Utiliser la programmation par ensembles réponses pour de petits problèmes

Yves Moinard
INRIA-Bretagne-Atlantique
campus de Beaulieu, 35042 RENNES Cedex
moinard@irisa.fr

Résumé

En traduisant en programmation par ensembles réponses deux devinettes classiques, nous illustrons la puissance et certaines limites des systèmes actuels. Il s'agit de les traduire d'une façon pas trop ad-hoc, tirant profit de l'aspect déclaratif de ce type de programmation. On n'est pas loin de cet objectif affiché, mais il reste des progrès à faire pour les systèmes existant. Nous suggérons quelques pistes : l'intégration de réelles listes, une meilleure intégration entre instantiateur et solveur, permettant au moins de détecter les cas où le premier suffit, une déclaration de prédicats satisfaisant une loi d'inertie*, une déclaration de prédicats "intermédiaires".*

Mots Clef

Programmation logique, Ensembles réponses, Planification.

Abstract

By taking as examples two classic riddles, we illustrate the power and limitations of current systems of answer set programming. We aim at reduce the work of the programmer, taking advantage of the declarative aspect of this type of programming. Current systems are not far from this claimed objective, but there is still room for improvements. We suggest some ideas : allowing real lists, better integration between instantiator and solver, allowing at least to detect when the first suffice, a declaration for predicates satisfying law of inertia, a declaration of " auxiliary" predicates.

Keywords

Answer sets, Logic programming, Planning.

1 Introduction

On examine la facilité avec laquelle on peut traduire de petits problèmes en programmation par ensembles réponses (abrégé par l'acronyme anglais ASP, "answer set programming") [1]. Depuis gringo3 (juillet 2010) la lignée clingo (gringo + clasp)[1] [4], est la plus efficace en général. Auparavant, il était souvent plus commode de traduire ces problèmes grâce à DLV [6][2] et ses extensions comme DLV-Complex[3] [2], avec des structures de données riches (suites, ensemble). L'usage de clingo exigeait que chaque variable du corps d'une règle soit introduite par un prédicat dont le domaine est donné au départ. Cette lourde contrainte a disparu depuis gringo3 (juillet 2010). L'aspect ouvert de clingo, contre un système DLV de plus en plus "fermé", ainsi que de meilleures performances en général, militent en faveur de clingo. D'autres sytèmes[4], comme Asperix[5] [5] ou Cmodels[6] demeurent en version préliminaire et/ou sont d'une utilisation plus complexe.

La programmation par ensemble réponse est proche de l'idéal de la programmation déclarative : *le problème est le programme*. La *solution* est un ensemble de modèles particuliers appelés *ensembles réponses* (que l'on abrégera encore par l'acronyme anglais *AS*). En réalité bien sûr il faut se préoccuper un peu de la façon dont le système va calculer les réponses. La programmation par ensembles réponses est bien adaptée à des problèmes de parcours dans des graphes et permet d'écrire des programmes élégants, y compris pour les règles avec exception. Ce formalisme proche de la logique classique traite de façon très naturelle de petits problèmes logiques comme le fameux problème de *l'eau et du zèbre*. Hakan Kjellerstrand[7] fournit l'énoncé, et surtout deux transcriptions en ASP élégantes, dont une seule est efficace. Cela illustre le fait (évident, et reconnu par les créateurs ou utilisateurs de systèmes ASP) qu'il est illusoire de s'affranchir de toute considération d'efficacité. Il reste que les critères permettant de juger du caractère "naturel" d'un programme ASP qui résout un problème posé sont flous. La compétition ASP tend naturellement à privilégier les systèmes qui permettent, pour de fins connaisseurs, de résoudre un problème rapidement et avec le moins de place mémoire. Pour tout langage informatique, ce seul critère est insuffisant, mais c'est encore plus vrai pour un langage déclaratif. La seconde compétition ASP [3] évoquait l'importance cet aspect, délicat à mesurer objectivement :

> The Potassco team of the University of Potsdam is the clear winner *[vrai aussi en 2011*[8] *].*
> [...]the goal of declarative problem solving is to

[1] `http ://sourceforge.net/projects/potassco/`
[2] `http ://www.dbai.tuwien.ac.at/proj/dlv/`
[3] `https ://www.mat.unical.it/dlv-complex`
[4] Liste des principaux systèmes : `www.cs.uni-potsdam.de/~torsten/asp/`
[5] `www.info.univ-angers.fr/pub/claire/asperix/`
[6] `www.cs.utexas.edu/users/tag/cmodels.html.`
[7] `www.hakank.org/answer_set_programming/zebra.lp`
[8] https ://www.mat.unical.it/aspcomp2011/Model%26SolveTrackFinalResults

minimize the effort of programmers,[...]

Voici deux petits exemples.

2 Un exemple de rêve : le pont miné

Nous avons évoqué l'exemple de l'eau et du zèbre dans l'introduction, qui admet une traduction très naturelle et efficace en ASP, au prix de quelques précautions, cf Hakan Kjellerstrand[7]. Cet exemple proche de la logique propositionnelle est convaincant, dans ce cadre particulier. Le pont miné est moins "logique pure", même s'il est clair que ce n'est pas un problème difficile :

> Un groupe de quatre soldats, disposant d'une seule lampe, doit franchir de nuit un pont qui sautera une heure après que le premier d'entre eux aura mis le pied sur le pont. Les soldats, plus ou moins blessés, ont des durées de franchissement du pont en minutes de 25, 20, 10 et 5. Le pont ne supporte que deux soldats à la fois et la lampe est indispensable pour traverser. Comment font-ils ?

Un programme qui suit littéralement l'énoncé suffit. Un programme ASP est une suite de règles "`tête :- corps.`" : *si chaque élément du corps est satisfait, la tête l'est*. Pour les données (temps de traversée, maximum,...), on utilise des *faits* (règles sans corps : `corps` $\equiv \top$) :

```
soldat(0..3). % (tout entier entre 0 et 3)
temps(0,25). temps(1,20). temps(2,10).
temps(3,5). % le soldat "0" met 25 mn,...
lieu(depart). lieu(arriv).
#const tmax= 60.
```

Voici la signification des variables et prédicats utilisés. L'entier `M` dénote les "étapes" (1 au départ et croît de 1 à chaque traversée), `S` est un soldat ou la lampe et `L` un lieu. À l'étape M :

`at(M,L,S)` : S est en L,
`move(M,L,S)` : S part vers L,
`duree(M,T)` : les soldats ont mis un temps T [en incluant la partie move(M,L,S), donc ici M traversées].

Les noms de variables (apparaissant comme paramètre de prédicat ou de fonction) commencent par une majuscule, contrairement aux noms de constantes, fixées une fois pour toutes à chaque exécution d'un programme.

Initialisation : 4 soldats et lampe au départ à l'étape 1, durée totale du trajet nulle à "l'étape 0" (avant 1) :

```
at(1,depart,S):-soldat(S).
at(1,depart,lampe). duree(0,0).
```

Génération des déplacements : Un soldat ne peut traverser que s'il part du côté de la lampe. On essaie de traverser tant que le but n'est pas atteint [`not goal(M-1)`, facultatif, mais accélère]. `not` est la *négation par défaut* : `not goal(M-1)` est vrai quand `goal(M-1)` ne figure pas dans l'ensemble réponse considéré. Ne plus traverser après "tmax" (explosion). Tout cela est aisément traduit par une règle particulière dite *de génération* indéterministe ("`{...}`", *si le corps est satisfait,* `move(M,L,S)` *peut être pris ou pas*) :

```
{move(M,L,S)} :- at(M,L0,S), soldat(S),
   at(M,L0,lampe), lieu(L), L0!=L,
   duree(M-1,T), T<tmax, not goal(M-1).
```

Résultat du déplacement représenté par `move(M,L,_)` :
(1) Ceux qui arrivent en L, à l'étape $M+1$ suivante :

```
at(M+1,L,S) :- move(M,L,S).
```

(2) *Axiomes du cadre* Ceux qui ne se sont pas déplacés restent en `LO` (2a) ou en `L` (2b) :

```
(2a) at(M+1,L0,S) :-
      move(M,L,S1), at(M,L0,S), L0!=L,
      not move(M,L,S).
(2b) at(M+1,L,S) :-
      move(M,L,S1), at(M,L,S).
```

Les *axiomes du cadre* résolvent un problème fondamental pour toute approche logique décrivant une situation évolutive : il faut rigoureusement préciser, non seulement tout ce qui est modifié, mais aussi tout ce qui ne l'est pas. Les axiomes du cadre précisent donc ce qui n'est pas modifié, et l'usage du `not` permet d'écrire cela de façon naturelle et concise. D'ailleurs les deux axiomes donnés ici sont un peu particuliers au problème, pour accélérer, mais un seul axiome du cadre, encore plus simple, et généralisable, suffirait.

Moins de 3 soldats traversent. On utilise une *contrainte*, règle sans tête, (`tête`$\equiv \bot$), qui élimine les ensembles réponses satisfaisant le corps :

```
:- move(M,L,S1;S2;S3),
   soldat(S1;S2;S3), S1<S2, S2<S3.
```

Utiliser la notation clingo en "`;`" allège l'écriture (et accélère) [`move(M,L,S1;S2...)` remplace `move(M,L,S1), move(M,L,S2),...`].

Utiliser $<$ au lieu de `!=` (qui représente $\neq$) est ici équivalent (symétrie entre les variables concernées) et diminue la taille de l'instantiation.

Un soldat suffit à porter la lampe, qu'il faut emporter à chaque traversée :

```
move(M,L,lampe) :- move(M,L,S).
```

Le calcul du temps de traversée est peu élégant, car la version actuelle de clingo n'autorise pas ici les "méta-prédicats" #max et #sum (cela devrait évoluer ?) :

```
duree22(M,T0+T2) :- move(M,L,S1;S2),
   duree(M-1,T0),temps(S1,T1),
   temps(S2,T2), T1<T2.    % (cas de 2 soldats)
duree21(M) :- duree22(M,T).
duree(M,T) :- duree22(M,T).
duree(M,T0+T) :- move(M,L,S),
  duree(M-1,T0), temps(S,T),
  not duree21(M). % (cas où un seul soldat traverse)
```

Comme souvent en ASP dans un problème de planification, on décrit le but, et une contrainte élimine les modèles ne satisfaisant pas ce but :

```
goal(M-1) :- at(M,arriv,0;1;2;3).
goal0 :- goal(M).     % (but atteint, à l' étape M)
:- not goal0.         % (tout AS doit satisfaire le but)
```

On en déduit la durée totale du trajet :
`duree1(T) :- goal(M), duree(M,T).`
Deux terminaisons sont possibles pour ce petit programme.

1. Un méta prédicat d'*optimisation* détecte les modèles minimisant la durée :
   `#minimize [ duree1(T)= T].` ou
2. une contrainte élimine les modèles qui dépassent le temps permis :
   `:- duree1(T), T > tmax.`

Les deux fournissent le même résultat avec cet énoncé car 60 est le temps minimum possible. L'optimisation est assez bien implémentée car elle accélère un peu mais le plus important est qu'elle accepterait aussi un énoncé moins coopératif (qui ne donne pas la valeur optimale !). Voici les deux solutions, fournies en 1/10 s, où `(M,Lieu,S)` abrège `move(M,Lieu,S)`

```
(1,arriv,2) (1,arriv,3) (1,arriv,lampe)
(2,depart,2) (2,depart,lampe)
(3,arriv,0) (3,arriv,1) (3,arriv,lampe)
(4,depart,3) (4,depart,lampe)
(5,arriv,2) (5,arriv,3) (5,arriv,lampe)
duree1(60) Optimization : 60
```

L'autre solution échange l'ordre de passage de deux soldats, puisqu'elle remplace
`move(2,depart,2), move(4,depart,3)` par
`move(2,depart,3) move(4,depart,2).`

Le programme ASP est immédiat et naturel : Chaque règle correspond de façon directe à une partie de l'énoncé. Cette facilité est due au très petit espace de recherche nécessaire : il est inutile de réfléchir au problème, traduire l'énoncé suffit. Remarquons, par rapport à un simple datalog, que l'utilisations de la négation par défaut `not` dans l'axiome du cadre ("inertie") facilite considérablement l'écriture.
Voici un exemple où écrire le programme est moins simple.

3 Un exemple plus récalcitrant

3.1 Le vieil éléphant mangeur de bananes

> Un planteur a produit 3 000 bananes. Il ne dispose que d'un vieil éléphant qui consomme une banane au kilomètre et ne peut pas porter plus de 1000 bananes. Le marché se trouve à 1000 km de la plantation. Combien de bananes le planteur pourra-t-il porter au maximum au marché ?

L'espace des solutions envisageables est grand. Il faut élaguer l'espace de recherche sous peine de crash (`bad_alloc`). Un gros avantage d'ASP apparaît ici : il est facile d'introduire le fruit de telles réflexions de façon naturelle dans le programme. Mais certaines limites de l'avantage de l'aspect déclaratif apparaissent.
On remarque aisément qu'un trajet direct n'apporte aucune banane, il faut donc plusieurs voyages avec des dépôts intermédiaires (considérés sans perte !), on trouve aisément une solution avec 1 dépôt (situé à 400km du départ) qui apporte 400 bananes au marché :
L'éléphant part avec la charge maximale (1000 bannes), en laisse 200 au dépôt 1 (unique ici), revient au départ et repart avec 1000 bananes. Il en laisse 200 nouvelles au dépôt, revient une dernière fois pour emporter les 1000 bananes restantes, prend au passage les 400 bananes du dépôt (il en a justement mangé 400) et arrive donc avec 400 bananes au marché. Ce n'est pas si mal, mais il semble raisonable d'espérer mieux avec deux dépôts, on va donc chercher la meilleure solution avec deux dépôts. Mais ces limitations sont encore insuffisantes. Les matheux résolvent ce problème "à la main", sans programmer. Au contraire, le but ici est de trouver des procédés généraux, nécessitant le moins de réflexion spécifique au problème possible, tout en facilitant une programmation efficace.

Une méthode générale consiste à changer les unités de mesure (par l'introduction d'un *pas* ou `step`). Cela fournit un résultat, optimal à cette granularité-là, puis on diminue l'unité, avec un encadrement issu du résultat déjà obtenu. Il semble naturel de faire porter l'encadrement sur les distances des dépôts intermédiaires (`dd2intervalkm`). Les distances fournies par les solutions à 100 près permettent d'encadrer une seconde recherche à 33 près, etc. jusqu'à 1 près. Toutefois, on doit encore ajouter des hypothèses qui nécessitent une réflexion sur ce problème précis. Appelons un ensemble d'hypothèses *pertinent* si, à partir de tout itinéraire satisfaisant l'énoncé, on peut construire un itinéraire satisfaisant ces hypothèses qui apporte au moins autant de bananes au marché. Un ensemble est *faiblement pertinent* s'il est pertinent dans tous les cas où existe au moins un itinéraire satisfaisant ces hypothèses : cela permet de résoudre aussi le problème si les données numériques sont légèrement modifiées. Le cas se présente avec les hypothèses faites ici, par exemple pour 2900 bananes (qui n'est plus un multiple de la charge maximum) au lieu de 3000. On essaie l'hypothèse forte (*correspondant à une pertinence faible*), et s'il n'y a pas de solution, l'hypothèse plus faible qui produira un espace de recherche plus grand. Voici les hypothèses considérées ici (vérifier leur pertinence sort du cadre de ce papier) :

1. Nombre de dépôts (2 intermédiaires) et d'étapes (9).
2. Les trajets s'arrêtent dès le marché atteint.
3. Distances entre dépôts croissantes.
4. Charge maximale des parcours aller (2 variantes) :
   (a) charge maximale toujours (*pertinence faible*),
   (b) ou charge maximale si disponible, sinon tout ce qui est disponible (*pertinence forte*).
5. Revenir sans bananes pour les parcours retour (pertinence facile à démontrer pour cette hypothèse).

3.2 Un programme simple (non satisfaisant)

Il est facile d'écrire un programme semblable au précédent §2, mais nécessitant trop d'appels successifs avec des unités décroissantes et des encadrements de plus en plus fins

(cf [7]). On a vu sur l'exemple avec un seul dépôt que les paramètres à faire varier sont les distances entre dépôts, et les charges emportées (ou laissées, l'une se déduisant de l'autre) à chaque passage à un dépôt.
Voici les points essentiels du programme.
D'abord, on va générer l'ensemble des couples de distances possibles entre les dépôts, satisfaisant nos hypothèses (la dernière distance se déduisant des deux premières puisque la somme des trois distances doit être 1000 km). On va générer, comme avec le pont miné, un ensemble réponse par trajet.

Voici quelques définitions préliminaires en commençant par les constantes (certaines venant de l'énoncé, d'autres des hypothèses, d'autres ajustables par l'utilisateur).
`#const load = 1000.` charge maxi de l'éléphant, en bananes
`#const dist = 1000.` distance à parcourir en km
`#const ban = 3000.` nombre de bananes au départ

`#const ndep = 3.` pour le nombre de dépôts
`d(0..ndep).` ndep+1 dépôts utilisés (départ/arrivée compris)
`#const stagemax = 9.` constante, nombre maxi d'étapes non finales
`stage(0..stagemax).` intervalle des étapes possibles

La constante "step" modifiable par l'utilisateur, permet de modifier la précision du résultat :
`#const step=33.` (l'objectif est de finir avec 1)

Valeurs des précédentes constantes en "unité" (step) :
`#const leu = load/step.` Charge maximale de l'éléphant
`#const du = dist/step.` Distance totale entre départ (dépôt 0) et arrivée (dépôt 3)
`#const bu = ban/step.` Nombre de bananes au départ
`ldn(0..bu).` Quantités de bananes possibles
`len(1..leu).` Charges de l'éléphant possibles

`dd012(DD0,DD1,DD2)` : `DDi`, distance possible entre dépôts `i` et `i+1`, tenant compte d'éventuelles restrictions imposées (grâce à `dd2valkm`), et de l'hypothèse 3 p. 3 :

`encadr(D) :- dd2valkm(D,Infkm,Supkm).`
L'utilisateur peut encadrer les distances entre dépôts possibles en spécifiant des limites en donnant des valeurs mini et maxi pour les deux premières distances (exemples en fin du présent §3.2 et en fin de §3.3). Si c'est le cas, `encadr(1)` et/ou `encadr(2)` seront satisfaits.

```
dd2val(D,1..leu/2-1) :- d(D), D+1 < 3,
    not encadr(D).
dd2val(D,Infkm/step..Supkm/step) :-
    D < 3, dd2valkm(D,Infkm,Supkm).
dd012(DD0,DD1,DD2) :- dd2val(0,DD0),
    dd2val(1,DD1), DD0 <= DD1,
    DD2 = du - DD0 - DD1, DD1 <= DD2.
dd01(DD0,DD1) :- dd012(DD0,DD1,DD2).
```

Le prédicat `dd` décrit à la fois les distances entre dépôts et la relation de voisinage entre dépôts, il est "indexé" par `dd01(DD0,DD1)` qui donne les distances entre les premiers dépôts (le symbole – de fonction et de prédicat – `dd01`, facultatif, facilite la lecture) :

```
dd(0,1,DD0) :- dd012(DD0,DD1,DD2),
    ddescr(dd01(DD0,DD1)).
dd(1,0,DD0) :- dd012(DD0,DD1,DD2),
    ddescr(dd01(DD0,DD1)).
dd(1,2,DD1) :- dd012(DD0,DD1,DD2),
    ddescr(dd01(DD0,DD1)).
dd(2,1,DD1) :- dd012(DD0,DD1,DD2),
    ddescr(dd01(DD0,DD1)).
dd(2,3,DD2) :- dd012(DD0,DD1,DD2),
    ddescr(dd01(DD0,DD1)).
```

(*Fin des définitions préliminaires* données ici)

Ensuite vient la règle de génération (1 {...} 1) des couples de distances qui génère un et un seul terme en `ddescr` parmi les possibles (satisfaisant les conditions décrites par `dd012(DD0,DD1,DD2)`):

```
1 { ddescr(dd01(DD0,DD1)) :
        dd012(DD0,DD1,DD2)} 1.
```

`dd01(DD0,DD1)` est un terme qui décrit partiellement l'ensemble réponse (il le décrit en ce qui concerne les distances, restera ensuite à tenir aussi compte des charges emportées).

On introduit les prédicats suivants :
`move(S,D,LE)` : À l'étape S (de 0 à stagemax, cf Hypothèse 1) l'éléphant quitte le dépôt D0 où il est [`move(S-1,D0,LE0)`] pour aller vers le dépôt D, voisin de D0, avec la charge LE.
`loadD(S,D,LD)` En S, la charge du dépôt D après le départ de l'éléphant du dépôt D0 est LD.
`loadDS(S,D,LD)` est un `loadD(S,D,LD)` où LD est suceptible d'être modifié (l'éléphant passe en D).
`loadE(S,LE)` : En S, la charge emportée est LE.

Initialisations (S=0 ou 1). Mouvement fictif (S=0) :
`move(0,0,0).` Stand by...
`loadDS(0,0,bu).` Toutes le bananes au départ en S=0
`loadDS(0,D,0) :- d(D), D>0.` Dépôts vides en S=0, sauf départ (dépôt 0).

Premier mouvement (S=1) : l'éléphant quitte 0 pour 1 avec la charge maximale possible leu (hypothèse 4, a ou b).

```
move(1,1,leu).
loadDS(1,0,B-LE) :- move(1,1,LE),
    loadD(0,0,B).
loadD(1,D,0) :- loadD(0,D,0), d(D), D>0.
```

Pour les mouvements suivants, on utilise une règle de

génération indéterministe (signalée par les {}, on génère, ou pas, un mouvement possible, on d'éliminera ensuite les modèles ne satisfaisant pas certaines conditions) :

Mouvement général "aller" (hypothèse 4 p. 3) :

```
{move(S+1,D+1,leu)} :- move(S,D,LE),
   move(S-1,D0,LE0), dd(D0,D,DD0),
   dd(D,D+1,DD), loadD(S,D,LD0),
  LDA=LD0+LE-DD0, LDA>=leu, S<stagemax.
```

Si l'hypothèse forte 4a p. 3 ne convient pas, ajouter la règle suivante pour obtenir l'hypothèse faible 4b :

```
{move(S+1,D+1,LDA)} :- move(S,D,LE),
   move(S-1,D0,LE0), dd(D0,D,DD0),
   dd(D,D+1,DD), loadD(S,D,LD0),
   LDA=LD0+LE-DD0, LDA<leu, S<stagemax.
```

Mouvement "retour", de D à D-1 (hypothèse 5 p. 3)

```
{move(S+1,D-1,DD)} :- move(S,D,LE),
   move(S-1,D0,LE0), dd(D0,D,DD0),
   dd(D,D-1,DD), loadD(S,D,LD0), LDA =
   LD0+LE-DD0, LDA >= DD, S<stagemax.
```

Un seul mouvement par étape S (contrainte) :

```
:- move(S,D,LE), move(S,D1,LE1), D<D1.
```

Calcul de loadD(S,_) ($2 \leq S \leq stagemax$) :

```
loadDS(S+1,D,LD1) :- move(S+1,D1,LE1),
   move(S,D,LE), move(S-1,D0,LE0), S>0,
   dd(D0,D,DD), loadD(S,D,LD),
   LD1 = LD+LE-DD-LE1, LD1 >= 0.

loadD(S,D,LD) :- loadDS(S,D,LD).
```

Axiome du cadre pour les autres dépôts (qui ne changent pas de charge), dans ce cas simple, inutile d'utiliser "not", " !=" ($\neq$) suffit :

```
loadD(S+1,D,LD) :- move(S+1,D1,LE1),
  move(S,D0,LE0), loadD(S,D,LD), D!=D0.
loadES(S) :- move(S,D,LE). S étape non finale.
```

La contrainte suivante élimine les trajets qui ne vont pas jusqu'en 3 (marché) :

```
:- move(S,D,LE), D<3, not loadES(S+1).
```

Charge apportée au marché (`move(stagemax,3,LE)` indique que le but -dépôt 3- est atteint, cf hypothèse 2) :

```
loadAu(stagemax+1,LE-DD2) :-
   move(stagemax,ndep,LE),
   dd(ndep-1,ndep,DD2).
```

Il reste à retourner aux unités compréhensibles (nombre de bananes et km), et à lancer l'optimisation (`#maximize` donne le plus grand nombre possible de bananes, cf §2) :

```
loadA(S,LU * step) :- loadAu(S,LU).
#maximize [ loadA(S,LD) = LD].
ddkm(I,J,DD * step) :- dd(I,J,DD), I<J.
```

Ce programme facile à écrire (donc, plus important, à modifier, par exemple pour améliorer ses performances) trouve très laborieusement la solution :

Un premier lancement avec `step=100` donne une solution à 100 près : `loadA(10,500) ddkm(1,2,300) ddkm(0,1,200) [...]` : 500 bananes au marché, 200km entre dépôts 0 et 1, 300km entre 1 et 2, ce dont on s'inspire pour relancer le programme avec les données suivantes (la précision passe de 100 à 33 bananes ou km ; distances 200 et 300 encadrées à 50 km près) :

`step=33, dd2intervalkm(0,150,250), dd2intervalkm(1,250,350).` Il faut encore relancer plusieurs fois en améliorant petit à petit la précision jusqu'à 1 (sauf à deviner tout de suite des intervalles de distance très précis). Diverses tentatives d'amélioration du programme ont échoué. Il en existe sûrement, mais l'aspect "déclaratif" affiché s'éloigne...

3.3 Un autre type de programme

Au lieu de placer, comme c'est naturel en ASP, une solution par AS, on place l'ensemble des solutions dans un seul AS. Chaque trajet nécessite donc un indice complexe qui contient le couple (DD0,DD1) des distances respectives entre les dépôts 0 (départ) et le premier intermédiaire, et entre ce premier et le second dépôt intermédiaire, et la liste des dépôts visités et des charges de l'éléphant.

On utilise des symboles fonctionnels. Les premiers systèmes ASP n'avaient pas cette possibilité, et une telle solution n'était pas envisageable car elle aurait nécessité une énumération trop coûteuse de l'ensemble des trajets possibles, afin de pouvoir les indexer. On indexe ici chaque trajet par une (simulation de) liste. Il serait préférable de disposer de *listes* explicites. Ces listes existent dans des extensions de certains systèmes ASP, comme DLV-Complex[3].

Ici, on n'a pas besoin de fonctions complexes sur les listes, et donc clingo convient, en étant plus efficace. En effet, des tests avec DLV seul et DLV-Complex (pour le pont miné et le premier programme de l'éléphant) ont montré une si nette supériorité de clingo qu'il semble peu probable que DLV-Complex se montre efficace ici.

On utilise des symboles de fonction pour construire les nouveaux termes, remplaçant le *prédicat* `move` par une *fonction* `m`. Le programme complet est un peu long pour figurer ici, mais voici les éléments essentiels (cf [7] pour plus de détails).

Le début est inchangé (cf définitions préliminaires p. 4) : `ddO12` et `dd01` sont identiques, par contre, le prédicat `dd` est ici différent de celui donné dans le programme "classique" donné en §3.2. Il fournit là encore les distances entre dépôts et la relation de voisinage entre dépôts, mais il doit ici être "indexé" par `dd01(DD0,DD1)` qui donne les distances entre les premiers dépôts. Cela permet de placer tous les résultats dans un seul AS, sans les "mélanger".

```
dd01(DD0,DD1) :-
     dd012(DD0,DD1,DD2).
```

```
dd(dd01(DD0,DD1),0,1,DD0) :-
     dd01(DD0,DD1).
dd(dd01(DD0,DD1),1,0,DD0) :-
     dd01(DD0,DD1).
dd(dd01(DD0,DD1),1,2,DD1) :-
     dd01(DD0,DD1).
dd(dd01(DD0,DD1),2,1,DD1) :-
     dd01(DD0,DD1).
dd(dd01(DD0,DD1),2,3,DD2) :-
     dd012(DD0,DD1,DD2).
```

Voici comment sont décrits les trajets, en utilisant des symboles de fonction `m` ("mouvement") et de prédicats `i` ("itinéraire") et `itiopt` (trajets `i` optimaux trouvés). On utilise aussi le symbole de fonction `l` (pour "load"), `l(D,LD)` signifie que la charge du dépôt D est LD.

Le premier paramètre de `i` (terme en `dd01`) contient l'information sur l'espacement des dépôts. Le second `S` indique le numéro de l'étape (redondant, contenu dans le terme en `m`, mais commode). Le troisième est le terme en `m` qui représente la suite des dépôts dans leur ordre de visite. Le quatrième est le terme en `l` déjà décrit.
Un mouvement est décrit en un seul terme fonctionnel, et non plus par un prédicat `move` (plus aisé à manipuler) ne décrivant qu'une étape.

`m(M,D,LE)` : le mouvement obtenu à la suite du mouvement précédent `M` , `D` est le dépôt où l'éléphant arrive, `LE` la charge avec laquelle il a quitté le dépôt `D0` précédent [`M` est `m(Ma,D0,LE0)`, `D0` étant le dépôt quitté, avec la charge `LE0`, `Ma` étant le mouvement d'avant].

L'écriture des règles n'est pas facilitée : l'aspect déclaratif est moins net !

Initialisations
Trajet initial (sur place) : `#const mi = m(m0,0,0).`

```
i(dd01(DD0,DD1),0,mi,l(0,bu)) :-
     dd01(DD0,DD1).
i(dd01(DD0,DD1),0,mi,l(D,0)) :-
     dd01(DD0,DD1), d(D), D > 0.
```

Premier vrai trajet, du dépôt 0 vers le dépôt 1 :

```
i(dd01(DD0,DD1),1,m(mi,1,leu),
  l(0,bu-leu)) :-  dd01(DD0,DD1).
i(dd01(DD0,DD1),1,m(mi,1,leu),
   l(D,0)) :-  dd01(DD0,DD1), d(D), D>0.
```

Poursuite du trajet : parcours "retour", D = D0 -1 :

```
i(dd01(DD0,DD1),S+1,m(m(m(Ma,Da,LEa),
   D0,LE0),D0-1,DP),l(D0,LD)) :-
    i(dd01(DD0,DD1),S,m(m(Ma,Da,LEa),
       D0,LE0),l(D0,LD0)),
    dd(dd01(DD0,DD1),D0,D0-1,DP),
    dd(dd01(DD0,DD1),Da,D0,DP0),
    S<stagemax,LD=LD0+LE0-DP0-DP,LD≥0.
```

La règle pour les parcours "aller" est similaire, et l'axiome du cadre est lui aussi assez complexe à écrire/lire) (cf [7]).

Il est clair que ces règles sont beaucoup moins lisibles que les règles plus classiques données par exemple en §2. L'avantage est que les performances sont très supérieures au premier programme, et on peut considérer le problème résolu (voir ci-dessous). Il est dommage que la méthode préconisée en général pour les problèmes de planification (un AS par tentative) rencontre un problème de taille mémoire. L'idéal serait que le système soit capable de traiter le programme en `move` de façon aussi économique que ce programme en `m`. Il n'est pas certain que cela soit si facile à implémenter que cela. En effet, dans le programme en `m`, l'utilisateur guide un peu les calculs, et il reste à voir ce qui est facilement automatisable dans ce passage de `move` et `at` vers l'utilisation de la fonction `m`.

Remarquons qu'ici la *structure* du programme est très simple (au détriment de l'*écriture des règles*). En particulier, la fonction clingo `#max` qui n'était pas utilisable dans le programme du pont miné (ou le premier programme de l'éléphant) est ici possible.

Voici une idée des performances de ce programme sur un ordinateur "ordinaire". `step=1` sans encadrement ne convient pas (voir toutefois ci-dessous §3.4) mais la suite
(1) `step =10` (seul : sans encadrement des distances)
(2) `step=1, dd2valkm(0,170,230), dd2valkm(1,300,360)`
passe sans problème en environ 5s, et donne ceci :

```
itiopt(dd01(200,333),loadA(533),
i(dd01(200,333),9,
  m(m(m(m(m(m(m(m(m(mi,1,1000),
  0,200),1,1000),0,200),1,1000),2,1000),
  1,333),2,1000),3,1000), l(2,1)))
```

et un autre trajet similaire avec également 533 bananes.

3.4 La surprise gringo

Si on lance gringo3 ("instantiateur" sans le "solveur" clasp), avec `step(1)` seul (sans intervalle), on obtient le résultat sur l'ordinateur utilisé. Cela donne donc le résultat cherché directement, sans utiliser d'unité interne ni d'encadrement a priori (mais en utilisant les hypothèses 1 – 5 p. 3). L'ennui est que clingo (gringo + clasp) *ne sait pas* que gringo seul suffirait. Et d'après Torsten Schaub, il n'est pas facile de faire que clingo s'en aperçoive assez tôt pour éviter d'utiliser de la mémoire de façon inutile (gringo3 seul pourrait-il s'en apercevoir ?). Des tests avec l'hypothèse faible suggèrent que cette marge où gringo suffit et réusssit et où clingo ne passe pas, n'est pas si étroite que cela, et donc détecter ces cas augmenterait les possibilités de gringo3/clingo.

3.5 Au sujet des listes

La version actuelle de clingo permet de simuler les *listes*. L'utilisation de vraies listes (et aussi d'ensembles) serait

souvent très utile, avec les fonctions classiques associées. DLV-Complex permet d'utiliser des listes et des ensembles explicites, avec quelques fonctions utiles. Comme la base est DLV, cela peut parfois nuire aux performances. Il convient donc, dans l'état actuel des choses, d'examiner ce que clingo permet au sujet des listes. Nous avons utilisé en §3.3 la méthode classique suivante. Soit `l` un symbole de fonction binaire, une liste $la = [a, b, c, a]$ est représentée par le terme clingo `l(a,l(b,l(c,l(a,end))))`, et la liste vide [] par `end`. Cette méthode, qui convient pour une utilisation rudimentaire des listes, présente certains inconvénients.

Supposons que l'on dispose d'un ensemble de listes comme celle donnée ci-dessus, et que l'on ait besoin de fonctions simples. Ces fonctions nécessitent des définitions récursives, et, pour que la récursivité "passe", il faut considérer d'autres listes que celles données au départ. Par contre, il faut éviter de considérer toutes les listes possibles (jusqu'à une certaine longueur par exemple) avec tous les éléments des listes connues, sous peine de débordement mémoire. Il faut donc, pour les récursivités les plus simples, au moins toutes les sous-listes commençantes et finissantes des listes données, voire toutes les sous-listes commençantes des finissantes. On peut alors définir les fonctions longueur, inversion,...

Voici une possibilité : On part d'un ensemble de listes L du type ci-dessus, satisfaisant un prédicat list (`list(L)`). On définit d'abord les prédicats listfin(L), listcom(L), ssliste(L) : L est une liste finissante (respectivement commençante, sous-liste contigüe) d'une liste donnée :

```
listfin(L) :- list(L).
listfin(L) :- listfin(l(E,L)).
listel(E) :- listfin(l(E,L)).
                        % E élément d'une liste donnée
exceptLast(l(E,end),end) :- listel(E).
exceptLast(l(E,LR),l(E,LR1)) :-
   sslist(l(E,LR)), exceptLast(LR,LR1).
listcom(L) :- list(L).
listcom(L) :- listcom(L1),
   exceptLast(L1,L).
sslist(L) :- listcom(L).
sslist(L) :- sslist(l(E,L)).
```

On peut maintenant définir longueur `length` et inversion [`rev`] des listes de départ (et en prime de chaque "sslist") :

```
length(end,0).
length(l(E,LR),L+1) :- sslist(l(E,LR)),
   length(LR,L).
rev(end,end). %[] est sa propre inverse
rev(l(E1,L),LRIE1) :- sslist(l(E1,L)),
   rev(L,LRI), addLastI(E1,LRI,LRIE1).
addLastI(E,end,l(E,end)) :- listel(E).
addLastI(E1,l(E2,LR2),l(E2,LR2E1)) :-
   listel(E1), rev(Lx,l(E2,LR2)),
   addLastI(E1,LR2,LR2E1).
```

Dans la définition de la fonction auxiliaire `addLastI`, il faut admettre les listes déjà inversées comme domaine (cf `rev(Lx,l(E2,LR2))`). Ces détails compliquent beaucoup l'écriture des programmes ASP. Et, cela milite en faveur d'une extension des systèmes ASP aux listes (et ensembles), avec des opérations prédéfinies comme actuellement les opérations arithmétiques de base. Sans résoudre le problème général (et plus complexe, mais qu'il faudra bien aussi résoudre petit à petit...) des "sous-programmes ASP", cela faciliterait déjà sensiblement l'utilisation pratique de listes et d'ensembles.

4 Conclusion

L'état actuel des systèmes ASP permet de traiter de nombreux problèmes, comme ceux de planification donnés ici, de difficulté moyenne. Il s'agit d'un formalisme très général, facilitant l'écriture et la modification des programmes, alors qu'un formalisme spécifique demande une phase de familiarisation. On a donné deux exemples : un cas simple où la traduction ASP est quasi littérale et très naturelle, un plus complexe nécessitant des optimisations, elles aussi aisément traduisibles en ASP. Pour une utilisation plus aisée, les systèmes actuels devraient encore être un peu améliorés, ce qui est naturel pour un formalisme général et récent. L'exemple de l'éléphant, plus aisé à traiter à l'aide d'un seul ensemble réponse (ce qui contredit la plupart des préconisations usuelles) semble indiquer qu'il serait bon que l'utilisateur puisse indiquer au système comment traiter certains prédicats. Ainsi, dans l'exemple du pont miné, le fait pour l'utilisateur de préciser que `at` doit satisfaire l'axiome du cadre faciliterait l'écriture du programme, mais surtout l'efficacité du calcul : des méthodes spécifiques permettraient d'épargner de la mémoire. Un méta prédicat du genre `#frame` où `#frame at/1/1/1.` signifierait que `at` doit être traité comme satisfaisant l'axiome du cadre : le premier paramètre étant le temporel, le second la valeur soumise à l'axiome du cadre (restant identique pour chaque indice, sauf si une règle demande de la modifier) et le troisième l'indice. Les axiomes du cadre seraient générés par le système, et traités de façon efficace. On ne peut pas aller trop loin dans cette direction car on arriverait à un système spécifique. Mais le gain en facilité d'expression et de calcul, ainsi que le caractère général de ce comportement, semble justifier ce genre d'ajout.

L'ajout de structures de données plus complexes (listes, ensembles etc.), serait aussi appréciable pour l'utilisateur. Là encore, cela améliorerait les performances car l'introduction dans les système de fonctions de base comme l'inversion de liste éviterait la complication de la définition précise du "domaine" dans des règles écrites par l'utilisateur pour effectuer les opérations classiques.

L'introduction de "sous-programmes" (comme cela existe déjà dans certains systèmes, mais pas toujours de façon très performante) serait évidemment elle aussi très utile. Il est

en effet très lourd de réécrire les mêmes règles en différents endroits, avec à chaque fois un "domaine" différent. Il faut toutefois reconnaître que cet ajout aux systèmes actuels serait beaucoup plus complexe que les autres suggestions faites ici, et que donc il faudra sans doute attendre encore un peu pour avoir des versions élégantes et efficaces... Il semble dans un premier temps que de (relativement...) petites évolutions des systèmes actuels, orientées confort de l'utilisateur, pourraient améliorer leurs performances effectives (et non pas théoriques et aisément mesurables). Un méta prédicat `frame` suffirait probablement à ce que l'exemple de l'éléphant soit immédiat à écrire avec clingo : Cela pourrait suffire pour que cet exemple soit résolu par la méthode classique, beaucoup plus facile à écrire que la méthode du §3.3. Une meilleure interconnexion entre instantiateur et solveur faciliterait aussi le traitement de cet exemple. Mais ce problème, reconnu comme fondamental depuis les débuts de l'ASP, est loin d'être simple. Toutefois, on peut espérer de petits progrès. Par exemple, le fait que l'"instantiateur" (qui est en fait déjà nettement plus qu'un simple instantiateur) gringo3 suffise à traiter le problème de l'éléphant montre que là, il "suffirait" que le système complet (instantiateur + solveur) détecte ces cas pour le problème soit résolu (par la seconde méthode). Et une combinaison d'un méta axiome du cadre et de cette reconnaissance, permettrait de résoudre ce problème par la première méthode, plus élégante.

Une autre amélioration, non évoquée ci-dessus, faciliterait l'écriture de nombreux programmes. En effet, il est fréquent qu'un programme ne passe pas, alors que si on le découpe, il passe sans problème en quelques secondes. Il s'agit de s'arrêter à des résultats intermédiaires, de lancer ensuite la partie suivante du programme en partant de ces résultats intermédiaires. C'est lourd et peu élégant. Si l'utilisateur pouvait indiquer au système ces prédicats intermédiaires comme *auxiliaires de niveau 1,2,..*, le système pourrait décharger l'utilisateur de ces complications. Une étape ultérieure consisterait pour le système à déterminer ces prédicats auxiliaires, mais cela nécessite de plus sérieux aménagements...

Nous espérons avoir convaincu le lecteur, malgré ces petits désagréments, de l'utilité d'ASP pour résoudre de nombreux problèmes. Et peut-être aussi d'avoir convaincu des créateurs de systèmes effectifs que certaines améliorations (pas toutes nécessairement complexes) pourraient, sans apporter d'avantage théorique visible, faciliter la vie des programmeurs en ASP et améliorer ainsi la diffusion de ce merveilleux outil.

Remerciements

L'auteur remercie les relecteurs pour leurs lecture attentive et leurs commentaires constructifs.

Références

[1] BARAL C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

[2] CALIMERI F., COZZA S., IANNI G. & LEONE N. (2009). An ASP System with Functions, Lists, and Sets. In LPNMR'09(LNAI 5753), p. 483–489.

[3] DENECKER M., VENNEKENS J., BOND S., GEBSER M. & TRUSZCZYŃSKI M. (2009). The 2^{nd} Answer Set Programming Competition. In LPNMR09 (LNAI 5753), p. 637–654.

[4] GEBSER M., KAUFMANN B. & SCHAUB T. (2009). The conflict-driven answer set solver *clasp* : Progress report. In LPNMR09 (LNAI 5753), p. 509–514.

[5] LEFÈVRE C., NGOMA S. & NICOLAS P. (2010). ASPeRiX : un solveur ASP du premier ordre. *IAF 2010* `//gdri3iaf.info.univ-angers.fr/spip.php?article121`

[6] LEONE N., PFEIFER G., FABER W., EITER T., GOTTLOB G., PERRI S. & SCARCELLO F. (2006). The DLV System for Knowledge Representation and Reasoning. *ACM Trans.TOCL*, **7**(3), 499–562.

[7] MOINARD,Y. Utilisation de la programmation par ensembles réponses (Answer Set Programming) sur de "petits" problèmes (JIAF 2011) `//hal.inria.fr/inria-00619527/en`