

MIF14

Intelligence Artificielle

TP5-6

LANCE Florian

MATHEOSSIAN Dimitri

A. Moteur en chaînage avant

a. Préambule

Le chaînage avant est une méthode de déduction qui applique des règles en partant des prémisses pour en déduire de nouvelles conclusions. Ces conclusions enrichissent la mémoire de travail et peuvent devenir les prémisses d'autres règles.

Pour réaliser ce genre de chaînage, on ajoute plusieurs règles, par exemple :

```
regle(r1) :- si([fleur,graine]), alors([phanerogame]).
```

Le “si” correspond aux prémisses attendues et le “alors” à la conclusion.





Si ces prémisses sont dans la base des faits, alors on ajoute la conclusion à cette même base, cela permet d’enrichir les connaissances.

b. Initialisation de la base des faits

A l’ajout d’un fait, on a :

```
faits([fleur, graine, un_cotyledone, non(rhizome)]).
```

Ce qui nous intéresse, c’est la liste des prémisses :

-  fleur
-  graine
-  un_cotyledone
-  non(rhizome)

Cette liste est parcourue par le prédicat « `init_faits` » qui va ajouter les prémisses une à une dans la base des faits. Pour cela, on a du déclaré des prédicats dynamiques de façon à utiliser « `assert` ».

On a choisit le codage suivant :

Une prémisses est positive si elle n’est pas précédée par un “non”, sinon elle est négative.

Ce qui nous amène à :

```
init_faits([TeteDeFait|ResteDesFaits]) :-  
  assert(vrai(TeteDeFait)),  
  init_faits(ResteDesFaits), !.
```

Et quand on croise “non”, on ajoute un fait négatif :

```
init_faits([non(TeteDeFait)|ResteDesFaits]) :-  
  assert(faux(TeteDeFait)),  
  init_faits(ResteDesFaits), !.
```

```
init_faits([]) :- !.
```

c. Lancer le programme

La saturation de la base de faits est lancée par le prédicat « `saturer` » :

```
saturer:- moteur(_), fail.
```

Ce dernier va permettre de tester chacune des règles.

d. Moteur

On a utilisé la logique du Moteur d'inférence (3) du cours pour créer le prédicat "moteur" :

"Il donne la réponse en un temps fini et quel que soit l'ordre des Ri et Fj"

```
moteur(Regle):-  
    test_regle(Regle, Fait, Conclusion),  
    not(marque(Regle)),  
    dans_nosFaits(Fait),  
    init_faits(Conclusion),  
    assert(continuer),  
    assert(marque(Regle)),  
    write(Regle),nl,  
    affiche,nl.
```

Le prédicat « `test_regle` » se présente sous la forme suivante :

```
test_regle(Regle, Fait, Conclusion) :-  
    clause(regle(Regle), (si(Fait), alors(Conclusion))).
```

Ce qui nous permet de récupérer la liste des faits de la règle, ainsi que la conclusion.

On vérifie ensuite que la règle n'est pas déjà marquée, si c'est le cas, on vérifie que les faits de cette règle correspondent à des faits de notre base. C'est seulement après que l'on va pouvoir ajouter la conclusion de la règle à notre base de faits, on va donc "enrichir nos connaissances".

La règle est marquée comme "Déjà exploitée", de façon à ne pas boucler indéfiniment.

B. Moteur en chaînage arrière

a. Préambule

Dans cette partie, on va donner des faits, tel que :

```
faits([fleur,graine,un_cotyledone,joli]).
```

On peut utiliser le moteur d'inférence pour savoir si ces faits pourront satisfaire à la création du muguet (par exemple).

On va donc lancer :

```
?-satisfait(muguet).
```

Le moteur d'inférence va chercher les conclusions de chaque règle, et si une règle a pour conclusion "muguet", on va récupérer les éléments qui engendrent ce muguet.

Puis on va tester si ces éléments récupérés sont eux-mêmes engendrés par d'autres éléments.

Si ce n'est pas le cas, alors ce sont des éléments de "base" (rien ne les engendre).

Si ces éléments de "bases" sont dans les faits donnés au départ :

```
[fleur,graine,un_cotyledone,joli]
```

Alors on va supprimer ces éléments dans la "liste des éléments nécessaires à la production du muguet" (s'ils sont dans la liste).

Au final, si cette liste n'est pas vide, alors le muguet ne peut pas être engendré (il manque des "ingrédients") et on retourne **No**, sinon **Yes**.

b. Lancer le programme

Il faut utiliser `satisfait(X)` . où X est l'élément à satisfaire (il faut peupler la BDC avant).

c. Moteur

L'initialisation des faits se passe toujours de la même façon.

Le prédicat `test_regle` n'a pas changé.

Pour lancer le moteur d'inférence, on utilise `satisfait(X)` .

Le prédicat `moteur` est donc exécuté :

```
moteur([X], [], ElemNecessNonTrouve, Resultat, ResultatElemNonTrouve)
```

Ou :

- ✚ X est l'élément que l'on doit essayer de satisfaire,
- ✚ La liste vide va permettre de stocker la liste des opérations au fur et à mesure du programme,
- ✚ ElemNecessNonTrouve : la liste des éléments nécessaires pour créer du muguet
- ✚ Resultat : la liste exploitable des opérations effectuées pendant le programme
- ✚ ResultatElemNonTrouve: la liste des éléments nécessaires pour créer du muguet qui n'ont pas été trouvés

Il y a 4 cas pour le prédicat `moteur` :

Cas d'arrêt : il ne reste plus de fait à relier à une conclusion

Cas 1 : si il existe une conclusion pour l'élément recherché

Cas 2 :

- ✚ S'il n'existe pas de conclusion pour l'élément recherché,
- ✚ et que cet élément **est dans notre base de faits**

Cas 3 :

- ✚ S'il n'existe pas de conclusion pour l'élément recherché,
- ✚ et que cet élément **n'est pas dans notre base de faits**

Pour chaque cas :

Cas 1 : on va conserver les éléments qui engendrent et on va sauvegarder la règle utilisée.

Cas 2 : on supprime l'élément dans la liste des éléments utiles pour engendrer du muguet

Cas 3 : on l'ajoute dans la liste des éléments utiles pour engendrer du muguet

On pourrait ne pas utiliser la liste des éléments nécessaires à la création du muguet. Pour cela, il faudrait utiliser un `assert` dans le cas 3, de façon à créer un prédicat d'erreur. Et s'il est créé, alors le muguet ne peut pas être satisfait.

C. Moteur en chaînage mixte

Le programme effectue en premier un chaînage avant, puis parcourt toutes les règles à la recherche de celles qui sont proches de pouvoir être inférées.

Les prédicats pour le chaînage mixte :

`terminal(F)` : ce prédicat retourne vrai si le fait F est terminal.

`observable(F)` : ce prédicat retourne vrai si le fait F passé en paramètre n'apparaît pas dans la conclusion.

Pas de code pour cette dernière version par manque de temps.