Intelligence Artificielle
Contraintes

Bruno Bouzy

http://web.mi.parisdescartes.fr/~bouzy
bruno.bouzy@parisdescartes.fr

Licence 3 Informatique

UFR Mathématiques et Informatique
Université Paris Descartes

Problèmes de satisfaction de contraintes

- Exemples de CSP
- Recherche en arrière pour les CSPs (backtracking search)
- Structure des problèmes
- CSP et recherche locale

Problèmes de satisfaction de contraintes

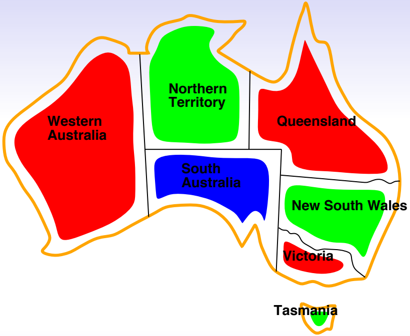
Problèmes de satisfaction de contraintes (CSP)

- 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 : $D_i = \{rouge, vert, bleu\}$
- Contraintes : les régions adjacentes doivent être de couleurs différentes
  - Par exemple, $WA \neq NT$ (si le langage le permet)
  - Ou $(WA, NT) \in \{(rouge, vert), (rouge, bleu), (vert, rouge), (vert, bleu) \ldots\}$

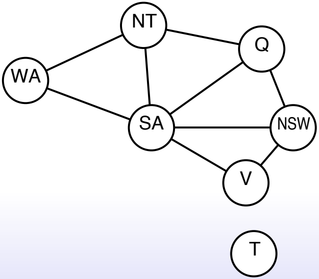
Exemple : coloriage de carte



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

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

Variétés de CSPs

- 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

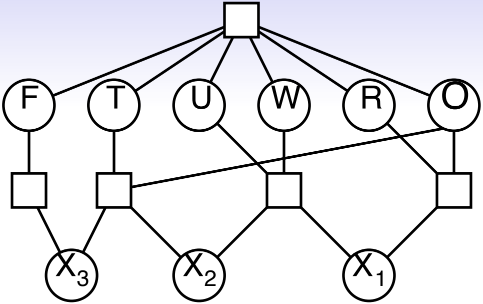
Variétés de contraintes

- 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 cryptarithmétiques
- Préférences (ou contraintes souples)
  - Par exemple, *rouge* est mieux que *vert*
  - Souvent représentable par un coût associé à chaque affectation de variable

$\Rightarrow$ Problèmes d'optimisation de variables

Exemple : puzzle cryptarithmétique

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



- 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 :
  - $Alldiff(F, T, U, W, R, O)$
  - $O + O = R + 10X_1$
  - $X_1 + W + W = U + 10X_2$
  - ...

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 mondé 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
  - $\rightarrow$ 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
  - $\rightarrow$ $n!d^n$ feuilles
  - $\rightarrow$ alors qu'il n'y a que d^n affectations possible!!

Problèmes de satisfaction de contraintes

- Exemples de CSP
- Recherche en arrière pour les CSPs (backtracking search)
- Structure des problèmes
- CSP et recherche locale

Backtracking search

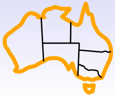
- L'affectation des variables est commutative
  - L'ordre dans lequel on affecte les variables n'a pas d'importance
  - $WA = rouge$ puis $NT = vert$ est la même chose que $NT = vert$ puis $WA = rouge$
- Il n'y a donc besoin de ne considérer qu'une seule variable par nœud de l'arbre de recherche

  $\rightarrow$ $b = d$, 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 $n \sim 25$

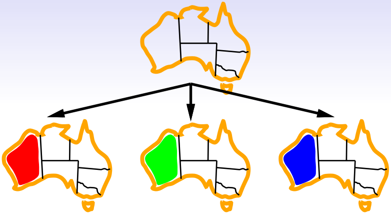
Algorithme de recherche par retour arrière

```
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
```

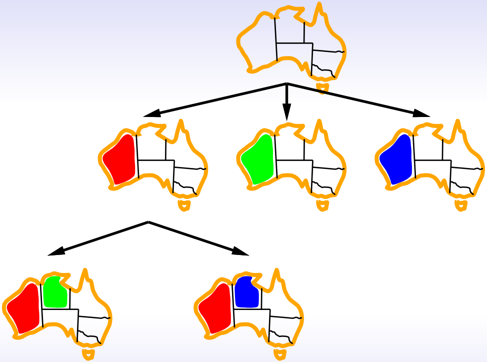
Exemple



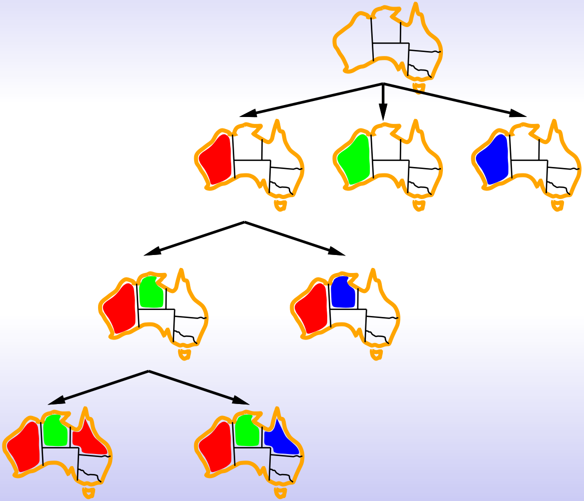
Exemple



Exemple



Exemple



Améliorer l'efficacité du la recherche par backtrack

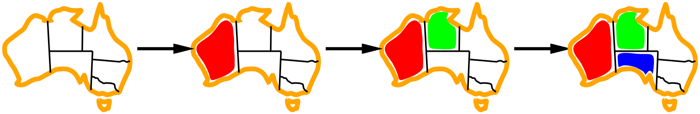
1. Comment choisir la variable à affecter ensuite? (Select-Unassigned-Variable)
2. Comment ordonner les valeurs des variables? (Order-Domain-Values)
3. Est-il possible de détecter un échec inévitable plus tôt?
4. Comment tirer avantage de la structure du problème?

Améliorer l'efficacité du 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?

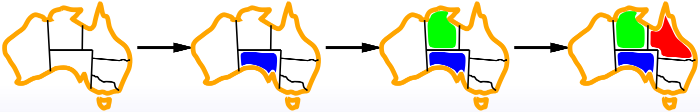
Valeurs minimum restantes (MRV)

- 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
  - $\Rightarrow$ choisir la variable qui a le plus de contraintes à respecter parmi les variables restantes



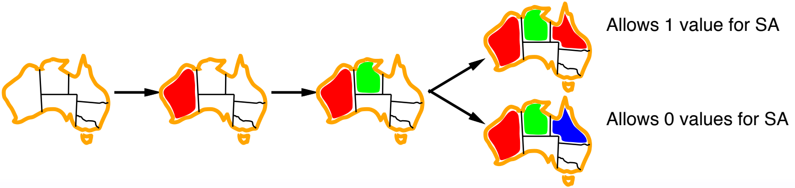
Améliorer l'efficacité du la recherche par backtrack

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

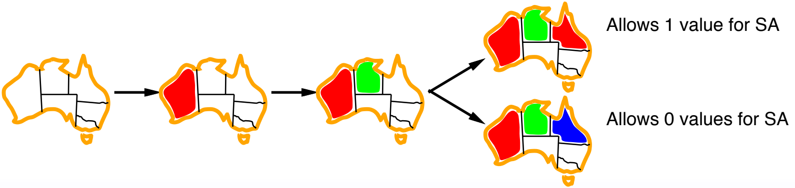
- Etant donné une variable, choisir celle qui a la valeur la moins contraignante

  $\Rightarrow$ 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
  - $\Rightarrow$ la variable qui empêche le moins d'affectations possibles sur les variables restantes



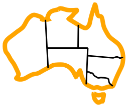
- Combiner ces heuristiques permet de résoudre le problème des n reines, avec $n = 1000$

Améliorer l'efficacité du 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?

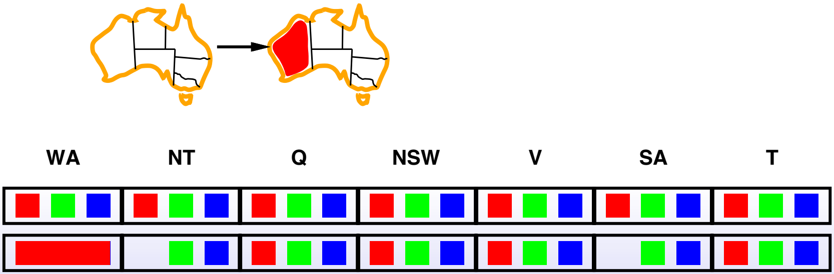
Vérification en avant

- 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 "légale" possible



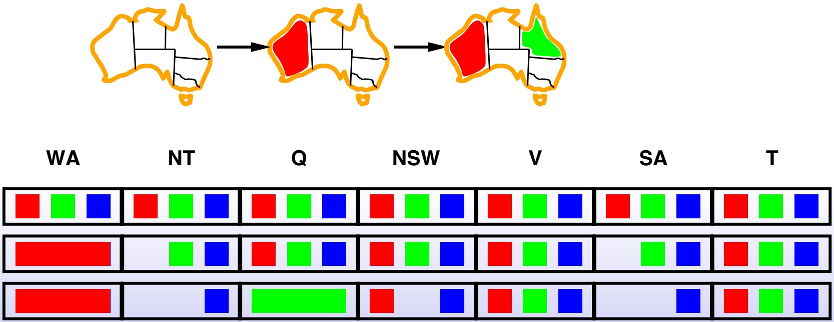
Vérification en avant

- 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 “légale” possible



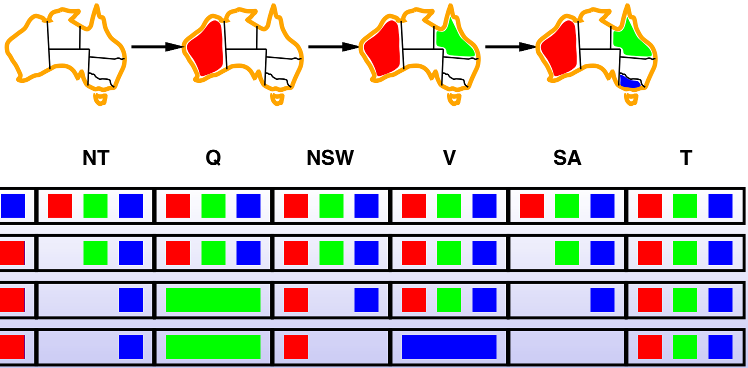
Vérification en avant

- 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 "légale" possible



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 "légale" possible

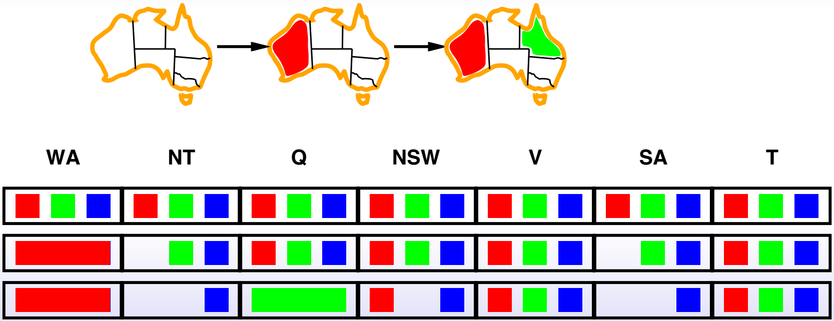


Propagation de contraintes

- 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

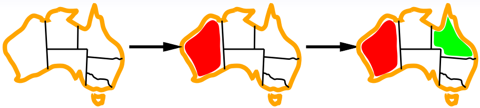
Propagation de contraintes

- 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



Propagation de contraintes

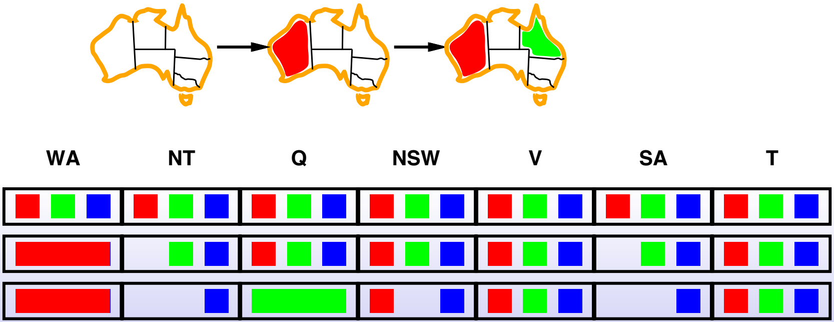
- La vérification en avant permet de propager l'information des variables affectées aux variables non affectées, mais ne permet pas de détecter tous les échecs



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

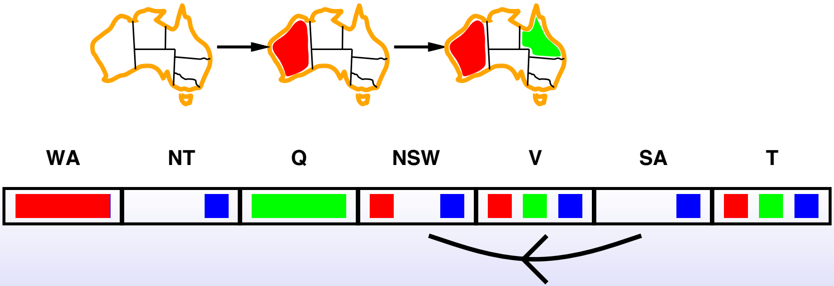
Consistence des arcs

- 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



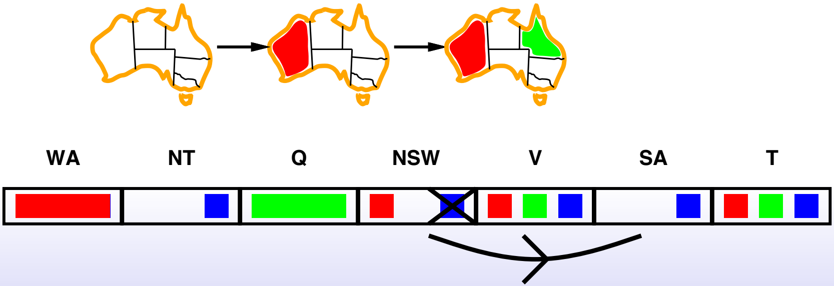
Consistence des arcs

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



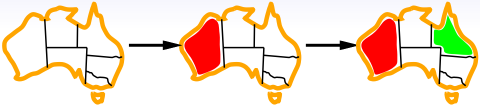
Consistence des arcs

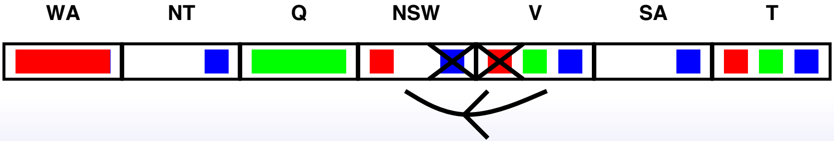
- La forme la plus simple de propagation est de rendre les arcs consistents
- $X \to Y$ est consisistant ssi pour **toute** valeur x de X, il y a **au moins un** y autorisé



Consistence des arcs

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

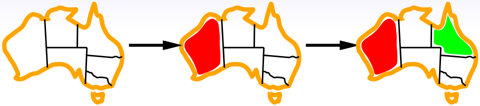


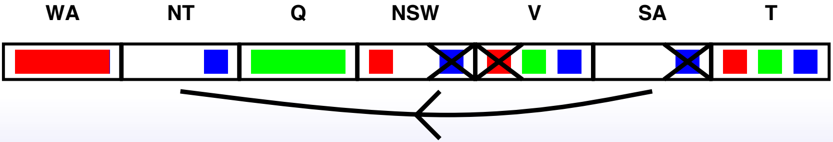


- Si X perd une valeur, les voisins de X doivent être revérifiés

Consistence des arcs

- La forme la plus simple de propagation est de rendre les arcs consistents
- $X \to Y$ est consisistant 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 consistence d'arcs

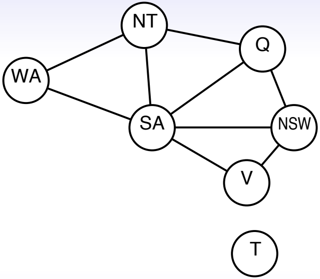
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi]; removed ← true
   return removed
```

Problèmes de satisfaction de contraintes

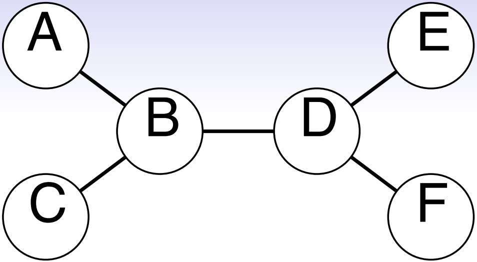
- Exemples de CSP
- Recherche en arrière pour les CSPs (backtracking search)
- Structure des problèmes
- CSP et recherche locale

Structure des problèmes



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

CSPs structurés sous forme d'arbre



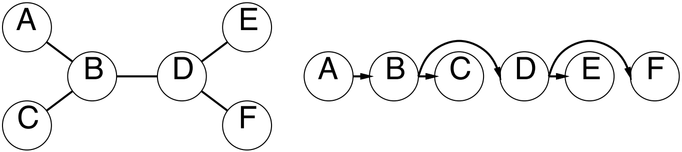
Theorem

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

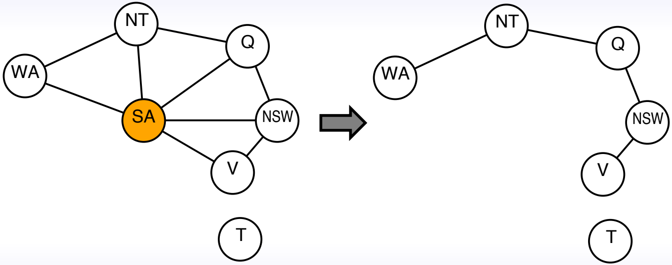
1. Choisir une variable comme étant la racine, et ordonner les variables de la racine aux feuilles, de façon à ce que le parent de chaque nœud le précède



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

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

Problèmes de satisfaction de contraintes

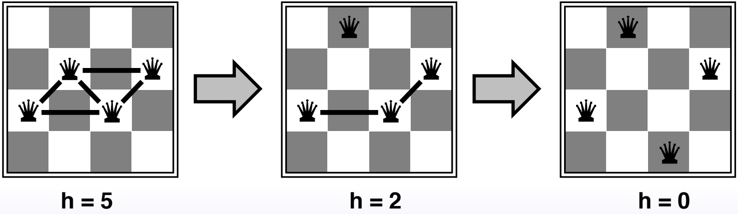
- Exemples de CSP
- Recherche en arrière pour les CSPs (backtracking search)
- Structure des problèmes
- CSP et recherche locale

CSP et recherche locale

- Les algorithmes de recherche locale fonctionnent avec des états "complets", c'est à dire dans lesquels toutes les variables sont affectées.
- Pour appliquer ces algorithmes aux CSPs :
  - Permettre d'avoir des états avec des contraintes non satisfaites
  - Les opérateurs permettent de réaffecter la valeur d'une variable
- Sélection des variables : n'importe quelle variable en conflit
- Sélection d'une valeur grace à l'heuristice min-conflict
  - choisir une valeur qui enfreint le moins de contraintes
  - par exemple, *hillclimb* avec $h(n) =$ nombre total de contraintes non respectées

Exemple : les n reines

- Etats : 4 reines sur 4 colonnes ($4^4 = 256$ états)
- Actions : déplacer une reine dans sa colonne
- Test du but : pas d'attaque entre les reines
- Evaluation : $h(n)$ est le nombre d'attaques sur le plateau



- Etant donné un état initial aléatoire, cet algorithme peut résoudre avec une grande probabilité le problème des n-reines pour tout n en temps presque constant (e.g., $n = 10000000$)