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 de l'algorithme *backtracking-search*
  - décrire les différentes façon d'accélérer *backtracking-search*, incluant les algorithmes d'inférence *forward-checking* et AC-3
  - pouvoir résoudre un problème de satisfaction de contraintes avec la recherche locale

Problème de satisfaction de contraintes

- La résolution de problèmes de satisfaction de contraintes peut être vu comme un cas particulier de la recherche heuristique
- La structure interne des états (noeuds) 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
- Sachant cela, on va pouvoir 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 $h(n)$ 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 (noeud)** 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**

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
- 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

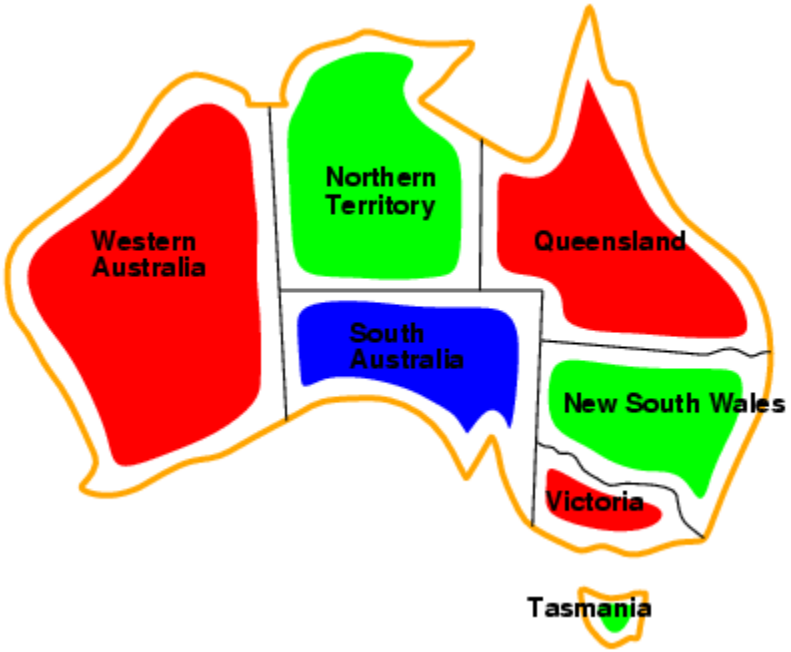
Exemple 2 : Colorier une carte

- 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
    - » $WA \neq NT, \ldots, NT \neq Q, \ldots$

Exemple 2 : Colorier une carte

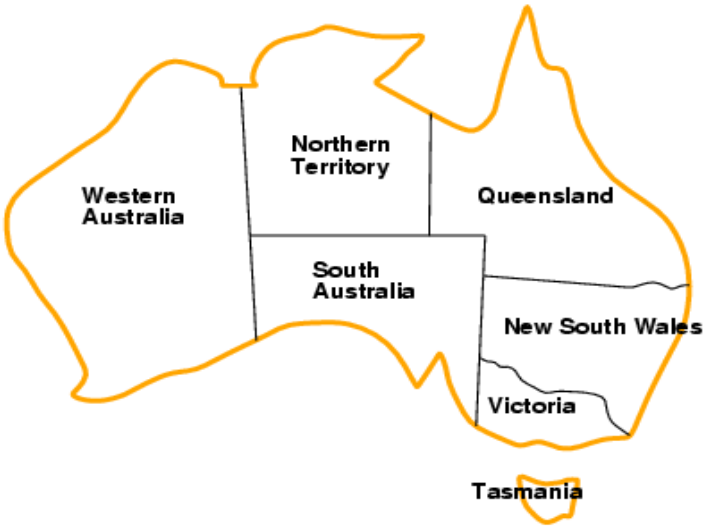
- Solution :

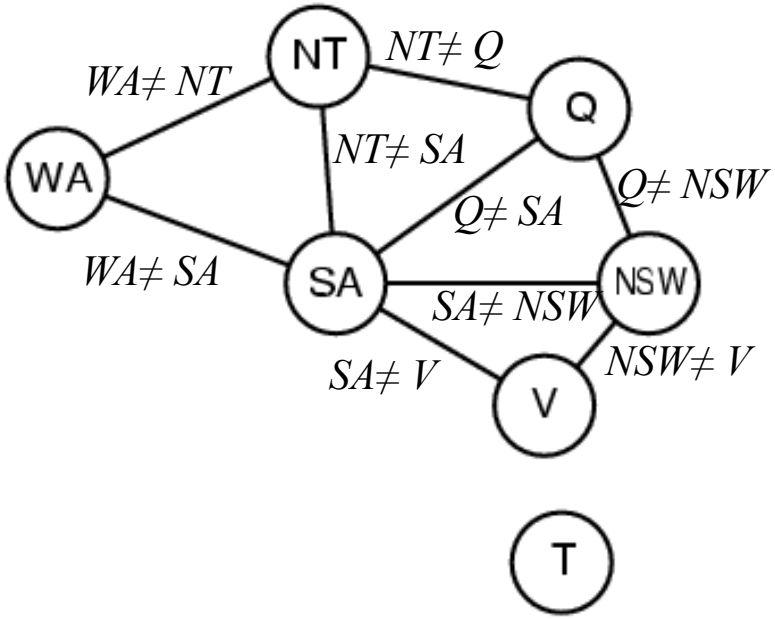


{ WA = R, NT = G, Q = R, NSW = G, V = R, SA = B, T = G }

Graphe de contraintes

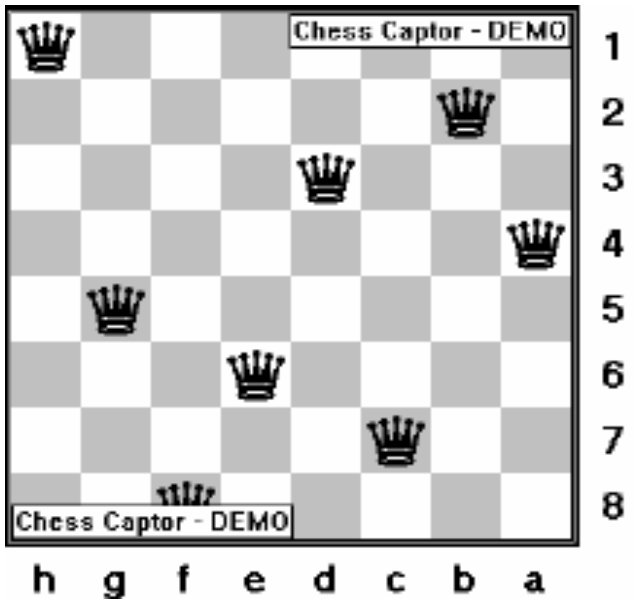
- Pour des problèmes avec des **contraintes binaires** (c-à-d. entre deux variables), on peut visualiser le problème CSP par **un graphe de contraintes**
- Un graphe de contraintes est un graphe dont les nœuds sont des variables (un nœud par variable) et les arcs sont des contraintes entre les deux variables





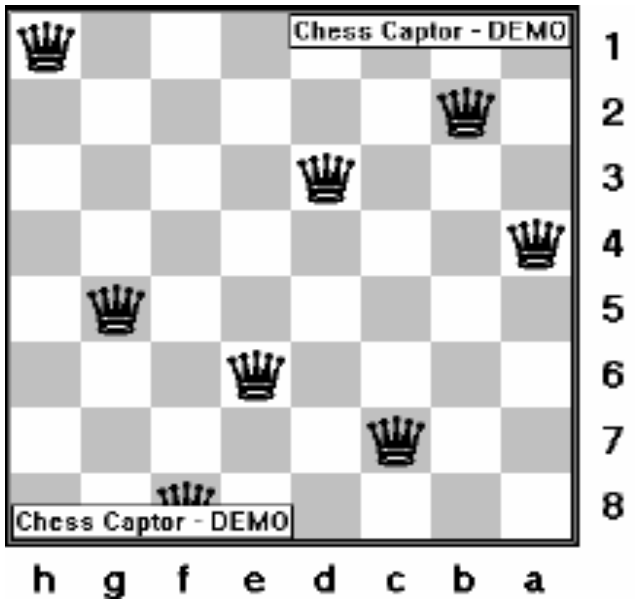
Exemple 3 : *N-Queens*

- 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



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 transition : 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
- **Idée :** considérer **une seule variable** à assigner à chaque niveau et **reculer (*backtrack*) lorsqu'aucune assignation compatible** est possible
- Le résultat est ***backtracking-search*** : c'est l'algorithme de base pour résoudre les problèmes CSP

Algorithme *backtracking-search*

Algorithme BACKTRACKING-SEARCH(*csp*)

1. retourner BACKTRACK({ }, *csp*)

Algorithme BACKTRACK(*assignations, csp*)

1. si *assignations* est complète, retourner *assignations*
2. *X* = VAR-NON-ASSIGNÉE(*assignations, csp*)
3. pour chaque *v* dans VALEURS-ORDONNÉES(*X*, *assignations, csp*)
    4. si COMPATIBLE((*X* = *v*), *assignations, csp*)
        5. ajouter (*X* = *v*) à *assignations*
        6. *csp** = *csp* mais où DOMAINE(*X*, *csp*) est { *v* }
        7. *csp**, *ok* = INFÉRENCE(*csp**)
        8. si *ok* = vrai
            9. *résultat* = BACKTRACK(*assignations*, *csp**)
            10. si *résultat* ≠ faux, retourner *résultat*
        11. enlever (*X* = *v*) de *assignations*
12. retourner faux

information sur les variables, domaines et contraintes du problème CSP

ensemble d'assignations de variables à des valeurs

choix de prochaine variable

ordre des valeurs à essayer

tente de simplifier le problème CSP (si détecte conflit, *ok* = faux)

Illustration de *backtracking-search*

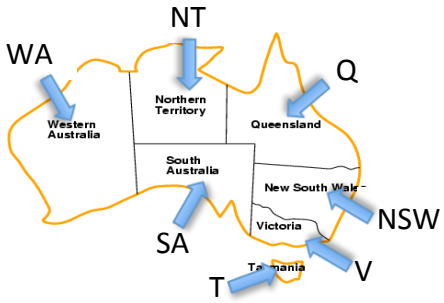
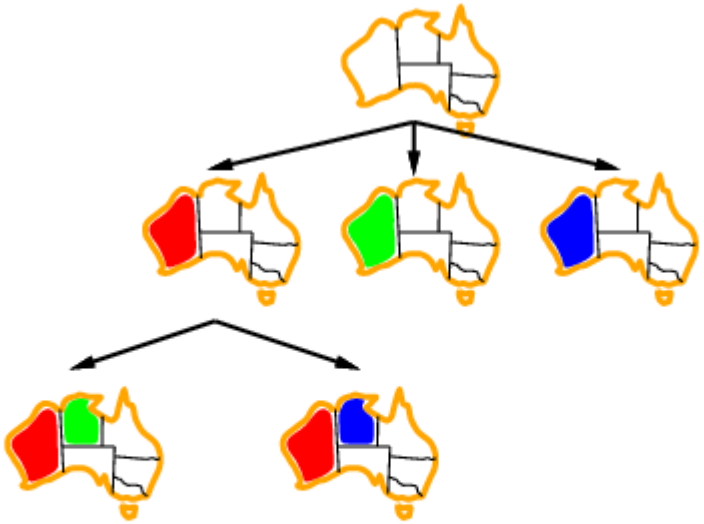


Illustration de *backtracking-search*



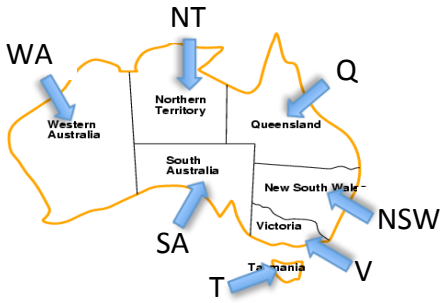
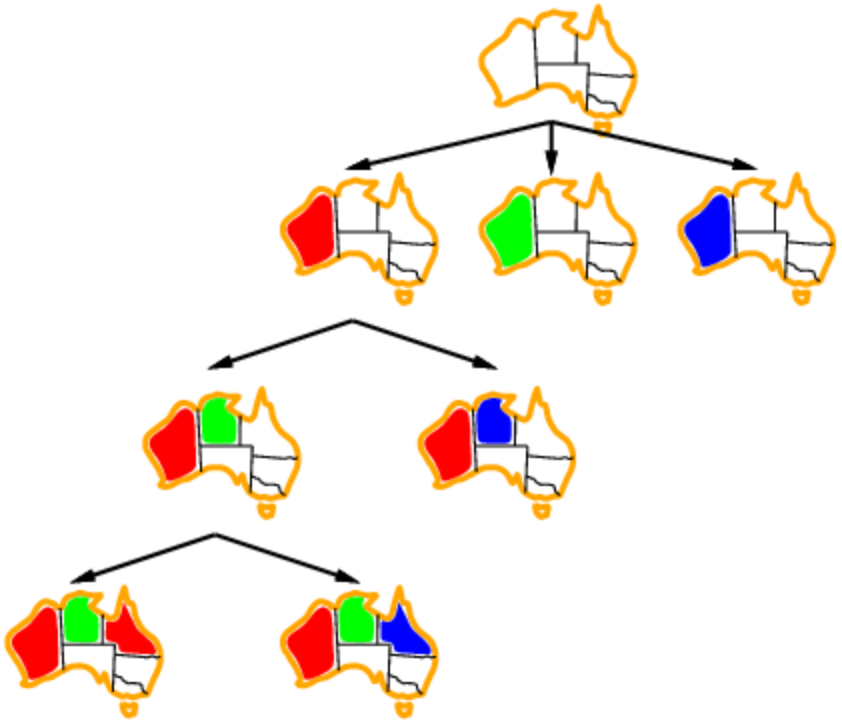
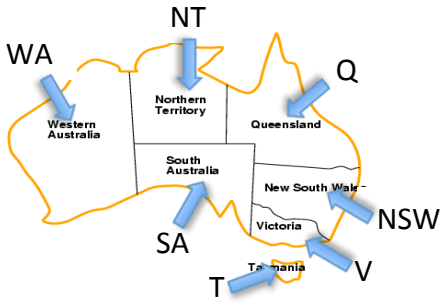


Illustration de *backtracking-search*





Amélioration de *backtracking-search*

- Sans heuristiques, l'algorithme est limité
  - il peut résoudre le problème de 25 reines

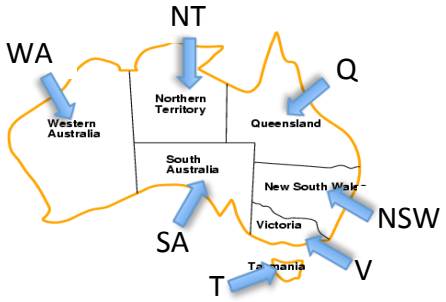
- Des **heuristiques générales** peuvent améliorer l'algorithme significativement :
  - choisir judicieusement la prochaine variable (VAR-NON-ASSIGNÉE)
  - choisir judicieusement la prochaine valeur à assigner (VALEURS-ORDONNÉES)
  - détecter les assignations conflictuelles et réduire les domaines (INFÉRENCE)

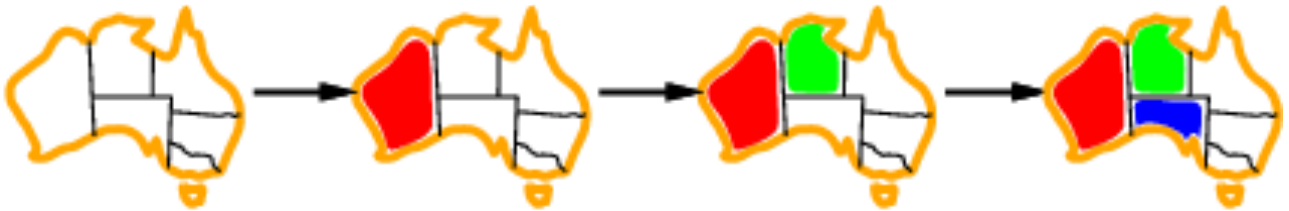
Choisir l'ordre d'assignation des variables

- 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*.

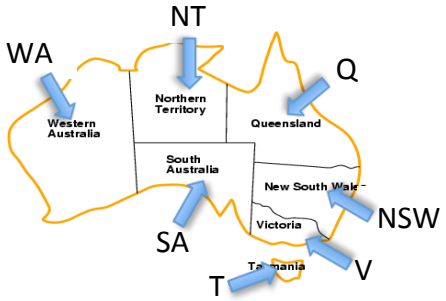
- Illustration :

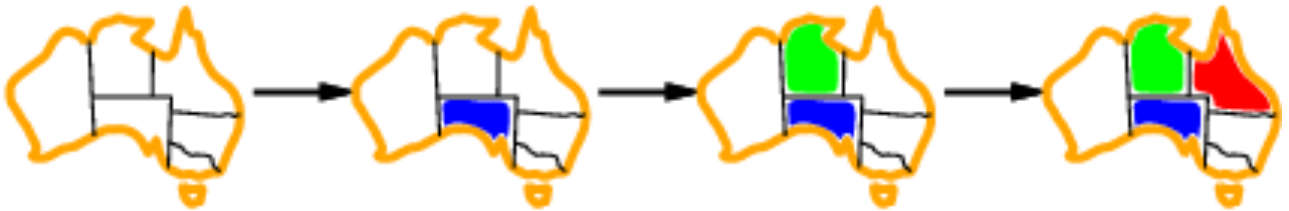




Choisir l'ordre d'assignation des variables

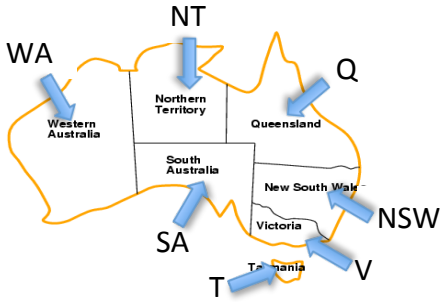
- Si le critère précédent donne des variables avec le même nombre de valeurs compatibles restantes :
  - choisir celle ayant le plus de contraintes impliquant des variables non encore assignées
  - appelée ***degree heuristic***.





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



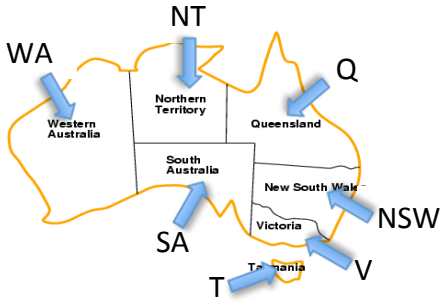
laisse une seule valeur pour *SA*

ne laisse aucune valeur pour *SA*

- Ces heuristiques permettent de résoudre un problème de 1000 reines

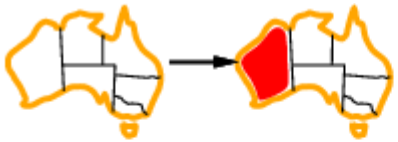
Détecter les assignations conflictuelles : algorithme *forward checking*

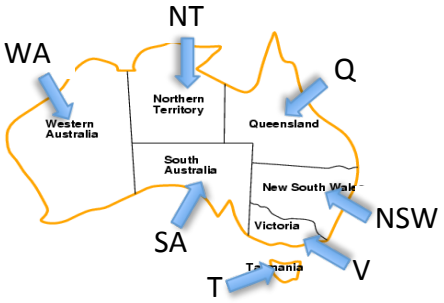
- L'idée de l'inférence *forward checking* (vérification anticipative) est de :
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité (conflit) lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide

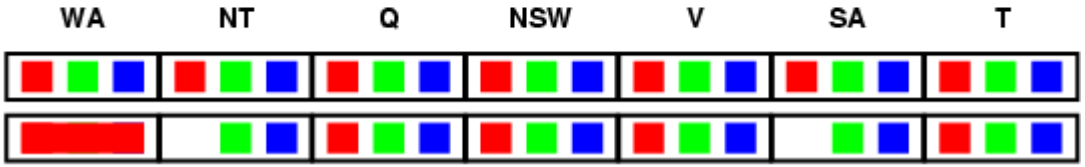


Détecter les assignations conflictuelles : algorithme *forward checking*

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

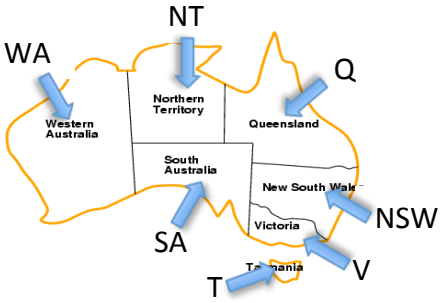






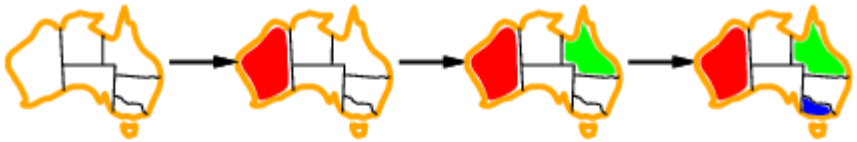
Détecter les assignations conflictuelles : algorithme *forward checking*

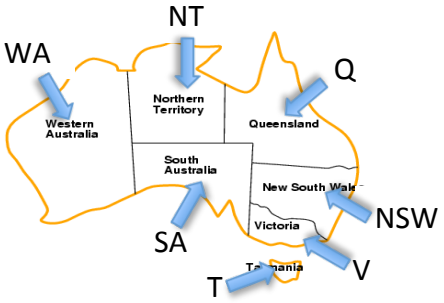
- L'idée de l'inférence *forward checking* (vérification anticipative) est de :
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité (conflit) lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide



Détecter les assignations conflictuelles : algorithme *forward checking*

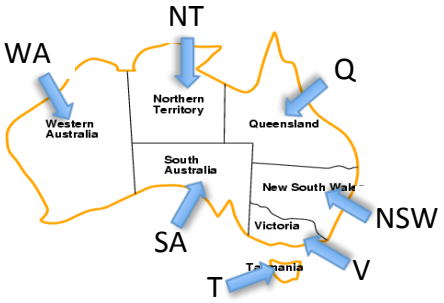
- L'idée de l'inférence *forward checking* (vérification anticipative) est de :
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité (conflit) lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide





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!
- L'inférence par propagation des contraintes permet de vérifier les contraintes localement

Algorithme *forward checking*

Algorithme FORWARD-CHECKING(X, *csp*)

1. pour chaque X_k dans VOISINS(X, *csp*)
   2. *changé*, *csp* = RÉVISER(X_k, X, *csp*)
   3. si *changé* et DOMAINE(X_k, *csp*) est vide, retourner (void, faux)
4. retourner (*csp*, vrai)

doit spécifier la variable impliquée

on suppose que *csp* est passé par copie

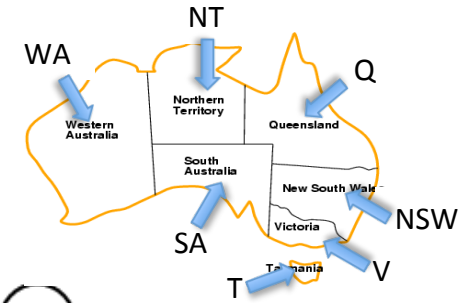
Algorithme RÉVISER(X_i, X_j, *csp*) // *réduit le domaine de X_i en fonction de celui de X_j*

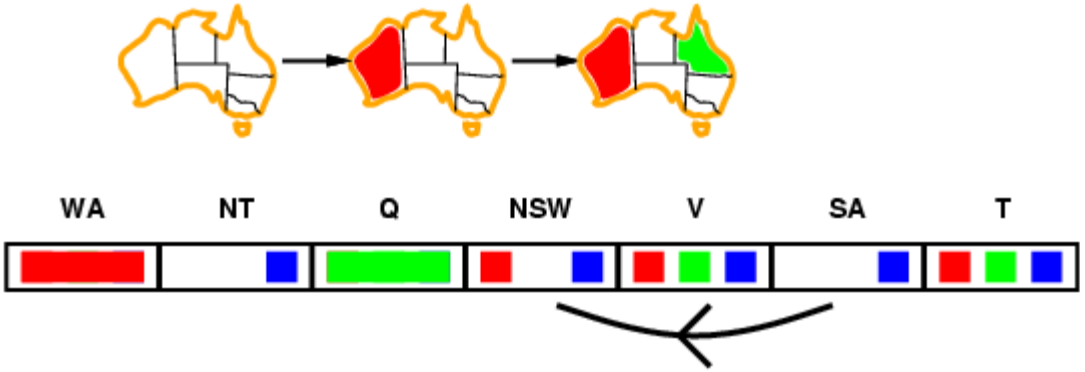
1. *changé* = faux
2. pour chaque x dans DOMAINE(X_i, *csp*)
   3. si aucun y dans DOMAINE(X_j, *csp*) satisfait contrainte entre X_i et X_j
      4. enlever x de DOMAINE(X_i, *csp*) // *ceci change la variable csp*
      5. *changé* = vrai
6. retourner (*changé*, *csp*)

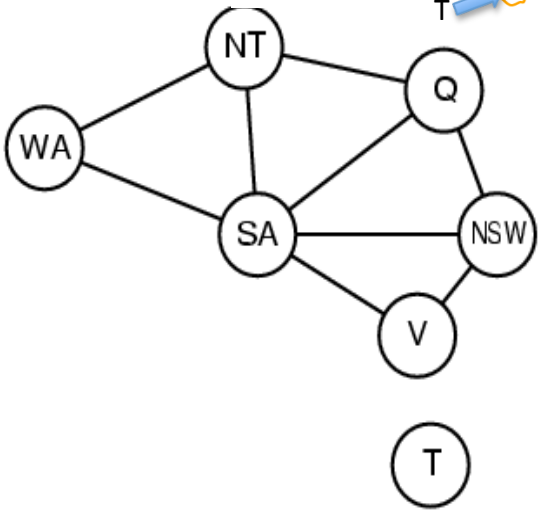
Arc consistency

- *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 $X \rightarrow Y$ est compatible si et seulement si
  - pour chaque valeur *x* de *X* il existe au moins une valeur permise *y* de *Y*

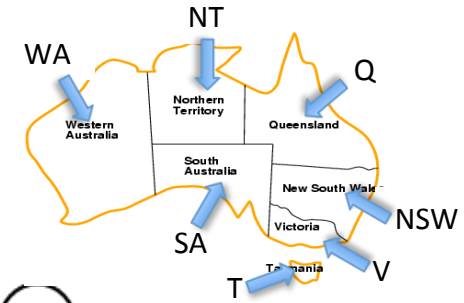




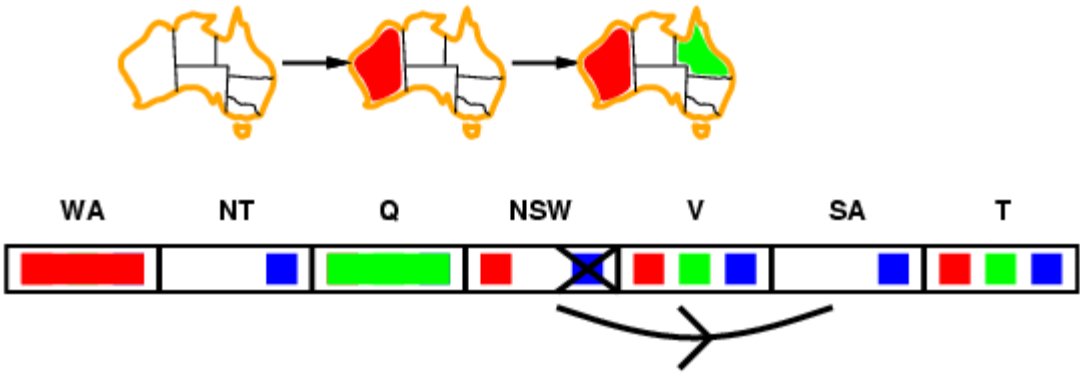


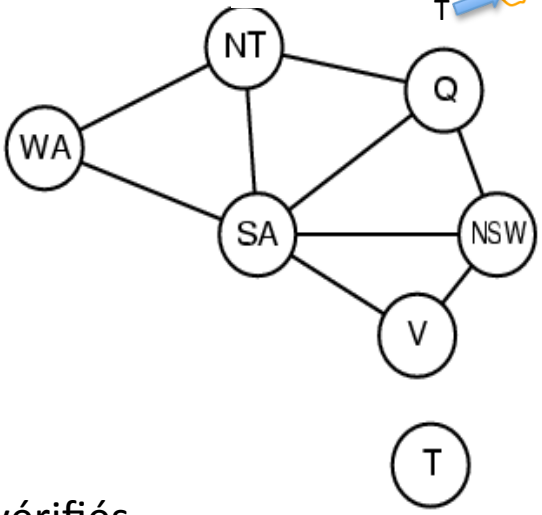
Arc consistency

- *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 $X \rightarrow Y$ est compatible si et seulement si
  - pour chaque valeur *x* de *X* il existe au moins une valeur permise *y* de *Y*



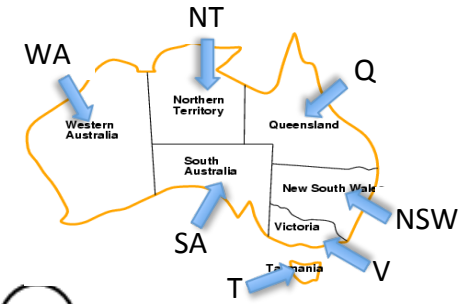


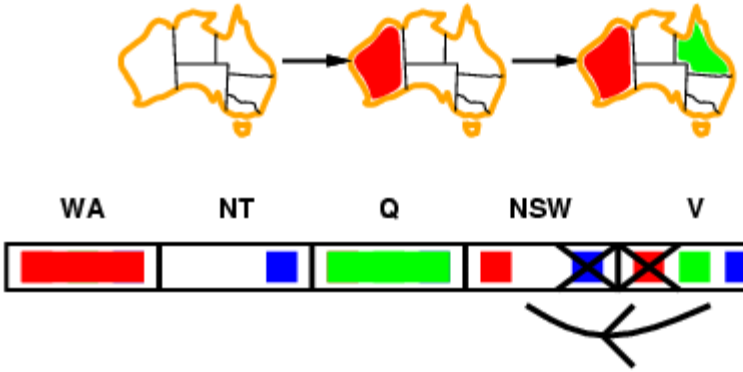
Si une variable perd une valeur, ses voisins doivent être revérifiés

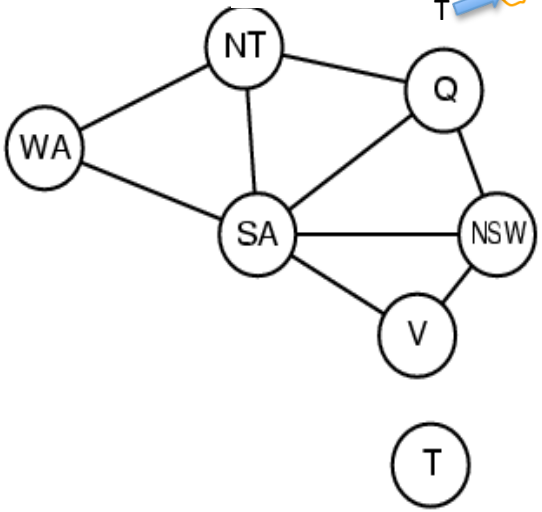
Arc consistency

- *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 $X \rightarrow Y$ est compatible si et seulement si
  - pour chaque valeur *x* de *X* il existe au moins une valeur permise *y* de *Y*



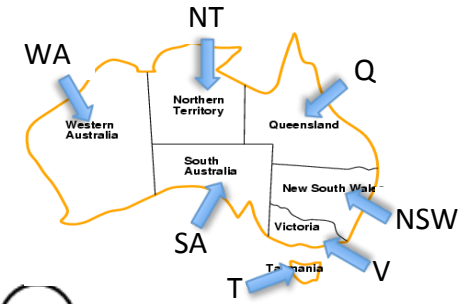


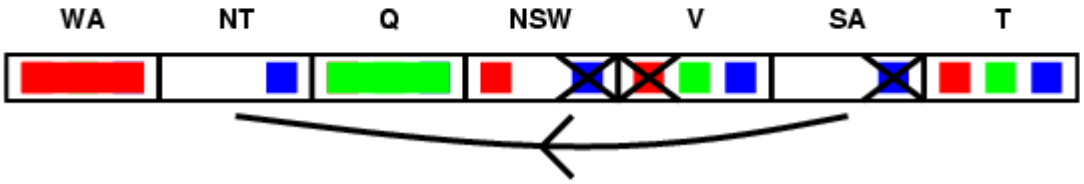


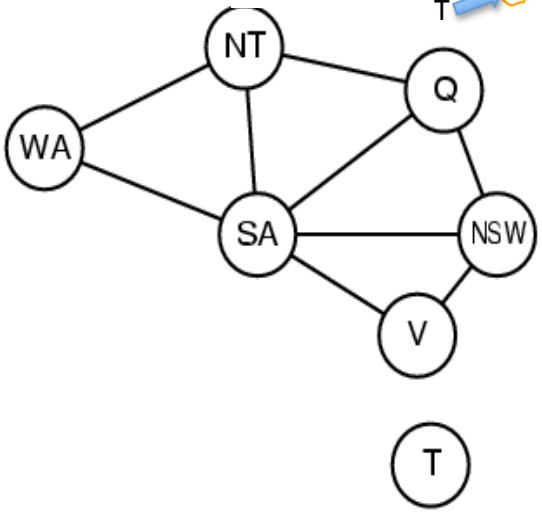
Arc consistency

- *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 $X \rightarrow Y$ est compatible si et seulement si
  - pour chaque valeur *x* de *X* il existe au moins une valeur permise *y* de *Y*







Algorithme *Arc consistency* (AC-3)

Algorithme AC-3(*csp*) // *retourne le CSP simplifié et un booléen vrai si pas de conflit*

1. *file_arcs* = ARCS-DU-CSP(*csp*)
2. tant que *file_arcs* n'est pas vide
   3. (X_i, X_j) = POP(*file_arc*) // *retire premier arc de la file*
   4. *changé*, *csp* = RÉVISER(X_i, X_j, *csp*) // *vérifie la compatibilité de l'arc*
   5. si *changé*
      6. si DOMAINE(X_i, *csp*) est vide, retourner (void, faux)
      7. pour chaque X_k dans VOISINS(X_i, *csp*)
         8. si $X_k \neq X_j$, ajouter (X_k, X_i) dans *file_arcs*
9. retourner (*csp*, vrai)

on suppose que *csp* est passé par copie

Algorithme RÉVISER(X_i, X_j, *csp*) // *réduit le domaine de X_i en fonction de celui de X_j*

1. *changé* = faux
2. pour chaque *x* dans DOMAINE(X_i, *csp*)
   3. si aucun y dans DOMAINE(X_j, *csp*) satisfait contrainte entre X_i et X_j
      4. enlever *x* de DOMAINE(X_i, *csp*) // *ceci change la variable csp*
      5. *changé* = vrai
6. retourner (*changé*, *csp*)

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($c\ d^3$) dans le pire cas, où c est le nombre de contraintes
  - complexité de RÉVISER : O(d^2)
  - on a O(c) arcs, qui peuvent être réinsérés dans la file O(d) fois par RÉVISER
  - RÉVISER peut donc être appelé O($c\ d$), pour une complexité globale de O($c\ d^3$)
- Une meilleure version en O($c\ d^2$) dans le pire cas existe : AC-4
  - par contre AC-3 est en moyenne plus efficace

- 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

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 objectif : on minimise le nombre de conflits
  - ressemble à *hill-climbing*, mais avec un peu de stochasticité

Algorithme *min-conflicts*

Algorithme MIN-CONFLICTS (*csp, nb_iterations*)

1. *assignations* = une assignation aléatoire complète (probablement pas compatible) de *csp*
2. pour *i* = 1 ... *nb_iterations*
    3. si *assignations* est compatible, retourner *assignations*
    4. *X* = variable choisie aléatoirement dans variables(csp)
    5. *v* = valeur dans DOMAINE(*X*, *csp*) satisfaisant le plus de contraintes de *X*
    6. assigner (*X* = *v*) dans *assignations*
7. retourner faux

- 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!)

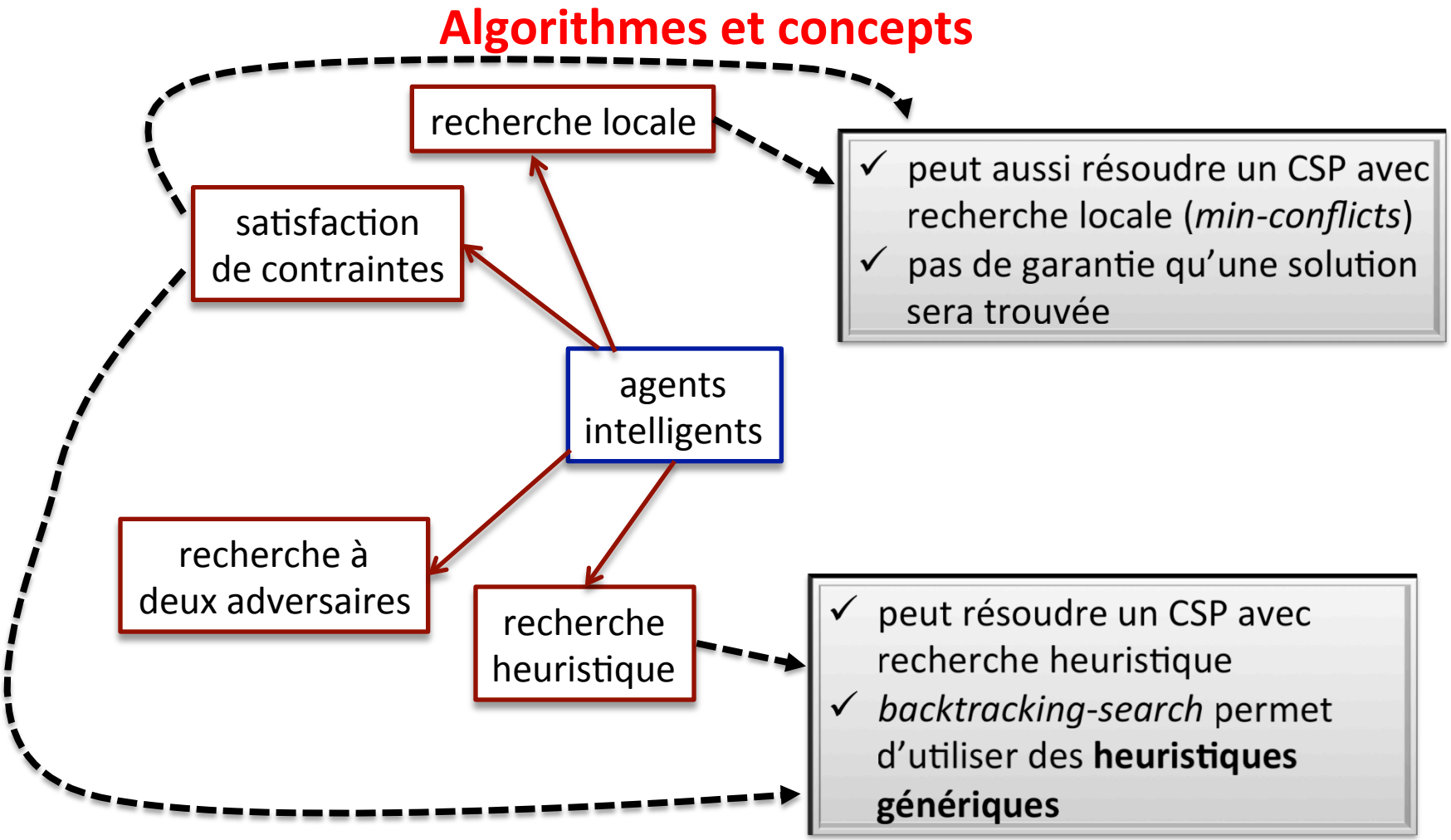
Types de problèmes CSP

- 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
- Voir le cours **ROP 317 – Programmation linéaire** pour en savoir plus sur le cas linéaire et continu

Applications

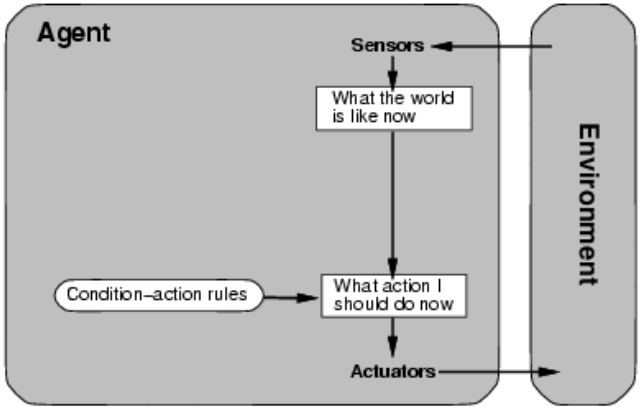
- Problèmes d'horaires (ex. : horaire des cours) :
  - dans ce cours, nous avons vu des méthodes simples, seulement pour des contraintes dures
  - la plupart des approches tiennent compte des contraintes souples
    - » http://www.springerlink.com/content/erylu61yx9tpj3hb/
    - » http://www.emn.fr/x-info/jussien/publications/cambazard-PATAT04.pdf
- 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.

Objectifs du cours

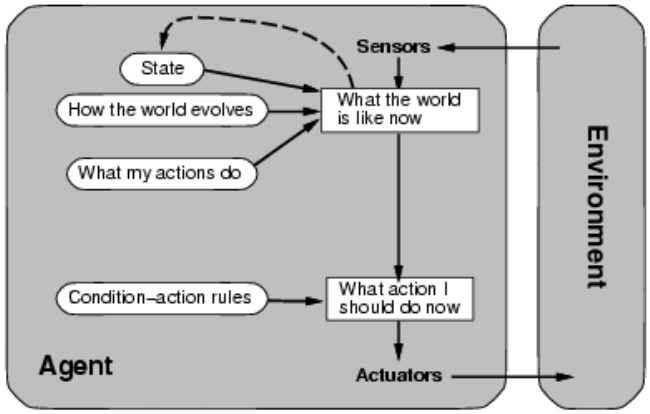


Satisfaction de contraintes : pour quel type d'agent?

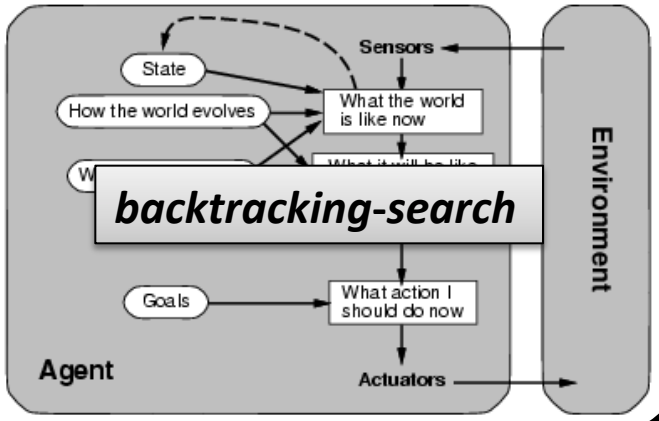
Simple reflex



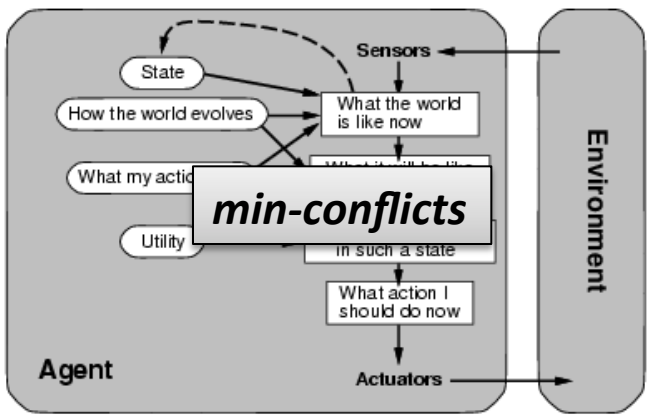
Model-based reflex



Goal-based



Utiliy-based



Conclusion

- Les problèmes CSP sont des problèmes de recherche dans un espace d'assignations 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, AC-3) détecte les incompatibilités locales
- Les méthodes les plus efficaces exploitent la structure du domaine
- Application surtout à des problèmes impliquant l'ordonnancement de tâche

Vous devriez être capable de...

- Formuler un problème sous forme d'un problème de satisfaction de contraintes (variables, domaines, contraintes)
- Simuler l'algorithme *backtracking-search*
- Connaître les différentes façon de l'améliorer
  - ordonnancement des variables
  - ordonnancement des valeurs
  - inférence (*forward checking*, AC-3)
- Savoir simuler *forward checking* et AC-3
- Décrire comment résoudre un problème de satisfaction de contraintes avec un algorithme de recherche locale