

ASP : Answer Set Programming

Carito Guziolowski

Département MATH-INFO, LS2N, École Centrale de Nantes

3 mars 2022

TP1 (partie II) ASP

Fichiers à rendre sur Hippocampus :

- aggr.lp
- queens_mod.pl
- rapport.pdf

1 Syntaxe : agrégats ou *aggregates*

Une instruction de type agrégat a la forme suivante :

$$l \prec_1 \text{op} \{t_1 : L_1; \dots; t_n : L_n\} \prec_1 u$$

Où t_i et L_i sont des termes et des prédicats (exprimant des conditions), respectivement. op est le nom d'une fonction qui sera appliquée pour les tuples des termes t_i qui restent après l'évaluation des conditions exprimés par L_i . Finalement le résultat d'appliquer op est comparé avec l'aide des opérateurs de comparaison \prec_1 et \prec_2 aux termes l et u (entiers) respectivement. Les opérateurs \prec_1 et \prec_2 peuvent être omis et l'opérateur de comparaison par défaut sera alors \leq .

Actuellement, gringo et clingo acceptent les opérateurs (op) suivants : $\#count$ (le nombre d'éléments), $\#sum$ (la somme des poids positifs), $\#min$ (le poids minimal), et $\#max$ (le poids maximale). Le poids est exprimé par le premier élément de la tuple des termes dénoté par t_i . Notez que, à différence des autres types d'opérateurs, $\#count$ n'a pas besoin des poids. Voici des exemples des agrégats avec l'opérateur somme :

Listing 1 – Agrégats

```
1 20 <= #sum { 4 : course(db); 6 : course(ai); 8 : course(project); 3 : course(xml) }.  
2 #sum { 3 : bananas; 25 : cigars; 10 : broom } <= 30.
```

Dans les deux cas d'agrégats, les atomes `course(ai)`, `broom`, parmi d'autres, sont associés à des poids. Si on fait l'hypothèse que `course(ai)`, `course(db)`, `bananas`, et `broom` sont vraies, alors les termes déduits seront $\{4;6\}$ et $\{3;10\}$ respectivement. Après avoir appliqué la fonction `#sum` à chaque ensemble, nous obtenons $20 \leq 10$ et $13 \leq 30$. Alors, dans ce cas, seulement le deuxième agrégat (Ligne 2 du Listing 1) est vérifié.

1.1 Duplication des termes

Comme indiqué par les accolades, les éléments dans les agrégats sont traités comme membres d'un ensemble. Par conséquent, **les doublons ne sont pas comptabilisés deux fois**. Par exemple, les agrégats suivants expriment la même chose :

- 1 `#count { 42 : a; t : not b } = 2.`
- 2 `#count { 42 : a; 42 : a; t : not b; t : not b } = 2.`

C'est à dire, si `a` est vrai et `b` est faux, alors l'ensemble se réduit à $\{42; t\}$ dans le deux cas. De même, les éléments d'autres agrégats sont compris comme des ensembles. Prenez en compte les agrégats de type somme suivants :

- 1 `#sum { 3 : cost(1,2,3); 3 : cost(2,3,3) } = 3.`
- 2 `#sum { 3,1,2 : cost(1,2,3); 3,2,3 : cost(2,3,3) } = 6.`


Imaginons que le prédicat `cost(1,2,3)` représente le coût (3) associé à l'arête entre les noeuds (1,2). Si à la fois `cost(1,2,3)` et `cost(2,3,3)` sont vraies, alors la première somme vaut 3, tandis que la seconde vaut 6. Notez que tous les tuples de termes (le tuple singulier 3 ainsi que les tuples ternaires 3,1,2 et 3,2,3) ont le même poids, à savoir 3. Cependant, la propriété des ensembles fait que le premier agrégat compte des prédicats de type `cost` avec le même poids seulement une fois pour chaque arête, tandis que le second agrégat exprime que chaque arête aura un coût associé de 3. Pour voir cela, observez qu'après évaluer les conditions de chaque agrégat, le premier se réduit à `#sum {3}`, tandis que le second donne `#sum {3,1,2; 3,2,3}`. En d'autres termes, associer chaque coût avec son arête respectif applique une propriété multi-ensembles; de cette façon, le même coût peut être pris en compte plusieurs fois.

1.2 Question 1 :

Considérez une situation où un étudiant en informatique voudrait s'inscrire à un nombre de cours pour le prochain semestre. Dans l'emploi du temps de l'étudiant, 8 cours sont éligibles et correspondent aux faits suivants :

- 1 `cours(1,1,5) . cours(1,2,5) .`
- 2 `cours(2,1,4) . cours(2,2,4) .`
- 3 `cours(3,1,6) .` `cours(3,3,6) .`
- 4 `cours(4,1,3) .` `cours(4,3,3) . cours(4,4,3) .`
- 5 `cours(5,1,4) .` `cours(5,4,4) .`
- 6 `cours(6,2,2) . cours(6,3,2) .`
- 7 `cours(7,2,4) . cours(7,3,4) . cours(7,4,4) .`
- 8 `cours(8,3,5) . cours(8,4,5) .`

Le prédicat `cours/3` (avec arité 3) a comme premier argument un numéro identifiant le cours et comme dernier argument le nombre d'heures par semaine du cours. Le deuxième argument correspond au domaine : 1 pour *informatique théorique*, 2 pour *informatique pratique*, 3 pour *informatique technique* et 4 pour *informatique appliquée*. Par exemple, le prédicat `cours(1,2,5)` exprime que le cours 1 a 5 heures par semaine et qu'il appartient au domaine 2 (informatique pratique). Un cours peut appartenir à plusieurs domaines. Écrire un programme logique en ASP aggr.lp qui permet d'exprimer certains choix et/ou contraintes de l'étudiant. Voici la liste de contraintes à vérifier :

1. L'étudiant voudrait s'inscrire à au minimum 3 cours et au maximum 6, utilisez un prédicat `inscrire(C)` pour exprimer ce choix.
2. Le nombre de domaines éligibles, associées à tous les cours inscrits, doit être supérieur (stricte) à 10. Par exemple une sélection des cours **6, 7 et 8**, comprendra en total 7 ($2 + 3 + 2$) domaines; cette sélection ne satisfait pas la contrainte d'être supérieur à 10 en nombre.  **Note :** Il ne s'agit pas de faire la somme des identifiants de domaines.
3. **L'étudiant ne peut ne pas s'inscrire qu'au plus 1 cours dans le domaine 2.** C'est-à-dire, parmi les n cours éligibles du domaine 2, il peut s'inscrire à tous ou à $n - 1$. **Hint :** utiliser le `not` dans cette règle.
4. Le nombre total des cours qui appartiennent aux domaines *informatique technique* et *informatique appliquée* doit être inférieur stricte à 6.
5. Rajoutez dans votre PL des règles pour gérer le nombre d'heures. Pour cela créez un prédicat `heures(C, T)` qui permettra d'obtenir les cours C associés aux heures T .
6. La somme d'heures doit être comprise entre 18 et 20.
7. Chaque cours inscrit doit représenter un investissement supérieur (stricte) à 2 heures par semaine et inférieur (stricte) à 6 heures par semaine.
8. **Bonus** Modifier votre implémentation pour représenter le nombre d'heures maximales par une variable M , et exprimer la contrainte 6 par rapport à $M - 2$ et M .

Rapport : Pour chaque règle du programme, justifier le choix fait pour exprimer cette contrainte en ASP (e.g. opérateur `#sum` ou `#count` ?). Pour les questions 1, 2 et 6, expliquer **à quoi correspond la sortie de l'exécution** `gringo -t aggr.lp`. Montrez et justifiez la sortie de ce programme (toutes contraintes traitées) avec la commande `gringo aggr.lp | clasp -n 0`. Vous pouvez comparer vos réponses à la solution de ce problème disponible sur le serveur pédagogique.

2 Problème de n-Dames

Le problème de n-Dames consiste à placer n dames dans un damier de taille $n \times n$, de façon qu'aucune dame soit une menace pour autre dame. Nous avons vu en cours une implémentation de ce problème.

2.1 Question 2

Reprendre la solution du problème de n-Dames vue en cours :

```

1 row(1..n).
2 col(1..n).
3 {queen(I,J) : row(I) : col(J)}.
4 :- { queen(I,J) } != n.
5 :- queen(I,J), queen(I,JJ), J != JJ.
6 :- queen(I,J), queen(II,J), I != II.
7 :- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I-J == II-JJ.
8 :- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I+J == II+JJ.
```

Voici la liste de questions à implémenter et à répondre dans votre rapport :

1. **Syntaxe** : Expliquer en terme d'agrégats la ligne 3 du LP ci-dessus.
2. Écrire dans le fichier `queens.lp` le PL correspondant. Ensuite générer les instantiations (programme *grounded*) possibles de ce programme pour un damier de taille 5×5 . Commenter les lignes de ce programme pour vous aider à mieux comprendre le processus d'instanciation.
 - **gringo** : L'instance (après la commande `gringo`) de la ligne 3 du PL contient elle toutes les combinaisons possibles des `queen(I, J)` dans un damier de taille 5×5 ?
 - **gringo** : Quelle est la différence entre le PL ci-dessus et le PL de sortie après la commande `gringo` ?
 - **clasp** : Tester la sortie du programme pour différentes tailles du damier (e.g. $n = 5, 7, 9, 11, 13$); trouver le nombre de solutions et le temps du calcul correspondant.
3. Écrire un PL `queens_mod.pl` qui modifie `queens.pl` pour trouver toutes les solutions pour un damier de taille 8 qui : (a) a une dame dans la position $(1, 1)$, et (b) n'a pas des dames dans le carré de 4×4 au milieu du damier.
4. Le PL suivant permet d'optimiser le code de `queens.pl` pour accélérer le temps du calcul.

```
1 { queen (I ,1.. n) } = 1 :- I = 1..n.
2 { queen (1..n,J) } = 1 :- J = 1..n.
3 :- { queen (D-J,J) } > 1, D = 2..2* n.
4 :- { queen (D+J,J) } > 1, D = 1-n..n -1.
```

 - Expliquer les lignes 1 à 4 du code ci-dessus.
 - Écrivez le code dans un PL `queens_opt.lp` et exécuter la commande `gringo`. Quelle est la différence par rapport au PL instantié de `queens.lp` ?
 - Trouver les answer sets de `queens_opt.lp` pour les mêmes tailles du damier choisies dans la partie 2. Faire la comparaison par rapport au temps du calcul de `queens.lp`.