Chrysostome Baskiotis

LOGIQUE COMPUTATIONNELLE & PROLOG

Fascicule 3

Bases de données et contraintes

EISTI

Année 2011

11

LES BASES DE DONNÉES EN PROLOG

NOUS montrons dans ce chapitre comment représenter les bases de données relationnelles en `Prolog`. Les notions de base comme les relations et les requêtes sont détaillées. Nous présentons ensuite, une brève introduction à la base de données déductives et la relation avec le modèle minimal de Herbrand. Les exercices montreront commet traiter les requêtes universelles en se servant de l'opérateur négation proposé par `Prolog`. Ensuite, la notion du monde fermé est introduit.

11.1 Définir les relations

En base de données relationnelle, une relation est exprimée par une table. Chaque colonne désigne un attribut particulier de la relation. Une ligne correspond à une enregistrement. Le schéma relationnel est décrit par le nom de la table et la désignation de chaque colonne. En `Prolog` une table est définie par un prédicat, le nombre de colonnes correspond à l'arité de ce prédicat, l'argument numéro *i* dans un tel prédicat correspondra à la colonne numéro *i*. Un enregistrement sera décrit par un fait représenté à l'aide du prédicat-relation et des termes filtrés. Prenons par exemple la relation *père-fils*, nous pouvons exprimer deux enregistrements (*toto-titi* et *toto-koko*)

de la table *père* en Prolog comme suit :

```
pere(toto, titi).
pere(toto, koko).
```

Le prédicat *père* est un prédicat d'arité 2, décrivant une relation entre deux données. Le schéma relationnel associé à *père* est *père (Pere,Fils)*. [1]

Remarquons bien que la base de faits en Prolog peut exprimer plusieurs relations appartenant à des schémas relationnels différents.

```
pere(toto, titi).
pere(toto, koko).
mere(lola, titi).
mere(lola, koko).
```

11.2 Définir les requêtes

Nous avons vu que nous pouvons constituer notre base de données avec un ensemble de faits. Le nombre de prédicats exprime le nombre de schémas relationnels constituant la base de données [2]. Nous classons les requêtes en deux catégories : *requêtes simples* et *requêtes composées*.

Les requêtes simples sont exprimées par un des prédicats constituant la base de faits. Ce sont des questions que l'on pose, par exemple :

```
?pere(toto,X). donne tous les enfants de toto.
?pere(X,koko). donne le p\'ere de koko.
```

Les requêtes composées sont exprimées par un ensemble de requêtes simples avec des relations entre eux (conjonction ou disjonction de requêtes). Pour effectuer ce type de requêtes en Prolog, nous définissons des règles permettant de simplifier la question ultérieurement. Le corps d'une clause exprime la conjonction des requêtes à poser. Sa tête est la définition simplifiée de la requête.
Par exemple, supposons que nous avons une base de données exprimant trois relations : *parent*, *homme et femme*, et que nous aimerions chercher les *pères* et les *mères*.

```
parent(toto, titi).
parent(toto, koko).
parent(lola, titi).
parent(lola, koko).
homme(toto).
homme(koko).
homme(titi).
femme(lola).
```

Pour cela nous définissons ces deux nouvelles relations sous forme de règles que nous ajoutons au programme :

1. Ceci est équivalent à la définition d'une table en Oracle appelée père et contenant deux colonnes, la première colonne représente le père et la deuxième représente le fils. Dans l'exemple précédent, cette table contenait deux lignes.
2. Le nombre de tables en Oracle

```
pere(X,Y):-parent(X,Y), homme(X).
mere(X,Y):-parent(X,Y), femme(X).
```

À partir de cela, nous pouvons poser les requêtes :

```
?pere(X,koko).  qui est \'equivalente \'a la requ\^ete compos\'ee:
                trouver le parent de koko X et X est homme.
?mere(X,koko).  qui est \'equivalente \'a la requ\^ete compos\'ee:
                trouver le parent de koko X et X est femme.
```

Remarquons que dans ce cas, la définition de chacune des relations *père* et *mère* a permis d'exprimer une relation de *conjonction* entre des requêtes simples.

D'un autre côté, si nous partons de l'ensemble de faits composés des schémas relationnels *père et mère* seulement, nous pouvons constituer la relation *parent* qui sera donnée par les règles suivantes :

```
parent(X,Y):-pere(X,Y).
parent(X,Y):-mere(X,Y).
```

Dans ce cas, la définition de la relation *parent* a permis d'exprimer une relation de *disjonction* entre requêtes simples.

11.2.1 Les requêtes recursives

Un cas particulier des requêtes composées sont les requêtes recursives. Une requête recursive est définie par une relation qui est appelée dans le corps d'une clause. Un exemple de ce type de requêtes est la recherche des *ancêtres* en utilisant seulement la relation *parent*

```
ancestre(X,Y):-parent(X,Z),parent(Z,Y).
ancestre(X,Y):-parent(X,Z), ancestre(Z,Y).
```

11.3 La programmation logique et le modèle relationnel des bases de données

La programmation logique permet d'exprimer plusieurs aspects du modèle relationnel des bases de données. Nous avons vu que les relations peuvent être décrites par des faits et que les expressions des requêtes sont souvent exprimées par des règles. L'arité d'une relation est exprimée par le nombre d'arguments du prédicat utilisé dans la définition du fait correspondant.

En général, cinq principaux types d'opérations définissent une algèbre relationnelle : l'union, la différence ensembliste, le produit cartésien, la projection et la sélection. Nous montrons comment implémenter chacune de ces opérations avec `Prolog` :

11.3.1 L'opération d'union

L'opération d'union permet de créer une relation d'arité n à partir de deux relations r et s d'arité n. La nouvelle relation est l'union des deux autres relations et est exprimée par les deux règles suivantes :

```
r_union_s(X1,...,Xn) :-r(X1,...,Xn).
r_union_s(X1,...,Xn) :-s(X1,...,Xn).
```

La relation *parent*, défini à partir des relations *père* et *mère*, est un exemple de ce type de relation.

11.3.2 L'opération de différence

La différence ensembliste est exprimée à l'aide de la négation :

```
r_diff_s(X1,...,Xn) :-r(X1,...,Xn), not s(X1,...,Xn).
r_diff_s(X1,...,Xn) :-s(X1,...,Xn), not r(X1,...,Xn).
```

Et ceci en supposant que r et s sont d'arité n.

ASCÈSE 11.1 *Soit les deux relations :* `enseignantPere` *et* `enseignantMere` *exprimées par deux prédicats d'arité trois chacune, définies par les shémas relationnels :* `enseignantPere(Enfant,Pere,Mere)`, `enseignantMere(Enfant,Pere,Mere)`.
Donner la relation `enseignantUnSeulParent(Enfant,Pere,Mere)`.

11.3.3 L'opération du produit cartésien

Cette opération permet de définir, à partir d'une relation r d'arité m et une relation s d'arité n, une nouvelle relation d'arité $k = m+n$ de la faon suivante :

```
r_prod_s(X1,...Xm,...,Xk) :-r(X1,...,Xm),s(Xm+1,...,Xm+n).
```

ASCÈSE 11.2 *Soit les deux relations :* `pere` *et* `mere` *exprimées par deux prédicats d'arité deux chacune, définies par les schémas relationnels :*
`pere(Enfant,Pere)`, `mere(Enfant,Mere)`.
Donner la relation `parents(Enfant,Pere, Mere)`.

11.3.4 L'opération de projection

La projection permet de définir une nouvelle opération contenant seulement un sous-ensemble des attributs composant la relation de départ. Par exemple la relation $r13$ est une projection de r selon le premier et le troisième argument :

```
r13(X1,X3) :-r(X1,X2,X3)
```

ASCÈSE 11.3 *Soit la relation :* `mere` *exprimée par un prédicat d'arité 2, avec le schéma relationnel suivant :* `mere(Enfant,Mere)`. *Donner la relation* `femmeayantEnfant(Mere)`. *Que remarquez-vous quand il s'agit d'une femme ayant plusieurs enfants ?*

11.3.5 L'opération de sélection

L'opération de sélection consiste à définir à partir d'une relation de départ, une autre relation dérivée et ceci en posant certaines contraintes sur certaines données. Ces contraintes peuvent être des contraintes d'ordre logique (exemple $X_1 < X_2$) ou bien des contraintes exprimées en relations définies antérieurement. Les deux exemples suivants illustrent ces deux cas de figure.

```
r1(X1,X3) :-r(X1,X2,X3), X2 > X3.
```

```
r2(X1,X3) :-r(X1,X2,X3), r3(X1).
r3(const1).
r3(const2).
```

La relation *r*2 permet de sélectionner l'ensemble des valeurs vérifiant la relation *r* à condition que *X*1 soit unifiée avec *const1* ou avec *const2*.

ASCÈSE 11.4 *Soient les relations :* `parent, fille, garcon`*, définies par les schémas relationnels :* `parent(Parent,Enfant),fille(Enfant),garcon(Enfant)`*. Donner la relation* `parentFille(Parent)` *qui exprime le fait que* `Parent` *a une fille. Que remarquez-vous quand il s'agit d'un parent ayant plusieurs filles ?*

Notons bien que dans ce paragraphe, nous avons présenté les opérations de base, nous pouvons bien entendu « fabriquer » nos propres opérations à partir des operations de base.

EXEMPLE 11.3.1 *Soit la base de données suivante, contenant les relations* `personne` *et* `voiture`*.*

```
voiture(123, fiat, toto, marron).
voiture(321, volvo, tata, rouge).
voiture(111, mazerati, cathy, blanche).
voiture(222, renault, loulou, rouge).
voiture(314, citroen, alma, verte).
personne(jojo, m, 16, toto, cathy).
personne(toto, m, 45, loulou, louloute).
personne(cathy, f, 40, koko, tata).
personne(anne, f, 14, toto, cathy).
personne(lolo, m, 30, koko,tata).
personne(louloute, f, 38, olive, alma).
personne(alma, f, 65, momo, mimi).
```

Pour répondre à la question : quelles sont les marques conduites par des femmes ? nous composons la requête suivante :

```
marque(X):- voiture(_,X,P,_), personne(P,f,_,_,_).
```

Remarquer bien que dans la composition de cette requête les opérations de base suivantes ont été appliquées :

- *Dans l'appel de* `voiture(_,X,P,_)` *une projection selon les arguments* `marque` *et* `personne` *est réalisée.*
- *Dans l'appel de* `personne (P,f,_,_,_)` *une selection selon les deux arguments* `personne` *et* `sexe` *est effectuée, les personnes trouvées via l'appel précédent et qui sont du sexe féminin sont choisies.*
- *Ensuite un produit cartésien et une projection selon le deuxième attribut du schéma relationnel* `voiture` *sont respectivement effectués.*

ASCÈSE 11.5 *Écrire les requêtes qui permettent de répondre aux questions suivantes :*

(1) Quels sont les propriétaires d'une volvo rouge ?

(2) Donner les couples (prénom, âge) des femmes qui conduisent des voitures.

11.4 Gestion de la base de données

Nous avons vu dans les sections précédentes qu'il est possible de définir les relations et les opérations sur les relations en Prolog, en utilisant les faits et les règles. Nous montrons dans cette section une technique permettant de gérer l'ensemble des relations et permettant de les traiter toutes en utilisant une structure unique.

Supposons que notre base de données contient les relations : r_1, ..., r_n. Supposons pour simplifier que la clé de chaque relation r_i est donnée par un seul attribut cle_i et que la valeur de cet attribut est donnée par le premier argument de la relation. Supposons que $nomAtt_{ij}$ désigne le nom de l'attribut numéro j concernant la relation r_i.

Nous définissons le prédicat bddVirt d'arité 4. Le premier argument contient le nom de la relation, le deuxième la valeur de la clé, le troisième le nom de l'attribut j et le quatrième sa valeur.

```
bddVirt(Ri, ValCLEi, NomATTij, ValATTij) :-
                              Ri(ValATTi1, ..., ValATTij, ...)
```

EXEMPLE 11.4.1 *Reprenons la base de données contenant les personnes et les voitures.*

La relation bddVirt *permet de définir une relation unique :*

```
bddVirt(voiture,Nu,numero,Nu) :- voiture(Nu,_,_,_).
bddVirt(voiture,Nu,marque,Ma) :- voiture(Nu,Ma,_,_).
bddVirt(voiture,Nu,proprietaire,Pr) :- voiture(Nu,_,Pr,_).
bddVirt(voiture,Nu,couleur,Co) :- voiture(Nu,_,_,Co).
bddVirt(personne,Nom,nom,Nom) :- personne(Nom,_,_,_,_).
bddVirt(personne,Nom,sexe,Se) :- personne(Nom,Se,_,_,_).
bddVirt(personne,Nom,age,Ag) :- personne(Nom,_,Ag,_,_).
bddVirt(personne,Nom,pere,Pe) :- personne(Nom,_,_,Pe,_).
bddVirt(personne,Nom,mere,Me) :- personne(Nom,_,_,_,Me).
```

Remarquez bien que la définition de cette relation permet de préciser explicitement le schéma relationnel de chaque relation dans la base de données, ainsi que sa clé. Remarquez aussi qu'il est possible de généraliser cette définition au cas où nous avons plusieurs clés et cela en augmentant l'arité du prédicat bddVirt.

Reprenons la question : quelles sont les marques conduites par des femmes ? Nous composons la requête suivante :

```
marque(X):- bddVirt(voiture,Nu,proprietaire,P),
            bddVirt(voiture,Nu,marque,X),
            bddVirt(personne,P,sexe,f).
```

Remarquez bien que chaque appel à bddVirt *permet d'effectuer une opération parmi les deux suivantes :*

(1) Une operation de projection selon le troisième argument, dans le cas où le quatrième argument est une variable libre.

(2) Une opération de selection selon le troisième argument dans le cas où le quatrième argument correspond à ensemble de valeurs donné.

Plus concrètement, dans l'exemple du calcul de `marque(X)` *l'appel* `bddVirt(voiture,Nu,propriétaire,P)` *permet de projeter les données de la base de données* `voiture` *selon l'argument* `propriétaire`*. Le deuxième appel permet de projeter ces même données selon l'argument* `marque`*. Le troisième appel* `bddVirt(proprietaire,P,sexe,f)` *effectue une selection sur l'ensemble des propriétaires trouvés par le premier appel selon l'argument* `sexe` *et avec valeur* `féminin`*. Finalement, le calcul de* `marque(X)` *est effectué par le produit cartésien des trois requêtes et ensuite par la projection sur la marque.*

ASCÈSE 11.6 *Réécrire les requêtes qui permettent de répondre aux questions suivantes en utilisant* `bddVirt` *:*

(1) Quels sont les propriétaires d'une volvo rouge ?

(2) Donner les couples `(prénom, age)` *des femmes qui conduisent des voitures.*

11.5 Introduction aux bases de données déductives

Nous avons vu comment exprimer une base de données relationnelle en utilisant des relations sous forme de faits. Nous avons vu comment utiliser les règles pour formuler des requêtes composées ou bien des requêtes recursives. En fait, nous pouvons utiliser les règles pour enrichir les bases de données avec des nouveaux faits. Ceci peut se faire soit en créant (en même temps) des nouveaux schémas relationnels (c'est-à-dire des nouveaux faits), soit en créant des nouvelles règles qui portent sur des faits existants. Par exemple nous pouvons ajouter le schéma relationnel `ancetreR` avec les données associées. Les faits `ancetreR` sont appelés des faits déduits car ce sont des faits qui ont été ajoutés à la base de faits et qui ont été déduits des faits existants et de la règle de déduction.

```
parent(toto, titi).
parent(toto, koko).
parent(lola, titi).
parent(lola, koko).
parent(koko,tintin)
parent(tintin,tino).
parent(tino,tony).
homme(toto).
homme(koko).
homme(titi).
homme(tintin).
homme(tino).
homme(tony).
femme(lola).
ancetre(X,Y):-parent(X,Z),parent(Z,Y).
ancetre(X,Y):-parent(X,Z), ancetre(Z,Y).
q2(X,Y):-ancetre(X,Y),assert(ancetreR(X,Y)).
```

A la fin de l'execution de ce programme, les faits ajoutés à la base de données sont : `ancetreR(toto, tintin). ancetreR(lola, tintin). ancetreR(koko, tino). ancetreR(tintin,`

`tony). ancetreR(toto, tino). ancetreR(toto, tony). ancetreR(lola, tino). ancetreR(lola, tony). ancetreR(koko, tony).` Remarquer bien que chacun de ces faits est deductible à partir du programme, en utilisant en particulier le modus-ponens et l'instantiation des variables. L'ensemble des faits obtenus correspond au modèle minimal de Herbrand car l'ensemble de départ est constitué d'un ensemble de faits filtrés.

ASCÈSE 11.7 *Soit le programme suivant :*

```
gr(a,b,2).
gr(a,g,6).
gr(b,e,2).
gr(b,c,7).
gr(g,e,1).
gr(g,h,4).
succImm(X,Y):-gr(X,Y,_).
suc(X,Y):-succImm(X,Y).
suc(X,Y):-succImm(X,Z),suc(Z,Y).
```

(1) Trouver le modèle minimal de Herbrand.

(2) Quelle sera la réponse à la question `?suc(X,Y)` *en* `Prolog`*.*

ASCÈSE 11.8 *Soit le programme suivant :*

```
acouleur(ciel).
acouleur(nuage).
acouleur(mer).
couleur(ciel,bleu).
couleur(nuage,blanc).
couleur(arbre,vert).
objetCouleur(X,Y):-acouleur(X),couleur(X,Y).
```

(1) Poser les questions : `?objetCouleur(X,Y)`, `?objetCouleur(mer,Y)`, `?objetCouleur(arbre,Y)`.

(2) Ajouter deux clauses qui traitent le problème de l'absence de relations.

ASCÈSE 11.9 *Soit la base de données relationnelle contenant les trois tables suivantes :*

NPRO	NOMP	QTES	COULEUR
100	Bille	100	Verte
200	Poupée	50	Rouge
300	Voiture	70	Jaune
400	Carte	350	Bleu

TABLE 11.1 – Table des produits

Écrire un programme `Prolog` *qui répond aux requêtes suivantes :*

(1) Donner la liste des noms et couleurs de tous les produits.

(2) Donner les noms et les quantités des produits de couleur rouge.

(3) Donner pour chaque produit en stock, le nom du fournisseur associé.

NVEN	NOMC	NPRV	QTEV	DATE
1	Dupont	100	30	08-03-1999
2	Martin	200	10	07-01-1999
3	Charles	100	50	01-01-2000
4	Charles	300	50	01-01-2000

TABLE 11.2 – Table des ventes - clients

NACH	NOMF	NPRA	QTEA	DATE
1	Fournier	100	70	01-03-1999
2	Fournier	200	100	01-03-1999
3	Dubois	100	50	01-09-1999
4	Dubois	300	50	01-09-1999

TABLE 11.3 – Table des achats - fournisseurs

(4) *Donner pour chaque produit en stock en quantité supérieure à 10 et de couleur rouge, les triplets nom de fournisseurs ayant vendu ce type de produit et nom de client ayant acheté ce type de produite et nom du produit.*

(5) *Donner les noms de clients ayant acheté au moins un produit de couleur verte.*

(6) *Donner les noms des clients ayant acheté tous les produits stockés.*

(7) *Donner les produits fournis par tous les fournisseurs et achetés par au moins un client.*

11.6 L'itérativité en Prolog

La programmation en `Prolog` nécessite l'utilisation de la récursivité. Mais il y a des situations où la récursivité est imposée de facomplètement artificielle et n'porte strictement rien à la cohérence du programme, comme par exemple le parcours des enregistrements d'un fichier séquentiel. Ce paragraphe est consacré à la programmation en `Prolog` de la structure itérative `tant que`.

Il s'agit de créer un opérateur de la forme suivante :

```
tantQue X alorsFaire Y
```

dont la programmation en `Prolog` est donnée par

PROGRAMME 11.6.1

```
    :- op(100, fy, tantQue).
    :- op(900, xfx, alorsFaire).

tantQue X alorsFaire Y :- (X,Y,fail); true.
```

On peut associer cet opérateur avec un autre qui teste si un indice numérique est entre une valeur minimale et une valeur maximale :

PROGRAMME 11.6.2

```
    :- op(100, xfx, estEntre).
    :- op(900, xfy, et).

    Indice estEntre Lmin et Lmax :- entre(Indice,Lmin,Lmax).
```

```
entre(Indice,Indice,Lmax) :- Lmax >= Indice.

entre(Indice,Lmin,Lmax) :- Lmin < Lmax, L is Lmin+1,
                           entre(Indice,L,Lmax).
```

On constate ici que la définition de l'opérateur `estEntre` nécessite un appel récursif à lui-même qui, s'agissant d'un opérateur, est impossible. D'où le recours au prédicat `entre` qui lui, en tant que prédicat, peut faire un appel récursif à lui-même.

En utilisant ce deux opérateurs on peut par exemple afficher la table de multiplication entre deux valeurs numériques `L1` et `L2` :

PROGRAMME 11.6.3

```
tableMultiplication(L1,L2 :- tantQue (I estEntre L1 et L2) alorsFaire
                             (tantQue (J estEntre L1 et L2) alorsFaire
                             (K is I*J, write(K), write(', ')),write(nl)).
```

11.7 Utilisation de l'itérativité aux bases de données

Nous reprenons les bases de données `voiture` et `personne` de l'exemple 4.3.1. Nous allons aussi utiliser la base de données virtuelles `bddVirt` de l'exemple 4.4.1.

Nous pouvons envisager d'afficher un élément particulier d'une base de données ou, encore, toute la base de données. On introduit les deux opérateurs :

PROGRAMME 11.7.1

```
:- op(1100, fx, afficheListe).
:- op(1100, fx, afficheTout).

afficheListe [X,Y] :- nl,
                      for(Y, (write(X), nl)),
                      fail.

afficheTout X :- afficheListe [X, X].
for(X,Y) :-  X, Y.
for(X,Y).
```

Nous pouvons en utilisant le premier opérateur de répondre à des questions du type « donner les marques de voitures dont les propriétaires sont des femmes et aussi les noms de ces femmes » comme suit :

PROGRAMME 11.7.2

```
question :- afficheListe [(Marque, Prop),
                    (personne(Prop, f, _,  _, _),
                     voiture(_, Marque, Prop, _))].
```

Nous pouvons aussi afficher toute une base de données en utilisant le deuxième opérateur :

PROGRAMME 11.7.3

```
question :- afficheTout voiture(X,Y,Z,W).
```

Pour améliorer l'interactivité lors de l'interrogation de bases de données, nous pouvons envisager d'introduire un nouvel opérateur qui permet de s'approcher du langage naturel lorsque nous posons une question.

PROGRAMME 11.7.4

```
:- op(900, xfx, avecCle).
:- op(100, xfx, 'aComme').

BdD avecCle Cle aComme (Attribut = Valeur) :-
                        bddVirt(BdD, Cle, Attribut, Valeur),
                       not(Valeur = nil).
```

qui permet de poser la question précédente comme suit :

PROGRAMME 11.7.5

```
question :- afficheListe [(Ma, Pr), (voiture avecCle N aComme (marque = Ma),
                                     voiture avecCle N aComme (proprietaire = Pr),
                                     personne avecCle Pr aComme (sexe = f))].
```

12

PROGRAMMATION LOGIQUE SOUS CONTRAINTES (PLC)

La programmation logique sous contraintes (PLC) est une généralisation de la programmation logique qui cherche à doter la programmation logique de l'efficacité des méthodes de la résolution des contraintes, de sorte que la résolution d'un problème soit aussi rapide qu'avec les langages impératifs et, au contraire, le temps de développement beaucoup plus court. Pour ce faire on rajoute, à côté du mécanisme d'inférence de Prolog, un solveur des contraintes qui peut examiner les contraintes non seulement dans le programme, mais aussi dans la question. La PLC prend en considération d'autres structures mathématiques en plus de l'univers de Herbrand.

La PLC a comme objectif de calculer des valeurs des variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ qui appartiennent aux domaines $\mathcal{D} = \{D_1, \ldots, D_n\}$ respectivement et qui satisfont à un ensemble des contraintes $\mathcal{C} = \{C_1(X_1, \ldots, X_n), \ldots, C_m(X_1, \ldots, X_n)\}$. Ce que nous appelons contraintes sont des relations logiques entre les variables, à savoir une contrainte est une fonction

$$C_i : D_1, \ldots, D_n \rightarrow \{vrai, faux\} = \{1, 0\}$$

Un tuple $S = \{x_1, \ldots x_n\}$ est une *solution* pour le système $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ ssi $\forall C_i \in \mathcal{C} : \ C_i(S) = 1$.

En Prolog pour résoudre des problèmes de PLC on cherche de manière itérative, à trouver une distribution des valeurs des variables dans leur domaine respectif, de sorte que les contraintes soient satisfaites.

Il est important de ne pas confondre programmation logique sous contraintes et programmation mathématique. La PLC est un programme qui code une méthode pour résoudre un problème particulier. En tant que programme, le code contient

– un ensemble des variables ;
– des contraintes entre ces variables, et

– des programmes qui indiquent la manière selon laquelle les variables doivent se modifier afin de pouvoir trouver des valeurs pour ces variables qui satisfont aux contraintes.

La difficulté de l'approche Prolog "pure" vient du fait que les objets sémantiques manipulés par Prolog sont déclarés de manière explicite. Or les contraintes expriment de manière implicite les relations entre ces objets sémantiques. Par exemple écrivons en Prolog, le programme qui calcule le minimum entre deux valeurs numériques. On a

PROGRAMME 12.0.6

```
mini(X,Y,Z) :- X =< Y, X = Z.
mini(X,Y,Z) :- Y =< X, Y = Z.
```

À la question

```
?- mini(1,2,Z).
```

on obtient la réponse

```
Z = 1
```

Mais à la question

```
?- mini(X,2,1).
```

on obtient la réponse

```
ERROR: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, 2, 1) ? Exception details
Exception term: error(instantiation_error, context(system: (=<)/2, _G353))
       Message: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, 2, 1) ? Can't ignore goal at this port
```

et aussi à la question

```
?- mini(X,X,Z).
```

on obtient la réponse

```
ERROR: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, _G286, _G288) ? abort
% Execution Aborted
```

Manifestement Prolog ne sait pas interpréter les relations qu'elles existent entre `X`, `Y` et `Z` et qui s'expriment à travers les conditions des règles et qui lui auraient permis d'établir que `X = 1` à la deuxième question et que `Z = X` à la troisième.

Si on utilise la bibliothèque des contraintes sur les nombres rationnels, et on écrit le programme

```
:- use_module(library(clpq)).

miniC(X,Y,Z) :- {X =< Y, X =:= Z}.
miniC(X,Y,Z) :- {Y =< X, Y =:= Z}.
```

dont la syntaxe s'éclaircira par la suite, on a

```
?- miniC(X,2,1).
X = 1

?- miniC(X,X,Z).
{Z=X}
```

c'est-à-dire Prolog arrive maintenant à interpréter les relations de manière implicite aussi.

Une expression (par exemple une expression arithmétique) est un terme formé en utilisant l'alphabet du domaine (les nombres dans le cas des expressions arithmétiques). Une contrainte est une formule du type $s \otimes t$ où par $\otimes$ on désigne un opérateur du domaine (par exemple pour les contraintes arithmétiques les opérateurs sont <, =<, =>, >,$\neq$).

Pour faire de la programmation avec contraintes en Prolog, il faut utiliser des librairies spécialisées, qui sont les suivantes :

- `clp/bounds` : Contraintes portant sur des domaines des nombres entiers.
- `clp/clp_distinct` : Contraintes portant sur des nombres entiers. Elle n'est pas très différente de `clp/bounds`
- `clp/clpfq` : Contraintes sur des nombres rationnels.
- `clp/clpqr` : Contraintes sur des nombres réels.

Pour pouvoir utiliser la librairie adéquate, il faut la charger, à l'aide de la requête `:- use_module`. On a, par exemple pour la librairie `clp/bounds` :

```
:- use_module(library('clp/bounds')).
```

Nous présentons ci-après les principales librairies.

12.1 PLC sur des domaines finis

Les domaines finis sont particulièrement utiles pour la résolution des problèmes combinatoires et de problèmes de distribution, de planification, etc.

L'appel au solveur se fait par la requête :

```
:- use_module(library(clpfd)).
```

Nous donnons ci-après la liste des principaux prédicats.

- `Expr op Expr`, où `op` est un des opérateurs `#=, #\=, #<, #=<, #>, #>=`
  Exemples :

```
X in 1..2, Y in 3..5, X #=<Y, Y #=< B, X+Y #= T.
X in 1..2,
X+Y #=T,
Y in 3..5,
B #>=Y,
B in 3..sup,
T in 4..7.
```

- `domain(+Var, +Min, +Max)` Les variables dans la liste `Var` sont dans `[Min, Max]`.
- Contraintes propositionnelles. Ce sont
  - `#\Q` .- Vraie si la contrainte `Q` est fausse.
  - `P #/\ Q` .- Vraie si les contraintes `P` et `Q` sont vraies toutes les deux.
  - `P #\/ Q` .- Vraie si au moins une de contraintes `P, Q` est vraie.

  - `P #==> Q, Q #<== P` .- Vraie si `P` implique `Q`.
  - `P #<==> Q` .- Vraie si les contraintes `P` et `Q` sont simultanément vraies ou fausses.
- `labeling(+Options, +Vars)` Indique à Prolog de tester systématiquement toutes les valeurs des domaines finis des variables qui sont dans `Vars`, afin de trouver des solutions compatibles avec les contraintes des ces variables. `Options` est une liste d'options dont le détail se trouve dans le manuel de SWI-Prolog, annexe A.7
- `label(+Vars)` Équivalent à `labeling([], +Vars)`.
- `all_different(+Vars)` Les variables sont distinctes deux à deux.
- `sum(+Vars, +Rel, ?Expr)` La somme des valeurs des variables de la liste `Vars` est en relation `Rel` avec l'expression `Expr`.
- `tuples_in(+Tuples, +List)` `List` est une liste de tuples d'entiers. `Tuples` est une liste de variables dont le nombre est égal au nombre d'éléments d'un tuple de `List`. Ces variables sont contraintes par les éléments correspondants des tuples de `List`.
  Exemple :
  PROGRAMME 12.1.1

```
tuples_in([[X,Y]], [[1,2],[1,5],[4,0],[4,3]]).
X in 1\/4,
tuples_in([[X, Y]], [[1, 2], [1, 5], [4, 0], [4, 3]]),
Y in 0\/2..3\/5.
```

La conception d'un programme avec contraintes sur des bornes finis se fait selon les quatre étapes suivantes :

(1) *Définition des variables*.- On précise les variables qui seront sous contraintes. Habituellement ces variables sont stockées dans une liste, par exemple `ListeVar`.

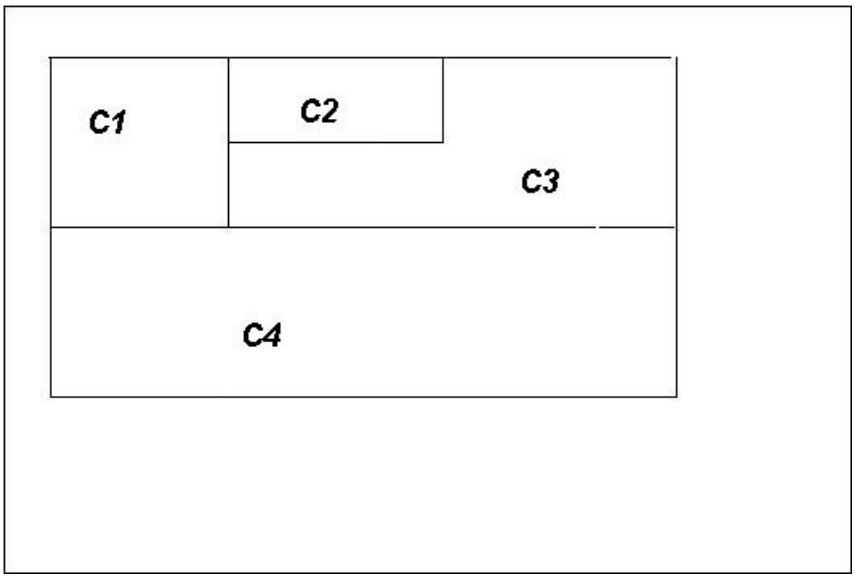
(2) *Définition des domaines*.- Il s'agit de définir les domaines de variation des variables déterminées précédemment.

(3) *Spécification des contraintes*.- On spécifie les relations (contraintes) que les variables ont entre elles. Pour les spécifications on utilise le prédicat `all_different/1` avec comme argument une liste dont les éléments doivent être tous différents et les opérateurs
`#=, # \=, #<, #=<, #>, #>=`
qui ont la même signification que les opérateurs habituels. La spécification des contraintes sert à restreindre l'espace de variation des contraintes qui est l'espace de recherche de solutions.

(4) *Recherche de solutions*.- Prolog utilise la technique de la force brute, à savoir :
  - Choix d'un variable.
  - Choix, pour cette variable, d'une valeur issue de son domaine de variation.
  - Réduction des domaines de variation des autres variables par propagation des contraintes. S'il n'y a pas de solution, retour en arrière et sélection d'une autre valeur pour la variable. Sinon, choisir une autre variable et recommencer.
  - Si on a assigné des valeurs à toutes les variables, retourner ces valeurs (la solution).
  - Recommencer l'algorithme jusqu'à épuisement des toutes les affectations possibles des variables.

L'exemple suivant fournit un modèle pour l'utilisation de cette librairie dans le cas d'un problème combinatoire.

Soit la carte suivante :



Nous voulons colorier cette carte en utilisant trois couleurs – bleue, jaune et rouge – de sorte que deux régions adjacentes n'aient pas la même couleur.

Le programme suivant fournit une réponse.

PROGRAMME 12.1.2

```
% Chargement de la librairie clp/bounds
:- use_module(library('clp/bounds')).

% Association num\'ero - couleur
couleur(1, bleu).
couleur(2, jaune).
couleur(3, rouge).

colorier :-
      ListeVar = [C1, C2, C3, C4], % Definition des variables
      ListeVar in 1..3,            % Definition des domaines
      label(ListeVar),             % Lancement de l algorithme
      all_different([C1,C2]),      % Contraintes sur les variables
      all_different([C1, C3]),     % Il faut que toutes soient differentes
      all_different([C1, C4]),
      all_different([C2, C3]),
      all_different([C3, C4]),
      affiche(ListeVar).           % Affichage du r\'esultat.

affiche([]):- nl.
affiche([H|T]) :- couleur(H,C),write(C),tab(2), affiche(T).
```

En lançant le programme, on obtient les réponses

```
bleu  jaune  rouge  jaune
bleu  rouge  jaune  rouge
jaune  bleu  rouge  bleu;
jaune  rouge  bleu  rouge
rouge  bleu  jaune  bleu;
rouge  jaune  bleu  jaune
```

Une façon de programmer avec moins de lignes est de grouper les contraintes selon la notion de voisinage. On a ainsi le programme ci-après :

PROGRAMME 12.1.3

```
% Chargement de la librairie clp/bounds
:- use_module(library('clp/bounds')).

% Association num\'ero - couleur
couleur(1, bleu).
couleur(2, jaune).
couleur(3, rouge).

colorier :-
      ListeVar = [C1, C2, C3, C4],  % Definition des variables
      ListeVar in 1..3,             % Definition des domaines
      label(ListeVar),              % Lancement de l algorithme, c-a-d recherche
                                    % d une affectation de valeurs a la liste
                                    % des variables ListeVar, qui satisfait
                                    % aux contraintes
      all_different([C1, C2, C3]),  % Contraintes sur les variables
      all_different([C1, C3, C4]),  % Il faut que toutes soient differentes
      affiche(ListeVar).            % Affichage du r\'esultat.

affiche([]):- nl.
affiche([H|T]) :- couleur(H,C),write(C),tab(2), affiche(T).
```

où `[C1, C2, C3]` et `[C1, C3, C4]` sont des listes qui regroupent des régions voisines.

Nous terminons ce paragraphe par l'examen d'un problème de planification d'un trajet par train. On suppose qu'on a une liste de quadruples qui indiquent, pour chaque train, les gares de départ et d'arrivée ainsi que les horaires de départ et d'arrivée. On représente les gares et les horaires par des nombres entiers. On a le programme suivant :

PROGRAMME 12.1.4

```
:- use_module(library(clpfd)).
trains([[1,2,0,1],[2,3,4,5],[2,3,0,1],[3,4,5,6],[3,4,2,3],[3,4,8,9]]).
itineraire(A, D, Ps) :- Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
                        T2 #> T1,
                        T4 #> T3,
                        trains(Ts),
                        tuples_in(Ps, Ts).
```

On cherche un itinéraire réalisable qui mène de la gare 1 à la gare 4. Nous avons

```
?- itineraire(1, 4, Ps).
Ps = [[1, 2, 0, 1], [2, 3, 4, 5], [3, 4, 8, 9]].
```

12.2 PLC sur les nombres rationnels et réels

L'appel au solveur de la librairie sur les nombres rationnels se fait par

```
:- use_module(library(clpq)).
```

et sur les nombres réels

```
:- use_module(library(clpr)).
```

Les principaux prédicats de cette librairie sont

- {⟨contraintes⟩} Exemple :`{ {A+B = 10, A = B}.`
- `entailed(+Contrainte)` Le prédicat se vérifie si la contrainte est vérifiée. Exemple : `{ A =< 4}, entailed(A= \=5).`
- `inf(+Expr, -Inf)` et `sup(+Expr, -Sup)` Unification de `Inf` (resp. `Sup`) avec l'infinimum (resp. supremum) de l'expression `Expr`.
  Exemple :

```
?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y}, sup(Z, Sup).

?- { 2*X+Y >= 16, X+2*Y >= 11, X+3*Y >= 15, Z = 30*X+50*Y}, inf(Z, Inf).
```

- `minimize(+Expr)` et `maximize(+Expr)` Calcul du minimum (resp. maximum) de l'expression `Expr` et unifie les variables aux valeurs correspondantes.
  Exemple :

```
?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y}, maximize(Z).

X = 6.6,

Y = 2.8,

Z = 338.0

?- { 2*X+Y >= 16, X+2*Y >= 11, X+3*Y >= 15, Z = 30*X+50*Y}, minimize(Z).

X = 7.0,

Y = 2.0,

Z = 310.0 ;
```

- `bb_inf(+Ints, +Expr, -Inf)` Unifie `Int` avec le minimum de l'expression `Expr` sous la condition que les variables de la liste `Ints` soient entières.
- `bb_inf(+Ints, +Expr, -Inf, -Vertex, +Eps)` Même chose que précédement mais en donnant une tolérance de `Eps` aux valeurs entières. Ainsi `X` est considéré comme entier si `abs(round(X)-X) < Eps`. Vertex contient la liste des valeurs de la solution pour les variables entières.

En fonction du problème considéré nous pouvons soit, dans le cas où le problème n'a pas de contraintes, appliquer l'algorithme, soit, dans le cas contraire, suivre la démarche présentée dans la section précédente.

Ainsi pour calculer la racine carrée d'un nombre avec l'algorithme de Newton, on établit l'algorithme et on obtient le programme suivant :

PROGRAMME 12.2.1

```
:- use_module(library(clpr)).
% Calcul racine carr\'{e}e
racine(N, R) :- racine(N, 1, R).
racine(0, S, R) :- !, S=R.
racine(N, S, R) :- N1 is N-1, \{ S1 = S/2 + 1/S \}, racine(N1, S1, R).
```

qui à la question

```
racine(5,R).
```

donne la réponse

```
R = 1.41421
```

Pour la programmation linéaire en nombres entiers, nous avons le programme

PROGRAMME 12.2.2

```
% Probl\'{e}me d'optimisation lin\'{e}aire en nombres entiers.

:- use_module(library(clpr)).

exemple(pl, Obj, Vs, Ints, []) :-       % D\'{e}finition des variables
                                  Vs = [ X1,X2,X3,X4,X5,X6,
                                         Y1,Y2,Y3,Y4,Y5,Y6,
                                         Z1,Z2,Z3,Z4,Z5,Z6],

Ints = [Y6, Y5, Y4, Y3, Y2,             % D\'{e}termination des variables enti\'{e}res
        X6, X5, X4, X3, X2, X1],

Obj =   2700*Y1 + 1500*X1 + 30*Z1       % La fonction \'{a} minimiser
      + 2700*Y2 + 1500*X2 + 30*Z2
      + 2700*Y3 + 1500*X3 + 30*Z3
      + 2700*Y4 + 1500*X4 + 30*Z4
      + 2700*Y5 + 1500*X5 + 30*Z5
      + 2700*Y6 + 1500*X6 + 30*Z6,
      allpos(Vs),                       % Les variables doivent \^{e}tre positives
    { Y1 = 60, 0.9*Y1 +1*X1 -1*Y2 = 0, % Les contraintes lin\'{e}aires
      0.9*Y2 +1*X2 -1*Y3 = 0, 0.9*Y3 +1*X3 -1*Y4 = 0,
      0.9*Y4 +1*X4 -1*Y5 = 0, 0.9*Y5 +1*X5 -1*Y6 = 0,
      150*Y1 -100*X1 +1*Z1 >= 8000,
      150*Y2 -100*X2 +1*Z2 >= 9000,
      150*Y3 -100*X3 +1*Z3 >= 8000,
      150*Y4 -100*X4 +1*Z4 >= 10000,
      150*Y5 -100*X5 +1*Z5 >= 9000,
      150*Y6 -100*X6 +1*Z6 >= 12000,
      -20*Y1 +1*Z1 =< 0, -20*Y2 +1*Z2 =< 0,
      -20*Y3 +1*Z3 =< 0,
      -20*Y4 +1*Z4 =< 0, -20*Y5 +1*Z5 =< 0,
      -20*Y6 +1*Z6 =< 0,
       X1 =< 18, 57 =< Y2, Y2 =< 75, X2 =< 18,
       57 =< Y3, Y3 =< 75, X3 =< 18, 57 =< Y4,
       Y4 =< 75, X4 =< 18, 57 =< Y5, Y5 =< 75,
       X5 =< 18, 57 =< Y6, Y6 =< 75, X6 =< 18}.

allpos([]).

allpos([X | Xs]) :- {X >= 0}, allpos(Xs).

% Minimum relax\'{e} (sans la contrainte de solution enti\'{e}re

optR(Inf) :- exemple(pl, Obj, _, _, _), inf(Obj, Inf).

% Minimum avec variables enti\'{e}res

opt(Sommets, Inf) :- exemple(pl, Obj, _, Ints, _),
```

```
bb_inf(Ints, Obj, Inf,Sommets, 0.001).
```

avec réponses

```
?- optR(Inf).
Inf = 1.16719e+006

?- opt(S,Inf).
S = [75, 70, 70, 60, 60, 0, 12, 7, 16|...],
Inf = 1.2015e+006
```

12.3 CHR : Constraint Handling Rules

SWI-Prolog à l'aide de CHR fournit la possibilité d'écrire ses propres solveurs pour des contraintes spécifiques. Il peut être utilisé à plusieurs types d'applications, comme planification, distribution, test de type, etc.

Pour utiliser la technique du CHR dans un programme Prolog, il faut au préalable

- inclure les contraintes et les règles dans un module à l'aide de la requête
  `:- module(nomModule, [nomContrainte1/arité, ...,nomContrainteN/arité]).`
- faire appel au solveur CHR avec la requête
  `:- use_module(library(chr)).`
- et enfin il faut déclarer chaque contrainte avec la requête
  `:- chr_constraint(nomContrainte1/arit\U{e9}, ...,nomContrainteN/arit\U{e9}).`

Ensuite il faut écrire les containtes, sous la forme

```
etiquette @ contrainte.
```

L'exemple suivant reprend le programme pour l'addition de deux naturels dans le cadre de satisfaction des contraintes.

PROGRAMME 12.3.1

```
:- module(addition,[add/3]).
:- use_module(library(chr)).
:- chr_constraint add/3.

zero1 @ add(0,Y,Z) <=> Y = Z.
zero2 @ add(X,0,Z) <=> X = Z.
zero3 @ add(X,Y,0) <=> X = 0, Y = 0.

meme1 @ add(X,E,E) <=> X = 0.
meme2 @ add(E,Y,E) <=> Y = 0.

succ1 @ add(s(X),Y,Z) <=> Z = s(W), add(X,Y,W).
succ2 @ add(X,s(Y),Z) <=> Z = s(W), add(X,Y,W).
succ3 @ add(X,X,s(Z)) <=> Z = s(W), X = s(Y), add(Y,Y,W).

cherche @ add(X,Y,s(Z)) <=> true | add(X1,Y1,Z),
                               (X = s(X1),Y = Y1 ; X = X1,Y = s(Y1)).
```

Utilisation

```
?- add(s(s(0)),s(s(s(0))),s(s(s(s(s(0))))).

yes.

?- add(s(s(0)), s(0), Z).

Z = s(s(s(0))).
```

12.4 Références

Pour la rédaction de ce chapitre, les références suivantes ont été utilisées :

[1] K. R. APT, P. ZOETEWEIJ : An Analysis of Arithmetic Constraints on Integer Intervals, Constraints, v. 12, nu. 4, 2007, pp. 429-468

[2] T. FRÜHWIRTH, S. ABDENNADHER : *Essentials of Constraint Programming*, Springer, 2003

[3] J. Jaffar, M. J. Maher : Constraint logic programming : a survey, Journal of Logic Programming 19 & 20, 503-581, 1994

[4] J. Jaffar et al. : The semantics of constraint logic programs, Journal of Logic Programming, 37, 1-46, 1998.

[5] M. WALLACE : Survey : Practical Applications of Constraint Programming, Imperial College, September 1995

12.5 Exercices

EXERCICE 12.1 *Soient les inégalités*

$$\begin{aligned} x-y &\leq 3 \\ x+y &\leq 1 \\ -x+z &\leq -3 \\ -x-z &\leq -1 \\ -20 \leq x &\leq 20 \\ -20 \leq y &\leq 20 \\ -20 \leq z &\leq 20 \end{aligned}$$

Calculer les valeurs maximales de x, y et z en utilisant la librairie `clpr`*, avec*

- `sup`
- `maximize`

Faire le même programme en utilisant la librairie `clpfd`*. Que constatez-vous*

EXERCICE 12.2 *En utilisant la librairie* `chr`*, écrire un solveur qui permet le calcul de nombres premiers de 1 jusqu'à N.*

EXERCICE 12.3 *Pour un casse, Toto doit ouvrir un coffre. Les informations dont il dispose sont les suivantes :*

- *Le code est composé de 9 chiffres au total, de 1 à 9.*
- *Chaque chiffre apparaît une seule fois dans le code.*
- *Le 1er chiffre est différent de 1, le 2ème différent de 2 et ainsi de suite.*
- *La différence entre le 4ème et le 6ème chiffres*[1] *est égale au 7ème chiffre.*

1. Attention : il n'est pas écrit la différence du 4ème chiffre moins le 6ème

- *Le produit de trois premiers chiffres est égal à la somme de deux derniers.*
- *La somme du 2ème, 3ème et 6ème chiffre est inférieure au 8ème chiffre.*
- *Le dernier chiffre est inférieur au 8ème chiffre.*

Pour aider Toto il faut choisir la librairie à utiliser, modéliser le problème et calculer la solution.

EXERCICE 12.4 *En utilisant le programme* `add/3` *du listing 12.3.1 écrire le programme* `mult(X,Y,R)` *qui calcule le résultat symbolique* `R` *de la multiplication de deux naturels* `X` *et* `Y`*.*

EXERCICE 12.5 *Une usine fabrique deux produits P_1 et P_2.*
Pour fabriquer une unité du produit P_1 il faut une unité de chacune de deux matières premières m_1 et m_2.
Pour fabriquer une unité du produit P_2, il faut deux unités de la matière première m_1 et une unité de chacune de deux matières premières m_2 et m_3.
Les stocks des matières premières sont 1500 unités pour m_1, 1200 unités pour m_2 et 500 unités pour m_3.
La vente d'une unité de P_1 procure un gain de 15 euros et de P_2 10 euros.

En utilisant la PLC calculer les quantités à produire pour chaque produit, de sorte que le gain soit maximal.

13

TROIS APPLICATIONS

LE dernier chapitre de ce support de cours est consacré à la présentation de trois applications issues des domaines de télécommunications, de la robotique et du génie logiciel. L'objectif est de montrer de faconcrète la souplesse et l'adaptivité de `Prolog`.

13.1 Algorithme de routage

Un problème des télécomunications qui concerne la couche réseau et qui est relatif à l'envoi des paquets de la source à tous les déstinataires, est le routage de ces paquets à travers le réseau([1]). Pour ce faire on considère le réseau comme un graphe dans lequel les sommets représentent des nœuds des « packets switch » ou IMP (Interface Message Protocol) et les arcs une ligne de communications. Un des algorithmes utilisés est celui du plus court chemin dû à Dijkstra. Pour faire fonctionner cet algorithme il faut à chaque arc qui relie deux nœuds associer un coût. Ce coût est calculé en fonction de la distance, de la bande passante, de la moyenne du trafic, du coût de communication, de la moyenne du délai d'attente, etc.

On suppose que la base de données est constituée par le graphe du réseau `gr(sommet, prédecesseur, poids(sommet, prédecesseur))`. L'algorithme que nous allons utiliser est une adaptation de l'algorithme connu en Intelligence Artificielle sous le nom « le meilleur d'abord ». On utilise les ensembles OUVERT et FERME dont les éléments sont les triplets (x,y,d(x)) où x est un sommet, y est le sommet prédecesseur de x et d(x) est le coût pour arriver de la racine à x. De plus l'ensemble OUVERT est constamment trié selon les valeurs ascendantes de la distance d(x).

L'algorithme, adapté pour le Prolog, de Dijkstra est le suivant :

1. Les idées de base pour ce paragraphe nous ont été fournies par notre collègue Bernard Glonneau. La référence est le livre de A.S.Tanenbaum : Computer Networks,2nd edition, Prentice Hall, 1988, pp.289-292.

```
Initialisation :
OUVERT ⟵  [x1,x1,0], où x1 le sommet initial.
FERME ⟵  []
tableRoutage ⟵  []

Tant que ( OUVERT ≠  [] ) faire
Début
    Si  OUVERT = [(x,y,d(x) | Reste], alors
    Début
        OUVERT ⟵  Reste,
        FERME ⟵  [(x,y,d(x) | FERME]
        tableRoutage ⟵  [(x,y,d(x) |  tableRoutage]
    Fin

    Si ( EXPANSION(x) ≠  [] ), alors
    Début
        Pour tout w ∈  EXPANSION(x) faire
        Début
            Si ( (w,_,_) ∉  OUVERT U FERME ), alors
            Début
                d(w) ⟵  d(x) + p(x,w)
                OUVERT ⟵  [(w,x,d(w)) | OUVERT]
            Fin
            Si ( (w,_,d(w)) ∈  OUVERT ), alors
            Début
                Si ( d(w) > d(x) + p(x,w) ), alors
                Début
                    OUVERT ⟵  OUVERT -{(w,_,d(w))}
                    d(w) ⟵  d(x) + p(x,w)
                    OUVERT  ⟵  [(w,x,d(w)) | OUVERT]
                Fin
            Fin
            Si ( (w,_,d(w)) ∈  FERME ), alors
            Début
                Si ( d(w) > d(x) + p(x,w) ), alors
                Début
                    FERME ⟵  FERME -{(w,_,d(w))}
                    d(w) ⟵ d(x) + p(x,w)
                    OUVERT  ⟵  [(w,x,d(w)) | OUVERT]
                Fin
            Fin

        Fin
    Fin

    Trier OUVERT selon les valeurs ascendantes de d(x).

Fin

Utiliser la liste tableRoutage pour reconstruire le chemin optimal.
```

Le programme complet est le suivant :

PROGRAMME 13.1.1

```
/* Base de donn\'ees */

gr(a,b,2).
gr(a,g,6).
gr(b,e,2).
gr(b,c,7).
gr(g,e,1).
gr(g,h,4).
gr(e,f,2).
gr(f,c,3).
gr(f,h,2).
gr(c,d,3).
gr(h,d,2).
sommetInit(a).

gr(X,Y,D) :- not sommetInit(X), gr(Y,X,D).


/* Programmes utilitaires */

membre((X,Y,Z),[(X,Y,Z) | Ouvert]).
membre((X,Y,Z),[(X1,Y1,Z1) | Ouvert]):- membre((X,Y,Z),Ouvert).

suppres((X,Y,Z),[(X,Y1,Z1) | R],R).
suppres((X,Y,Z),[(F,G,W) | R],[(F,G,W) | R1]) :-
suppres((X,Y,Z),R,R1).

conc([],L2,L2).
conc([T | R],L2,[T | Rs]) :- conc(R,L2,Rs).

for(X,Y) :- X, Y, fail.
for(X,Y).

/* Affichage du routage */

decort2([]).
decort2([(X,Y) | Ouvert]) :- write(X), tab(2), write(Y),nl,
decort2(Ouvert).

decort3([]).
decort3([(X,Y,D) | Ouvert]) :- write(X), tab(2), write(Y), tab(2),
                               write(D),nl, decort3(Ouvert).

/* Inverse d'une liste */

reverse([T | R],Acc,Res) :- reverse(R,[T | Acc],Res).
reverse([],Res,Res).

/* Remplacement d'un \'el\'ement d'une liste par un autre \'el\'ement */

remplace((X,Y),[(X,Y) | R],(Z,W),[(Z,W)\(\mid\)R]).
remplace((X,Y),[(F,G) | R],(Z,W),[(F,G) | R1]) :- remplace((X,Y),R,(Z,W),R1).

/* Remplacement d'une liste par une autre liste */

transfert([],[]).
transfert([X | R],[X | R1]) :- transfert(R,R1).

/*  Tri d'une liste  */
```

```
tri([(X,Y,Val) | R],L) :- partition(R,(X,Y,Val),P,G),
                          tri(P,Ps), tri(G,Gs),
                          conc(Ps,[(X,Y,Val) | Gs],L).
tri([],[]).

partition([(Z,W,V) | Xs],(X,Y,Val),[(Z,W,V) | Ps],G)  :-
                       V < Val,
                       partition(Xs,(X,Y,Val),Ps,G).
partition([(Z,W,V) | Xs],(X,Y,Val),P,[(Z,W,V) |  Gs]) :-
                       V >= Val,
                       partition(Xs,(X,Y,Val),P,Gs).
partition([],(X,Y,Val),[],[]).


/* MAJ de l'OUVERT */

majOuvert([],S,Dist,Ferme,Ferme,Ouvert,Ouvert).
majOuvert([(Ss,Pds)|Expansion],S,Dist,Ferme,Ferme1,Ouvert,[(Ss,S,Dist1)|Ouvert1]):-
                       not membre((Ss,Pred,Val),Ouvert),
                       not membre((Ss,Pred,Val),Ferme),
                       Dist1 is Dist+Pds,
                       majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).
majOuvert([(Ss,Pds)|Expansion],S,Dist,Ferme,Ferme1,Ouvert,[(Ss,S,Dist1)|Ouvert2]):-
                       membre((Ss,Pred,Val),Ouvert),
                       Dist1 is Dist+Pds,
                       Dist1 < Val,
                       suppres((Ss,Pred,Val),Ouvert,Ouvert1),
                       majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert1,Ouvert2).
majOuvert([(Ss,Pds)|Expansion],S,Dist,Ferme,Ferme1,Ouvert,Ouvert1):-
                       membre((Ss,Pred,Val),Ouvert),
                       Dist1 is Dist+Pds,
                       Dist1 $\geq$ Val,
                       majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).
majOuvert([(Ss,Pds)|Expansion],S,Dist,Ferme,Ferme2,Ouvert,[(Ss,S,Dist1)|Ouvert1]):-
                       membre((Ss,Pred,Val),Ferme),
                       Dist1 is Dist+Pds,
                       Dist1 < Val,
                       suppres((Ss,Pred,Val),Ferme,Ferme1),
                       majOuvert(Expansion,S,Dist,Ferme1,Ferme2,Ouvert,Ouvert1).
majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme1,Ouvert,Ouvert1) :-
                       membre((Ss,Pred,Val),Ferme),
                       Dist1 is Dist+Pds,
                       Dist1 $\geq$ Val,
                       majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).

/* Calcul du routage pour un r\'eseau donn\'e.  Algorithme de Dijsktra */

routage([],Ferme,[]) :- write('Fin algorithme - Routage '), nl.
routage([(S,Pr,Dist)$\mid$Ouvert],Ferme,[(S,Pr,Dist)$\mid$ Lr]) :-
                       trouveTout(S,Expansion),
                       majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1),
                       tri(Ouvert1,Ouvert2),
                       routage(Ouvert2,[(S,Pr,Dist) | Ferme1],Lr).

trouveTout(X,R) :- asserta(grlst([])),
                   for(gr(X,Y,D),(write(Y), tab(2),
                   write(D),nl,
                   grlst(L), retract(grlst(V)),
                   asserta(grlst([(Y,D) | L])))),
                   grlst(R1),retract(grlst(V)),reverse(R1,R).
```

Exemple de question à poser pour ce programme :

```
? question :- sommetInit(S), routage([(S,S,0)],[],L), decort3(L).
```

13.2 Planification

Un problème de la robotique est la construction d'une suite d'actions – planification – qui permet au robot d'agir sur l'environnement. Bien sûr cette suite d'actions doit être cohérente et minimale, en ce sens que les actions ne remettent pas en cause les résultats des actions antérieures. Afin de réaliser la planification on doit pouvoir décrire le monde et son évolution en utilisant un ensemble des concepts. Essentiellement nous avons besoin de décrire les objets du monde et les actions qui permettent son évolution. À chaque instant le monde se trouve à un état donné. On peut passer d'un état à un autre par l'intermédiaire d'une action. En appliquant donc une suite d'actions on peut passer d'un état initial à un état final (but). La construction de la suite d'actions se fait à l'aide d'un générateur de plans d'actions.

Il existe des générateurs universels de plans d'actions, c'est-à-dire des générateurs qui peuvent s'appliquer à différents types de problèmes, comme par exemple STRIPS ou WARPLAN qui d'ailleurs a été écrit en `Prolog`. Le générateur que nous présentons ici est un simple générateur qui est universel. Son principal défaut est qu'il ne protège pas – contrairement à WARPLAN – la destruction d'un but déjà réalisé par la réalisation d'un autre but.

À titre d'exemple nous présentons ce générateur dans le cas du monde des blocs. Ce monde contient des blocs (cubes) qu'il s'agit de déplacer. La situation d'un cube se décrit à l'aide de deux prédicats : `libre(X)` qui indique si le bloc `X` est libre et `sur(X,Y)` qui indique si le bloc `X` se trouve sur le bloc `Y` et où `Y` peut aussi être la `table`. L'évolution du monde se fait à l'aide d'une seule action qui est le déplacement d'un cube A du cube B sur lequel se trouve au cube C : `deplacer(A,B,C)`. Les contraintes pour la réalisation de cette action sont `libre(A)`, `libre(C)`, `different(B,C)`. L'évolution du monde due à cette action se traduit par le retrait des certains faits et par l'ajout d'autres faits. Ainsi les faits qu'on supprime sont `libre(C)` et `sur(A,B)`. Les faits qu'on rajoute sont `libre(B)` et `sur(A,C)`.

Le programme `Prolog` de ce générateur est le suivant :

PROGRAMME 13.2.1

```
/* Le g\'en\'erateur de plans */

genplan(G,S) :- planif(G,S,S1), ecrire(S1).

/* Le programme qui effectue la planification */

planif([B | R],S,S2) :- realiseFait(B,S,S1), planif(R,S1,S2).
planif([],S,S).

/* Le programme qui r\'ealise un fait */

realiseFait(B,S,S) :- B.
realiseFait(B,S,S) :- sitSuc(B,S).
realiseFait(B,S,[Act | S1]) :-ajout(B,Act),action(Act,Cond),
                              planif(Cond,S,S1).
```

```
/* Teste si un fait peut \^etre obtenu par un \'etat qui
   succ\'ede imm\\'ediatement \'a l'\'etat pr\'esent.                    */

sitSuc(F,[Act | S]) :- ajout(F,Act).
sitSuc(F,[Act | S]) :- sitSuc(F,S), non(supp(F,Act)).
sitSuc(F,[init]) :- F.

/* D\'efinition de l'action et de ses contraintes pour le d\'eclechement  */

action(deplace(A,B,table),[sur(A,B),libre(A),ne(B,table)]).
action(deplace(A,B,C),[libre(C),sur(A,B),libre(A),ne(A,C)]).

/* Les effets d'une action    */

ajout(sur(A,C),deplace(A,B,C)).
ajout(libre(B),deplace(A,B,C)).
supp(sur(A,B),deplace(A,B,C)).
supp(libre(C),deplace(A,B,C)).

/*  Programmes utilitaires    */

ecrire([S1 | S2]) :- ecrire(S2), nl, write(S1), nl.
ecrire([]).

non(X) :- X, !, fail.
non(X).

/* Base de donn\'ees    */

sur(c,table).
sur(d,table).
sur(a,c).
sur(e,d).
sur(b,e).
libre(a).
libre(b).
libre(table).
```

Une question à poser est par exemple la suivante

```
?-question :- genplan([sur(a,table),sur(c,a),libre(c),
                       sur(b,table),sur(e,b),sur(d,e),libre(d)],[init]).
```

13.3 Programmation pour Sudoku

La programmation en Prolog de la résolution d'une grille de Sudoku est extremement facile à mettre en œuvre ; Nous allons principalement s'appuyer sur les contraintes que nous avons vu au chapitre 4.

Pour écrire le programme il faut partir des règles du jeu qui peuvent se resumer comme suit :

- Le sudoku est un plateau de 81 cases disposées sur un carré de 9 cases de côté.
- Le plateau est divisé en 9 sous-plateaux, appelés voisinages, de 9 cases chacun, disposés en des carrés de 3 cases de côté.
- Chaque case peut recevoir un nombre entier entre 1 et 9.

– Toutes les lignes du plateau doivent avoir des nombres différents.
– Toutes lescolonnes du plateau doivent avoir des nombres différents.
– Tous les voisinages du plateau doivent avoir des nombres différents.

On reprend ces règles et on écrit le programme.

Le programme est le suivant :

PROGRAMME 13.3.1

```
:- use_module(library('clp/bounds')).

/*
   test lit le fichier grille.pl dans lequel est stock\'ee la grille du jeu
   et lance le programme de r\'esolution.
*/
sudoku :- consult('c:/tmp/grille.pl'), go.

sudoku(Grille) :-
%  Affichage de la grille d'entr\'ee
        nl, writeln('Grille avant resolution :'), nl,
        affiche1(Grille), nl, writeln('Solution'), nl,
%  Suppression des sous-listes de la grille et stockage de grille sans sous-liste dans GrilleF
        flatten(Grille,GrilleF),
%  Tous les \'el\'ements de GrilleF sont des nombres entiers dans l'intervalle [1, 9]
        GrilleF in 1..9,
%  Calul de la transpos\'ee de la grille et stockage dans TGrille
        transpMat(Grille,TGrille),
%  D\'efinition des lignes de Grille
        [Ligne1,Ligne2,Ligne3,Ligne4,Ligne5,Ligne6,Ligne7,Ligne8,Ligne9] = Grille,
%  D\'efinition des colonnes de Grille (= lignes de TGrille)
        [Colonne1,Colonne2,Colonne3,Colonne4,Colonne5,Colonne6,Colonne7,Colonne8,Colonne9] = TG
%  D\'efinition des noms des cases de chaque ligne
        [X11,X12,X13,X14,X15,X16,X17,X18,X19] = Ligne1,
        [X21,X22,X23,X24,X25,X26,X27,X28,X29] = Ligne2,
        [X31,X32,X33,X34,X35,X36,X37,X38,X39] = Ligne3,
        [X41,X42,X43,X44,X45,X46,X47,X48,X49] = Ligne4,
        [X51,X52,X53,X54,X55,X56,X57,X58,X59] = Ligne5,
        [X61,X62,X63,X64,X65,X66,X67,X68,X69] = Ligne6,
        [X71,X72,X73,X74,X75,X76,X77,X78,X79] = Ligne7,
        [X81,X82,X83,X84,X85,X86,X87,X88,X89] = Ligne8,
        [X91,X92,X93,X94,X95,X96,X97,X98,X99] = Ligne9,
%  D\'efinition des voisinages qui sont des sous-grilles de dimension (3x3)
        [X11,X12,X13,X21,X22,X23,X31,X32,X33] = Vois11,
        [X14,X15,X16,X24,X25,X26,X34,X35,X36] = Vois12,
        [X17,X18,X19,X27,X28,X29,X37,X38,X39] = Vois13,
        [X41,X42,X43,X51,X52,X53,X61,X62,X63] = Vois21,
        [X44,X45,X46,X54,X55,X56,X64,X65,X66] = Vois22,
        [X47,X48,X49,X57,X58,X59,X67,X68,X69] = Vois23,
        [X71,X72,X73,X81,X82,X83,X91,X92,X93] = Vois31,
        [X47,X48,X49,X57,X58,X59,X67,X68,X69] = Vois32,
        [X77,X78,X79,X87,X88,X89,X97,X98,X99] = Vois33,
%  Les \'el\'ements des lignes, colonnes et voisinages ont des valeurs diff\'erentes
        all_different(Ligne1), all_different(Ligne2), all_different(Ligne3),
        all_different(Ligne4), all_different(Ligne5), all_different(Ligne6),
        all_different(Ligne7), all_different(Ligne8), all_different(Ligne9),
        all_different(Colonne1), all_different(Colonne2), all_different(Colonne3),
        all_different(Colonne4), all_different(Colonne5), all_different(Colonne6),
        all_different(Colonne7), all_different(Colonne8), all_different(Colonne9),
        all_different(Vois11), all_different(Vois12), all_different(Vois13),
        all_different(Vois21), all_different(Vois22), all_different(Vois23),
```

```
        all_different(Vois31), all_different(Vois32), all_different(Vois33),
%  Chaque variable de GrilleF esta ssign\'ee dans son domaine de variation
        label(GrilleF),
%  Afficage du r\'esultat
        affiche(Grille).

affiche([]).
affiche([H|T]) :- write(H),nl, affiche(T).

affiche1([]).
affiche1([H|T]) :- write('['),afficheVect(H), affiche1(T).

afficheVect([]) :- nl.
afficheVect([T|R]) :- ((number(T), write(T));(not(number(T))), write('_')),
                      ((R\==[],write(', '));(R==[],write(']'))),
                      afficheVect(R).

premiereCol([],[]).
premiereCol([[T|R]|Rs],[T|Ls]) :- premiereCol(Rs,Ls).

suiteCol([],[]).
suiteCol([[T|R]|Rs],[R|Ls]) :- suiteCol(Rs,Ls).

transpMat([[]|_],[]).
transpMat(R,[T|Rs]) :- premiereCol(R,T),suiteCol(R,H), transpMat(H,Rs).
```

Le programme grille.pl qui sert à écrire la grille à résoudre est le suivant :

PROGRAMME 13.3.2

```
go :-
  Grille = [
 [3,_,_,_,6,8,_,_,_],
 [_,7,_,_,_,_,_,_,_],
 [_,1,6,_,_,_,_,2,9],
 [_,_,_,_,_,_,_,9,_,8],
 [7,_,_,6,_,_,_,_,_],
 [1,8,_,_,_,5,_,_,4],
 [6,3,_,4,_,_,_,_,_],
 [_,_,_,_,_,3,_,2,_,_],
 [_,_,4,_,5,_,1,_,_]
 ],
 sudoku(Grille).

 /*  Pour \'ecrire une autre grille
    utilisez la grille vide suivante :

[
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_]
 ]
 */
```

Pour appeler ce programme on entre la question `?- sudoku`.

Table des matières