Programmation Logique par Contraintes

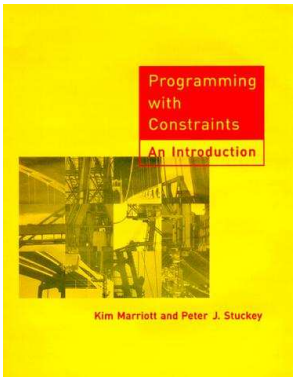
Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes
treinen@pps.jussieu.fr

Organisation

- Page web du cours (copies des transparents):
  `http://www.pps.jussieu.fr/~treinen/teaching/clp`
- Avertissement : Les transparents ne contiennent pas tout.
- Contrôle des connaissances :
  Note finale première session =
  1/2 Note partiel + 1/2 Note examen
  Note finale deuxième session = examen
- TD/TP : Jeudi 8h30 ou 10h30, Salle T. Début : 20/11.
- Examen : Lundi 5 janvier matin (durée 2 heures)
  seulement sur la partie *Programmation logique par contraintes*

Livre principalement utilisé



The MIT Press, 1998, `www.cs.mu.oz.au/~pjs/book/book.html`

Qu'est-ce que c'est les contraintes ?

- Une contrainte est une formule logique, construite sur un langage fixé d'avance.
- Une contrainte dénote un ensemble de solutions (les solutions de la formule) pour une interprétation logique fixée d'avance.
- Exemple : La contrainte $X + Y = 1$ dénote les deux solutions $\{X = 0, Y = 1\}$ et $\{X = 1, Y = 0\}$ si le domaine d'interprétation est $\mathbb{N}$.
- C'est une généralisation des problèmes d'unification :
  - Problème d'unification $\Rightarrow$ contrainte (formule)
  - Unificateur $\Rightarrow$ Solution
- Utilité: Utiliser les techniques de la programmation logiques pour des domaines autres que les termes symboliques.

À quoi est-ce que ça sert ?

- Problèmes combinatoires (jeux, ...)
- Problèmes de planification (par exemple un emploi de temps)
- Problèmes d'ordonnancement (par exemple trouver une affectation de tâches à exécuter à des machines)
- Problèmes de placement (par exemple placer des objets dans un volume limité)
- Tous ces problèmes avec en plus l'optimisation (utiliser un espace minimal, un temps minimal, un nombre minimal de machines, ...)

Comment est-ce que ça marche ?

- Le programmeur utilise des contraintes (formules logiques) pour modéliser son problème.
- L'interprète Prolog peut faire appel à des solutionneur de contraintes pour savoir si une contrainte a une solution ou pas.
- Il y a des solutionneur de contraintes pour des domaines différents : ("systèmes de contraintes") : arithmétique, domaine finis, ...
- Difficulté : en général ces solutionneurs ne sont pas complets !

Le rôle du programmeur

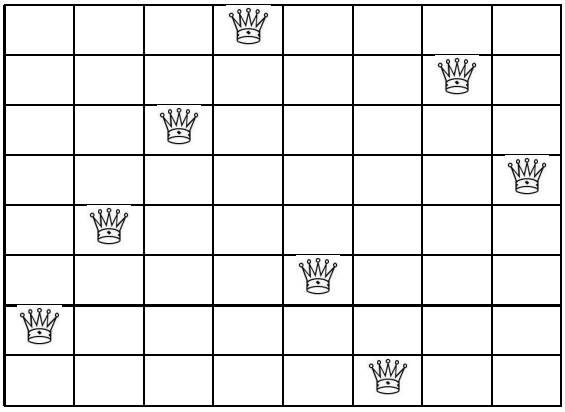
- Choisir le bon système de contraintes
- Choisir la bonne modélisation du problème par contraintes
- Programmer l'entrée/sortie
- Programmer la génération de contraintes
- Comprendre les conséquences de l'incomplétude du solutionneur de contraintes, programmer une stratégie de recherche.

Tout ça sera objet de ce cours ...

Exemple : Sudoku

		9			1	6	2	
5	7			2	8		3	
3			7					4
8	9			7		4		
	6		5		3		9	
		1		9			7	6
6					7			8
	4		1	3			6	5
	2	7	6			9		

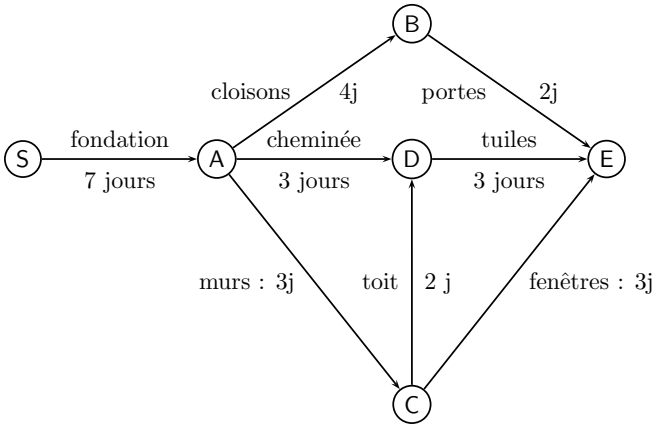
Exemple : le problème des n Reines



Exemple : coupures

- Étant données largeur et hauteur d'une vitre, ou d'une planche.
- Étant données les dimensions des morceaux qu'on souhaite obtenir.
- Est-ce qu'on peut obtenir les morceaux de la la vitre donnée ? (c.-à-d., est ce qu'on peut placer tous les morceaux sur la surface de départ, sans de se recouper ?)
- Est-ce aussi possible si on ne peut couper un morceaux que sur toute sa largeur ou toute son hauteur (cas de la vitre) ?

Exemple : Peut-on construire la maison en 14 jours ?



Exemple : Ordonnancement

- Exécuter des tâches sur plusieurs machines.
- Un ensemble de tâches est donné
  - avec des préséances (des tâches doivent être terminées avant des autres)
  - et des ressources partagées (des tâches ont besoin de la même machine)
- Déterminer pour toute tâche la machine et le temps de démarrage tels que
  - les contraintes sont satisfaites
  - le temps global est minimisé

Plan du cours (préliminaire)

- Introduction
- Généralités, contraintes arithmétiques sur $\mathbb{R}$
- Contraintes sur un domaine fini
- Optimisation
- Programmation logique avec contraintes

Plan de cette partie

Généralités, Contraintes Arithmétiques sur $\mathbb{R}$

- Contraintes : Syntaxe et sémantique
- Exemple : contraintes linéaires sur $\mathbb{R}$
- Programmation logique avec contraintes
- Contraintes non-linéaires, solutionneurs incomplets.

Contraintes : Syntaxe

- Donnée un langage de la logique du premier ordre :
  F: symboles de fonctions (et constantes);
  P: symboles de prédicat.
- *Contrainte simple* : Prédicat appliqué à des termes.
- *Contrainte* : conjonction de contraintes simples
  $C = c_1 \wedge c_2 \wedge \cdots \wedge c_k$
  Exemple $X \geq 42 \wedge X = Y + 2$
- Contraintes spéciales :
  - *true* : conjonction vide, toujours vraie
  - *false* : toujours fausse

Une contrainte est une formule de la logique du premier ordre.
(Normalement sans négation, disjonction, quantificateurs)

Système de Contraintes

- *Domaine* de contraintes : D. Par exemple : l'ensemble des entiers, l'ensemble des nombres réels, ...
- Un *système de contraintes* est donné par F, P, D et une interprétation des symboles en F et P.
- Par exemple: Système des contraintes numériques linéaires :

$$F = \{+, -, 0, 1, \ldots\}, \quad P = \{=, \leq, <, \geq, >, \neq\}, \quad D = \mathbb{N}$$

  Interprétations : comme d'habitude.
- Autre systèmes de contraintes : nombres réels, contraintes de Herbrand, contraintes de domaine fini, contraintes d'ordre, ...

Contraintes : Sémantique

- *Affectation* : fonction partielle des variables vers le domaine de contraintes.
- Une affectation θ *viole* une contrainte simple, si elle la rend fausse, et viole une contrainte si elle viole au moins une de ses contraintes simples.
- Une affectation θ est *consistante* pour une contrainte si elle ne la viole pas.
- *Solution* : une affectation totale et consistante
  - p.e. $X \geq 42 \wedge X = Y + 2$ a une solution $\theta = \{X \leftarrow 43, Y \leftarrow 41\}$
- C'est exactement la sémantique de la logique du premier ordre.

Contraintes : Satisfaisabilité, Équivalence

- Une contrainte est *satisfaisable*, si elle a une solution.
- L'ordre des contraintes simples peut être important, certains algorithmes dépendent de l'ordre.
- Pour $C = c_1 \wedge c_2 \wedge \cdots \wedge c_k$ on définit $\mathit{ensemble}(C) = \{c_1, c_2, \ldots, c_k\}$.
- Une contrainte c_1 *implique* une contrainte c_2 si toute solution de c_1 est aussi solution de c_2.
  Anglais : c_1 *entails* c_2.
- Deux contraintes sont *équivalentes* si elles ont le même ensemble de solutions (équivalence logique).

Problèmes de satisfaction de contraintes

Anglais *Constraint Satisfaction Problems – CSP*
Données :

- Les variables du problème avec leur domaines
- Une contrainte C

Questions :

- C est satisfaisable ?
- Donnez une solution, si C en a une.

Un solutionneur de contraintes répond à la première question.
Mais souvent aussi à la deuxième.

Satisfaction de contraintes

- Comment résoudre le problème de satisfaction de contrainte ?
- Approche naïve : essayer toutes les affectations
- ne marchera pas pour les réelles, entiers, etc.
- pour les domaines finis, on va essayer d'être plus intelligent.

On risque de rencontrer des limites :

- Non-décidabilité
- Complexité (problèmes NP-complets, ou pire)

Exemple : Équations linéaires sur $\mathbb{R}$

- Langage :
  - Constantes : $\mathbb{R}$ (on peut écrire toutes les constantes)
  - Fonctions : $+$ (binaire), $-$ (unaire et binaire), $*$ (binaire)
  - Prédicats : $=$ (binaire)
- Pour l'instant restriction à des termes arithmétiques *linéaires* : pas de produits entre variables.
- Domaine : $\mathbb{R}$
- Interprétation : comme d'habitude.

Exemple : Équations linéaires sur $\mathbb{R}$

Exemples de contraintes arithmétiques linéaires :

- $X = Y + Z \wedge Y = 1 + Z$
- $2 * Y = 17 * (X + 42) - 3 * X$

Ne sont *pas* de contraintes arithmétiques linéaires :

- $X = 5 * Y * Z$
- $Y = X * (42 + Z)$
- $2 * X + Y * Y = 3 * Z + Y * Y$

Exemple : Résolution de contraintes arithmétiques linéaires

- Forme résolue : $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ où
  - $x_i \not\equiv x_j$ si $i \neq j$
  - $x_i \notin \mathcal{V}(t_j)$ pour tous i, j.
- Toute forme résolue est satisfaisable en $\mathbb{R}$.
- $x_1, \ldots, x_n$: variables *déterminées*
- On a même le droit de choisir les valeurs des variables non déterminées.
- En général : définition des formes résolues fait partie du solutionneur de contraintes.

Formes résolues

Exemple d'une forme résolue :

$$\begin{aligned} x_1 &= 2 * y + 5 * z \\ x_2 &= 3 - y - z \\ x_3 &= 42 * y - 17 * z \end{aligned}$$

Une solution est:

$$y \mapsto 1, z \mapsto 1, x_1 \mapsto 7, x_2 \mapsto 1, x_3 \mapsto 25$$

On peut même, pour n'importe quel choix de valeurs pour y et z, trouver des valeurs de x_1, x_2, x_3 telles que les équations sont satisfaites.

Formes résolues

N'est *pas* une forme résolue :

$$\begin{aligned} x_1 &= 2 * y + 5 * z \\ 17 &= 42 \end{aligned}$$

N'est *pas* une forme résolue :

$$\begin{aligned} x_1 &= 2 * x_2 + 5 * z \\ x_2 &= 3 - y - x_3 \\ x_3 &= 42 * y - 17 * x_1 \end{aligned}$$

Résoudre des contraintes arithmétiques linéaires

- L'algorithme est donné par des *règles de transformation*.
- On applique les règles tant que possible, dans n'importe quel ordre.
- Si on ne peux plus appliquer une règle on s'arrête, et on renvoie la contrainte obtenue.
- *Équation normalisée :* Soit une équation entre deux constantes, soit une équation de la forme $x = t$ où $x \notin \mathcal{V}(t)$.
- Exemple d'une équation normalisée : $x = 17 + 3 * y + 5 * z$.
- On peut transformer toute équation linéaire en une équation normalisée qui lui est équivalente.

Résoudre des contraintes arithmétiques linéaires

- Règle 1 : Choisir une équation non normalisée, et la normaliser.
- Règle 2 : S'il y a une équation $c_1 = c_2$, où c_1 et c_2 sont des constantes différentes, alors remplacer toute la contrainte par $\bot$.
- Règle 3: S'il y une équation $c = c$, où c constante, la supprimer.
- Règle 4: S'il y a une équation normalisée $x = t$ (avec $x \notin \mathcal{V}(t)$) et x parait dans des autres équations alors remplacer dans toutes les *autres* équations x par t.

Exemple

$$\begin{aligned} x + 1 &= y + 2 \\ y + 3 &= z + 4 - 2x \\ z + 2 &= 2x + u \end{aligned}$$

En résout la première équation pour x, et remplace x par $y + 1$:

$$\begin{aligned} x &= y + 1 \\ 3y + 3 &= z + 2 \\ z + 2 &= 2y + 2 + u \end{aligned}$$

En résout la deuxième équation pour z, remplace z par $3y + 1$, résout la dernière équation pour u, et obtient une forme résolue :

$$\begin{aligned} x &= y + 1 \\ z &= 3y + 1 \\ u &= -3y - 1 \end{aligned}$$

Correction du solutionneur

- Toute règle est une transformation d'équivalence (les deux contraintes sont équivalentes)
- L'application de règles termine toujours : trouver un ordre de terminaison.
- Si aucune règle est applicable alors on a soit $\bot$, soit une forme résolue.

Contraintes réelles en Yap

Avec la bibliothèque `clpr` :

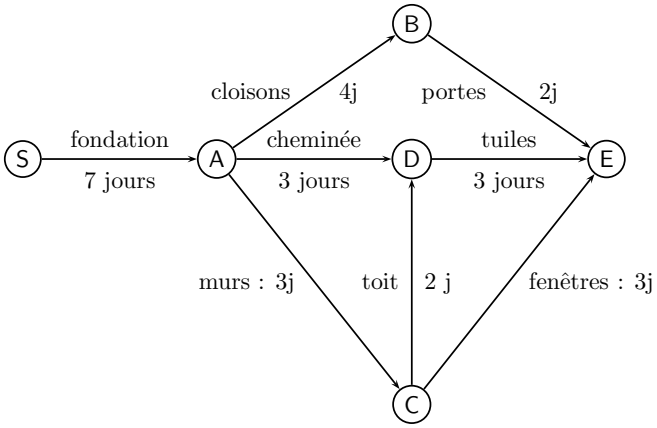
$$\begin{aligned} F &= \{+, -, *, /, \sin, \cos, \tan, \ldots\} \\ P &= \{=, <, >, =<, >=, =:=, \ldots\} \\ D &= \text{les nombres réelles} \end{aligned}$$

- Pas oublier : `(use_module(library(clpr)))`
- Écrire des contraintes entre accolades { et }
- Écrire une virgule pour la conjonction logique.

L'exemple en Yap Prolog

```
   ?- (use_module(library(clpr))).
 % reconsulting /usr/share/Yap/clpr.yap...
 % including clpr.pl...
 % clpr.pl included in module user, 60 msec 1703936 bytes
 % reconsulted /usr/share/Yap/clpr.yap in module clpr, 76 r
yes
   ?- {X+1=Y+2,Y+3=Z+4-2*X,Z+2=2*X+U}.
{Z= -2.0+3.0*U}
{X=U}
{Y= -1.0+U}
no
```

Construction de la maison en $\leq$ 14 jours



Modélisation

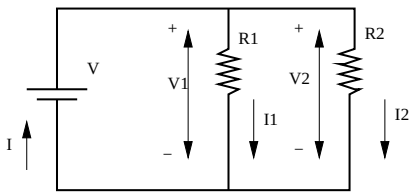
- Extension du système de contraintes : $<, \leq$
- On introduit une variable T par tâche T qui dénote le début de cette tâche.
- La fin de la tâche T de durée d_T est $T + d_T$.
- Si tâche R dépend de la tâche S : $S + d_S \leq R$.
- Toutes les tâches ≥ 0.
- Toutes les fins de tâches ≤ 14.

La contrainte pour la construction de la maison :

$$
\begin{array}{llll}
 & \text{Fond} \geq 0 & & \wedge\ \text{Fonds} + 7 \leq 14 \\
\wedge & \text{Cloisons} \geq 0 & \wedge\ \text{Cloisons} \geq \text{Fond} + 7 & \wedge\ \text{Cloisons} + 4 \leq 14 \\
\wedge & \text{Cheminée} \geq 0 & \wedge\ \text{Cheminée} \geq \text{Fond} + 7 & \wedge\ \text{Cheminée} + 3 \leq 14 \\
\wedge & \text{Murs} \geq 0 & \wedge\ \text{Murs} \geq \text{Fond} + 7 & \wedge\ \text{Murs} + 3 \leq 14 \\
\wedge & \text{Toit} \geq 0 & \wedge\ \text{Toit} \geq \text{Murs} + 3 & \wedge\ \text{Toit} + 2 \leq 14 \\
\wedge & \text{Portes} \geq 0 & \wedge\ \text{Portes} \geq \text{Cloisons} + 4 & \wedge\ \text{Portes} + 2 \leq 14 \\
\wedge & \text{Fenêtres} \geq 0 & \wedge\ \text{Fenêtres} \geq \textit{Murs} + 3 & \wedge\ \text{Fenêtres} + 3 \leq 14 \\
\wedge & \text{Tuiles} \geq 0 & \wedge\ \text{Tuiles} \geq \text{Cheminée} + 3 & \wedge\ \text{Tuiles} + 3 \leq 14 \\
 & & \wedge\ \text{Tuiles} \geq \text{Toit} + 2 &
\end{array}
$$

Il n'y a pas de solution.

Exemple: Modélisation d'un circuit



$V1 = I1 * R1 \wedge V2 = I2 * R2 \wedge$
$V - V1 = 0 \wedge V - V2 = 0 V1 - V2 = 0 \wedge$
$I - I1 - I2 = 0 \wedge -I + I1 + I2 = 0$

Ne sont pas de contraintes numériques

en Prolog:

- e_1=:=e_2 car il faut que e_1, e_2 soient closes.
- e_1=e_2 car Prolog traite les deux expressions de façon *syntaxique* (unification).
- X `is` e car il faut que e soit close, et X une variable.

Les contraintes expriment des *relations*.

Intégration des contraintes dans la programmation logique

- *Atome :* Prédicat, ou contrainte
- *Configuration :* liste d'atomes, plus une contrainte résolue
  Notée : $l \mid c$
- Tant que la liste l n'est pas vide : configuration $a, l \mid c$
  - si le premier atome est un prédicat : le remplacer par le corps d'une clause de sa définition (comme Prolog)
  - si le premier atome est une contrainte : appliquer le solutionneur de contraintes à la contrainte $a \wedge c$.
    - Si résultat $\bot$: échec.
    - Si résultat est une forme résolue c' : passer à la configuration $l \mid c'$.
- Donne lieu à un arbre de recherche comme Prolog.

Exemple : contraindre la somme d'une liste

```
:- use_module(library(clpr)).

listsum([],X) :- {X=0}.
listsum([H|R],X) :- {X = H + XR}, listsum(R,XR).
```

Exemple : contraindre la somme d'une liste

```
   ?- listsum([2,3,4],X).
X = 9.0 ?
yes
   ?- listsum([2,X,4],9).
X = 3.0 ?
yes
   ?- listsum([2,X,Y],9).
{X=7.0-Y}
no
   ?- listsum(L,9).
L = [9.0] ? ;
L = [_A,_B] ? ;
L = [_A,_B,_C] ? ;
```

Exécution du programme `listsum`

```
listsum([],X) :- {X=0}.
listsum([H|R],X) :- {X = H + XR}, listsum(R,XR).
```

$\text{listsum}([2, Y], 5)$	$\mid \top$
$\{5 = 2 + XR\}, \text{listsum}([Y], XR)$	$\mid \top$
$\text{listsum}([Y], XR)$	$\mid XR = 3$
$\{XR = Y + XR'\}, \text{listsum}([], XR')$	$\mid XR = 3$
$\text{listsum}([], XR')$	$\mid Y = 3 - XR'$
$\{XR' = 0\}$	$\mid Y = 3 - XR'$
	$\mid Y = 3$

Contraintes non-linéaires

- Maintenant on permet des équations arithmétiques quelconques, pas nécessairement linéaires.
  Par exemple $X + (Y * Z) = 3 * Z * Z * Z + 2 * Y * Y$
- Il est toujours *théoriquement* possible d'écrire un solutionneur de contraintes (résultat de Tarski, 1951). Mais cet algorithme a une complexité catastrophique.
- C'est possible car il s'agit des nombres *réels*.
- Le problème est *non décidable* quand on change le domaine en $\mathbb{N}$ (résultat de Yu Matijacevič, 1970).

Solutionneurs incomplets

- En général, un solutionneur peut être *incomplet*.
- Un solutionneur peut donner trois réponses possibles :
  - « non » (ou $\bot$)
  - « oui » (ou une forme résolue)
  - « je ne sais pas » (ou une formule seulement partiellement résolue)
- Dans le cas du solutionneur pour $\mathbb{R}$: les équations qui contiennent des produits entre variables ne peuvent pas être traitées (sauf si l'équation devient linaire à cause de l'instantiation de variables).

Intégration de solutionneurs incomplets en Prolog

- Quand une contrainte ne peut pas être traitée par le solutionneur elle reste en suspense, et Prolog continue sur l'atome suivante.
- Quand la contrainte résolue est modifié, les contraintes en suspense sont examinées de nouveau.
- Implémentation plus efficace : maintenir une liste de contraintes en suspense par variable, reexaminer seulement les contraintes en suspense qui contiennent une variable pour laquelle la contrainte résolue a des nouvelles informations.

Exemple : somme des carrés

```
listsqsum([],X) :- {X=0}.
listsqsum([H|R],X) :- {X = H*H + XR}, listsqsum(R,XR).

   ?- listsqsum([2,3,4],X).
X = 29.0 ?
yes
   ?- listsqsum([2,X,4],29).
{9.0-X^2.0=0.0}
no
```

Exemple : Approximation de la racine carré

```
:- use_module(library(clpr)).

root(0,0).
root(Input,Result) :-  Input >= 1 , newton(Input,Result,1)

newton(Input,Result,Current) :-
           Current*Current = Input, Result=Current.
newton(Input,Result,Current) :-
           Next = Current/2 + Input/(2*Current) ,
          newton(Input,Result,Next).
```

- Parfois, des équations non-linéaires peuvent devenir linéaires par simplification.
  Exemple : $X + Y * Y = 3 + Z + Y * Y$
- Souvent (comme en Yap): Nombres réels avec une précision limitée.
  Conséquence : Erreurs d'arrondi, des termes peuvent « disparaître » quand un coefficient devient 0.
  Cela peut aussi rendre un terme linéaire.
- Il n'est pas toujours évident si le solutionneur va arriver à résoudre une contrainte ou pas.
- Le Cut (!) devient encore plus dangereux.