INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Avril 2021
PERIER Nathan

Bilan personnel du cours « Programmation par contraintes »

I. Résumé du cours

1. Problèmes de satisfaction de contraintes

Un problème à satisfaction de contraintes $\langle X, D, C \rangle$ est consistué de trois ensembles finis :

- $X = \{X_1, X_2, \dots, X_n\}$ l'ensemble des variables du problème
- $D = \{D_1, D_2, \dots, D_n\}$ l'ensemble des domaines de ces variables (i.e. chaque variable X_i doit prendre une valeur dans D_i)
- C l'ensemble des contraintes, qui sont des règles réduisant le nombre de possibilités sur les valeurs que peuvent prendre les variables

On définit une évaluation des variables comme une fonction qui à tout élément $X_i \in X$ associe une unique valeur dans D_i. Une solution du problème est une évaluation qui satisfait l'ensemble des contraintes de C.

L'objectif du problème peut être de trouver une solution quelconque, une solution optimale, l'ensemble des solutions ou de déterminer s'il existe ou non des solutions.

2. Défis des problèmes à contraintes

Ces problèmes sont très souvent NP-complets, donc on n'a à priori pas d'algorithmes qui permet de les résoudre en un temps raisonnable dans tous les cas. Cela ne signifie pas pour autant que la recherche sera nécessairement en temps exponentiel, c'est vrai dans le pire des cas mais on peut essayer de trouver des stratégies pour aller plus vite.

Le temps de résolution (pour une taille de problème donnée) va dépendre de différents paramètres, qui peuvent être adaptés afin de résoudre les problèmes plus rapidement :

- L'ordre dans lequel les variables sont traitées
- L'ordre dans lequel les contraintes sont traitées
- L'ordre dans lequel les valeurs du domaine sont essayées

Il n'y a cependant pas de méthode générale pour trouver les bonnes heuristiques à utiliser, il faut réfléchir au cas par cas.

L'ajout de contraintes redondantes peut également accélérer la recherche car certaines valeurs seront éliminées plus rapidement.

Il est à noter que si les points évoqués ci-dessus ont une influence sur la vitesse à laquelle on est capable de résoudre le problème, les solutions que l'on va obtenir à la fin seront les mêmes.

Un autre aspect des problèmes à contraintes auquel il faut être vigilent est la présence de symétries dans la structure du problème. S'il en existe, alors on va passer de temps à chercher des solutions qui n'apportent pas d'informations en plus. Par exemple, si on a un problème de la forme $P = A + B$ (où A et B ont les mêmes contraintes) et qu'on trouve la solution $\{A = 1,\ B = 2\}$, alors $\{A = 2,\ B = 1\}$ est aussi une solution. Pour éviter de passer du temps à énumérer des solutions redondantes, on peut poser des contraintes supplémentaires qui vont supprimer les symétries.

3. De « Generate and Test » à « Constraint and Generate »

En Prolog de base, on va essayer toutes les valeurs possibles dans l'ordre jusqu'à tomber sur une solution. Ce n'est pas très efficace et peut s'avérer très long en fonction du nombre de variables et de la taille du problème. On va donc essayer d'utiliser les contraintes pour parcourir l'espace de recherche plus efficacement en éliminant d'abord les valeurs impossibles, puis en itérant sur ce qu'il reste.

Quand on pose une contrainte, on va tout d'abord propager cette contrainte pour réduire les domaines des variables. Quand toutes les contraintes sont propagées, on va essayer d'attribuer une valeur à chaque variable afin de constituer une évaluation. À chaque attribution de valeur (étiquetage), on continue de propager les contraintes pour réduire encore plus les domaines des variables. Si le domaine d'une variable devient vide, alors la solution est invalide et on utilise le backtracking pour la supprimer de l'espace de recherche. Si on parvient à réduire le domaine de chaque variable à une seule valeur après application de toutes les contraintes, alors on a trouvé une solution au problème.

Les algorithmes suivants peuvent être utilisés pour propager les contraintes :

Cohérence de nœuds

La cohérence de nœuds consiste à utiliser une contrainte portant sur une unique variable pour en réduire le domaine. Pour cela, on va prendre chaque valeur du domaine de la variable et ne garder que celles qui satisfont la contrainte. Il est à noter que le domaine est un ensemble fini de valeurs à une dimension.

Plus précisément, on dit qu'une contrainte $c \in C$ est cohérente par nœuds avec un domaine D si elle ne fait intervenir qu'une seule variable x et que pour tout $d \in D(x)$, l'assignation de d à x est une solution de C.

Cohérence d'arcs

La cohérence d'arcs ressemble à la cohérence de nœuds mais sur des contraintes binaires. On ne travaille donc à présent sur un espace fini en deux dimensions, dont on va tester chacune des valeurs pour déterminer lesquelles sont à garder.

On peut généraliser cette idée avec la cohérence d'hyperarcs (ou cohérence d'arcs généralisée) qui travaille avec des contraintes n-aires ($n > 2$). Il est cependant à noter que c'est une approche qui est coûteuse en temps (NP-difficile) et qui va demander de stocker de grands domaines (puisqu'on travaille avec des ensembles à n dimensions).

Cohérence de bornes

La cohérence de bornes est un compromis, dans le sens où on ne va pas supprimer toutes les valeurs incohérentes, mais on va tout de même garantir qu'on ne perd pas de solutions. En effet, on ne travaille plus sur des ensembles mais sur des intervalles, ce qui signifie qu'il n'y a plus qu'à stocker la borne inférieure et la borne supérieure des domaines au lieu d'énumérer chaque valeur possible. Cela signifie aussi qu'on ne peut travailler qu'avec des valeurs numériques et qu'on ne peut pas supprimer des valeurs incohérentes au milieu des domaines, ce sera fait lors de la phase de test. Pour trouver les bornes des intervalles, on utilise des propriétés sur les opérations arithmétiques.

Cohérence spécialisée

Pour certaines contraintes que l'on appelle contraintes globales, les méthodes ci-dessus ne suffisent pas nécessairement et il est plus intéressant de créer des méthodes de cohérence spécialisées pour traiter ces cas. Par exemple si on veut que trois variables $A, B, C \in X$ soient différentes deux à deux mais que ces trois variables ont pour domaine $D = \{1, 2\}$, on est dans un cas où il n'existe pas de solution au problème mais ça ne serait pas détectable par cohérence de bornes, alors qu'une méthode spécialisée serait capable d'être plus précise.

4. Programmation par contraintes en Prolog

Pour résoudre des problèmes à contraintes en Prolog, on utilisera la bibliothèque `ic`. Elle introduit des variables particulières qui, en plus des propriétés classiques des variables de Prolog, ont un domaine ainsi qu'une liste des contraintes dans lesquelles elles interviennent.

La bibliothèque gère toute seule les optimisations sur les contraintes (ordre d'appel, simplifications, ...) ainsi que les listes de contraintes déjà utilisées ou en attente. Il nous reste donc à choisir l'ordre de traitement des variables et l'ordre d'essai des valeurs d'un domaine (voir prédicat `search/6`).

Elle permet également de créer des contraintes dites passives qui ne sont doivent être utilisées que pour tester des valeurs (pas pour de la génération). En Prolog quand une le prédicat `X > 2` est exécuté, la valeur de `X` est immédiatement évaluée puis un résultat est retourné (`Yes` ou `No`). En revanche, si la variable n'a pas encore de valeur, on aura une erreur. Avec la librairie `suspend`, on peut attendre que la valeur de `X` (une variable `ic`) ait été fixée avant d'effectuer le test, ce qui s'écrit `suspend:(X>2)`. Dans les logs, on voit qu'il y a un `DELAY` au moment où l'instruction est exécutée, puis quand la valeur de `X` est fixée il y a un appel à `wake` suivi d'un `RESUME`, puis de `EXIT` ou `FAIL` suivant que le résultat du test soit positif ou négatif.

Il est aussi possible de créer des variables avec des domaines qui ne sont pas des ensembles de nombres mais de symboles (bibliothèque `ic_symbolic`).

Enfin, on peut trouver plus de 400 contraintes globales (bibliothèque `ic_global`) avec des méthodes de cohérence spécialisées (http://sofdem.github.io/gccat). On notera par exemple :

- `alldifferent(+List)` contraint toutes les variables de `List` à être différentes deux à deux
- `element(?Index, ++List, ?Value)` contraint `Value` à être égale à l'élément à la position `Index` de `List`
- `atmost(+N, ?List, +V)` contraint les éléments de `List`, de telle sorte à ce qu'il y ait au maximum `N` occurences de la valeur `V`
- `maxlist(+List, ?Max)` contraint le maximum de `List` à être égal à `Max`
- `minlist(+List, ?Min)` contraint le minimum de `List` à être égal à `Min`
- `sumlist(+List, ?Sum)` contraint la somme des éléments de `List` à être égale à `Sum`

La structure de programme que nous utiliserons sera la suivante :

```
:- lib(ic).

solve_problem(Problem) :-
      create_data(Data),
      create_vars(Problem)
      constrain(Data, Problem),
      get_vars(Problem, Vars),
      labeling(Vars).

% définition de create_data

% définition de fonctions utilitaires

% définition de create_vars

% définition de constrain

% définition de get_vars
```

II. Schémas de programmation

1. Minimisation du résultat

```
:- lib(branch_and_bound).

solve_min(Arg, Res) :-
        minimize(solve(Arg, Res), Res).

% Structure plus complexe (où solve ne fait pas le labeling)
solve_min(Vars, Sum) :-
        minimize(
                (
                        solve(Vars, Sum),
                        labeling(Vars)
                ),
                Sum
        ).
```

Note : pour maximiser, on contraint une variable à être l'opposé du résultat et on minimise cette variable

2. Domaines symboliques

```
:- lib(ic_symbolic).

?- local domain(greeting(bonjour, hello, hola, hej)).

% appartenance au domaine -> Greet &:: greeting
% valeur exacte -> Greet &= bonjour
% alldifferent -> ic_symbolic:alldifferent([Greet1, Greet2, Greet3])

% équivalent de labeling pour les domaines symboliques
labeling_symbolic(Vars) :-
        (foreach(V, Vars) do
                ic_symbolic:indomain(V)
        ).
```

3. Prédicat `search/6`

```
search(+L, ++Arg, ++Select, +Choice, ++Method, +Option)

% dans notre cas d'utilisation, certains paramètres sont toujours les mêmes
search(+L, 0, ++Select, +Choice, complete, [])

% Note : labeling(L) <=> search(L, 0, input_order, indomain, complete, [])
```

Select	Choice
input_order	Indomain
first_fail	indomain_min
anti_first_fail	indomain_max
smallest	indomain_reverse_min
largest	indomain_reverse_max
occurrence	indomain_middle
most_constrained	indomain_median
max_regret	indomain_split
	indomain_reverse_split
	indomain_random
	indomain_interval

Pour plus d'informations : https://eclipseclp.org/doc/bips/lib/ic/search-6.html

4. Itérateurs

```
fromto(First, In, Out, Last)
foreach(X, List)
foreacharg(X, Struct, Idx)
foreachelem(X, Array, Idx)
foreachindex(Idx, Array)
for(I, MinExpr, MaxExpr, Increment)
multifor(List, MinList, MaxList, IncrementList)
```

Agrégation de données

```
% construction de liste, ordre inverse
(for(I, 1, Size), fromto([], In, Out, Res) do
        Out = [I|In]
).

% construction de liste, ordre conservé
(for(I, 1, Size), fromto(Res, In, Out, []) do
        In = [I|Out]
).

% somme de nombres
nb_val(Tab, Val, Res) :-
        (foreachelem(E, Tab), fromto(0, In, Out, Res), param(Val) do
                % ici, le booléen E #= Val est utilisé comme valeur numérique
                Out #= In + (E #= Val)
        ).
```

Imbrication de `foreach` ou `foreachelem`

```
(foreach(X, List), param(List) do
        X = x(A, B, C)
        % Maybe do something with X
        (foreach(X1, List), param(A, B, C) do
                X1 = x(A1, B1, C1)
                % ...
        )
).
```

Itération sur plusieurs indices

```
flatmap(T) :-
        dim(T, [MaxI, MaxJ]),
        (multifor([J, I], [MaxJ, MaxI], [1, 1], [-1, -1]), fromto([], In, Out,
L), param(T) do
                E is T[I,J],
                Out = [E|In]
        ).
```

5. Transformer une liste en tableau (ou inversement) `array_list(Array, List)`

6. Action en fonction du succès ou non d'un prédicat (ressemble à des ternaires)

```
Labeling(Vars) -> writeln(ok); writeln(impossible)
```