Rapport du Projet Système Expert
IA01- Automne 2013
Aide au choix des UVs en GI

Caimin Xie et Florent Thévenet

13 janvier 2014

Table des matières

1 Problématique

1.1 Aide au choix des uvs en GI

Le but de notre système expert est d'aider les étudiants en Génie Informatique à choisir leurs uvs, pour cela le système leur indiquera si il est possible de faire une certaine uv en fonction de leur profil et quelles uvs ils peuvent et/ou doivent faire.

2 Formalisation

2.1 Besoins

Notre système est d'ordre 0+, par conséquent notre base de fait sera une liste associative (fait, valeur), et les prémisses seront de la forme (nom du fait, opérateur de comparaison, valeur attendue).

Notre système a besoin d'avoir le profil de base de l'utilisateur : semestre, saison en cours et éventuellement origine de l'étudiant (dut ou utc) et filière mais on ne veut pas importuner l'utilisateur en lui posant des questions inutiles.
Par conséquent il faut pouvoir poser des questions pendant le fonctionnement du moteur d'inférence, il y aura donc deux types de conclusions : des conclusions qui assignent une valeur à un fait dans la base et des conclusions qui demandent à l'utilisateur une valeur à entrer dans la base.

2.2 Représentation en Lisp

Une règle aura la forme suivante : (liste_de_prémisses conclusion1 conclusion2 ...)

Une prémisse aura la forme suivante : (fait operateur valeur), où l'opérateur peut être
- un opérateur de comparaison mathématique (< <= >= >)
- un opérateur de comparaison générale (= !=)

Une conclusion est une liste qui commence par un verbe d'action puis comporte les arguments nécessaires à l'action : Nous avons 2 types d'actions :
- set : Définit la valeur d'un fait dans la base de fait. Prend en paramètres le nom du fait et la valeur à stocker.
- ask : Demande à l'utilisateur une valeur. Prend en paramètres le fait à définir et la question à poser.

3 Programme en Lisp

3.1 Moteur d'inférence avec chaînage avant

Notre chaînage avant applique la stratégie suivante : choix de la première règle applicable, exécution des ses conclusions et ainsi de suite jusqu'à ce qu'il n'y ai plus de règles applicables. Chaque règle est utilisée au maximum une seule fois.
Pour des raisons de complémentarité avec le moteur en chaînage arrière nous avons décidé de ne pas faire s'arrêter l'algorithme dès qu'il a trouvé le fait demandé mais d'exécuter toutes les

règles applicables afin de renvoyer toutes les UVs obligatoires / conseillées / faisables selon le profil de l'étudiant.
De par son fonctionnement cet algorithme peut parfois poser des questions inutiles à la détermination du but puisqu'il applique toutes les règles possibles. Ce point est résolu par le moteur en chaînage arrière.

```
Algorithme: verifier_avant (but BF BR)
Debut
    Si (but appartient a la BF)
        alors retourner (assoc but bf) // fin de fonction
    Fin_si

    regles := BR
    faits := BF
    r := chercher_1ere_r_applicable (regles faits)
    f := conclusion (r)
    b := nil
    Tant_que (r <> nil)
        faire
            regles := regles - r
            faits := faits + f
            r := chercher_1ere_r_applicable (regles faits)
            f := appliquer_conclusions ((conclusion (r))
            b := (assoc but faits)
    Fin_tant_que

    retourner b

Fin
```

3.2 Moteur d'inférence avec chaînage arrière

Contrairement au moteur précédent, ce moteur en chaînage arrière et profondeur d'abord permet uniquement de vérifier si une uv est faisable.
Comme il applique uniquement les règles utiles à notre but cet algorithme ne pose jamais de questions inutiles contrairement à l'algorithme en chaînage avant.
Bien sûr il est impossible de lister les uvs faisables par l'étudiant avec ce moteur à moins de le rappeler pour toutes les uvs possibles, mais cette fonction est remplie par le moteur en chaînage avant donc ce n'est pas un problème.
Cela montre qu'il peut être utile d'avoir plusieurs moteurs d'inférence en fonction de ce que l'on veut obtenir.

3.2.1 `verifier_arriere`

La fonction d'entrée pour le moteur en chaînage arrière, on lui passe en argument l'uv à rechercher, le profil de l'étudiant (base de faits) et la base de règles.
Cette fonction est nécessaire dans notre cas puisque nous recherchons la valeur d'un fait, et pas à vérifier si un fait a une valeur donnée comme dans un chaînage arrière classique.
La fonction "classique" de chaînage arrière est `verifier-premisse-arriere`.

3.3 fonctions de services

3.3.1 verif_premisse_simple (premisse fait)

Cette fonction permet de vérifier une seule prémisse à partir d'un fait. Elle retourne le fait si la prémisse est vérifiée, sinon nil. Il s'agit d'une simple comparaison : si les deux arguments ont le même nom, alors on vérifie leur valeur. On utilise le bloc `cond` pour réaliser les différents opérateurs.

```
(defun verif_premisse_simple (premisse fait)
  (if (not (equal (car premisse) (car fait)))
      (return-from verif_premisse_simple nil))
  (let ((v_p (caddr premisse))
    (v_f (cadr fait))
    (op (cadr premisse)))
    (cond
      ((and (equal op '>) (> v_f v_p))
       fait)
      ((and (equal op '>=) (>= v_f v_p))
       fait)
      ((and (equal op '=) (equal v_f v_p))
       fait)
      ((and (equal op '<) (< v_f v_p))
       fait)
      ((and (equal op '<=) (<= v_f v_p))
       fait)
      ((and (equal op '!=) (not (equal v_f v_p)))
       fait)
      (t nil))))
```

3.3.2 verif_premisse (premisses faits)

En utilisant la fonction précédente, on peut maintenant vérifier plusieurs prémisses à partir d'une liste de faits.

```
Algorithme:
Debut
    Si (length faits) < (length premisses)
        alors retourner nil    // fin de fonction
    Fin_si

    ok := vrai
    p := (car premisses)
    Tant_que (p <> nil) ou (ok=vrai)
        faire
            res := faux
            f := (car faits)
            Tant_que (f <> nil) ou (res <> vrai)
                faire
                    res := (or res (verif_premisse_simple p f))
                    f := (car (cdr faits))
            Fin_tan_que
```

```
            ok := (and ok res)
            p := (car (cdr premisses))
    Fin_tant_que

    retourner ok
Fin
```

3.3.3 verif_regle (regle bf)

Elle vérifie si une regèle est applicable à partir de la base de faits : une règle est composée d'une liste de prémisses suivie par des conclusions. Si toutes les prémisses de la règle sont vérifiées dans la base de faits par la fonction `verif_premisse`, elle retourne les conclusions de la règle ; sinon elle retourne nil.

3.3.4 chercher_regle (br bf)

La fonction `chercher_regle` parcourt la base de règle pour trouver la premier règle applicable.

3.3.5 process-conclusions (conclusions BF)

Cette fonction applique les conclusions passées en premier argument à la base de faits puis renvoie la base de fait modifiée.

```
Algorithme: process-conclusions (conclusions BF)
Pour chaque conclusion dans conclusions
    Faire
        si fait(conclusion) n'est pas dans BF
        Faire
            resultat = process-one-conclusion(conclusion)
            si resultat faire
                BF + resultat
            Fin_si
       Fin_si
Fin_pour

Algorithme: process-one-conclusion (conclusion)
si verbe-action = set Faire
    renvoyer (list fait-cible valeur)
sinon si verbe-action = ask Faire
    renvoyer (list fait-cible poser-question())
```

3.3.6 regles-possibles

Renvoie toutes les règles qui définissent le fait passé en paramètre.

3.3.7 regle-define

Si l'une des conclusions de la règle passée en argument définit le fait passé en argument la fonction renvoie cette conclusion, sinon elle renvoie nil.

3.3.8 fait-set?

Récupère le couple (fait valeur) dans la base de faits à partir du fait passé en argument. Renvoie nil si le fait n'est pas défini.

3.3.9 premisses-verifiees?

Cette fonction remplit le même rôle que "verifier-regle" dans le TD5 : Elle vérifie si toutes les prémisses d'un règle sont vérifiées.

```
Algorithme : premisses-verifiees? (regle bf br)
Pour chaque premisses dans premisses(regles)
Faire
    Si Non ( verif_premisse_simple(premisse, fait(premisse), bf) )
    Faire
        Si Non ( verifier-premisse-arrier(premisse, bf, br)
        Faire
            renvoyer NIL
        Fin si
    Fin si
Fin Pour
Renvoyer ok
```

3.3.10 verifier-premisse-arriere

Cette fonction remplit le même rôle que "verifier-fait" dans le TD5 : Elle vérifie si une prémisse peut être vérifiée

```
Algorithme  verifier-premisse-arriere(premisse, br, bf)
Si fait-existe(fait(premisse), bf)
Faire
    renvoyer verif_premisse_simple(premisse, assoc(fait(premisse), bf))
Fin si
Pour chaque regle dans br
Faire
    conclusion := regle-define(fait(premisse), regle)
    si conclusion
    Faire
        ; on regarde si on a le droit d'appliquer cette regle
        ; la recursion se fait ici
        si premisses-verifiees(regle, bf, br)
        Faire
          ; on execute la conclusion pour voir si elle definie
          ; bien la valeur attendue
          ; on peut poser une question a l'utilisateur a ce moment
          couple_resultat := process-one-conclusion(conclusion)
          si verif_premisse_simple(premisse, couple_resultat)
          Faire
              bf += couple_resultat
              renvoyer couple_resultat
          Fin si
        Fin si
```

```
    Fin si
Fin Pour
Renvoyer NIL
```

4 Utilisation

4.1 Utilisation basique

Selon les résultats à afficher on peut utiliser plusieurs fonctions qui prennent toutes les mêmes arguments (uv_recherchee base_de_faits base_de_regles) :

- `verifier_arriere` renvoie l'uv trouvée avec sa valeur associée ou nil si il est impossible de la faire.
- `verifier_avant` renvoie juste l'uv trouvée avec sa valeur associée ou nil si il est impossible de la faire.
- `verifier_avant_` renvoie l'uv trouvée ainsi que la base de fait. Le résultat est à passer en argument à la fonction `resultat_complet` qui affiche de façon lisible le résultat, les uvs obligatoires, conseillées et faisables.

4.2 Avec saisie directe du profil

La profil de l'étudiant peut être saisi dynamiquement grâce à la fonction `ask_profil`. Cette fonction renvoie une base de faits.

```
(verifier_arriere 'sr04 (ask_profil) *br*)
> Quel est votre semestre ? 1
> Somme nous au Printemps (P) ou en Automne (A) ? P
NIL

(verifier_arriere 'sr04 (ask_profil) *br*)
> Quel est votre semestre ? 4
> Somme nous au Printemps (P) ou en Automne (A) ? A
> Quelle est votre filiere ? SRI
(SR04 OBLIGATOIRE)

(verifier_avant 'nf16 (ask_profil) *br*)
> Quel est votre semestre ? 4
> Somme nous au Printemps (P) ou en Automne (A) ? A
> Quelle est votre filiere ? SRI
NIL

(resultat_complet (verifier_avant_ 'nf16 (ask_profil) *br*))
> Quel est votre semestre ? 1
> Somme nous au Printemps (P) ou en Automne (A) ? P
> Venez vous de DUT ou de TC ? TC
* UV choisie : (NF16 OBLIGATOIRE)
* UVs obligatoires:
LO21 : 6 credits TM
NF16 : 6 credits CS
* UVs faisables:
SY02 : 6 credits CS
```

```
MT12 : 6 credits CS
SY14 : 6 credits CS
SR02 : 6 credits CS
RO04 : 6 credits CS
RO03 : 6 credits CS
LO22 : 6 credits TM
SY06 : 6 credits CS
MT10 : 6 credits CS
IA02 : 6 credits CS
FQ01 : 6 credits TM
```

4.3 Avec un profil pré-enregistré

On peut aussi enregistrer son profil dans un fichier texte.
Voici par exemple notre profil de test p3 :

```
semestre
4
saison
a
filiere
strie
uv_acquise
nf16
lo21
end_uv
end
```

Où seuls les champs semestre et saison sont obligatoires.

Pour charger le profil on utilise la fonction `load_profil` qui prend en argument le nom du profil (sous la forme d'un symbole) et crée la base de fait à partir du fichier texte.
Le moteur d'inférence pose toujours des questions si besoin.

```
(resultat_complet (verifier_avant_ 'sy27 (load_profil 'p3) *br*))
* UV choisie : (SY27 OBLIGATOIRE)
* UVs obligatoires:
SY27 : 6 credits TM
* UVs faisables:
LO23 : 6 credits TM
RV01 : 6 credits TM
LO12 : 6 credits TM
GE38 : 6 credits TM
FQ01 : 6 credits TM
GE37 : 6 credits TM

(resultat_complet (verifier_avant_ 'fq01 (load_profil 'p3) *br*))
* UV choisie : (FQ01 FAISABLE)
* UVs obligatoires:
SY27 : 6 credits TM
```

```
* UVs faisables:
LO23 : 6 credits TM
RV01 : 6 credits TM
LO12 : 6 credits TM
GE38 : 6 credits TM
FQ01 : 6 credits TM
GE37 : 6 credits TM

(resultat_complet (verifier_avant_ 'nf16 (load_profil 'p3) *br*))
* UV choisie : (NF16 FAIT)
* UVs obligatoires:
SY27 : 6 credits TM
* UVs faisables:
LO23 : 6 credits TM
RV01 : 6 credits TM
LO12 : 6 credits TM
GE38 : 6 credits TM
FQ01 : 6 credits TM
GE37 : 6 credits TM

(resultat_complet (verifier_avant_ 'sr04 (load_profil 'p3) *br*))
* UV choisie impossible à faire
* UVs obligatoires:
SY27 : 6 credits TM
* UVs faisables:
LO23 : 6 credits TM
RV01 : 6 credits TM
LO12 : 6 credits TM
GE38 : 6 credits TM
FQ01 : 6 credits TM
GE37 : 6 credits TM

Si on modifie le profil p3 pour supprimer la filiere voici ce qu'il
se passe :
(resultat_complet (verifier_avant_ 'sr04 (load_profil 'p3) *br*))
> Quelle est votre filiere ? SRI
* UV choisie : (SR04 OBLIGATOIRE)
* UVs obligatoires:
SR06 : 6 credits TM
SR05 : 6 credits CS
SR04 : 6 credits CS
* UVs faisables:
LO23 : 6 credits TM
RV01 : 6 credits TM
LO12 : 6 credits TM
GE38 : 6 credits TM
FQ01 : 6 credits TM
GE37 : 6 credits TM
```