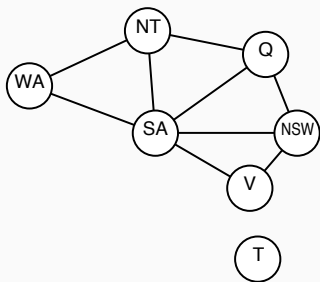
**return** *removed*

1. Comment choisir la variable à affecter ensuite ?  
(Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables ?  
(Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt ?
4. Comment tirer avantage de la structure du problème ?

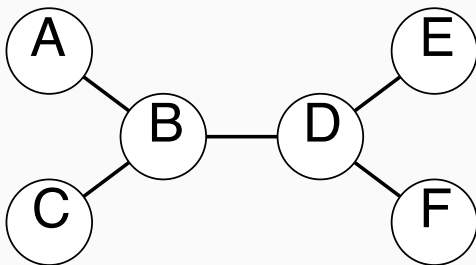
# Structure des problèmes

---

# Structure des problèmes



- La Tasmanie est un **sous-problème indépendant**
- Identifiables comme étant des **composants connexes** du graphe de contraintes



### Théorème

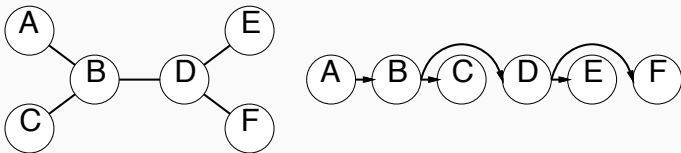
Si le graphe de contraintes ne contient pas de cycles, le CSP a une complexité en temps de  $O(nd^2)$

Cas général : complexité en temps de  $O(d^n)$



# Algorithme pour les CSPs structurés sous forme d'arbre

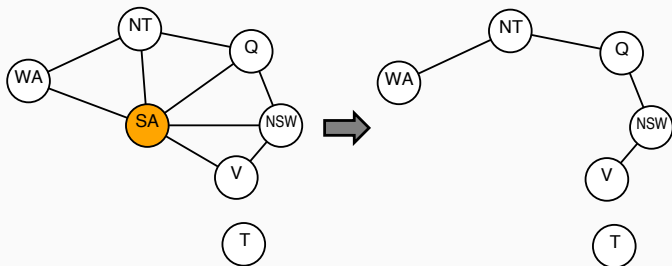
1. Choisir une variable comme étant la racine, et ordonner les variables de la racine aux feuilles, de façon à ce que le parent de chaque nœud le précède



2. Pour  $j$  de  $n$  à 2, appliquer  $\text{RemoveInconsistent}(\text{Parent}(X_j), X_j)$
3. Pour  $j$  de 1 à  $n$ , affecter  $X_j$  de façon à ce qu'il soit consistant avec  $\text{Parent}(X_j)$

## CSPs *quasiment* structurés sous forme d'arbre

- **Conditionnement** : instancier une variable, restreindre les domaines de ses voisins



- **Conditionnement du coupe-cycle** : instancier (de toutes les façons possibles) un ensemble de variables de façon à ce que le graphe de contraintes restant soit un arbre
- Cycle coupé de taille  $c \Rightarrow$  complexité en  $O(d^c \times (n - c)d^2)$
- Très rapide pour  $c$  petit

## Conclusion

---

- Formulation particulière d'un état avec un ensemble de couples valeurs/attributs
- Les conditions d'une solution sont représentées par un ensemble de contraintes sur les variables
- La recherche avec backtrack est une technique efficace
- Des heuristiques générales (indépendantes du domaine) permettent de résoudre plus rapidement un CSP
- D'autres techniques utilisant la décomposition peuvent aussi être efficaces.