TP de Complexité

V.Berry & J. Fortin – Polytech Montpellier, dépt I.G.

16 décembre 2014

Résumé

Objectifs d'apprentissage :
- savoir modéliser un problème sous la forme d'un CSP (*Constraint Satisfaction Problem*)
- Comprendre comment on peut détecter des inconsistances dans la modélisation CSP d'un problème (*arc-consistance*)
- Comprendre le processus de résolution d'un CSP (algorithme de *backtrack*).
- Savoir utiliser la librairie java Choco (si possible dans un IDE)

1 Modélisation : un problème de déplacement familial

Pour mobiliser les connaissances vues en cours, rien de tel qu'un premier exemple de modélisation. Imaginons le scénario suivant :

1. une famille américaine de quatre personnes (mère, père, fille, fils) doit organiser ses déplacements pour se rendre le matin au travail et à l'école.
2. Chaque membre de la famille peut circuler en vélo ou monter dans la voiture
3. La famille ne possède que deux vélos
4. Le fils a aussi un bâton sauteur (pogo stick [1]) qu'il peut emprunter pour se rendre à l'école.
5. La voiture n'a que 3 places
6. Si le fils ou la fille prenne la voiture il faut un parent pour les conduire
7. Le fils et la fille doivent prendre le même mode de transport.

Nous allons d'abord proposer une modélisation de ce problème :

Question 1
- *Quelles variables pensez-vous utiliser pour modéliser ce problème en CSP ?*
- *Quel domaine pour chaque variable ?*

Ecrivez vos réponses avant de passer à la suite du TP.

Question 2 *Quelles phrases numérotées dans l'énoncé correspondent à des contraintes ?*

Question 3 *Ecrivez de deux ces contraintes* en extension *sur la base des variables que vous avez définies précédemment.*

1. cherchez une image sur internet si vous ne voyez pas ce que peut être ce truc typiquement américain

2 Un outil graphique de résolution de CSP

Pour vous familiariser avec la résolution de CSP, nous avons recours à une petite application graphique qui peut être lancée en ligne à partir du site AI space (menu *Main Tool* puis *Consistency Based CSP Solver* puis lancez l'application ("click here" en haut de page))[2].

Cette petite application Java va permettre de comprendre la détection d'instances faciles et la résolution progressive de CSP. Lorsque l'application est lancée, un fenêtre graphique s'ouvre, permettant de commencer la description d'un problème. Cette application permet de créer et résoudre des problèmes à variables entières, booléennes ou sous forme de chaînes de caractères.

2.1 Modélisation d'un problème simple

Pour commencer à utiliser cette application explorez l'onglet *Create* pour modéliser le problème suivant : on a trois variables entières x1,x2 et x3, qui prennent leurs valeurs dans $[1, 2, 3]$ sous les contraintes que $x1 \neq x2$ et $x1 + x2 = x3$.

Question 4 *Avec l'interface, codez ces deux variables, et les deux contraintes correspondantes. Puis sauvegardez le CSP obtenu sous le nom* `x1x2x3`.

2.2 Résolution du CSP

Passons ensuite à l'onglet *Solve*. Celui-ci va permettre d'enchaîner deux opérations essentielles pour résoudre le CSP :

- *arc-consistancy* : la réduction justifiée des valeurs possibles pour certaines variables, étant donné les contraintes.
- *domain split* : la réduction arbitraire mais temporaire des valeurs d'une variable afin d'avancer dans la recherche d'une solution.

2.2.1 Consistance d'arc

La **consistance d'arc** consiste à passer en revue chaque contrainte tour à tour, et pour chacune à regarder si elle ne permet pas d'exclure certaines valeurs que peuvent prendre les variables. Si au bout de ce processus une variable n'a plus de valeur possible, alors c'est que le CSP n'admet pas de solution (impossible de donner des valeurs aux variables en respectant les contraintes indiquées). Ce processus d'arc-consistance se fait par un algorithme simple (demandez à votre enseignant) en temps polynomial. En quelque sorte c'est l'équivalent pour le pbm du chemin hamiltonien de la vérification en temps polynomial que l'instance donnée possède (ou pas) une propriété qui fait qu'on peut répondre très vite "non" au problème (c-a-d cette propriété implique que l'instance ne possède pas de chemin hamiltonien).

Dans l'onglet *Solve* de l'application, nous avons 4 zones, de bas en haut : une zone de texte indiquant en cours de résolution l'instanciation progressive ou finale des variables, le dessin du CSP, une zone de commentaires / actions proposées, un ensemble de boutons pour conduire la résolution.

2. Si le *java web start* n'est pas activé sur votre navigateur/système, vous pouvez obtenir une copie de la librairie `constraint.jar` sur cette page web (enregistrez le fichier et lancez-le avec la commande `java -jar constraint.jar`)

Positionnez *arc-consistency speed* au plus lent (dans le menu *CSP options*). Choisissez *auto-solve* et regardez le processus de résolution se produire **en arrêtant le processus** (bouton stop) dès qu'une valeur a été supprimée pour une variable.

Question 5 *Quelle valeur a été supprimée, pour quelle variable, et pour quelle raison ? Si vous n'avez pas réussi à arrêter le processus à temps, recommencez (bouton* reset*) ou bien faite un pas en arrière (*step back*) et doucement en avant (*fine step *ou* step*). Cliquer sur une contrainte dans le graphique permet de vérifier juste la consistance de cette contrainte.*

2.2.2 Instanciation progressive des variables et backtrack

Une fois l'arc consistance passée, il reste peut-être encore trop de valeurs possibles aux variables pour que le solveur trouve immédiatement une solution au problème. Pour avancer dans la résolution, il procède alors au *salit* du domaine d'une variable : il considère temporairement qu'un sous-ensemble des valeurs restantes pour cette variable et va continuer la résolution. Si cela aboutit à trouver une solution, alors tout va bien. En revanche, si aucune solution n'est possible avec ce sous-ensemble, alors il viendra examiner le sous-ensemble restant.

Lancez la résolution par le bouton *auto-solve*, et regardez ce qui s'inscrit dans la zone de texte du bas. Chaque ligne signifie que l'on a fait un choix (ou atteint une solution). A chaque choix dépendant des précédents, une tabulation est ajoutée, à chaque sous-ensemble alternatif à un choix initial, on revient au même niveau de tabulation (*backtrack*).

Vérifiez qu'une solution est bien atteinte.

Question 6 *Si la résolution ne va pas trop vite, vous devez remarquer qu'après avoir fait une étape de* split *d'une variable, l'algorithme de résolution recommence une étape d'arc-consistance de contraintes.*

- *Pourquoi est-ce justifié ?*
- *Est-ce justifié pour toutes les contraintes ?*

Dans ce petit problème trop simple, il est probable qu'aucun retour arrière n'est nécessaire. Mais c'est loin d'être le cas général. Pour observer de tels retours arrière, demandez à l'application de chargez depuis l'URL suivante le problème auquel vous avez réfléchis en début de TP : http ://www.aispace.org/exercises/FamCommuteCSP.xml.

Question 7 *Regardez bien la modélisation qui est proposée*

- *En quels points ne correspond-elle pas à votre modélisation initiale ?*
- *Est-ce justifié ?*

Question 8 *Etant donné ce que vous connaissez de ce problème, vous attendez-vous à ce que l'arc-consistance élimine des valeurs (lesquelles) ?*

Lancez maintenant la résolution automatique et détectez un point de retour arrière (*backtrack*).

Question 9 *Quelle restriction de domaine a fait que l'on n'a pu trouver de solution au problème ? Expliquer pourquoi cette restriction conduit à ne pas avoir de solution au problème.*

Malgré les étapes de retour-arrière, vous devez voir la résolution aboutir et pouvoir vérifier que la solution proposée est bien correcte (elle respecte chaque contrainte donnée).

3 CHOCO

Cette partie de TP est consacrée à l'utilisation de *CHOCO* (Chic, un Outil Contraintes avec des Objets), une librairie Java permettant la modélisation et la résolution de problèmes de satisfaction de contraintes de façon bien plus avancé que le petit outil que nous venons de voir. Nous utiliserons la version 2.1.0 de la librairie CHOCO, que vous pouvez télécharger sur cette page web. Une version plus récente de *CHOCO*, sa documentation ainsi qu'un grand nombre de tutoriels peuvent être trouver sur le site officiel. Mais restons sur la version 2 pour l'instant.

3.1 Un exemple arithmétique

Nous allons modéliser et résoudre ensemble pas à pas le problème suivant : Trouver une affectation des variables x_1, x_2 et x_3 telles que $x_1, x_2, x_3 \in [0..5]$ et $x_2 > x_3$, $x_2 \neq x_3$ et $x_1 = x_2 + x_3$.

Téléchargez et tester cet exemple toujours à partir de la même page web : ClasseTest-Choco1.java

Vous pourrez dans un premier temps compiler à la main ce fichier dans le terminal de commandes en utilisant la bibliothèque Choco (option `-jar`) et demander son exécution aussi depuis le terminal. Attention dans les deux cas à bien positionner la variable `CLASSPATH` dans le shell. Une autre solution est de créer un projet Java dans un IDE (Eclipse est disponible dans vos salles de TP), en ajoutant la librairie *CHOCO* comme une *libraire jar externe*.

3.1.1 Modèle

Lorsque l'on utilise *CHOCO*, la première chose à faire est de définir le modèle du problème que l'on compte résoudre. L'instruction crée un nouveau modèle m.

```
Model m = new CPModel();
```

3.1.2 Variables

Il faut ensuite déclarer les différentes variables utiles à la définition du problème, ainsi que leurs domaines de définition. Attention, chaque variable du problème d'optimisation à un type propre à *CHOCO*, c'est à dire par exemple qu'une variable qui peut être instanciée par valeurs entière ne doit pas être déclarée de type `int`, mais de type `IntegerVariable`. Les trois types de variables possibles sont les variables entières (`IntegerVariable`), réelles (`RealVariable`) et ensemblistes (`SetVariable`). Nous allons nous contenter d'utiliser les variables entières dans ce TP.

Les lignes de code suivantes déclarent nos 3 variables x_1, x_2 et x_3 à valeur dans $[0..5]$ auxquelles on donne les noms `var1`, `var2` et `var3` :

```
IntegerVariable x1 = makeIntVar("var1", 0, 5);
IntegerVariable x2 = makeIntVar("var2", 0, 5);
IntegerVariable x3 = makeIntVar("var3", 0, 5);
```

Pour définir des domaines de valeur entières non contiguës, se rapporter à la JavaDoc de *CHOCO*.

3.1.3 Contraintes

Une fois les variables du problèmes définies, on peut modéliser les contraintes du problème. Les principales contraintes sur les entiers sont `eq`, `geq`, `gt`, `leq`, et `neq`, leur signification est donnée dans le tableau suivant :

Contrainte	en anglais	en mathématique
`eq(x1,x2)`	equal	$x_1 = x_2$
`geq(x1,x2)`	greater or equal	$x_1 \geq x_2$
`gt(x1,x2)`	greater than	$x_1 > x_2$
`leq(x1,x2)`	less or equal	$x_1 \leq x_2$
`neq(x1,x2)`	not equal	$x_1 \neq x_2$

Pour notre problème on peut donc déclarer les contraintes suivantes :
```
Constraint C1 = gt(x2, x3);
Constraint C2 = neq(x2, x3);
```

Pour la dernière contrainte ($x_1 = x_2 + x_3$), nous avons besoin de créer une expression qui modélise la somme des variables x_2 et x_3 le signe + n'étant pas utilisable sur les données de type IntegerVariable. Cela se fait de la manière suivante :
```
IntegerExpressionVariable sumx2x3 = plus(x2,x3);
```

On peut ensuite déclarer la contrainte suivante :
```
Constraint C3 = eq(x1, sumx2x3);
```

Pour d'autre opérations, se rapporter à la JavaDoc de *CHOCO*. La dernière opération à faire avant de passer à la résolution du problème est d'ajouter les contraintes nouvellement crées dans notre modèle :
```
m.addConstraint(C1);
m.addConstraint(C2);
m.addConstraint(C3);
```

3.1.4 Résolution

Afin de résoudre notre problème il faut maintenant créer un solveur, lui indiquer quel est le modèle du problème à résoudre, éventuellement fixer les stratégies de choix des variables à instancier, et lancer la résolution. Ceci peut se faire par les instructions suivantes :
```
Solver s = new CPSolver();
s.read(m);
s.solve();
```

On peut alors vérifier la présence d'une solution et si elle existe lire les valeurs des variables dans la solution atteinte :
```
if(s.isFeasible()){
   System.out.println("var1 =" + s.getVar(x1) .getVal());
   System.out.println("var2 =" + s.getVar(x2) .getVal());
   System.out.println("var3 =" + s.getVar(x3) .getVal());
}
else System.out.println("No solution");
```

On peut aussi demander toutes les solutions d'un problème, demander à minimiser une quantité données etc...

3.2 Camping – inspiré d'un problème de Projet Industriel IG4 2009-2010

Un petit camping à la ferme dispose de 3 emplacements pour loger des campeurs sur une semaine de vacances (on considère juste une période de 7 jours). Les demandes des campeurs consistent chacune en un jour d'arrivée et un jour de départ (chacun compris entre 1 et 7 et avec départ $\geq$ arrivée).

Etant donné une liste de demandes, le camping doit essayer d'affecter un emplacement (et un seul) à chaque demande (il n'est pas envisageable de demander à des campeurs de changer d'emplacement au milieu de leur séjour).

3.2.1 Un algorithme glouton est insuffisant pour ce problème

Supposons l'algorithme glouton suivant : on ordonne les emplacements suivant un numéro ; on prend les demandes dans l'ordre de leur arrivée et affecter ; quand une demande est considérée on essaye de la positionner dans le premier emplacement, si elle ne tient pas on essaye dans le second, etc. Cet algorithme ne remet jamais en question les choix qu'il effectue.

Question 10 *Pour le cas de 2 emplacements seulement, trouvez une combinaison de demandes arrivant dans un certain ordre tel que l'algorithme glouton ne trouve pas de solution au problème, alors qu'une solution existe pourtant.*

3.2.2 Utilisation des CSP et de *CHOCO* pour résoudre le problème

Proposez une modélisation de ce problème où on considère que l'on dispose de 3 emplacements et de 8 demandes (qui peuvent être générées aléatoirement ou qui sont choisies pour la phase de test)

Question 11

- *Quelles variables ?*
- *Quel domaine pour chacune ?*
- *Quelles contraintes ?*

Vérifiez ensuite la cohérence de votre solution avec votre enseignant.

Implémentez cette solution dans *CHOCO* en essayant un certain nombre de combinaisons de demandes : une combinaison qui peut être casée et une combinaison qui ne peut pas être casée dans les 3 emplacements.

3.3 Carré magique

Un carré magique est un carré de nombre entiers (tous distincts) dont la somme des nombre de chaque ligne, de chaque colonne et de chaque diagonal est identique. Écrire à l'aide de Choco un programme qui permet de trouver un carré magique de taille quelconque.

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

FIGURE 1 – Une grille de Sudoku

3.4 Sudoku

Grâce à *CHOCO* trouvez la solution du Sudoku de la Figure 1.

4 Conclusion

La programmation par contraintes est un formidable paradigme déclaratif permettant la modélisation et la résolution de nombreux problèmes de complexités diverses. Attention, un problème peut se modéliser en général de plusieurs façons, et le choix d'une modélisation plutôt qu'une autre peut avoir un impact important en temps de résolution par le solveur.

Gardez donc à l'esprit que ces outils existent, que vous pouvez les utilisez pour résoudre bon nombre de problèmes d'optimisation que vous rencontrerez, y compris des problèmes NP-difficiles. Si toutefois le solveur ne réussit pas à résoudre une instance que vous considérez (c-a-d ne peut répondre ni oui ni non au bout d'un long temps calcul), cela peut venir d'une modélisation non pertinente ou de l'utilisation de contraintes non-adaptées au problème. Dans ce cas il est raisonnable de faire appel à l'aide de spécialistes de la discipline afin de vérifier si votre modélisation est pertinente ou bien si vous avez la malchance d'être tombé sur une instance réellement difficile du problème à résoudre.