



M1Info/BIA/TP1/2011
TP3/4 PROLOG
(PROgrammation LOGique)
Compte-Rendu à rendre avant le TP5
(Avant le lundi 21/11/2011)

Avertissement : Ce TP est noté. Vous devez rendre un CR correspondant au plan proposé avant la séance du TP 5/6.

Objectif du TP : 1) Illustrer le cours sur la résolution de problème (se munir du cours est utile) 2) Pratiquer Prolog pour réaliser un « moteur » de résolution de problèmes.

Organisation du travail : Les questions 1 et 2 doivent être traitées lors de la première séance. Préparez les questions 3 et 4 à la maison. Tout doit être terminé à la fin de la seconde séance.

1 Etudiez le code suivant (trouvé sur le Web, marche selon son auteur...) [4 points]

/* Ecrit par Nicolas Zinovieff (p6mip138) que nous remercions de nous autoriser à utiliser son code pour cet exercice*/

/* representation:

1 est une rive, 0 l'autre, et on parle des différents protagonistes dans cet ordre: moi, chou, chevre, loup, chacun pouvant être sur une rive (0) ou l'autre(1)

*/

/* etats impossibles */

impossible(etatcourant(PasCa,Ca,Ca,PasCa)) :- PasCa \= Ca.

impossible(etatcourant(PasCa,PasCa,Ca,Ca)) :- PasCa \= Ca.

/* transbahuter le chou */

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(0,0,Chevre,Loup)) :- Chou = 1, Moi = 1, not(impossible(etatcourant(0,0,Chevre,Loup))).

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(1,1,Chevre,Loup)) :- Chou = 0, Moi = 0, not(impossible(etatcourant(1,1,Chevre,Loup))).

/* transbahuter la chevre */

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(0,Chou,0,Loup)) :- Chevre = 1, Moi = 1, not(impossible(etatcourant(0,Chou,0,Loup))).

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(1,Chou,1,Loup)) :- Chevre = 0, Moi = 0, not(impossible(etatcourant(1,Chou,1,Loup))).

/* transbahuter le loup */

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(0,Chou,Chevre,0)) :- Loup = 1, Moi = 1, not(impossible(etatcourant(0,Chou,Chevre,0))).

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(1,Chou,Chevre,1)) :- Loup = 0, Moi = 0, not(impossible(etatcourant(1,Chou,Chevre,1))).

/* transbahuter le moi tout seul */

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(0,Chou,Chevre,Loup)) :- Moi = 1, not(impossible(etatcourant(0,Chou,Chevre,Loup))).

etatsuiant(etatcourant(Moi,Chou,Chevre,Loup),etatcourant(1,Chou,Chevre,Loup)) :- Moi = 0, not(impossible(etatcourant(1,Chou,Chevre,Loup))).

/* recherche avec accumulateur */

possible_riviere(X, X, CList, CList).

possible_riviere(X, Y, CList, List) :-

etatsuiant(X, Z),

not(member(Z, CList)),

```

possible_riviere(Z, Y, [Z | CList], List).

chemin_riviere(X, Y, FoundPath) :- possible_riviere(X, Y, [X], FoundPath).

/* recherche en largeur */
recherche_largeur(X,Y,C) :- cherche(Y,[[X]],C).

cherche(Y,[[Y|Chs]|_],[Y|Chs]) :- !. /* la solution est la bonne mais inversee */
cherche(Y,[[S|Chs]|AutresChemins],Sol) :- findall([Suc,S|Chs],etatsuiivant(S,Suc),NewChemins),

    append(AutresChemins,NewChemins,Chemins),

    cherche(Y,Chemins,Sol).

but_riviere :- chemin_riviere(etatcourant(0,0,0,0),etatcourant(1,1,1,1), Liste),reverse(Liste,R),write(R).

```

1.1 Travail à rendre : Décrire l'état de départ, l'état but, les opérateurs possibles (avec leurs contraintes de mise en œuvre), et faire le schéma de l'arbre d'exploration des états sur 2 niveau (état de départ et les états accessibles par les opérateurs valides) [2 points]

1.2 Réécrire le programme pour le rendre plus lisible. Commenter son exécution. Fournir le code et son exécution commentés [2 points]

2 En vous inspirant du code précédent (celui que vous aurez amélioré !), écrivez un programme Prolog qui résout le problème suivant avec un exemple d'exécution [6 points]:

- j'ai deux cruches : une cruche de 8 litres et une cruche de 5 litres, je souhaite avoir exactement 4 litres. La fontaine coule indéfiniment.
- Commencer par poser le problème sous la même forme que ci-dessus (état initial, état but, opérateurs et leurs conditions de mise en œuvre).

2.1 Modélisation : Donner état initial, but, opérateurs et premier niveau de l'arbre d'exploration [1 point]

2.2 Réalisation : Donnez le code commenté avec un exemple d'exécution (test à présenter en séance) [5 points]

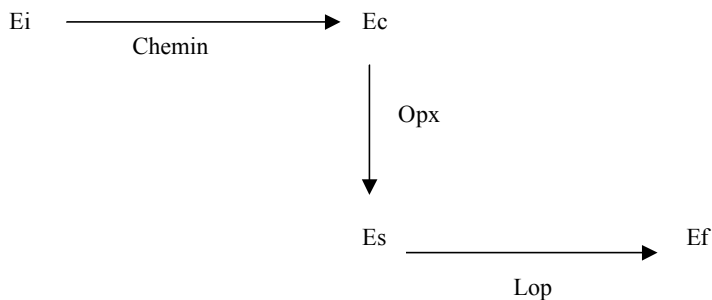
3 Comparer l'approche codée avec l'approche suivante

On considère des problèmes du type recherche d'un chemin entre un état initial E_i et un état final E_f , avec des opérateurs de transition pour passer d'un état à un autre. Par développement des nœuds au fil de la recherche, un graphe orienté est explicité : les sommets sont associés aux états du problème et les arêtes sont étiquetées par les opérateurs. Le graphe (arbre) obtenu s'appelle graphe (arbre) de recherche ou graphe OU.

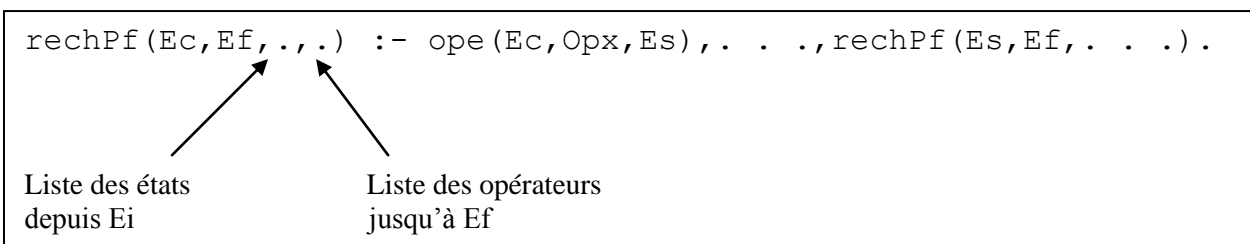
On se propose d'écrire un programme de recherche général -en profondeur d'abord-, que l'on appliquera par la suite à un problème concret.

Pour un problème, l'état initial est connu par le fait `initial(Ei)`. L'état final est connu par le fait `final(Ef)`. Les opérateurs sont connus par le prédicat `ope(Ec, Opx, Es)`, `Ec` désignant l'état courant, `Opx` un opérateur de transition et `Es` l'état de sortie, atteint par application de `Opx`.

- On atteint l'état `Ec` à partir de `Ei` en passant par une **liste d'états** mémorisés dans la liste `Chemin`.
- L'application de l'opérateur `Opx` à l'état courant `Ec` fait passer à un état `Es`.
- `Lop` est la **liste des opérateurs** qu'il faut appliquer depuis `Es` pour atteindre `Ef`.



Définir le prédicat de recherche en profondeur `rechPf` :



Une solution est ci-dessous.

`%Recherche en profondeur simple et "brutale". Sans oublier de ne pas passer 2 fois par le même État.`

```
rechPf(Ef, Ef, Letats, []) :- !, print(Letats).
```

```
rechPf(Ec, Ef, Letats, [Opx|Lop]) :- opF(Ec, Opx, Esuivant),
```

```
not(member(Esuivant, Letats)),
```

```
rechPf(Esuivant, Ef, [Esuivant|Letats], Lop).
```

Définir le prédicat `resoudre(S)` qui donne la liste `S` des opérateurs à appliquer pour passer de l'état initial à l'état final.

`%Resolution générale`

```
resoudre(S) :- initial(Ei), final(Ef), rechPf(Ei, Ef, [Ei], S).
```

On va maintenant appliquer le prédicat `rechPf` au problème suivant, en définissant pour chacun d'entre eux l'état initial, l'état final et les opérateurs.

Le problème des flèches

3.1 Définition du problème

Six flèches sont dans la position de la figure 1. On souhaite les positionner comme dans la figure 2.

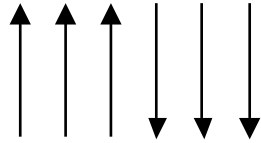


Figure 1

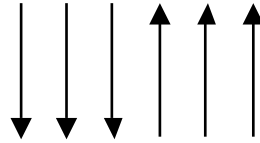


Figure 2

On symbolise ces deux états par « hhhbbb » et « bbbhhh » avec “h” pour “flèche haute” et “b” pour “flèche basse”.

On définit quatre opérateurs de transition :

- R1 : retournement de deux flèches hautes adjacentes (« hh » devient « bb »)
- R2 : retournement d’une flèche haute et d’une flèche basse adjacentes (« hb » devient « bh »)
- R3 : retournement d’une flèche basse et d’une flèche haute adjacentes (« bh » devient « hb »)
- R4 : retournement de deux flèches basses adjacentes (« bb » devient « hh »)

3.2 Modélisation du problème

- On définit un prédicat `remp1(S1, S2, L1, L2)` qui est satisfait si le remplacement de la sous-liste S1 par S2 dans la liste L1 donne L2

%Remplacement d'une sous-liste par une autre dans une liste quelconque.

```
remp(S1, S2, L1, L2) :- append(Tmp1, LSuffixe, L1), append(LPrefixe, S1, Tmp1),  
                        append(LPrefixe, S2, Tmp2), append(Tmp2,  
LSuffixe, L2).
```

- On définit les prédicats `initial`, `final` et `ope` pour le problème des flèches et utiliser le prédicat `resoudre` pour trouver des solutions au problème.

%Probleme des fleches

```
initial( [h,h,h,b,b,b] ).  
final( [b,b,b,h,h,h] ).
```

```
opF( L1, r1, L2) :- remp([h,h], [b,b], L1, L2).  
opF( L1, r2, L2) :- remp([h,b], [b,h], L1, L2).  
opF( L1, r3, L2) :- remp([b,h], [h,b], L1, L2).  
opF( L1, r4, L2) :- remp([b,b], [h,h], L1, L2).
```

- 3.3** Rédiger une explication montrant en quoi cette approche est similaire avec les codes étudiés et réalisés aux questions 1) et 2) et en quoi elle diffère. Quelle est la méthode la plus générale ? Pourquoi ? [1 point]
- 3.4** Quelle heuristique proposez-vous pour éviter d'explorer toutes les possibilités [2 points] Illustrer sur le schéma de l'arbre d'états en montrant les chemins qui seront ainsi écartés au profit de chemins plus prometteurs.
- 3.5** Proposer une modification du programme des flèches qui permet de tenir compte de l'heuristique. Rendre le code commenté, le présenter en séance [2 points]
- 4** Proposer un code prolog sur ce modèle pour l'un des problèmes « chou, chèvre, loup » ou « les cruches ».
- 4.1** Codage sans heuristique. Rendre le code commenté et le faire tester en séance [5 points]
- 4.2** Proposez une heuristique et le code correspondant. Rendre le code et le faire tester en séance [bonus 5 points]