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

Yves Moinard

INRIA Bretagne-Atlantique, IRISA, Rennes

Cinquièmes Journées de l'Intelligence Artificielle Fondamentale (JIAF) juin 2011, Lyon

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.

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:** Solutions = modèles (au lieu de preuves).
  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!)

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... ***DLV-Complex, DLT (Templates),...***

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... ***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,...)

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... ***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,...)
- ***CModels*** Yulia Lierler et autres du Texas Action Group at Austin (Vladimir Lifschitz),

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... ***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,...)
- ***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).

ASP Intro (suite)

The Second Answer Set Programming Competition (Int. Conf. on Logic Programming and Nonmonotonic Reasoning 2009, Leuven, Belgique). "The Potassco [Torsten Schaub et al., Postdam] team is the clear winner"

Autre extrait du compte rendu: "The goal of declarative problem solving is to minimize the effort of programmers"

ASP Intro (suite)

The Second Answer Set Programming Competition (Int. Conf. on Logic Programming and Nonmonotonic Reasoning 2009, Leuven, Belgique). "The Potassco [Torsten Schaub et al., Postdam] team is the clear winner"

Autre extrait du compte rendu: "The goal of declarative problem solving is to minimize the effort of programmers"

Third (Open) ASP Competition -May 16th 2011: 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**.

Règle d'un programme logique

$$\mathit{Tête} \leftarrow \mathit{Corps}. \qquad (1)$$

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***).

Règle d'un programme logique

$$\mathit{Tête} \leftarrow \mathit{Corps}. \qquad (1)$$

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.

Règle d'un programme logique

$$\mathit{Tête} \leftarrow \mathit{Corps}. \qquad (1)$$

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)

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

c'est (entre autres) la tâche du ***grounder*** (`LParse` ou `gringo` pour Smodels ou clasp).

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

Règle d'un programme logique et ensemble réponse (1) Instantiation (rôle de l'instantiateur)

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

c'est (entre autres) la tâche du ***grounder*** (`LParse` ou `gringo` pour Smodels ou 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

Règle d'un programme logique et ensemble réponse (1) Instantiation (rôle de l'instantiateur)

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

c'est (entre autres) la tâche du ***grounder*** (`LParse` ou `gringo` pour Smodels ou 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

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 $(q(b) \wedge s(a,b)) \rightarrow p(a)$ (issue de ▸ (r2))

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 $(q(b) \wedge s(a,b)) \rightarrow p(a)$ (issue de ▸ (r2))
  2. et est ***mimimal pour*** $\subseteq$ ***avec cette propriété***.

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 $(q(b) \wedge s(a,b)) \rightarrow p(a)$ (issue de ▸ (r2))
  2. et est ***mimimal pour*** $\subseteq$ ***avec cette propriété***.

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

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.

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*.

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**.*

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* ...)

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* ...)

  Deux AS (ensembles réponse / "answer sets"): $\{p, \neg s\}$ et $\{q\}$.

  (Le $\vee$ des *Tête* tend à devenir exclusif par minimisation.)

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* ...)

  Deux AS (ensembles réponse / "answer sets"): $\{p, \neg s\}$ et $\{q\}$.

  (Le $\vee$ des *Tête* tend 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* $q \leftarrow .$]

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* ...)
  Deux AS (ensembles réponse / "answer sets"): $\{p, \neg s\}$ et $\{q\}$.
  (Le $\vee$ des *Tête* tend 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*** (tête = singleton sans *not*, pas de 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).

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 = $triste \leftarrow not\ \ couci\text{-}couça.$
  r10 = $couci\text{-}couça \leftarrow not\ \ heureux.$ ***Aucun AS***

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* ← *not triste*. r9 = *triste* ← *not couci-couça*.
  r10 = *couci-couça* ← *not heureux*. ***Aucun AS***

- Π: r7 = 0 {*s*} 1. (***règle particulière: génération indéterministe;*** *tente de prendre entre* 0 *et* 1 *élément de l'ensemble dans chaque Answer set.*)

  (Les nombres 0 et 1 sont facultatifs ici)

  $F = \{s\}$ et $G = \emptyset$: $\Pi^{\{s\}} = \{s.\}$, $\Pi^{\emptyset} = \emptyset$

  F et G sont les 2 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 $\leftarrow a$.)

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.

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:

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(1),soldat(2),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: soldats & lampe au départ au temps (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

- 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 (`not goal(M-1)` est facultatif, voir ▸ là pour `goal`.)
- ne plus traverser après "tmax" (explosion).

Traduction ASP: une règle de génération ▸ indéterministe
d'une traversée de $L0$ vers L

(chaque soldat éligible peut traverser, ou pas):

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).
   ```
   ```

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 ";": `soldat(S1), soldat(S2),...;`
`<` au lieu de `!=` ($\neq$) limite l'instantiation.]

8. 1 soldat suffit pour porter la lampe:

```
move(M,L,lampe) :- move(M,L,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 ";": `soldat(S1), soldat(S2),...;`
`<` 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 pour calculer la durée des traversées effectuées (`#max` pas admis 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 (10),
suivi d'une contrainte (11) qui élimine les modèles ne le satisfaisant pas.
On donne enfin (12) la durée de chaque trajet solution trouvé.

Le pont miné: programme, atteindre le but

Comme souvent en ASP dans un problème de planification:
on décrit d'abord le but (10),
suivi d'une contrainte (11) qui élimine les modèles ne le satisfaisant pas.
On donne enfin (12) la durée de chaque trajet solution trouvé.

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.
```

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.

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.

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.

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 sur son dos. Le marché se trouve à 1000 km de la plantation. Combien de bananes le planteur pourra-t-il porter au maximum au marché?

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 sur son dos. 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 au problème avant de programmer sous peine de crash (`bad_alloc`).

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 sur son dos. 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 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?) comment calculer.

L'éléphant et les bananes: hypothèses pour élagage

Méthode générale nécessitant peu de réflexion particulière:

1. Changer d'unité: l'utilisateur permet de prendre les bananes (ou km) par 100, par 33, par 10..... Fournit une solution, en particulier les distances entre dépôts intermédiaires.
   L'utilisateur diminue alors la taille de l'unité interne et encadre les distances possibles entre dépôts en s'inspirant de cette première réponse.

L'éléphant et les bananes: hypothèses pour élagage

Méthode générale nécessitant peu de réflexion particulière:

1. Changer d'unité: l'utilisateur permet de prendre les bananes (ou km) par 100, par 33, par 10..... Fournit une solution, en particulier les distances entre dépôts intermédiaires. L'utilisateur diminue alors la taille de l'unité interne et encadre les distances possibles entre dépôts en s'inspirant de cette première réponse.

Ce qui suit nécessite (hélas...) d'étudier plus en détail le problème:

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

L'éléphant et les bananes: hypothèses pertinentes

Il me semble que toutes ces hypothèses forment un ensemble ***pertinent***, mais je laisse les démonstrations à d'autres...

L'éléphant et les bananes: hypothèses pertinentes

Il me semble que toutes ces hypothèses forment un ensemble ***pertinent***, 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.

L'éléphant et les bananes: hypothèses pertinentes

Il me semble que toutes ces hypothèses forment un ensemble ***pertinent***, 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.

Un ensemble est ***potentiellement pertinent*** s'il est pertinent quand il existe au moins un trajet qui le satisfait. L'intérêt est que cela permet de traiter le problème avec des données numériques légèrement différentes: on essaie les hypothèses les plus fortes, et si le programme ne trouve pas de solution, on affaiblit l'ensemble d'hypothèses jusqu'à ce qu'il devienne pertinent.

L'éléphant et les bananes: hypothèses pertinentes

Il me semble que toutes ces hypothèses forment un ensemble ***pertinent***, 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.

Un ensemble est ***potentiellement pertinent*** s'il est pertinent quand il existe au moins un trajet qui le satisfait. L'intérêt est que cela permet de traiter le problème avec des données numériques légèrement différentes: on essaie les hypothèses les plus fortes, et si le programme ne trouve pas de solution, on affaiblit l'ensemble d'hypothèses 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!) avec la méthode utilisée pour le pont.

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!) avec la méthode utilisée pour le pont.
J'ai donc utilisé une autre méthode, plus lourde à écrire:
Au lieu de construire un ensemble réponse par itinéraire possible, tous les itinéraires possibles (satisfaisant les hypothèses) sont calculés dans un seul gros ensemble réponse. Cela nécessite d'indexer ces itinéraires pour les identifier, c'est fait par une suite comprenant:

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!) avec la méthode utilisée pour le pont.
J'ai donc utilisé une autre méthode, plus lourde à écrire:
Au lieu de construire un ensemble réponse par itinéraire possible, tous les itinéraires possibles (satisfaisant les hypothèses) sont calculés dans un seul gros ensemble réponse. Cela nécessite d'indexer ces itinéraires pour les identifier, c'est fait par une suite comprenant:

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

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!) avec la méthode utilisée pour le pont.
J'ai donc utilisé une autre méthode, plus lourde à écrire:
Au lieu de construire un ensemble réponse par itinéraire possible, tous les itinéraires possibles (satisfaisant les hypothèses) sont calculés dans un seul gros ensemble réponse. Cela nécessite d'indexer ces itinéraires pour les identifier, c'est fait par une suite comprenant:

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

Je n'y croyais pas trop, mais cela a marché.

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

Extrait: une règle de construction d'un itinéraire (cf ▸ pont ▸ miné):

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

Extrait: une règle de construction d'un itinéraire (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.
```

hurry up!

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

Extrait: une règle de construction d'un itinéraire (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.
```

hurry up! *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, mais 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: le programme (2)

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 est plus aisé à manipuler.

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

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 est 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].

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

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 est 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...

hurry up!

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

Le cas d'un parcours aller est similaire avec l'hypothèse forte (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 par ex. avec 2900 bananes au lieu de 3000.

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

Le cas d'un parcours aller est similaire avec l'hypothèse forte (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 par ex. avec 2900 bananes au lieu de 3000.
Pour calculer la charge maximale apportée au marché, on cherche les trajets `iti` qui vont au marché (dépôt `ndep`) et la charge `LEf` qu'ils y apportent (fonction `loadA` facultative, ainsi que...):

```
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: le programme (4)

Une fois les trajets satisfaisant les hypothèses calculés, calcul de la charge maximale apportée au marché (ne garder que ces trajets):

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

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

Une fois les trajets satisfaisant les hypothèses calculés, calcul de la charge maximale apportée au marché (ne garder que ces trajets):

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

Remarquer la différence avec End1 ▸ le pont miné : on n'utilise plus un méta prédicat d'optimisation `#minimize`, qui cherche l'extrémum (ici minimum) entre tous les ensembles réponses mais un méta-prédicat qui calcule le maximum d'une valeur à l'intérieur d'un même ensemble-réponse, `#max`.

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

Une fois les trajets satisfaisant les hypothèses calculés, calcul de la charge maximale apportée au marché (ne garder que ces trajets):

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

Remarquer la différence avec End1 ▸ le pont miné : on n'utilise plus un méta prédicat d'optimisation `#minimize`, qui cherche l'extrémum (ici minimum) entre 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 d'ailleurs ici une première fois de la simplicité structurelle du présent programme, car le `#max` de gringo n'aime (pour l'instant) pas trop la récursivité (utilisée ici) dans les programmes de structure un peu complexe (comme le pont miné où `#max` n'était pas admis).

hurry up!

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

On en déduit les itinéraires optimaux satisfaisant les hypothèses grâce aux règles suivantes:

```
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: le 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)
```

L'éléphant et les bananes: le 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)
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 3 adaptée ici, en 7.4 s avec l'hypothèse faible de charge maxi ou dispo):

```
Answer:  1 ddO12otpkm(200,330,470) maxloadA(530)
```

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 3 adaptée ici, en 7.4 s avec l'hypothèse faible de charge maxi ou dispo):

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

Ainsi, 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 `dd2valkm`

```
          :
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...]

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, mais l'essentiel semble occupé à écrire le résultat, en majeure partie inutile...):

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, mais l'essentiel semble occupé à écrire le résultat, en majeure partie inutile...):

Et là, surprise: 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).

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).

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).

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 itinéraire (la méthode utilisée pour le pont miné) avec d'autres que clingo.

Conclusion

- La programmation par ensemble réponse est bien adaptée à ce genre de problème.
- 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.

Conclusion

- La programmation par ensemble réponse est bien adaptée à ce genre de problème.
- 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.

Conclusion

- La programmation par ensemble réponse est bien adaptée à ce genre de problème.
- 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)

- Systèmes en progrès constant, mais on peut encore espérer:
  - Structures de données moins simplistes (listes, ensembles,...) et “sous programmes” (pas concerné sur ces deux exemples). Existe en partie avec DLV (DLV-Complex ***et*** DLT)

Conclusion (2)

- Systèmes en progrès constant, mais on peut encore espérer:
  - Structures de données moins simplistes (listes, ensembles,...) et "sous programmes" (pas concerné sur ces deux exemples). Existe en partie avec DLV (DLV-Complex ***et*** DLT) (clingo, quand?).
  - Introduction d'autre "méta-prédicats" (axiome du cadre,...).

Conclusion (2)

- Systèmes en progrès constant, mais on peut encore espérer:
  - Structures de données moins simplistes (listes, ensembles,...) et "sous programmes" (pas concerné sur ces deux exemples). Existe en partie avec DLV (DLV-Complex ***et*** DLT) (clingo, quand?).
  - Introduction d'autre "méta-prédicats" (axiome du cadre,...).
  - 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.

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

Conclusion (2)

- Systèmes en progrès constant, mais on peut encore espérer:
  - Structures de données moins simplistes (listes, ensembles,...) et "sous programmes" (pas concerné sur ces deux exemples). Existe en partie avec DLV (DLV-Complex ***et*** DLT) (clingo, quand?).
  - Introduction d'autre "méta-prédicats" (axiome du cadre,...).
  - 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.

    Quand l'instantiateur donne le résultat, le système ne le sait 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.