INP-ENSEEIHT

RAPPORT
BASES DE CONNAISSANCES
RENDU DES PROGRAMMES RÉALISÉS EN TP ET DU BUREAU D'ÉTUDES

Selove OKE CODJO
Solène EHOLIE

6 juin 2014

Table des matières

1 Introduction

La planification est un outil très important, particulièrement dans l'intelligence artificielle car il permet d'automatiser des recherches fastidieuses. Le projet de bases de connaissances nous a permis de mettre en pratique la théorie de ce domaine à travers la génération de lan en utilisant des stratégies collaboratives ou non ainsi que des logiques modales. Prolog est le language qui a servi de support à la réalisation de ce projet.

Nous vous présenterons dans ce rapport les résultats de nos travaux.

2 Rendu des programmes réalisés en TP

2.1 TP1 : generation de plan

Le but du TP1 était de prendre en main la programmation en prolog à travers un problème simple mettant en scène un agent et un objet. L'agent est capable de se déplacer d'un point a à un point b, de prendre l'objet et de le poser. Il va donc falloir générer le plan à suivre par l'agent pour aller d'une certaine situation à une autre

2.1.1 Résolution du problème

La première chose à faire était de définir les différentes actions dont notre agent est capable. Une action étant définie par une specification, une liste de conditions nécessaire à son accomplissement, une liste de propriétés qui ne seront plus vraies et une liste de nouvelles propriétés.

Ainsi, on définit l'action aller demandant à un robot de se deplacer d'un point X à un point Y. il faut donc que le robot soit en X, propriété qui deviendra fausse après le déplacement où il se retrouve en Y. La définition est donnée ci-dessous.

```
action(aller(robot,X,Y),
        [lieu(robot) = X],
        [lieu(robot) = X], [lieu(robot) = Y]) :-
            member(X,[a,b]), member(Y,[a,b]), X \= Y.
```

Ensuite on définit prendre, il faut que le robot et la boite soient au même endroit et que la main du robot soit libre, bien sûr la main n'est plus libre aprè l'action et l'objet n'est plus à l'endroit où il était mais dans la main du robot.

```
action(prendre(robot,O),
        [lieu(robot) = L, lieu(O) = L, libre(main(robot))],
        [libre(main(robot)), lieu(O)=L],
        [lieu(O)=main(robot)]) :-
            member(L,[a,b]), member(O,[boite]).
```

Enfin, on définit poser, l'objet doit être dans la main du robot avant de la quitter et se retrouver à la même position que le robot.

```
action(poser(robot,O),
        [lieu(robot) = L, lieu(O) = main(robot)],
        [lieu(O)=main(robot)],
        [lieu(O) = L, libre(main(robot))]) :-
            member(L, [a,b]), member(O, [boite]).
```

Il faut maintenant decrire ce que c'est qu'une transition entre un état E et un autre F, cele consiste juste en la réalisation d'une action dans E, il faut donc que les conditions soient vérifiées en E, il faut supprimer les propriétés qui ne seront plus vérifiées et ajouter les nouvelles. On a donc :

```
transition(A,E,F) :- action(A, C, S, AJ), verifcond(C,E),
                     suppress(S, E, EI), ajouter(AJ,EI,F).
```

La fonction verifcond(C,E) vérifiée l'inclusion de C dans E, suppress(S, E, EI) supprime S de E pour donner EI et ajouter(AJ,EI,F) ajoute AJ à EI pour obtenir F.

```
%verifcond(C,E) est l inclusion de C dans E
verifcond([], _). %la liste vide est toujours incluse
verifcond([C|LC], L) :- member(C,L), verifcond(LC,L).
```

```
%suppression
suppress([],L,L).
suppress([X|Y], Z, T) :- delete(Z,X,U), suppress(Y,U,T).
```

```
%ajout
ajouter(AJ,E,F) :- union(AJ,E,F).
```

Pour finir, il nous faut generer un plan d'une certaine profondeur, et une autre qui nous permet d'entrer la profondeur que l'on veut sans avoir à modifier le code ces deux fonctions sont données ci-dessous :

```
%generation de plan
genere(E,F,[A],1):-
    transition(A,E,F).

genere(EI,EF,[ACT|PLAN],M):-
   M > 1, transition(ACT,EI,E), N is M-1,
    between(1,N,P), genere(E,EF,PLAN,P).
```

```
planifier(Plan) :-
    init(E),
    but(B),
    nl,
    write(' Profondeur limite : '),
    read(Prof),
    nl,
    genere(E,F,Plan,Prof),
    verifcond(B,F).
```

2.1.2 Test

FIGURE 1 – etat initial

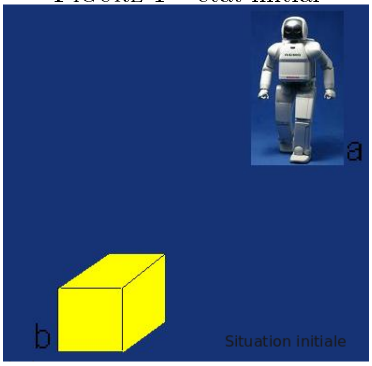
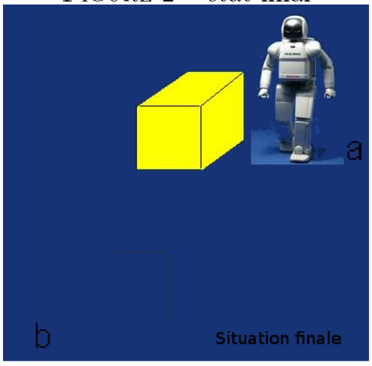


FIGURE 2 – etat final



Les résultats du test proposé par le sujet et illustré par les images ci-dessus sont donné ci-dessous.

```
?- planifier(P5, Temps).

Profondeur limite : 5.

P5 = [aller(robot, a, b), prendre(robot, boite), aller(robot, b, a),
poser(robot, boite), aller(robot, a, b)] .

Temps = 0.03 s
```

On précise que pour n<5, aucun plan n'est trouvé et pour n>=5, on trouve P5.

2.2 TP2 : planification multi-agents

L'objectif ici était de faire de la planification comme au TP1 à la seule différence qu'il y a ici plusieurs agents différents. Ce qui offre un nombre de situations plus élevé et des cas plus complexes à traiter.

2.2.1 Résolution du problème

La différence principale avec le TP1 réside dans la définition des actions. On a plusieurs agents qui ont des capacités différentes, il faudra donc spécifier les agents capables d'effectuer telle ou telle autre action. De plus, concernant les lieux, un objet peut se retrouver dans la 'main' de tel ou tel autre agent, il faut donc un moyen de distinguer les 'mains'. Les objets peuvent aussi être empilés les uns sur les autres. La définition des action est donc un peu plus complexe comme nous allons le voir dans ce qui suit.

La première action définie est $aller\_a\_vide(R, Ld, La)$. C'est la possibilité pour un agent de se déplacer de la position Ld vers la position La sans transporter d'objet, ce qui constitue la seule différence avec l'action aller du TP1. Seul le robot1 sait le faire.

```
action( aller_a_vide(R,Ld,La),
        [position(R) = Ld, libre(main(R))],
        [position(R) = Ld],
        [position(R) = La] ) :-
        member(Ld,[a,b,c]), member(La,[a,b,c]), member(R,[robo1]), Ld \= La.
```

Ensuite il fallait définir $transporter(R, Ld, La, O)$ qui permet à un agent R d'aller de la position Ld à la position La en transportant l'objet O. Il faut donc que l'agent et l'objet soient en Ld et qu'il tienne l'objet O. Après, ils se retrouvent tout les deux en La. Le code est donné dans l'encadré ci-dessous.

```
action( transporter(R,Ld,La,O),
        [position(R) = Ld, position(O) = Ld, lieu(O) = main(R)],
        [position(R) = Ld, position(O) = Ld],
        [position(R) = La, position(O) = La] ) :-
        member(Ld,[a,b,c]), member(La,[a,b,c]), member(R,[robo1]),
        member(O,[cube1,cube2]), Ld \= La.
```

L'action $attraper(R, O, L)$ décrit le fait, pour un agent R, de prendre un objetO sur la table de la position L. Il faut que l'objet soit accessible, c'est à dire qu'il n'y ait pas d'objet au dessus, l'objet n'est donc plus sur la table, n'est plus accessible et la main du robot n'est plus libre. L'objet se trouve maintenant dans la main du robot R. Elle peut être exécutée par les trois robots.

```
action( attraper(R,O,L),
        [position(R) = L, position(O) = L, sur(O,table(L)),
        accessible(O), libre(main(R))],
        [sur(O,table(L)), accessible(O), libre(main(R))],
        [lieu(O) = main(R)] ) :-
        member(L,[a,b,c]), member(R,[robo1, robo2, robo3]), member(O,[cube1,cube2]).
```

L'action $saisir(R, O, L)$ permet à un robot R de prendre un objet O qui se trouve en haut d'une pile et au même lieu L que le robot. La main du robot doit être libre et l'objet doit se trouver sur un autre Osous. Bien sû, après l'action, O ne se trouve plus sur Osous, il n'est plus accessible et la main du robot n'est plus libre. O est maintenant dans la main de R, et l'objet qui était en dessous est désormais accessible. Seuls les robots 2 et 3 peuvent le faire.

```
action( saisir(R,O,L),
        [position(R) = L, position(O) = L, sur(O,Osous),
        accessible(O), libre(main(R))],
        [sur(O,Osous), accessible(O), libre(main(R))],
        [lieu(O) = main(R), accessible(Osous)] ) :-
        member(L,[a,b,c]), member(R,[robo2, robo3]),
        member(O,[cube1,cube2]), member(Osous,[cube1,cube2]), O \= Osous.
```

Les actions $deposer(R, O, L)$ et $empiler(R, Osur, Osous, L)$ sont les actions contraires respectivement à $attraper(R, O, L)$ et $saisir(R, O, L)$, on ne détaillera pas leur fonctionnement. Il suffit de dérouler les scénarios dans le sens opposé à ceux d'attraper et saisir.

```
action( deposer(R,O,L),
        [position(R) = L, position(O) = L, lieu(O) = main(R)],
        [lieu(O) = main(R)],
        [sur(O,table(L)), accessible(O), libre(main(R))] ) :-
        member(L,[a,b,c]), member(R,[robo1, robo2, robo3]),member(O,[cube1,cube2]).
```

```
action( empiler(R,Osur,Osous,L),
        [position(R) = L, position(Osur) = L, lieu(Osur) = main(R), accessible(Osous)],
        [lieu(Osur) = main(R), accessible(Osous)],
```

```
        [sur(Osur,Osous), accessible(Osur), libre(main(R))] ) :-
        member(L,[a,b,c]), member(R,[robo2, robo3]),
        member(Osur,[cube1,cube2]), member(Osous,[cube1,cube2]), Osur \= Osous.
```

2.2.2 Tests

Un premier test demandait à un robot d'inverser deux objets emplilés sur une table. Le code est donné ci-dessous :

```
%situation initiale
initInv(EI) :- EI = [position(robo2) = b,
               position(cube1) = b, position(cube2) = b,
               accessible(cube1),
               libre(main(robo2)),
               sur(cube1,cube2), sur(cube2,table(b))].
%situation finale
butInv(BUT) :- BUT = [position(robo2) = b,
               position(cube1) = b, position(cube2) = b,
               accessible(cube2),
               libre(main(robo2)),
               sur(cube2,cube1), sur(cube1,table(b))].
```

Le résultat de ce premier test.

Un autre test simple consistait à faire collaborer des robots aux capacités différentes. Le cube 1 est empilé sur le cube 2 en b et il faut les poser en c. Le code est donné ci-dessous.

```
init2Dep(EI) :- EI = [position(robo1) = b, position(robo2) = b,
               position(cube1) = b, position(cube2) = b,
               accessible(cube1),
               libre(main(robo2)), libre(main(robo1)),
               sur(cube1,cube2), sur(cube2,table(b))].
but2Dep(BUT) :- BUT = [position(robo2) = b, position(robo1) = c,
               position(cube1) = c, position(cube2) = c,
               accessible(cube2), accessible(cube1),
               libre(main(robo2)), libre(main(robo1)),
               sur(cube1,table(c)), sur(cube2,table(c))].
```

Le resultat du deuxième test.

```
?- planifier2Dep(P,T).

Profondeur limite : 9.

P = [saisir(robo2, cube1, b), attraper(robo1, cube2, b),
transporter(robo1, b, c, cube2), deposer(robo2, cube1, b),
deposer(robo1, cube2, c), aller_a_vide(robo1, c, b),
attraper(robo1, cube1, b), transporter(robo1, b, c, cube1),
deposer(..., ..., ...)],
T = 149.2
```

Les images ci-dessous illustrent le test du sujet.

Les résultats du test proposé par le sujet et illustré par les images ci-dessus sont donné ci-dessous.

```
?- planifier(P,T).

Profondeur limite : 11.

P = [saisir(robo2, cube1, b), attraper(robo1, cube2, b),
transporter(robo1, b, c, cube2), deposer(robo2, cube1, b),
deposer(robo1, cube2, c), aller_a_vide(robo1, c, b),
attraper(robo1, cube1, b), transporter(robo1, b, c, cube1),
deposer(..., ..., ...)|...],
T = 1055.22,
```

FIGURE 3 – etat initial

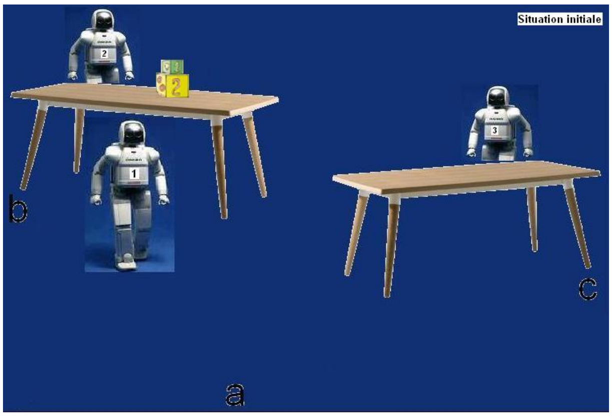


FIGURE 4 – etat final

2.3 TP3 : générateur-démonstrateur en logique modale

Au cours de cette séance, il fallait écrire un programme capable de démontrer des théorèmes sur une logique modale.

2.3.1 Résolution du problème

Dans un premier temps il fallait définir le fait qu'une proposition élémentaire P soit satisfaite dans un monde W. Déjà, il faut que P soit bien dans la liste des propositions et W dans la liste des mondes, il suffit que P soit une proposition élémentaire du monde en question, la liste étant donnée . Cela se définit comme suit :

```
satisfait(W, P) :-
    proposition(P),
    monde(W),
   m(W,L),
    member(P,L).
```

Mais on ne va pas se limiter à des formules élémentaire qui ne contiennent que les propositions listées. On veut pouvoir utiliser des opérateurs. Pour cela il faut d'abord les définir avant de définir l'opération correspondante.

la négation (non) : c'est un opérateur de la forme fy auquel on donnera une priorité de 200. Pour que 'non P' soit satisfait dans W, il faut que W ne fasse pas parti des mondes dans lesquels P est satisfait. Il faut donc déterminer la liste des mondes dans lesquels P est satisfait.

```
%operateur
:- op(200, fy, non).
satisfait(W,non P) :-
   monde(W),
   list_w_satisfait(P, LW),
   not(member(W, LW)).
%liste des mondes qui satisfont P
list_w_satisfait(P, LW) :-
   findall(W, satisfait(W, P), LW).
```

la conjonction (et) : c'est un opérateur de la forme xfy auquel on donnera une priorité de 400. Pour que 'P et Q' soit satisfait dans W, il faut que P soit satisfait dans W et que Q soit satisfait dans W.

```
%operateur
:- op(400, xfy, et).
satisfait(W, P et Q) :-
   monde(W),
   satisfait(W,P),
   satisfait(W,Q).
```

la disjonction (ou) : c'est un opérateur de la forme xfy auquel on donnera une priorité de 400. On utilise ici l'écriture du 'ou' en fonction du 'et et du 'non', en effet 'P ou Q = non(non P et non Q).

```
%operateur
:- op(400, xfy, ou).
```

```
satisfait(W,P ou Q) :-
  satisfait(W, non(non P et non Q)).
```

l'implication (=>) : c'est un opérateur de la forme xfy auquel on donnera une priorité de 300. L'implication se défini par 'non P ou Q'.

```
%operateur
:- op(300, xfy, =>).
satisfait(W, P => Q) :-
  satisfait(W, non (P et non Q)).
```

l'équivalence (<=>) : c'est un opérateur de la forme xfy auquel on donnera une priorité de 300. L'équivalence n'est rien d'autre qu'une implication dans les deux sens.

```
%operateur
:- op(300, xfy, <=>).
satisfait(W, P <=> Q) :-
  satisfait(W, P => Q),
  satisfait(W, Q => P).
```

l'implication stricte (–») : c'est un opérateur de la forme xfy auquel on donnera une priorité de 300. On a une équivalence entre 'P » Q' et '#(P=>Q)', on aura la définition de '#' un peu plus loin.

```
%operateur
:- op(300, xfy, -->>).
satisfait(W, P -->> Q) :-
  satisfait(W, #(P => Q)).
```

la possibilité (<>) : c'est un opérateur de la forme fy auquel on donnera une priorité de 200. On dit que P est possible dans W s'il existe un monde W2 en relation avec W tel que W2 satisfait P. Il faut donc trouver tous les mondes en relation avec W et voir si il y en a au moins un qui satisfait P

```
%operateur
:- op(200, fy, <>).
satisfait(W, <> P) :-
  monde(W),
  un_rel_satisfait(W,P).
%une des relations de W satisfait P
un_rel_satisfait(W, P) :-
  bagof(M, rel(W,M), LM),
  un_satisfait(LM, P).
%un des lments de la liste satisfait p
un_satisfait([M | LM], P) :-
  (satisfait(M,P) -> true; un_satisfait(LM, P)).
```

la nécessarité (#) : c'est un opérateur de la forme fy auquel on donnera une priorité de 200. Il se définit aisément à partir de la possibilité. En effet '#P = non(<>(non P))'.

```
%operateur
:- op(200, fy, #).
satisfait(W, # P) :-
  satisfait(W,non(<>(non P))).
```

On rappelle que le but est d'écrire un programme capable de nous lister toutes les formules d'une certaine complexité qui sont des théoremes ou de nous dire si une formule donnée est un théorème.

Pour cela, on veut déjà être capable de générer toutes les formules d'une certaine complexité. Une formule de complexité 0 n'est qu'un proposition et les formules de complexité n sont définies récursivement comme une proposition suivie d'un opérateur binaire et d'une formule de complexité n-1 ou juste comme un opérateur unaire suivi d'une formule de complexité n-1. Le code est donné ci-dessous.

```
%generation de formules
 genere(P,0):-
   proposition(P).

 genere(F,M):-
   M > 0,
```

```
    N is M-1,
    genere(FBIS,N),
    operateur(O),
    proposition(X),
    (O=(=>) -> F=..[O,X,FBIS],F = X=>FBIS;
      O=(<=>) -> F=..[O,X,FBIS],F = X<=>FBIS;
        O=(et) -> F=..[O,X,FBIS], F = X et FBIS;
          O=(ou) -> F=..[O,X,FBIS],F = X ou FBIS;
            O=(-->>) -> F=..[O,X,FBIS],F = X-->>FBIS;
              O=(non) -> F=..[O,X,FBIS],F = non FBIS;
                O=(<>) -> F=..[O,X,FBIS],F = <> FBIS;
                  O=(#) -> F=..[O,X,FBIS],F = # FBIS).
```

Enfin, il faut écrire 'theoreme(P)', si P est une variable, on demande la complexité des formules à générer et on les gérère avant de renvoyer celles qui sont des théorèmes, sinon on vérifie juste si P est un théoreme. P est un théorème si il est vrai dans tout les mondes existant, pour arriver à cela, on récupère la liste des mondes, celle des mondes dans lesquels P est vrai et on vérifie qu'ils sont égaux.

```
%THEOREME : p doit  tre  satisfait dans tous les mondes
theoreme(P) :-
      (nonvar(P) ->theoreme_aux(P);
      nl,
      write(' complexite : '),
      read(C),
      nl,
      genere(P, C),
      theoreme_aux(P)).

theoreme_aux(P) :-
      list_w_satisfait(P, LW),
      findall(W,monde(W), ALLW),
      egal(LW, ALLW)
%l'egalite est une double inclusion
egal(LW, ALLW) :-
      inclusion(LW, ALLW),
      inclusion(ALLW, LW).

%inclusion(C,E) est l inclusion de C dans E
inclusion([], _). %la liste vide est toujours incluse
inclusion([W|LW], L) :- member(W,L), inclusion(LW,L).
```

2.4 TP4 : générateur de plan en logique modale

Concernant le TP4, il fallait implanter un générateur de plan en logique modale. Le problème à résoudre étant :

```
?- est(w0,[p1, ... , pk]), satisfait(<<< Plan >>> (F1 ^ ... ^ Fn), w0).
```

Nous supposons l'ensemble des mondes connus. Il nous faut trois nouveaux operateurs '«', '»' et ' : :'. Ils sont défini ci-après.

```
%<< a :: b :: c >> (P et Q)
:- op(100, fy, <<).
:- op(150, xfy, <<).
:- op(100, xfy, ::).
```

Il nous faut donc définir ce que c'est que de satisfaire '«A»F' dans le monde W. Ici, soit A est une suite d'actions, soit A est une seule action. Soit X la première action de A (si A est une suite) ou A lui-même (si A est une action). Il faut que les conditions de X soient vérifiées dans le monde de départ W $est\_entrepeunable(A,W)$, il faut ensuite trouver un monde W2 dans lequel les effets de A sont effectifs $est\_effective(A,W)$. Enfin, si A est une action, il faut que F soit satisfait dans W2, sinon si A est une suite d'actions 'X : :ABIS', il faut que '«ABIS»F soit satisfait dans W2. Le code est donné ci-dessous.

```
satisfait(W, << A >> F) :-
      monde(W),
      rel(W, W2), % \E un monde W2
```

```
        operateur_act(O),
        (O=(::) -> A=..[O,X,ABIS], %si on a une suite d'actions
                    A = X::ABIS,
                    est_entrepenable(X, W),
                    est_effective(X, W2),
                    satisfait(W2, <<ABIS>> F);
         est_entrepenable(A, W), %si A est une action
         est_effective(A, W2),
         satisfait(W2, F)). % F est satisfait dans W2
```

Il faut donc définir est_entrepenable et est_effective. $est\_entrepeunable(A,W)$ permet de vérifier que les conditions de A sont satisfaites dans W, pour cela, il faut récupérer la liste des conditions, la convertir en conjonction de ses différents éléments avant de voir si cette conjonction est satisfaite dans W. Pour $est\_effective(A,W)$, c'est le même principe mais on prend la négation de la conjonction des effets.

```
% les conditions de A sont satisfaites dans W
est_entrepeunable(A, W) :-
        monde(W),
        action(A, Cond, Suppr, Ajout),
        list2conj(Cond, P),
        satisfait(W, P).

% les effets de A sont effectifs dans W
est_effective(A, W) :-
        monde(W),
        action(A, Cond, Suppr, Ajout),
        list2conj(Suppr, SU),
        list2conj(Ajout, AJ),
        satisfait(W, non (SU)),
        satisfait(W, AJ).

%transforme une liste en conjonction des   lments   de la liste
list2conj([A], A).
list2conj([A|LA], A et LC) :-
        list2conj(LA, LC)
```

3 Travaux réalisés lors du bureau d'étude

Durant le bureau d'étude, nous avons été amenés à analyser différentes logiques modales et les comparer à des logiques modales vues en cours.

3.1 Etude de Lm{p}

3.1.1 Definition de 'nécessaire'

La définition de 'necessaire' se fait à partir de celle de 'possible'.

```
 [p] = ~<p>~ donc
      |= [p]X <=> |= ~ (<p>(~X))
              <=> |= ~ (p ou ~X) <=> ~(X => p)
              <=> |= ~p et X
```

3.1.2 Idempotence de 'necessaire' et 'possible'

Cela revient à montrer que

```
<p>^n  =  <p><p>...<p>  = <p> et [p]^n  =  [p][p]...[p]  =  [p]
```

On fait une démonstration par récurrence

```
        Pour n=1, c'est␣trivial.
␣␣␣␣␣␣␣␣Supposons␣la␣propriete␣vraie␣au␣rang␣n=k
␣␣␣␣␣␣␣␣\A␣X␣\in␣F,␣on␣a
␣␣␣␣␣␣␣␣|=␣<p>^k+1␣X␣<=>␣|=␣<p>(<p>^k␣X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣<p>(<p>X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣p␣ou␣(<p>X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣p␣ou␣(p␣ou␣X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣p␣ou␣X
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣<p>X

␣␣␣␣␣␣␣␣de␣meme␣pour␣[p]
␣␣␣␣␣␣␣␣|=␣[p]^k+1␣X␣<=>␣|=␣[p]([p]^k␣X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣[p]([p]X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣~p␣et␣(<p>X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣~p␣et␣(~p␣et␣X)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣~p␣et␣X
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<=>␣|=␣[p]X
␣␣␣
```

On a bien l'idempotence.

Montrons aussi que toute séquence de modalités $Modi \in <p>, [p]$ préfixant une formule X quelconque, telle que $Mod1Mod2...ModnX$, est une formule équivalente à $Mod1X$. Autrement dit : $Mod1Mod2...Modn = Mod1$ pour toute modalité $Modi \in <p>, [p]$.

Cela revient a dire que la premiere modalite "absorbe les autres", il faut donc montrer que <p>[p] = <p> et [p]<p> = [p]

```
        \forall X \in F, on a
        |= <p>[p]X <=> |= <p>(~p et X)
                   <=> |= p ou (~p et X)
                   <=> |= (p ou ~p) et (p ou X)
                   <=> |= Vrai et (p ou X)
                   <=> |= p ou X
                   <=> |= <p>X
        et
        |= [p]<p>X <=> |= [p](p ou X)
                   <=> |= ~p et (p ou X)
                   <=> |= (~p et p) ou (~p et X)
                   <=> |= Faux ou (~p et X)
                   <=> |= ~p et X
                   <=> |= [p]X
```

Avec les propriétés $<p>^n = <p>$ et $[p]^n = [p]$, Cela montre bien que la première modalité "absorbe" les suivantes pour$Modi \in <p>, [p]$.

3.1.3 Comparaison à S5

```
   S5 = {epsilon, <>, []} Union {~, ~<>, ~[]}
```

On a :

```
   Sigma5 = {epsilon, <p>, [p]} Union {~, ~<p>, ~[p]},
```

En effet :

```
   <p><p> = <p>, [p][p] =[p], <p>[p]=<p>, [p]<p>=[p],
   ~<p> = [p]~ et finalement ~[p] = <p>~,
```

le reste des compositions étant trivial. Il y a donc une certaine ressemblance entre Sigma5 et S5.

Mais Sigma5 est-il équivalent à S5 ? Pour savoir cela, il suffit de voir si les théorèmes caractéristiques de S5 sont aussi vérifiés dans Sigma5.

Les théorèmes vérifiés par S5 sont :

```
        $\A P,Q \in F$
        (K): [](P => Q) => ([]P => []Q)
        (T): []P => P
        (4): []P => [][]P
```

```
(B): P => []<>P
ou
(E): <>P => []<>P
```

Essayons de les vérifier dans Sigma5

```
\A P,Q \in F, on a :
(K): [p](P => Q) => ([p]P => [p]Q)
 <=> [p](~P ou Q) => ((~p et P) => (~p et Q))
 <=> (~p et (~P ou Q)) => ((p ou ~P) ou (~p et Q))
 <=> (p ou (P et ~Q)) ou ((p ou ~P) ou (~p et Q))
 <=> p ou p ou ~P ou (P et ~Q) ou (~p et Q)
 <=> p ou (~p et Q) ou ~P ou (P et ~Q)
 <=> ((p ou ~p) et (p ou Q)) ou ((~P ou P) et (~P et Q))
 <=> Vrai
```

K est vérifié dans Sigma5.

```
(T): [p]P => P
<=> (~p et P) => P
<=> ~(~p et P) ou P
<=> p ou ~P ou P
<=> Vrai
```

T est vérifié dans Sigma5

```
(4): [p]P => [p][p]P
<=> [p]P => [p]P
<=> Vrai
```

4 est vérifié dans Sigma5.

```
(B): P => [p]<p>P
<=> P => [p]P
<=> ~P ou (~p et P)
<=> (~P ou ~p) et (~P ou P)
<=> ~P ou ~p
```

Proposition fausse si p et P sont vrais donc B n'est pas vérifié dans Sigma5.

```
(E): <p>P => [p]<p>P
<=> <p>P => [p]P
<=> (p ou P) => (~p et P)
<=> ~(p ou P) ou (~p et P)
<=> (~p et ~P) ou (~p et P)
```

Proposition fausse si p est faux donc E n'est pas vérifié dans Sigma5.

Sigma5 vérifie donc KT4 mais pas B ni E.

3.1.4 Modalités

Comme on a montré a la section précédente, S5 et Sigma5 ont le même nombre de modalités, on a

```
S5 = {epsilon, <>, []} Union {~, ~<>, ~[]}
et
Sigma5 = {epsilon, <p>, [p]} Union {~, ~<p>, ~[p]}.
```

On a comme règles :

```
\A X \in F
|= epsilon X ssi |= X
|= <p>X ssi |= p ou X
|= [p]X ssi |= ~p et X
|= ~X ssi |= ~X
|= ~<p>X ssi |= [p]~X
         ssi |= ~p et ~X
|= ~[p]X ssi |= <p>~X
         ssi |= p ou ~X
```

3.2 Etude de Lm{E0}

On généralise cette approche en étudiant la logique Lm{E0} dédiée aux formules $< \phi > Xet[\phi]Xo\phi \in Eo = \{p,\ p\}$

3.2.1 Comparaison à S4

Les théorèmes vérifiés par S4 sont :

```
\A P,Q \in F
(K): [](P => Q) => ([]P => []Q)
(T): []P => P
(4): []P => [][]P
```

Essayons de les vérifier dans Sigma4 :

```
(K): \A P,Q \in F et \A phi \in E0 on a :
[phi](P => Q) => ([phi]P => [phi]Q) <=> Vrai,
```

la démonstration se passe comme avec Sigma5. K est vérifié dans Sigma4 De même T et 4 sont vérifiés dans Sigma4.

3.2.2 Modalités

On a :
S4 = espilon, [], <>, []<>, <>[], []<>[], <>[]<> Union , [], <>, []<>, <>[], []<>[], <>[]<>
Sigma4 a aussi 14 modalités distinctes :

```
Sigma4 = { espilon, <~p>, <p>, [~p], [p], <p>[~p], <~p>[p] }
         Union { ~, ~<~p>, ~<p>, ~[~p], ~[p], ~<p>[~p], ~<~p>[p] }
```

On a comme règles :

```
\A X \in F
|= epsilon X ssi |= X
|= <p>X ssi |= p ou X
|= <~p>X ssi |= ~p ou X
|= [p]X ssi |= ~p et X
|= [~p]X ssi |= p et X
|= ~X ssi |= ~X
|= ~<p>X ssi |= ~p et ~X
|= ~<~p>X ssi |= p et ~X
|= ~[p]X ssi |= p ou ~X
|= ~[~p]X ssi |= ~p ou ~X
|= <p>[~p]X ssi |= p ou (p et X)
|= ~<p>[~p]X ssi |= ~p et (~p ou ~X)
|= <~p>[p]X ssi |= ~p ou (~p et X)
|= ~<~p>[p]X ssi |= p et (p ou ~X)
```

On a de plus les propriétés suivantes :
Propriété 1 :

```
\A X,
(<p><~p>)X |= p ou ~p ou X |= Vrai ou X |= X
donc (<p><~p>) == (<~p><p>) == espilon
Plus generalement,
Pour toute modalite Mod,
\A n >= 0, (<p><~p>)Mod== (<~p><p>)Mod == Mod
```

Propriété 2 :

```
\A X
[p][~p]X |= ~p et (p et X) |= Faux et X |= Faux
Plus generalement,
Pour toute modalite Mod,
\A X
([p][~p])(Mod X) == ([~p][p])(Mod X) |= Faux
```

D'où, Propriété 2-a :

```
<p>[p][~p] X |=  p ou [p][~p]X |= p ou Faux |= p
```

Propriété 2-b :

```
~<p>[p][~p] X |=  ~p
avec ~<p>[p][~p] == <~p>[p][~p]
```

On vérifie que toute autre combinaison de 3 revient à une modalité précitée.

3.3 Etude d'autres logiques

3.3.1 Lm{E1}, E1 = {Faux}

Modalités :
Lm{E1} correspond à Sigma5 avec p = Faux d'où
Lm{E1} = {epsilon, <Faux>, [Faux]} Union { , <Faux>, [Faux]}.
avec <Faux> == [Faux] == espilon car
|= <Faux>X ssi |= Faux ou X |= X
|= [Faux]X ssi |= Vrai et X |= X
Ainsi

```
Lm{E1} = {epsilon, ~} soit 2 modalites distintes definies par
\A X \in F
|= epsilon X ssi |= X
|= ~X ssi |= ~X
```

Propriétés caractéristiques :
LmE1 vérifie donc KT4 tout comme Sigma5
De plus,
(B) : P => [p]<p>P
<=> P ou p
<=> P ou Vrai (dans E1 car p = Faux)
<=> Vrai
LmE1 vérifie donc B
(E) : <p>P => [p]<p>P
<=> (p et P) ou (p et P)
<=> (Vrai et P) ou (Vrai et P) (dans E1 car p = Faux)
<=> P ou P
<=> Vrai
LmE1 verifie donc E

```
Lm{E1} verifie donc KT4BE comme S5
```

3.3.2 Lm{E2}, E2 = {Vrai}

Modalités :
LmE2 correspond à Sigma5 avec p = Vrai d'où
LmE2 = epsilon, <Vrai>, [Vrai] Union , <Vrai>, [Vrai]
Ainsi

```
Lm{E2} = {epsilon, <Vrai>} Union {~, ~<Vrai>} soit 4
modalit s distinctes definies par
\A X \in F
|= epsilon X ssi |= X
|= ~X ssi |= ~X
|= <Vrai>X ssi |= Vrai ou X |= Vrai
|= ~<Vrai>X ssi |= Faux et ~X |= Faux
```

Propriétés caractéristiques :
LmE2 vérifie donc KT4 tout comme Sigma5. De plus,
(B) : P => [p]<p>P
<=> P ou p
<=> P ou Faux (dans E2 car p = Vrai)
<=> P

Pas nécessairement vrai donc LmE2 ne vérifie pas B
(E) : <p>P => [p]<p>P
<=> (p et P) ou (p et P)
<=> (Faux et P) ou (Faux et P) (dans E1 car p = Faux)
<=> Faux
LmE2 ne verifie donc pas E

```
Lm{E2}  v   rifie  KT4  comme  Sigma5
```

3.3.3 Lm{E3}, E3 = {Vrai,Faux}

LmE3 correspond à Sigma4 avec p=Vrai d'où
LmE3 = espilon, <Faux>, <Vrai>, [Faux], [Vrai], <Vrai>[Faux], <Faux>[Vrai]
Union   , <Faux>, <Vrai>, [Faux], [Vrai], <Vrai>[Faux], <Faux>[Vrai]
or <Faux> == [Faux] == espilon d'où,

```
Lm{E3} = { espilon , <Vrai>,  [Vrai]} Union { ~, ~<Vrai>, ~[Vrai] = Lm{E2}
```

3.3.4 Lm{E4}, E4 = {Vrai,p} et Lm{E6}, E6 = {Vrai,Faux,p}

On sait que <Faux> == [Faux] == espilon donc

```
Lm{E4} == Sigma4, Lm{E6} = Lm{E5}.
```

3.3.5 Lm{E5}, E5 = {Faux,p}

Pour toute modalité Mod, on a,
|= <Vrai>ModX ssi |= Vrai
|= <Vrai>ModX ssi |= Faux
|= Mod<Vrai>X ssi |= Mod(Vrai)
|= Mod <Vrai>X ssi |= Mod(Faux)
et [Vrai] == <Vrai> et [Vrai] == <Vrai>
Ainsi LmE5 se réduit à LmE2 Union Sigma5, soit

```
Lm{E5} =  {epsilon , <Vrai>, <p>, [p]} Union {~, ~<Vrai>, ~<p>, ~[p]}
```

3.3.6 Lm{E7}, E7 = {Vrai,Faux,p, p}

```
 %<Faux> et [Faux] sont neutres donc
 Lm{E7} = Lm{Vrai, p, ~p}
        = Lm{E2} Union Sigma4  %par le meme raisonnement que precedemment
```

3.3.7 Lm{E8}, E8 = {p,q}

(Lm{p} ∪ Lm{q}) est inclus dans LmE8.
On a :
<p><q> == <q><p> et est idempotent (car le 'ou' est commutatif et 'X ou X == X')
[p][q] == [q][p] et est idempotent (car le 'et' est commutatif et ' X et X == X')
avec :
|= <p><q>X ssi |= p ou q ou X
|= [p][q]X ssi |= p et q et X
|= <p><q>X ssi |= p et q et X
|= [p][q]X ssi |= p ou q ou X
p et q jouent des rôles symétriques.
|= <p>[q]X ssi |= p ou (q et X) |= (p ou q) et (p ou X)
|= <p>[q]X ssi |= p et (q ou X) |= (p et q) ou (p et X)
|= [p]<q>X ssi |= p et (q ou X) |= (p et q) ou (p et X) donc [p]<q> == <p>[q]

4 Conclusion

Ce projet nous a permis de nous familiariser avec avec la programmation en Prolog et de nous rendre compte de la puissance de ce language et des possibilités presque infinies qu'il offre. Il nous a aussi permis d'avoir une meilleure compréhension de la notion de planification et de la manipulation des logiques modales. Nous avons été agréablement surpris de la complexité des résultats que l'on peu obtenir avec très peu de code.