## 8. Programmation avancée

### 8.1. Programmation par essais successifs *(VIDEO-PPT 4.2)*

Le tri par permutation abordé avec le coupe-choix est intéressant pour la méthode utilisée et sa structure. En effet, dans cette approche « *générer-tester* » on ne met pas en place un algorithme qui conduit directement à une solution. L'idée est de proposer *successivement des solutions* possibles puis, pour chaque solution proposée, de la tester afin de ne retenir que la (les) solution(s) effective(s). Dans le cas du tri, il n'y a qu'une seule solution.

En Prolog, c'est le mécanisme de retour arrière qui met en œuvre les essais et tests successifs :

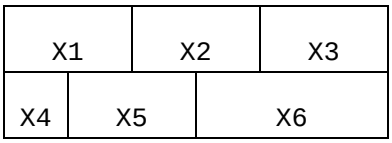
```
problème(X) :- generer(X) , tester(X).
```

☝ ***Attention** : L'ensemble des solutions possibles peut se révéler très grand (au point que cette technique peut être inexploitable), un des enjeux est de minimiser la génération des solutions possibles….*

➢ *Premier exemple* : Coloriage d'une carte

*Il s'agit de colorier les différentes zones d'une carte de telle sorte que deux zones ayant une frontière commune aient des couleurs différentes. Une telle carte peut se représenter par un graphe planaire. Il a été démontré qu'une solution n'utilisant que 4 couleurs est toujours possible.*

Soit la carte suivante :




Pour colorier cette carte avec 4 couleurs, il faut indiquer dans la base de connaissances quelles sont les couleurs disponibles :

```
couleur(bleu).
couleur(blanc).
couleur(rouge).
couleur(vert).
```

Puis essayer ces couleurs en respectant les conditions liées aux frontières des zones `Xi` :

```
colorier(X1,X2,X3,X4,X5,X6) :-  couleur(X1), couleur(X2),
                                 couleur(X3), couleur(X4),
                                 couleur(X5), couleur(X6),
                                 X1\=X2, X1\=X4, X1\=X5, X2\=X3,
                          X2\=X5, X2\=X6, X3\=X6, X4\=X5, X5\=X6.
```

Ce programme génère un grand nombre de solutions.

➢ *Deuxième exemple* : Résolution d'une énigme

*Trois amis se trouvent premier, second et troisième d'un concours de programmation. Ils ont tous une nationalité différente, un prénom différent et pratiquent un sport différent. Michael aime le basket et a fait mieux que l'Américain. Simon, l'Israélien, a fait mieux que le tennisman. Le joueur de cricket est le premier. Qui est l'Australien ? Quel est le sport pratiqué par Richard* ?

Il faut identifier le nom, le sport et la nationalité de chacun. Les valeurs possibles sont respectivement, `[michael, simon, richard]`, `[tennis, cricket, basket]`, `[israelien, australien, americain]`. L'objectif est également de déterminer le classement au concours de programmation.

Pour modéliser ce problème, une liste `L` de triplets *(prénom, nationalité, sport)* est construite, l'ordre des triplets dans `L` correspond au résultat du concours.

La méthode utilisée est de type « générer-tester ». Différents triplets vont être générés au moyen du prédicat `permut` (*cf* « *triParPemutation.pl* »). Il faut ensuite vérifier que les conditions du problème sont satisfaites.

```
enigme(L) :- L = [[P1, N1, cricket], [P2, N2, S2], [P3, N3, S3]],
         permut([michael, simon, richard], [P1,  P2, P3]),
         permut([israelien, australien, americain],[N1, N2, N3]),
         permut([tennis, basket], [S2, S3]),
         avant([michael, N4, basket], [P5, americain, S5], L),
         avant([simon, israelien, S6], [P7, N7, tennis], L).

avant(X, Y, [X, Y|L]).
avant(X, Y, [X, Z |Q]) :- avant(X, Y, [X|Q]).
avant(X, Y, [Z|Q]) :- avant(X, Y, Q).


?- enigme(L).
L=[[simon,  israelien,  cricket],  [michael,  australien,  basket],
[richard, americain, tennis]]
```

➢ *Troisième exemple* : Placer N reines sur un échiquier

*Le problème consiste à placer 4 reines sur un échiquier 4x4. Le résultat attendu est une liste de 4 couples indiquant la position `[Ligne, Colonne]` des 4 reines bien placées.*

Une solution naïve consiste à générer tous les quadruplets de couples possibles et, pour chacun d'eux, vérifier que les conditions « *ne pas être sur la même ligne, sur la même colonne et sur la même diagonale* » sont vérifiées.

Pour limiter les essais, comme une seule reine peut être sur une ligne et une colonne, dès lors qu'une ligne et une colonne ont été utilisées pour le placement d'une reine, celles-ci ne sont plus disponibles pour générer les positions des reines suivantes.

Une reine est placée en première colonne, puis une seconde reine est placée dans la seconde colonne, en vérifiant qu'elle ne se trouve sur une même diagonale que la reine déjà placée. Le processus est itéré pour placer les reines suivantes sur les colonnes suivantes sans qu'elles soient en prise avec les reines déjà positionnées.

Le prédicat sera de la forme :

```
reines(Ldisponibles, Cdisponibles, Rplacees, Resultat)
```

D'où le programme :

```
reines([], [], R, R).

reines(Ldisponibles, [C1 | Crestants], Rplacees, R) :-
   enlever(L1, Ldisponibles, Lrestants),
   compatible([L1, C1], Rplacees),
   reines(Lrestants, Crestants, [[L1, C1]|Rplacees], R).
```

La structure de ce programme est intéressante car elle met en œuvre à la fois un processus itératif (au moyen d'une récursivité terminale) pour placer chacune des reines et un processus d'essais successifs grâce au retour arrière sur le prédicat `enlever` utilisé ici pour générer des positions possibles sur les colonnes disponibles.

Le prédicat `compatible([L1, C1], Rplacees)` vérifie que la position `[L1, C1]` ne se trouve pas sur une diagonale avec les couples de positions `[L, C]` des reines placées de la liste `Rplacees`.

```
compatible([L1, C1], []).

compatible([L1, C1], [[L2, C2]|Q]) :- noDiag([L1, C1], [L2, C2]),
                                      compatible([L1, C1], Q).

noDiag([L1, C1], [L2, C2]) :-   X1 is C1-C2, Y1 is L1-L2,
                                abs(X1, X2),  abs(Y1, Y2),
                                X2 =\= Y2.

abs(X, Y) :- X < 0, Y is -1 * X.

abs(X, X) :- X >= 0.
```