Cours 10 – Programmation par contraintes : Algorithmique

Informatique Musicale
Master 2 - ATIAM

Pierre Talbot (talbot@ircam.fr)

UPMC/IRCAM

16 novembre 2017

Le menu

- Introduction
- Construction de l'arbre de recherche
- Inférence
- Stratégie d'exploration

Programmation par contraintes

Holy grail of computing

- Paradigme déclaratif pour résoudre des problèmes combinatoires.
- On pose le problème et l'ordinateur le résout pour nous.

Paradigme reconnu

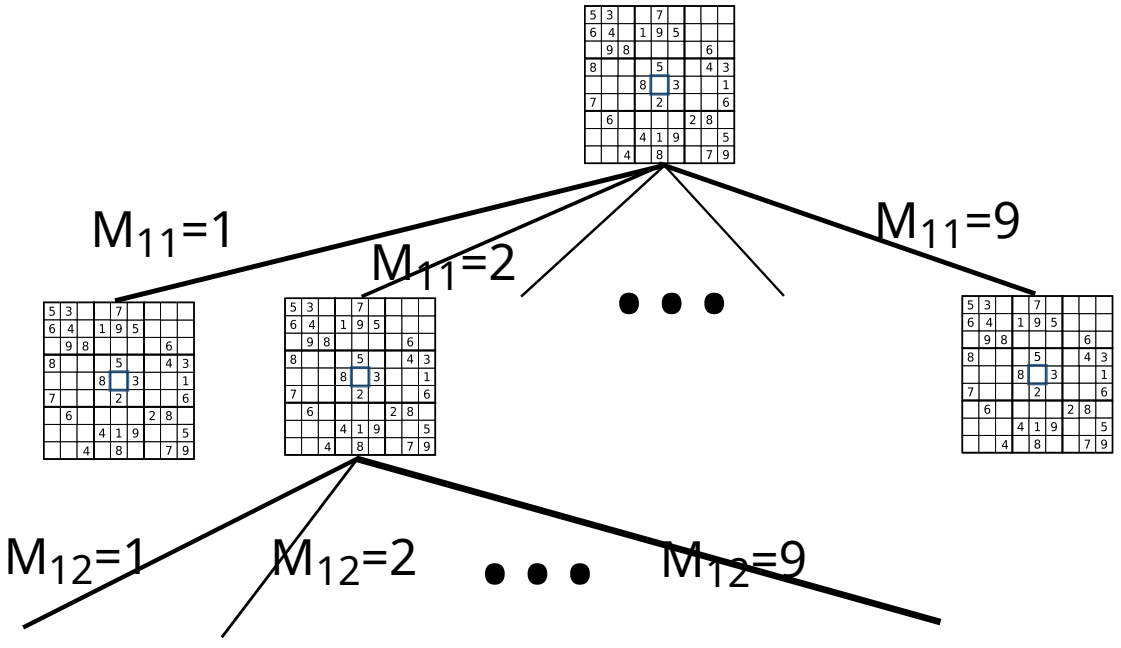
Applications

La PPC a beaucoup d'applications du solveur de Sudoku, planification, packing, composition musicale...

Comment trouver une solution ?

Nature NP-complete

- Essayer toutes les combinaisons jusqu'à ce qu'on trouve une solution.
- Algorithme de *backtracking* construisant un arbre de recherche.



Solveur de contraintes

Le but d'un solveur de contraintes est de :

- Construire l'arbre de recherche à partir d'un modèle.
- Réduire l'arbre de recherche via différent type d'inférence (propagation, nogoods,...).
- Explorer cet arbre via une stratégie d'exploration (DFS, BFS, LDS, IDS, ...).
- Gérer la mémoire pour pouvoir restaurer les variables quand on backtrack.

La programmation par contraintes est l'étude des algorithmes de backtracking.

Le menu

- Introduction
- Construction de l'arbre de recherche
- Inférence
- Stratégie d'exploration

Problème de satisfaction de contraintes

Définition classique d'un CSP

Triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où :

- $\mathcal{X}$ est l'ensemble des variables du problème.
- $\mathcal{D}$ est l'ensemble des domaines des variables tel que $\forall i, dom(\mathcal{X}_i) \subseteq \mathcal{D}_i$.
- $\mathcal{C}$ est l'ensemble des contraintes.

Exemple des n reines

- $\mathcal{X} = \{x_1, \ldots, x_n\}$, une variable par ligne dénotant l'indice de la colonne.
- $\forall x \in \mathcal{X}, \mathcal{D}(x) = \{1, \ldots, n\}$, la taille de l'échiquier.
- $\mathcal{C} =$
  1. distinct$(x_1, \ldots, x_n)$, chaque colonne différente.
  2. $\forall i, j, i > j, x_i + i \neq x_j + j$, chaque diagonale montante.
  3. $\forall i, j, i > j, x_i - i \neq x_j - j$, chaque diagonale descendante.

Définition alternative

Définition alternative d'un CSP

Couple $\langle d, \mathcal{C} \rangle$ où :

- d est une fonction $\mathcal{X} \to \mathcal{P}(\mathcal{V})$
- $\mathcal{C}$ est l'ensemble des contraintes.

Affectation

Une affectation a est une fonction $Asn : \mathcal{X} \to \mathcal{V}$. L'ensemble des valeurs $\mathcal{V}$ est l'union des domaines de toutes les variables.

Contrainte

Une contrainte $c \in \mathcal{P}(Asn)$ est un ensemble d'affectations. Une affectation $a \in c$ est une solution de c.

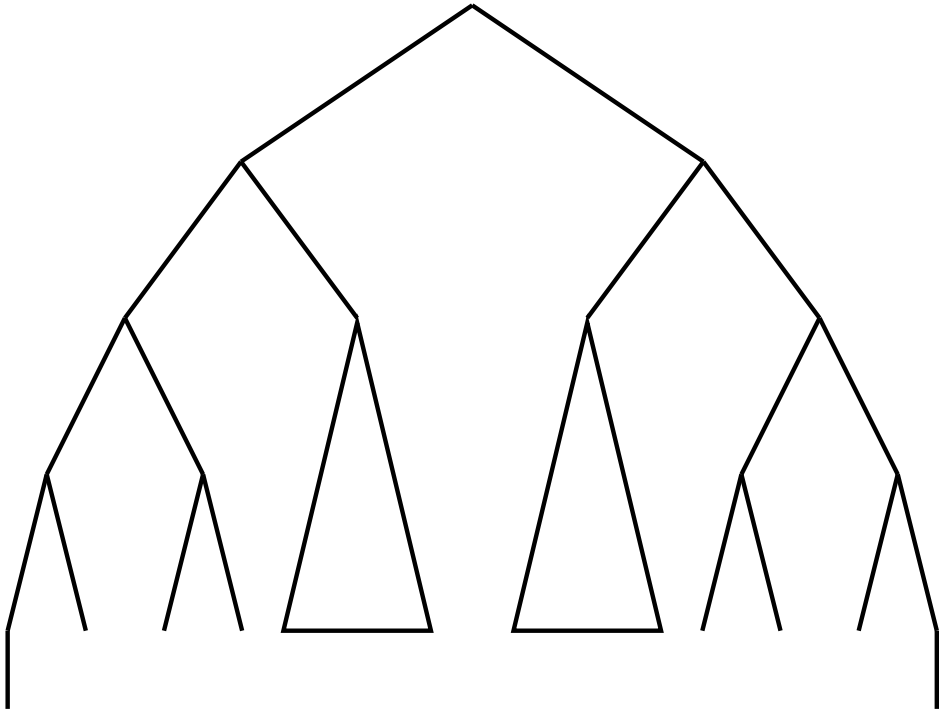
Résolution d'un CSP

Comment résoudre ce problème ?

Méthode naïve : générer et tester

1. On démarre au nœud racine avec un ensemble d'affectation vide, $\{\}$.
2. On sélectionne une variable x non instanciée et on crée $|d(x)|$ nœuds fils dans lesquels l'ensemble d'affectation est augmenté avec $\{x = v_i\}$ où $v_i \in d(x)$ et v_i est différent pour chaque nœud.
3. L'exploration d'un nœud est terminée si celui-ci viole une contrainte ou si toutes les variables sont affectées (solution).

En image...



$\{M_{1,1} = 1, M_{1,2} = 1, \ldots, M_{9,9} = 1\}$ $\{M_{1,1} = 9, \ldots, M_{9,9} = 9\}$

Crafter son solveur en Python

Étape 1 : Permutation

Soit $n \in \mathbb{Z}$ et $dom \in \mathbb{Z}$, écrire une fonction `permutation(n, dom)` qui affiche toutes les permutations de n variables prenant une valeur $0 \leq v < dom$.

Example

Avec $n = 3$ et $dom = 2$.

```
# python3 permutation.py
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
(...)
```

Co-routines en Python

- Dans le code de permutation, on affiche la solution quand on arrive sur un nœud feuille.
- Comment faire pour afficher une seule solution ?
- Comment faire pour faire un autre traitement qu'un affichage ?

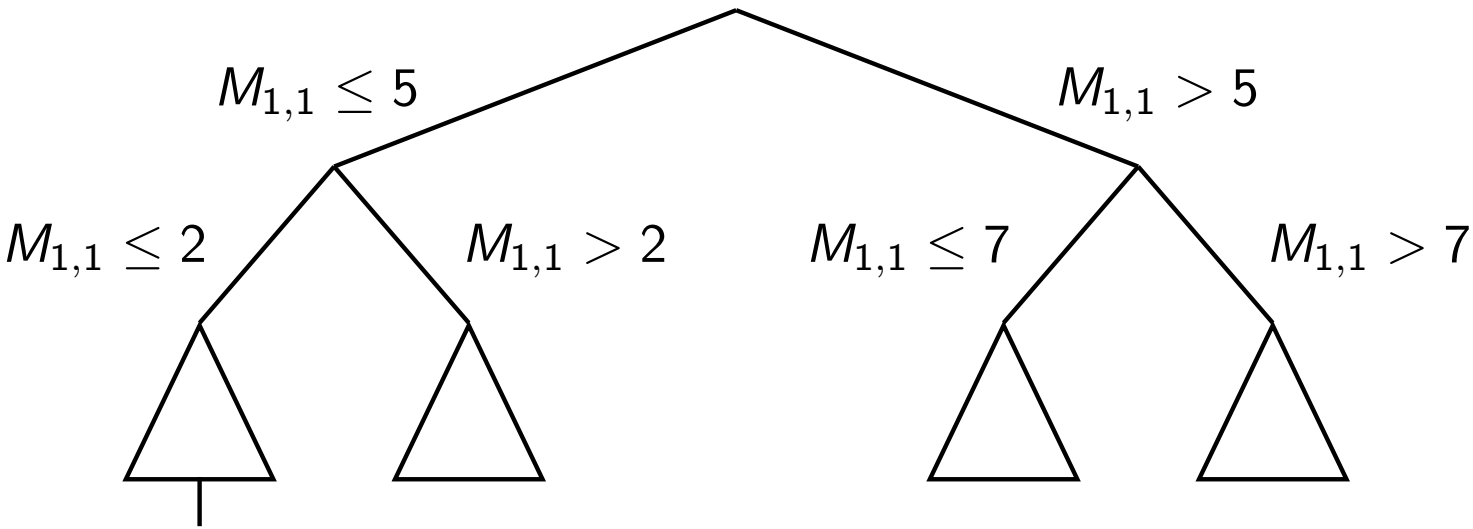
Co-routine

- Utilisation du mot clé `yield` et `yield from` pour créer des générateurs.
- La fonction `permutation` devient un générateur de solution sur laquelle on peut itérer :

```
for sol in permutation(3, 2):
  print(sol)
```

Stratégies de branchement

- Énumération : idem méthode naïve.
- Choix binaire : $x = c$ et $x \neq c$ (deux branches).
- Séparation de domaine : $x < c$ et $x \geq c$ (deux branches), généralisation du choix binaire.



$\{M_{1,1} = 1, M_{1,2} = 1, \ldots, M_{9,9} = 1\}$

Branching : Composition de trois fonctions

- `x = select_var(vars)` retourne l'indice de la variable sur laquelle on veut "brancher".
- `v = select_val(x, vars)` retourne la valeur de la variable a sélectionner.
- `distribute(x, v, vars)` réalise le branching en tant que tel en splittant la variable x avec v.

En MiniZinc

MiniZinc fournit des annotations pour spécifier la stratégie de branching.

```
array[0..10] of var 0..10: vars;
solve :: int_search(vars, first_fail, indomain_min, complete)
```

Ajout des variables

Étape 2 : Liste de variables

Soit $d : \mathcal{X} \to \mathcal{P}(\mathcal{V})$, écrire une fonction `solve(d)` qui affiche toutes les permutations de $len(d)$ variables prenant une valeur $\forall x_i \in d . v_i \in d(x_i)$.

Example

Avec $d(x) = [0..1], d(y) = [0..2], d(z) = [0..0]$, soit en Python :

```
solve(dict(
  x=set(range(2)),
  y=set(range(3)),
  z=set(range(1))))
```

```
# python3 backtracking.py
{'x': {0}, 'y': {0}, 'z': {0}}
{'x': {0}, 'y': {1}, 'z': {0}}
{'x': {0}, 'y': {2}, 'z': {0}}
(...)
```

Satisfiabilité des contraintes

Consistance d'une contrainte

Soit une contrainte c et les variables d, on peut décider la consistance d'une contrainte :

- `SAT` : si $\forall a \in d.a \in c$.
- `UNSAT` : si $\forall a \in d.a \notin c$.
- `UNKNOWN` sinon.

La fonction de consistance d'une contraintes à le type $Dom \rightarrow Consistency$.

Consistance de $x \neq y$

Require: `consistency`$(x \neq y)$

1: **if** $|x| = 1 \wedge |y| = 1 \wedge x = y$ **then**
2: **return** `UNSAT`
3: **else if** $|x| = 0 \vee |y| = 0$ **then**
4: **return** `UNSAT`
5: **else if** $x.\mathit{isdisjoint}(y)$ **then**
6: **return** `SUBSUMED`
7: **else**
8: **return** `OK`
9: **end if**

Solution d'un CSP

Domaine

Un domaine d représente un ensemble d'affectation et peut être vu comme la contrainte $con(d) = \{a \in Asn \mid \forall x \in \mathcal{X} : a(x) \in d(x)\}$.

Solutions

Les solutions d'un CSP peuvent être définies par $sol(\langle d, \mathcal{C} \rangle) : \{a \in con(d) \mid \forall c \in \mathcal{C}, a \in \mathcal{C}\}$.

Consistance d'un ensemble de contraintes

Lorsque toutes les contraintes retournent SAT.

Ajout des contraintes

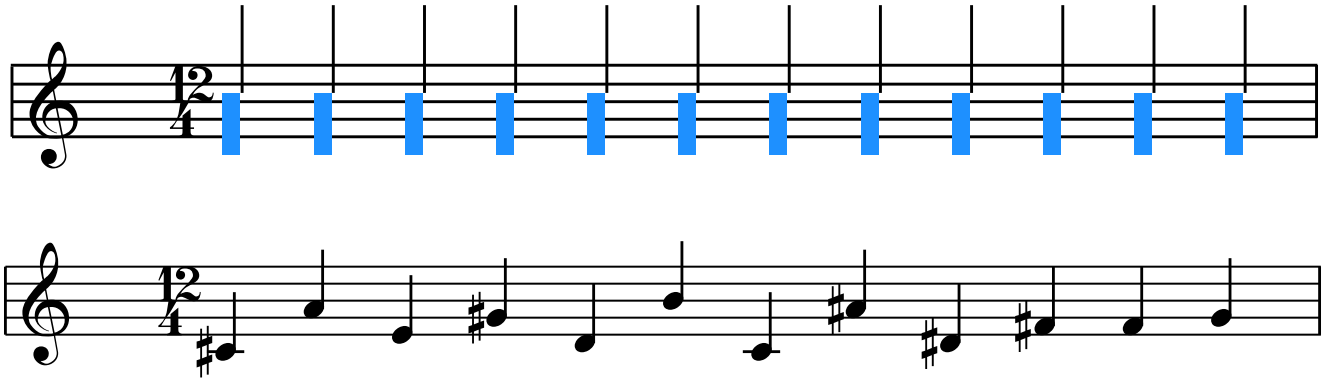
Étape 3 : Liste de contraintes

On représente une contrainte par une fonction $\mathit{Dom} \to \mathit{Consistence}$. Écrire la fonction de consistance pour la contrainte $x \neq y$ et d'un ensemble de contraintes.

All-Interval Series (AIS)

Soit une série de N notes avec une hauteur entre 1 et P, on s'assure que :

- Toutes les hauteur de notes sont différents.
- Tous les intervalles entre chaque note successive sont différents.



Étape 4 : Modéliser un problème

- Modéliser le problème AIS dans votre système.
- Est-ce qu'il vous manque une contrainte ? Si oui, ajoutez là.

Le menu

- Introduction
- Construction de l'arbre de recherche
- Inférence
- Stratégie d'exploration

Propagation

- Jusqu'à présent on se contente d'énumérer toutes les combinaisons possibles mais...
- Une optimisation consiste à réaliser la propagation.

Propagation

Consiste à associer des opérateurs de réduction de domaine, appelés propagateurs, aux contraintes.

Example

Soit $x > y$, $x, y \in [0..1]$, la propagation consiste à filtrer les domaines de x et y et elle inférera que $x = 1 \wedge y = 0$.

Propagateur

Propagateur

Un propagateur est une fonction $p : Dom \to Dom$ qui a deux rôles :

- Décider si une affectation satisfait la contrainte induite par le propagateur, $p(\{a\})$ retournera $\emptyset$ si elle est rejetée et $\{a\}$ sinon.
- Éliminer les affectations du domaine ne satisfaisant pas la contrainte.

On a déjà programmé le premier rôle avec `consistance`.

Réfléchissez à l'implémentation du propagateur $x \neq y$.

Propagateur $x \neq y$

Require: `propagate`$(x \neq y)$

1: **if** $|x| = 1$ **then**
2: $\quad y \leftarrow y \setminus x$
3: **else if** $|y| = 1$ **then**
4: $\quad x \leftarrow x \setminus y$
5: **end if**
6: **return** $|x| > 0 \wedge |y| > 0$

Propagation + Recherche

- La propagation peut être réalisée à chaque nœud de l'arbre de recherche.
- Point fixe : la fonction de propagation se résume à calculer $p_1(\ldots p_n(d))$ jusqu'à ce que tous les propagateurs soient stables, c'est-à-dire que $p_1(\ldots p_n(d)) \equiv d$.

Require: $\texttt{solve}(\langle d, P \rangle)$

1: $d' \leftarrow \texttt{propagate}(\langle d, P \rangle)$
2: **if** $d' = \emptyset$ **then**
3: **return** $\emptyset$
4: **else if** $d' = \{a\}$ **then**
5: **return** $\{a\}$
6: **else**
7: $\langle d_1, \ldots, d_n \rangle \leftarrow \texttt{branch}(d')$
8: **return** $\bigcup_{i=0}^{n} \texttt{solve}(\langle d_i, P \rangle)$
9: **end if**

Un solveur Python avec propagation

Étape 5 : Ajout de la propagation

- Ajouter le propagateur $x \neq y$.
- Ajouter la fonction de propagation.
- Tester avec AIS, est-ce qu'il se résout plus vite ?
- En en MiniZinc ?

Le menu

- Introduction
- Construction de l'arbre de recherche
- Inférence
- Stratégie d'exploration