



Logique - Projet

TAYLOR MATT / SOCHAJ YOANN

15.05.2020

Sommaire:

- I. [Présentation de notre programme](#)
- II. [Temps de notre programme NON optimisé](#)
- III. [Optimisation de notre programme](#)
- IV. [Comparaison des temps entre non optimisé et optimisé](#)
- V. [Conclusion et résultat avec la matrice Hardcore](#)

I. Présentation de notre programme / TP-6

Ce projet a été vraiment très captivant et intéressant pour nous deux. La compréhension du sujet nous a pris pas mal de temps au début. L'exemple donné sur le sujet était crucial pour comprendre de quoi il s'agissait. Après avoir suivi les étapes données nous avons décidé d'en faire une clause pour chacune pour faciliter la tâche.

Avant de commencer le projet en soit, nous avons suivi les instructions du TP-6 pour nous aider à démarrer. Nous avons décidé d'écrire les 4 clauses pour le **XOR**.

```
boolean(v).  
boolean(f).  
  
xor(v,f,R):-  
    R=v.  
xor(f,v,R):-  
    R=v.  
xor(v,v,R):-  
    R=f.  
xor(f,f,R):-  
    R=f.
```

Concernant notre manière de représenter les vecteurs et les matrices dans Prolog nous avons choisi cette notation:

Vecteur: une liste

Matrice: une liste de listes

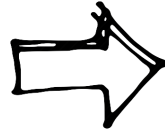
```
%Vecteur: [v,f,v,f,v,...,f]  
%Matrice: [ [v,v,f], [f,f,v], [f,v,v], [v,v,v] ]
```

Pour la 3ème question notre clause **vecteur(X,L)** permet de renvoyer tous les vecteurs possible pour une **longueur L** passe en paramètre, voici un exemple:

```
vecteur(X,L):-
    vecteur(X,[],L).

vecteur(X,X,0).

vecteur(X,Acc,L):-
    L>0,
    boolean(Z),
    Acc2=[Z|Acc],
    L2 is L-1,
    vecteur(X,Acc2,L2).
```



```
| ?- vecteur(V,2).
V = [v,v] ? ;
V = [f,v] ? ;
V = [v,f] ? ;
V = [f,f] ? ;
(1 ms) no
```

Beaucoup de clauses vu en TP / TD nous ont aidé tout au long de notre projet, certaines ont dû être modifiées pour s'adapter.

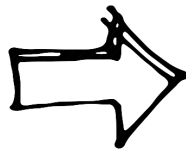
La fonction **long(L,X)**, **longll(L,X)**, **ajoute_fin(X,L1,L2)** et **n_ieme(L,Index,X)**

Le "ll" à la fin de "longll" signifie liste de listes (matrice) et donne la longueur d'une sous-liste d'une liste (nombre de colonnes dans une matrice).

```
long([], 0).

long([_|T], N) :-
    long(T, N1),
    N is N1+1.

longll([H|_], X) :-
    long(H,X).
```



```
| ?- long([4,2,0],L).
L = 3
yes
| ?- longll([[1,2], [4,5], [6,8]], L).
L = 2
yes
```

```
ajoute_fin(X,[],[X]).

ajoute_fin(X,[Y|L1],[Y|L2]):-
    ajoute_fin(X,L1,L2).
```

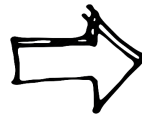
```
n_ieme(L,Index,X):-
    n_acc(L,Index,1,X).

n_acc([H|_],Y,Y,H).

n_acc([_|T],Index,Acc,X):-
    Index\==Acc,
    Acc2 is Acc+1,
    n_acc(T,Index,Acc2,X).
```

Pour la première étape du produit à *gauche* d'une matrice par un vecteur il faut créer un **vecteur** de longueur donnée qui ne **contient que des faux**. Voici notre clause et un exemple:

```
creerVecteur([],0).  
creerVecteur(R,N):-  
    creerVecteur(R,[],N).  
creerVecteur(R,R,0).  
creerVecteur(R,Acc,N):-  
    N>0,  
    NI is N-1,  
    creerVecteur(R,[f|Acc],NI).
```



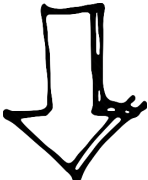
```
| ?- creerVecteur(V,4).  
V = [f,f,f,f] ? ;  
no
```

Avant d'aller plus loin nous avons aussi créé une clause **xorliste** qui renvoie le **XOR de 2 listes** passées en paramètre. La voici:

```
| ?- xorliste([v,f,v,v], [v,f,f,v], R).  
R = [f,f,v,f] ? ;
```

C'est ici que les choses se compliquent légèrement mais toutes les clauses écrites jusqu'à présent vont nous permettre d'assembler les clauses principales **produit(V,M,R)** et **solution(M,V)**. Le fonctionnement des ces clauses est bien expliqué en commentaire dans le programme.

Premièrement la clause **produit(V,M,R)**. Prenons le produit donné sur l'énoncé du problème pour vérifier notre résultat. Pour notre premier produit non optimisé, nous utilisons la clause **N-IÈME()** et un compteur pour récupérer la N-ième sous liste et on faisait le **XOR()** entre l'accumulateur et la sous liste courant si il y avait un vrai dans le vecteur V et si dans l'accumulateur il y avait une liste. Si l'accumulateur est la liste vide on ajoute la liste courante dans l'accumulateur, et si c'était un f dans V alors on passer à la suite sans rien faire.

$$(f \ f \ v \ v) \times \begin{pmatrix} v & f \\ f & v \\ v & v \\ f & v \end{pmatrix} = (v \ f).$$


```
produit(V,M,R):-
    produitAcc(V,M,[],1,R).

produitAcc([],_,R,_,R).

produitAcc([H1|T1],M,Acc,Compte,R):-
    H1==v,
    Acc==[],
    n_ieme(M,Compte,X),
    Compte1 is Compte+1,
    produitAcc(T1,M,X,Compte1,R).

produitAcc([H1|T1],M,Acc,Compte,R):-
    H1==v,
    Acc\==[],
    n_ieme(M,Compte,X),
    xorliste(X,Acc,S),
    Compte1 is Compte+1,
    produitAcc(T1,M,S,Compte1,R).

produitAcc([H1|T1],M,Acc,Compte,R):-
    H1==f,
    Compte1 is Compte+1,
    produitAcc(T1,M,Acc,Compte1,R).
```

```
| ?- produit([f,f,v,v],[[v,f],[f,v],[v,v],[f,v]],X).
X = [v,f] ? ;
```

Puis sans préciser le nombre de "vrais" souhaité dans notre vecteur V on peut également tester notre clause **solution(M,V)**. Il est bien en accord avec l'énoncé.

```
| ?- solution([[v,f],[f,v],[v,v],[f,v]],_,V).
V = [v,f,v,v] ? ;
V = [f,v,f,v] ? ;
V = [v,v,v,f] ? ;
V = [f,f,f,f] ? ;
```


III. Optimisation de notre programme

Dans un premier temps nous voulions tenter d'optimiser la clause **XOR**, car celle ci va être appelé un bon nombre de fois durant le programme. Notre ancien **XOR** était assez naïf on pouvait donc faire mieux. Notre solution a était d'au lieu de faire 4 cas différents en faire seulement deux. Remarquons qu'un **XOR** est vrai quand les deux variables sont différentes. Nous arrivons donc à ce nouveau **XOR**:

```
xor(A,A,R):-  
R=f.  
  
xor(A,B,R):-  
A\=B,  
R=v.
```

Cette nouvelle clause nous a déjà permis de gagner du temps sur l'exécution de **SOLUTION()**. Pour le niveau facile on a une moyenne de 2484 ms, pour le niveau moyen on a 9521 ms et pour le niveau difficile 57163 ms. Ceci a déjà bien raccourci le temps d'exécution (de 16 secondes pour le niveau difficile) mais nous pouvons encore faire mieux !

Dans un second temps l'optimisation la plus importante était sur la clause **PRODUIT()**. Nous voulions réduire le temps d'exécution en essayant de limiter le nombre de clause qu'on appelle. Notre première clause a été codé en fonction de notre première pensée comme le **XOR()** qui était très classique. Mais en réfléchissant plus loin nous avons remarqué qu'on pouvait retirer la clause **N-IÈME()**

et donc notre compteur. Nous arrivons donc à ce nouveau **PRODUIT()**:

```
produit(V,M,R):- %produit d'une ma  
produitAcc(V,M,[],R). % pas envie  
  
produitAcc([],[],R,R). % quand on  
  
produitAcc([v|T1],[H2|T2],[],R):-  
    produitAcc(T1,T2,H2,R).  
  
produitAcc([v|T1],[H2|T2],Acc,R):-  
    xorliste(H2,Acc,S),  
    produitAcc(T1,T2,S,R).  
  
produitAcc([f|T1],[_|T2],Acc,R):-  
    produitAcc(T1,T2,Acc,R).
```

pour commencer utilisons maintenant le vecteur M de la sorte: [H|T] et la tête sera égal à la sous-liste courante. C'est cela qui va remplacer la clause **N-IÈME()** et du coup retirer la variable COMPTE. Pour pousser encore plus loin l'optimisation nous avons retiré les évaluations de variables dans les clauses pour les mettre directement dans l'appel (ex: [v|T1]). Grâce à cette nouvelle clause nous gagnons beaucoup de temps et tout sera abordé dans la prochaine partie !

IV. Comparaison des temps entre non optimisé et optimisé

Voici la comparaison des temps entre notre programme initial (0 optimisation) et notre première optimisation sur la clause **XOR**. On remarque une diminution importante surtout au niveau difficile.

	No optimisation	1st optimisation
	Average (ms)	Average (ms)
Easy (10V)	3062	2484
Medium (7V)	11902	9521
Hard (6V)	73071	57163

Après la 2eme optimisation nous avons obtenus des résultats encore meilleurs:

	No optimisation	1st optimisation	1st + 2nd optimisation
	Average (ms)	Average (ms)	Average (ms)
Easy (10V)	3062	2484	1515
Medium (7V)	11902	9521	5700
Hard (6V)	73071	57163	34119

Encore une fois l'écart le plus important est au niveau difficile (6 vrais) avec une différence d'environ **40 secondes** entre notre programme initial et notre programme final ce qui est énorme!

V. Conclusion et résultat avec la matrice Hardcore

En conclusion nous avons adorés faire ce projet sur Prolog et avons appris beaucoup de choses sur comment rendre nos programmes plus optimisés et comment fonctionne Prolog d'une manière générale.

Concernant les autres matrices données (*difficile.txt* et *hardcore.txt*) malgré avoir laissé plus d'une heure pour certains calculs nous avons seulement obtenus des temps avec la difficile (119330 ms pour 20 vrais et 266285 ms pour 18 vrais). Peut être que si on avait laissé beaucoup plus longtemps on aurait fini par obtenir un résultat mais ce n'était pas notre but.

Nous avons regroupés **tous nos résultats** dans un tableau excel pour ne pas surcharger le rapport, voici un lien pour les voir si vous êtes intéressé:

[TOUS LES RESULTATS!](#)