IFT 615 – Intelligence artificielle

Satisfaction de contraintes

Hugo Larochelle
Département d'informatique
Université de Sherbrooke
http://www.dmi.usherb.ca/~larocheh/cours/ift615.html

Objectifs

- À la fin de cette leçon vous devriez:
  - pouvoir modéliser un problème donné comme un problème de satisfaction de contraintes
  - pouvoir expliquer et simuler le fonctionnement des algorithmes de satisfaction de contraintes de base: *backtrack-search*, *forward-checking*, AC-3

Sujets couverts

- Modèle général des problèmes de satisfaction de contraintes
- Algorithme *backtracking-search*.
- Algorithme AC-3.
- Applications.

Rappel: Recherche dans un espace d'états

- Nous avons vu qu'un certain nombre de problèmes intéressants peuvent être résolus en les formulant comme des problèmes de recherche dans un graphe d'états:
  - on tient compte des aspects spécifiques à l'application en définissant une fonction heuristique (h) qui guide l'exploration efficace du problème
  - la fonction de transition tient compte de l'aspect dynamique de l'application
  - par contre, les états (les nœuds) du graphe sont « opaques » vis-à-vis de la l'algorithme de recherche:
    - l'algorithme de recherche ne sait pas comment le choix des successeurs d'un état est fait par la fonction de transition

Problème de satisfaction de contraintes

- La résolution de problèmes par la satisfaction de contraintes est un cas particulier de la recherche heuristique
- La structure interne des états a une représentation particulière
  - un état est un ensemble de variables avec des valeurs correspondantes
  - les transitions entre les états tiennent comptent de contraintes sur les valeurs possibles des variables
- Les algorithmes de satisfaction de contraintes vont utiliser des **heuristiques générales**, plutôt que des **heuristiques spécifiques à une application**
- En traduisant un problème sous forme de satisfaction de contraintes, on élimine la difficulté de définir l'heuristique pour notre application

Problème de satisfaction de contraintes

- Formellement, un problème de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*) est défini par:
  - un **ensemble fini de variables** $X_1, \ldots, X_n$.
    - » chaque variable X_i a un **domaine** D_i de valeurs possibles
  - un **ensemble fini de contraintes** $C_1, \ldots, C_m$ sur les variables.
    - » une contrainte restreint les valeurs pour un sous-ensemble de variables
- Un état d'un problème CSP est défini par une **assignation de valeurs** à certaines variables ou à toutes les variables
  - $\{X_i=v_i, X_j=v_j, \ldots\}$.
- Une assignation qui ne viole aucune contrainte est dite **compatible** ou **légale**
- Une assignation est **complète** si elle concerne toutes les variables
- Une solution à un problème CSP est une assignation **complète et compatible**
- Parfois, la solution doit en plus maximiser une fonction objective donnée

Exemple 1

- Soit le problème CSP défini comme suit:
  - ensemble de variables V = $\{X_1, X_2, X_3\}$
  - un domaine pour chaque variable $D_1 = D_2 = D_3 = \{1,2,3\}$.
  - une contrainte spécifiée par l'équation linéaire $X_1 + X_2 = X_3$.

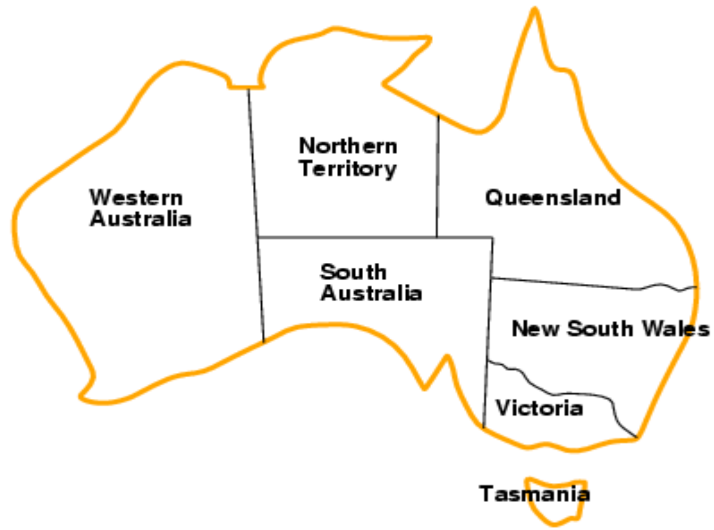
- Il y a trois solutions possibles:
  - (1,1,2)
  - (1,2,3)
  - (2,1,3)

Exemple 2: Colorier une carte

- On vous donne une carte de l'Australie
- Et on vous demande d'utiliser seulement trois couleurs (rouge, vert et bleu) de sorte que deux états frontaliers n'aient jamais les mêmes couleurs
- On peut facilement trouver une solution à ce problème en le formulant comme un problème CSP et en utilisant des algorithmes génériques pour CSP

Exemple 2: Colorier une carte

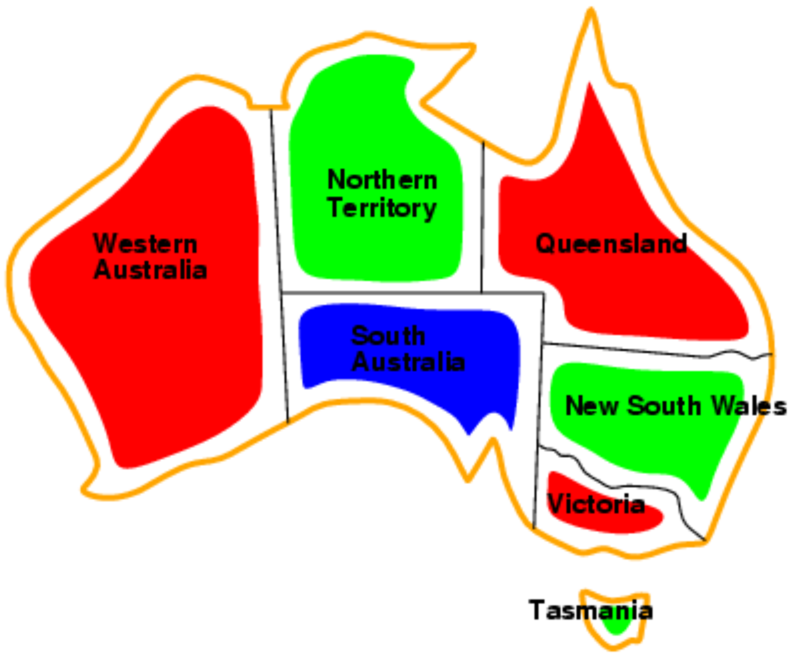
- Formulation du problème CSP:
  - les variables sont les régions:
    - » V = { WA, NT, Q, NSW, V, SA, T }
  - le domaine de chaque variable est l'ensemble des trois couleurs: {R, G, B}



  - contraintes: les régions frontalières doivent avoir des couleurs différentes
    - » WA≠ NT, …, NT≠ Q, …

Exemple 2: Colorier une carte

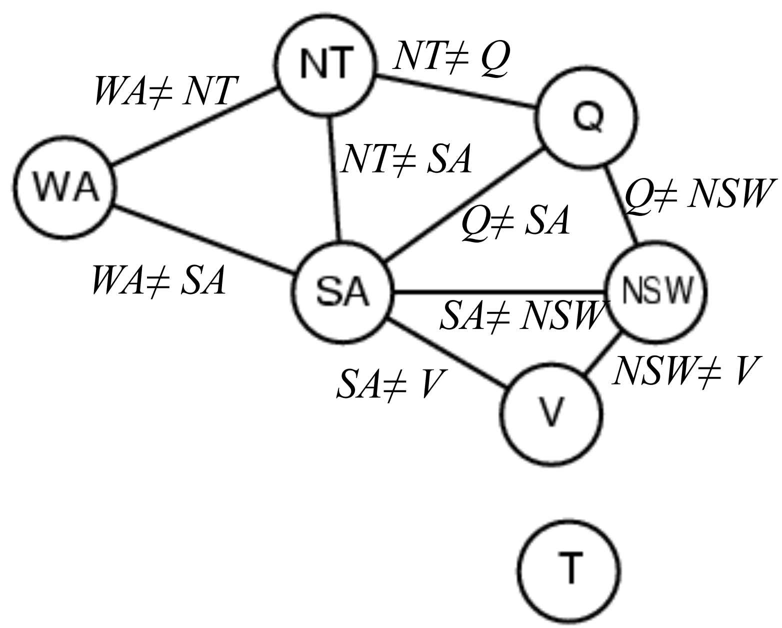
- Solution:



{ WA = R, NT = G, Q = R, NSW = G, V = R, SA = B, T = G }

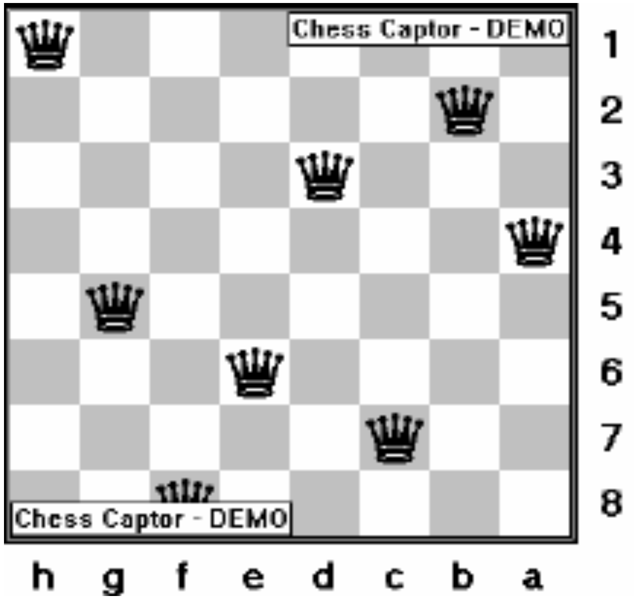
Graphe de contraintes

- Pour des problèmes avec des **contraintes binaires** (c-à-d. entre deux variables), on peut visualiser le problème CSP par **un graphe de contraintes**
- Un graphe de contraintes est un graphe dont les nœuds sont des variables (un nœud par variable) et les arcs sont des contraintes entre les deux variables



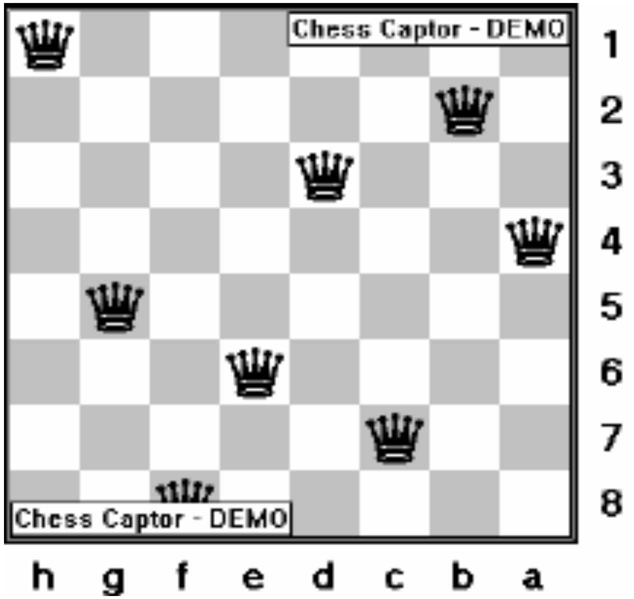
Exemple 3: N-Queens

- Positionner N reines sur un échiquier de sorte qu'aucune d'entre elles ne soit en position d'attaquer une autre
- Exemple avec 8 reines (*8-Queens*)
- Une reine peut attaquer une autre si elles sont toutes les deux sur la même ligne, la même colonne, ou la même diagonale.
- Avec N=4: 256 configurations.
- N=8: 16 777 216
- N= 16: 18,446,744,073,709,551,616 configurations



Exemple 3: N-Queens

- Modélisation comme problème CSP:
  - variables: colonnes 1, …, N
  - domaines: rangées 1, …., N
  - la colonne i a la valeur k si la reine dans la colonne i est dans la rangée k
  - contraintes: pas deux reines sur même ligne, colonne ou diagonale



Types de problèmes CSP

- CSP avec des domaines finis (et discrets).
- CSP booléens: les variables sont vraies ou fausses.
- CSP avec des domaines continus (et infinis)
  - par exemple, problème d'ordonnancement de travaux sans contraintes sur les durées
- CSP avec des contraintes linéaires (ex.: $X_1 < X_2 + 10$)
- CSP avec des contraintes non linéaires (ex.: $\log X_1 < X_2$)
- …
- Les problèmes CSP sont étudiés de manière approfondie en recherche opérationnelle

Algorithme Depth-First-Search pour CSP

- On peut utiliser la recherche dans un graphe avec les paramètres suivants:
  - un état est une assignation
  - état initial: assignation vide { }
  - fonction successeur: assigne une valeur à une variable non encore assignée
  - but: assignation complète et compatible
- L'algorithme est général et s'applique à tous les problèmes CSP
- Comme la solution doit être complète, elle apparaît à une profondeur n, si nous avons n variables
- Pour cette raison, *depth-first-search* fonctionne souvent bien

Limitations de l'approche précédente

- Supposons une recherche en largeur:
  - le nombre de branches au premier niveau, dans l'arbre est de n*d (d est la taille du domaine), parce que nous avons n variables, chacune pouvant prendre d valeurs
  - au prochain niveau, on a (n-1)d successeurs pour chaque nœud
  - ainsi de suite jusqu'au niveau n
  - cela donne $n!*d^n$ nœuds générés, pour seulement d^n assignations complètes
- L'algorithme ignore la **commutativité** des transitions:
  - SA=R suivi de WA=B est équivalent à WA=B suivi de SA=R
  - si on tient compte de la commutativité, le nombre de nœuds générés est d^n
- *Depth-first-search* avec l'assignation **d'une seule variable** dans un nœud et qui **recule (*backtrack*) lorsqu'aucune assignation compatible** est possible est appelé ***backtracking-search***
- C'est l'algorithme de base pour résoudre les problèmes CSP

Backtracking search (S., Figure 6.3, page 215)

function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
  return BACKTRACK({ }, *csp*)

function BACKTRACK(*assignment*, *csp*) **returns** a solution, or failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* **then**
      add {*var* = *value*} to *assignment*
      inferences ← INFERENCE(*csp*, *var*, *value*)
      if *inferences* ≠ *failure* **then**
        add *inferences* to *assignment*
        result ← BACKTRACK(*assignment*, *csp*)
        if *result* ≠ *failure* **then**
          return *result*
    remove {*var* = *value*} and *inferences* from *assignment*
  return *failure*

Les fonctions successeurs et buts ne sont pas explicites.

Pas besoin de maintenir un graphe de nœuds (*closed*).

Open implicitement représenté par la pile de récursivité.

Illustration de backtracking-search

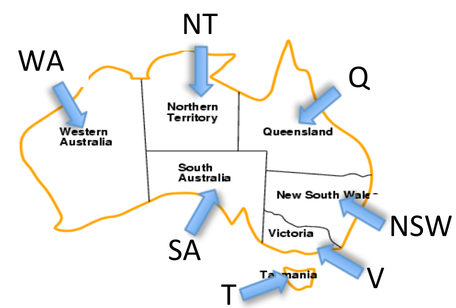
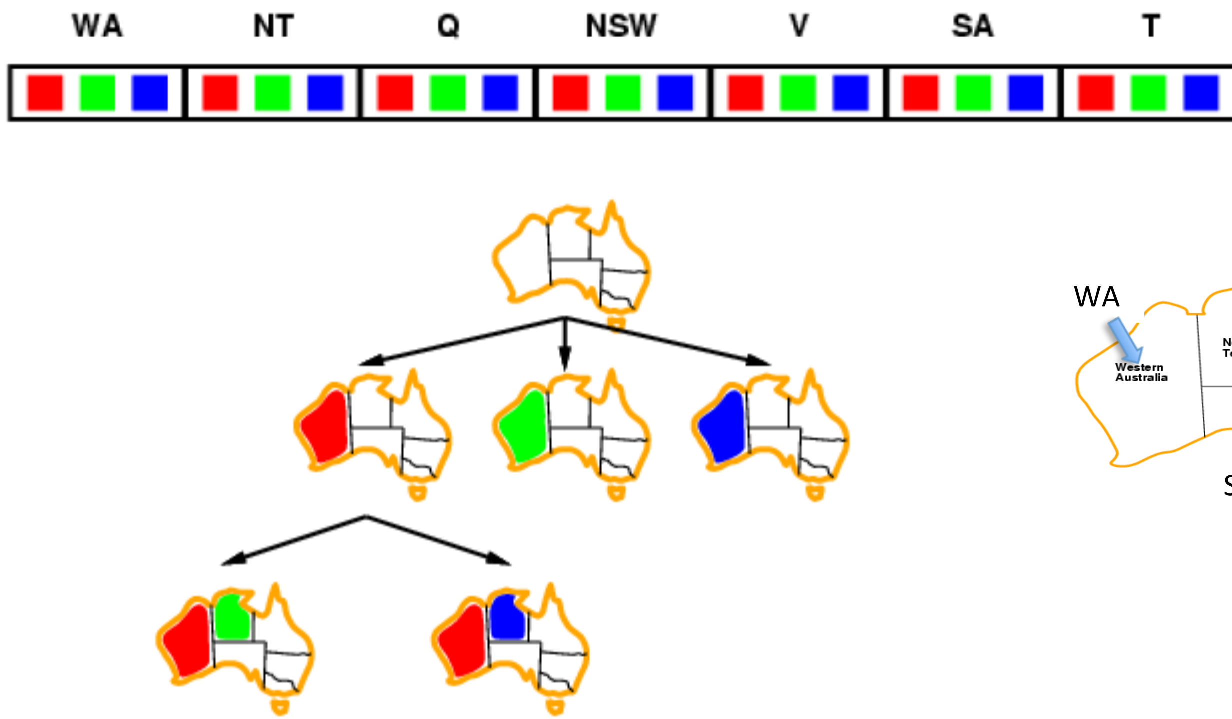


Illustration de backtracking-search



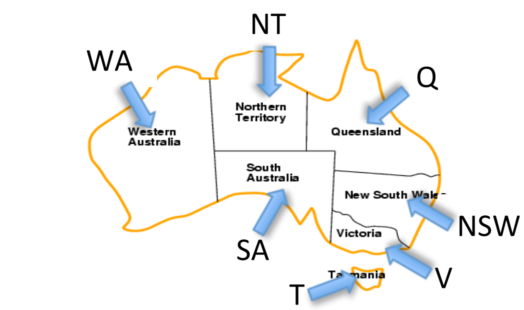
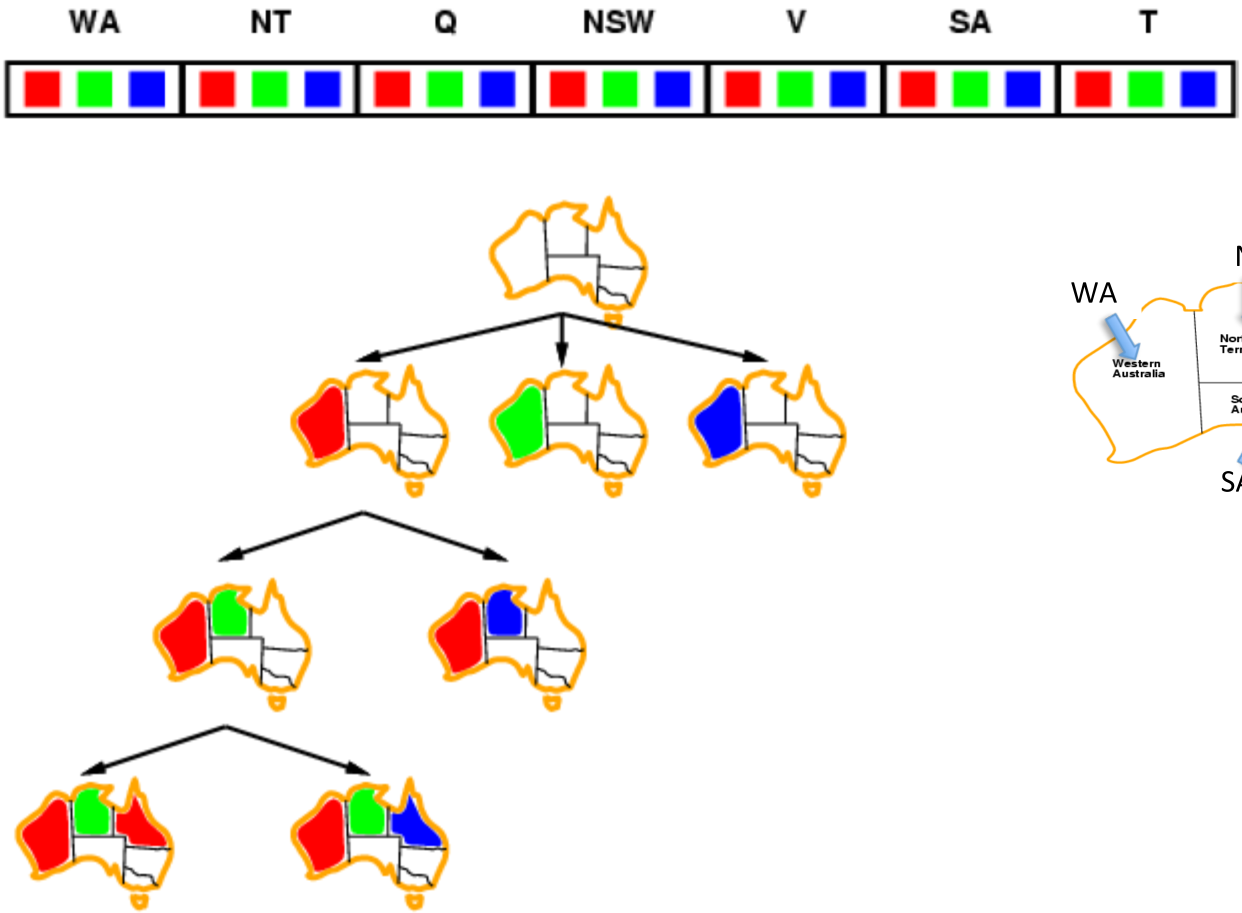
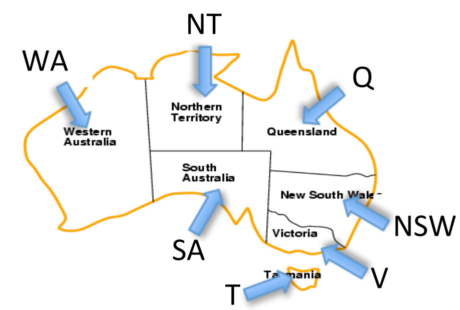


Illustration de backtracking-search





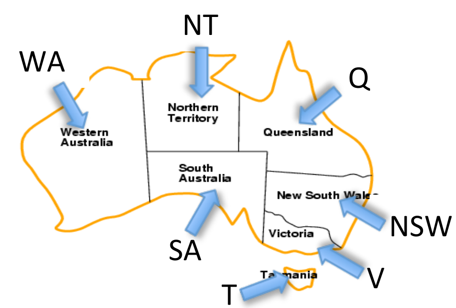
Amélioration de backtracking-search

- Sans heuristiques, l'algorithme est limité
  - il peut résoudre le problème de 25 reines
- Des **heuristiques générales** peuvent améliorer l'algorithme significativement:
  - choisir judicieusement la prochaine variable (SELECT-UNASSIGNED-VARIABLE)
  - choisir judicieusement la prochaine valeur à assigner (ORDER-DOMAIN-VALUES)
  - détecter les assignations conflictuelles et réduire les domaines (INFERENCE)

Choisir l'ordre d'assignation des variables

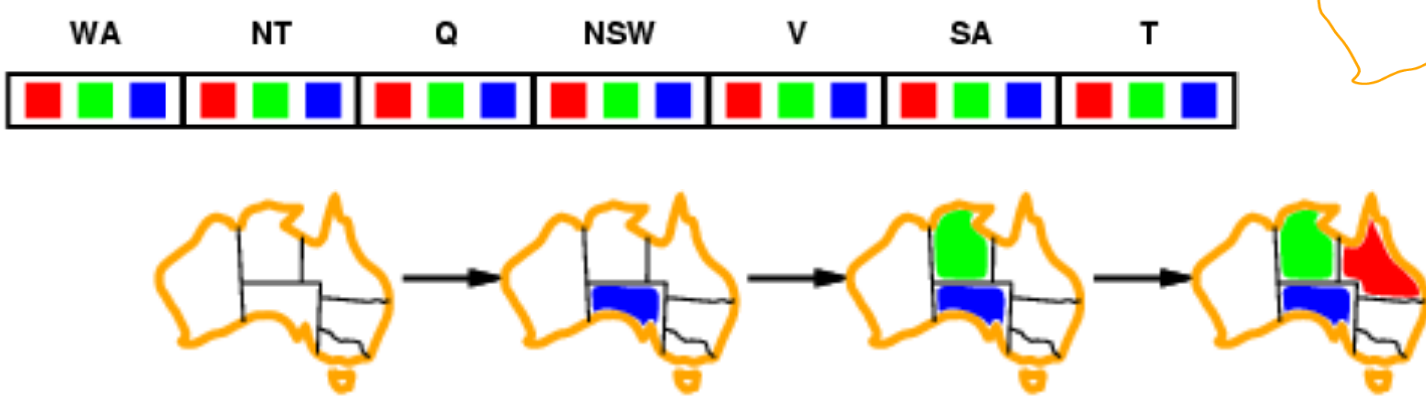
- À chaque étape, choisir la variable avec le moins de valeurs compatibles restantes
    - c-à-d., la variable « posant le plus de restrictions »
    - appelée ***minimum remaining value*** (MRV) *heuristic*
      ou ***most constrained variable*** *heuristic*.

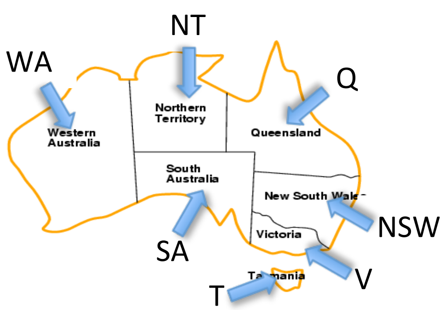
- Illustration:



Choisir l'ordre d'assignation des variables

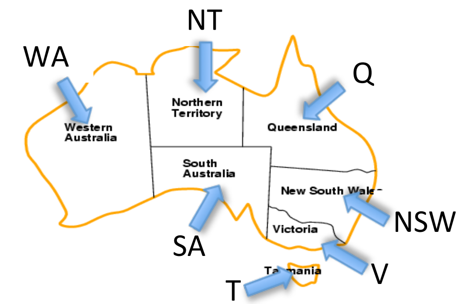
- Si le critère précédent donne des variables avec le même nombre de valeurs compatibles restantes:
  - choisir celle ayant le plus de contraintes impliquant des variables non encore assignées
  - appelée ***degree heuristic***.





Choisir la prochaine valeur à assigner

- Pour une variable donnée, choisir une valeur qui invalide le moins de valeurs possibles pour les variables non encore assignées



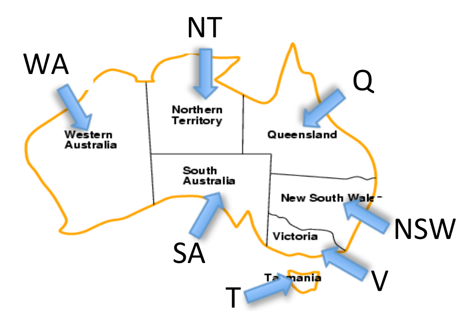
laisse une seule valeur pour SA

ne laisse aucune valeur pour SA

- Ces heuristiques permettent de résoudre un problème de 1000 reines

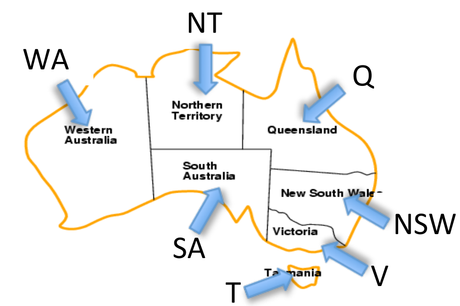
Détecter les assignations conflictuelles: algorithme forward checking

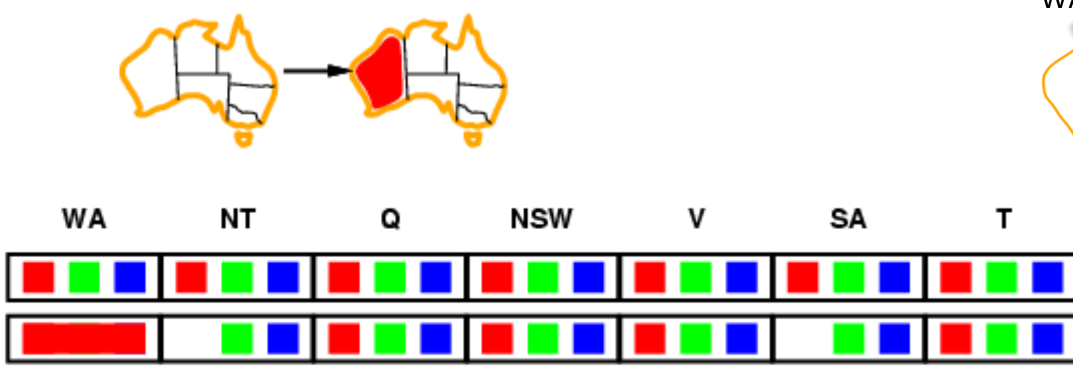
- L'idée de l'algorithme *forward checking* (vérification anticipative) est de:
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide



Détecter les assignations conflictuelles: algorithme forward checking

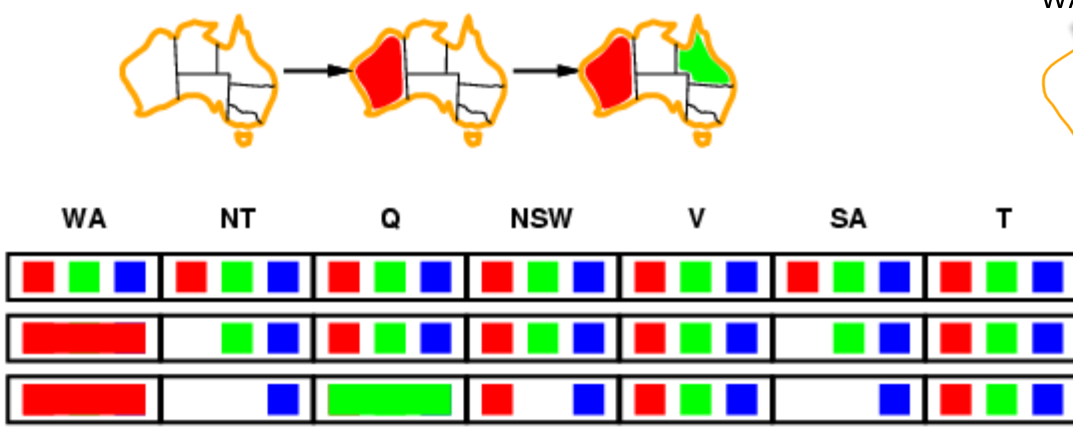
- L'idée de l'algorithme *forward checking* (vérification anticipative) est de:
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide

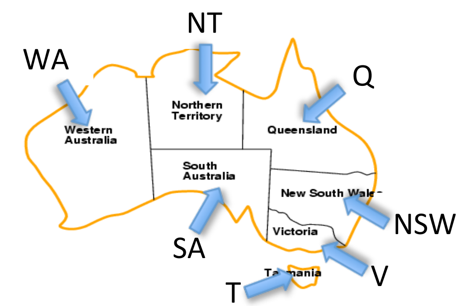




Détecter les assignations conflictuelles: algorithme forward checking

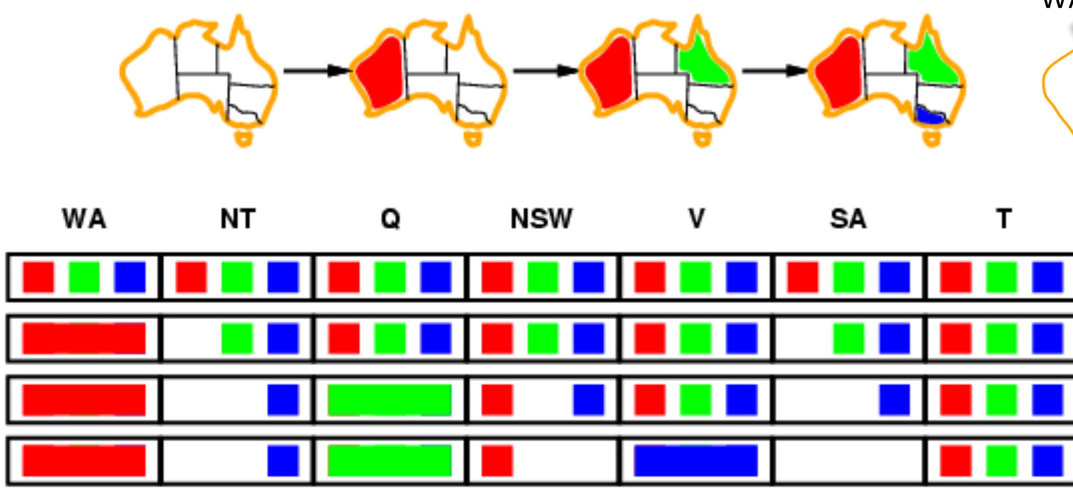
- L'idée de l'algorithme *forward checking* (vérification anticipative) est de:
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide

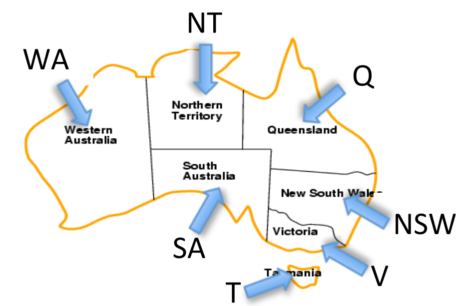




Détecter les assignations conflictuelles: algorithme forward checking

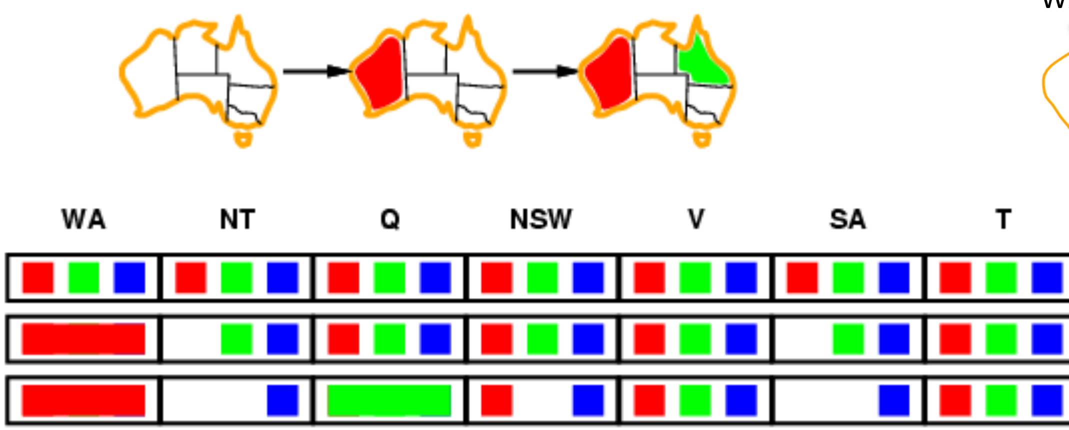
- L'idée de l'algorithme *forward checking* (vérification anticipative) est de:
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide

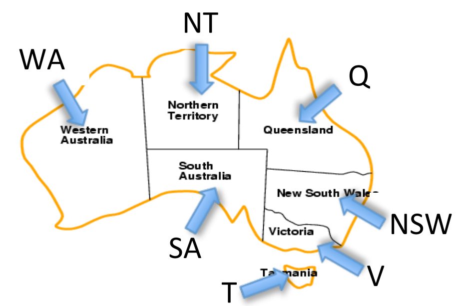




Détecter les assignations conflictuelles: algorithme forward checking

- *Forward checking* propage l'information des variables assignées vers les variables non assignées, mais ne détecte pas les conflits locaux entre variables:

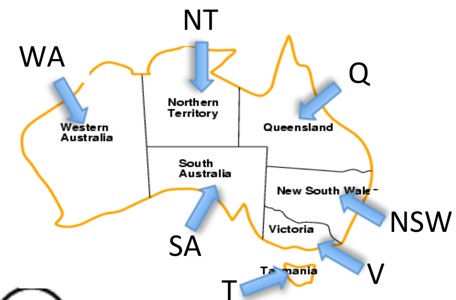




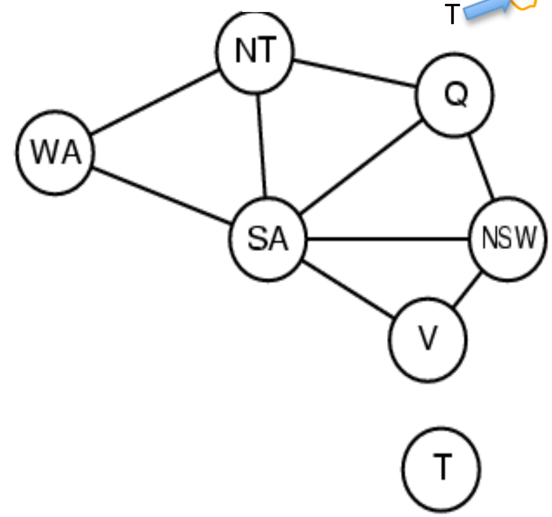
- NT et SA ne peuvent pas être bleues ensemble!
- La propagation des contraintes permet de vérifier les contraintes localement

Arc consistency

- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - vérifie la compatibilité entre les arcs
  - c-à-d., la compatibilité des contraintes entre deux variables

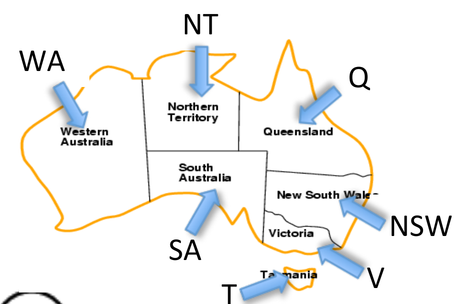


- L'arc X →Y est compatible si et seulement si
  - pour chaque valeur x de X il existe au moins une valeur permise y de Y

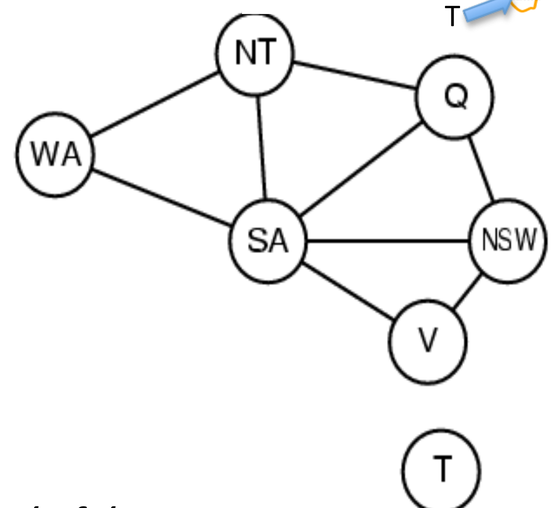


Arc consistency

- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - vérifie la compatibilité entre les arcs
  - c-à-d., la compatibilité des contraintes entre deux variables



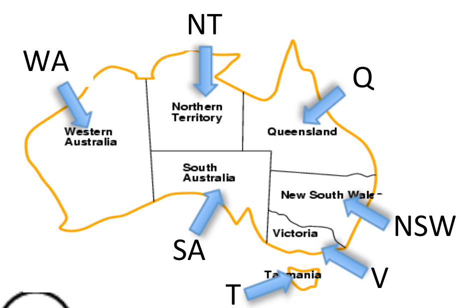
- L’arc X →Y est compatible si et seulement si
  - pour chaque valeur x de X il existe au moins une valeur permise y de Y



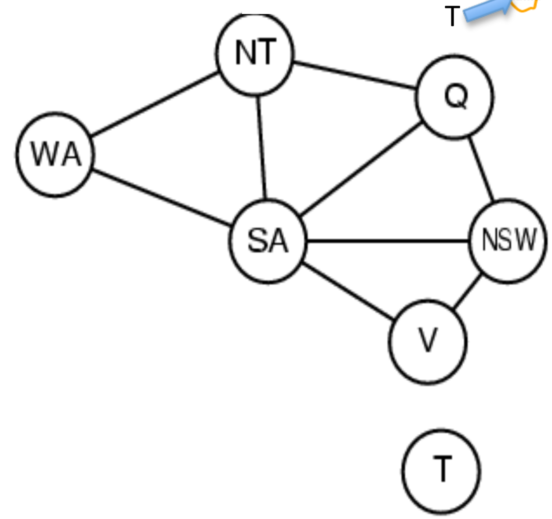
Si une variable perd une valeur, ses voisins doivent être revérifiés

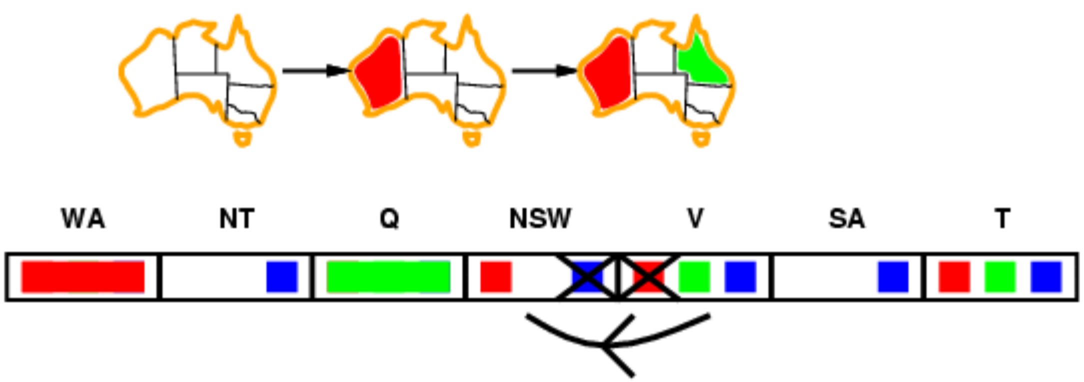
Arc consistency

- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - vérifie la compatibilité entre les arcs
  - c-à-d., la compatibilité des contraintes entre deux variables



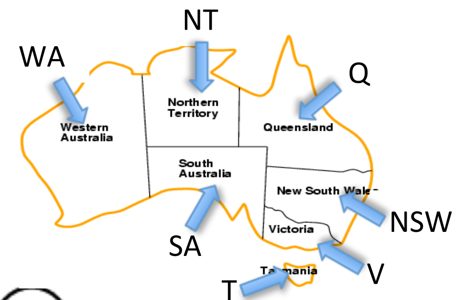
- L'arc X →Y est compatible si et seulement si
  - pour chaque valeur x de X il existe au moins une valeur permise y de Y



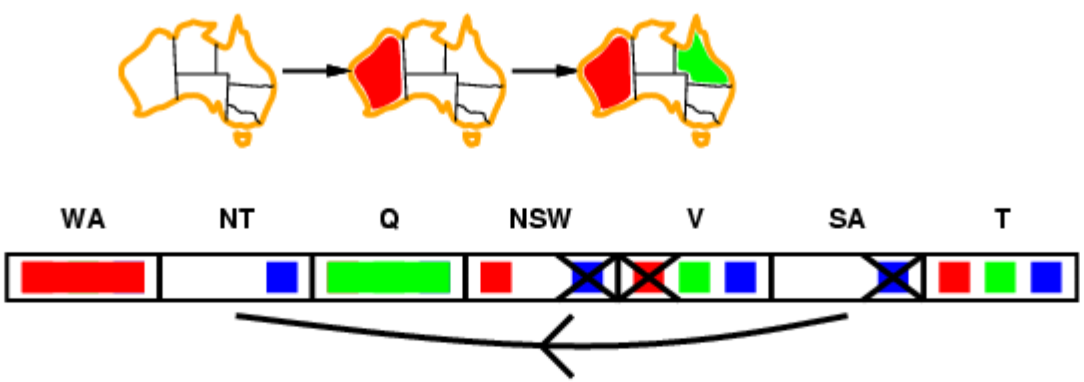


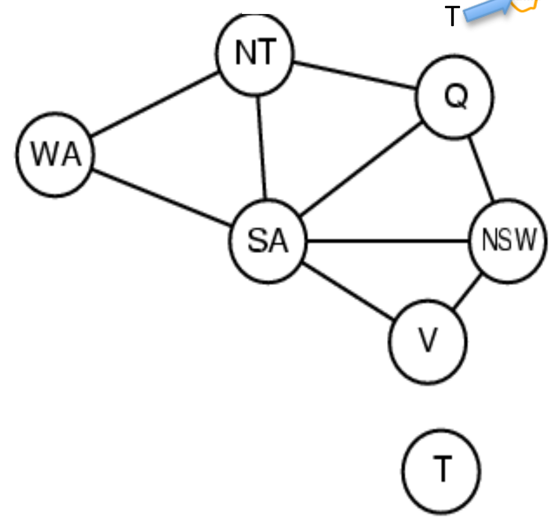
Arc consistency

- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - vérifie la compatibilité entre les arcs
  - c-à-d., la compatibilité des contraintes entre deux variables



- L'arc X →Y est compatible si et seulement si
  - pour chaque valeur x de X il existe au moins une valeur permise y de Y





Arc consistency algorithm AC-3

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with components (X, D, C)
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REVISE(csp, X_i, X_j) then
         if size of D_i = 0 then return false
         for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
   return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
   revised ← false
   for each x in D_i do
      if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
         delete x from D_i
         revised ← true
   return revised
```

Arc consistency algorithm AC-3

- Appliqué au début de *backtracking-search* et/ou juste après chaque nouvelle assignation de valeur à une variable
- Complexité: $O(n^2d^3)$. Une meilleure version ($O(n^2d^2)$) existe

Recherche locale pour les CSP

- Le chemin à la solution est sans importance
  - on peut utiliser une méthode de recherche locale (hill-climbing, etc.)
  - on peut travailler avec des états qui sont des assignations complètes (compatibles ou non)
- Le problème de la recherche locale est qu'elle peut tomber dans des optima locaux
- Par contre, pour *N-Queens*, l'algorithme *min-conflicts* fonctionne étonnamment bien
  - fonction objective: on minimise le nombre de conflits
  - ressemble à *hill-climbing*, mais avec un peu de stochasticité

Recherche locale pour les CSP

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
   inputs: csp, a constraint satisfaction problem
           max_steps, the number of steps allowed before giving up

   current ← an initial complete assignment for csp
   for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
   return failure
```

- Peut résoudre un problème *1,000,000-Queens* en 50 étapes!
- La raison du succès de la recherche locale est qu’il existe plusieurs solutions possibles, « éparpillés » dans l’espace des états
- A été utilisé pour céduler les observations du Hubble Space Telescope (roule en 10 minutes, plutôt que 3 semaines!)

Au-delà de AC-3

- Exploiter la structure du domaine (Section 6.5)
  - certains graphes de contraintes ont une structure « simple » qui peut être exploitée (ex.: un arbre)
  - peut améliorer le temps de calcul exponentiellement

Applications

- Problèmes d'horaires (ex.: horaire des cours):
  - dans ce cours, nous avons vu des méthodes simples, seulement pour des contraintes dures. La plupart des approches tiennent compte des contraintes souples.
    - » http://www.asap.cs.nott.ac.uk/publications/pdf/ekb-compj.pdf
    - » http://www.springerlink.com/content/erylu61yx9tpj3hb/
    - » http://www.emn.fr/x-info/jussien/publications/cambazard-PATAT04.pdf
- D'autres applications:
  - certains algorithmes de planification invoquent des algorithmes CSP
  - planification de caméras dans les jeu vidéo:
    - » O. Bourne and A. Sattar. Automatic Camera Control with Constraint Satisfaction Methods. In AI Game Programming Wisdom 3, by Steve Rabin, Section 3.2, pages 173—187, 2006.

Résumé

- Les problèmes CSP sont des problèmes de recherche dans un espace d'assignation de valeurs à des variables
- *Backtracking-search = depth-first-search* avec une variable assignée par nœud et qui recule lorsqu'aucune assignation compatible
- L'ordonnancement des variables et des assignations de valeur aux variables jouent un rôle significatif dans la performance
- *Forward-checking* empêche les assignations qui conduisent à un conflit
- La propagation des contraintes (par exemple, *arc consistency*) détecte les incompatibilités locales
- Les méthodes les plus efficaces exploitent la structure du domaine
- Application surtout à des problèmes impliquant l'ordonnancement de tâche