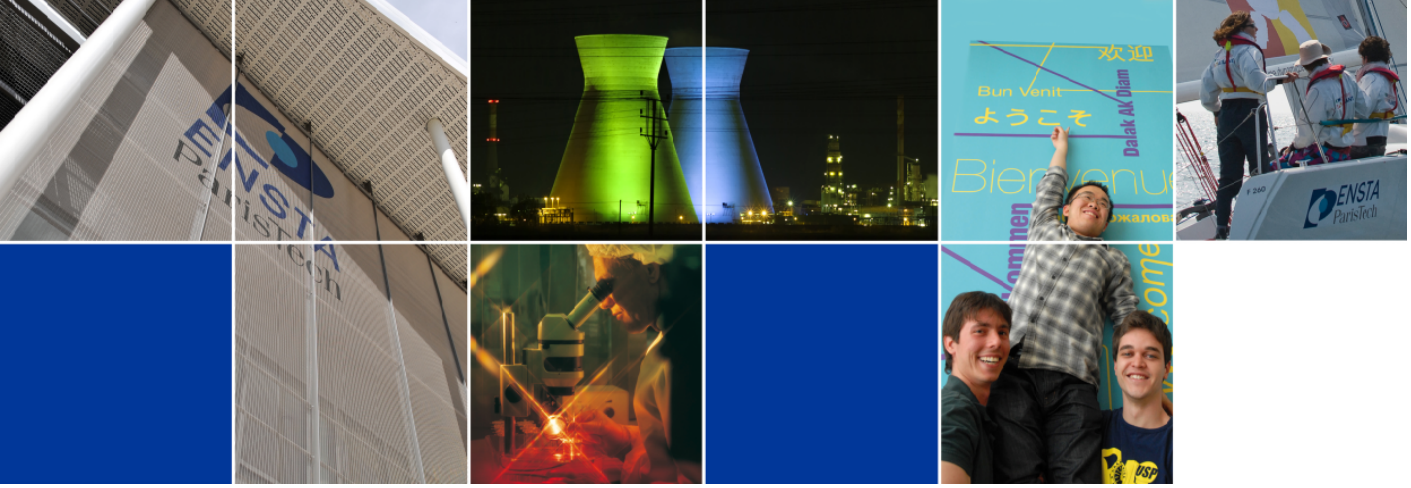


Algorithmes de résolution

Julien Alexandre dit Sandretto

Department U2IS
ENSTA Paris
IA302-2020-2021

ENSTA

IP PARIS

Introduction

Cours précédent

Contrainte et problèmes de satisfaction de contraintes (CSPs)

Comment les résoudre ?

Idée générale des algorithmes de résolution

Explorer l'ensemble des affectations que peuvent prendre les variables dans leur domaine respectif jusqu'à trouver une solution (CSP consistant), ou de démontrer qu'il n'existe pas de solution (CSP inconsistant).

$\Rightarrow$ Algorithmes **complets** : certain de trouver une solution si le CSP est consistant (on teste toutes les possibilités, approche exhaustive).

Résolution d'un CSP

La résolution d'un CSP consiste donc dans l'affectation de valeurs aux variables de telle sorte que **toutes** les contraintes soient satisfaites. On se place dans le cas où les domaines des variables sont finis. Il est peut être utile de détailler le type d'affectation :

- affectation : instanciation des variables sur les domaines
- affectation totale : affectation de toutes les variables
- affectation partielle : affectation de certaines variables
- affectation consistante : affectation qui ne viole aucune contrainte
- affectation inconsistante : affectation qui viole au moins une contrainte

Ainsi une **solution** est une affectation totale et consistante.

Systématique

L'approche la plus simple et la plus naïve : énumérer toutes les affectations totales jusqu'à en trouver une de consistante. Si aucune n'est trouvée alors le CSP est inconsistant.

On appelle cette approche la méthode **systématique**.

Deux algorithmes implémentent cette approche :

- "génère et teste"
- "simple retour arrière".

Génère et teste

Algorithme très simple

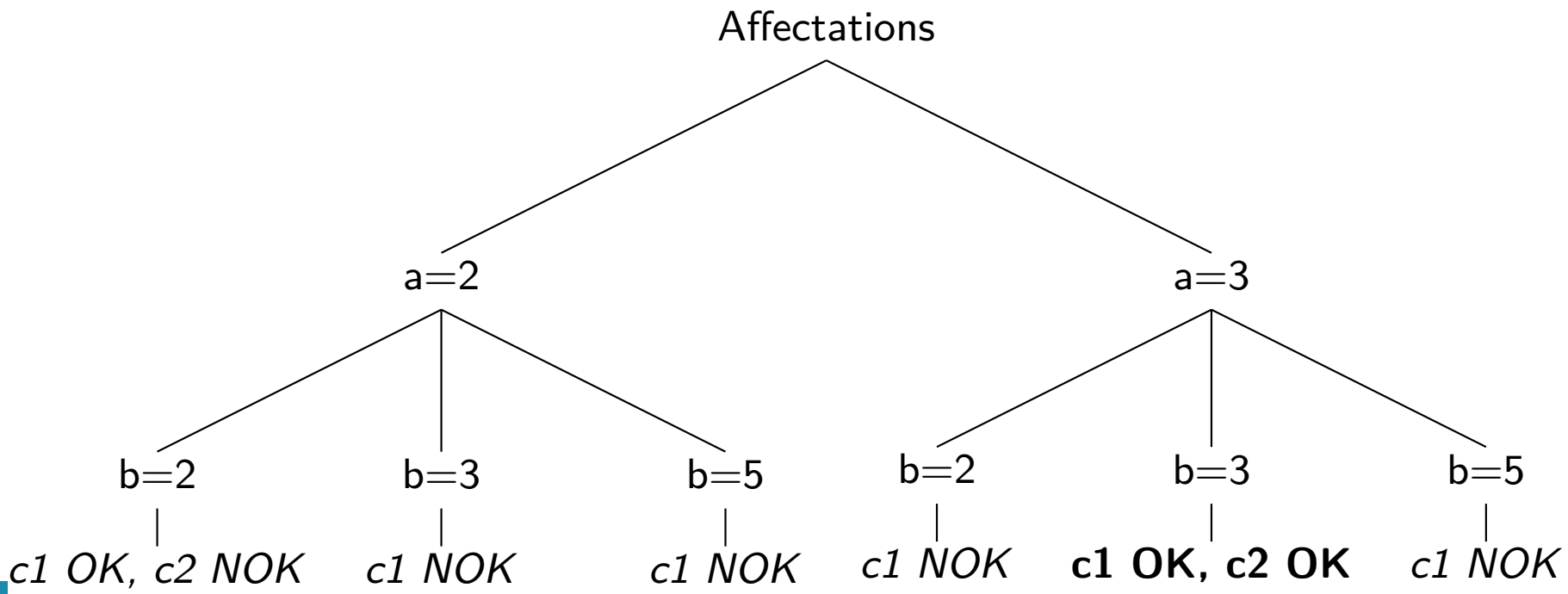
Il consiste en deux étapes :

1. génération d'une affectation totale
2. test de sa consistance

Génère et teste

Considérons l'exemple suivant :

- $\mathcal{X} = \{a, b\}$
- $\mathcal{D} = \{\{2, 3\}, \{2, 3, 5\}\}$
- $\mathcal{C} = \{a = b, a + b \geq 5\}$



Génère et teste

Simple à mettre en oeuvre, mais potentiellement long à terminer.

Inconsistence prouvée qu'après avoir testé toutes les configurations

Au pire cas : $|E| = |D_1| \times |D_2| \times \cdots \times |D_n|$ affectations !

CSP à 5 variables dont chacune associée à un domaine à 4 valeurs : 1024 affectations totales

Simple retour arrière

Simple d'améliorer l'algorithme "Génère et teste" :

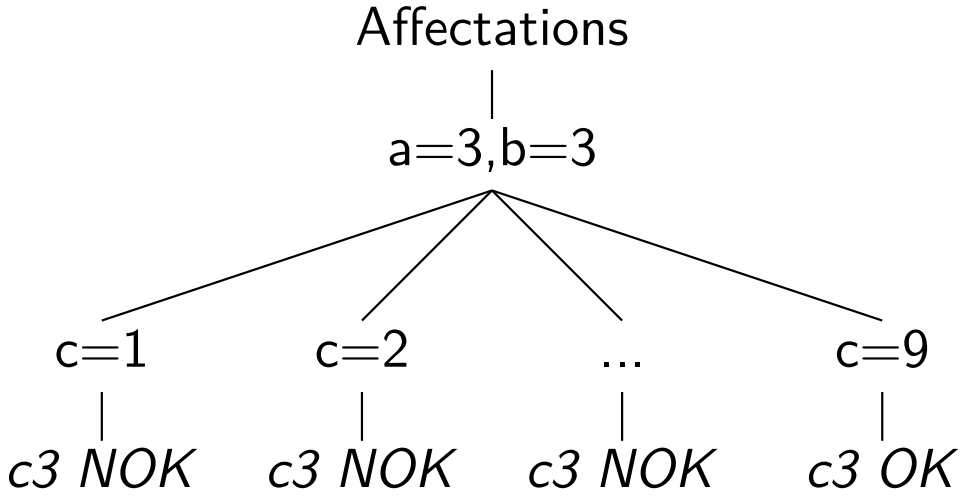
1. générer une affectation partielle **consistante**
2. affecter une variable supplémentaire
3. si l'affectation partielle devient inconsistante par cet ajout, on retourne en arrière pour modifier l'affectation de la variable supplémentaire. Si elle est consistante, on continue à affecter une nouvelle variable jusqu'à l'affectation totale.

Simple retour arrière

Considérons l'exemple suivant :

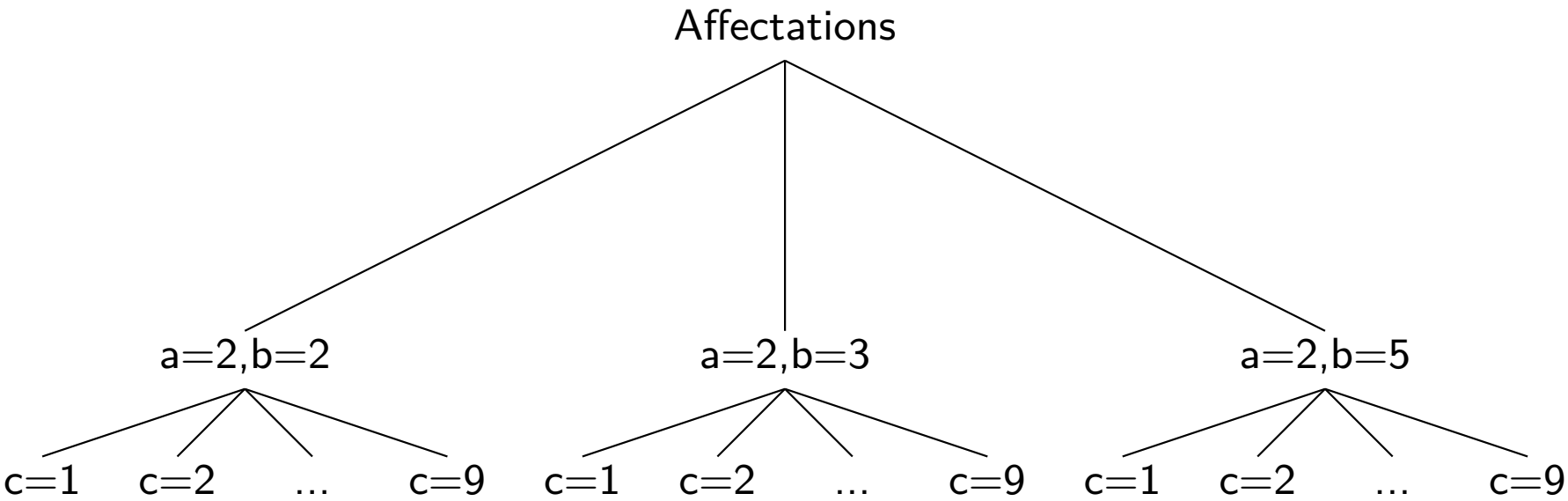
- $\mathcal{X} = \{a, b, c\}$
- $\mathcal{D} = \{\{2, 3\}, \{2, 3, 5\}, \{1, 2, \ldots, 9\}\}$
- $\mathcal{C} = \{a = b, a + b \geq 5, a + 2b = c\}$

On part de l'affectation partielle consistante trouvée précédement, et on affecte seulement c.



Simple retour arrière

5 affectations partielles pour trouver $(a, b) = (3, 3)$, puis 9 pour trouver c (45 avec l'approche "génère et teste")



Simple retour arrière

Avantage

Réduire l'espace de recherche

Toujours une approche systématique

Plus efficace

Exploiter les contraintes pour réduire l'espace de recherche

$\Rightarrow$ filtrage

Filtrage

Filtrage du domaine = affectation partielle + contraintes

Enlever *a priori* les valeurs localement inconsistantes avant d'affecter une nouvelle variable.

Anticiper l'inconsistance d'une branche avant de réaliser une affectation (si le domaine d'une variable est vide) et donc d'abandonner la branche courante pour revenir en arrière plus tôt

Filtrage à différents niveaux suivant le type de **consistances**

Consistance de noeud

La consistance de noeud s'intéresse aux contraintes unaires

Un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est consistant de noeud si pour toute variable x_i de $\mathcal{X}$, et pour toute valeur v de D_{x_i}, l'affectation partielle $\{(x_i, v)\}$ satisfait toutes les contraintes unaires de $\mathcal{C}$.

Consistance de noeud

Exemple précédent, si a a été affecté à 2, alors la contrainte $a = b$ devient la contrainte unaire $2 = b$. La consistance de noeud nous permet de réduire le domaine de b au singleton $\{2\}$ (mais cette valeur viole la seconde contrainte).

Consistance rapide à calculer et réduit les domaines des variables

Consistance d'arc

Consistance plus forte mais plus difficile à utiliser = anticiper l'affectation de deux valeurs : consistance d'arc

Un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est consistant d'arc si pour tout couple de variables (x_i, x_j) de $\mathcal{X}$, et pour toute valeur v_i appartenant à D_{x_i}, il existe une valeur v_j appartenant à D_{x_j} telle que l'affectation partielle $\{(x_i, v_i), (x_j, v_j)\}$ satisfasse toutes les contraintes binaires de $\mathcal{C}$.

Consistance d'arc

Exemple précédent, si aucune variable n'a été affectée, les deux premières contraintes conduisent à $(a, b) = (2, 2)$ ou $(a, b) = (3, 3)$ pour la première contrainte, $(2, 2)$ violant la second contrainte, seule l'affectation $(3, 3)$ est consistante d'arc.

Consistance de chemin

Encore plus fort, mais plus long à effectuer = consiste à anticiper de trois étapes l'énumération : consistance de chemin, aussi appelée 3-consistance.

S'il reste k variables à affecter, et si l'on anticipe de k étapes l'énumération pour établir la k-consistance, l'opération de filtrage revient à résoudre le CSP

Un ensemble $\{x_i, x_j\}$ est consistant de chemin par rapport à la variable x_k si pour toute affectation partielle $\{(x_i, v_i), (x_j, v_j)\}$, il existe une valeur v_k de D_{x_k} telle que $\{(x_i, v_i), (x_k, v_k)\}$ **et** $\{(x_k, v_k), (x_j, v_j)\}$ sont consistants.

Consistance de chemin

Ainsi dans notre exemple où 3 variables doivent être affectées, la consistance de chemin nous donne directement $(3, 3, 9)$.

Propagation

Filtrage = anticipation d'affectation + propagation

Si domaine vide pour au moins une variable : affectation non consistante et on revient en arrière

Cette approche permet la définition de deux algorithmes :

- le forward/checking : anticipation + consistance de noeud
- le look ahead : anticipation + consistance d'arc

Heuristiques

Algorithmes précédents : choisir quelle variable instancier et quelle valeur attribuer

Si chance : plus rapide, sinon solution trouvée à la fin

Problème général de la satisfaction d'un CSP est NP-complet
$\Rightarrow$ impossible de connaitre l'ordre optimal

Utilisation d'heuristiques !

Heuristiques

Une heuristique est une règle non systématique (elle fonctionne dans certains cas mais pas dans tous, sans que l'on sache *a priori* si elle va marcher) qui nous donne des indications sur la direction à prendre dans l'arbre de recherche.
L'ordre d'instanciation peut être :

- **statique** : il a été décidé avant de lancer l'algorithme. Par exemple, on classe les variables par le nombre de contraintes où elles apparaissent.
- **dynamique** : à chaque étape d'anticipation, la prochaine variable à instancier est choisie en fonction d'une règle. Par exemple, la variable dont le domaine a été le plus réduit par l'affectation partielle courante, ainsi il y a plus de chance de rendre ce domaine vide par le filtrage et ainsi d'éliminer la branche courante.

Heuristiques

- Sur les variables :
  - choisir en premier la variable la plus contrainte (celle apparaissant le plus dans les contraintes) pour plus d'effet lors de la propagation
  - choisir en premier la variable avec le plus petit domaine (moins de valeurs possibles) pour garder l'arbre de recherche le plus petit possible
- Sur les valeurs : essayer d'attribuer en premier une valeur faisable à une variable pour trouver une solution le plus rapidement possible
- Sur les contraintes :
  - la plus "simple" en premier pour le plus d'effet ($x = 3$ contraint plus que $x^2 + y > 2$)
  - classer par ordre croissant du nombre de variables (optimisation de la propagation)
  - évaluer dynamiquement l'effet de chaque contrainte par rapport à chaque variable...

TP : algorithme de résolution d'un Sudoku

En reprenant le formalisme des CSPs, décrivez le problème du Sudoku puis proposez un algorithme de résolution en s'appuyant sur les idées vues aujourd'hui. Si le temps le permet, implémentez cet algorithme en Python (des fichiers pour démarrer vous sont fournis).

Solution

Le problème du Sudoku sous forme de CSP s'écrit :

- $\mathcal{X} = \{81$ cases identifiées par (ligne,colonne)$\}$
- $\mathcal{D} = \{0$ (non initialisée), $\{1, 2, \ldots, 9\}$ (initialisée)$\}$
- $\mathcal{C} = \{$Toutes différentes(ligne), Toutes différentes(colonne), Toutes différentes(cellule)$\}$

Solution

Voici l'algorithme pour résoudre un Sudoku. Il utilise un simple retour arrière et un filtrage (consistance de noeud).
Tant que toutes les cases ne sont pas remplies :

- P = Valeurs possibles de la case à remplir courante (filtrage)
- Si P est vide alors on revient à la case remplie précédement
- Sinon on renseigne la case courante avec une valeur de P et on passe à la case suivante

L'étape de filtrage sur la consistance de noeud est très simple :
$P = \{1, \ldots, 9\}$ - (toutes les valeurs initialisées dans la même ligne, colonne et cellule que la case courante).

Question supplémentaire

Rappelez vous, en introduction nous avions exploité le fait qu'une seule case pouvait accepter la valeur 9 dans une ligne et que comme toutes les valeurs doivent être attribuées, alors forcément cette case devait accueillir le 9. De quel type est cette consistance ?

Solution

Consistance de chemin.

Solution

Pour le Python :

```
while caseAremplir < len(trous):
  possibles[caseAremplir] = casePossibles(trous[caseAremplir],sudoku)
  try:
    while not possibles[caseAremplir]:
      sudoku[trous[caseAremplir][0]][trous[caseAremplir][1]] = 0
      caseAremplir -= 1
  except IndexError:
    print("Le sudoku n'a pas de solution.")
    exit(1)
  sudoku[trous[caseAremplir][0]][trous[caseAremplir][1]] = possibles[caseAremplir].pop()
  caseAremplir += 1
```