

## Formalisation de systèmes experts

### 1. PROLOG ET LES SYSTEMES EXPERTS

PROLOG est parfaitement adapté pour formaliser des systèmes experts. En effet, un système expert est un programme informatique simulant l'intelligence humaine dans un champ particulier de la connaissance ou relativement à une problématique déterminée. Or PROLOG a justement été conçu dans cette optique là puisqu'il a été fait par des chercheurs en intelligence artificielle.

Un système expert a trois composantes essentielles :

- 1 • une base de connaissances, formée des énoncés relatifs aux faits de tous ordres constitutifs du domaine
- 2 • un ensemble de règles de décision, consignait les méthodes, procédures et schémas de raisonnement utilisés dans le domaine
- 3 • un moteur d'inférence, sous-système qui permet d'appliquer les règles de décision à la base de connaissances.

Or ces trois points sont extrêmement simples à implémenter dans programme PROLOG :

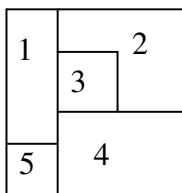
- 1 • la base de connaissances est constituée par les faits et quelques règles pour éviter l'énumération exhaustive de tous les faits
- 2 • les règles de décision sont des règles (au sens de PROLOG)
- 3 • le moteur d'interface est l'interpréteur PROLOG lui-même.

#### 1.1. Constitution de la base de connaissance

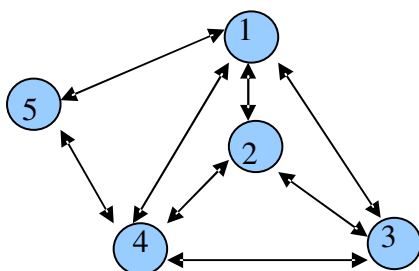
En guise d'exemple de système expert, nous allons formaliser le problème de coloriage de région. Les règles de ce problème sont les suivantes :

- 1 • Une surface est découpée en un certain nombre de régions de surfaces et de formes variables
- 2 • Chaque région doit être coloriée
- 3 • Deux régions adjacentes doivent avoir deux couleurs différentes

Nous essaierons de colorier les régions suivantes :



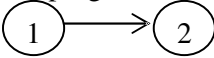
que nous représenterons plutôt par



La deuxième représentation permet de s'abstraire de toute géométrie :

- 1 •Un sommet représente une région
- 2 •Une arête reliant deux sommets signifie que les régions équivalentes sont adjacentes

C'est cela qu'on va traduire dans le programme PROLOG par le prédicat adjacent

Ex : adjacent(1,2). signifie 2 

ce qui n'est pas exactement « La région 1 est adjacente à la région 2. »

## 1.2. Enumération exhaustive

La première solution pour constituer la base de connaissance est de faire une énumération exhaustive des régions adjacentes.

### Programme 1: Base de connaissances exhaustive

%Liste exhaustive des regions adjacentes

adjacent (1, 2) . adjacent (2, 1) .

adjacent (1, 3) . adjacent (3, 1) .

adjacent (1, 4) . adjacent (4, 1) .

adjacent (1, 5) . adjacent (5, 1) .

adjacent (2, 3) . adjacent (3, 2) .

adjacent (2, 4) . adjacent (4, 2) .

adjacent (3, 4) . adjacent (4, 3) .

adjacent (4, 5) . adjacent (5, 4) .

Cette base de connaissance fonctionne parfaitement.

Ex : Les régions 1 et 2 sont adjacentes :

```
?- adjacent (1, 2) , adjacent (2, 1) .
Yes
```

La région 2 n'est pas adjacente à la région 11 (qui n'existe pas)

```
?- adjacent (2, 11) .
No
```

La région 2 n'est pas adjacente à la région 5

```
?- adjacent (2, 5) .
No
```

Cependant, on voit très bien que ce genre de base de connaissances est limité : si le problème devient trop complexe, on ne peut pas saisir à la main toutes les possibilités. De plus cette écriture sous-exploite le potentiel de PROLOG qui grâce aux règles peut grandement alléger cette écriture.

## 1.3. Ecritures condensées

Nous allons condenser l'écriture exhaustive en remplaçant certains faits par des règles ce qui comme nous le verrons n'est pas sans danger.

### 1.3.1. Règle de commutation

Essayons tout d'abord d'introduire une règle de commutativité. En effet, cette règle si naturelle pour nous quand on parle de coté adjacent fait cruellement défaut à PROLOG et permettrait de réduire de moitié le nombre de faits.

**Programme 2: Base de connaissances commutative (erronée)**

*/\*Liste des régions adjacentes 1 fois (sans commutation)\*/*

adjacent (1,2) .

adjacent (1,3) .

adjacent (1,4) .

adjacent (1,5) .

adjacent (2,3) .

adjacent (2,4) .

adjacent (3,4) .

adjacent (4,5) .

*/\*La regle de commutation \*/*

adjacent (X,Y) :-adjacent (Y,X) .

Cette règle de commutation fonctionne très bien dans certains cas.

**Ex :** Le programme détecte parfaitement les régions adjacentes :

?- adjacent (2,4) .

Yes

?- adjacent (4,2) .

Yes

2 et 4 sont bien deux régions adjacentes

Malheureusement, dans d'autres cas elle engendre des boucles infinies.

**Ex :** Le programme boucle systématiquement pour détecter des régions non adjacentes :

?- adjacent (2,5) .

Action (h for help) ? abort

Execution Aborted

**Ex :** Enumérer toutes les régions adjacentes à 1

?- adjacent (1,X) . X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5

...

PROLOG ne boucle pas tout seul mais il retourne toujours la même série de solutions tant que l'utilisateur lui en demande encore : il ne s'arrête pas tout seul pour signifier qu'il n'y en a plus d'autres.

En fait, la règle de commutativité remplit parfaitement son rôle pour déterminer que deux régions sont adjacentes car soit elle n'est jamais invoquée soit elle est utilisée une seule fois avant de trouver le fait permettant d'arrêter la recherche.

**Ex :**

```
[debug] ?- adjacent(2,4) .
T Call: ( 8) adjacent(2, 4)
T Exit: ( 8) adjacent(2, 4)
```

PROLOG trouve la preuve directement dans les faits.

```
[debug] ?- adjacent(4,2) .
T Call: ( 8) adjacent(4, 2)
T Call: ( 9) adjacent(2, 4)
T Exit: ( 9) adjacent(2, 4)
T Exit: ( 8) adjacent(4, 2)
```

Cette fois il est obligé d'appliquer la règle de commutativité 1 fois (2ème call) puis il trouve un fait pour elle, il en sort et il répond à la requête.

Par contre, si la requête porte sur deux régions non adjacentes, PROLOG va appeler une infinité de fois la règle de commutativité.

**Ex :**

```
[debug] ?- adjacent(2,5) .
T Call: ( 8) adjacent(2, 5)
T Call: ( 9) adjacent(5, 2)
T Call: (10) adjacent(2, 5)
T Call: (11) adjacent(5, 2)
T Call: (12) adjacent(2, 5)
T Call: (13) adjacent(5, 2)
T Call: (14) adjacent(2, 5)
T Call: (15) adjacent(5, 2)
T Call: (16) adjacent(2, 5)
T Call: (17) adjacent(5, 2)
T Call: (18) adjacent(2, 5) ...
```

Les zone 2 et 5 n'étant pas adjacentes, ni adjacent(2,5), ni adjacent(5,2) ne fait partis des faits. La première action de PROLOG est de regarder si la requête est un fait. Puisque ce n'est pas le cas, il applique la règle de commutation en espérant que cela lui permet de conclure. Il re-parcourt donc les faits mais il ne trouve encore aucune preuve pour répondre à la requête. Donc il applique une nouvelle fois la règle de commutation.

Dans le cas d'une énumération de zones adjacentes à une autre, la règle de commutation crée en quelque sorte une infinité de faits en répétant les faits une fois à l'endroit, une fois en commutant les paramètres et ceci une infinité de fois.

**Ex :**

```
debug] ?- adjacent(1,X) .
T Call: ( 7) adjacent(1, _G304)
T Exit: ( 7) adjacent(1, 2) X = 2 ;
T Exit: ( 7) adjacent(1, 3) X = 3 ;
T Exit: ( 7) adjacent(1, 4) X = 4 ;
T Exit: ( 7) adjacent(1, 5) X = 5 ;
T Redo: ( 7) adjacent(1, _G304)
T Call: ( 8) adjacent(_G304, 1)
T Redo: ( 8) adjacent(_G304, 1)
T Redo: ( 8) adjacent(_G304, 1)
T Call: ( 9) adjacent(1, _G304)
T Exit: ( 9) adjacent(1, 2)
T Exit: ( 8) adjacent(2, 1)
```

```
T Exit: ( 7) adjacent(1, 2) X = 2
...
```

Au 1<sup>er</sup> passage, PROLOG trouve tous les faits correspondant à la requête (les régions 2, 3, 4 et 5 sont adjacentes à la 1). Ensuite il appelle la règle de commutation `adjacent(X,Y) :- adjacent(Y,X)` et relit les faits. Cette fois aucun ne convient mais il arrive de nouveau à la règle de commutation. Il l'applique et recommence à lire les faits. Après deux commutations, il est revenu (malheureusement il ne le sait pas) à l'état initial ; il trouve donc les mêmes solutions qu'au premier passage...

**Rem :** Il existe des méthodes pour prévenir ces boucles, cependant pour des raisons d'efficacité, elles ne sont pas utilisées dans PROLOG.

### 1.3.2. Énumération par variables

Une autre façon de condenser l'écriture est de faire l'énumération des quatre premiers faits (et leurs images commutées) par l'intermédiaire d'une variable. En effet, on remarque que ces faits sont tous de la forme `adjacent(1,x)` avec  $2 \leq x \leq 5$ .

#### Programme 3: Base de connaissances avec énumération par variable (erronée)

```
/*Liste des regions adjacentes autres que la 1 adjacent(2,3). */
```

```
adjacent(2,4) .
```

```
adjacent(3,4) .
```

```
adjacent(4,5) .
```

```
adjacent(3,2) .
```

```
adjacent(4,2) .
```

```
adjacent(4,3) .
```

```
adjacent(5,4) .
```

```
/* Règles concernant la région 1 */
```

```
adjacent(1,X) :- X>=2, X<=5 .
```

```
adjacent(X,1) :- X>=2, X<=5 .
```

Ce programme marche très bien avec des requêtes simples

**Ex :** On peut savoir que les régions 3 et 1 sont adjacentes :

```
?- adjacent(1,3), adjacent(3,1) .
```

```
Yes
```

ou que les régions 5 et 2 ne le sont pas :

```
?- adjacent(5,2) .
```

```
No
```

Par contre, une erreur peut apparaitre lorsqu'on emploie des requêtes un peu plus complexes.

**Ex :** Enumérer toutes les régions adjacentes à 2

```
?- adjacent(2,X) .
```

```
X = 3 ;
```

```
X = 4 ;
```

```
X = 1 ;
```

```
No
```

La région 2 est bien adjacente aux régions 1, 3 et 4.

**Ex :** Enumérer toutes les régions adjacentes à 1

```
?- adjacent(1,X) .
ERROR: Arguments are not sufficiently instantiated
^ Exception: (7) _G158>=2
? abort
% Execution Aborted
```

PROLOG refuse d'exécuter la requête. En effet, il s'agit d'un des rares contrôles de PROLOG. Sans ce contrôle, il regarderait `adjacent(1,X)` pour tous les  $x \geq 2$  puis pour tous les  $x \leq 5$  ; or il y en a une infinité dans les deux cas et il entrerait donc dans une boucle infinie.

Il existe une solution à ce problème. PROLOG fournit un système pour borner une variable entière : `between`.

Ainsi, il faut remplacer les deux règles concernant la région 1 par :

*/\* Regles concernant la region 1\*/*

```
adjacent(1,X):-between(2,5,X) .
```

```
adjacent(X,1):-between(2,5,X) .
```

### 1.3.3. Relations entre paramètres

On remarque que concernant les régions (2,3), (3,4) et (4,5) les deux paramètres X et Y du prédicat `adjacent` sont liés par la relation  $Y=X+1$ . On pourrait donc écrire explicitement cette relation.

#### Programme 4 : Base de connaissances avec relation entre paramètres (erronée)

*% Regions adjacentes explicitement declarees*

```
adjacent(2,4) .
```

```
adjacent(4,2) .
```

*% Regles concernant la region 1*

```
adjacent(1,X):-between(2,5,X) .
```

```
adjacent(X,1):-between(2,5,X) .
```

*% Regles concernant les couples (2,3), (3,4) et (4,5) adjacent(X,X+1). adjacent(X+1,X).*

Ce programme présente plusieurs défauts :

1 • D'abord, il est syntaxiquement incorrect. En effet, PROLOG ne reconnaît pas «  $X+1$  » comme étant la valeur de X augmenté de 1. Pour cela, il dispose du mot clé `is` :

```
adjacent(X,Y):-Y is X+1 .
```

```
adjacent(X,Y):-Y is X+1 .
```

2 • Ensuite, comme nous l'avons vu au chapitre précédent, il faut spécifier que X est un entier borné en utilisant `between`

```
adjacent(X,Y):-between(2,4,X), Y is X+1 .
```

```
adjacent(X,Y):-between(2,4,X), Y is X+1 .
```

Finalement, nous avons pu condenser la base de connaissances en :

#### Programme 5 : Base de connaissances condensée (finale)

*% Regions adjacentes explicitement declarees*

```
adjacent(2,4) .
```

```
adjacent (4, 2) .
```

```
% Regles concernant la region 1
```

```
adjacent (1, X) :-between (2, 5, X) .
```

```
adjacent (X, 1) :-between (2, 5, X) .
```

```
% Regles concernant les couples (2,3), (3,4) et (4,5)
```

```
adjacent (X, Y) :-between (2, 4, X), Y is X+1 .
```

```
adjacent (Y, X) :-between (2, 4, X), Y is X+1 .
```

Cette écriture condensée est exactement équivalente au programme exhaustif page 11.

**Ex :** Test sur deux régions adjacentes

```
?- adjacent (1, 2), adjacent (2, 1) .
```

```
Yes
```

1 et 2 sont bien adjacentes

**Ex :** Test sur deux régions non adjacentes

```
?- adjacent (2, 5) .
```

```
No
```

2 et 5 ne sont effectivement pas adjacentes

**Ex :** Enumération de régions adjacentes

```
?- adjacent (1, X) .
```

```
X = 2 ;
```

```
X = 3 ;
```

```
X = 4 ;
```

```
X = 5 ;
```

```
No
```

1 est adjacente exactement à 2, 3, 4 et 5

#### 1.4. Mise en place des règles de décision

Pour les règles de décision, on introduit deux nouveaux prédicats :

- `conflit (Coloriage)` qui permet de voir si un coloriage des régions respecte les contraintes que nous nous sommes fixées
- `conflit (X, Y, Coloriage)` qui permet de savoir quelles régions adjacentes ont la même couleur

Un coloriage des régions est une fonction qui à chaque région associe une couleur. Elle est réalisée par le prédicat :

```
color (Region, Couleur, Coloriage)
```

```
adjacent (2, 4) .conflit (Coloriage) :-adjacent (X, Y), color (X, Couleur, Coloriage),  
color (Y, Couleur, Coloriage) .
```

```
conflit (X, Y, Coloriage) :-adjacent (X, Y), color (X, Couleur, Coloriage), color (Y,  
Couleur,
```

```
adjacent (4, 2) .
```

```
adjacent (1, X) :-between (2, 5, X) .
```

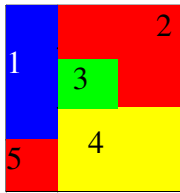
```
adjacent (X, 1) :-between (2, 5, X) .
```

```
adjacent (X, Y) :-between (2, 4, X), Y is X+1 .
```

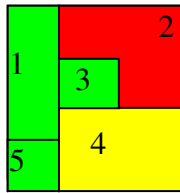
```
adjacent (Y, X) :-between (2, 4, X), Y is X+1 .
```

Trois coloriages ont été définis directement dans le programme :

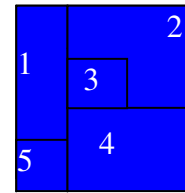
coloriage1



coloriage2



coloriage3



### Programme 6: Formalisation d'un système expert (complète)

% Description des zones

1 2

% Règles de décisions (Coloriage).

% Exemples de coloriages possibles

*/\* Coloriage sans conflit \*/*

color(1,bleu,coloriage1).

color(2,rouge,coloriage1).

color(3,vert,coloriage1).

color(4,jaune,coloriage1).

color(5,rouge,coloriage1).

*/\* Coloriage avec conflit \*/*

color(1,vert,coloriage2).

color(2,rouge,coloriage2).

color(3,vert,coloriage2).

color(4,jaune,coloriage2).

color(5,vert,coloriage2).

*/\* Coloriage avec conflit \*/*

color(1,bleu,coloriage3).

color(2,bleu,coloriage3).

color(3,bleu,coloriage3).

color(4,bleu,coloriage3).

color(5,bleu,coloriage3).

Nous pouvons maintenant vérifier si un coloriage est valide et dans le cas contraire, connaître les régions adjacentes qui ont la même couleur.

**Ex :** Un coloriage sans conflit

?- conflit(coloriage1).

No

Le coloriage 1 ne possède effectivement pas de régions adjacentes de même couleur.



**Ex :** Un coloriage avec conflit

```
?- conflit(coloriage2).  
Yes
```

Les régions adjacentes de même couleur sont :

```
?- conflit(X,Y,coloriage2),between(X,5,Y).  
X = 1 Y = 3 ;  
X = 1 Y = 5 ;  
No
```

La région 1 a la même couleur que la région 3 et la région 5.

**Ex :** Toutes les régions de la même couleur

```
?- conflit(coloriage3).  
Yes
```

Toutes les régions adjacentes entrent en conflit

```
?- conflit(X,Y,coloriage3), between(X,5,Y).  
X = 2 Y = 4 ;  
X = 1 Y = 2 ;  
X = 1 Y = 3 ;  
X = 1 Y = 4 ;  
X = 1 Y = 5 ;  
X = 2 Y = 3 ;  
X = 3 Y = 4 ;  
X = 4 Y = 5 ;  
No
```

Effectivement, il s'agit bien là de toutes les régions adjacentes.