Amélioration de l'efficacité du pattern matching dans le langage à base de règles BOOJUM‡

Jean-Luc DORMOY
Direction des Etudes et Recherches d'Electricité de France
1, avenue du Général de Gaulle
92141 Clamart CEDEX
FRANCE
&
LAFORIA Université Paris 6

Résumé

BOOJUM [Dormoy, 1986, 1987] est un langage à base de règles d'ordres 0, 1 et 2 et le moteur d'inférences associé. Inspiré à l'origine de SNARK [Laurière & Vialatte, 1985] [Vialatte, 1985], il se distingue des langages de même type par une amélioration notable de l'efficacité de l'algorithme de "pattern matching".
La recherche des instanciations n'est effectuée que pour une règle à la fois. On considère que l'on a à chaque examen de règle un problème combinatoire de satisfaction de contraintes à résoudre. On procède avant chaque choix à une propagation des contraintes plus ou moins poussée puis à une estimation du problème le plus contraint. Une batterie d'heuristiques accélère le choix. Deux algorithmes de propagation de contraintes sont proposés, dont l'un a une complexité démontrée optimale [Mohr & Henderson, 1986] (cet algorithme était déjà présent dans ALICE [Laurière, 1976, 1978]).
Deux autres aspects ont été particulièrement étudiés:

Les accès à la base de faits: Le moteur construit à la compilation des règles un filtre de mots-clés, qui sert à un filtrage des règles à examiner, mais aussi à déterminer les index sur la base de faits permettant d'accélérer le pattern matching. Deux bases de règles, même apparemment proches, pourront nécessiter des index très différents.

Les problèmes d'auto-récursivité: Il s'agit de la situation où une règle, en se déclenchant, modifie son ensemble d'instanciations. Un algorithme fin évite les remontées inutiles dans l'arbre des choix.

Un tableau des performances sur différents types de matériels est fourni.

Son efficacité permet à BOOJUM d'être actuellement utilisé à E.D.F. dans plus de 50 applications, notamment en génération automatique de programme, gestion et interrogation intelligentes de bases de données, supervision de codes de calcul.

‡ BOOJUM a constitué la version 0 du générateur de systèmes experts GENESIA II commercialisé par la société STERIA.

1 Introduction

BOOJUM est un langage à base de règles - en anglais *production system* - d'ordres 0, 1 et 2, dont le moteur d'inférence associé fonctionne en chaînage avant. La principale difficulté pour mettre en oeuvre ce genre de programme réside dans *l'algorithme de pattern matching*, ou *algorithme d'instanciation*, qui doit trouver, au vu de la base de faits courante, les substitutions valables des variables d'une règle.
D'un côté, on a tenté d'apporter des solutions propres à ce problème, cf. les nombreux travaux autour de l'algorithme basé sur le réseau RETE [Forgy, 1982]. De l'autre, de nombreux et importants travaux de recherche ont été consacrés à trouver des méthodes générales pour résoudre des *problèmes de satisfaction de contraintes*, qui consistent à trouver les assignations correctes à des variables soumises à un ensemble de contraintes. On peut citer à ce sujet le langage ALICE de Jean-Louis Laurière [1976], qui est un résolveur général de problèmes combinatoires, et les nombreux travaux sur les méthodes de *propagation de contraintes*. Or il s'avère que, si l'on prend le contre-pied de ce qui est à la base de l'algorithme basé sur RETE, les deux problèmes se rejoignent: trouver les instanciations d'une règle, c'est résoudre le problème combinatoire qui consiste à trouver les "assignations de valeurs" (les substitutions) des variables de la règle soumises à un "ensemble de contraintes" (les prémisses de la règle). Le système SNARK [Laurière & Vialatte, 1985] [Vialatte, 1985] est le premier à être basé sur les méthodes issues de la résolution de problèmes de satisfaction de contraintes.
Nous montrons dans ce papier comment nous avons systématisé la mise en oeuvre de ces méthodes dans BOOJUM, et les gains que l'on en tire. Après une description sommaire du langage, nous proposons deux algorithmes d'instanciation, basés sur les notions de *choix dynamique* et de *propagation de contraintes*. Pratiquement, ces algorithmes sont accompagnés de la mise en oeuvre d'*heuristiques*. L'efficacité de ces algorithmes passe par une structuration de la base de faits. Un système de *mots-clés*, qui servent à la fois au filtrage des règles candidates et à cette structuration est mis en oeuvre. On montre enfin comment sont abordés les problèmes de *récursivité* entre règles différentes ou d'une règle avec elle-même.
BOOJUM était à l'origine un logiciel de recherche. Il s'avère être maintenant utilisé dans plus de 50 applications d'ampleur dans des domaines très divers à la Direction des Etudes et Recherches d'EDF. Cela a permis de largement tester les idées exposées ici. Un tableau des performances sur des matériels variés est donné.

2 Description du langage

Le langage BOOJUM est une extension des langages qui lui sont comparables, notamment en ce qui concerne l'existence de *faits parlant de faits* en base de faits *(triplets emboîtés)* et la possibilité d'avoir des prémisses entièrement variables *(ordre 2)*. Ces possiblités nouvelles offrent des avantages essentiels pour la représentation des connaissances. Nous nous limitons cependant ici à une description succinte du langage.

2.1 La base de faits

La base de faits est, *comme dans un moteur 0*$^+$, constituée de couples:

<Fait> ::= *<Objet> ' <Objet>*

Le second objet est la *valeur*, que l'on appelle en BOOJUM le *statut* du premier, lequel est appelé un *attribut* ou une *entité*.

Un objet est défini par:

<Objet> ::= *<Nombre>*
<Objet atomique>
(*<Objet> <Objet> <Objet>****)***

<Nombre> se réfère à une chaîne de caractères ayant la syntaxe d'un nombre, entier ou réel, par exemples:

```
203    -54.231              0.564E-6
```

<Objet atomique> se réfère a toute chaîne de caractères n'ayant pas la syntaxe d'un nombre, par exemples:

```
Pression_Circuit_Primaire        "Ceci est un objet"
Est_Pere_de         Socrate      ==>
```

Si la définition d'un objet se limitait aux deux premières lignes, on aurait, grosso modo, un moteur 0$^+$. La dernière partie de la définition permet d'aller au-delà:

```
(Toto Est_Pere_de Lulu)
(Zonzon Sait (Toto Est_Pere_de Lulu))
(((P Sait A) Et (P Sait (A ==> B))) --> (P Sait B))
```

sont des objets.

Quelques exemples de faits:

```
Pression_Circuit_Primaire ' 2.352E2
(Toto Est_Pere_de Lulu) ' VRAI
```

ce qui, par convention, peut aussi s'écrire:

```
Toto Est_Pere_de Lulu ' VRAI
```

ou même

```
Toto Est_Pere_de Lulu
```

(`VRAI` est le statut par défaut d'un objet).

Autre exemple:

```
Zonzon Sait (Toto Est_Pere_de Lulu) 'FAUX
```

Ce fait affirme que `Zonzon` ne sait pas que `Toto` est le père de `Lulu`, lequel fait était affirmé précédemment.

Bien que cela ne soit pas possible ici, on peut montrer [Dormoy, 1987] que cette syntaxe de la base de faits *généralise* les frames, relations binaires, etc, couramment utilisés, et s'approche d'une représentation des relations d'arité quelconque.

Citons enfin deux statuts spéciaux: tout nombre a implicitement le statut `NOMBRE`, et un objet ayant le statut `INEXISTANT` est ... inexistant, au même titre que les objets non mentionnés dans la base de connaissances (mais on peut en parler).

2.2 La base de règles

Une règle a, classiquement, la forme suivante:

`REGLE` *<Nom de la règle>*
`SI` *<Prémisses>*
`ALORS` *<Conséquents>*

Les prémisses peuvent être essentiellement de trois formes différentes:

<Terme> ' <Comparateur> <Terme>
<Terme> <Terme> <Terme> ' <Terme> :<Terme>
<Terme> <Comparateur> <Terme>

La première forme correspond à la définition élémentaire d'un fait. La seconde prend en compte les triplets. Le *:<Terme>* est généralement une variable instanciée par l'objet-triplet qui précède. La troisième forme est une simple comparaison d'objets. Un*<Terme>* correspond soit à une variable, soit à un objet. On peut mettre des variables à n'importe quelle place.

Un exemple:

```
REGLE Modus_Ponens_du_Savoir
SI    (P) Sait (I)
      (A) ==> (B) :(I)
      (P) Sait (A)
ALORS
      (P) Sait (B)
```

`(P)`, `(I)`, etc sont des variables.

Remarque: on peut aussi par convention d'écriture écrire cette règle:

```
REGLE Modus_Ponens_du_Savoir
SI    (P) Sait ((A) ==> (B))
      (P) Sait (A)
ALORS
      (P) Sait (B)
```

2.3 L'inférence

L'inférence se fait classiquement en chaînage avant jusqu'à saturation de la base de faits. Plusieurs conditions d'arrêt sont possibles, et paramètrables pour chaque règle par l'utilisateur.

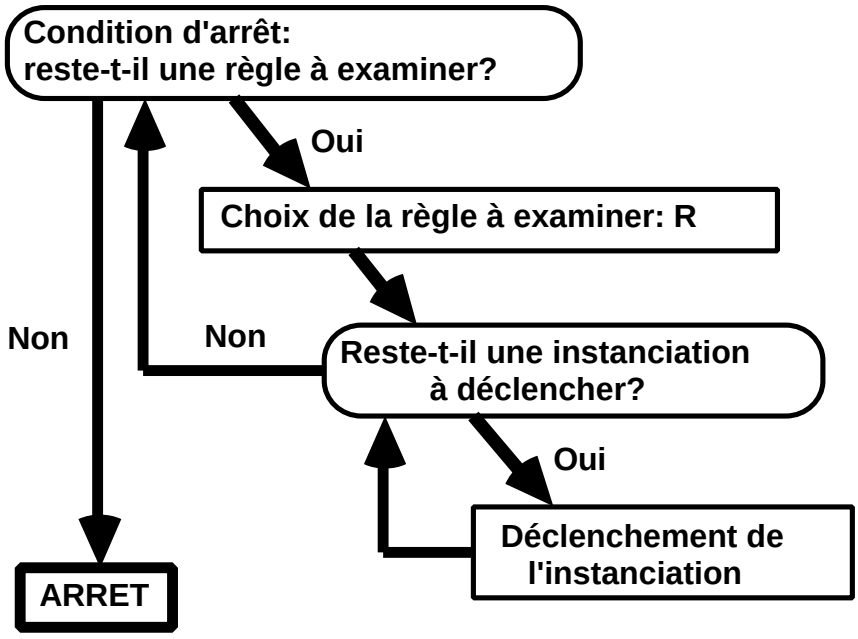

Figure 1: Mécanisme d'inférence

Le choix de la règle à appliquer est fait par BOOJUM selon une condition très simple: il choisit d'appliquer la première règle applicable dans la liste des règles. On a évité tout algorithme de choix compliqué, car il s'avère à l'expérience qu'il est impossible d'obtenir un critère algorithmique de choix qui choisisse effectivement la "bonne" règle (cela nécessiterait une masse importante de connaissances sur les connaissances à traiter - des méta-connaissances - que ne peut avoir un moteur d'inférence [Pitrat, 1985]). Ce critère permet également à l'utilisateur de conserver un minimum la maîtrise de l'inférence. Il n'a cependant pas la possibilité de modifier l'ordre des règles en cours d'inférence: cette contrainte est là pour lui éviter de retomber dans un style de programmation procédural.
Plus précisément, le schéma du cycle d'inférence est celui de la figure 1.

3 Principes de l'algorithme d'instanciation

3.1 Examiner une règle à la fois

Le principe du mécanisme d'inférence de BOOJUM est de calculer les instanciations d'une seule règle à la fois. Cela est très différent de ce qui est à la base de nombreux algorithmes de pattern matching, notamment celui basé sur le réseau RETE [Forgy, 1982] décrit dès 1979 par Forgy. Celui-ci était justifié par la combinaison d'une contrainte sur le mécanisme d'inférence que se sont donnés les auteurs d'OPS et d'une constatation sur le déroulement pratique de l'inférence:

© on veut que l'ensemble de toutes les instanciations de toutes les règles - appelé l'ensemble des conflits - soit déterminé avant chaque déclenchement de règle pour lui appliquer un critère de choix de l'instanciation à déclencher.

© le déclenchement d'une règle modifie - probablement - peu de faits par rapport à l'ensemble de la base de faits, et donc - probablement - peu de règles sont concernées par ces changements. On peut donc espérer que l'application d'une règle modifie peu l'ensemble de conflits, et donc qu'une bonne part du travail effectué au cycle précédent ne soit pas à refaire.

Comme cela a été souligné précédemment, il s'avère que le moteur n'a pas les éléments nécessaires pour faire un "bon choix de règle". Il est donc préférable de fournir à l'utillisateur un critère de choix de la règle à appliquer très simple - et cela évite bien des perversions - ce qui permet en retour de ne chercher les instanciations que pour la règle choisie.
Le deuxième argument tombe également en défaut dans la pratique. Il s'avère en effet fréquemment que le déclenchement d'une règle interdit quelques instanciations d'autres règles encore non choisies. Supposons pour fixer les idées qu'en moyenne chaque déclenchement de règle détruise 5 autres instanciations auparavant valables. On aura alors calculé 500 instanciations inutilement pour seulement 100 déclenchements. La stratégie "paresseuse" qui consiste à ne calculer à chaque instant *qu'une* instanciation aboutit à une meilleure efficacité. Par ailleurs, il est intéressant, ayant commencé l'examen d'une règle, d'épuiser ses instanciations.

3.2 Représentations graphiques du problème du pattern matching

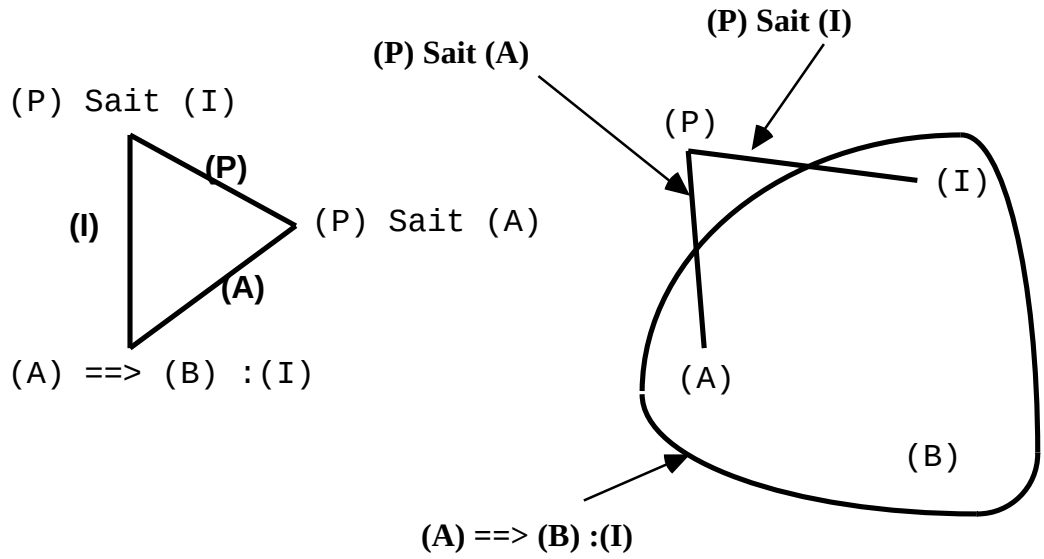

Figure 2: Représentations prémisse et variable

Deux représentations graphique permettent de visualiser les opérations accomplies par un algorithme de pattern matching et de mieux juger de son efficacité potentielle: la représentation-prémisse, et la représentation-variable duale. Dans chacune des deux représentations, on construit à partir des prémisses d'une règle un graphe (en général un hypergraphe):

© dans le premier cas, on associe à chaque prémisse un noeud, et à chaque variable une - hyper - arête reliant les prémisses qui la contiennent.

© dans le second, on associe à chaque variable un noeud, et à chaque prémisse une - hyper - arête reliant les variables qu'elle contient.

Les arêtes pourront être labellées par, respectivement, les variables et les prémisses.
Par exemple, la règle:

```
REGLE Modus_Ponens_du_Savoir
SI    (P) Sait (I)
      (A) ==> (B) :(I)
      (P) Sait (A)
ALORS
      (P) Sait (B)
```

admet les représentations-prémisse et -variable de la figure 2.

3.3 Instancier les variables, pas les prémisses

> Notations valables dans la suite quant aux notions manipulées:
> ce qui commence par un P sera une prémisse de la règle en examen
> ce qui commence par un f sera un fait (pouvant instancier une prémisse)
> ce qui commence par un X, Y, Z sera une variable de la règle en examen
> ce qui commence par un x, y, z sera un objet (pouvant instancier une variable)

La plupart des algorithmes de pattern matching proposés à ce jour sont basés sur la représentation-prémisse. On y cherche d'une manière ou d'une autre les *faits* satisfaisant les prémisses considérées, puis on considère les liens créés par les occurences de mêmes variables dans les différentes prémisses. On manipule donc en cours d'algorithme pour

chaque prémisse P un ensemble F(P) de faits retenus comme instanciations potentielles de P et tel que, pour toute instanciation globale de la règle, P soit instanciée par un fait de F(P). On appellera F(P) un *domaine d'instanciation* de la prémisse P. Un tel F(P) est par exemple l'ensemble de tous les faits matchant la prémisse P. Par ailleurs, ce type d'algorithme manipule à un moment donné des *instanciations partielles*: étant donnés $P_1,...,P_k$ des prémisses d'une règle et $f_1,...,f_k$ des faits de $F(P_1),...,F(P_k)$, l'ensemble des couples $\{(P_1,f_1),...,(P_k,f_k)\}$ est appelé une *instanciation partielle* de la règle ssi les variables qui s'avèrent être instanciées le sont par un même objet dans les différentes prémisses parmi $P_1,...,P_k$ où elles apparaissent.

La représentation-variable nous suggère une approche différente. Au lieu de déterminer les faits satisfaisant chaque prémisse, on pourrait déterminer les objets se substituant de manière correcte aux variables. Cela signifie que l'on associe à chaque variable X un ensemble I(X) d'objets tel que, pour toute instanciation globale de la règle, l'instanciation de X correspondante se retrouve dans I(X). On appellera I(X) un *domaine d'instanciation* de la variable X. Un tel I(X) pourra être l'ensemble $I_P(X)=P_{P,X}(F(P))$ des projections de F(P) sur la place qu'occupe X dans P, où P est une prémisse quelconque contenant X. Par exemple, si l'on considère la prémisse P=P(X,Y) et l'ensemble F(P) égal à $F(P)=\{P(x_1,y_1),P(x_1,y_2),P(x_2,y_1),P(x_3,y_1)\}$, on a

$$P_{P(X,Y),X}(F(P))=\{x_1,x_2,x_3\} \text{ et } P_{P(X,Y),Y}(F(P))=\{y_1,y_2\}.$$

De même $I_0(X)=$ $X\in P$

L'algorithme manipule là aussi des *instanciations partielles*, mais avec un autre sens que pour la représentation-prémisse:

© soit $P(X_1,...,X_p)$ une prémisse contenant exactement les variables $X_1,...,X_p$ (cette notation signifie seulement que P contient ces variables, mais n'a pas de signification quant à la syntaxe des règles et des faits). Etant donnés $h\in[0,p]$ et $x_1,...,x_h$ des objets de $I(X_1),...,I(X_h)$, l'ensemble des couples $\{(X_1,x_1),...,(X_h,x_h)\}$ est appelé une *instanciation partielle* de la prémisse P ssi il existe des objets $x_{h+1},...,x_p$ de $I(X_{h+1}),...,I(X_p)$ tels que $P(x_1,...,x_p)$ est satisfaite.

© soient $X_1,...,X_n$ des variables des prémisses d'une même règle, et $x_1,...,x_n$ des objets de $I(X_1),...,I(X_n)$; l'ensemble des couples $\{(X_1,x_1),...,(X_n,x_n)\}$ est appelé une *instanciation partielle* de la règle ssi il s'agit d'une instanciation partielle pour toutes les prémisses de la règle. (Cela implique en particulier que toutes les prémisses sans variable sont satisfaites, et que les prémisses contenant au moins une variable, mais dont aucune n'est instanciée, sont satisfaites par au moins un fait de la base de faits).

Il pourrait sembler à première vue que les deux points de vue sont équivalents. Le petit exemple qui suit montre qu'il n'en est rien. Soit une règle dont les prémisses sont P(X,Y) et Q(X,Z). Supposons qu'il existe dans la base de faits courante 1000 faits de la forme $P(a,b_i)$ et 2000 faits de la forme $Q(a_i,c_i)$, mais tels qu'aucun a_i ne soit égal à a. Il est immédiat que la règle n'a aucune instanciation possible. Or, ses deux prémisses ont respectivement 1000 et 2000 faits les matchant, et un algorithme basé sur la représentation-prémisse retrouvera donc au minimum 1000 fois qu'il n'existe pas de fait de la forme Q(a,z). Par contre, un algorithme basé sur la représentation-variable pourra trouver (s'il est bien conçu) immédiatement que I(X) doit se réduire à {a} à cause de la première

prémisse, puis qu'il n'existe aucun fait de la forme $Q(a,z)$, donc que $I(X)$ est vide, et donc que la règle n'a aucune instanciation possible.
Pour montrer que l'exemple précédent a une portée générale, considérons l'ensemble des projections des faits de $F(P)$ vérifiant une prémisse $P(X_1,...,X_p)$ sur ses variables X_i, que nous noterons $P_i(F(P))$. Si l'on doit vérifier les liens des instanciations potentielles de $F(P)$ avec d'autres prémisses contenant des variables parmi $X_1,...,X_p$, il n'est pas plus coûteux (sous certaines conditions sur l'indexation des faits, cf. §4) de faire les vérifications variable après variable à la manière d'un backtracking brutal, et ce quelque soit l'ordre de parcours des variables. Autrement dit nous remplaçons le programme $Verif_{Prem}$

Pour tout $f \in F(P)$
faire la vérification des liens sur $(X_1,...,X_p)$

par le programme $Verif_{Var}$

```
i ← 1
tant que i ≠ 0 faire
      s'il existe x_i ∈ P_i(F(P)) encore non examiné
            choisir x_i comme instanciation de X_i
            faire la vérification des liens sur X_i
            si le test est un succès alors
                  i ← i + 1
            sinon  i ← i - 1
      sinon  i ← i - 1
```

3.4 Faire des choix dynamiques et limiter les domaines d'instanciation

Ce qui précède montre *qu'il vaut mieux instancier les variables plutôt que les prémisses*. Nous adoptons donc - provisoirement - la procédure $Verif_{Var}$. On peut cependant améliorer son efficacité en jouant sur deux degrés de liberté:

Faire des choix dynamiques: L'ordre d'examen des variables est arbitraire. Un *bon ordre, déterminé de manière heuristique*, doit exister qui diminue le parcours de l'espace de recherche.

Fixer un *bon ordre* d'examen des variables d'une règle peut se faire de différentes manières. On peut chercher un *bon ordre a priori*. Cela signifie que les variables sont ordonnées une fois pour toutes (par exemple dès la phase de *compilation* de la base de règle), et que c'est cet ordre qui est pris en compte dans la procédure $Verif_{Var}$. C'est ce qu'on peut appeler un *choix statique*. Mais on peut aussi effectuer un *choix dynamique*: on choisit *à chaque étape* la variable que l'on va instancier parmi celles qui ne le sont pas encore. Par ailleurs, on n'est nullement obligé d'instancier toutes les variables d'une même prémisse avant de passer à une autre prémisse. Considérons par exemple la règle

```
REGLE Rxxx
SI P(X,Y) et Q(X,Z) et ... ALORS ...
```

et la partie de base de faits concernant P et Q

```
P(x1,y1)  P(x1,y2)          P(x1,y3)          P(x1,y4)
P(x2,y1)  P(x2,y5)
Q(x1,z1)  Q(x1,z2)     Q(x1,z3)
Q(x2,z2)  Q(x2,z4)          Q(x2,z5)
```

Supposons que X soit choisie en premier. Pour le choix X/x_1, on a intérêt à choisir Z plutôt que Y, puisque Z a trois instanciations possibles et Y quatre. Par contre, pour le choix X/x_2, il vaut mieux faire le choix inverse.

Ainsi, *le choix à l'étape* i *dépend du choix à l'étape* $i-1$. En général, le choix se fait selon le critère du *sous-problème le plus contraint:* si les variables $Y_1,...,Y_k$ restent à instancier, on choisit la variable dont une *estimation* du nombre d'instanciations est minimum. L'algorithme dépend évidemment de cette *estimation*.

Propager le choix: Il n'est pas nécessaire lorsque la variable $X_i=X$ est choisie à l'étape i de parcourir tout l'ensemble $I_P(X)$ (où P est une prémisse quelconque contenant X) ou $I_0(X)$: il est par exemple inutile, si $X_1/x_1,...,X_{i-1}/x_{i-1}$ sont les choix déjà effectués, d'examiner les $x_i=x \in I_P(X)$ tels que $\{(X_1,x_1),...,(X_i,x_i)\}$ n'est pas une instanciation partielle de la règle. Si l'on reprend l'exemple précédent, et si l'on suppose que nous en sommes à l'étape 2, que le choix précédent est X/x_1, et que Z est choisie, alors il est inutile d'examiner z_4 et z_5 qui appartiennent à $I_{Q(X,Z)}(Z)$ et qui peuvent appartenir à $I_0(Z)$ si d'éventuelles prémisses contenant Z non mentionnées ne l'interdisent pas. On n'examine donc à chaque étape qu'un *sous-ensemble* de $I_P(X)$. Restreindre ce sous-ensemble peut permettre un nombre de choix bien moindre.

En définitive, l'algorithme d'instanciation revient à un échange entre deux tâches:

© choisir une variable X ayant une estimation $e(X)$ du nombre de ses instanciations minimum,

© construire et restreindre par un certain algorithme de propagation Propag, et en tenant compte des choix précédents, le domaine d'instanciation $I(X)$ de la variable X choisie.

Il est clair que l'efficacité résultante dépend de la définition précise de chacune de ces tâches. Remarquons qu'elles peuvent *s'améliorer mutuellement*: si le choix est bon, la phase de propagation pourra éliminer rapidement les choix potentiels inconsistents, et si la propagation est bonne, l'estimation permettant de faire le choix suivant n'en sera que meilleure.

Nous présentons maintenant deux réalisations possibles de ce schéma d'algorithme (toutes deux mises en oeuvre dans BOOJUM).

3.5 "Premier moteur"

Le *premier moteur* est le premier exemple que nous donnons d'algorithme d'instanciation basé sur le couple choix dynamique/propagation.

Considérons une règle dont les variables $X_1,...,X_k$ sont déjà instanciées par $x_1,...,x_k$, et dont $X_{k+1},...,X_n$ restent à instancier. Supposons que $\{(X_1,x_1),...,(X_k,x_k)\}$ est une instanciation partielle de la règle: l'algorithme que nous allons décrire, baptisé *premier moteur*, sera tel que cette condition sera toujours satisfaite.

Considérons une variable X parmi celles non instanciées, et une prémisse P contenant X. P peut contenir d'autres variables que X, certaines parmi elles étant déjà instanciées. Notons $I_{k,P}(X)$ l'ensemble des objets qui, s'ils sont substitués à X, fournissent avec les choix déjà effectués une instanciation partielle de P. Il est clair que X ne peut avoir d'autre instanciation, sous les choix déjà effectués, en dehors de cet ensemble.

Le *premier moteur* utilise l'estimation

$$e(X) = \min_{X \in P} (l_{k,P}(X))$$

Autrement dit, on choisit le couple (variable, prémisse) (X,P), où X apparaît dans P, tel que le nombre d'instanciations de X dans P prise isolément, mais en tenant compte des choix déjà faits, est minimum.
La procédure de propagation `Propag` consiste à définir pour la variable $X_{k+1}=X$ choisie le domaine d'instanciation $l_{k+1}(X) = \bigcap_{\Xi \in P}$

Il est clair que, pour tout x_{k+1} de $l_{k+1}(X)$, l'ensemble $\{(X_1,x_1),...,(X_{k+1},x_{k+1})\}$ est une instanciation partielle de la règle (et la condition que l'on prétendait ci-dessus vérifiée l'est).

3.6 Propager les contraintes : "second moteur"

L'algorithme du premier moteur est déjà une stratégie complexe et "intelligente": avant d'effectuer toute tâche, le moteur *"réfléchit"* pour savoir comment s'y prendre au mieux. Cependant, il existe des problèmes plus complexes où le type de choix ou de propagation effectuée par le premier moteur s'avère inefficace.
Considèrons par exemple le problème des n reines, résolu par une règle dont les n(n+1)/2 prémisses ont la forme

$\text{Colonne Reine}_i = (C_i)$, avec $1 \le i \le n$

(qui signifie que l'on place la reine de la ligne i sur la colonne C_i)

et $\text{En\_Prise}(C_i) <> (C_j)$, avec $1 \le i < j \le n$

(qui signifie que les reines des lignes i et j ne doivent pas être en prise - la relation `En_Prise` est décrite en base de faits)

L'estimation par le premier moteur du nombre d'instanciations d'une variable *pris prémisse par prémisse* sera toujours n. Autrement dit, la procédure de choix fournit la même estimation pour tous les choix possibles. Cela est du au fait que le nombre potentiel d'instanciations $e(X)$ d'une variable X est calculé *prémisse par prémisse, sans tenir compte des liens existant entre les variables encore non instanciées*. (Il faut cependant remarquer que la procédure Propag revient à interdire de placer une reine en prise avec celles déjà placées, et ce quelque soit l'ordre dans lequel les prémisses ont été écrites: il fait automatiquement de *l'évaluation paresseuse*).
Le *second moteur* permet de résoudre ce problème. Il utilise une méthode de propagation de contraintes, appelée *consistance des arcs*, qui consiste à construire à la première étape, puis à restreindre après chaque choix, un domaine d'instanciation $l_1(X)$ de telle sorte que la condition **(C)** soit vérifiée:

> Pour toute variable X apparaissant dans une prémisse P,
> pour tout x de $l_1(X)$,
> $\{(X,x)\}$ est une instanciation partielle de P.

On a à chaque étape $X \in P$

L'algorithme utilisé est une adaptation de celui décrit dans [Mohr & Henderson, 1986], généralisé par Mohr [Mohr & Masini, 1988] (voir aussi [Macworth & Freuder ,1985] et [Montanari, 1974]):

1) (Initialisation). Au début de l'examen de la règle, marquer tous les couples (X,P), où X est une variable apparaissant dans la prémisse P. Après un choix sur la variable X, marquer tous les couples (Y,P), où Y est une variable différente de X et P une prémisse quelconque où apparaissent à la fois X et P.

2) Choisir un couple (X,P) marqué. S'il n'y en a pas, **STOP**.

3) Enlever la marque du couple (X,P).

4) Faire en sorte que la condition **(C)** soit vérifiée pour le couple (X,P), c'est-à-dire enlever les éléments x de I_1(X) tels que {(X,x)} n'est pas une instanciation partielle de P.

5) Si l'étape **4** a permis d'enlever au moins un élément de I_1(X), marquer tous les couples (Y,Q) tels que Y est une variable différente de X et Q une prémisse différente de P où apparaissent à la fois X et Y. Aller en **2**.

Mohr et Henderson [op. cit.] ont prouvé l'optimalité de cet algorithme. Pratiquement, il permet des gains de temps d'autant plus significatifs que le problème est plus "difficile". Il a par ailleurs l'avantage de non seulement restreindre l'espace de recherche, mais aussi *d'améliorer la qualité de l'heuristique de choix* (le phénomène *d'amélioration mutuelle* dont il est question ci-dessus). Le problème des n reines en est un exemple: on choisira à chaque étape la ligne où il reste le moins de cases possibles où placer légalement une reine.

3.7 Limiter les choix

Faire des choix revient à *"réfléchir" sur la meilleure manière de résoudre le problème posé* plutôt que de le résoudre directement. De ce point de vue, le *second moteur* "réfléchit" plus que le *premier*. Il est clair cependant que passer du temps à la réflexion n'est intéressant que si cela en épargne pour la résolution du problème posé. Ce n'est pas le cas pour certains problèmes: ils sont trop simples, ou, bien plus souvent, la structure du problème permet de définir par avance comment les opérations seront effectuées au mieux. Le *premier moteur*, et plus encore le *second*, sont donc trop complexes pour certaines tâches. Les limitations des choix qui suivent montrent comment, dans certains cas, toute estimation de choix peut être évitée.

3.7.1 Variables libres

Une variable n'apparaissant que dans une prémisse *ne contraint pas* les instanciations des autres variables, pourvu que l'algorithme choisi n'utilise que des instanciations partielles. Elle n'intervient donc pas dans le choix. On dira qu'il s'agit d'une *variable libre*. En conséquence, les algorithmes d'instanciation ci-dessus ne cherchent que des *instanciations partielles des variables contraintes*. Considérons par exemple la règle

```
REGLE Rxxx
SI P(X,Y)
   Q(X,Z)
   R(Z,T)
ALORS ...
```

Les variables Y et T sont libres. Il suffit pour instancier la règle de déterminer les couples (x,z) tels que {(X,x),(Z,z)} soit une instanciation partielle de la règle. On est alors sûr de trouver au moins une substitution pour Y et T qui complète l'instanciation. Par ailleurs, il suffit pour compléter cette instanciation partielle d'instancier Y et T par n'importe quelle substitution valable dans la prémisse où elles apparaissent. Cela constitue

un autre avantage de la détection des variables libres: après avoir déterminé une instanciation partielle des variables contraintes, et *lorsqu'il n'y a pas deux variables libres apparaissant dans la même prémisse*, il suffit de dresser la liste des substitutions valables des variables libres, puis d'adjoindre à l'instanciation partielle des variables contraintes n'importe quelle substitution des variables libres.

3.7.2 Variables prioritaires

Les variables libres sont considérées *en dernier lieu*. Les variables prioritaires vont l'être en premier.
Une variable prioritaire est une variable qui, pour des raisons de structure de la règle, ne peut avoir au plus qu'une substitution possible. Par exemple dans le langage de BOOJUM, la variable `(N)` de la prémisse `Compteur' = (N)` est prioritaire: par définition de la syntaxe du langage, l'objet `Compteur`, comme tout objet, a au plus un statut. Ces variables seront donc regardées en priorité à toutes les autres.
Par ailleurs, il peut arriver que certains choix rendent une variable prioritaire en cours d'instanciation. Par exemple, la variable `(S)` de la prémisse `(O)' = (S)` est prioritaire dès que `(O)` est substituée. Il est alors inutile d'appeler la procédure de choix: il faut choisir `(S)`.
Les variables peuvent se rendre prioritaires récursivement. A la limite, il peut être inutile de faire tout choix, par exemple pour la règle:

```
REGLE Rxxx
SI Element_de_Pile' = (Pile)
   (Car1) Liste (Cdr1) :(Pile)
   (Car2) Liste (Cdr2) :(Cdr1)
   (Car2) P a
ALORS ...
```

L'ordre optimal d'examen des variables est `(Pile)`, `(Cdr1)`, `(Car2)` (les autres variables sont libres). Cet ordre est déterminé à la compilation des règles.

3.7.3 Prémisse négatives

Les prémisses négatives sont des prémisses ayant a priori une infinité d'instanciations potentielles (prémisse du type `(X) <> (Y)`, prémisse niant la présence d'un fait en base de faits,...). Il est inutile:

© pour le *premier moteur* de faire entrer ces prémisses dans l'évaluation du couple (variable,prémisse) le plus contraint.

© en général, de propager les choix dans ces prémisses *tant qu'elles ne sont pas complètement instanciées*. Par contre, il faut propager en priorité dans ces prémisses dès qu'elles le sont.

4 Les accès à la base de faits

Les algorithmes d'instanciation basés sur la représentation-prémisse possèdent tous peu ou prou un *filtre* et un *système d'indexation*. Le *filtre* est l'ensemble des *patterns* ou *mots-clé* des prémisses - c'est-à-dire des objets qui caractérisent la forme des prémisses sans tenir compte des noms des variables - qui permet lorsqu'un fait est déduit ou supprimé d'agir sur les règles concernées. Le *système d'indexation* est un moyen de mémorisation permettant de retrouver rapidement les faits satisfaisant un pattern. (Ces deux notions sont pratiquement confondues dans OPS5).

4.1 Le filtre des mots-clés

BOOJUM ne manque pas à la règle: il possède un filtre de mots-clés. Toute modification de la base de faits "passe" par ce filtre, et agit sur les règles qu'elle concerne. Outre le regain d'efficacité qu'il procure, le filtre est indispensable à nos algorithmes d'instanciation, puisqu'ils exigent avant d'examiner une règle que chacune de ses prémisses prise à part possède au moins une instanciation (plus formellement, l'instanciation vide doit être une instanciation partielle de la règle). Il permet donc d'interdire l'examen d'une telle règle. Il permet aussi de réactiver des règles déjà examinées qui, par une modification de la base de faits, peuvent avoir de nouvelles instanciations.

4.2 Le système d'indexation

Avoir basé nos algorithmes d'instanciation sur la représentation-variable impose un système d'indexation très différent de ceux communément adoptés pour la représentation-prémisse, où les systèmes les plus évolués se contentaient en général d'associer à chaque pattern l'ensemble des faits courants le satisfaisant.
Considérons la prémisse `(X) P (Y)`. Pendant l'examen de la règle, il est possible que:

X1a l'instanciation de `(X)` commence dans cette prémisse, `(Y)` n'étant pas encore instanciée,

X1b on propage une instanciation de `(X)` établie dans une autre prémisse, `(Y)` n'étant pas instanciée,

X2a l'instanciation de `(X)` commence dans cette prémisse, `(Y)` étant déjà instanciée,

X2b on propage une instanciation de `(X)` établie dans une autre prémisse, `(Y)` étant déjà instanciée.

On en déduit par symétrie les situations **Y1a** à **Y2b**.
Il est donc possible a priori que l'algorithme d'instanciation ait besoin d'un des ensembles suivants:

© l'ensemble $L_2^1(P)=\{x/\text{il existe } y \text{ tq } (x\ P\ y)\}$ (cas **X1a** et **X1b**),

© l'ensemble $L_2^3(P)=\{y/\text{il existe } x \text{ tq } (x\ P\ y)\}$ (cas **Y1a** et **Y1b**),

© pour tout `y` de $L_2^3(P)$, l'ensemble $L_{23}^1(P,y)=\{x/(x\ P\ y)\}$ (cas **X2a** et **X2b**),

© pour tout `x` de $L_2^1(P)$, l'ensemble $L_{21}^3(P,x)=\{y/(x\ P\ y)\}$ (cas **Y2a** et **Y2b**).

De plus, dans les cas **a** on parcourt un ensemble, dans les cas **b** on y procède à des recherches répétées. Une "bonne" algorithmique devrait donc être différente dans les deux cas: dans les cas **a** une structure de liste suffit, mais les cas **b** exigent une structure bien adaptée à des recherches multiples (arbre binaire ou B-arbre par exemple). Une telle structure (en l'occurrence dans BOOJUM des arbres binaires équilibrés) est satisfaisante pour les deux types de cas.
Le principe du système d'indexation est donc d'être basé sur la *factorisation* des faits satisfaisant les différents pattern. La représentation en machine des *ensembles* manipulés doit être bien adaptée à des recherches multiples.
Cependant, il n'est pas toujours nécessaire de disposer en permanence de *toutes* les factorisations associées aux mots-clés. Supposons par exemple que la variable `(Y)` de la prémisse `(X) P (Y)` soit libre, ou que la variable `(X)` soit prioritaire par rapport à `(Y)`, bref que pour une raison ou une autre le moteur ait décidé à la compilation des règles que `(X)` serait instanciée avant `(Y)`. Il est alors clair que les ensembles $L_2^3(P)$ et

$L_{23}^{1}(P,y)$ seront inutiles pour cette prémisse. S'il s'avère que toutes les prémisses ayant ce pattern vérifient la même priorité, ces ensembles seront *toujours* inutiles. Il ne faut donc pas les mémoriser, ce qui épargne du temps et de la mémoire.
On associe donc à chaque mot-clé correspondant à des prémisses avec au moins deux variables un *indicateur d'index* qui précise sous quelle(s) forme(s) un fait doit être mémorisé. Un fait nouveau (ou supprimé), en passant dans le filtre, *saura comment se mémoriser*.

4.3 Un nombre d'index raisonnable

Les considérations qui précèdent ont une portée générale dépassant le cadre strict du langage BOOJUM. Cependant, le principe de la factorisation des faits serait problématique avec des faits n-aires, n quelconque. La limitation aux *triplets* (qui n'est pas propre à BOOJUM) est nécessaire, sous peine de devoir construire trop d'index. A ce prix, le système d'indexation s'avère dans la pratique efficace. La limitation des index est en effet très fréquente, et épargne une quantité de mémoire importante. Le fait que la limitation des index soit liée aux heuristiques combinatoires des moteurs (choix du problème estimé le plus contraint) améliore encore cette limitation: la factorisation la plus concise des faits satisfaisant un pattern à au moins deux places est celle commençant par la place la plus contrainte. Par ailleurs, les index correspondant à différents patterns se recouvrent.

5 Problèmes de récursivité

On appelle *récursivité* entre règles le phénomène où, en se déclenchant, une règle `R2` modifie la base de faits de telle sorte qu'une instanciation d'une prémisse de la règle `R1` est ajoutée ou retranchée. Ce phénomène est évidemment extrèmement fréquent, puisqu'un raisonnement est la résultante d'un enchainement de déclenchements de règles. Le type d'algorithme d'instanciation que nous avons choisi nécessite un traitement particulier de ce phénomène. En effet, les algorithmes cherchent à un moment donné *toutes* les instanciations d'*une* règle en appliquant des heuristiques pour parcourir la plus patite partie possible de l'espace de recherche. Hors, un phénomène de récursivité revient à *modifier* l'espace de recherche. Pratiquement, deux types de récursivité sont à distinguer et demandent des traitements distincts:

Auto-récursivité: `R1=R2`. Deux sous-cas sont possibles: la règle *s'ajoute* une instanciation potentielle, et l'espace de recherche est *augmenté* (auto-récursivité *positive*); la règle *s'ôte* une instanciation potentielle, et l'espace de recherche est *diminué* (auto-récursivité *négative*).

Récursivité entre règles différentes: `R1?R2`. Seul le cas où la règle `R2` ajoute une instanciation potentielle à `R1` et où `R1` *a déjà été examinée* présente une difficulté: si `R1` vient à être réexaminée, une partie de l'espace de recherche *a déjà été parcourue*. Il ne faut parcourir que la différence.

5.1 Traitement de l'auto-récursivité

5.1.1 Auto-récursivité positive

Le traitement est fort simple. Les instanciations potentielles que s'auto-ajoute la règle sont notées dans un agenda . Il s'agit d'instanciations partielles de *prémisses*. Lorsque le processus d'instanciation normal est terminé, une procédure spéciale extrait de l'agenda (s'il

n'est pas vide) une telle instanciation, instancie brutalement la prémisse concernée, propage dans les autres prémisses qui partagent une variable avec elle, et relance le processus d'instanciation.

5.1.2 Auto-récursivité négative

Le traitement consiste à déterminer précisément les choix qu'il faut défaire parmi ceux qui ont abouti à l'instanciation permettant le déclenchement actuel: il faut rétablir un état *cohérent*. Le fait que l'on désire parcourir *tout* l'espace de recherche impose des contraintes sur la manière de procéder. En particulier, des techniques de TMS [Doyle, 1979]ne sont pas adaptées à ce genre de traitement. Par ailleurs, les ATMS [De Kleer, 1986a, 1986b, 1986c], s'ils fournissent théoriquement une réponse, sont en pratique trop gourmands en mémoire. En fait, la procédure mise en oeuvre est beaucoup plus simple. Il s'agit de trouver parmi les choix affectés par l'auto-récursion négative $X_1/x_1,...,X_n/x_n$ le choix le plus tardif possible et fournissant une instanciation partielle, c'est-à-dire l'indice k tel que $X_1/x_1,...,X_k/x_k$ est une instanciation partielle, mais pas $X_1/x_1,...,X_{k+1}/x_{k+1}$. Les choix de X_{k+1}/x_{k+1} à X_n/x_n sont alors défaits.

5.2 Traitement de la récursivité entre règles différentes

La méthode s'inspire de la *différentiation* de Yves Caseau [1987]. Lorsque l'on réexamine une règle déjà examinée, on cherche les faits *nouveaux* instanciant les prémisses de la règle. (La *date* d'un fait, c'est-à-dire le numéro de déclenchement de règle qui l'a établi, est connue). On procède ensuite comme pour l'auto-récursivité positive: on extrait une telle instanciation, on instancie brutalement toutes les variables de la prémisse concerné, on propage dans les autres prémisses, puis on lance le processus d'instanciation normal. Il est clair que l'espace parcouru est la différence entre l'espace actuel et l'espace parcouru lors du dernier déclenchement.

6 Boojum en pratique

La version initiale de BOOJUM a été développée de 1984 à 1986 à la Direction des Etudes et Recherches d'EDF sur IBM 30xx (MVS/TSO). Il a depuis 1986 été l'objet de nombreux ajouts, notamment en ce qui concerne les communications avec les langages de programmation classiques, les bases de données, etc. Mais ces possibilités fonctionnelles supplémentaires du langage n'ont pas conduit à changer les principes de mise en oeuvre exposés ici. Par ailleurs, il a depuis été porté sous le nom de GENESIA II (version 0) sur des matériels très différents, stations de travail et micro-ordinateurs. BOOJUM est écrit en PASCAL et comprend environ 7000 instructions.
Le tableau de la figure 3 donne une idée des performances que l'on peut attendre de BOOJUM/GENESIA II (où **d/s** signifie déclenchement par seconde, au sens du déclenchement effectif d'une règle).

	PC AT disque dur 640 Ko	MacIntosh II 2 Mo	IBM 30xx 5 Mo
Taille du code	260 Ko	160 Ko	550 Ko
Nombre max. de règles	>1000	>5000	$>5.10^4$
Nombre max. de faits	4000	15000	50000
Vitesse moyenne	12 d/s	50 d/s	500 d/s
Vitesse max.	300 d/s	1000 d/s	10000d/s

Figure 3: Performances de BOOJUM

"Taille du code" représente la taille du moteur d'inférence seul. En fait, la place nécessaire au dictionnaire des objets - c'est-à-dire des chaînes de caractères utilisées dans une base de connaissances- est incluse dans cette mesure. Celui-ci est dimensionné de manière à accepter des bases de connaissances de la taille maximale indiquée. Mais les différences constatées sont également dues en partie à "l'économie" des compilateurs utilisés.
La ligne **"Nombre max. de règles"** indique que l'auteur détient un exemple fonctionnant avec le nombre indiqué (le problème essentiel est la place mémoire). Il est évidemment impossible de donner un chiffre absolu, puisque la place occupée par une règle dépend - linéairement - du nombre de ses variables, prémisses et conséquents. On pourrait par contre donner une formule précise en fonction de ces trois éléments.
La valeur **"Nombre max. de faits"** a une signification mieux avérée: pratiquement, ce nombre a été atteint dans plusieurs applications. En effet, les principes de mémorisation des faits dans BOOJUM (même si la complexité théorique en mémoire est linéaire) rendent impossible une formule numérique place mémoire = fonction(nombre de faits). Il est donc nécessaire de se baser sur une utilisation "moyenne", c'est-à-dire sur des échantillons que l'on espère représentatifs.
La **"Vitesse max."** a un peu la même signification que le **"Nombre max. de règles"**: l'auteur détient des exemples où ces vitesses sont respectées.
La **"Vitesse moyenne"**, par contre, a une valeur beaucoup plus expérimentale. Elle représente effectivement la moyenne des vitesses constatées par une cinquantaine d'utilisateurs de BOOJUM à la Direction des Etudes et Recherches d'EDF, sur des dizaines de milliers d'exécutions. La variété des sujets traités, la taille des bases de règles (de 1 à 1352) et des bases de faits (jusqu'à 50000) exécutées, et la diversité de l'expérience des utilisateurs constituent à notre avis un échantillon valable. Et en définitive, par delà les discussions théoriques, c'est ce chiffre qui importe à l'utilisateur.

Remerciements

Je tiens à remercier les utilisateurs de BOOJUM de la DER-EDF, qui ont rendu possible la validation des idées présentées ici. Je tiens également à remercier le "referee" anonyme pour la qualité et la perspicacité de ses remarques.

Références

[Caseau, 1987] Yves Caseau. Etude et réalisation d'un langage objet: LORE. Thèse de l'Université Paris-Sud, novembre 1987.
[De Kleer, 1986a] Johan De Kleer. An assumption-based TMS. Artificial Intelligence Vol. 28 n° 2.
[De Kleer, 1986b] Johan De Kleer. Extending the ATMS. Artificial Intelligence Vol. 28 n° 2.
[De Kleer, 1986c] Johan De Kleer. Problem solving with the ATMS. Artificial Intelligence Vol. 28 n° 2.
[Dormoy, 1986] Jean-Luc Dormoy. Notice du langage et guide d'utilisation de S2.BOOJUM. Note EDF HI/5551/02, septembre 1986.
[Dormoy, 1987] Jean-Luc Dormoy. Résolution qualitative: complétude, interprétation physique et contrôle. Mise en oeuvre dans un langage à base de règles: BOOJUM. Thèse de l'Université Paris 6, décembre 1987.
[Doyle, 1979] J. Doyle. A truth maintenance system. Artificial Intelligence Vol. 12.
[Faller, 1988] Benoît Faller. Le système METRO, rapport préliminaire. Université Paris-Sud, LRI, rapport de recherche n° 421.
[Forgy, 1982] C.L. Forgy. RETE, a fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence Vol. 19.
[Laurière, 1976] Jean-Louis Laurière. Un langage et un programme pour résoudre et énoncer des problèmes combinatoires: ALICE. Thèse d'Etat présentée à l'Université Paris 6, mai 1976.
[Laurière, 1978] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. Artificial Intelligence Vol. 10.
[Laurière et Vialatte, 1985] Jean-Louis Laurière & Michèle Vialatte. Manuel d'utilisation de SNARK. Mars 1985.
[Mackworth & Freuder, 1985] A.K. Mackworth & E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraints satisfaction problems. Artificial Intelligence Vol. 25 1985.
[Mohr & Henderson, 1986] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. Artificial Intelligence Vol. 28 n° 2.
[Mohr & Masini 1988] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. Proceedings of ECAI'88.
[Montanari, 1974] U. Montanari. Network of constraints: fundamental properties and applications to picture processing. Inform. Sci. 7, pp. 95-132.
[Pitrat, 1985] Jacques Pitrat. Utilisation des connaissnces déclaratives. Cours fait à l'école Internationale d'informatique de l'AFCET, Aix-en-Provence, juillet 1985. Publications du LAFORIA, Université Paris 6.
[Vialatte, 1985] Michèle Vialatte. Description et applications du moteur d'inférence SNARK. Thèse de l'Université Paris 6, mai 1985.