Sixième partie VI

Problèmes de satisfaction de contraintes

Plan

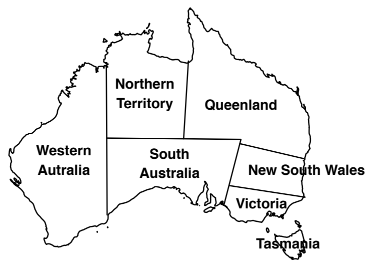
En bref ...

Exemples de CSP

Problème de satisfaction de contraintes (CSP)

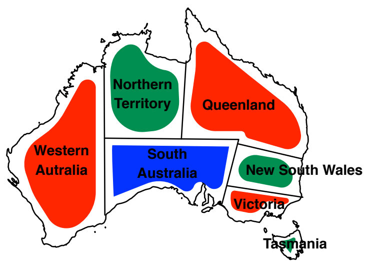
- Problèmes de recherche "classique"
  - Un état est "une boite noire"
  - N'importe quelle structure de données qui contient un test pour le but, une fonction d'évaluation, une fonction successeur
- CSP
  - Un état est défini par un ensemble de variables X_i, dont les valeurs appartiennent au domaine 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
  - Exemples de contraintes :
    - $WA \neq NT$
    - $(WA, NT) \in \{(rouge, vert), (rouge, bleu), (vert, rouge), (vert, bleu), \ldots\}$

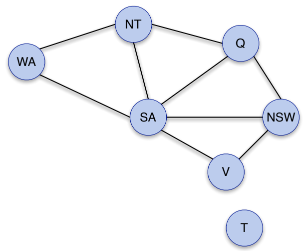
Exemple : coloriage de carte



- Les solutions sont des affectations de couleur qui satisfont toutes les contraintes
- Exemples : $\{WA = rouge, NT : vert, Q = rouge, NSW = vert, V = rouge, SA = blue, T = vert\}$

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'accélerer la recherche : par exemple, colorier la Tasmanie est un sous-problème indépendant

Variétés de CSPs

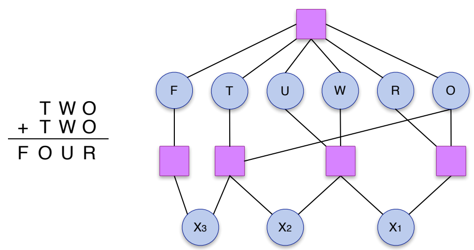
- Variables discrètes
  - Domaines finis : si de taille d, il y a $O(d^n)$ affectations complètes
    - Par exemple, CSPs booléens
  - Domaines infinis (entiers, caractères ...)
    - Par exemple, mise en place d'un planning, avec date de début/de fin pour chaque tâche
    - Nécessite un langage de contraintes, e.g., $StartJob1 + 5 \leq StartJob5$
    - 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

Exemple : puzzle cryptarithmétique



- 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 CSP du monde réel

- Problèmes d'affectation (e.g., qui enseigne quel cours ?)
- Problèmes d'emploi du temps
- Configuration de matériels
- Planification pour les transports (SNCF)
- Planification dans les usines, e.g, usinage de pièces
- Allocation de salles
- ...

Remarque

Beaucoup de problèmes du monde réel impliquent des variables à valeurs réelles

Formulation de la recherche standard

- Les états sont définis par les valeurs des variables déjà affectées
- État 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

Recherche arrière pour la résolution CSP

Formulation de la recherche standard

- Cet algorithme de recherche marche pour tous les CSP
- Chaque solution apparait à une profondeur de n s'il y a n variables
  - $\Rightarrow$ Utiliser la recherche en profondeur d'abord
- Le chemin de recherche n'est pas pertinent
- La taille de l'espace de recherche à explorer est très grand $b = (n - l)d$ avec
  - n : nombre de variables
  - d : taille du domaine des variables
  - b : facteur de branchement
  - p : la profondeur
- Soit $n!d^n$ feuilles alors qu'il n'y a que d^n affectations possibles !

Recherche en chaînage arrière

- 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 en chaînage arrière

Algorithme

```
funtion Backtracking-Search(csp) returns a Solution or Failure
    return Recursive-Backtracking({}, csp)

funtion Recursive-Backtracking(assignment, csp) returns a Solution or Failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(Variable[csp], assignment, csp)
    foreach value in Order-Domain-Value(var, assignment, csp) do
        if value is consistent with assigment given Constaints[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtraking(assigment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

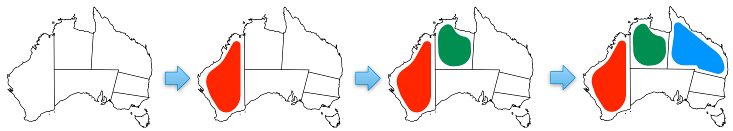
Exemple

Améliorer l'efficacité de la recherche arrière

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

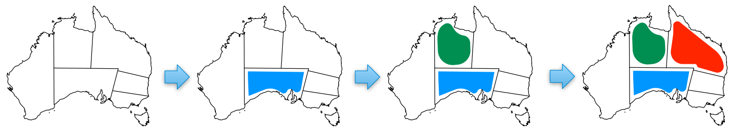
Comment choisir la variable à affecter ensuite ?

- Heuristique des valeurs minimales restantes (MRV)
  - ⇒ Choisir une des variables ayant le moins de valeurs "legales" possibles



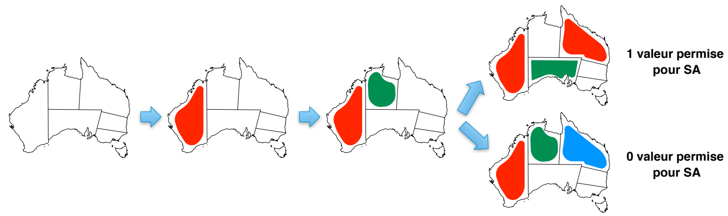
Comment choisir la variable à affecter ensuite ?

- Si plusieurs variables ne peuvent pas être départagées par l'heuristique MRV
- Heuristique du degré
  - ⇒ Choisir la variable qui apparaît dans le plus de contraintes



Comment ordonner les valeurs des variables ?

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



Remarque

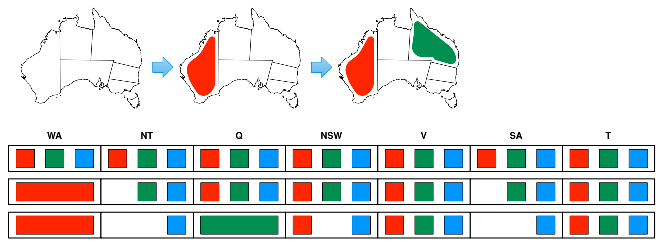
Combiner ces heuristiques permettent de résoudre le problème des n-reines, avec $n = 100$

Est-il possible de détecter un échec inévitable plus tôt ?

- Idée
  - Garder en mémoire les valeurs autorisées pour les variables qu'il reste à affecter
  - La recherche s'arrête lorsqu'une variable n'a plus d'affection possible

Est-il possible de détecter un échec inévitable plus tôt ?

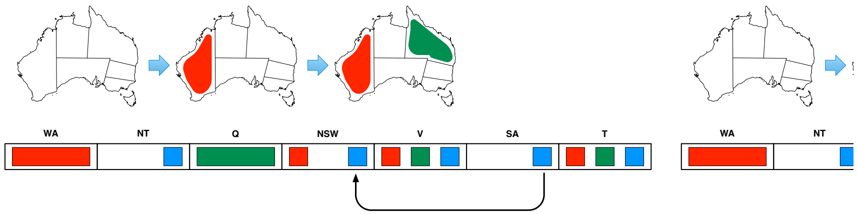
- La vérification avant permet de propager l'information des variables affectées aux variables non encore affectées, mais ne permet pas de détecter tous les échecs



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

Est-il possible de détecter un échec inévitable plus tôt ?

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



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

Algorithmes de vérification de consistence d'arcs

Algorithme

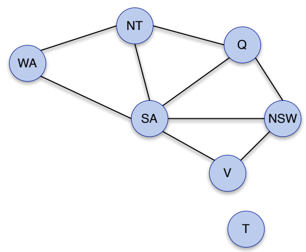
funtion AC-3(csp) **returns** *the CSP, possibly with reduce domains*
  input : *csp, a binary CSP with variable* $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue, a queue of arcs, initially all the arc in csp*
  while *queue is not empty* **do**
    $(X_i, X_j) \leftarrow$ Remove-First(*queue*)
    if Remove-Inconsistent-Value(X_i, X_j) **then**
      foreach X_k *in* Neighbors$[X_i]$ **do** add (X_k, X_i) *to queue*

funtion Remove-Inconsitent-Values(X_i, X_j) **returns** *true iff succeeds*
  removed $\leftarrow$ *false*
  foreach x *in* Domain$[X_i]$ **do**
    if *no value* y *is* Domain$[X_j]$ *allows* (x, y) *to satisfy the constraint* $X_i Xj$ **then**
      delete x *from* Domain$[X_i]$; *removed* $\leftarrow$ *true*
  return *removed*

Structure des problèmes

Structure des problèmes

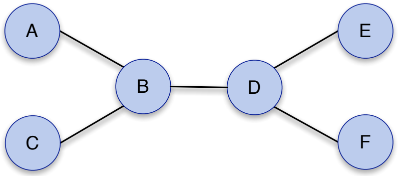
- Constat
  - Certaines parties d'un problème sont indépendantes
    - Exemple : La Tasmanie est un sous-problème indépendant
- Les sous-problèmes sont identifiables commes les composantes connexes du graphe de contraintes



Structure des problème

- Supposons que chaque sous-problème possède c variables des n variables du problème global
- Dans le pire des cas, la complexité en temps est de $n/c \times d^c$, i.e., linéaire en fonction de n
- Par exemple avec $n = 80$, $d = 2$ et $c = 20$
  - $2^{80} = 4$ milliard d'année à 10 millions de nœuds explorés par seconde
  - $4 \times 2^{20} = 0{,}4$ seconds à 10 millions de nœuds explorés par seconde

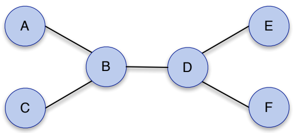
Les problèmes CSP structuré en arbre

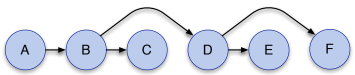


- Théorème
  - Si le graphe de contraintes ne contient pas de cycle, le CSP a une complexité en temps de $O(nd^2)$
- Dans le cas général, un problème CSP a une complexité dans le pire des cas en $O(d^n)$

Algorithme pour les problèmes CSP structurés en 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

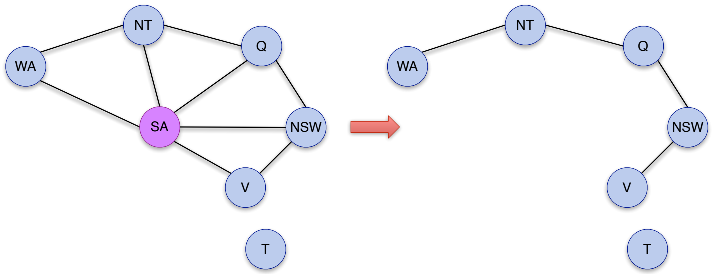




2. Pour j de n à 2, appliquer $\textsc{Remove-Inconsistent-Value}(\mathit{Parent}(X_j), X_j)$
3. Pour j de 1 à n, affecter X_j de façon à ce qu'il soit consistent avec $\mathit{Parent}(X_j)$

CSPs quasiment structurés en arbre

- Modification à apporter
  - Instancier une variable, restreindre les domaines des voisins



- Modification de coupe
  - instantancier de toutes les manières possibles un ensemble de variables tel que le graphe de contraintes reste un arbre
  - Coût pour un cycle de taille c : $O(d^c \times (n-c)d^2$
  - Très rapide pour un c petit

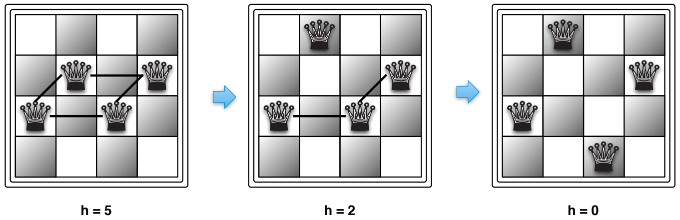
CSP et recherche locale

CSP et recherche locale

- Il est possible d'utiliser des algorithmes de recherche locale, e.g, descente de gradient, pour la résolution de CSP
- Attention : Les algorithmes de recherche locale fonctionnent avec des états "complets", i.e., dans lesquels toutes les variables sont affectées.
- Pour appliquer ces algorithmes aux CSPs il faut :
  - Permettre d'avoir des états avec des contraintes non satisfaites
  - Avoir des opérateurs de réaffectation de variable
- La sélection des variables s'effectue en choisissant n'importe quelle variable en conflit
- La sélection d'une valeur s'effectue grâce à l'heuristique min-conflict
  - choisir une valeur qui enfreint le moins de contraintes
  - par exemple, $h(n)$ = nombre total de contraintes non respectées

CSP et recherche locale

- États : 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 l'échiquier



- 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 pour $n = 10000000$

Conclusion

- CSPs sont des types de problèmes particuliers
  - les états sont définis par des valeurs associées à des variables
  - le but est défini comme des contraintes sur les valeurs des variables
- La recherche de chaînage arrière en profondeur d'abord avec une variable affectée par nœud est l'algorithme de base
- L'ordre des variables ainsi que l'ordre d'affection des valeurs impactent les performances de la recherche
- La vérification avant permet de limiter les echecs d'affectation
- La propagation de contraints (la consistence d'arc) permet de réduire le domaine des variables et améliore les performances des algorithmes de résolution
- Il est possible d'exploiter la structure d'un problème pour améliorer la résolution
- Les problèmes CSP sous forme d'arbre peuvent être résolus en temps linéaire