# Backjumping pour le calcul d'ensembles réponses dans les solveurs ASP guidés par les regles

Stéphane N'goma, Laurent Garcia, Claire Lefevre, Igor Stéphan

▶ **To cite this version:**

Stéphane N'goma, Laurent Garcia, Claire Lefevre, Igor Stéphan. Backjumping pour le calcul d'ensembles réponses dans les solveurs ASP guidés par les regles. 6es Journées de l'Intelligence Artificielle Fondamentale, IAF 2012, 2012, Toulouse, France. pp.189-198. hal-03350667

**HAL Id: hal-03350667**
**https://hal.univ-angers.fr/hal-03350667**

Submitted on 22 Sep 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

*Actes IAF 2012*

# Backjumping pour le calcul d'ensembles réponses dans les solveurs ASP guidés par les règles

**Stéphane N'Goma** **Laurent Garcia** **Claire Lefèvre** **Igor Stéphan**
LERIA, Université d'Angers
2, bd Lavoisier, 49045 Angers Cedex 01

{stephane.ngoma,laurent.garcia,claire.lefevre,igor.stephan}@info.univ-angers.fr

**Résumé**

Nous nous intéressons dans cet article au calcul des ensembles réponses dans les solveurs ASP guidés par les règles pour lesquels nous étudions l'utilisation du backjumping. Nous en donnons l'étude théorique ainsi que quelques pistes de l'implémentation en cours de réalisation dans le solveur ASPeRiX.

**Abstract**

The present article is about the use of backjumping for computing answer sets with ASP solvers guided by the rules. We give the theoretical aspects of this proposal and some points about the implementation which is in progress in the solver ASPeRiX.

## 1 Introduction

La programmation par ensembles réponses, plus connue sous l'appellation anglaise d'Answer Set Programming (ASP), est un formalisme déclaratif de raisonnement non monotone apparu dans les années 90 adapté à de nombreux domaines de la représentation des connaissances en IA (raisonnement de sens commun, web sémantique, raisonnement causal, ...) et à la résolution de problèmes combinatoires (planification, théorie des graphes, configuration, bioinformatique ...). L'approche a été rendue particulièrement attractive grâce au développement de divers solveurs efficaces comme les systèmes dédiés `Smodels` [16], `DLV` [9] et `Clasp` [3], ou ceux basés sur des solveurs SAT comme `Assat` [10] et `Cmodels` [6]. Pour résoudre un problème via l'ASP, on l'encode donc sous forme d'un programme logique dont les modèles correspondent aux solutions du problème et le solveur ASP a pour rôle de calculer ces modèles.

La représentation des connaissances en ASP va passer par l'écriture d'un programme contenant généralement des variables. Or, la quasi-totalité des solveurs travaillent sur un programme propositionnel obtenu par instanciation préalable du programme initial via un logiciel dédié appelé grounder. L'ensemble de ces solveurs sont basés sur des algorithmes de recherche pour lesquels le point de choix est effectué sur la présence ou non d'un littéral dans un modèle. Mais d'autres systèmes, tels `Gasp` [14] et `ASPeRiX` [7, 8], ont été conçus pour mettre en œuvre un calcul des modèles qui ne nécessite pas cette instanciation préalable du programme : l'approche par les règles. Si l'algorithme sous-jacent est encore un algorithme de recherche, le point de choix est sur l'application ou non d'une règle instanciée qui est générée à la volée.

Le *retour-arrière* (ou *backtracking*) est une technique pour simuler le non-déterminisme (choisir sans la garantie de faire un choix pertinent pour la suite) dans une procédure : le choix, sur lequel la procédure pourra revenir, ainsi que son contexte sont conservés dans une *pile de retour-arrière*. L'utilisation de la pile de retour-arrière revient à un parcours d'arbre dans lequel les nœuds sont les points de choix. Cette technique est à la base de l'implémentation des algorithmes de recherche. Le *retour-arrière intelligent* (*intelligent backtracking* ou *backjumping* ou *dependency-directed backtracking*) est une technique issue de la résolution du problème SAT [12] (et avant cela, du domaine de la propagation de contraintes [17]) qui modifie le retour-arrière en éliminant la seconde branche d'un nœud dont le symbole propositionnel n'a pas participé à l'insatisfiabilité de la première branche (i.e. en modifier sa valeur de vérité ne changera pas l'insatisfiabilité).

Les techniques d'apprentissage de lemmes ont été

adaptées à l'ASP aussi bien dans [19, 20], par extension du solveur `Smodels` [13], que dans [3] comme idée première de la conception du solveur `Clasp`. Dans le cadre de l'extension du solveur `DLV` [15, 11], il a été montré que la seule utilisation du *backjumping* était pertinente.

Le présent article adapte les techniques de *backjumping* aux solveurs basés sur une approche par les règles.

La section 2 présente les notions fondamentales de l'ASP. La section 3 décrit l'approche par les règles (et son implémentation dans le cadre du solveur `ASPeRiX`) et met en exergue les limites du *backtracking* chronologique. La section 4 spécifie la manière d'étendre l'approche par les règles au *backjumping* et en particulier les différences induites de l'approche par les règles. La section 5 présente quelques éléments pour l'implémentation (en cours) du *backjumping* dans le solveur `ASPeRiX`. La section 6 conclut par une discussion et des perspectives.

## 2 Answer Set Programming

Soit $\mathcal{A}$ l'ensemble des *atomes*. Un *littéral* est soit un atome, soit la *négation forte* $\neg x$ d'un atome $x$. L'expression *not a* est la *négation faible* ou *par défaut* d'un littéral $a$. Un *programme logique* $\Pi$ est un ensemble fini de *règles* de la forme ($n \geq 0, m \geq 0$) :

$$c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m.$$

Si $a_1$, …, $a_n$, $b_1$, …, $b_m$ et $c$ sont des littéraux, on a un *programme logique étendu*. Si ce sont des atomes, on a un *programme logique normal*. Enfin, si $m = 0$, on parle de *programme logique défini*. La tête d'une règle $r$ est notée $tête(r) = c$, le corps positif $corps^+(r) = \{a_1, \ldots, a_n\}$, le corps négatif $corps^-(r) = \{b_1, \ldots, b_m\}$ et $r^+ = tête(r) \leftarrow corps^+(r)$. Le *réduit* d'un programme logique étendu $\Pi$ par rapport à un ensemble de littéraux $L$ est le programme $\Pi^L = \{r^+ | r \in \Pi \text{ et } corps^-(r) \cap L = \emptyset\}$. L'ensemble $Cn(R)$ des conséquences d'un ensemble de règles définies $R$ (sans négation faible) est le plus petit ensemble de littéraux $L$ qui soit clos par rapport à $R$ (i.e., pour toute règle $r$, si $corps^+(r) \subseteq L$ alors $tête(r) \in L$) et qui soit cohérent ou égal à l'ensemble des littéraux. Un ensemble de littéraux $S$ est un *ensemble réponse* (appelé initialement *modèle stable* [4, 5]) d'un programme logique étendu $\Pi$ si $Cn(\Pi^S) = S$. Par exemple, le programme $\{a \leftarrow not\ b.,\ b \leftarrow not\ a.\}$ a deux ensembles réponses $\{a\}$ et $\{b\}$. On utilise aussi des règles sans tête, appellées *contraintes*, qui sont considérées équivalentes à des règles de la forme $bug \leftarrow \ldots, not\ bug.$ où $bug$ est un nouveau symbole. Par exemple, le programme $\{a \leftarrow not\ b.,\ b \leftarrow not\ a.,\ \leftarrow a.\}$ a pour unique ensemble réponse $\{b\}$.

En pratique, les programmes logiques vont généralement contenir des règles avec variables, que nous appellerons règles du *premier ordre*. Les atomes sont alors construits à partir de prédicats n-aires, de constantes, de variables et de symboles de fonctions. De telles règles sont interprétées comme représentant l'ensemble de leurs instances propositionnelles où chaque variable a été remplacée par chaque constante de l'univers de Herbrand. Si $P$ est un programme logique de 1er ordre, on note $ground(P)$ le programme propositionnel équivalent (relativement à la sémantique considérée).

Les principaux systèmes qui permettent le calcul effectif d'ensembles réponses opèrent en deux étapes. Un premier logiciel, le *grounder* (par exemple, `Lparse` [18], `Gringo` [2]) transforme le programme avec variables en un programme propositionnel équivalent lors d'une phase d'instanciation des règles. Puis un second logiciel, le *solveur* (par exemple, `Clasp` [3], `Smodels` [16], `DLV` [9]) recherche les modèles (ensembles réponses) de ce programme propositionnel. Cette instanciation préalable du programme est souvent problématique, voire même bloquante pour certains problèmes [14, 7]. En effet, les termes et atomes nécessaires à la résolution ne sont généralement pas circonscrits avant que le problème ne soit résolu, et on va donc générer en extension tout un espace de recherche potentiel dont une grande partie risque d'être inutile.

L'approche présentée dans la section suivante aborde différemment le calcul des ensembles réponses en travaillant directement avec le programme logique de premier ordre, sans passer par la phase préalable d'instanciation des règles.

## 3 Solveur ASP guidé par les règles

### 3.1 Principe du solveur

Nous présentons ici une méthode de calcul des ensembles réponses pour des programmes du premier ordre, qui est basée sur les règles, et qui intégre l'instanciation des règles à la recherche de modèles [7]. Le solveur `ASPeRiX` [8] [1] implémente cette approche. Les programmes, dits de premier ordre, sont des programmes logiques étendus [2] dont les règles peuvent contenir des variables, des symboles de fonctions et des

1. `http://www.info.univ-angers.fr/pub/claire/asperix`
2. On ne considère pas les ensembles réponses incohérents, un programme possédant un modèle incohérent est traité comme n'ayant pas de modèle. Dans ces conditions, un programme logique étendu peut se ramener à un programme logique normal dans lequel, pour chaque atome $a$, on ajoute un nouvel atome $na$ représentant la négation forte de $a$, et une contrainte $\leftarrow a,\ na.$ interdisant les incohérences.

expressions arithmétiques[3], avec pour seule restriction que les règles soient *sûres*, i.e., que toute variable apparaissant dans une règle doit aussi apparaître dans son corps positif. Pour cette présentation, les contraintes sont écrites avec une tête constituée du symbole $\bot$ (intuitivement interprété comme *inconsistant*).

Une *interprétation partielle* pour un programme $P$ est un couple $\langle IN, OUT \rangle$ d'ensembles disjoints de littéraux de la base de Herbrand de $P$. Intuitivement, $IN$ représente les éléments appartenant au modèle en cours de construction, et $OUT$ ceux qui en sont exclus. Les littéraux n'apparaissant ni dans $IN$, ni dans $OUT$ sont *indéterminés*.

Soit une règle instanciée $r$ et une interprétation partielle $I = \langle IN, OUT \rangle$. On dit que :

- $r$ est *supportée* ssi $corps^+(r) \subseteq IN$ ;
- $r$ est *bloquée* ssi $corps^-(r) \cap IN \neq \emptyset$ ;
- $r$ est *non blocable* ssi $corps^-(r) \subseteq OUT$ ;
- $r$ est *applicable* ssi $r$ est supportée et non bloquée ;
- $r$ est *déclenchable* ssi $r$ est supportée et non blocable.

Notons bien la différence entre une règle non bloquée ($corps^-(r) \cap IN = \emptyset$) et une règle non blocable ($corps^-(r) \subseteq OUT$).

Le principe de l'approche est le suivant. On construit incrémentalement une interprétation partielle $\langle IN, OUT \rangle$ en partant de $IN = \emptyset$ et $OUT = \{\bot\}$[4]. Pour ce faire, on applique des règles en chaînage avant en instanciant les règles de premier ordre au fur et à mesure des besoins grâce aux littéraux précédemment produits. On distingue deux types d'inférence :

- D'une part, une étape de propagation (monotone) qui consiste à déclencher toute instance de règle $r$ telle que $corps^+(r) \subseteq IN$ et $corps^-(r) \subseteq OUT$, i.e., ajouter $tête(r)$ dans $IN$.
- D'autre part, une étape de choix (non monotone) qui consiste à choisir une instance de règle $r_0$ qui soit supportée, $corps^+(r_0) \subseteq IN$, et non bloquée, $corps^-(r_0) \cap IN = \emptyset$, et décider soit de forcer le déclenchement de $r_0$ (en ajoutant $corps^-(r_0)$ dans $OUT$), soit d'interdire son déclenchement en bloquant la règle par l'ajout d'une contrainte $\bot \leftarrow corps^-(r_0)$ (ce qui garantit d'empêcher d'avoir tous les littéraux de $corps^-(r_0)$ dans $OUT$, sans quoi $r_0$ pourrait être utilisée).

Si les ensembles $IN$ et $OUT$ restent bien disjoints tout au long du processus, l'ensemble $IN$ obtenu lorsque plus aucune inférence ne peut être effectuée est un ensemble réponse. La figure 1 illustre ce fonctionnement.

3. L'univers de Herbrand étant potentiellement infini, l'intervalle des entiers et la profondeur d'imbrication des symboles fonctionnels dans les termes sont limités.

4. $\bot$ est mis au départ dans $OUT$ afin de gérer correctement les contradictions lors de son ajout dans $IN$.

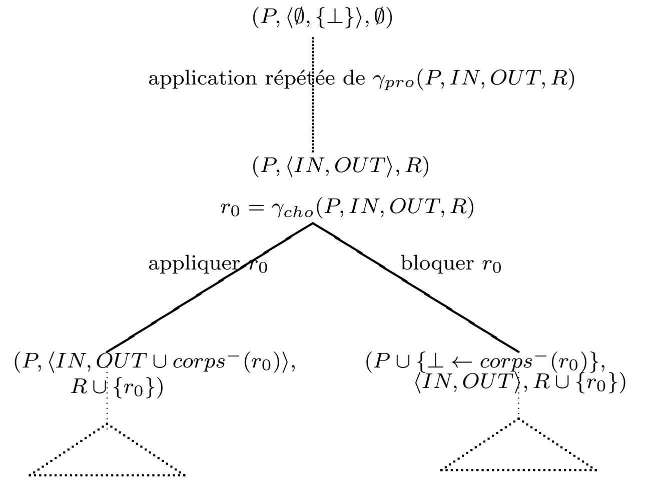


FIGURE 1 – Principe de la procédure de recherche.

Les deux fonctions définies ci-dessous permettent de sélectionner une règle instanciée en effectuant simultanément la sélection d'une règle et une instanciation possible, l'ensemble $ground(P)$, correspondant au programme propositionnel équivalent à $P$, n'étant jamais explicitement construit.

$\gamma_{pro}$(*P, IN, OUT, R*) est une fonction non déterministe qui sélectionne une règle supportée et non blocable par rapport à $\langle IN, OUT \rangle$ dans $ground(P) \setminus R$, ou retourne `faux` si aucune règle n'est déclenchable.

$\gamma_{cho}$(*P, IN, OUT, R*) est une fonction non déterministe qui sélectionne une règle supportée et non bloquée par rapport à $\langle IN, OUT \rangle$ dans $ground(P) \setminus R$, ou retourne `faux` si aucune règle n'est applicable.

La fonction Solve détaille la procédure de recherche pour un programme $P$. Elle doit être initiée par `Solve(`$P_R$, $P_K$, $\emptyset$, $\{\bot\}$, $\emptyset$`)` où $P_K = \{x \in P \mid tête(r) = \bot\}$ (les contraintes) et $P_R = P \setminus P_K$. $CR$ (pour *Chosen Rules*) est l'ensemble des règles instanciées retournées par $\gamma_{pro}$ et $\gamma_{cho}$ pendant la recherche.

La phase de propagation ne fait pas la distinction entre contraintes et règles de $P_R$. Si une contrainte est déclenchée à cette étape, sa tête $\bot$ est ajoutée à $IN$ et une contradiction sera détectée. L'étape de choix, quant à elle, ne considère que les règles de $P_R$ (forcer le déclenchement d'une contrainte n'aurait pas de sens). Par contre, lorsque plus aucune règle de $P_R$ n'est applicable, $IN$ n'est un ensemble réponse que si aucune contrainte de $P_K$ n'est applicable ; sinon $\bot$ pourrait être ajouté à $IN$, la fonction retourne donc faux dans ce cas. L'algorithme présenté ne calcule qu'un ensemble réponse, mais peut facilement être étendu au calcul de tous les ensembles réponses.

**Function** Solve($P_R$, $P_K$, IN, OUT, CR)

**répéter** // phase de propagation
  $r_0 := \gamma_{pro}(P_R \cup P_K,\ IN,\ OUT,\ CR)$;
  **si** $r_0$ **alors**
    $IN := IN \cup \{tête(r_0)\}$;
    $CR := CR \cup \{r_0\}$;
**jusqu'à** $\neg r_0$;
**si** $IN \cap OUT \neq \emptyset$ **alors** // contradiction
  **retourner** faux
**sinon**
  $r_0 := \gamma_{cho}(P_R,\ IN,\ OUT,\ CR)$;
  **si** $\neg r_0$ **alors** // plus de règle applicable
    **si** $\gamma_{cho}(P_K,\ IN,\ OUT,\ \emptyset)$ **alors** // contrainte non satisfaite
      **retourner** faux
    **sinon**// ensemble réponse trouvé
      **retourner** $IN$
  **sinon**// point de choix
    // branche gauche
    $stop :=$ Solve($P_R,\ P_K,\ IN,\ OUT \cup corps^-(r_0),\ CR \cup \{r_0\}$);
    **si** $\neg stop$ **alors** // branche droite
      $stop :=$ Solve($P_R,\ P_K \cup \{\bot \leftarrow corps^-(r_0)\},\ IN,\ OUT,\ CR \cup \{r_0\}$);
    **retourner** $stop$

**Exemple 3.1** *Illustrons la procédure de recherche via le programme suivant. Nous utilisons ici un programme propositionnel pour la simplicité de la compréhension, un exemple utilisant un programme avec variables est développé dans la section suivante.*

$$P_{3.1} = \left\{ \begin{array}{l} x \leftarrow ., \\ a \leftarrow x,\ not\ b,\ not\ d., \\ b \leftarrow x,\ not\ a., \\ c \leftarrow not\ a,\ not\ b., \\ y \leftarrow not\ b., \\ \bot \leftarrow a,\ not\ c. \end{array} \right\}$$

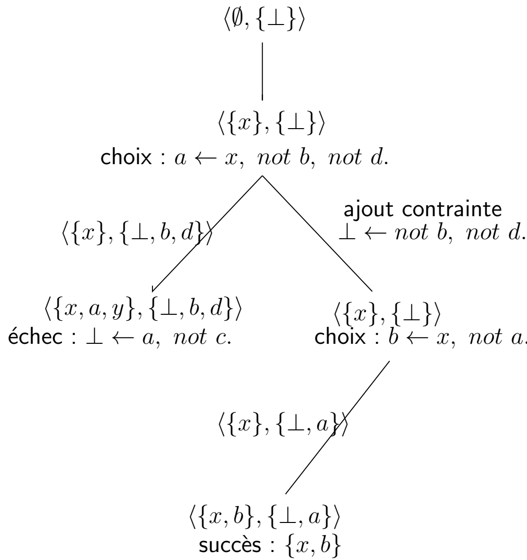


FIGURE 2 – l'arbre de recherche du programme $P_{3.1}$

*Au départ $\langle IN, OUT \rangle = \langle \emptyset, \{\bot\} \rangle$. Une première propagation permet d'ajouter $x$ dans $IN$, on obtient $\langle \{x\}, \{\bot\} \rangle$. Il n'y a plus de règle déclenchable, on choisit donc une règle, par exemple $a \leftarrow x,\ not\ b,\ not\ d.$ et on force son déclenchement en ajoutant son corps négatif dans $OUT$, l'interprétation partielle devient $\langle \{x\}, \{\bot, b, d\} \rangle$. Un nouvelle phase de propagation permet alors d'ajouter $a$ et $y$ dans $IN$, et on obtient $\langle \{x, a, y\}, \{\bot, b, d\} \rangle$. Plus aucune règle n'est alors applicable. Par contre, la contrainte $\bot \leftarrow a,\ not\ c.$ l'est car $a \in IN$ et $c$ est indéterminé. La fonction retourne faux et on revient sur le point de choix précédant (avec l'interprétation $\langle \{x\}, \{\bot\} \rangle$) en bloquant cette fois la règle choisie par l'ajout de la contrainte $\bot \leftarrow not\ b,\ not\ d.$ Cela ne permet aucune propagation, une nouvelle règle applicable est donc choisie, par exemple $b \leftarrow x,\ not\ a.$, et on force son déclenchement : l'interprétation devient $\langle \{x\}, \{\bot, a\} \rangle$. La propagation permet alors d'ajouter $b$ dans $IN$, ce qui donne $\langle \{x, b\}, \{\bot, a\} \rangle$. Il n'y a alors plus de règle applicable, et pas de contrainte applicable non plus, l'ensemble $IN = \{x, b\}$ est donc un ensemble réponse de $P_{3.1}$.*

## 3.2 Les limites du backtracking

**Exemple 3.2** *Soit le programme $P_{3.2}$ suivant qui code une 2-coloration d'un graphe à quatre sommets et deux arêtes.*

$$P_{3.2} = \left\{ \begin{array}{l} s(1) \leftarrow ., s(2) \leftarrow ., s(3) \leftarrow ., s(4) \leftarrow ., \\ arete(1,3) \leftarrow .,\ arete(3,4) \leftarrow ., \\ vert(4) \leftarrow ., \\ rouge(X) \leftarrow s(X),\ not\ vert(X)., \\ vert(X) \leftarrow s(X),\ not\ rouge(X)., \\ \bot \leftarrow arete(X,Y), rouge(X), rouge(Y)., \\ \bot \leftarrow arete(X,Y), vert(X), vert(Y). \end{array} \right\}$$

*Le principe de la recherche guidée par les règles est illustré en figure 3. Aucune propagation ne peut être effectuée au départ. Une instance de règle supportée non bloquée est choisie, par exemple, $rouge(1) \leftarrow s(1),\ not\ vert(1).$ ; on force alors le déclenchement de la règle en supposant $vert(1)$ faux, ce qui permet de déduire que $rouge(1)$ est vrai. Puis une seconde instance de règle est choisie, par exemple, $rouge(2) \leftarrow s(2),\ not\ vert(2).$ ; on suppose $vert(2)$ faux, et on déduit que $rouge(2)$ est vrai. Et enfin, la règle $rouge(3) \leftarrow s(3),\ not\ vert(3).$ est choisie, on suppose $vert(3)$ faux, et on déduit $rouge(3)$. Une contradiction est alors détectée car la contrainte $\bot \leftarrow arete(1,3), rouge(1), rouge(3).$ est déclenchée. On revient alors à la dernière règle choisie et on interdit son application en la bloquant par l'ajout d'une contrainte $\bot \leftarrow not\ vert(3)$. Notons que, avec `ASPeRiX`, le mécanisme de propagation permet dans ce cas précis[5] de déduire que $vert(3)$ doit être vrai. La contrainte*

5. contrainte instanciée avec un corps contenant un seul littéral

*$\bot \leftarrow arete(3,4), vert(3), vert(4).$ est alors déclenchée et cette branche se termine à son tour en échec.*

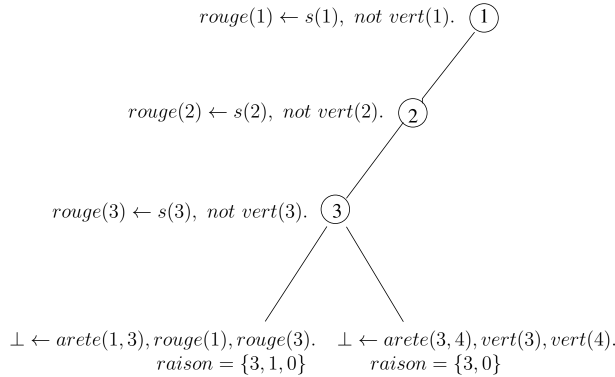


FIGURE 3 – Extrait de l'arbre de recherche de $P_{3.2}$

*Arrivé à ce point, il n'y a plus moyen de colorer le sommet 3. En cas de backtrack simple, on revient au choix précédent (2) et on reprend la recherche avec la nouvelle hypothèse que le sommet 2 doit être vert. Ce sous-arbre est voué à l'échec puisque la coloration du sommet 2 est totalement indépendante du problème rencontré pour colorer le sommet 3. En effet, l'échec au point de choix n° 3, provient des faits que le sommet 3 est connecté aux sommets 1 et 4, que 1 est rouge, et que 4 est vert. Si aucune de ces conditions n'est modifiée, on ne peut espérer trouver une solution au problème. Il est donc inutile d'explorer la branche droite du point de choix n° 2 dans laquelle on retrouvera nécessairement les mêmes inconsistances. Et on peut, en toute sécurité (i.e., en ayant l'assurance de ne « rater » aucune solution), revenir directement au point de choix n° 1 qui porte sur la couleur du sommet 1 et qui est en cause dans l'échec de la coloration du sommet 3.*

Le *backtracking* utilisé par cette version d'`ASPeRiX` est entièrement chronologique. Le *backjumping*, qui consiste à « sauter » des points de choix lors du retour arrière, est présenté dans la section suivante. Il nécessite de connaître les raisons des échecs, ce qui permet de revenir au point de choix le plus proche « en rapport » avec l'échec, i.e., susceptible de modifier les conditions ayant provoqué l'échec.

## 4 Backjumping pour un solveur ASP guidé par les règles

Le *backjumping* est souvent mis en œuvre conjointement au *clause learning* [20, 3], ce qui n'est pas la démarche adoptée ici. Le *clause learning*, qui consisterait pour nous à ajouter des contraintes au programme (contraintes qui résultent d'une analyse des échecs), serait peu efficace car l'approche par les règles privilégie les inférences en chaînage avant et exploite peu les contraintes pour la propagation [6]. Par ailleurs l'utilisation du *backjumping* seul a montré son efficacité dans le solveur `DLV` [15]. Et puisqu'il nécessite le calcul de raisons des échecs, le *backjumping* peut être vu comme une étape préliminaire au *clause learning* qui nécessite l'analyse de ces raisons.

### 4.1 Les raisons

Le *backjumping* nécessite de justifier les échecs, afin d'éviter d'explorer des branches dans lesquelles on retrouvera ces mêmes échecs. Mais qu'est-ce que la raison d'un échec ? Intuitivement, ce sont les littéraux et règles ayant provoqué l'échec. Par exemple, un échec provoqué par le déclenchement de la contrainte $\bot \leftarrow a,\ not\ c.$ est dû à la présence de la contrainte dans le programme et au fait qu'elle soit déclenchable ($a \in IN$ et $c \in OUT$). Il nous faut donc également déterminer les raisons pour lesquelles on ajoute des contraintes dans le programme, et les raisons pour lesquelles on met des littéraux dans $IN$ et dans $OUT$.

La procédure de recherche actuelle ne permet d'ajouter des littéraux dans $IN$ que lors de la phase de propagation : un littéral $a$ est obtenu grâce à une règle de la forme simplifiée $a \leftarrow b,\ not\ c.$ où $b \in IN$ et $c \in OUT$. C'est donc « parce que » $b$ est dans $IN$ et $c$ dans $OUT$ que l'on ajoute $a$.

De même, un littéral ne peut être ajouté dans $OUT$ que lors de la phase de choix, quand on force le déclenchement d'une règle : le corps négatif de la règle est ajouté dans $OUT$ sans autre justification qu'une décision arbitraire (qui implémente le non-déterminisme de la recherche).

Enfin, une contrainte est ajoutée dans le programme uniquement lors de cette même phase de choix, quand on bloque une règle. Là aussi, la seule justification est un choix arbitraire.

Dans une optique de *backjumping*, ce qui nous intéresse dans les justifications ne sont pas directement les littéraux et/ou contraintes en cause, mais les nœuds (points de choix) dans l'arbre de recherche où ces informations ont été produites. A chaque élément de $IN$ et $OUT$, et à chaque règle du programme (les contraintes en particulier), on va associer l'ensemble des points de choix qui ont participé à leur ajout. Dans le cas d'un choix arbitraire, le seul point de choix incriminé suffit à justifier l'ajout. Dans le cas d'une propagation, l'ensemble des raisons des littéraux du corps de la règle sont nécessaires.

6. dans l'état actuel du solveur `ASPeRiX`. Si les contraintes apprises sont des contraintes instanciées, les techniques de propagation des solveurs travaillant avec des programmes propositionnels pourraient être mises en œuvre.

Une *raison* est définie comme un ensemble de niveaux (numérotant les points de choix) dans l'arbre de recherche symbolisés par des entiers positifs ou nuls :

- le niveau 0 correspond à ce qui précède le premier choix (le programme initial et ce qui peut être déduit par propagation de celui-ci) ;
- les points de choix sont numérotés à partir de 1, par ordre chronologique ; un niveau $n$ correspond au $n$-ième point de choix, ainsi que ce qui peut en être directement déduit.

On note, pour un littéral $l$ :

- $R_{IN}(l)$ : la raison pour laquelle $l$ est dans $IN$ ;
- $R_{OUT}(l)$ : la raison pour laquelle $l$ est dans $OUT$ ;
- $R_{\overline{IN}}(l)$ : la raison pour laquelle $l$ n'est pas dans $IN$ ($l$ peut être dans $OUT$ ou être indéterminé) ;

et, pour une règle $r$, $R(r)$ est la raison pour laquelle $r$ a été ajoutée au programme.

La différence entre $R_{OUT}(l)$ et $R_{\overline{IN}}(l)$ est importante ; elle rejoint celle entre une règle non blocable et une règle non bloquée.

### 4.1.1 Les raisons des littéraux de IN, des littéraux de OUT et des contraintes

On définit dans cette section la façon de calculer les raisons pour lesquelles un littéral est ajouté dans $IN$ ou dans $OUT$, et celles pour lesquelles une règle est ajoutée dans le programme. En pratique, seules des contraintes sont ajoutées en cours de recherche mais, afin d'uniformiser le traitement des contraintes et des autres règles, nous définissons des raisons pour toutes les règles du programme.

Les raisons sont définies comme suit, relativement à une interprétation partielle $\langle IN, OUT\rangle$ :

**Les règles.** A chaque règle et contrainte du programme initial, on associe la raison $\{0\}$ correspondant à ce qui est indépendant de tout choix effectué lors de la recherche : pour toute règle $r$ appartenant au programme initial, $R(r) = \{0\}$.

Les seules règles qui peuvent être ajoutées au cours de la procédure de recherche sont les contraintes ajoutées pour bloquer les règles lors des branches droites des points de choix. A une contrainte générée pour bloquer une règle $r$ choisie lors d'un point de choix de niveau $n$, on associe la raison $\{n\}$, le choix arbitraire justifiant à lui seul l'ajout : $R(\bot \leftarrow corps^-(r)) = \{n\}$.

**Les littéraux de $IN$.** Lors de la phase de propagation, on ajoute dans $IN$ la tête de toute règle $r$ supportée et non blocable. Si on ajoute la tête dans $IN$, c'est donc parce que la règle est déclenchable : tous les littéraux du corps de $r$ sont déterminés avec les littéraux du corps positif qui sont dans $IN$ et les littéraux du corps négatif dans $OUT$. La raison de l'ajout de $tête(r)$ dans $IN$ est l'ensemble des raisons pour lesquelles les littéraux du corps positif (resp. négatif) sont dans $IN$ (resp. $OUT$), auquel on ajoute la raison de la règle elle-même (qui est toujours $\{0\}$ si ce n'est pas une contrainte) :

$R_{IN}(tête(r)) =$

$$\bigcup_{l \in corps^+(r)} R_{IN}(l) \cup \bigcup_{l \in corps^-(r)} R_{OUT}(l) \cup R(r)$$

**Les littéraux de $OUT$.** Lors d'un point de choix de niveau $n$, où une règle $r$ est choisie pour être appliquée (branche gauche), les littéraux du corps négatif de la règle sont ajoutés dans $OUT$ avec pour seule justification que l'on a décidé d'appliquer la règle à ce niveau : $\forall l \in corps^-(r) \setminus OUT, \quad R_{OUT}(l) = \{n\}$

### 4.1.2 Les raisons des échecs

Deux formes d'échec sont possibles : soit on découvre une contradiction suite à la phase de propagation ($IN \cap OUT \neq \emptyset$), soit il n'y a plus aucune règle applicable (nous sommes sur une feuille de l'arbre) mais il y a une contrainte non satisfaite, i.e., supportée et non bloquée.

Dans le premier cas, il existe au moins un littéral $l \in IN \cap OUT$, la raison de la contradiction est simplement la raison pour laquelle $l$ est dans $IN$ et la raison pour laquelle $l$ est dans $OUT$ :

$$R_{IN}(l) \cup R_{OUT}(l)$$

Le second cas, où il existe au moins une contrainte non satisfaite, est spécifique à notre approche. En effet, la grande majorité des solveurs opérant leurs choix sur les atomes, lorsque tous les choix sont faits, la valeur de chaque atome est déterminée. Ce n'est pas le cas de notre approche où, les choix portant sur les règles, l'interprétation $\langle IN, OUT\rangle$ peut rester partielle jusqu'à la fin de la recherche ; les littéraux indéterminés (qui ne sont ni dans $IN$ ni dans $OUT$) sont des littéraux non prouvables (puisqu'il n'y a plus de règle qui peut être appliquée) et qui ne peuvent donc pas appartenir au modèle. On peut alors considérer ces littéraux comme appartenant à $OUT$, et $IN$ ne sera un ensemble réponse que si cette nouvelle hypothèse ne rend pas de contrainte déclenchable. Ou encore, sans modifier l'ensemble $OUT$, $IN$ sera un ensemble réponse si aucune contrainte n'est applicable relativement à l'interprétation partielle $\langle IN, OUT\rangle$. S'il existe une contrainte $c$ non satisfaite, la raison de l'échec est l'ensemble des raisons qui rendent la contrainte supportée et non bloquée :

$$\bigcup_{l \in corps^+(c)} R_{IN}(l) \cup \bigcup_{l \in corps^-(c)} R_{\overline{IN}}(l) \cup R(c)$$

La difficulté ici est que la contrainte est non bloquée ($corps^-(c) \cap IN = \emptyset$) mais pas non blocable

($corps^-(c) \nsubseteq OUT$), sinon elle aurait été déclenchée lors d'une phase de propagation. Il y a donc au moins un littéral du corps négatif dont le statut est resté indéterminé et donc pour lequel la raison de l'absence dans $IN$ est inconnue. Nous allons détailler dans la section suivante la manière de calculer la raison de l'absence d'un littéral, i.e. la raison pour laquelle il n'a pas pu être prouvé.

### 4.1.3 Les raisons des littéraux indéterminés

Les raisons pour lesquelles un littéral $l_0$ n'appartient pas à $IN$ sont connues uniquement si $l_0$ appartient à $OUT$, sinon on sait seulement que $l_0$ n'a pas pu être prouvé (il est indéterminé). Pourquoi un littéral n'a-t-il pas été prouvé dans une interprétation partielle $\langle IN, OUT \rangle$ ? Intuitivement, si $l_0$ n'a pas été prouvé, c'est parce qu'aucune des règles instanciées concluant $l_0$ n'a pu être déclenchée. Il nous faut donc déterminer pourquoi une règle n'est jamais déclenchée tout au long d'une branche de l'arbre de recherche. Commençons par définir quelques notions préliminaires.

Soient $\langle IN, OUT \rangle$ une interprétation partielle, et $r$ une règle instanciée. On dit qu'un littéral $l$ du corps de la règle *neutralise* $r$ si $l \in corps^+(r) \setminus IN$ ou $l \in corps^-(r) \cap IN$. Un littéral neutralise donc une règle s'il empêche la règle de pouvoir être appliquée : c'est un littéral du corps positif qui n'est pas dans $IN$ et donc empêche la règle d'être supportée, ou un littéral du corps négatif qui est dans $IN$ et donc bloque la règle. On peut alors dire qu'une règle $r$ est non applicable si et seulement si il existe un littéral $l$ qui neutralise $r$. Chaque littéral $l$ neutralisant $r$ est donc une cause, une raison pour laquelle la règle $r$ n'est pas applicable, raison que nous notons *RNonApp(r)*. Si $l$ neutralise $r$, alors

$$RNonApp(r) = \begin{cases} R_{\overline{IN}}(l) & \text{si } l \in (corps^+(r) \setminus IN) \\ R_{IN}(l) & \text{si } l \in (corps^-(r) \cap IN) \end{cases}$$

Revenons maintenant aux raisons pour lesquelles un littéral $l_0$ n'a pas pu être prouvé dans une interprétation partielle $\langle IN, OUT \rangle$ : c'est parce qu'aucune règle qui conclut $l_0$ n'a pu être déclenchée. Notons $Rl_0$ l'ensemble des règles instanciées de tête $l_0$. Si aucune règle de $Rl_0$ n'a été déclenchée durant le recherche, c'est que chaque règle $r$ de $Rl_0$ soit est non applicable ($r$ est neutralisée par un littéral), soit, si elle était applicable, a été choisie lors d'un point de choix et bloquée par l'ajout d'une contrainte dans le programme. Dans ce dernier cas, la raison du non-déclenchement de la règle est simplement ce choix arbitraire.

**Exemple 4.1** *Considérons l'interprétation partielle $\langle\{x\}, \{c, d\}\rangle$, et supposons que $a$ n'est pas prouvable dans cette interprétation et que les deux seules règles (instanciées) concluant $a$ sont $r_1 = a \leftarrow y,\ not\ c.$ et $r_2 = a \leftarrow x,\ not\ b,\ not\ d.$.*

*$a$ n'est pas prouvable parce que, d'une part, $r_1$ n'est pas supportée, donc pas applicable (neutralisée par $y$) et, d'autre part, $r_2$ n'a pas été déclenchée (alors qu'elle est pourtant applicable). $r_2$ a donc nécessairement été sélectionnée lors d'un point de choix, et bloquée par l'ajout d'une contrainte $\bot \leftarrow not\ b,\ not\ d.$[7]. Finalement, $a$ n'a pas pu être prouvé parce que $y$ neutralise $r_1$ et $\bot \leftarrow not\ b,\ not\ d.$ bloque $r_2$. La raison pour laquelle $a$ n'est pas dans $IN$ sera l'union de $R_{\overline{IN}}(y)$ et $R(\bot \leftarrow not\ b,\ not\ d.)$.*

Pour distinguer les règles de $Rl_0$ qui sont bloquées par une contrainte lors d'un point de choix de celles qui sont neutralisées par des littéraux, on utilise l'ensemble $CR$ des instances de règles choisies durant la recherche (ces règles incluent toutes les règles déclenchées par propagation ainsi que toutes celles sélectionnées lors des points de choix, voir fonction Solve) :

- $Rl_0 \cap CR$ sont les règles de $Rl_0$ qui ont été choisies sur cette branche mais bloquées par une contrainte. Une raison pour laquelle on n'a pas déclenché la règle est simplement la présence de la contrainte.
- $Rl_0 \setminus CR$ sont des règles non applicables [8] de $Rl_0$, donc non supportées ou bloquées. Chaque littéral qui neutralise la règle fournit une raison pour laquelle elle n'a pas été déclenchée.

Notons *RNonDec(r)* la raison pour laquelle la règle $r$ n'est pas déclenchable. On a :

$$RNonDec(r) = \begin{cases} R(\bot \leftarrow corps^-(r)) & \text{si } r \in CR \\ RNonApp(r) & \text{sinon} \end{cases}$$

On peut alors définir précisément les raisons des littéraux n'appartenant pas à $IN$ :

$$R_{\overline{IN}}(l_0) = \begin{cases} R_{OUT}(l_0) & \text{si } l_0 \in OUT \\ \bigcup_{r \in Rl_0} RNonDec(r) & \text{sinon} \end{cases}$$

## 4.2 Le backjumping

### 4.2.1 Principe général

En cas d'échec sur une branche, dont la raison est *raison_echec*, on revient sur le point de choix le plus

7. La contrainte n'est d'ailleurs pas satisfaite dans $\langle\{x\}, \{c, d\}\rangle$ (sinon la règle ne serait plus applicable, car bloquée) mais c'est une raison d'échec, comme nous l'avons expliqué dans la section précédente, et cela n'entre pas en cause dans la raison pour laquelle $r_1$ n'a pas été déclenchée.

8. En fait, il est possible que des règles de $Rl_0 \cap CR$ (bloquées par des contraintes) soient devenues non applicables (c'est le cas dès que la contrainte correspondante est satisfaite, ce qui entraîne que la règle est bloquée). Mais la règle ayant été bloquée par la contrainte *avant* d'être devenue non applicable, la première raison sera forcément « meilleure » que la seconde (elle permettra de remonter plus haut dans l'arbre).

récent participant à l'échec, i.e., on *backjumpe* au niveau $MAX(raison_echec)$ avec

$$MAX(S) = \begin{cases} x \in S \mid \forall y \in S, y \leq x & \text{si } S \neq \emptyset \\ 0 & \text{sinon} \end{cases}$$

Notons que lorsque l'échec est dû à une contradiction (suite à la phase de propagation), le point de choix courant est toujours en cause dans l'échec, le *backjumping* se ramène alors à un simple *backtracking.*

### 4.2.2 Combinaison des échecs

Considérons la situation où, en un point de choix donné $n$, les branches gauche et droite échouent et aucune d'elle, prise isolément, n'a permis de *backjumping* ; autrement dit, le niveau $n$ est incriminé dans l'échec de chaque branche. Dans ce cas, il faut revenir au point de choix le plus récent, hormis $n$, qui participe à l'un ou l'autre des échecs.

L'exemple de la figure 3 illustre ce point. Au point de choix n° 3, l'échec de la branche gauche provient de la contrainte $\bot \leftarrow arete(1,3), rouge(1), rouge(3)$ dont la raison est {0}, des littéraux $arete(1,3)$ de raison {0}, $rouge(1)$ de raison {1,0} et $rouge(3)$ de raison {3,0}, ce qui donne la raison {3,1,0}. L'échec de la branche droite provient de la contrainte $\bot \leftarrow arete(3,4), vert(3), vert(4)$ dont la raison est {0}, des littéraux $arete(3,4)$ de raison {0}, $vert(3)$ de raison {3,0} et $vert(4)$ de raison {0}, ce qui donne la raison {3,0}. Le point de choix le plus récent, hormis 3, participant à l'un au moins des deux échecs est 1, c'est le plus haut niveau auquel on peut revenir de façon sûre.

### 4.2.3 Algorithme avec backjumping

L'algorithme Solvebj reprend la fonction Solve de la section 3 en lui ajoutant un mécanisme de *backjumping.* Un nouveau paramètre, *niveau*, représente le numéro du point de choix courant, qui vaut 0 au premier appel. En cas d'échec, la fonction ne retourne plus seulement $faux$ mais lui adjoint la raison de l'échec. En cas de succès, elle retourne l'ensemble réponse trouvé (avec la raison ∅).

La phase de propagation est inchangée. Si une contradiction est détectée, la fonction retourne $faux$ et calcule la raison de la contradiction (comme indiqué dans la section 4.1.2). De même, lors de la phase de choix, s'il n'y a plus de règle de $P_R$ applicable mais qu'une contrainte est applicable (donc non satisfaite), la fonction retourne $faux$ et calcule la raison de l'échec (voir sections 4.1.2 et 4.1.3). Si une règle applicable de $P_R$ a été choisie, on crée un nouveau point de choix en incrémentant la variable *niveau.* On commence par forcer l'application de la règle. Si ce choix échoue, la raison de l'échec est récupérée dans la variable *Raison_g* qui est examinée pour savoir s'il faut explorer la branche droite ou si on peut remonter à un niveau inférieur. Si le niveau de *backjumping*, $MAX(Raison_g)$ (qui représente alors le niveau du point de choix le plus récent impliqué dans l'inconsistance), est inférieur au niveau courant *niveau*, alors on « saute » le point de choix courant et on remonte au choix précédent. Sinon le niveau courant est impliqué dans l'échec, on explore donc la branche droite du point de choix. Si celle-ci mène elle aussi à un échec, on récupère la raison *Raison_d* et on l'examine à son tour pour déterminer si le *backjumping* est possible. S'il n'est pas possible, cela signifie que le point de choix courant est impliqué dans les deux échecs, la raison globale de l'échec du nœud est alors l'union des deux raisons (sans le point de choix courant, déjà exploré).

**Function** Solvebj($P_R$,$P_K$,IN,OUT,CR,niveau)

```
répéter // phase de propagation
    r0 := γpro(PR ∪ PK, IN, OUT, CR);
    si r0 alors
        IN := IN ∪ {tête(r0)};
        CR := CR ∪ {r0};
jusqu'à ¬r0;
si IN ∩ OUT ≠ ∅ alors // contradiction
    retourner (faux,
    raison_contradiction(IN ∩ OUT))
sinon
    r0 := γcho(PR, IN, OUT, CR);
    si ¬r0 alors // plus de règle applicable
        si γcho(PK, IN, OUT, ∅) alors // contrainte
        non satisfaite
            retourner (faux,
            raison_contrainte_nonSat(PR, PK, IN,
            OUT, CR))
        sinon// ensemble réponse trouvé
            retourner (IN, ∅)
    sinon// point de choix
        niveau := niveau + 1;
        // branche gauche
        (stop, Raison_g) := Solvebj(PR, PK, IN,
        OUT ∪ corps⁻(r0), CR ∪ {r0}, niveau);
        si ¬stop alors
            si MAX(Raison_g) < niveau alors
            // backjump
                retourner (faux, Raison_g)
            sinon // branche droite
                (stop, Raison_d) := Solvebj(PR,
                PK ∪ {⊥ ← corps⁻(r0)}, IN, OUT,
                CR ∪ {r0}, niveau);
                si ¬stop alors
                    si MAX(Raison_d) < niveau alors
                    // backjump
                        retourner (faux, Raison_d)
                    sinon // combinaison des échecs
                        retourner (faux,
                        (Raison_g∪Raison_d)\{niveau})
                sinon
                    retourner (stop, ∅)
        sinon
            retourner (stop, ∅)
```

## 5 Éléments d'implémentation

La mise en œuvre du *backjumping* dans le solveur `ASPeRiX` est en cours d'implémentation. Nous en présentons quelques particularités dans cette section.

### 5.1 Calcul de raisons

Comme vu précédemment, il existe deux formes d'échec (cf. section 4.1.2) : une contradiction détectée suite à phase de propagation, et l'existence d'une contrainte non satisfaite une fois l'exploration d'une branche terminée. Dans les deux cas, plusieurs raisons peuvent être à l'origine d'un échec. Aucune raison n'est *a priori* meilleure qu'une autre, et se contenter de celle associée à la première cause trouvée est suffisant. Cependant, elles n'ont pas toutes le même impact sur le parcours de l'arbre de recherche : certaines permettent de « sauter » plus haut, i.e. d'élaguer plus de branches. La théorie voudrait donc que l'on détermine toutes les raisons d'un échec puis que l'on sélectionne la meilleure (celle qui rend le *backjumping* plus efficace) ; mais en pratique, ce calcul peut être très coûteux.

L'implémentation actuelle d'`ASPeRiX` s'arrête à la première contradiction ou à la première contrainte non satisfaite détectée. Il n'y a donc jamais plus d'un littéral contradictoire ni plus d'une contrainte non satisfaite connu.

La problématique est la même pour les littéraux neutralisant une règle (cf. section 4.1.3). En effet, pour une règle donnée $r$, il suffit qu'un littéral la neutralise pour qu'elle ne soit pas applicable, et ce indépendamment des autres (s'ils existent).

Pour des raisons de simplicité calculatoire, l'implémentation actuelle d'`ASPeRiX` se contente du premier littéral neutralisant trouvé, en commençant par ceux du corps négatif : puisqu'ils appartiennent aussi à $IN$, leur raison est forcément connue.

Ces choix d'implémentation mériteraient d'être étudiés plus avant en évaluant diverses stratégies (leur impact, leur coût) : est-il préférable de calculer toutes les raisons et de garder la meilleure, ou de se contenter d'en calculer une, voire quelques unes ?

### 5.2 Détermination des règles instanciées

Nous avons vu dans la section 4.1.3 que si un littéral $l_0$ n'a pas pu être prouvé, c'est parce qu'aucune règle de $Rl_0$ n'a été déclenchée. `ASPeRiX` travaillant directement sur un programme du premier ordre, l'ensemble des règles instanciées de tête $l_0$ (c'est-à-dire $Rl_0$) n'est pas connu *a priori* et doit donc être construit.

Un moyen simple de construire entièrement un tel ensemble consiste à effectuer une recherche en chaînage arrière. À cette fin, nous faisons appel à `Prolog` sur un programme défini construit à partir des règles du programme (à l'exception des contraintes) dont on a ôté le corps négatif[9]. On cherche alors à construire l'ensemble des règles du programme ASP telles que le littéral indéterminé est vrai dans le programme `Prolog` construit. Nous faisons alors appel à la primitive *setof* (qui calcule toutes les solutions d'un but) pour chaque règle dont la tête est le littéral.

**Exemple 5.1** *Soit le programme $P_{5.1}$ suivant :*

$$P_{5.1} = \left\{ \begin{array}{l} q(1,1) \leftarrow ., q(1,2) \leftarrow ., q(1,3) \leftarrow ., \\ q(2,1) \leftarrow ., q(3,2) \leftarrow ., \\ p(X,Y) \leftarrow q(X,Z), q(Z,Y), not\ p(Y,X)., \\ p(X,Y) \leftarrow r(X,V), s(Y,T), s(X,T)., \\ \bot \leftarrow q(X,X), p(Y,X). \end{array} \right\}$$

*Le programme passé à `Prolog` sera le suivant :*

$$P'_{5.1} = \left\{ \begin{array}{l} q(1,1)., q(1,2)., q(1,3)., \\ q(2,1)., q(3,2)., \\ p(X,Y)\ \text{:-}\ q(X,Z), q(Z,Y)., \\ p(X,Y)\ \text{:-}\ r(X,V), s(Y,T), s(X,T). \end{array} \right\}$$

*On cherche alors à construire l'ensemble des règles du programme $P_{5.1}$ telles que $p(1,2)$ est vrai dans le programme `Prolog` $P'_{5.1}$. Nous faisons alors appel à la primitive* setof *pour chaque règle dont la tête est $p(X,Y)$. Dans notre exemple, nous exécutons donc $setof([Z], (q(1,Z), q(Z,2)), I_1)$ et $setof([V,T], (r(1,V), s(2,T), s(1,T)), I_2)$. `Prolog` répondra $I_1 = [[1],[3]]$ et $I_2 = [\ ]$ ; ce qui représente les instanciations $\{[Z \leftarrow 1], [Z \leftarrow 3]\}$ pour $p(X,Y) \leftarrow q(X,Z), q(Z,Y), not\ p(Y,X).$, soit les instances $p(1,2) \leftarrow q(1,1), q(1,2), not\ p(2,1).$ et $p(1,2) \leftarrow q(1,3), q(3,2), not\ p(2,1).$ et l'absence d'instanciation pour la seconde règle.*

## 6 Conclusion

Nous avons présenté dans cet article l'adaptation des techniques de *backjumping* aux solveurs basés sur une approche par les règles. Le backjumping dans ce cadre se différencie des applications dans les autres domaines par le traitement sur des informations indéterminées au moment de l'échec dont il faut obtenir par ailleurs les raisons. Notre algorithme de *backjumping* est en cours d'implémentation au cœur d'`ASPeRiX`, un solveur basé sur une approche par les règles. Les résultats expérimentaux nous permettront de déterminer si l'introduction du *backjumping* offre des gains substantiels aussi bien en nombre de points de choix qu'en temps de calcul et si le surcoût en temps et en espace se révèle négligeable même lorsque le *backjumping* est inopérant. Nous envisageons d'explorer par la suite l'adaptation des techniques de *clause learning* aux solveurs basés sur une approche par les règles.

9. en l'état actuel, le traitement des boucles n'est pas implanté mais le sera via un mécanisme de tabulation [21]

## Références


[1] E. Erdem, F. Lin, and T. Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *LNCS*. Springer, 2009.

[2] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 345–351. Springer-Verlag, 2011.

[3] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http ://www.ijcai.org/papers07/contents.php.

[4] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[5] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4) :365–386, 1991.

[6] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *J. Autom. Reason.*, 36 :345–377, April 2006.

[7] C. Lefèvre and P. Nicolas. A first order forward chaining approach for answer set computing. In Erdem et al. [1], pages 196–208.

[8] C. Lefèvre and P. Nicolas. The first version of a new ASP solver : `ASPeRiX`. In Erdem et al. [1], pages 522–527.

[9] 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*, 7(3) :499–562, 2006.

[10] Fangzhen Lin and Yuting Zhao. Assat : computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157 :115–137, August 2004.

[11] M. Maratea, F. Ricca, W. Faber, and N. Leone. Look-back techniques and heuristics in DLV : Implementation, evaluation, and comparison to QBF solvers. *Journal of Algorithms*, 63(1-3) :70–89, 2008.

[12] J.P. Marques-Silva and K.Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Confenrence on Computer-Aided Design (ICCAD'96)*, pages 220–227, 1996.

[13] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429, 1997.

[14] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In P. Hill and D. Warren, editors, *ICLP*, volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[15] F. Ricca, W. Faber, and N. Leone. A backjumping technique for Disjunctive Logic Programming. *AI Communications*, 19(2) :155–172, 2006.

[16] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2) :181–234, 2002.

[17] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2) :135–196, 1977.

[18] T. Syrjänen. Implementation of local grounding for logic programs for stable model semantics. Technical report, Helsinki University of Technology, 1998.

[19] J. Ward. Answer Set Programming with Clause Learning, PhD thesis, ohio state university. 2004.

[20] J. Ward and J.S. Schlipf. Answer Set Programming with Clause Learning. In *Proceedings of the 7th Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, pages 302–313, 2004.

[21] David S. Warren. Memoing for logic programs. *Commun. ACM*, 35 :93–111, March 1992.