IFT 615 – Intelligence Artificielle

Été 2022

Recherche locale par la satisfaction de contraintes

Professeur: Froduald Kabanza

Assistants: D'Jeff Nkashama & Jean-Charles Verdier

Université de Sherbrooke

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 façons 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

Sujets couverts

- Modèle général des problèmes de satisfaction de contraintes
- Algorithme *Backtracking-search*.
- Algorithme *AC-3*.
- Min-Conflicts
- Applications.

Problème de satisfaction de contraintes

- La résolution de problèmes de satisfaction de contraintes est une autre façon d'implémenter recherche heuristique locale

- La structure interne des états (nœuds) 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

Exemple 1

- Soit le problème 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)

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* permises.
  - 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.
  - $\{X_i=v_i, X_n=v_1, \ldots\}$.
- Une assignation qui ne viole aucune contrainte est dite *consistante* ou *légale*.
- Une *assignation* est *complète* si elle concerne toutes les variables.
- Une *solution* à un problème CSP est une *assignation complète* et *consistante*.
- Parfois, la solution doit en plus *maximiser une fonction objective* donnée.

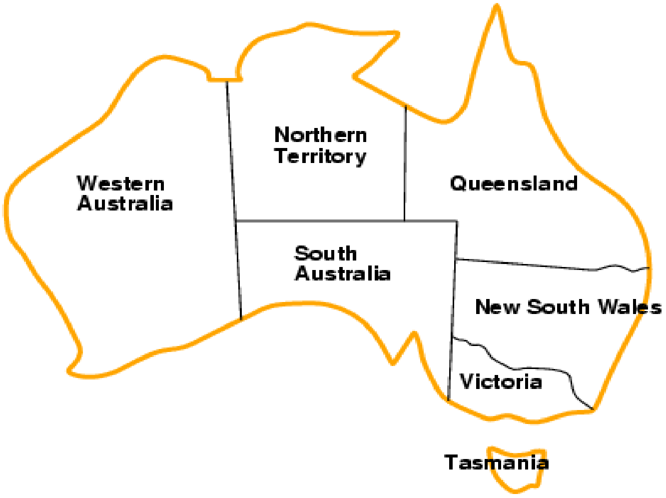
Exemple 2 : Colorier une carte

- On vous donne une carte de l'Australie :

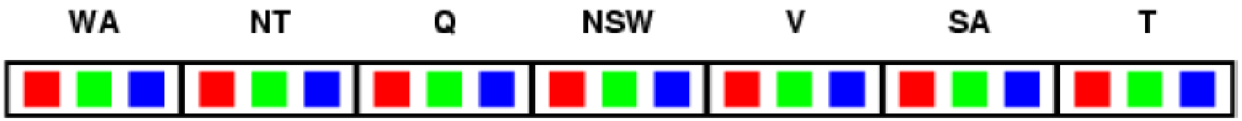
- Et on vous demande d'utiliser seulement trois couleurs (*rouge*, *vert* et *bleu*) de sorte que deux états frontaliers n'aient jamais les mêmes couleurs.
- On peut facilement trouver une solution à ce problème en le formulant comme un problème CSP et en utilisant des algorithmes généraux pour CSP.

Exemple 2: Colorier une carte

- Formulation du problème CSP :



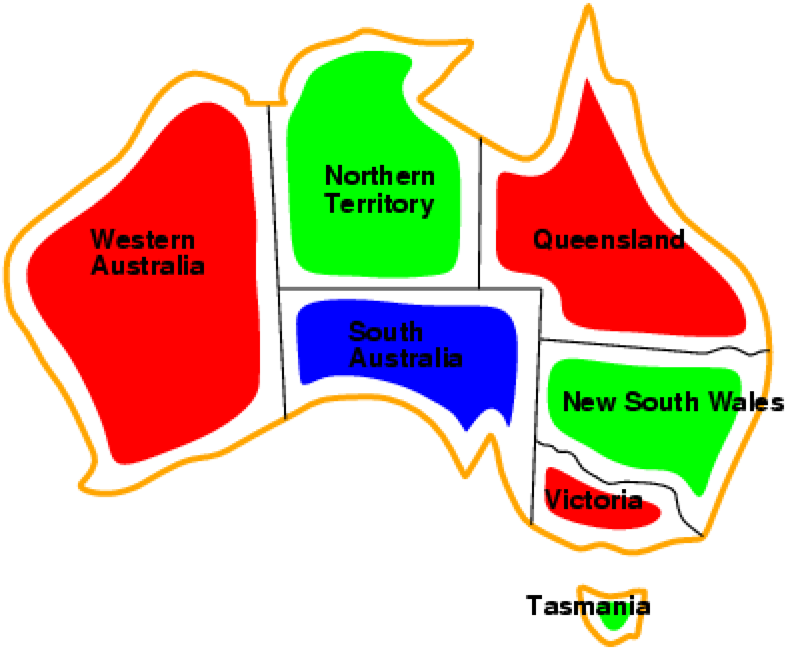
- Les variables sont les états : *V = { WA, NT, Q, NSW, V, SA, T }*
- Le domaine de chaque variable est l'ensemble des trois couleurs : *{R, G, B}*



- Contraintes : *Les régions frontalières doivent avoir des couleurs différentes*
  - *WA≠ NT, …, NT≠ Q, …*

Exemple 2: Colorier une carte

- Solution :

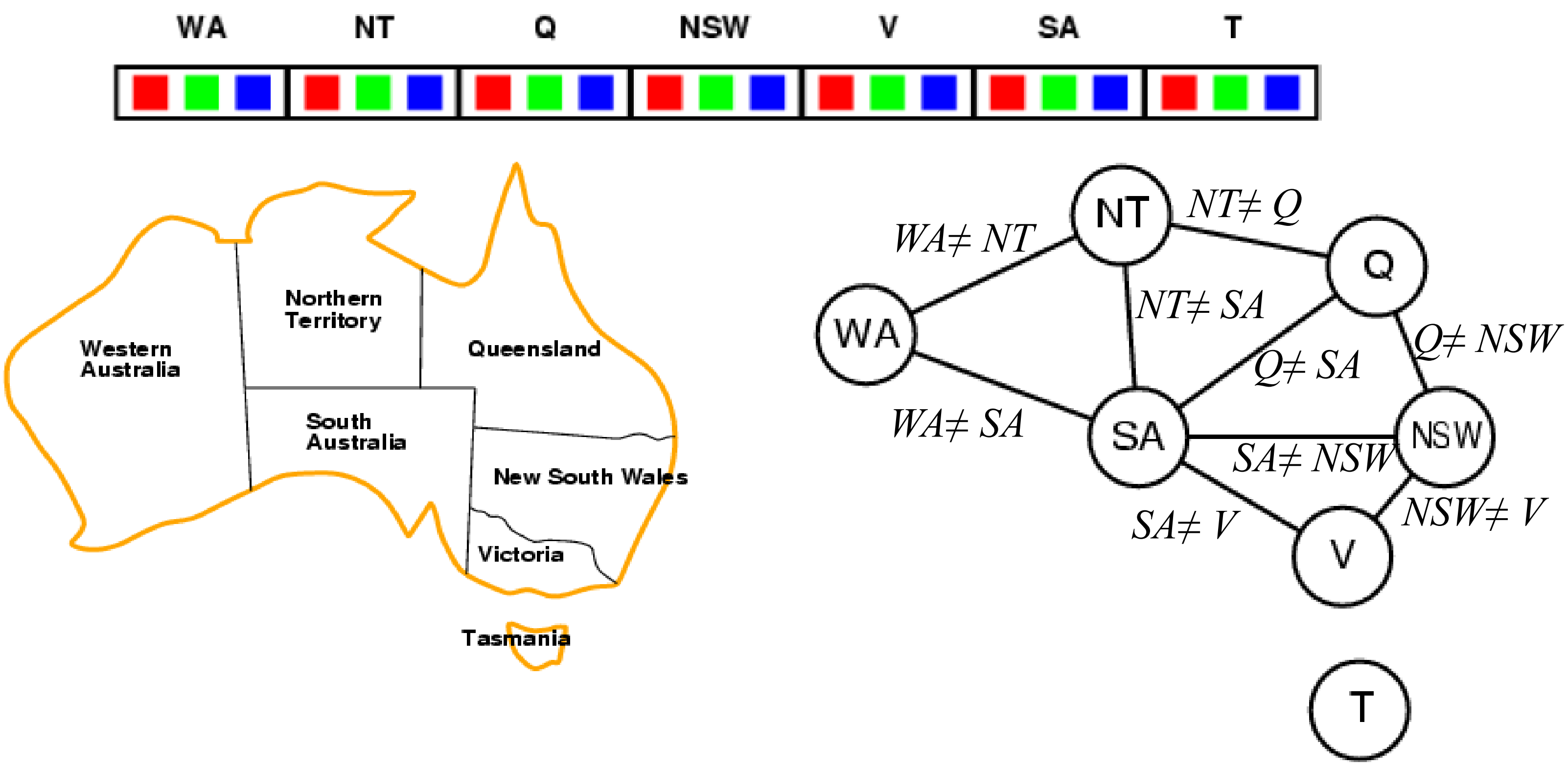


$$\{ WA = R, NT = G, Q = R, NSW = G, V = R, SA = B, T = G \}$$

Graphe de contraintes

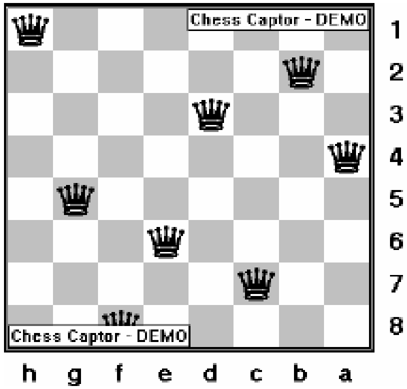
- Pour des problèmes avec des **contraintes binaires** (c-à-d., entre deux variables), on peut visualiser le problème CSP par un **graphe de contraintes**.

- 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 2 : *N-Queens*

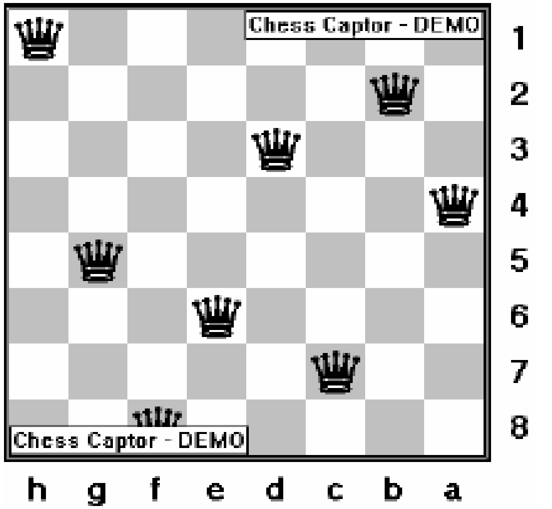
- Positionner *N* reines sur un échiquier de sorte qu'aucune d'entre elles n'est en position d'attaquer une autre.
- Exemple avec 8 reines (*8-Queens*)



- Une reine peut attaquer une autre si elles sont toutes les deux sur: la même ligne, la même colonne, ou la même diagonale.

Exemple 3 : *N-Queens*

- Modélisation comme problème CSP:



- Variables : Q_1 … Q_n correspondant aux colonnes *1, …, N*.
- Domaines : chaque variable a le domaine de valeurs {*1, …., N*}

  La colonne *i* a la valeur *k* si la reine (*Queen*) dans la colonne *i* est dans la rangée *k*.

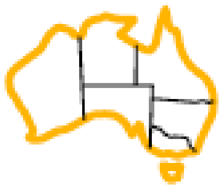
- Contraintes : Pas deux reines sur même rangée: $Qi \neq Qj$

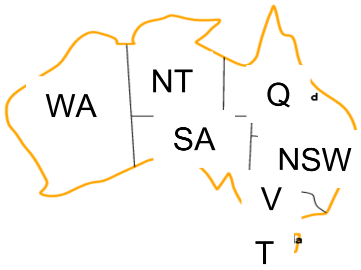
  Pas deux reines sur même diagonale : $|i-j| \neq |Qi-Qj|$

Algorithme Depth-First-Search Naïve pour CSP

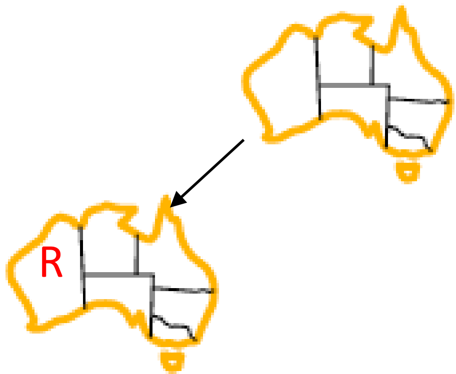
- On pourrait être tenté d'utiliser la recherche dans un graphe (Algorithme rechercheDansGraphe) ou un *depth-first-search naïf* avec les paramètres suivants:
  - Un état est une assignation.
  - État initial : assignation vide { }
  - Fonction successeur : assigne une valeur à une variable non encore assignée, en respectant les contraintes.
  - But : Assignation complète et consistante.
- Comme la solution doit être complète, elle apparaît à une profondeur *n,* si nous avons *n* variables.
- Cependant, ici le chemin à la solution est sans importance.
  - On peut travailler avec des états qui sont des assignations complètes (consistantes ou non).
  - On peut utiliser une méthode de recherche locale (*hill-climbing*, etc.)

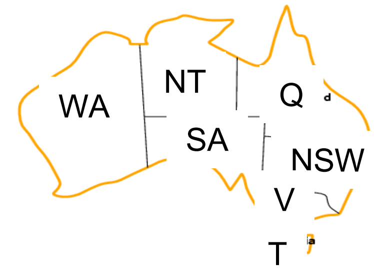
Recherche en profondeur naïve



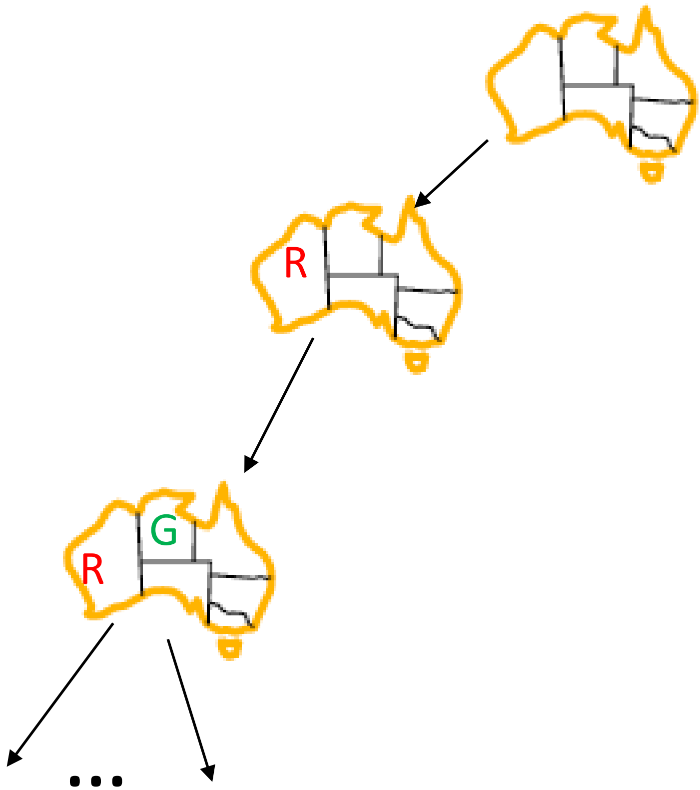


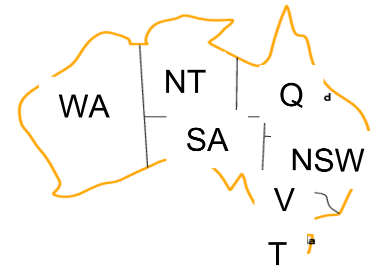
Recherche en profondeur naïve



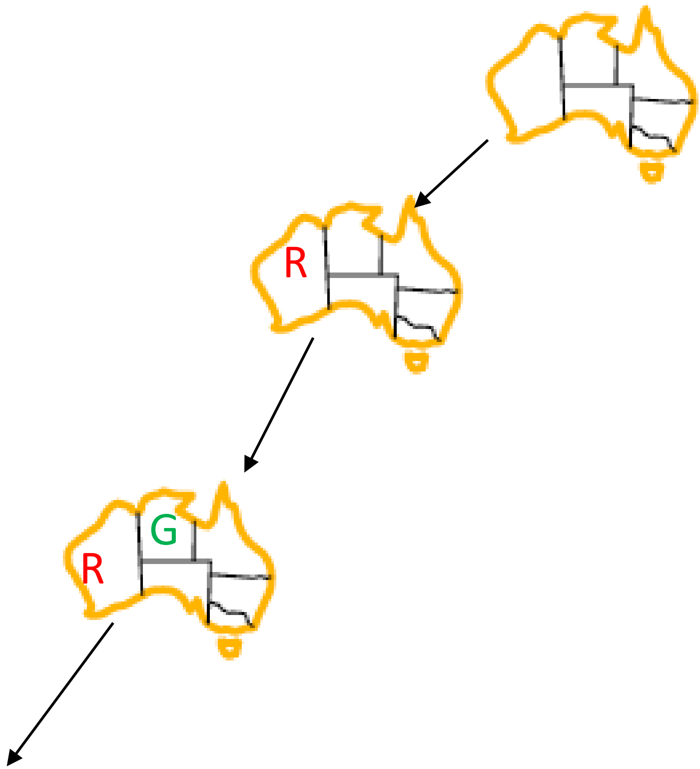


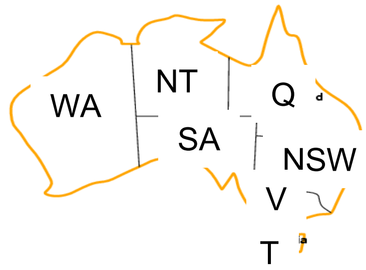
Recherche en profondeur naïve



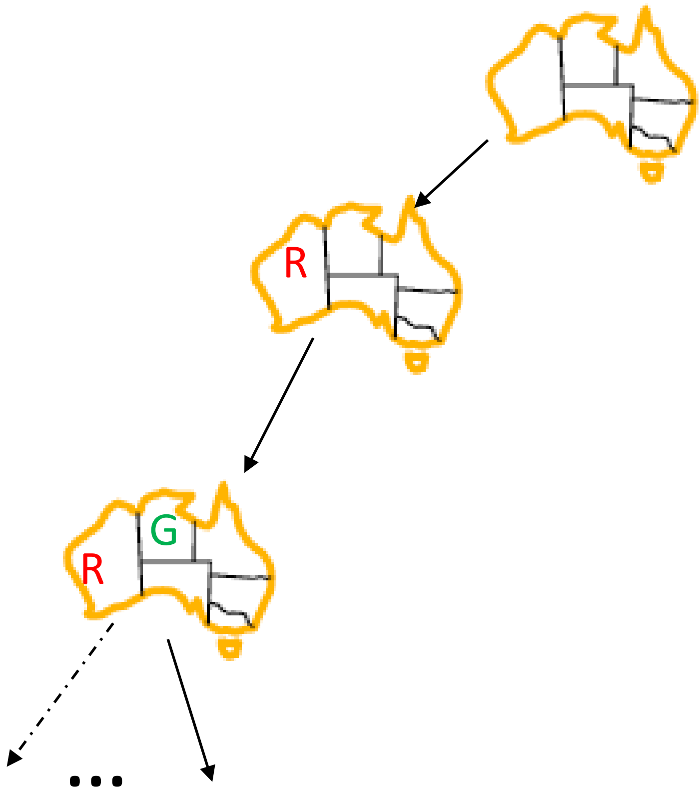


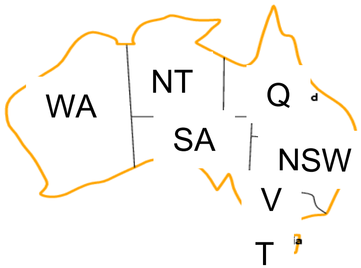
Recherche en profondeur naïve



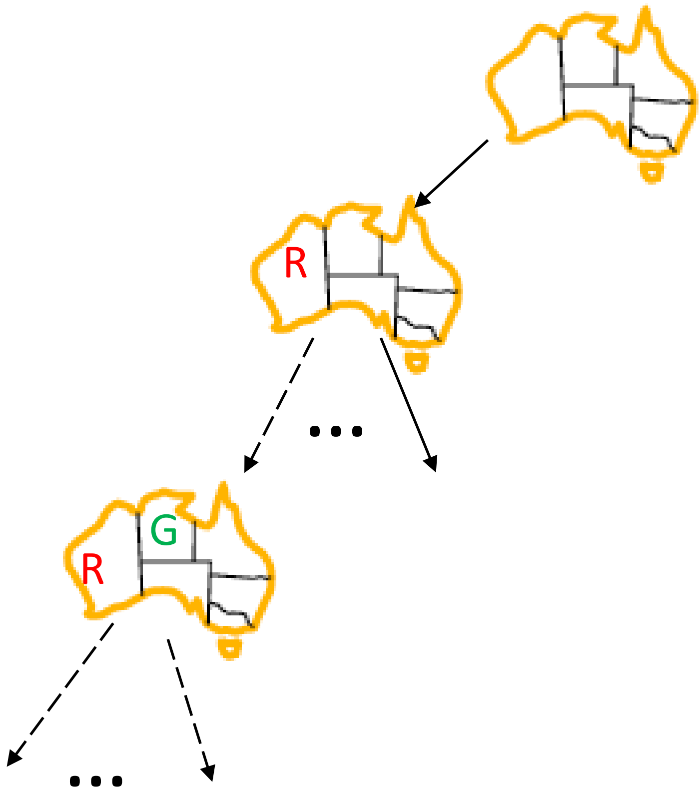


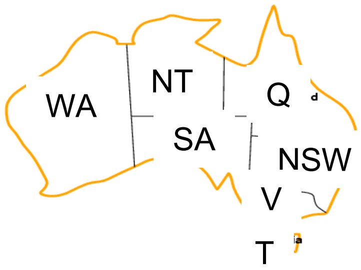
Recherche en profondeur naïve



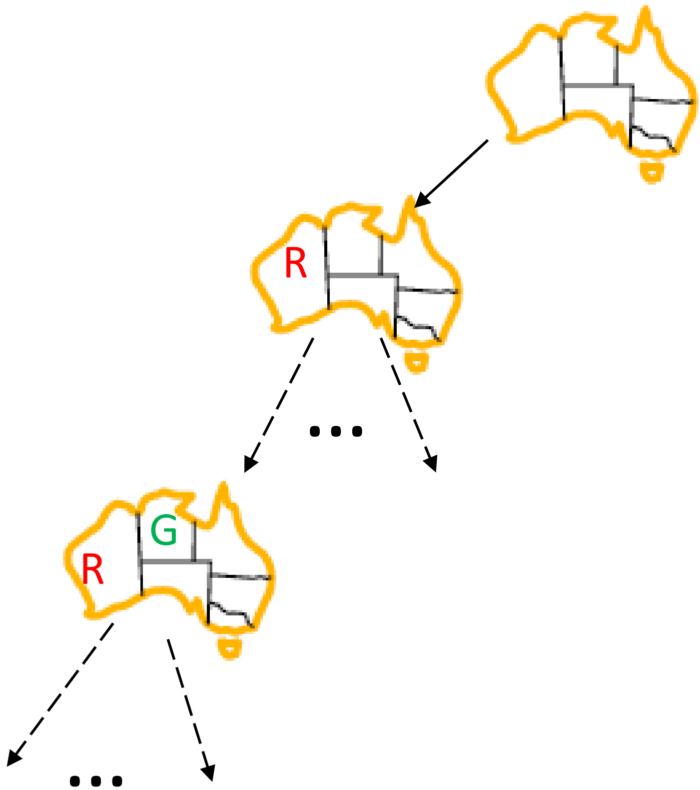


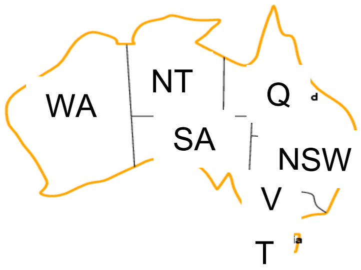
Recherche en profondeur naïve



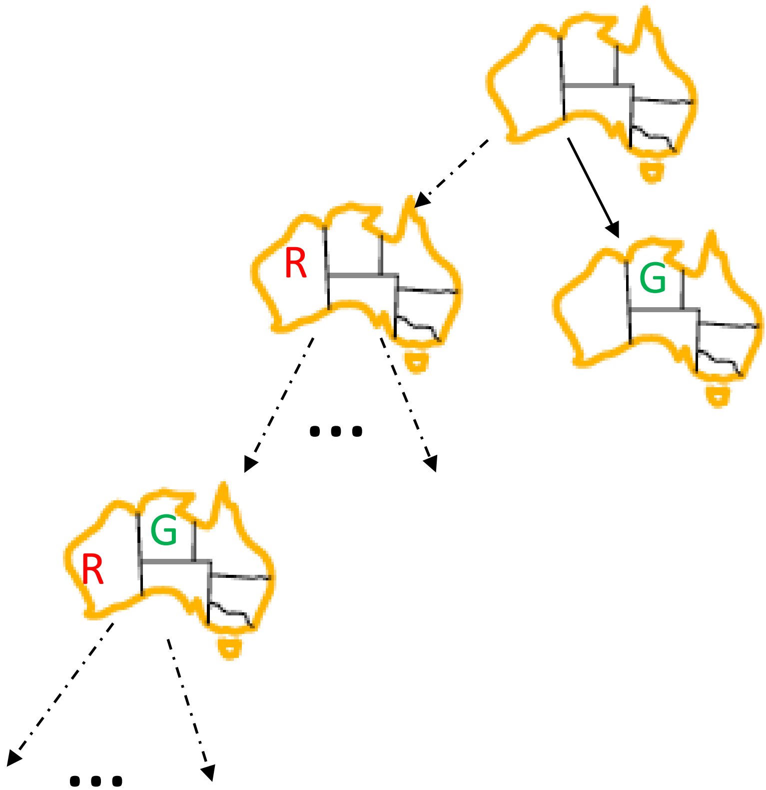


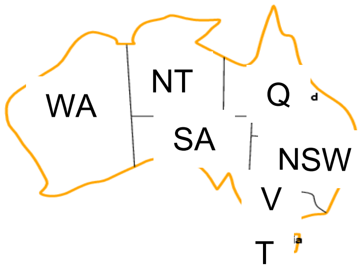
Recherche en profondeur naïve



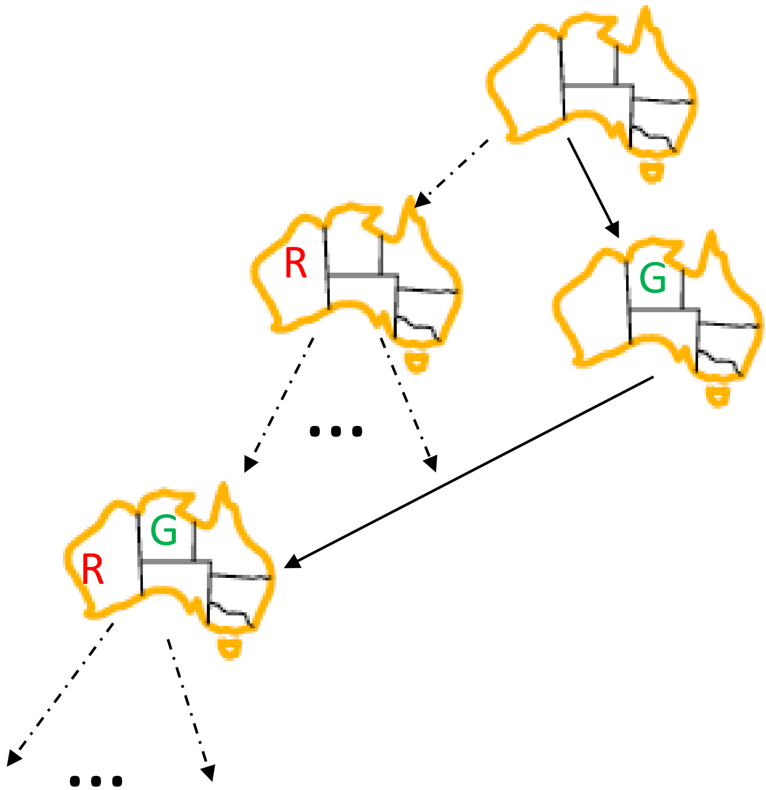


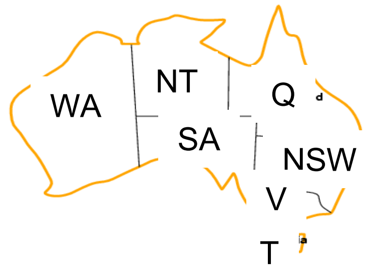
Recherche en profondeur naïve





Recherche en profondeur naïve





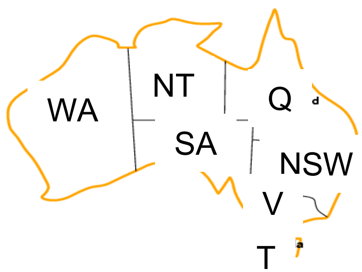
Limitations de l'approche précédente

- Pour calculer la taille de l'espace de recherche :
  - 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 :
  - *WA=R* suivi de *NT=G* est équivalent à *NT=G* suivi de *WA=R*

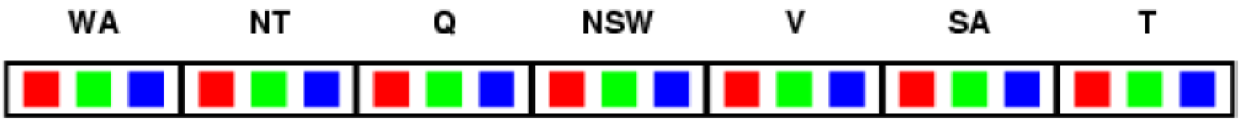
Backtracking Search

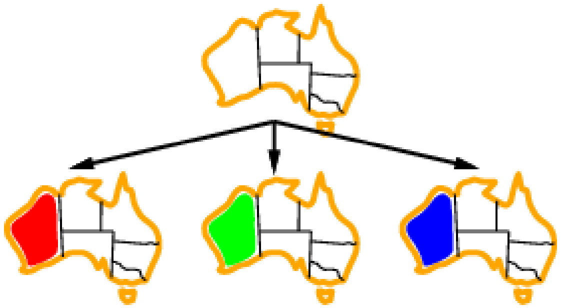
- Comme une recherche en profondeur, mais on tient compte de la commutativité
  - le nombre de nœuds générés est d^n, au lieu $n!*d^n$
- **Idée :** considérer **une seule variable** à assigner à chaque niveau et **reculer (*backtrack*) lorsqu'aucune assignation compatible** n'est pas possible
- Le résultat est ***backtracking-search*** : c'est l'algorithme de base pour résoudre les problèmes CSP

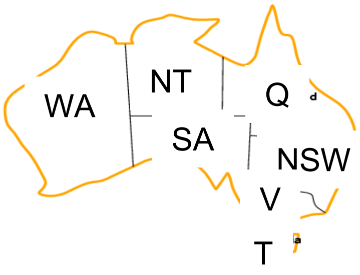
Backtracking Search



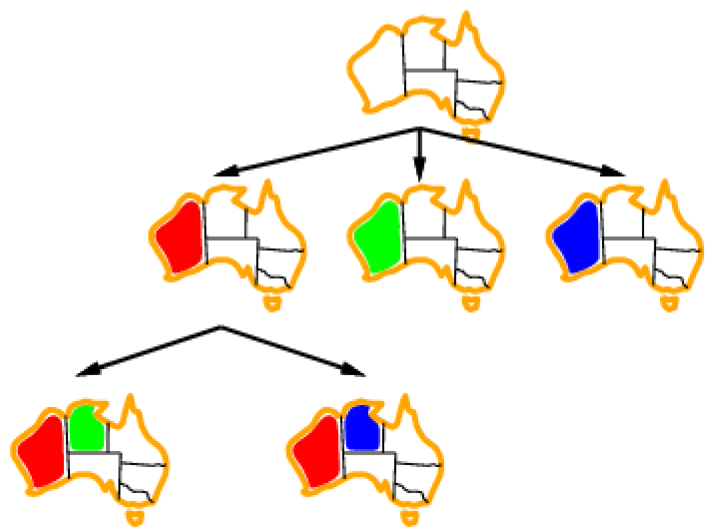
Backtracking Search

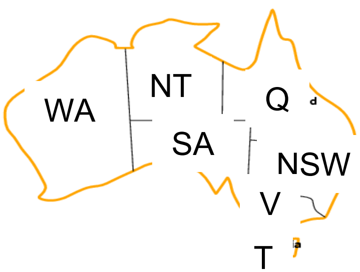




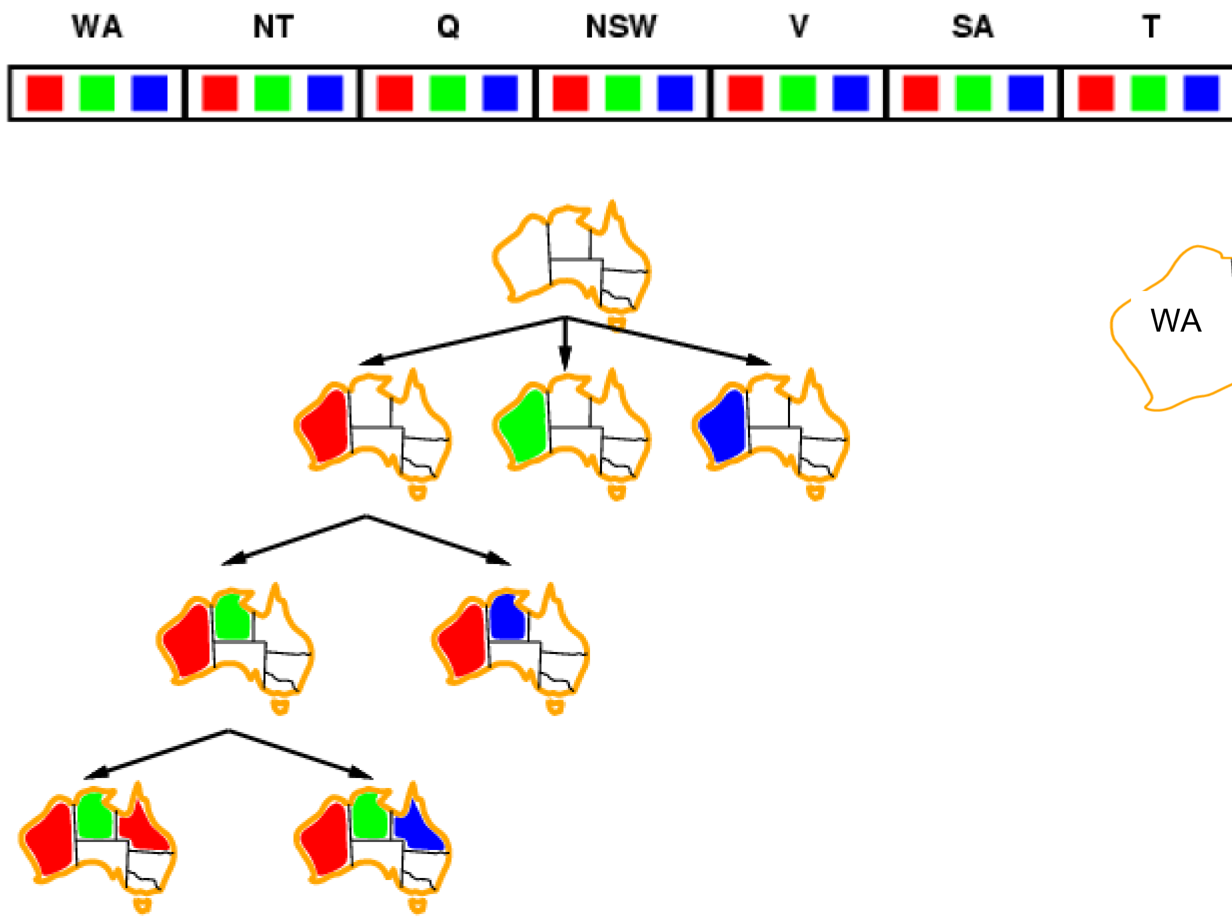


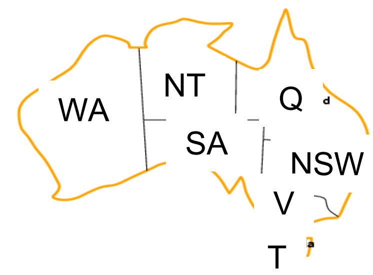
Backtracking-Search





Backtracking Search





Backtracking Search (page 215)

```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return BACKTRACK({} , csp)
function BACKTRACK(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(var, assignment, 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 ← INFERENCES(csp, var, value) // e.g., AC-3
            if inferences ≠ failure then
                add inferences to assignment
            result ← BACTRACK (assignment, csp)
                if result ≠ failure  then return result
        remove {var=value} and inferences from assignment
    return failure
```

Amélioration de backtracking-search

- Sans heuristiques, l'algorithme est limité.

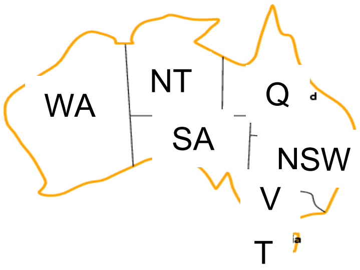
- Des heuristiques générales peuvent améliorer l'algorithme significativement :

  - Choisir judicieusement la prochaine variable:
    - » SELECT-UNASSIGNED-VARIABLE
  - Choisir judicieusement la prochaine valeur à assigner:
    - » ORDER-DOMAIN-VALUES

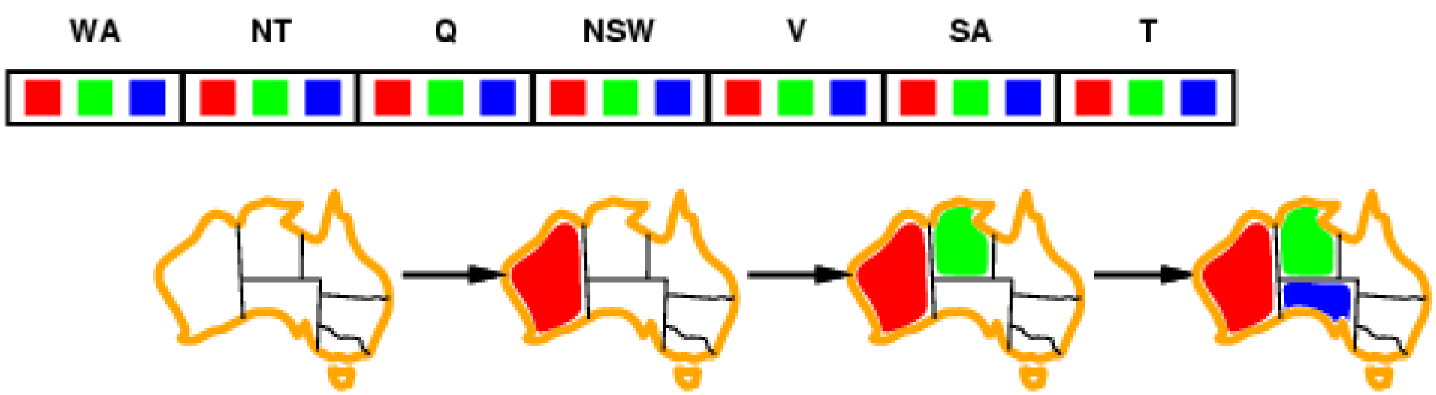
  - Faire des inférences pour détecter plus tôt les assignations conflictuels:
    - » INFERENCES

Choisir la prochaine variable

- À chaque étape, choisir la variable avec le moins de valeurs consistantes restantes.
  - C-à-d., la variable « posant le plus de restrictions ».
  - Appelé: ***Minimum RemainingValue (MRV)*** *Heuristic*
    ou ***Most Constrained Variable (MCV)*** *Heuristic.*

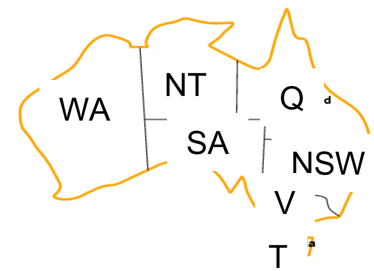


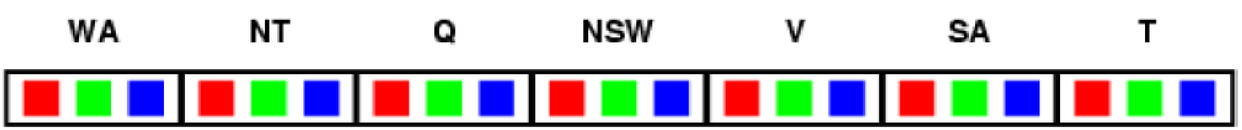
- Illustration:

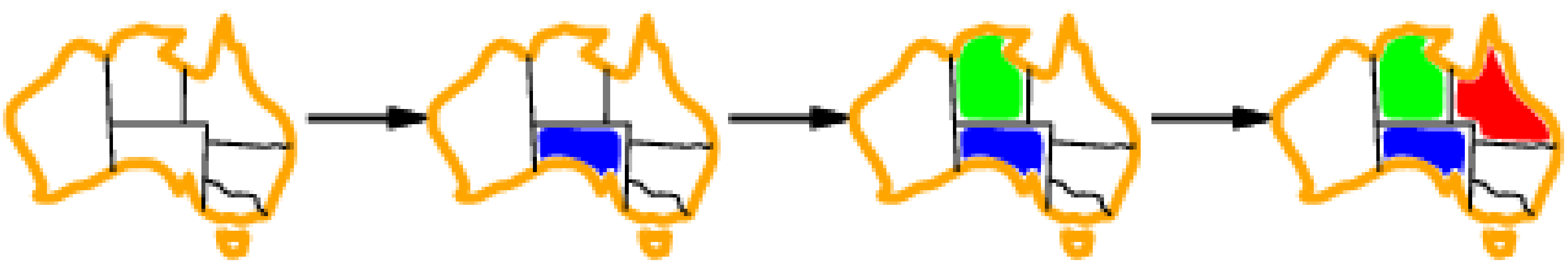


Choisir la prochaine variable

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

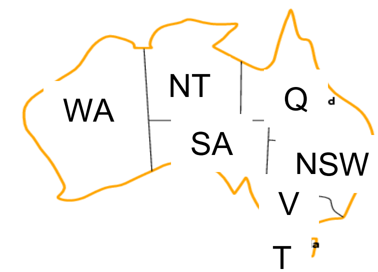


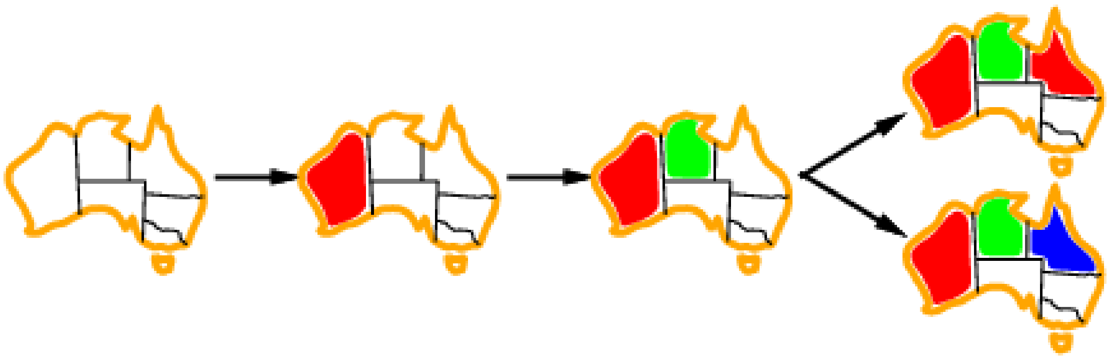




Choisir la prochaine valeur

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





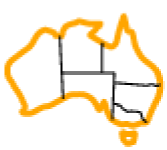
Laisse une seule valeur pour SA

Ne laisse aucune valeur pour SA

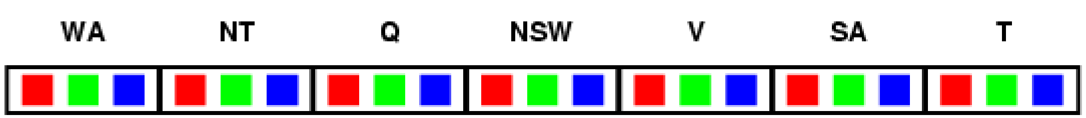
Forward-Checking Inference

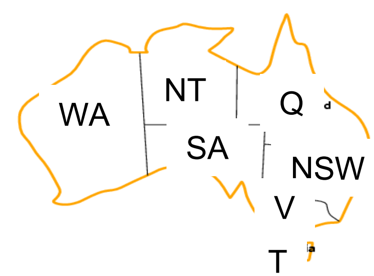
- L'idée de *forward-checking* *(vérification anticipative)* est :
  - vérifier les valeurs compatibles des variables non encore assignées
  - terminer la récursivité (conflit) lorsqu'une variable (non encore assignée) a son ensemble de valeurs compatibles qui devient vide

- Exemple



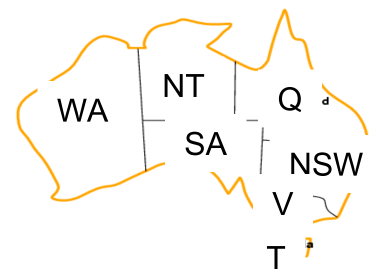
Domaines initiaux

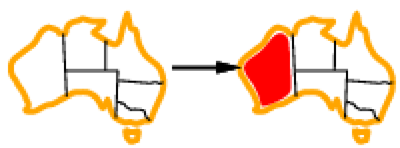




Algorithme *Forward checking*

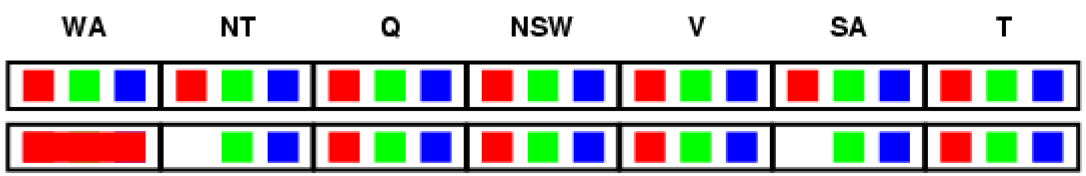
- Supposons que l'on choisisse au départ la variable WA (première étape de la récursivité de *backtracking-search*). Considérons l'assignation WA=Rouge. On voit ici le résultat de *forward-checking*.





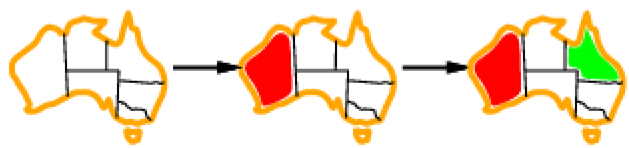
Domaines initiaux

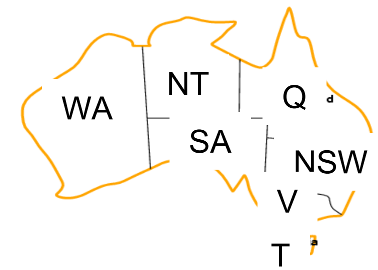
Après WA=Red



Algorithme *Forward checking*

- Supposons maintenant que l'on choisisse la variable Q à la prochaine étape de la récursivité de *backtracking-search*. Considérons l'assignation Q=Vert. On voit ici le résultat de *forward-checking*.

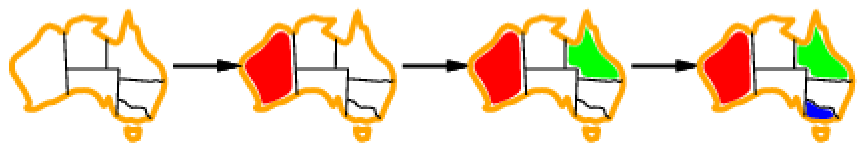


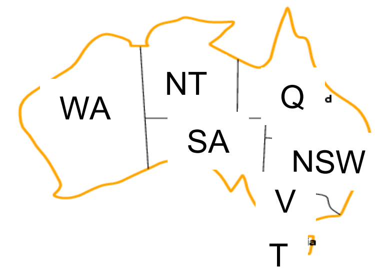


Domaines initiaux
Après WA=Red
Après Q=Green

Algorithme *Forward checking*

- Supposons maintenant que l’on choisisse la variable V à la prochaine étape de la récursivité de *backtracking-search*. Considérons l’assignation V=Bleu. On voit ici le résultat de *forward-checking*.

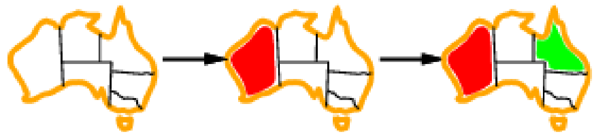




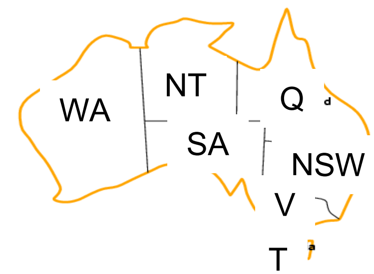
Domaines initiaux
Après WA=Red
Après Q=Green
Après V=Blue

Propagation de contraintes

- *Forward checking* propage l'information d'une variables assignée vers les variables en contraintes avec elle, mais ne propage pas l'effet des modifications de ces dernières.



Domaines initiaux
Après WA=Red
Après Q=Green



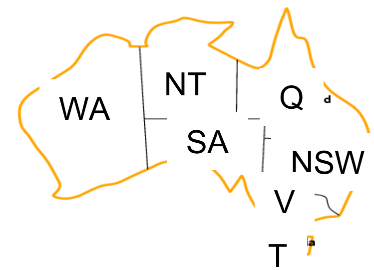
- Revenons à l'étape de backtracking-search, après que nous ayons choisi la variable Q et assigné la valeur 'Green'.
  - On voit ici le résultat de forward-checking
  - Forward-checking ne propage pas la modification du domaine SA vers NT pour constater que NT et SA ne peuvent pas être en bleu ensemble!
- La propagation des contraintes permet de vérifier ce type de conflits dans les assignations de variables.

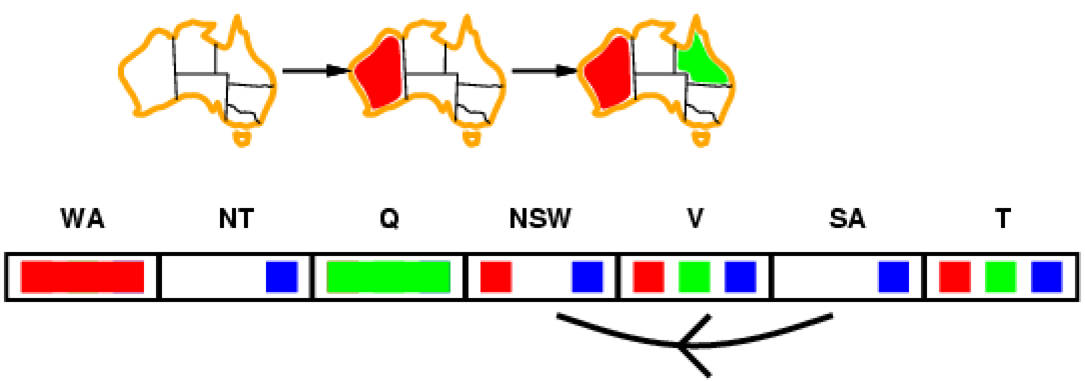
Arc consistency

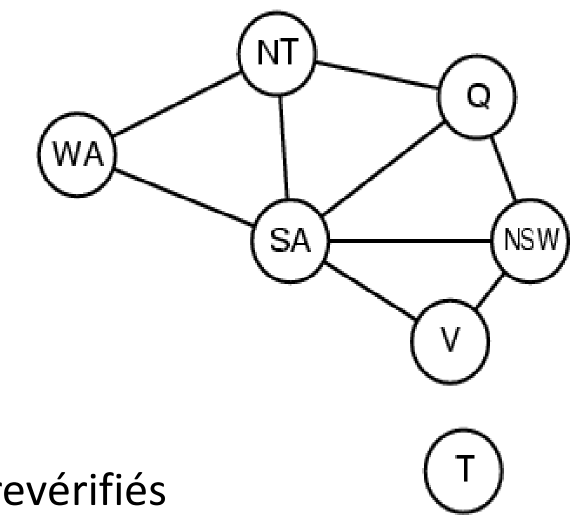
- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - Vérifie la consistance entre les arcs.
  - C-à-d., la consistance des contraintes entre deux variables.

- *L'arc X →Y* est consistante si et seulement si

  Pour chaque valeur *x* de *X* il existe au moins une





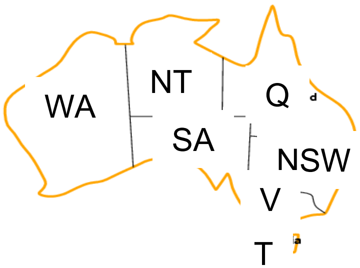


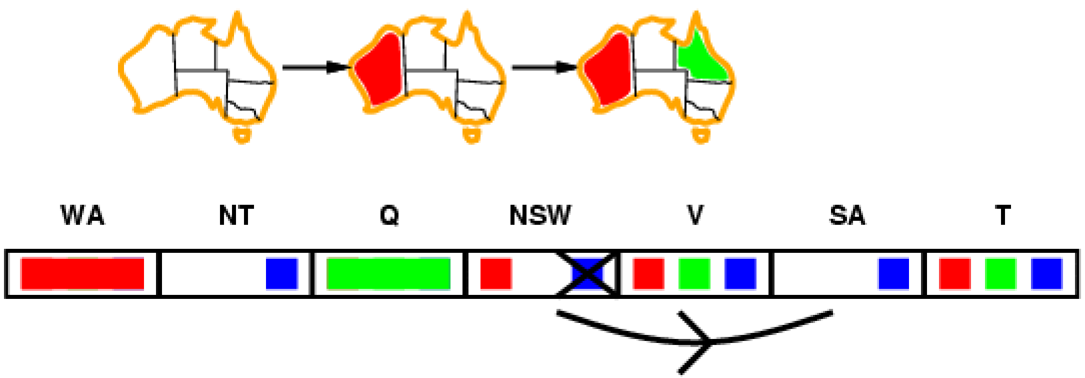
Si une variable perd une valeur, ses voisins doivent être revérifiés

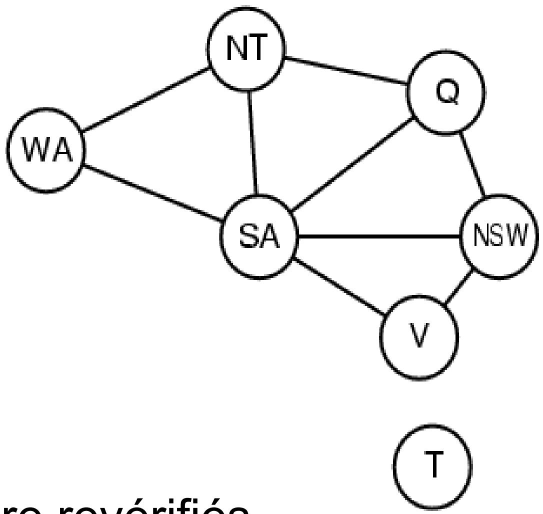
Arc consistency

- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - Vérifie la consistance entre les arcs.
  - C-à-d., la consistance des contraintes entre deux variables.

- *L'arc X →Y* est consistante si et seulement si
  Pour chaque valeur *x* de *X* il existe au moins une



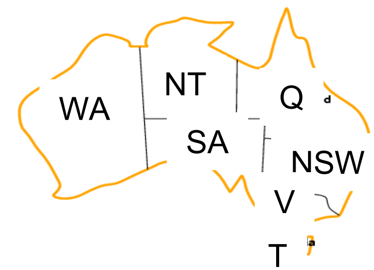




Si une variable perd une valeur, ses voisins doivent être revérifiés.

Arc consistency

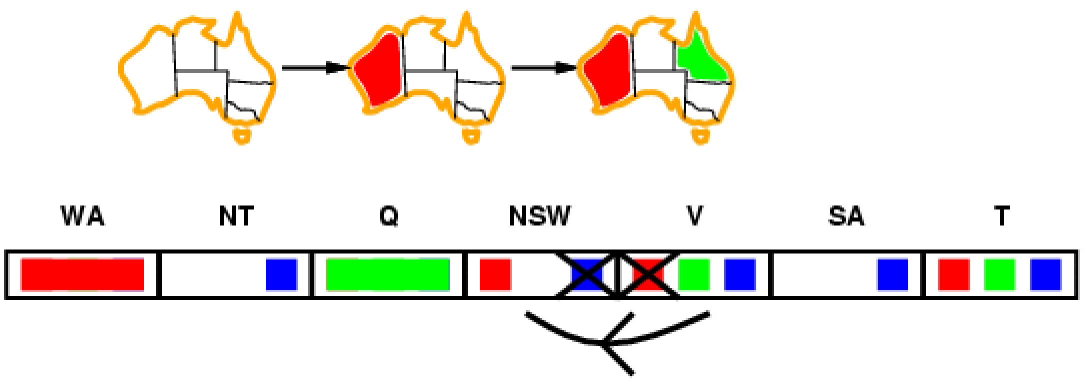
- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - Vérifie la consistance entre les arcs.
  - C-à-d., la consistance des contraintes entre deux variables.

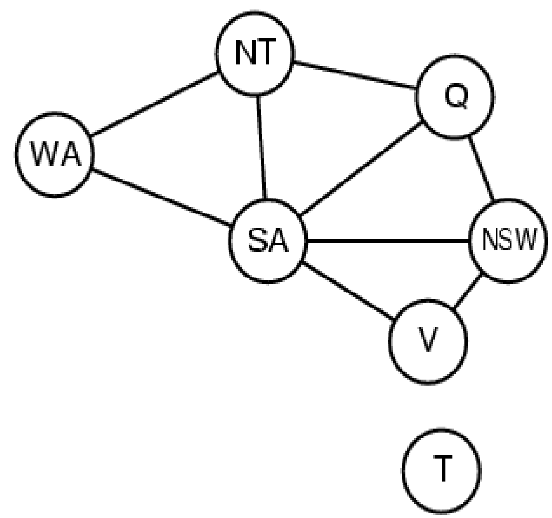


- *L'arc X →Y* est consistante si et seulement si

  Pour chaque valeur *x* de *X* il existe au moins une

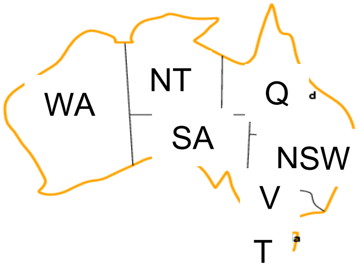
  consistante avec x.





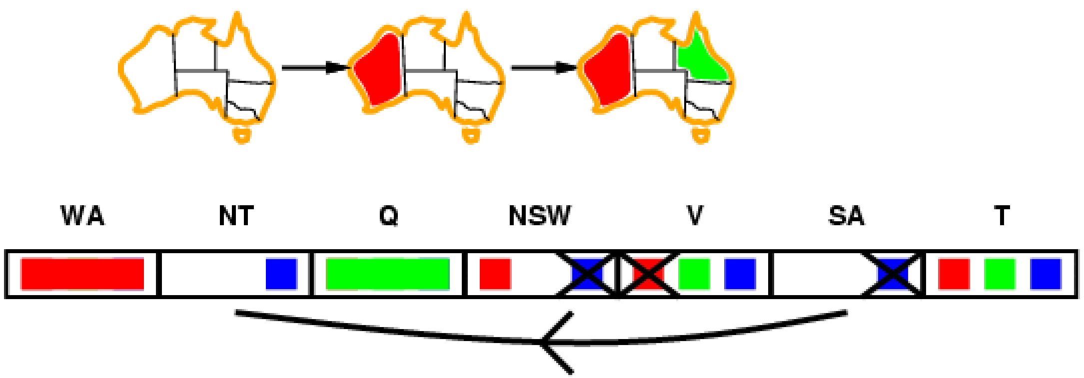
Arc consistency

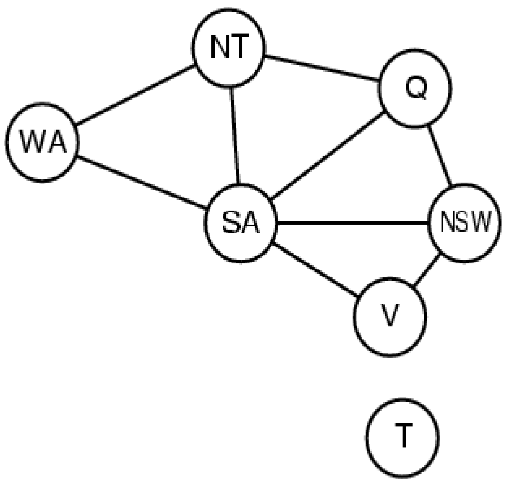
- *Arc consistency* est la forme de propagation de contraintes la plus simple
  - Vérifie la consistance entre les arcs.
  - C-à-d., la consistance des contraintes entre deux variables.



- *L'arc X →Y* est consistante si et seulement si

  Pour chaque valeur *x* de *X* il existe au moins une valeur permise de *y*.





Arc consistency 3 (AC-3)

function AC-3(*csp*) **return** the CSP, possibly with reduced domains
  inputs: *csp*, a binary csp with components (X, D, C)
  local variables: *queue,* a queue of arcs initially 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) **return** *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 constraints between X_i and X_j **then**
      delete *x* from D_i;
      removed ← *true*
  return *revised*

Arc consistency algorithm AC-3

- Complexité : O($c\ d^3$) dans le pire cas, où c est le nombre de contraintes
  - complexité de REVISE : O(d^2)
  - on a O(c) arcs, qui peuvent être réinsérés dans la file O(d) fois par REVISE
  - REVISE 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

Au de là de AC-3

- Min-conflicts (Section 5.3)
  - On commence avec une assignation complète, aléatoirement choisie.
  - Répétitivement:
    - » Choisir une variable parmi celles ayant des valeurs violant les contraintes
    - » Assigne à cette variable la valeur qui engendre le moins de conflits possibles avec les variables ayant des contraintes avec *elle*.
- Exploiter la structure du domaine (Section 6.5)
  - certains graphes de contraintes ont une structure « simple » qui peut être exploitée
  - peut améliorer le temps de calcul exponentiellement

Algorithme *min-conflicts*

Algorithme min-conflicts (*csp, nb_iterations*)

1. *assignation* = une assignation aléatoire complète (probablement pas compatible) de *csp*
2. pour *i* = 1 ... *nb_iterations*
    3. si *assignation* est compatible, retourner *assignation*
    4. *X* = variable conflictuelle choisie aléatoirement dans variables(csp)
    5. *v* = valeur dans domaine(*X*, *csp*) satisfaisant le plus de contraintes de *X*
    6. assigner (*X* = *v*) dans *assignation*
5. 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* (a réduit le temps d'exécution de 3 semaines! À 10 minutes)

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, problèmes d'ordonnancement avec des 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 étudiées de manière approfondies en recherche opérationnelle.
- Voir le cours **ROP 317 – Programmation linéaire** pour en savoir plus sur le cas linéaire et continu

Applications

- Problèmes d'ordonnancement (*scheduling*)
  - Par exemple planification des horaires pour les cours ou pour les quarts de travail des employés.

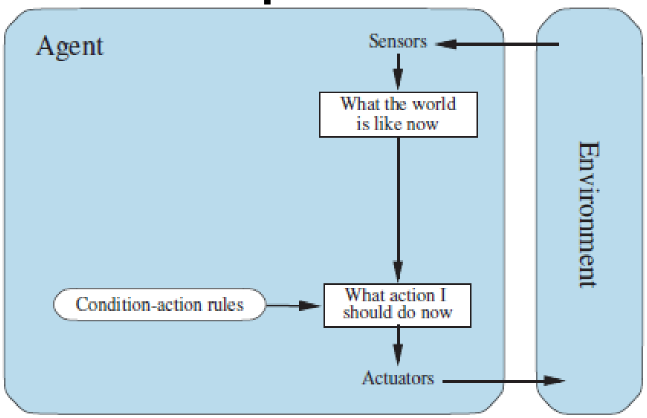
- Certains algorithmes de planification à base de connaissances invoquent des algorithmes CSP.

Conclusion

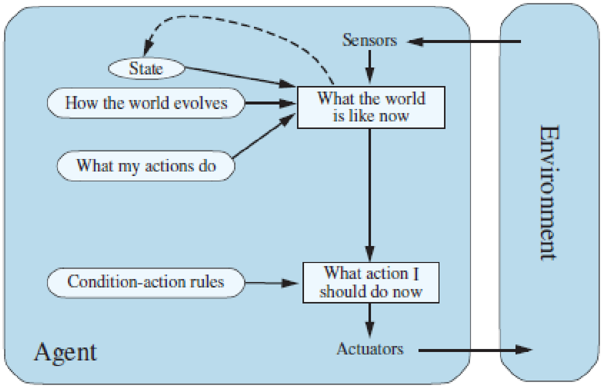
- Les problèmes CSP sont des problèmes de recherche dans un espace d'assignations de valeurs à des variables
- *Backtracking-search* revient à *Depth-First-Search* avec une variable assignée par nœud et qui recule lorsqu'aucune assignation compatible
- L'ordonnancement des variables et des assignations de valeurs aux variables jouent un rôle significatif dans la performance
- *Forward checking* empêche les assignations qui conduisent à un conflit direct
- 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 tâches

Satisfaction de contraintes pour quels d’agents?

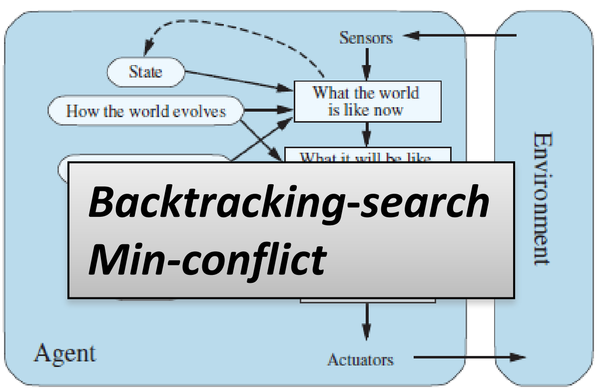
Simple reflex



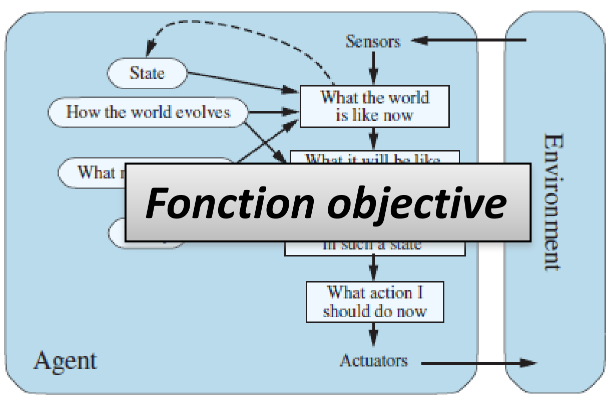
Model-based reflex



Goal-based



Utiliy-based



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 façons de l'améliorer
  - ordonnancement des variables
  - ordonnancement des valeurs
  - inférence (*forward checking*, AC-3)

- Savoir simuler *forward checking*, AC-3.

- Décrire comment résoudre un problème de satisfaction de contraintes avec *Min-Conflicts* (recherche locale)

Sujets couverts par le cours