Utilisation de la programmation par ensembles réponses (Answer Set Programming) sur de “petits” problèmes

Yves Moinard[1]

INRIA-Bretagne Atlantique
IRISA, campus de Beaulieu, 35042 RENNES yves.moinard@inria.fr

Résumé : En traduisant en programmation par ensembles réponses deux petites devinettes classiques, nous illustrons la puissance et certaines limites des systèmes actuels. Le problème n’est pas de traduire le problème de la manière la plus efficace. Il s’agit de le traduire d’une façon pas trop ad-hoc, qui tire pleinement profit de l’aspect déclaratif de ce type de programmation. Ces exemples montrent qu’on n’est pas loin de cet objectif affiché de la programmation par ensembles-réponses sur de tels exemples, même s’il reste des progrès à faire pour les systèmes existant. Nous évoquons deux pistes : une meilleure intégration entre l’instantiateur et le solveur, permettant au moins de détecter les cas où le premier suffit, et une recherche d’indexation automatique permettant de diminuer la mémoire nécessaire.

Mots-clés : Programmation logique, ensembles réponses, planification

1 Introduction

On examine ici la facilité avec laquelle on peut traduire de petits problèmes en un programme exécutable en programmation par ensembles réponses (que nous abrégerons par l’acronyme anglais ASP pour “answer set programming”) (Baral, 2003). Depuis gringo3 (juillet 2010) la lignée clingo (gringo + clasp)[1], (Gebser *et al.*, 2009) semble la plus efficace en général. Auparavant, il était souvent plus commode de traduire ces problèmes grâce à DLV (Leone *et al.*, 2006)[2] et ses extensions comme DLV-Complex[3] (Calimeri *et al.*, 2009), qui admettent des structures de données plus riches (suites, ensemble), avec parfois la possibilité d’intégrer de petits scripts de calcul. 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, l’intégration de scripts a aussi été ajoutée (non utilisée ici). D’autre part les symboles de fonction permettent de simuler (plus ou moins élégamment) des suites. Ainsi, les

[1] http ://sourceforge.net/projects/potassco/
[2] http ://www.dbai.tuwien.ac.at/proj/dlv/
[3] https ://www.mat.unical.it/dlv-complex

principaux avantages qui existaient en faveur de l'utilisation de DLV se sont estompés. L'aspect ouvert de clingo, contre un système DLV moins transparent, en plus de ses performances en général meilleures, apporte des arguments en faveur de clingo. D'autres sytèmes[4], comme Asperix[5] (Lefèvre *et al.*, 2010) ou Cmodels[6] demeurent en version préliminaire ou (parfois inclusif) sont d'une utilisation pratique plus complexe.

La programmation par ensemble réponse constitue sans doute le système le plus proche de l'idéal de la programmation déclarative : *le problème est le programme*, inutile de se préoccuper de la façon de calculer la solution. 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. Selon le problème posé, cette préoccupation est plus ou moins complexe. La programmation par ensembles réponses est bien adaptée par exemple à des problèmes de parcours dans des graphes, et elle permet d'écrire des programmes élégants et naturels. Ce formalisme proche la logique classique traite de façon très naturelle de petits problèmes comme le fameux problème de *l'eau et du zèbre*. `http ://www.hakank.org/answer_set_programming/zebra.lp` de Hakan Kjellerstrand 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'une solution (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 ces systèmes, de résoudre un problème le plus rapidement possible et avec le moins de place mémoire. Pour tout langage informatique, ce seul critère est insuffisant, mais c'est encore plus insuffisant pour un langage déclaratif. Une citation extraite des résultats de la seconde compétition ASP évoque l'importance cet aspect, mais il demeure délicat de mesurer objectivement les performances en ce domaine :

> The Potassco team is the clear winner [semble encore vrai pour la Third competition] [...]. Given that the goal of declarative problem solving is to minimize the effort of programmers, it is of equal interest to investigate the performance of teams that used a single collection of systems with uniform parameter settings. (Denecker *et al.*, 2009).

Voici deux petits exemples avec des traductions ASP qui semblent "naturelles". Le premier est simple. Le second n'est pas trop complexe, mais il semble qu'aucun programme "simple et naturel" ne donne directement le résultat. Les systèmes actuels (comme clingo) permettent de le résoudre, mais il ne suffit plus de transcrire l'énoncé.

2 Un exemple de rêve : le pont miné

> Quatre soldats doivent franchir de nuit un pont qui sautera une heure après que le premier soldat aura mis le pied sur le pont. Les soldats sont plus

[4]Liste de systèmes : `http ://www.cs.uni-potsdam.de/~torsten/asp/`.

[5]`http ://www.info.univ-angers.fr/pub/claire/asperix/`

[6]`http ://www.cs.utexas.edu/users/tag/cmodels.html`

ou moins blessés ce qui donne leur durée respective de franchissement du pont en minutes : 25, 20, 10 et 5. Le pont ne supporte que deux soldats à la fois et l'unique lampe est indispensable pour traverser. Comment font-ils ?

Le programme qui suit est une copie quasi littérale de l'énoncé. C'est une suite de règles "`tête :- corps.`" : si chaque élément du corps est satisfait, la tête l'est. Pour les données on utilise des faits, ou règles sans corps (`corps`$\equiv \top$) :

```
soldat(0..3). temps(0,25). temps(1,20). temps(2,10).
temps(3,5). lieu(depart). lieu(arriv). #const tmax= 60.
```

(La notation `N1..N2` signifie que toute valeur entre N1 et N2 est possible.)

Voici la signification des prédicats introduits :

`at(M,L,S)` : À l'étape M, le soldat (ou la lampe) S est en L.
`move(M,L,S)` : En M, S part vers L, qu'il atteindra en M+1.
`duree(M,T)` : En M, les soldats ont mis un temps T pour effectuer leur trajet.

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, durée du trajet actuel nulle :

```
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 sensiblement]. `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 *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).
```

Description du résultat du déplacement représenté par les `move(M,L,_)` :

(1) Ceux qui arrivent en L : `at(M+1,L,S) :- move(M,L,S).`

(2) *Axiomes du cadre* Ceux qui ne se sont pas déplacés restent en `L0` (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).`

(Ces deux axiomes 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, ce qui équivaut à `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 `move(M,L,S1;S2...)` qui remplace `move(M,L,S1), move(M,L,S2),...` allège l'écriture (et accélère).

Utiliser $<$ au lieu de `!=` (qui représente $\neq$) 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, puis une contrainte élimine les modèles ne satisfaisant pas ce but :

```
goal(M-1) :- at(M,arriv,0;1;2;3).     (le but est atteint en M)
goal0 :- goal(M).                     (le but est atteint)
:- not goal0.                         (élimine les AS ne satisfaisant pas le but)
```

On en déduit la durée totale : `duree1(T) :- goal(M), duree(M,T).`

Deux terminaisons. Un méta prédicat d'*optimisation* détecte les modèles minimisant la durée (1) ; ou une contrainte élimine les modèles qui dépassent le temps permis (2) :

1. `#minimize [ duree1(T)= T].`
2. `:- 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 (0.140s. avec et 0.170 s. sans) et elle accepte un énoncé moins coopératif. Il y a deux solutions :

```
move(1,arriv,2) move(1,arriv,3) move(1,arriv,lampe)
move(2,depart,2) move(2,depart,lampe)
move(3,arriv,0) move(3,arriv,1) move(3,arriv,lampe)
move(4,depart,3) move(4,depart,lampe)
move(5,arriv,2) move(5,arriv,3) move(5,arriv,lampe)
duree1(60) Optimization : 60          L'autre solution est similaire.
```

Le programme ASP est immédiat et naturel, et fournit la solution demandée sans effort de programmation. 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 facilite sensiblement l'écriture.

On va maintenant étudier un exemple où écrire le programme n'est pas si simple.

3 Un exemple récalcitrant : l'éléphant et les bananes

3.1 Présentation du 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 porter que 1000 bananes au plus. 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 réfléchir avant de programmer sous peine de crash (`bad_alloc`). Il faut élaguer l'espace de recherche. Un gros avantage d'ASP apparaît ici : il est facile d'introduire le fruit de telles réflexions de façon

naturelle dans le programme. Mais des limites de l'aspect déclaratif apparaissent. Les matheux résolvent ce problème sans programmer. Au contraire, le but ici est de trouver des procédés les plus généraux possibles qui facilitent la programmation efficace.

Une méthode applicable à tout problème de ce type change les unités de mesure. On introduit un *pas* par exemple de 10 bananes (ou km), et on lance le programme. Cela fournit un résultat, optimal à cette granularité-là, qui permet de diminuer le pas, mais avec un encadrement issu du résultat grossier déjà obtenu. Il semble naturel de faire porter l'encadrement sur les distances des dépôts intermédiaires (indispensables ici). Les distances fournies par les solutions à 10 près permettront de lancer une seconde recherche, encadrée, jusqu'à obtenir le résultat à l'unité près. Malgré cela, il reste très difficile d'écrire un programme satisfaisant. Il faut donc élaguer l'espace de recherche par l'ajout d'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 hypothèse : cela permet de résoudre le problème si les données numériques sont légèrement modifiées. On essaie d'abord les hypothèses fortes, et s'il n'y a aucune solution, on essaiera l'hypothèse plus faible, laquelle produira un espace de recherche plus grand. Démontrer la pertinence des hypothèses considérées ici dépasse le cadre de cet article :

1. Imposer le nombre de dépôts (deux intermédiaires) et d'étapes (9),
2. Se limiter aux trajets qui s'arrêtent dès le marché atteint,
3. et aux distances entre dépôts croissantes.
4. Imposer la charge maximale aux parcours aller, avec deux variantes :
   (a) charge maximale toujours, ou
   (b) charge maximale si disponible, sinon tout ce qui est disponible.
5. Revenir sans bananes pour les parcours retour (facile à démontrer).

3.2 Un premier programme, semblable au précédent

Voici un programme, facile à écrire, mais à piètres performances.

`#const ban = 3000.` `#const load = 1000.` `#const dist = 1000.`
Ex. d'"unité de calcul" : `#const step =100.` Constantes en "unité de calcul" :
#const bu = ban/step. #const leu = load/step. #const du = dist/step.

`d(0..3).` (Dépôts utilisés, y compris départ 0 et arrivée 3)
`#const stagemax = 9.` (Nombre maxi d'étapes)
`stage(0..stagemax).` ("Étapes" – ou "temps"– possibles)

Valeurs possible, en "unités" (quantité de bananes, charge, distance) :
`ldn(0..bu). len(1..leu). dn1(1..leu).`

`dd01(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 :

```
encadr(D) :- dd2valkm(D,Infkm,Supkm).
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).
```

Première séparation des "ensembles réponse" (AS) : une seule configuration (atome en `descr`) de dépôts par AS. On génère un seul `ddescr(dd01(DD0,DD1))` par ensemble réponse grâce à la règle suivante (" `:` " signifie "tel que" pour clingo) :

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

`dd(D1,D2,DD)` : Les dépôts D1 et D2 sont voisins et distants de DD, fournit les distances, et les dépôts D2 voisins de D1 (l'hypothèse 2 évite de définir `dd(3,2,_)`). Le symbole fonctionnel facultatif `ddO1` facilite l'écriture de certaines règles :

```
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)).
```

`move(S,D,LE)` : À l'étape S l'éléphant quitte D0 [move(S-1,D0,LE0)] pour aller vers un voisin D, avec la charge LE.
`loadD(S,D,LD)` : En S, la charge de D après le départ de l'éléphant de D0 est LD.
`loadDS(S,D,LD)` est un `loadD(S,D,LD)` en un S où LD est modifiée.
`loadE(S,LE)` : En S, la charge emportée par l'éléphant est LE.

Initialisations, mouvement fictif (S=0) :

```
move(0,0,0):-d(0). loadDS(0,0,bu). loadDS(0,D,0):-d(D),D>0.
```

Premier vrai mouvement (S=1) : quitte 0 pour 1 avec LE.

```
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.
```

Mouvement général : 1. "aller", de D à D+1 (hypothèse 4a, une règle supplémentaire serait nécessaire pour l'hypothèse faible 4b) :

```
{ 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.
```

2. "retour", de D à D-1 (hypothèse 5) :

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

(Un seul `move` par étape `S`.)

Calcul de loadD (loadDS est utilisé pour le seul dépôt modifié à chaque étape) :

```
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).
```

```
loadD(S+1,D,LD) :- move(S+1,D1,LE1), move(S,D0,LE0),
   loadD(S,D,LD), D != D0.
```
(Axiome du cadre pour les dépôts non modifiés.)

```
loadES(S) :- move(S,D,LE).
```
(S est une étape non finale.)

Élimine les trajets qui n'arrivent pas au marché (3) :

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

`LE` dans `move(stagemax,3,LE)` est la charge apportée au marché, en unité `step`, et `#maximize` ne garde que les AS avec le plus grand nombre possible de bananes :

```
loadAu(stagemax+1,LE-DD2) :- move(stagemax,ndep,LE),
     dd(ndep-1,ndep,DD2).
loadA(S,LU * step) :- loadAu(S,LU).
```
(en bananes)

```
#maximize [ loadA(S,LD) = LD].
```

`ddkm(I,J,DD * step) :- dd(I,J,DD), I < J.` (Distances entre les dépôts en km, utile pour l'utilisateur qui diminuera la valeur de l'unité interne.)

Ce programme assez facile à écrire (et donc, et c'est plus important, à modifier, par exemple pour améliorer ses performances) trouve la solution, mais il faut plusieurs tentatives car lancer avec `step=1` sans encadrement produit `bad_alloc`.

On peut obtenir laborieusement le résultat ainsi :
Premier lancement `step=100.` donne (en 1/3s) deux AS optimaux (similaires), avec en particulier `loadA(10,500) ddkm(1,2,300) ddkm(0,1,200)`
dont on s'inspire pour relancer le programme avec les données suivantes :
`step=33.dd2intervalkm(0,150,250).dd2intervalkm(1,250,350).`
On trouve encore 2 AS optimaux, en moins de 2s :
`loadA(10,528) ddkm(2,3,462) ddkm(1,2,330) ddkm(0,1,198)`
On relance, avec ces données :
`#const step =10. dd2valkm(0,190,210). dd2valkm(1,320,350).`
On trouve encore deux optimaux en 1 s., avec un peu plus de bananes à vendre (530) :
`loadA(10,530) ddkm(2,3,470) ddkm(1,2,330) ddkm(0,1,200)`

À ce stade, il est possible mais long d'obtenir le résultat, car
`#const step=1. dd2valkm(0,194,206). dd2valkm(1,324,336).`
produit `bad_alloc`, bien que les intervalles exigés soit assez petits.
L'encadrement très fin suivant donne le résultat, mais c'est peu satisfaisant :
`#const step=1. dd2valkm(0,197,203). dd2valkm(1,327,333).`

Résultat (un des 2 optimaux fournis) en plus de 6mn :
move(0,0,0) move(1,1,1000) move(2,0,200) move(3,1,1000) move(4,2,1000) move(5,1,333) move(6,0,200) move(7,1,1000) move(8,2,1000) move(9,3,1000) loadA(10,533) ddkm(2,3,467) ddkm(1,2,333) ddkm(0,1,200)

De nombreuses tentatives d'amélioration du programme ont échouées. Il en existe sûrement, mais l'aspect "déclaratif" affiché s'éloigne.

3.3 Un autre type de programme

Au lieu de faire, comme c'est naturel en ASP, une solution par AS, on place la solution dans un seul AS. Les règles qui génèrent des solutions sont remplacées par des règles qui calculent toutes les solutions dans un seul AS. La règle suivante est supprimée :

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

Les deux règles générant les "mouvements généraux (allers et retours)" sont remplacées par des règles qui calculent tous les trajets possibles dans le même AS. Chaque trajet nécessite donc un indice complexe qui contient le couple (DD0,DD1) concerné et la liste des dépôts déjà visités, ainsi que des charges de l'éléphant. On utilise des symbole 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 quelques règles. Le début, jusqu'à la règle de génération suivante, est inchangé.

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

Cette règle est remplacée par un nouveau prédicat `dd` qui "indexe" chaque ancien `dd` par (DD0,DD1) (ici encore le symbole de fonction `dd01` est facultatif) :

```
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 fonctions `m` ("mouvement") et `i` ("itinéraire") au lieu du prédicat `move`. Un prédicat `itiopt` décrit les trajets `i` optimaux trouvés (voir un exemple ci-dessous dans le résultat). 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 ici, contenu dans le terme en `m`, mais simplifie des règles). 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` décrit ci-dessus.

Ainsi, un mouvement est maintenant représenté grâce à un symbole de fonction `m`, qui permet de le décrire en un seul terme, et non plus par un prédicat `move` qui ne décrit qu'une étape (mais était plus aisé à manipuler).

`m(M,D,LE)` : le mouvement obtenu à la suite du mouvement précédent `M` ,
`D` est le dépôt où l'éléphant arrive maintenant, `LE` étant la charge avec laquelle il est parti du dépôt `D0` précédent [donc `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 !
Un mouvement est décrit à l'aide de la fonction `m` au lieu du prédicat `move` ci-dessus.

Initialisations : `#const mi = m(m0,0,0).`

```
i(dd01(DD0,DD1),0,m(m0,0,0),l(0,bu)) :- dd01(DD0,DD1).
i(dd01(DD0,DD1),0,m(m0,0,0),l(D,0)) :- dd01(DD0,DD1),
    d(D), D > 0.
i(dd01(DD0,DD1),1,m(m(m0,0,0),1,leu),l(0,bu-leu)) :-
    dd01(DD0,DD1).
i(dd01(DD0,DD1),1,m(m(m0,0,0),1,leu),l(D,0)) :-
    dd01(DD0,DD1), d(D), D>0.
```

Poursuite du trajet : cas d'un 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 (là encore, deux règles avec l’hypothèse faible 4b). Voici l’axiome du cadre pour les dépôts autres que D0 (D = D0 +/- 1) :

```
i(dd01(DD0,DD1),S+1,m(m(m(Ma,Da,LEa),D0,LE0),D,LE),l(Di,LDi))
:-i(dd01(DD0,DD1),S+1,m(m(m(Ma,Da,LEa),D0,LE0),D,LE),l(D0,LD)),
   S < stagemax, Di != D0,
   i(dd01(DD0,DD1),S,m(m(Ma,Da,LEa),D0,LE0),l(Di,LDi)).
```

Le reste du programme est immédiat. Les performances sont très supérieures, et on peut considérer le problème résolu. Il est dommage que la méthode préconisée en général pour les problèmes de planification (un AS par tentative, ou presque) 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`. En effet, dans le programme en `m`, l’utilisateur guide les calculs. Remarquons qu’ici la structure du programme est très simple (au détriment de la facilité d’écriture). Il faudrait un système capable de constituer quelque chose de ce genre à partir des données en `move` et `at`. Est-ce faisable ?

Voici les performances de ce programme : `step=1` sans encadrement ne passe pas mais la suite (1) `step =10` (seul) (2) `step=1, dd2valkm(0,170,230)` `dd2valkm(1,300,360)` passe sans problème en environ 5s, et donne ceci :

```
itiopt(dd01(200,333),loadAU(533),loadA(533),
  i(dd01(200,333),9,m(m(m(m(m(m(m(m(m(m(m0,0,0),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.

3.4 La surprise gringo (ne résout pas tout)

Afin de voir comment limiter l’explosion en taille, lançons gringo (“instantiateur” sans le “solveur” clasp), avec `step(1)` seul (sans intervalle). Cela donne le résultat (en 96 s sur l’ordinateur utilisé, mais l’essentiel du temps est sans doute utilisé pour écrire le fichier résultat car le filtrage ne fonctionne pas avec gringo). Cela donne donc le résultat cherché directement, sans utiliser d’unité interne ni d’encadrement a priori (mais en utilisant les hypothèses 1 – 5 ci-dessus). L’ennui est que clingo (gringo + clasp) ne sait pas que gringo seul suffirait ici à donner le résultat. Et il semble (d’après Torsten Schaub) qu’il ne soit pas si facile que cela de faire que clingo s’en aperçoive assez tôt pour éviter d’utiliser de la mémoire de façon inutile ici, et catastrophique (gringo pourrait-il s’en apercevoir ?). On se situe donc ici dans une marge où gringo seul passe et pas clingo, mais des tests avec l’hypothèse faible suggèrent que cette marge n’est pas si étroite que cela, et que donc détecter ces cas augmenterait les possibilités de clingo.

4 Conclusion

L'état actuel des systèmes ASP permet d'envisager à court terme une réelle utilisation pour des problèmes de planification de difficulté moyenne. Il s'agit d'un formalisme 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 qui sont elles aussi aisément traduisibles en ASP. Pour une utilisation plus aisée, les systèmes actuel 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 la version "usuelle" avec un essai par ensemble réponse, le fait pour l'utilisateur de préciser que `loadD` doit satisfaire l'axiome du cadre est susceptible de faciliter l'écriture du programme, mais surtout l'efficacité du calcul : des méthodes spécifiques permettraient d'épargner de la mémoire. Un méta prédicat du genre `#cadre` où `#cadre loadD/1/1/1.` signifierait que `loadD` doit être traité comme satisfaisant l'axiome du cadre : le premier paramètre étant le temporel, le second l'indice et le troisième la valeur soumise à l'axiome du cadre, devant rester identique, pour chaque indice, sauf si une règle demande de la modifier. On ne peut pas aller trop loin dans cette direction car on reviendrait 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. On peut aussi envisager soit un méta prédicat `#mvt` signalant ici que `move` doit être traité de façon particulière, ou mieux (si possible)une détection automatique de ce genre de situation. Il semble que de telles petites évolutions des systèmes actuels, orientées confort de l'utilisateur, amélioreraient sensiblement leurs performances effectives (et non pas théoriques).

Références

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

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

DENECKER M., VENNEKENS J., BOND S., GEBSER M. & TRUSZCZYŃSKI M. (2009). The 2nd Answer Set Programming Competition. LPNMR'09 p. 637–654.

GEBSER M., KAUFMANN B. & SCHAUB T. (2009). The conflict-driven answer set solver *clasp* : Progress report. LPNMR'09, p. 509–514.

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

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