Programmation Déclarative

Programmation Par Contraintes

Nicolas Barnier
nicolas.barnier@enac.fr

ENAC

2020–2021

Objectifs

- Connaître le formalisme des **problèmes de satisfaction de contraintes** (CSP)
- Savoir établir la **cohérence d'arc** sur un CSP
- Connaître les algorithmes de résolution **Branch & Prune** et **Branch & Bound**
- Savoir **modéliser** un problème d'**optimisation combinatoire** avec un programme en contraintes et développer un **solveur**
- Expérimenter diverses **stratégies de recherche** de solution

Problème de satisfaction de contraintes

Contexte

- **Recherche opérationnelle** (RO) : allocation de ressources, scheduling, tournées de véhicules (VRP $\supset$ TSP), configuration, rotation de personnel...
- **Intelligence artificielle** : SAT, puzzle logique, graphes (coloration, clique, couverture), partitionnement...

Problèmes **non-linéaires**, en **nombres entiers** (discrets), disjonction, combinaisons arbitraires...

Optimisation combinatoire

- **Contraintes** : propriétés que doit vérifier une solution
- **Satisfaction** : difficulté de construire une solution admissible (NPC)
- **Optimisation** : difficulté de trouver une solution optimale (NPC)
- **CSP** : formalisme de modélisation

Programmation par contraintes

Résolution exacte des CSP

- **Solveur** de contraintes : extension de la **programmation logique** à diverses structures mathématiques CLP(X)
- Paradigme **déclaratif** : séparation de la **spécification** du problème et des **algorithmes de résolution**
- Problème $\xrightarrow{\text{Modélisation}}$ CSP $\xrightarrow{\text{PPC}}$ Programme Stratégie $\xrightarrow{\text{Résolution}}$ Solution(s)
- Programme : **variables**, **contraintes**
- Stratégie de recherche : **but** de résolution
- Algorithmes de résolution **exacts** :
  - **preuve d'absence** de solution
  - obtention possible de **toutes les solutions**
  - **preuve d'optimalité**
- Pas que Prolog : IBM CP Optimizer/C++, Choco/Java, FaCiLe/OCaml

Plan du cours

Problème de satisfaction de contraintes

Définition (CSP / Réseau de contraintes)

Un CSP *ou* réseau de contraintes *est défini par un triplet* (X, D, C) *:*

- $X = \{x_1, \dots, x_n\}$ *est l'ensemble des* variables *(inconnues).*
- *Chaque variable* $x_i \in X$ *est associée à un* domaine $d_i \in D$ *des valeurs qu'elle peut prendre.*
- C *est l'ensemble des* contraintes*. Chaque contrainte* $c \in C$ *est définie sur un sous-ensemble de variables* $X_c \subseteq X$ *par une relation* $R_c \subset \prod_{x_i \in X_c} d_i$ *spécifiant les combinaisons de valeurs autorisées pour les variables de* X_c*.*

Les contraintes peuvent être :

- définies en **extension** : tuples autorisées (ou interdits)
- **arithmétique** : $+, \times, /, \dots, <, \leq, =, \neq$
- **globales/symboliques** : AllDiff, indexation, cardinalité. . .
- **méta** (logiques) : $\vee, \Rightarrow \dots$ ou **réifiées** : variables 0/1

Solution

Définition (Affectation partielle)

Une **affectation partielle** ϕ_V *sur un sous-ensemble de variable* $V \subseteq X$ *est une fonction telle que* $\phi_V(x_i) \in d_i$.

Définition (Satisfaction de contrainte)

Une affectation partielle ϕ_V **satisfait** *une contrainte* $c \in C$ *telle que* $X_c \subseteq V$ *ssi* $\phi_V(X_c) \in R_c$ *(sinon elle la viole).*

Définition (Solution)

Une **solution** *d'un CSP est une affectation totale* ϕ_X *qui satisfait toutes les contraintes de* C.

Résoudre un CSP : trouver une/toutes/la meilleure solution ou prouver $\nexists$

Domaines

Cadre générique Constraint Logic Programming CLP(X)

- Arbres finis (Prolog)
- **Domaines finis** (entiers)
- Rationnels, réels (flottants)
- Ensembles finis : $s \in [\emptyset, \{1, 2, 3\}]$, i.e. $\emptyset \subseteq s \subseteq \{1, 2, 3\}$
- Graphes...

Puzzles logiques

Arithmétique cryptée

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
 & + & M & O & R & E \\
\hline
 & M & O & N & E & Y
\end{array}
$$

Sudoku

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

Générer et tester (au fur et à mesure)

Backtracking (BT)

BT(V, ϕ) : *bool*

- **if** $V = \emptyset$ **then** return true;
- $x \in V$;
- **for** $a \in d_x$ **do**
  - $\phi' \leftarrow \phi \cup \{(x, a)\}$;
  - **if** ϕ' *ne viole aucune contrainte* **then**
    - **if** *BT*($V \setminus \{x\}$, ϕ') **then** return true;
- return false;

On ne vérifie que les contraintes dont toutes les variables sont affectées

Filtrage

Suppression des valeurs incohérentes

- BT peu efficace sur les CSP : on vérifie les contraintes « trop tard »
- **Look-ahead** : quand une variable est affectée ou que son domaine est restreint, on peut déduire que certaines valeurs ne peuvent pas faire partie d'une solution
- Une valeur qui ne peut pas faire partie d'une solution peut être supprimée : **élagage** de l'arbre de recherche
- Le domaine de chaque variable est **mémorisé et maintenu** au cours de la recherche, i.e. **filtré** et **rétabli** en cas de retour arrière

Définition (Support d'une valeur sur une contrainte)

Un **support** *pour une valeur* $a \in d_i$ *de* x_i *sur une contrainte binaire* c *avec* $X_c = \{x_i, x_j\}$ *est une valeur* $b \in d_j$ *telle que* $(a, b) \in R_c$.

C'est une **justification** pour conserver la valeur dans le domaine

Cohérence d'arc (*Arc-Consistency*)

Définition (Cohérence d'arc)

Une contrainte binaire c *vérifie la* **cohérence d'arc** *ssi toutes les valeurs de* d_i *et de* d_j *ont un support sur* c.
Un CSP vérifie la **cohérence d'arc** *ssi toutes ses contraintes la vérifient.*

Filtrage de d_i

Revise(x_i,x_j) : *bool*

```
modif ← false;
for a ∈ d_i do
    if a n'a pas de support dans d_j sur c then
        d_i ← d_i \ {a};
        modif ← true;
return modif;
```

Établissement de la cohérence d'arc

AC-3 [Mackworth 77]

AC3(C)

$Q \leftarrow \{(x_i, x_j), (x_j, x_i),\ \forall c \in C\}$;
while $Q \neq \emptyset$ **do**
  $(x_i, x_j) \in Q$;
  $Q \leftarrow Q \setminus \{(x_i, x_j)\}$;
  if *Revise*(x_i, x_j) **then**
    $Q \leftarrow Q \cup \{(x_k, x_i),\ \forall c \in C \text{ t.q. } X_c = \{x_i, x_k\}, k \neq j\}$;

- **Propagation de contraintes**
- Un « arc » peut être « révisé » plusieurs fois
- Complexité : $O(md^3)$
- Améliorations : AC-4 [Mohr 86] en $O(md^2)$, AC-6 [Bessière 93], GAC [Bessière 97]...

Branch & Prune (séparation et élagage)

BT + filtrage à chaque affectation

- La cohérence d'arc **ne suffit pas** (en général) pour résoudre un CSP
- Niveau de filtrage : **compromis** entre le temps passé dans la propagation des contraintes et la puissance d'élagage
  - cohérence d'arc (plus besoin de tester la cohérence locale)
  - cohérence de bornes : contraintes arithmétiques (intervalles)
  - approximations : Forward Checking...
- Dès qu'un domaine est **vidé** par filtrage : **échec** (retour arrière)

Maintaining Arc-Consistency (MAC)

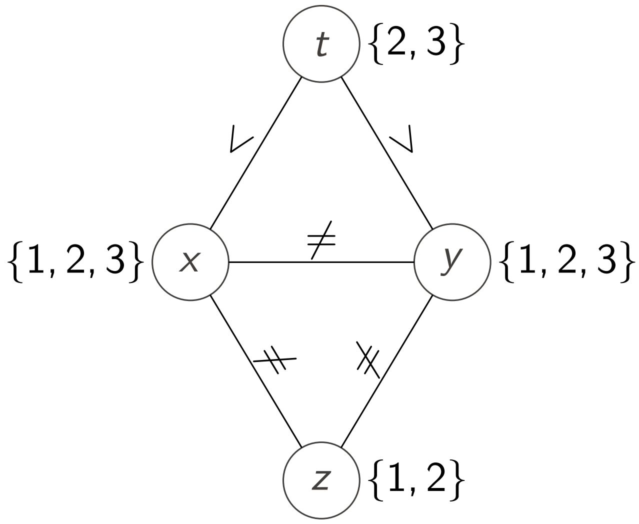
Établissement **incrémental** de la cohérence d'arc à chaque affectation :

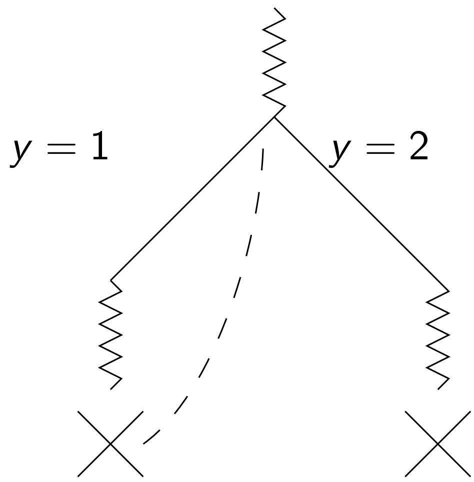
- on ne filtre une contrainte que si l'une de ses variables a été modifiée
- conditions de propagation (affectation, bornes, domaine)
- maintien de structures de données internes

Résolution et graphe des contraintes

Graphe des contraintes : $(X, \{X_c, c \in C\})$

$$x, y \in \{1,2,3\} \qquad z \in \{1,2\} \qquad t \in \{2,3\}$$
$$x \neq y \qquad y \neq z \qquad z \neq x \qquad t > x \qquad t > y$$





Ordonnancement des variables et des valeurs

Heuristiques

Choix de :

- la **variable** à affecter : **first-fail principle**
- la **valeur** d'affectation : celle qui a le plus de chance de mener à une solution

Exemple :

$$\begin{aligned} X &= \{x, y, z, t\} \\ D &= \{[1..2], [1..2], [1..2], [1..100]\} \\ C &= \{x \neq y, x \neq z, y \neq z\} \end{aligned}$$

Heuristiques d'ordonnancement des variables

- Statique :
  - peu efficace
  - exemple : les items d'un Knapsack triés par efficacité
- **Dynamique** (Dynamic Variable Ordering) :
  - robuste
  - s'adapte à l'état de la rercherche
  - exemple : taille de domaine minimale
- Plusieurs critères :
  - si plusieurs variables ont la même évaluation
  - exemple : (min-size, max-degré), cf. DSATUR
- Spécifique au domaine :
  - exemple : ressource critique et ranking de tâches pour le *scheduling*
- Apprentissage :
  - prend en compte les échecs précédents
  - exemple : *weighted degree* [Lecoutre 04]

Optimisation

Caractérisation des solutions

- En général, plusieurs (voire beaucoup de) solutions : choix
- Préférences : consommation de ressources, distance...
- **Coût** : fonction des variables du CSP

$$\text{cost} = f(x_1, \ldots, x_n) \quad \text{avec} \quad f = \max, \sum, \text{card} \ldots$$

Branch & Bound (& Prune)

- Contrainte **dynamique** cost $<$ ub (pour une minimisation) mise à jour après chaque solution de coût ub trouvée
- **Preuve d'optimalité** : pas de solution pour cost $<$ opt
- **Borne inférieure** : preuve d'optimalité dès que cost $=$ lb
- Si l'intervalle du coût est grand et qu'il y a de nombreuses solutions intermédiaires : recherche **dichotomique**

GNU Prolog

Système Prolog Open Source avec solveur de contraintes

- Daniel Diaz @ INRIA, 1999
- **Interpréteur** (top-level, boucle d'interaction) : `gprolog`
  interactif, debugger, lent
- **Compilateur** en code natif : `gplc`
  optimisé pour un processeur cible (rapide), exécutable (autonome)
- Prolog + Contraintes sur les domaines finis
- Autres solveurs Prolog : Prolog IV, ECL^iPS^e, SICStus...
- Solveurs hybrides : Mozart, Mercury...

Variables à domaine fini (FD)

Nouveau type de variables logiques

- Substitution dans un **domaine entier** (associé à la variable).
- Le domaine d'une variable ne peut qu'être **réduit** (inclusion).

Déclaration

```
fd_domain(VarList_or_Var, LB, UB)
fd_domain(VarList_or_Var, IntList)
fd_domain_bool(VarList_or_Var)
```

Variables à domaine fini (FD)

Utilisation

Si un prédicat attend une variable FD comme argument, on peut utiliser :

- une variable classique (sans domaine) : le domaine $[0, +\infty]$ lui est associé
- un entier : équivalent à une variable FD avec un domaine singleton

Accès

```
fd_min(Var, LB)       fd_max(Var, UB)
fd_size(Var, Size)    fd_dom(Var, IntList)
```


Contraintes

Relation entre des variables

- Logiquement équivalente à un prédicat
- La différence est opérationnelle :
  - les buts sont **résolus** immédiatement
  - les contraintes sont **satisfaites** (approximation par arc-consistance ou autre, e.g. B-consistance) en **coroutines**, *i.e.* quand le domaine d'une variable est **réduit**, les contraintes concernées sont **réveillées**
  - en réduisant les domaines, on dit qu'une contrainte effectue une **propagation**

Arithmétiques

Expressions

Combinaison d'entiers, variables FD et opérateurs :

`+   -   *   //   **   rem`

Contraintes

Entre deux expressions arithmétiques :

`#=   #\=   #<   #=<   #>   #>=`

```
|?- fd_domain(X,0,10), X**2-5*X+4 #= 0, fd_labeling(X).
X = 1 ? ;
X = 4
|?- fd_domain([X,Y],0,10), X #< Y.
X = _#3(0..9)
Y = _#25(1..10)
```


Contraintes booléennes

Contraintes sur des variables booléennes

Avec des variables de domaine [0..1] :

`#\   #<=>   #==>   #/\   #\/   ##`

Réification

Contrainte considérée comme une variable

Si une contrainte arithmétique est utilisée à la place d'une expression booléenne, elle est **réifiée** : elle n'est pas imposée (i.e. satisfaite) mais associée à une (nouvelle) variable booléenne :

- instanciée à 1 ssi elle est vérifiée
- instanciée à 0 ssi elle est violée
- de domaine [0..1] sinon

```
| ?- fd_domain([X,Y], 0, 10), X #=< Y #<=> B.
B = _#44(0..1)  X = _#3(0..10)  Y = _#25(0..10)
| ?- fd_domain([X,Y], 0, 10), X #=< Y #<=> B, B #= 0.
B = 0  X = _#3(1..10)  Y = _#25(0..9)
| ?- fd_domain([X,Y], 0, 10), X #=< Y #<=> B, X #< 4, Y #> 6.
B = 1  X = _#3(0..3)  Y = _#25(7..10)
```


Réification

Disjonction de contraintes (cf. ordonnancement)

```
taches_exclusives(T1, D1, T2, D2):-
    T1+D1 #=< T2 ## T2+D2 #=< T1.
| ?- fd_domain([X,Y], 1, 10),
     taches_exclusives(X,5,Y,5), X #< 5.
X = _#3(1..4)  Y = _#25(6..10)
```

Contrainte de cardinalité

- `fd_cardinality(CstrList, Card)` : `Card` est égale au nombre de contraintes vérifiées dans `CstrList`
- `fd_at_least_one(CstrList)`
- `fd_at_most_one(CstrList)`
- `fd_only_one(CstrList)`

Contraintes globales

- `fd_element_var(I, VarList, Var)` : indexation,
  i.e. `Var` est le `I`ème élément de `VarList`
- `fd_all_different(VarList)` : toutes différentes.
- `fd_atmost|fd_atleast|fd_exactly(N, VarList, Val)` :
  au plus, au moins, exactement `N` variables de `VarList` sont égales à l'entier `Val`.

Contraintes en extension

`fd_relation(IntListList, VarList)` :
`IntListList` sont les tuples autorisées pour les variables de `VarList`

```
and(X,Y,Z):-
    fd_relation([[0,0,0],[0,1,0],[1,0,0],[1,1,1]],
                                               [X,Y,Z]).
```

Étiquetage

Instanciation des variables de décision du CSP

- `fd_labeling(VarList_or_Var)` : variable inconnue la plus à gauche, plus petite valeur
- `fd_labeling(VarList_or_Var, Options)` : `Options` est une liste de termes qui modifie la stratégie de recherche
  - sur l'ordre des variables `variable_method(V)`, avec `V` :
    - `first_fail` : plus petit domaine
    - `most_constrained` : `first_fail` + le plus de contraintes
    - `smallest` : plus petite valeur + le plus de contraintes
    - `random`
  - sur l'ordre des valeurs `value_method(V)`, avec V parmi :
    - `min`, `max`, `middle`, `bounds`, `random`

Optimisation

Obtenir la meilleure solution

- Définition d'un coût qui dépend des variables de décision
- Branch & Bound (BT avec contrainte dynamique sur le coût)

`fd_minimize(Goal, Cost) / fd_maximize(Goal, Cost)`

- `Goal` est un but (ordre supérieur) qui **doit** instancier le coût `Cost`.
- Typiquement : `fd_minimize(fd_labeling(Vars), Cost)`

Modélisation : les n reines

Problème

Placer n reines sur un échiquier (de $n \times n$ cases) sans qu'aucune n'en menace une autre, i.e. sans que deux reines soient sur une même horizontale, verticale ou diagonale.

N-Queens $\notin$ NPC, mais souvent utilisé dans les benchmarks

Différentes modélisations

Différents :

- domaines
- nombres de variables
- tailles d'espace
- contraintes
- stratégies

dont dépendent les :

- performances
- tailles de problème traitable

Des booléens

Variables : $n \times n$ variables booléennes $Q_{i,j} \in [0..1], 0 \leq i, j < n$
Contraintes :

- seulement n reines : $\sum_{i,j} Q_{i,j} = n$
- pas de prise sur les lignes, colonnes et diagonales :

$$
\begin{array}{lcl}
\forall i \forall j & 0 < k < n - i & Q_{i,j} + Q_{i+k,j} < 2 \\
\forall i \forall j & 0 < k < n - j & Q_{i,j} + Q_{i,j+k} < 2 \\
\forall i \forall j & 0 < k < min(n - i, n - j) & Q_{i,j} + Q_{i+k,j+k} < 2 \\
\forall i \forall j & 0 < k < min(n - i, j + 1) & Q_{i,j} + Q_{i+k,j-k} < 2
\end{array}
$$

Des couples

Dans le modèle précédent, peu de variables prennent la valeur vrai ; il y a beaucoup plus de cases vides que de cases occupées.
On ne représente que les positions des reines :

- $2 \times n$ variables : $(X_i, Y_i) \in [0..n-1]^2, 0 \leq i < n$
- pas de prise sur les lignes, les colonnes et diagonales :

$$\begin{array}{lll}
\forall i \forall j & i < j & X_i \neq X_j \\
\forall i \forall j & i < j & Y_i \neq Y_j \\
\forall i \forall j & i < j & X_j - X_i \neq Y_j - Y_i \\
\forall i \forall j & i < j & X_i - X_j \neq Y_j - Y_i
\end{array}$$

Des entiers

Les résultats du modèles précédents

```
[0] [1] [2] [3] [4] [5] [6] [7]
[0] [4] [7] [5] [2] [6] [1] [3]
```

suggèrent de ne positionner qu'une reine par ligne :

- n variables : $C_i \in [0..n-1], 0 \leq i < n$
- pas de prise sur les colonnes et diagonales :

$$\begin{array}{lll}
\forall i \forall j & i < j & C_i \neq C_j \\
\forall i \forall j & i < j & C_j - C_i \neq j - i \\
\forall i \forall j & i < j & C_j - C_i \neq i - j
\end{array}$$

```
queens(N, L):-
    length(L, N),
    fd_domain(L, 1, N),
    fd_all_different(L), % colonnes
    constrain_queens(L), % diagonales
    fd_labeling(L).

constrain_queens([]).
constrain_queens([Q|Qs]):-
   safe(Q, Qs, 1),
   constrain_queens(Qs).

safe(_, [], _).
safe(Q1, [Q2|Qs], I):-
   Q1 - Q2 #\= I,
   Q2 - Q1 #\= I,
   I1 is I+1,
   safe(Q1, Qs, I1).
```