Pont miné, vieil éléphant mangeur de bananes, et utilisation de la programmation par ensembles réponses (Answer Set Programming)

Yves Moinard

Pont miné, vieil éléphant mangeur de bananes, et utilisation de la programmation par ensembles réponses (Answer Set Programming)

Yves Moinard

INRIA Bretagne-Atlantique Rennes

15 juin 2012, IRIT, Toulouse

Programmation par ensembles-réponses (ASP pour "Answer Set Programming")

- **Programmation déclarative**
  On décrit un problème, et non plus comment le résoudre.
- **Programmation logique**
  Algorithme = Logique + Contrôle [Kowalski] (Prolog). La logique du premier ordre considérée comme un langage de programmation.
- **Description du problème:**
  "règles" dans un langage logique simple et naturel à la Datalog (proche du propositionnel: pas de $\forall$ ni de $\exists$ explicites).
- **Schisme ASP:** Les modèles, et plus les preuves, sont les solutions.
  Théorie: ICLP 1988 Seattle, papier de Gelfond/Lisfchitz.
  Terme "ASP" introduit par Lifschitz vers 1999.
  Plus proche de la logique classique que Prolog (l'ordre des prédicats n'intervient pas, pas de "cut",...).
  Modèle (ensemble réponse): ensemble d'atomes satisfaisant l'énoncé.

ASP quelques systèmes actuels

- ***Smodels et LParse*** (Ilkka Niemelä, Patrik Simons, Tommi Syrjänen, NMR2000) le premier système répandu (toujours vivant!)
- ***DLV*** Nicola Leone, Gerald Pfeifer, Wolfgang Faber, F. Calimeri, Giovambatista Ianni, Thomas Eiter, Georg Gottlob, Simona Perri, Francesco Scarcello... ***DLVSystem, et tj: DLV-Complex, DLT (Templates)***
- ***Gringo, clasp, claspD, clingo,***... (Potassco team, Postdam) Torsten Schaub, Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Sven Thiele... Amélioration de SModels. Évolution rapide (gringo 3, juillet 2010, un livre va sortir).
- ***CModels*** Yulia Lierler et autres du Texas Action Group at Austin (Vladimir Lifschitz),
- ***ASPeRix*** (Le petit Gaulois Angevin) Initié par Pascal Nicolas. Claire Lefèvre, Stéphane Ngoma.

ASP Intro (suite)

The Second Answer Set Programming Competition (Int. Conf. on Logic Programming and Nonmonotonic Reasoning 2009, Leuven, Belgique).
"The Potassco [T. Schaub et al., Postdam] team is the clear winner"
"The goal of declarative problem solving is to minimize the effort of programmers"
Third (Open) Answer Set Programming Competition -May 16th 2011 (yc SAT)
La partie concernant +/- notre sujet semble être "Model and Solve Competition" :

System	Total	Instance	Time	
clasp	2431	1432	999	potassco
aclasp	1953	1140	813	id.: variant de clasp
bpsolver	1878	1064	814	B-Prolog: constraint logic progr.
ezcsp	1773	993	780	integration of ASP yc clasp & CP
idp	1442	918	524	ID: "Inductive Definition"
fastdownward	367	217	150	a planning system, # config.

Règle d'un programme logique

$$\mathit{Tête} \leftarrow \mathit{Corps}. \tag{1}$$

où *Tête* et *Corps* sont des ensembles finis d'**éléments de règle**.

Élément de règle:

1. ***atome*** a, propositionnel ou du premier ordre $p(t_1, \cdots, t_n)$, ou
2. ***littéral classique*** l (a ou $\neg a$), (***négation forte***, ou ***"vraie nég."***), ou
3. *not* l avec l littéral classique (*not* : ***négation par défaut*** ou ***par l'échec***).

terme t_i: symbole de constante ou de variable, ou fonctionnel.

Les systèmes admettent maintenant tous des symboles de fonction, avec restrictions (domaine fini).

Par contre ils interdisent souvent *not* en tête, et parfois "ou" en tête.

Souvent, "$\leftarrow$" noté ":-" et "$\neg$" noté "-" (frappe au clavier).

Règle d'un programme logique et ensemble réponse

(1) Instantiation (rôle de l'*instantiateur*, lequel passera le résultat à un *[ré]solveur*.

• Règle avec variables $\rightsquigarrow$ l'ensemble de toutes ses instances concrètes

c'est (entre autres) la tâche de l'*instantiateur* ou *grounder*: `LParse` ou `gringo` pour les *[ré]solveurs* Smodels ou clasp (clingo est gringo+clasp).

Exemple (*Les noms de variables commencent par une Majuscule*.):

r: $p(X) \leftarrow q(Y), s(X, Y).$ f1: $q(a).$ f2: $q(b).$ f3: $s(a, b).$ devient

r1: $p(a) \leftarrow q(a), s(a, a).$ r2: $p(a) \leftarrow q(b), s(a, b).$

r3: $p(b) \leftarrow q(a), s(b, a).$ r4: $p(b) \leftarrow q(b), s(b, b).$ (plus f1, f2, f3)

Règle d'un programme logique et ensemble réponse
(2) ensemble réponse

- Pour un programme sans *not*, un ***ensemble réponse*** est un ensemble consistant de littéraux instantiés qui
  1. "satisfait toutes les règles" : précisément, satisfait les formules (cf ci-dessous) comme $(q(b) \wedge s(a,b)) \longrightarrow p(a)$ (issue de ▸ (r2))
  2. et est ***mimimal pour $\subseteq$ avec cette propriété***.

Ici un seul ***ensemble réponse*** : $\{q(a),\ q(b),\ s(a,b),\ p(a)\}$

Transformer une **règle** (comme r2) en ***formule logique*** :
1. la retourner ($\longleftarrow \rightsquigarrow \longrightarrow$),
2. "," d'un **Corps** $\rightsquigarrow \wedge$ et "," d'une **Tête** $\rightsquigarrow \vee$.

Ensemble réponse d'un programme logique

Programme quelconque Π (avec *not*):

- Soit E un ensemble consistant de littéraux.
  On ne conserve que les règles (r) de Π pour lesquelles les littéraux l_i de E rendent les éléments *not* l_i de (r) superflus,
  c'est-à-dire les règles telles que E
  1. ne contient aucun littéral l_i tel que le corps de (r) contient *not* l_i,
  2. et contient tous les littéraux l_i dont la tête de (r) contient *not* l_i.
- ***Reduct*** Π^E: toutes les règles qui restent, dans lesquelles on a ôté tous les éléments *not* l_i. Cela donne un programme sans *not*.
- E ***ensemble réponse*** de Π si E ensemble réponse de Π^E.

Exemples d'ensembles réponse d'un programme logique

N'est vrai que ce qui doit l'être, tout doit être justifié.

- Π: r1: $p, q.$ r2: $\neg s \leftarrow p.$
  (r1, tête sans corps, s'écrit aussi
  "$p, q \leftarrow .$" *ou* "$p; q \leftarrow .$" *ou* "$p \vee q \leftarrow .$" *ou* ...)
  Π a deux AS (ensembles réponse / "answer sets"): $\{p, \neg s\}$ et $\{q\}$.
  (Le $\vee$ des *Tête* a tendance à devenir exclusif par minimisation.)

- On ajoute la ***contrainte*** (règle sans tête)
  c1 : $\leftarrow q.$ (interdit q), seul reste le 1^{er} AS $\{p, \neg s\}$.
  [c1 équivaut à *not* $q.$, aussi écrit *not* $q \leftarrow.$]

- Au lieu de c1 on ajoute le ***fait*** (règle où tête = singleton sans *not* et sans corps) f1 : $\neg q.$ ($\neg q$ doit être satisfait),
  le seul AS est $\{p, \neg q, \neg s\}$.

Exemples d'ensembles réponse d'un programme logique (2)

- Π: r4: $p \leftarrow not\ q.$ r5: $q \leftarrow not\ s.$ r6: $s \leftarrow not\ t.$

  Si $E = \{p, s\}$, AS de $\Pi^E = \{p., s.\}$ donc de E (le seul).

- Π: r8 : *heureux* $\leftarrow$ *not triste*. r9 : *triste* $\leftarrow$ *not couci-couça*.
  r10 : *couci-couça* $\leftarrow$ *not heureux*. ***Aucun AS***

- Π: r7 $= 0\ \{s1, s2\}\ 2.$ (***règle particulière: génération indéterministe;*** *tente de prendre entre* 0 *et* 2 *éléments de l'ensemble dans chaque Answer set.*)

  $F0 = \emptyset$, $F1 = \{s1\}$, $F2 = \{s2\}$ et $F3 = \{s1, s2\}$:
  $\Pi^{\emptyset} = \emptyset$, $\Pi^{\{s1\}} = \{s1.\}$, ...

  les *Fi* sont les 4 AS de Π.

Ensembles réponse d'un programme logique (intuitions)

- N'est vrai que ce qui a une "raison forte" de l'être.
- Les "si" ($\leftarrow$) tendent à être transformés en "ssi" ($\leftrightarrow$).
- *not a*. Dans un corps: pas de raison de croire (d'avoir) *a*.
  Dans une tête: ne pas avoir *a* (comme la contrainte $\leftarrow a.$)

Règles admises dans les systèmes ASP (et un peu d'histoire: gringo 3 été 2010)

Il existe quelques restrictions à l'écriture des règles.

Safe rules ($\pm$ avec tous les systèmes): Les variables apparaissant dans une règle doivent toutes apparaître dans un terme sans *not* du corps de la règle.

```
p(X,Y) :- q(X).      p(X,Y) :- q(X), not r(X,Y).      Non admis
q(X,Y) :- r(X), q(X,Y).      q(Y)= :- p(X,Y).      Admis
```

Avec clingo, ces termes devaient concerner des prédicats définis assez directement (sans récursivité,...) ***domain predicates***. Cela compliquait beaucoup l'écriture des programmes clingo/clasp avant gringo3 (été 2010).

DLV était plus facile d'utilisation. Sauf cas particulier (exponentiation), cela n'interdisait pas d'écrire un programme, mais compliquait la tâche. (Ces "domain predicates" sont encore parfois utiles, voire nécessaires.)

Quelques fonctionnalités et limites des systèmes actuels

- Il arrive qu'un programme logiquement correct et à domaine fini déborde car l'instantiateur ne tient pas compte du fait qu'une règle permettrait d'arrêter l'instantiation assez tôt. Parfois, ajouter des "domain predicates" ou des `X <= Y` dans le corps des règles suffit...

- Arithmétique: DLV impose de fixer a priori une limite aux entiers considérés (pas clingo depuis gringo3, même s'il peut être utile de fixer une limite).
  Les réels ne sont pas directement admis.

- Raisonnement ***prudent (cautious)*** et ***brave***.
  Des options de lancement permettent de donner l'intersection de tous les ensembles réponses ou leur réunion. Il y a des restrictions, en particulier DLV n'admet cela que si on pose une ***question*** (requête) précise et assez simple.

Quelques fonctionnalités et limites des systèmes actuels

- On peut préciser si on veut un, tous, ou un certain nombre, d'ensembles réponses (clingo: 1 par défaut, 0 pour tous).
- Méta-prédicats `#max`, `#min`, `#sum`, `...` sur des multi ensembles valués (et même avec "niveaux de priorité") (clingo, et DLV). Permettent d'écrire des ***contraintes de cardinalité*** similaires aux ***contraintes faibles*** de DLV.
- ...

Quelques variantes autour de DLV

- DLV-Complex: version plus "conviviale" de DLV, avec ensembles, suites, diverses fonctionnalités,...
- DLT: extension de DLV qui admet des ***templates*** qui sont une manière naturelle de régler en grande partie un des inconvénients des systèmes actuels: l'absence de "sous-programmes". Ces templates facilitent beaucoup écriture et relecture. Aux dernières nouvelles, DLV-Complex et DLT sont incompatibles, et pas incorporés au "DLV System" officiel (à vérifier).

Quelques variantes autour de DLV: "template" DLT)

Ces templates sont très pratiques et ont une syntaxe simple:

- Données: `person(mary,f,29).` `person(mark,m,25).` `person(sarah,f,26).` `person(frank,m,28).`
- Définition d'un "template":

```
#template max{formal(1)}(1)
      { max(X) :- formal(X), not overcome(X).
        overcome(X) :- formal(X), formal(Y), Y>X.}
```

- Invocation de ce template:

```
older(Name,Sex,Age) :- max{person($,$,*)}(Age),
                       person(Name,Sex,Age).
```

  Donne nom, sexe et âge de la personne la plus vieille.
  Deux types de paramètres: $ pour les "auxiliaires", et
  * pour le(s) "vrai paramètre" du template défini ici.

- Résultat: `older(mary,f,29)`

Quelques variantes autour de clasp

- claspD: admet disjonctions en tête de règle (comme DLV et CModels) mais n'est plus à jour),
- iclingo: ***("i"ncrémental)*** Incrémenter un paramètre jusqu'à ce qu'on obtienne le résultat, augmente les interactions gringo $\leftrightarrow$clasp. Permet parfois d'écrire des programmes plus efficaces (parties ***static*** et ***cumula tive***, difficiles à écrire et à lire...),
- clasPar: Calcul parallèle, se veut une approche simple et transparente à l'utilisation de la puissance de clasp en calcul distribué.
- clingo (gringo + clasp), ClingCon (constraint processing), Oclingo (on line),..., claspfolio (choisit automatiquement une ***bonne configuration*** de clasp),...

Le pont miné: énoncé

- On examine maintenant la facilité avec laquelle on peut traduire de petits problèmes en un programme exécutable en programmation par ensembles-réponse.
- Deux exemples: un premier où tout va pour le mieux, un second plus coriace où des hypothèses additionnelles sont indispensables.
- Voici d'abord l'énoncé du premier:

> *Quatre soldats doivent franchir de nuit un pont miné (il sautera* ***une heure*** *après que le premier soldat aura mis le pied sur le pont). Les soldats sont blessés plus ou moins gravement ce qui explique leur durée respective de franchissement du pont en minutes: 25, 20, 10 et 5. Le pont ne peut supporter que* ***deux soldats*** *à la fois et il n'y a qu'****une lampe****, laquelle est indispensable pour traverser le pont. Comment font-ils?*

Le pont miné: Le programme (1)

Voici ce petit programme (traduction quasi littérale de l'énoncé):

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

   `soldat(0..3)` signifie `soldat(0),...,soldat(3)`

2. Les 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 traverse vers L (atteint en M+1).
   - ***duree(M,T)***: La traversée en M prend un temps T.

3. Initialisation: 4 soldats et lampe au départ au temps (ou étape) 1
   traversée effectuée de durée nulle au temps 0.

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

Le pont miné: programme, génération des déplacements

Expression en langage naturel (très proche de l'énoncé):

- Un soldat ne traverse que s'il part du côté où est la lampe,
- traverser tant que tout le monde n'est pas de l'autre côté,
- ne plus traverser après "tmax" (explosion).

Traduction ASP: Traversée de *L0* vers *L* (chaque soldat éligible peut traverser, ou pas) une règle de génération ▸ indéterministe

4.
```
{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).
```

(`not goal(M-1)` est facultatif, voir ▸ ci-dessous pour `goal`.)

Le pont miné: programme, description de déplacement (1)

5. Ceux qui se sont déplacés vers L arrivent en L:

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

6. Un ***Axiome du cadre*** traduit ce qui n'est pas modifié (ici, ceux qui ne se sont pas déplacés):

```
at(M+1,L,S) :- move(M,L0,S1), at(M,L,S), lieu(L2),
               not move(M,L2,S).
```

Le pont miné: programme, description de déplacement (2)

7. Au plus 2 soldats traversent

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

[Notation gringo ";" (pooling): `soldat(S1),..., soldat(S3);`
Remarque: "<" au lieu de "!=" ($\neq$) limite l'instantiation!]

8. 1 soldat suffit pour porter la lampe:

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

9. Complication (provisoire...) pour calculer la durée des traversées effectuées (fonction `#max` pas admise ici avec l'actuel clingo):

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

Le pont miné: programme, atteindre le but

Comme souvent en ASP dans un problème de planification:

- on décrit d'abord le but `goal` (10),
- puis une contrainte qui élimine les modèles ne le satisfaisant pas. (11)

Ici, on décrit ensuite la durée de chaque trajet solution trouvé. (12)

10.
```
goal(M-1) :- at(M,arriv,0), at(M,arriv,1),
             at(M,arriv,2), at(M,arriv,3).
goal0 :- goal(M).
```

11.
```
:- not goal0.
```

12.
```
duree(T) :- goal(M), duree(M,T).
```

Le pont miné: programme (fin)

Voici deux manières de terminer ce programme:

End1 ***Optimisation*** ne garder que les modèles minimisant le temps de trajet:

```
#minimize [ duree(T)= T].
```

End2 Contrainte éliminant les modèles ne satisfaisant pas l'énoncé:

```
:- duree(T), T > tmax.
```

Les deux fournissent le même résultat (car l'énoncé donne le meilleur temps possible).
Utiliser l'optimisation de clingo accélère un peu (0.110 s. avec, 0.180 s sans), ce qui montre qu'elle est assez bien implémentée;
et surtout, cela marche aussi avec un énoncé moins coopératif.

Le pont miné: commentaires

Il s'agit d'un cas idéal: la combinatoire est faible, et on peut se permettre de transcrire presque littéralement l'énoncé. Le programme est court et lisible, donc facile à écrire et surtout à ***relire*** et ***modifier*** si nécessaire.
Il est facile d'y introduire des accélérations.
Notons (par rapport à un simple datalog qui ne connaît pas la négation par défaut) que l'usage du `not` dans l'axiome du cadre ▸ simplifie l'écriture.

C'est donc un exemple de rêve, même si bien sûr il serait facile à écrire dans tout langage. L'avantage est qu'ici on est proche d'un langage naturel. Nul besoin en particulier d'utiliser un langage orienté planification.

La méthode utilisée (préconisée dans ce genre de situation pour ASP), donne un trajet par ensemble réponse.

L'exemple suivant est beaucoup moins directement traduisible.

Le vieil éléphant et le planteur de bananes

> *Un planteur a produit* ***3 000 bananes****. Comme moyen de transport, 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 marché?*

L'espace des solutions envisageables sans réflexion est grand: Il faut réfléchir au problème avant de programmer sous peine de crash (`bad_alloc`).

Un avantage d'ASP apparaît ici: il est facile d'introduire le fruit de telles réflexions de façon naturelle dans le programme.

Par contre, les limites de l'aspect déclaratif apparaissent aussi: les problèmes de temps et surtout taille mémoire obligent à examiner (un peu trop?) comment calculer.

L'éléphant et les bananes: hypothèses pour élagage

Méthode générale nécessitant peu de réflexion particulière:

1. Changer d'unité: prendre les bananes (ou km) par 100, 33, 10.....
   Fournit une solution, dont les distances entre dépôts intermédiaires.
   Diminuer alors la taille de l'unité interne et encadrer les distances possibles entre dépôts en s'inspirant de cette réponse, etc..

Ce qui suit nécessite d'étudier plus en détail le problème:

2. - [H2dep] Imposer le nombre de dépôts (2 intermédiaires) et d'étapes (9),
   - [HStop] se limiter aux trajets qui s'arrêtent dès le marché atteint,
   - [HDCr] se limiter aux distances entre dépôts croissantes, ou
     [HDCr] aux distances croissantes sauf éventuellement la dernière.
3. [HAIMax] Charge maximale aux parcours aller, avec deux variantes:
   [HAIMaxtj] charge maximale autorisée toujours (variante forte), ou
   [HAIMaxPoss] (variante faible) charge maximale si possible,
   sinon ce qui est disponible, si suffit pour le parcours.
4. [HRet0] Revenir vide pour les retours (évident).

L'éléphant et les bananes: types de trajets

On peut envisager plusieurs trajets possibles, en utilisant des dépôts intermédiaires $\mathbf{1}, \mathbf{2}, \cdots, \mathbf{d}$:

Trajet **incrémental**: Si l'éléphant arrive pour la première fois à un dépôt, il retourne au départ (s'il faut refaire des voyages): trajet en dents de scie de + en + grandes (sauf éventuellement la dernière), à creux alignés.

Trajet **dépôt par dépôt**: L'éléphant effectue N aller retours du dépôt $\mathbf{i}$ au dépôt $\mathbf{i+1}$ puis un dernier aller simple (trajet partiel de longueur $2N-1$ de $\mathbf{i}$ vers $\mathbf{i+1}$). Cela donne un trajet avec séries de dents de scies égales, avec décrochements successifs, et légère augmentation de la taille à chaque décrochement (sauf éventuellement au dernier).
Dans chaque cas, la distance parcourue est la même.

[HTrajI] On se limite aux trajets incrémentaux. ou bien
[HTrajD] On se limite aux trajets dépôt par dépôt.

L'éléphant et les bananes: trajet dépôt par dépôt

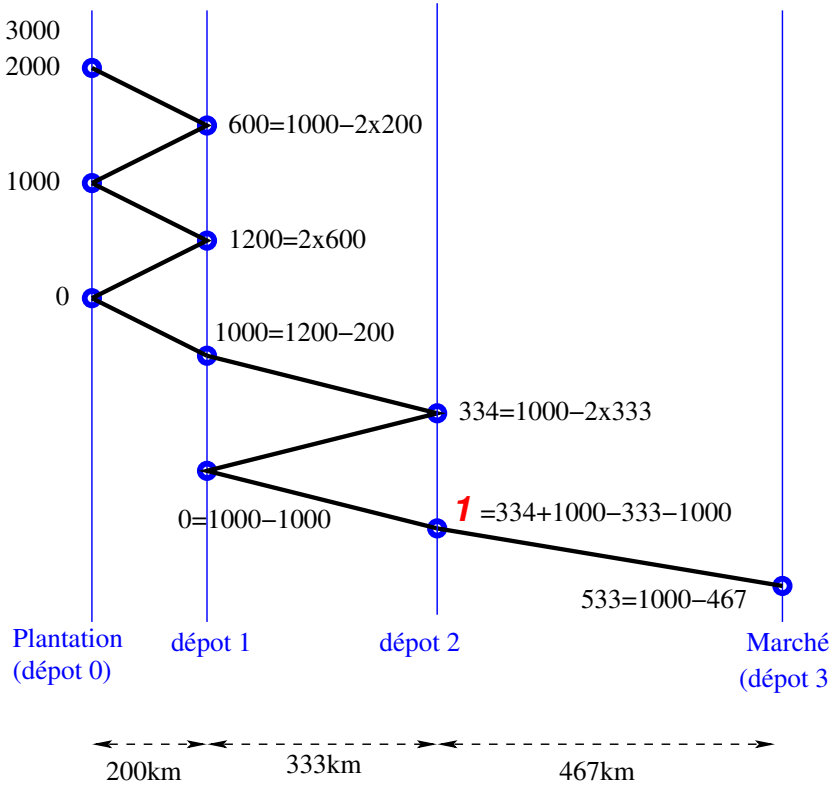

Figure: Trajet ***dépôt par dépôt***

3 départs de 0 vers 1 apportent $3x1000-(2x3-1)x200 = 2000$ bananes en 1.

2 départs de 1 vers 2 apportent $2x1000-(2x2-1)x333 = 2000-3x333 = 1001$ bananes en 2.

1 banane abandonnée en 2,
1 départ de 2 vers 3 apporte $1x1000-(2x1-1)x467 =$ ***533 bananes*** en 3.
2466 km ***1 banane*** abandonnée

L'éléphant et les bananes: trajet incrémental

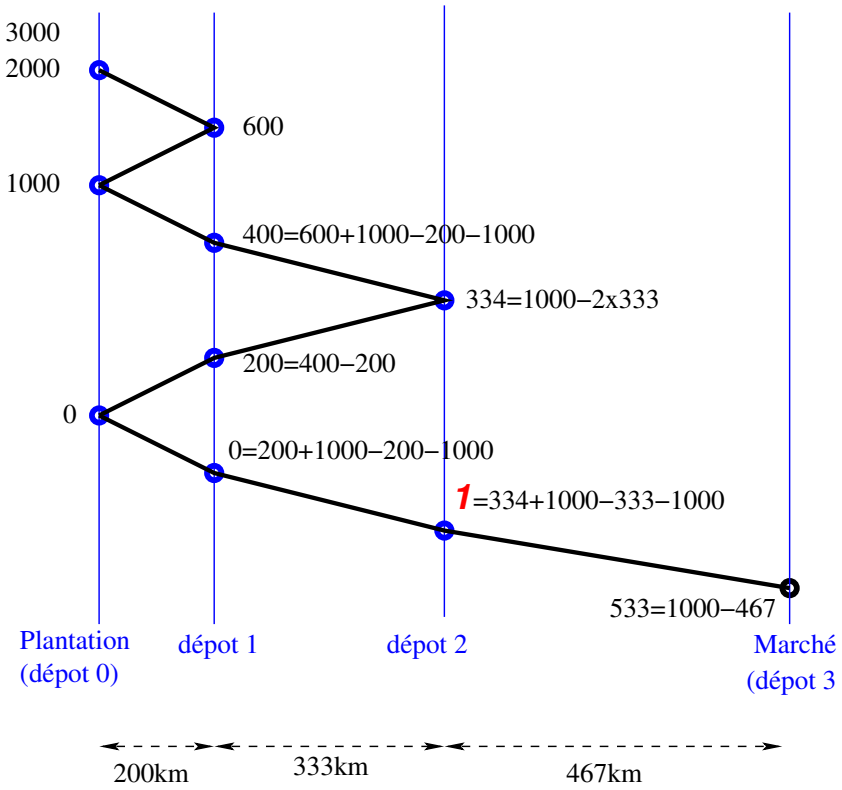

Figure: Trajet ***incrémental***

Le nombre de bananes apportées au marché est le même:
On a encore 3 départs à pleine charge de 0 vers 1, 2 de 1 vers 2 et 1 de 2 vers 3, la distance totale parcourue est la même $2466 =$
$5x200 + 3x333 + 1x467$
et on perd là aussi **1** banane, apportant ***533 bananes*** en 3 et parcourant ***2466 km***

L'éléphant et les bananes: hypothèses pertinentes

Il semble que les ***hypothèses bleues*** [HDCr], [HAIMaxPoss], [HRet0] et [HTrajI] ou [HTrajD] forment un ensemble ***pertinent***.
Hypothèses vertes là: que certaines données dont (3000,1000,1000)).
Mais je laisse les démonstrations à d'autres...

Un ensemble d'hypothèses est

- ***pertinent*** si tout trajet peut être transformé en un trajet qui respecte les hypothèses et apporte au moins autant de bananes au marché;
- ***potentiellement pertinent*** s'il est pertinent quand il existe au moins un trajet qui le satisfait
  Peut traiter des données numériques légèrement différentes: essai avec hypothèses les plus fortes, et si aucune solution, affaiblir l'ensemble jusqu'à ce qu'il devienne pertinent.

Ces hypothèses réduisent considérablement la taille de l'espace de recherche, et sont immédiates à transcrire en ASP.

Programme avec un seul ensemble réponse

Malgré ces méthodes (granularité variable et hypothèses pertinentes), je n'ai pas trouvé de programme qui donne la solution de façon satisfaisante (sans trop d'unités successives et sans trop restreindre l'encadrement des distances)
en utilisant la méthode classique du pont
(programme aisé à lire, une solution par ensemble-réponse).

Autre méthode, plus lourde à écrire:
Au lieu de construire ***un ensemble réponse par trajet possible***, tous les trajets possibles (satisfaisant les hypothèses) sont calculés ***dans un seul gros ensemble réponse***. Il faut alors indexer ces trajets pour les identifier,
ce qui est fait par une suite comprenant:

- Le nombre d'étapes déjà effectuées,
- les deux premières distances entre dépôts,
- la suite des couples (dépôt visité, charge de l'éléphant quittant ce dépôt).

Cela a fini par marcher ($\pm$).

L'éléphant et les bananes: un programme (1)

Extrait: une règle de construction d'un trajet (▸ incrémental, détail)
(cf ▸ pont ▸ miné):

Poursuite du trajet: cas d'un parcours "retour", de $D0$ à $D0-1$:

```
i(dd01(DD0,DD1),S+1,m(m(m(M,D,LE),D0,LE0),D0-1,DP),l(D0,LD))
:- i(dd01(DD0,DD1),S,m(m(M,D,LE),D0,LE0),l(D0,LD0)),
   S < stagemax, dd(dd01(DD0,DD1),D0,D0-1,DP),
   dd(dd01(DD0,DD1),D,D0,DP0), LD = LD0+LE0-DP0-DP,
   LD >= 0.
```

*dd*01, *m* et *l*: symboles de fonctions.

1. `dd01(DD0,DD1)`: `DDi` distance entre dépôts `i` et `i+1` (`dd01` fac.).
2. `S`: Numéro de l'étape (redondant, facilite des calculs).
3. `m(M,D,LE)`: Un mouvement (récursivité : M=mouvement précédent).
4. `l(D0,LD)`: La charge du dépôt D0 est LD à l'arrivée en D0-1.

L'éléphant et les bananes: un programme (2)

Mouvement représenté par symbole de fonction `m` (décrit en un seul terme),
et non plus par prédicat `move` (décrit 1 étape, 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 `M0` précédent
[donc `M` est `m(Ma,D0,LE0)`, `D0` étant le dépôt quitté, avec la charge `LE0`, `Ma` étant le mouvement d'avant].

Écriture des règles plus compliquée, aspect déclaratif moins net...

L'éléphant et les bananes: un programme (3)

Le parcours aller est similaire avec l'hypothèse forte [HAIMaxtj] (charge maximale possible à chaque parcours aller), suffisante pour l'énoncé donné. Deux règles sont nécessaires pour l'hypothèse faible (charge maximale si possible, et sinon tout ce qui est disponible, nécessaire avec 2900 bananes au lieu de 3000).

Charge `LEf` apportée au marché (dépôt `ndep`) (fonction `loadA` fac.):

```
iti(DD0,DD1,stagemax+1,lastload(leu),loadA(LEf),
    i(DD0,DD1,stagemax,m(M2,ndep,LE),l(ndep-1,LD2))) :-

        i(DD0,DD1,stagemax,m(M2,ndep,LE),l(ndep-1,LD2)),
        dd012(DD0,DD1,DD2), LEf = LE - DD2.
```

L'éléphant et les bananes: un programme (4)

Une fois tous les trajets satisfaisant les hypothèses calculés, calcul de la charge maximale:

```
maxloadA(A) :- A = #max [iti(DD0,DD1,stagemax1,
                Lastload,LoadAU,loadA(LEfb),I) = LEfb].
```

Différence avec End1 ▸ le pont miné: on n'utilise plus un

- **méta-prédicat d'optimisation** `#minimize`, qui cherche le minimum parmi **tous les ensembles réponses**, mais un
- méta-prédicat qui calcule le **maximum d'une valeur à l'intérieur d'un même ensemble-réponse**, `#max`.

On profite ici une première fois (la seconde fois étant qu'on a maintenant une solution!) de la simplicité structurelle du présent programme, car le `#max` de gringo n'aime (pour l'instant) pas la récursivité dans les programmes de structure moins simple (comme le pont miné où `#max` n'était pas admis).

L'éléphant et les bananes: un programme (5)

Définition des trajets optimaux satisfaisant les hypothèses:

```
itiopt(DD0,DD1,stagemax1,Lastload,LoadAU,loadA(LEfb),I) :-
      maxloadA(LEfb),
      iti(DD0,DD1,stagemax1,Lastload,LoadAU,loadA(LEfb),I).
```

Il est pratique de mettre en évidence les distances entre dépôts, en km:

```
dd012otpkm(DD0 * step, DD1 * step, DD2 * step) :-
    maxloadA(LEfb),
    itiopt(DD0,DD1,stagemax1,Lastload,LoadAU,loadA(LEfb),I),
    dd012(DD0,DD1,DD2).
```

L'éléphant et les bananes: un programme (6)

Exemple de réponse (en 3s avec clingo) en fournissant un encadrement des distances entre dépôts (`step(1)` avec les données utilisateur, ▸ cf suite):

`dd2valkm(0,170,230) dd2valkm(1,300,360)`

`maxloadA(533)` deux trajets optimaux (incrémentaux, rappel):

```
itiopt(200,333,10,lastload(1000),loadAU(533),loadA(533),
   i(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)))
```

```
itiopt(200,333,10,lastload(1000),loadAU(533),loadA(533),
   i(200,333,9,m(m(m(m(m(m(m(m(m(m(m0,0,0),1,1000),0,200),
     1,1000),2,1000),1,333),0,200),1,1000),2,1000),3,1000),
   l(2,1)))
```

`dd012otpkm(200,333,467)`

L'éléphant et les bananes: épilogue (provisoire)

Clingo résout le problème en quelques secondes, mais avec deux lancements: un premier avec `step =10` (bananes ou km) et sans autre restriction donne (en 0.8s avec l'hypothèse forte ▸ [HAIMaxtj] 3 adaptée ici, en 7.4 s – temps donnés pour ordre de grandeur – avec l'hypothèse faible de charge maxi ou dispo):

```
Answer:  1 dd012otpkm(200,330,470) maxloadA(530)
```

Les distances entre les premiers dépôts sont 200 et 330 km, ce qui a conduit à proposer, lors du second lancement avec `step=1` et l'hypothèse forte, les deux encadrements (à $\pm$ 30 km) `dd2valkm` ▸ données ci-dessus: `dd2valkm(0,170,230) dd2valkm(1,300,360)`.

L'éléphant et les bananes: la surprise gringo

Clingo arrive à résoudre le problème avec un peu de complications et environ 1s en tout (disons 30s avec les petites manip!). Mais il ne résout pas le problème, même avec l'hypothèse forte, avec un seul lancement sans idée préalable du résultat (distances entre dépôts): crash `bad_alloc`! [C'est sans doute (?) possible, mais plus compliqué à écrire...]

Pour voir précisément ce qui explosait, j'ai lancé gringo seul, l'instantiateur. Comme on ne peut pas filtrer le résultat, le fichier résultat est énorme (570 MO) et nécessite "grep" pour le lire, à part cela c'est assez rapide (96 s sur l'ordi testé, l'essentiel semble occupé à écrire le résultat, en majeure partie inutile...):

Et j'ai eu la surprise de constater que gringo 3 seul fournit le résultat...

Le "solveur" clasp ne sert ici qu'à encombrer inutilement la mémoire, mais clingo (gringo 3 + clasp) ne s'en aperçoit pas. sauter autres systèmes

L'éléphant et les bananes: autre systèmes

J'ai aussi testé DLV-Complex et Asperix (petites modif. du programme).

DLV-Complex est sensiblement plus lent, ne crashe pas aussi vite, mais je n'ai pas trouvé d'exemple où il aille plus loin que clingo (une bonne nuit d'attente et cela tournait tj...avec clingo, on est plus vite fixé).

Asperix est assez sensiblement plus lent (sa méthode différente, qui ne sépare pas instantiateur et solveur, ne semble pas l'avantager ici).

Je n'ai pas testé la version avec `move` et un AS par trajet (la méthode utilisée pour le pont miné) avec d'autres que clingo, je crains que cela ne serve à rien.

L'éléphant et les bananes: hypothèse [HalMaxRec]

Un étudiant de Marie-Odile Cordier (Mayeul De Werbier d'Antigneul) a écrit en OCaml un programme très efficace, incompréhensible pour moi, donc me semblant miraculeux...

L'astuce réside dans l'utilisation de ▸ [HTrajD] ▸ (Fig) et d'une généralisation (par récursivité) de ▸ [HAlMax]:

La consommation ne dépend pas de la charge, il vaut donc mieux s'arranger pour transporter la valeur de la charge d'un dépôt au suivant: on fait une "extension récursive" [HAlMaxRec] de [HAlMax]: On part d'un trajet optimal avec $N \times ChMax$, et on essaie $(N+1) \times ChMax$. On ajoute un dépôt N°1 (1^{er} dépôt intermédiaire), à la distance $DD0$ de 0, et on y apporte $N \times ChMAx$, ce qui consomme $ChMax$. Il y avait $(N+1) \times ChMax$ bananes, cela fait $N+1$ départs de 0 vers 1, avec N retours, donc $DD0 = ChMax/(2 \times N+1)$. On calculera toutes les distances successives entre dépôts. Finie l'explosion combinatoire (ou presque...)

L'éléphant et les bananes sans explosion combinatoire
Hypothèse très forte sur les charges max (donc distances)

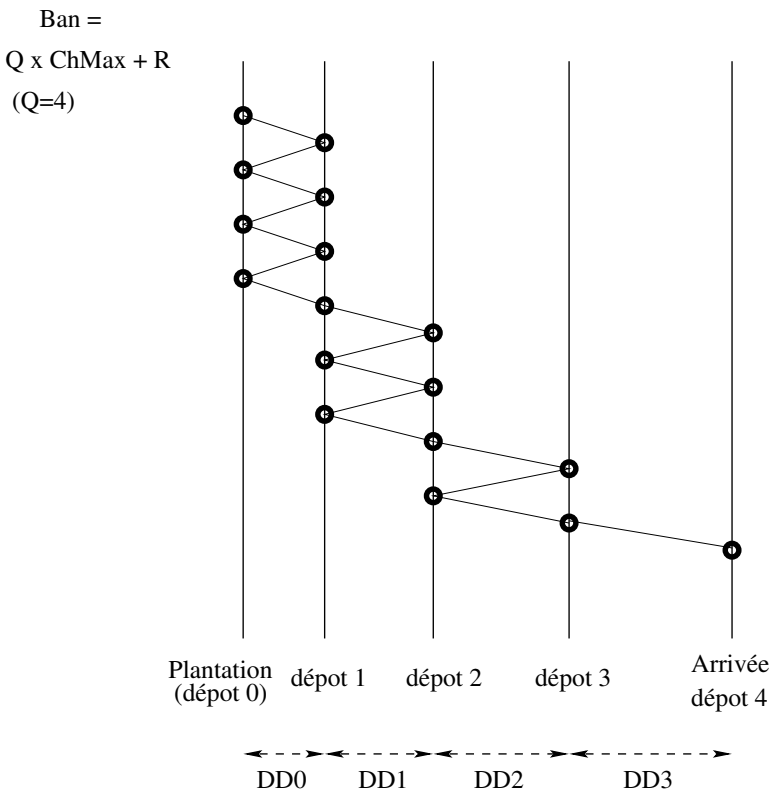

Figure: Hypothèse charge max très forte

Données:
Ban (quantité initiale),
ChMax (charge maximale) et
Dist (distance totale).

Division euclidienne $Ban = Q \times Chmax + R \ (\leq Chmax)$.

On en déduit les distances entre dépôts (quotients entiers):

départs $i \rightsquigarrow i+1$	DDi
Q	$ChMax/(2Q-1)$
$Q-1$	$ChMax/(2Q-3)$
$Q-2$	$ChMax/(2Q-5)$
$Q-3$	$ChMax/(2Q-7)$

L'éléphant et les bananes: reste initial ($1^{ère}$ méthode)

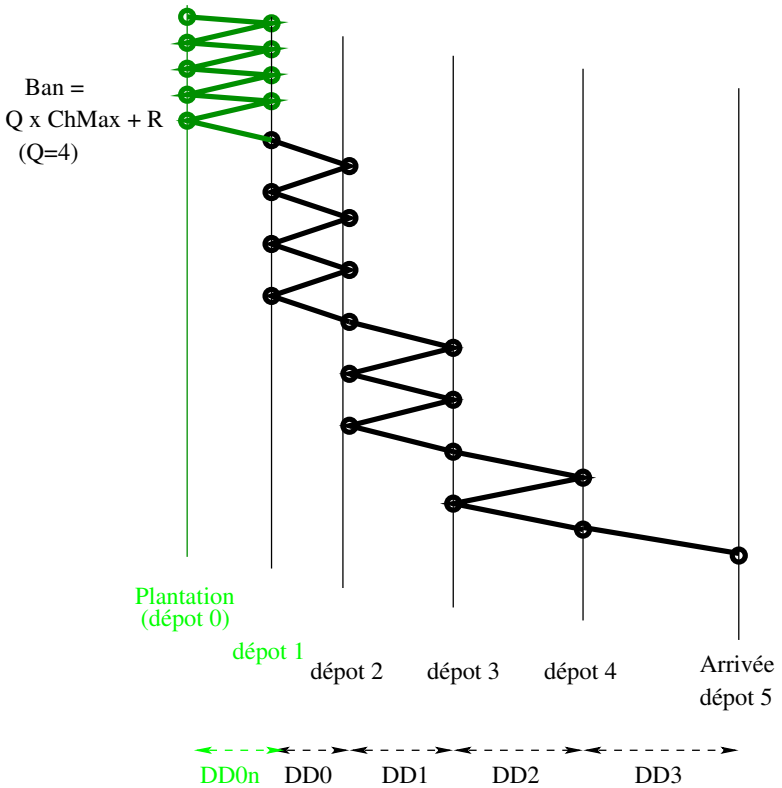

Figure: Problème du reste

Si $R > 2Q+1$, un nouveau premier dépôt intermédiaire est possible avec $DD0n = R/(2Q+1)$ (gain: $DDOn$ km et $\sim R-2Q-1$ bananes).

Avantage: souvent bien (gain en km et bananes.)

Inconvénient: Ajoute un dépôt (augmente le nombre de parcours): pb lors d'appel récursif.

L'éléphant et les bananes: reste initial ($2^{ème}$ méthode)

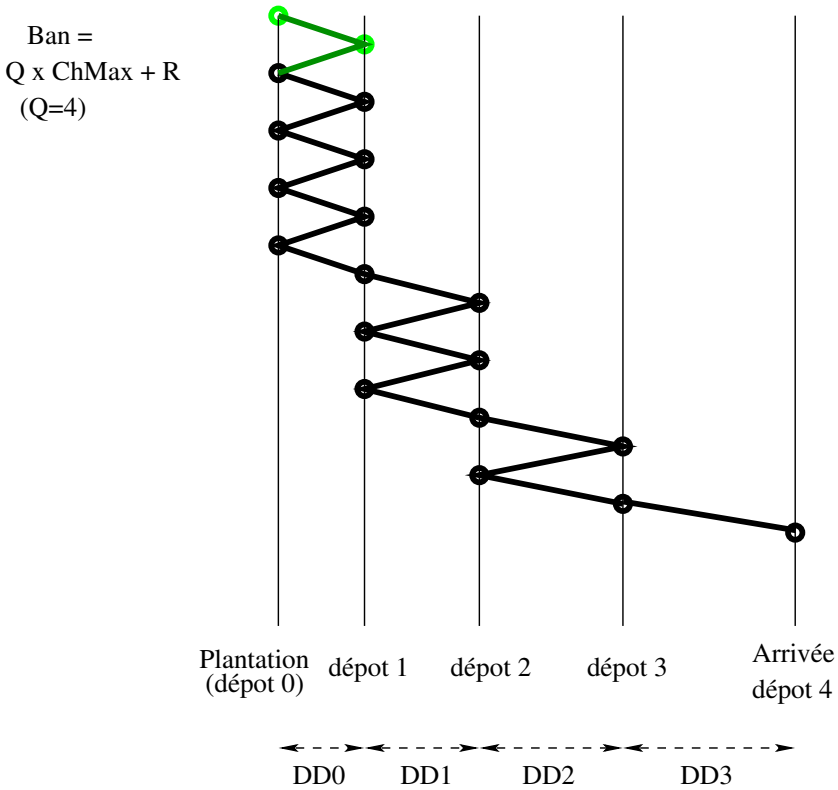

Figure: Problème du reste

$Ban = QChmax + R(\leq Chmax)$.
$DD0 = ChMax/(2Q-1)$
$DD1 = ChMax/(2Q-3)$
$DD2 = ChMax/(2Q-5), ...$

Si $R > 2DD0$, un $Q+1^{eme}$ départ (vert) apporte $R-2DD0$ en plus en 1.

Avantage: Pas de nouveau dépôt.
Inconvénient: Pas sûr que ce soit très supérieur à la première méthode (nombre final, appels récursifs,...).
Le mieux est d'ailleurs d'utiliser une des deux

L'éléphant et les bananes sans explosion combinatoire, vraiment?

- Facilité (factice) d'écriture de programmes ASP traduisant ces algorithmes, même s'ils semblent moins miraculeux car on voit mieux les hypothèses faites qu'avec OCaml (subjectif?). Facile de tester de nombreuses variantes.
- En fait, la méthode décrite semble optimale si on accepte de couper les bananes! Mais la solution en nombres entiers est plus délicate. La meilleure méthode que j'ai trouvée utilise cet algorithme, mais en le relançant de temps en temps en cours de parcours. Cela provoque des appels récursifs, compliqués à écrire à cause de l'absence de "templates". Il faut ajouter un paramètre (qui permet les appels récursifs) à chaque prédicat. La facilité d'écriture et la lisibilité des programmes en souffrent beaucoup.
- De plus, on arrive vite à des "bad_alloc": un programme qui marche avec une seule relance, ne marchera plus avec deux...

L'éléphant sans explosion combinatoire (?): relances

Un exemple de règles permettant ces "relance"":

- % On décrit le champ sur lequel on veut tenter des relances
  % entre 1/2 et 3/4 du nombre de dépôts nécessaires, ou au dernier
  (fin des relances) I,J sont ici des entiers représentant les dépôts.

```
rel(I,J) :- dep(I,J), ndep(I,Ndep), I+Ndep/2 <= J,
            J <= I+3*Ndep/4.
rel(I,I+Ndep+1) :- ndep(I,Ndep).
```

- % génération indéterministe de ces relances

```
{relan(I,J)} 1 :- rel(I,J).
:- relan(I,J), relan(I,J1), J < J1. % <= 1 relance par AS
```

Plusieurs restrictions actuelles de clingo compliquent l'écriture:
parfois nécessité d'utiliser des "domain predicates"
et d'autres parties redondantes dans les corps de règles.

L'éléphant sans explosion combinatoire, vraiment?

On pourrait aussi, au lieu de relancer récursivement en cours de trajet permettre de petites variations autour des distances calculées. C'est facile en ASP, mais gare à l'explosion:

Avec les données initiales, on peut le faire ($7^2 = 49$, permet des variations de ± 3, qui devraient donner la valeur optimale).

Mais avec 50000 bananes, charge max 990, distance 2000 km (50 dépôts intermédiaires, pauvre éléphant!), une variation de ± 1 des 50 distances donne 3^{50} possibilités.

Je n'ai toujours pas de programme satisfaisant, pour de telles données (qui de toute façons épuiseraient l'éléphant...).

L'éléphant sans explosion combinatoire (?) (1)

Ici donc, pas de relance récursive, mais petites variations autour des distances calculées. Même ainsi, le programme est moins joli...

```
diveucl(ban,load,N,R,N-1) :- N := ban / load,
        R := ban #mod load, N > 0, R <= 2*N+1.
diveucl(ban,load,0,R,0) :- 0 == ban / load, R := ban.
diveucl(ban,load,N,R,N) :- N := ban #div load,
        R = ban #mod load, R > 2*N+1.

nbrN(N) :- diveucl(ban,load,N,R,D). % quotient de ban par load
ndepp(D) :- diveucl(ban,load,N,R,D). % nbre de dépôts
                                        intermédiaires potentiels
d(0..D+1):- ndepp(D). % d(D): D est un dépôt.

reste :- diveucl(ban,load,N,R,N), R >= 2*N + 1. %∃ reste utile
```

L'éléphant et les bananes sans explosion combinatoire quelques règles d'un tel programme (moins joli...) (2)

L'utilisateur fournit un écart delta pour générer des distances proches des calculées, grâce à des règles de génération indéterministes:

```
delta(C1..delta) :- C1 := 0 - delta.
```
% intervalle $[-delta, delta]$

Autant d'AS que de suites de distances testées. Initialisation et itérations:

```
1 {ddp(0,1,DD1+Delta) :  delta(Delta)} 1 :- d(0;1), reste,
    diveucl(ban,load,N,R,D), DD1 := R / (2*N+1).
1 {ddp(0,1,DD1+Delta) :  delta(Delta)} 1 :- d(0;1),
    not reste, nbrN(N), DD1 := load / (2*N-1).
1 {ddp(I,I+1,DDI+Delta) :  delta(Delta)} 1:- d(I;I+1), I>0,
    reste, nbrN(N), DDI := load / (2*(N-I)+1).
1 {ddp(I,I+1,DDI+Delta) :  delta(Delta)} 1:- d(I;I+1), I>0,
    not reste, nbrN(N), DDI := load / (2*(N-I)-1).
```

L'éléphant et les bananes sans explosion combinatoire à quoi ressemble un tel programme

Il faut ensuite traiter ces distances (en tenant compte de la distance totale qui n'intervient pas ci-dessus!) et construire les trajets.
On obtient des résultats comme:

```
Answer:  19      Données, puis trajets résultats:
ldbdelta("load",1000,"dist",1500,"ban",5888,"c",1,"delta",2)
trajet(0,11,0,1,78,6,5888,-112,5030) 0 ⇝ 1
trajet(11,20,1,2,109,5,5030,30,4019) ...  1 ⇝ 2
trajet(35,34,5,6,641,0,998,998,641) 5 ⇝ 6
res("nbrTraj",34,"nbrDépInt",5,"perdues",5247,"Aband",1077,
    "Mangées",4170,"arriv",6,"bananesArr",641)
Optimization:  17163376
OPTIMUM FOUND Models :  5 Enumerated:  19 Optimum :  yes
Optimization:  17163376 Time :  131.460 (!)
```

(Des non optimaux sont affichés: (ici 14, et les "5 optimaux"(?)).

Utilisation de règles d'optimisation (en clingo)

- Pour un programme qui teste différentes possibilités (une par answer set), des règles d'optimisations existent dans tous les systèmes.
- On maximise le nombre de bananes du marché. On voudrait aussi (secondairement) minimiser la distance parcourue.
- Deux prédicats donnent les résultats:
  `vendable(Ban)`: *Ban* bananes apportées au marché
  `pertesDistTot(Pert,Cons)`: *Pert* bananes abandonnées, *DistTot* km parcourus.

- ```
  #maximize [vendable(Ban) = Ban @ 2 ].
  #minimize [pertesDistTot(Pert,Cons) = Cons @ 1 ].
  ```

  Parmi les answers sets qui maximisent *Ban* ("niveau 2"), on préfère ceux qui minimisent *Cons* ("niveau 1").
  On pourrait de même, en troisième priorité, minimiser *Pert*.
- Ces méta prédicats fonctionnent plutôt bien (affichage curieux).
  ```

Un autre exemple de rêve pour finir: résolution de sudoku

Exemple très facile à écrire et très efficace (et utile, si on a n'a pas la solution d'un problème...).
(Un programme pour écrire un problème serait plus délicat: ici le programme résout le problème "globalement" et quasi instantanément, sans s'occuper de la façon dont un humain raisonne.)

```
#const racnbr=3.                          % Les sudokus "classiques" (9x9)
#const maxN = racnbr*racnbr.                    % racnbr est un carré
indptcarre(1..racnbr).              % indices relatifs dans les pts carrés
nbr(1..maxN).                                   % Les nombres concernés

depptcarre(0).
depptcarre(I+racnbr) :- depptcarre(I), nbr(I+racnbr+1).
                                    % indices de départs des pts carrés
```

Résolution de sudoku (suite)

```
1 { case(X,Y,Z) : nbr(Z) } 1 :- not erreur("donnees"),
     nbr(X), nbr(Y).        % génération des valeurs des cases (X, Y)

:- case(X,Y,Z), case(X1,Y,Z), X<X1. % 1 seul nbre par colonne
:- case(X,Y,Z), case(X,Y1,Z), Y<Y1. % 1 seul nbre par ligne
:- case(X+I,Y+J,Z), case(X+I1,Y+J1,Z), depptcarre(X;Y),
  indptcarre(I;I1;J;J1), I!=I1, J <=J1.
:- case(X+I,Y+J,Z), case(X+I1,Y+J1,Z), depptcarre(X;Y),
  indptcarre(I;I1;J;J1), I<=I1, J != J1. % 1 par pt carré
% Attention ici: "!=" indispensable ("<" ne convient pas).
```

Résolution de sudoku (suite)

```
% Liste des cases données: suite de cc(X,Y,Z).
% Vérification des données (du luxe...):
erreur("no ligne",X) :- cc(X,Y,Z), not nbr(X).
erreur("no col",Y) :- cc(X,Y,Z), not nbr(Y).
erreur("nbr",Z) :- cc(X,Y,Z), not nbr(Z).
erreur("lign",X) :- cc(X,Y,Z), cc(X,Y1,Z), Y<Y1.
erreur("col",Y) :- cc(X,Y,Z), cc(X1,Y,Z), X<X1.
erreur("pt carre",X,Y) :- cc(X+I,Y+J,Z), cc(X+I1,Y+J1,Z),
  depptcarre(X;Y), indptcarre(I;I1;J;J1), I<I1, J<=J1.
erreur("pt carre",X,Y) :- cc(X+I,Y+J,Z), cc(X+I1,Y+J1,Z),
  depptcarre(X;Y), indptcarre(I;I1;J;J1), I<=I1, J<J1.
erreur("donnees") :- erreur(X,Y).

% recopie des cases données au départ:
case(X,Y,Z) :- not erreur("donnees"), cc(X,Y,Z).
```

Conclusion

- La programmation par ensemble réponse est bien adaptée à ce genre de problème (sudoku, tours de Hanoï, pont miné, mais aussi éléphant).
- Il est facile d'ajouter des règles pour accélérer les calculs, ou si l'énoncé est un peu modifié, car
  la transcription en ASP de ces règles est souvent quasi littérale.
- Mais des problèmes demeurent, malgré les progrès constants des systèmes actuels, dès que la combinatoire est trop grande.
  - Il faut essayer à tout prix de diminuer cette taille (sûrement plus qu'avec des systèmes non déclaratifs mais plus efficaces), ce qui selon le problème concerné peut être complexe.
  - Si cela ne suffit pas, il faut se plonger dans les calculs internes, et alors l'avantage de l'aspect déclaratif devient moins évident
    (même s'il en demeure une partie sur l'exemple?).

Conclusion (2)

- Les systèmes sont en progrès constant, mais on peut encore espérer:
  - Structures de données moins simplistes (vraies listes, ensembles,...) et "sous programmes" (cas des appels récursifs pour l'éléphant). Existe en partie avec DLV (cf DLV-Complex ***et*** DLT) (clingo, quand?).
  - Introduction d'autre "méta-prédicats" (pour l'axiome du cadre,...).
  - Meilleure aide au débugage.
  - L'intégration instantiateur/ solveur demeure un problème: L'instantiateur fait plus que la transformation en propositionnel mais l'intégration n'est pas encore assez poussée: débordement trop fréquemment en grande partie dû à ce problème. (Des solutions comme iclingo demeurent trop complexes d'utilisation et trop partielles.)

    Quand l'instantiateur donne le résultat, le système ne s'en aperçoit pas. Ne semble pas simple à régler (Torsten Schaub)...

- La programmation par ensemble réponse commence réellement à répondre aux espoirs placés en elle, mais a encore des progrès à faire.

Quelques phrases en fin du livre de Torsten Schaub et al.

(coming soon)

Modeling in ASP is still an art; it requires craft, experience, and knowledge. Although the resulting ASP encodings are usually quite succinct and easy to understand, crafting an ASP encoding that also leads to the best possible system performance is yet not as obvious as it might seem (and we sincerely hope that this shortcoming will be resolved in the future).

True declarativity Unlike traditional logic programming, ASP has succeeded in strictly separating logic from control. Despite this, it is arguably not fully declarative because the way we encode a problem influences the performance of finding its solution. In other words, two equivalent encodings may result in a significantly different runtime behavior in terms of grounding and/or solving. Although the ideal of declarativity appears to be unattainable, ASP still leaves plenty of room for improvement.

Petite liste de sites

- http://www.cs.uni-potsdam.de/t̃orsten/asp/ T. Schaub
- http://sourceforge.net/projects/potassco/ clingo
- http://www.cs.utexas.edu/ vl/teaching/lbai/clingo_guide.pdf direct vers le manuel clingo(bien fait)
- http://www.dlvsystem.com/dlvsystem/index.php/Home DLV
- http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html#AEN928 Bien fait, à lire même sans utiliser DLV.
- http://asptut.gibbi.com/ tester DLV sans l'installer obsol.?)
- https://www.mat.unical.it/dlv-complex
- https://www.mat.unical.it/ianni/wiki/dlt DLT
- http://www.cs.utexas.edu/users/tag/cmodels.html CModels
- http://www.info.univ-angers.fr/pub/claire/asperix/
- http://www.hakank.org/answer_set_programming/zebra.lp. Intéressant: 2 solutions dont une seule efficace. Autres ex.

Petite bibliographie

- L'historique: Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. Logic Programming (LP-88),1070–1080, Seattle, 1988. MIT Press.
- L'incontournable, mais pas très récent:
  Chitta Baral Knowledge representation, reasoning and declarative problem solving. Cambridge University Press. (2003)
- Le futur (plus "pratique"):
  M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. Answer set solving in practice. Morgan & Claypool pub. 2012
- Présentation de DLV, mais aussi de ASP en général:
  N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic (TOCL), 7(3):499–562, 2006.
- M. Gelfond Answer sets. In: Handbook of Knowledge Representation, Elsevier, pp. 285–316. 2008