Université de Technologie de Compiègne

IA01

Intelligence Artificielle : Représentation

Rapport TP3

Réalisation d'un Système Expert à base de règles

Guillaume GALASSO GI01
Alexandre MAZGAJ GI01

9 janvier 2017

Table des matières

Introduction

Dans ce troisième TP de l'UV IA01, on nous propose de réaliser le développement d'un système expert à base de règles de sa phase d'expertise à sa phase d'utilisation.

Dans un premier temps, nous expliciterons le principe et les connaissances manipulée par notre système expert puis nous détaillerons son implémentation en langage Lisp et les raisons de nos choix. Enfin, nous analyserons les résultats obtenus .

1

Principe général du système expert développé

1.1 Enjeux du système expert

Nous avons choisi de réaliser un système expert en relation avec les tours de magie utilisant spécifiquement des cartes. Le but du système est de déterminer le tour auquel l'utilisateur à été soumis en fonction de différents paramètres relatifs à :

— **La manipulation des cartes** qui regroupe l'ensemble des caractéristiques matérielles du tours : nombre de cartes à retrouvées, type de carte (si plusieurs cartes à retrouver), nombre de fois où le magicien a coupé le paquet, $\cdots$

— **Le détournement d'attention** qui désigne l'ensemble des techniques de détournement d'attention fréquemment utilisés dans les tours de magie et la puissance du détournement d'attention qui en résulte (effets combinés possibles).

L'expertise nécessaire à l'élaboration du système à été effectuée par Alexandre MAZGAJ d'après ses connaissances accumulées depuis 4 ans lorsqu'il a commencé à faire de la magie.

1.2 Faits manipulés

Pour commencer le rassemblement de connaissance nécessaire à l'élaboration des règles et des faits, nous avons décider d'établir d'abord les faits qui correspondent aux manipulations effectués et aux comportement qui peuvent aboutir à un détournement d'attention.

Un fois les règles nécessaires définies, cela nous a permis plus facilement d'établir les faits concernés (toujours classés selon les trois même catégories). Ainsi, les faits manipulés représentés sous la forme, *fait (valeur)*, par le système portent sur :

- le nombre de cartes sur lequel porte le tour (nombre positif non nul)
- le type de carte sur lequel porte le tour (rang de (des) la carte(s))
- le comptage des cartes effectuée ou non par le magicien (vrai ou faux)
- le nombre de cartes comptées par le magicien (nombre positif pouvant être nul)
- si le magicien a coupé le paquet
- la fréquence de coupes du paquet effectué par le magicien (une ou beaucoup)
- si le magicien a effectuée des coupes de contrôle : le magicien met une carte à une position qui l'arrange à l'aide d'une coupe
- si le magicien a effectué une fausse coupe : le magicien donne l'illusion de couper le jeu, alors que le jeu reste dans le même ordre
- si le magicien a pris une carte dans sa main
- si le magicien a plié la carte dans sa main : le magicien plie une carte en 4 et l'empalme, sans que les spectateurs le voit faire
- si le magicien a contrôle une carte : le magicien contrôle la position d'une carte dans le paquet (sur le dessus ou le dessous du paquet)
- si la (les) carte(s) concernées est (sont) face(s) retournée(s)
- si des fioritures ont été faites par le magicien : mouvements agréables à oeil
- si le magicien a fait des mouvement remarquable avec des cartes
- si ces mouvements sont suspects
- s'ils sont restreints dans l'espace
- si le magicien a fait un empalmage : le magicien cache une carte dans sa
- si le magicien a fait un pinky break : le magicien garde la position de la carte avec l'aide de son petit doigt.
- si le magicien a effectué un levé double : le magicien prend deux cartes en donnant l'illusion de n'en prendre qu'une seule

— si le magicien a effectué une passe : le magicien coupe le paquet de façon discrète

— si le magicien a effectué un "donné en dessous" : le magicien donne l'illusion de distribuer la première carte du paquet alors qu'il distribue la dernière

— si le magicien a effectué un "Buckle the card" : Pinky break du dessous du paquet

— si le magicien fait un "compte du Biddle" : le magicien peut donner l'illusion qu'il est entrain de compter (par exemple 4 cartes alors qu'il n'en a que 3 dans sa main)

— l'intensité du détournement d'attention fait sur le spectateur (faible fort moyen) : le magicien attire le regard du spectateur vers quelque chose (autre que ses mains) pour pouvoir faire une manipulation

1.3 Règles sur la manipulation des faits

Il nous a également fallu définir des règles permettant de retrouver des mouvements faits durant certains tours spécifiques, pour pouvoir retrouver ces tours.

Nous avons donc rédigé des règles que nous avons regroupé en 3 parties. La première partie correspond aux règles permettant d'arriver au tours rechercher, les prémisses correspondent à des mouvements utilisés pour réaliser le tour.

La deuxième partie correspond aux mouvements que fait le magicien durant le tour, chaque prémisse correspond au ressenti du spectateur sur les mains du magicien.

La troisième partie correspond aux règles sur le détournement d'attention, une partie très importante dans un tour de magie. La magie repose sur le fait que la main est plus rapide que l'oeil, mais certains mouvements demande des gestes trop visible et suspect, donc parfois il faut détourner l'attention du spectateur sur ses mains. Ces règles ont pour prémisses les ressentis du spectateur sur l'attitude du magicien tout le long du tour.

De ce fait, le moteur d'inférence de notre système expert a les caractéristiques suivantes :

— Il est d'ordre 0^+ compte tenu de la nature des faits manipulés indiquée plus haut.

— Nous avons choisi un fonctionnement en chainage avant : on va des faits qu'a rencontré l'utilisateur jusqu'au tour dont il a été sujet en appliquant des règles.

2 Implémentation en Lisp

2.1 Représentation des connaissances utilisées par le système

2.1.1 La base de faits

Un fait est implémenté sous forme de liste d'association (attribut . valeur).

Nous avons initialisé notre base de faits avec une valeur neutre (tel que 0 ou nil) pour pouvoir rechercher et mettre à jour plus facilement notre base de fait avec le moteur d'inférence.

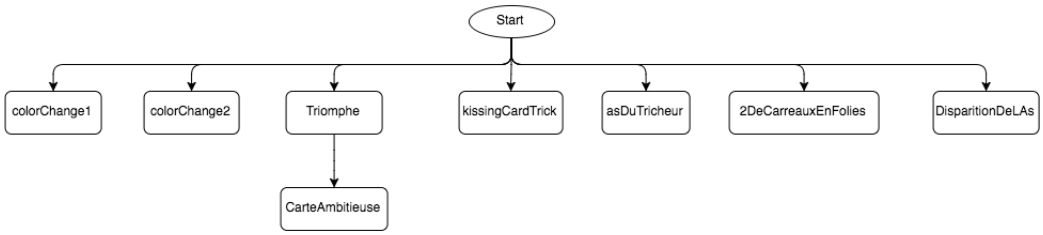
Dans la base de fait ainsi que dans la base de règle, nous avons défini tour1 et tour2 quand à leur priorité car certains tour peuvent nécessiter un autre tour pour leur réalisation.

2.1.2 La base de règles

Toutes nos règles suivent le même schéma : (RX (liste des prémisses) conclusion)

Nous avons défini des règles ayant seulement une seule conclusion, donc nous n'avions pas besoin d'avoir une liste de conclusion, ce qui nous a simplifié la tâche pour appliquer les règles.

Voici ci-dessous l'arbre correspondant aux tours implémentés dans le système.



2.1.3 La base de question

1.3.1 Intérêt de construction d'une base de questions

Dans le but de donner une valeur au faits de base, des questions seront posées à l'utilisateur. En effet, les faits tels que décrits au dessus ne sont pas renseignais par un utilisateur de connaissant pas suffisamment la magie des cartes. Il faut donc les déduire à partir des ses réponses (et de leur succession notamment) nous avons alors décidé de créer une base de questions dans laquelle on a pour chaque question :

— Le numéro de la question

— Le texte de la question

— Les modalités de réponses

A chaque question, un fait est associé.

De plus, certains faits de base ne sont renseignais par l'utilisateur que s'il a répondu à des questions préliminaires. Ces questions sont donc des sous-questions d'autres questions. Il y a donc une hiérarchie entre les questions. Par exemple, on ne peut pas demander le type de carte avant de savoir sur combien de cartes porte le tour.

De même, les questions ne peuvent pas être posées dans n'importe quel ordre. En effet, mise à part la cohérence de l'évolution du dialogue avec l'utilisateur, comme les règles utilisent des faits précis, il faut qu'ils soient renseignés au bon moment pour appliquer les règles. Par exemple, on ne peut pas demander si le magicien compte à un moment les cartes avant de demander sur combien de cartes porte le tour.

Pour chaque question, on attribut un numéro unique qui correspond à une priorité d'ordre minimum. Celui-ci désigne l'étape à laquelle la question doit être posée. Cela permet de renforcer l'intégrité des données et de ne pas être tributaire d'un ordre non formalisé, comme lorsque la question prioritaire par rapport à une autre se situe en dessous ou eu dessus, par exemple. Dans notre cas, l'ordre d'apparition des question dans la base n'a aucune importance.

Voici la liste des questions ordonnée selon leur priorité. Lorsque la question conduit à des sous questions, on donne les modalités de réponse correspondantes.

- Est-ce que le magicien doit retrouver une seule carte en particulier ?
  - oui
  - non : A-t-il fait son tour avec plusieurs cartes ?
    - → oui : Combien de cartes est-ce que le magicien devait retrouver ?
      Quel est le type de carte ?
    - → non
- Est-ce que le magicien a coupé le paquet durant le tour ?
  - oui : A quelle fréquence a-t-il coupé les cartes durant le tour ?
  - non
- Est-ce que des mouvements vous ont parut inhabituels durant le tour ?
  - oui : Est-ce que les mouvements répondaient à une esthétique particulière ?
    Est-ce que le magicien vous a semblait avoir les mouvements restreints (au niveau de la main) durant le tour ?
  - non
- Est-ce que le magicien vous a posé des questions durant le tour
- A-t-il chercher à établir un contact visuel avec vous durant le tour ?
- Est-ce que le magicien à demander de poser toute votre attention sur quelque chose en particulier durant le tour ?
- Est-ce que le magicien vous à donné des infos sur ce qu'il allait faire ou sur le tour en général durant le tour ?
  - oui : Est-ce que le magicien était en train de tenir les cartes nonchalamment dans sa main ?
  - non
- Est-ce que le magicien vous a raconté une histoire pour faire son tour ?
- Est-ce que le magicien a retourne le paquet durant le tour ?

Nous avons tenu à ce que les questions puissent apparaitre dans la base de questions dans n'importe quel ordre. Ceci permet, grâce à la priorité de chaque question, de pouvoir rajouter des questions à la fin en donnant des priorités plus grandes

1.3.2 Représentation Lisp

Pour représenter une question, compte tenu des composantes d'une question, nous avons opté pour une liste de listes d'association selon le format suivant :

((`num_question` numero) (`priorite` priorite) (`question` "question") (`reponse_question` (oui non autres_possibilites)))

Si la question dépend d'une autre question, alors on rajouté les éléments permettant d'identifier une sous question :

((`num_question` numero) (`priorite` priorite) (`question` "question") (`reponse_question` (oui non autres_possibilites)) (`est_sous_question` (`num_question_mere` numero-QuestionMere) (`reponse_question_mere` reponseQuestionMere)))

2.2 Le moteur d'inférence

Notre moteur d'inférence doit, tant que le tour n'est pas trouvé

- Chercher la question à poser selon la sélection de la priorité d'ordre minimum
- Soumettre la question l'utilisateur et récupérer sa réponse
- Sélectionner et appliquer une règle pour mettre à jour le fait concerné par la question. A noter qu'il existe une règle pour mettre à jour chaque fait qui correspond à la réponse à une question posée à l'utilisateur.
- Sélectionner et appliquer des règles pour déduire d'autre faits (et éventuellement le tour de magie)
- Supprimer la question posée

La démarche donnée ci-dessus correspond au processus général pour trouver un tour selon les spécification précédemment décrites. Ce modèle doit être complété pour répondre au problème. C'est l'enjeu de cette seconde partie. Chaque fonction outil renseigne sur l'implémentation réelle du système.

Le système de gestion des question (suppression essentiellement) est affiné et chaque fonction outil concernée définie si besoin de nouvelles contraintes quant au processus de suppression de la question à chaque itération du moteur d'inférence.

2.2.1 Définition des fonctions outils

Compte tenu de l'implémentation de la base de faits, de règles et de questions, la séléction d'un attribut d'une entité est effectuée grâce à la fonction `assoc`. Ensuite, on sélectionne la bonne partie de la liste obtenue avec le car et le cdr.

2.1.1 Fonction `conclusion (regle baseregle)`

Cette fonction renvoie la conclusion d'une règle à partir de son numéro, il suffit de faire un assoc dans la base de règle avec le numéro de a règle puis d'en extraire le car du cdr du cdr.

Listing 2.1 – Fonction `conclusion (regle baseregle)`

```
1 (defun conclusion (regle baseregle)
2    (caddr (assoc regle baseregle)))
```

2.1.2 Fonction `premisse (regle baseregle)`

Cette fonction renvoie la liste des prémisses d'une règle à partir de son numéro.

Listing 2.2 – Fonction `premisse (regle baseregle)`

```
1 (defun premisse (regle baseregle)
2   (cadr (assoc regle baseregle)))
```

2.1.3 Fonction `variableQuestion (question)`

Cette fonction renvoie la variable changé par la réponse d'une question passé en paramètre.

Listing 2.3 – Fonction `variableQuestion (question)`

```
1 (defun variableQuestion (question)
2   (cadr (assoc 'variable question)))
```

2.1.4 Fonction `prioQuestion (question)`

Cette fonction renvoie la variable changé par la réponse d'une question passé en paramètre.

Listing 2.4 – Fonction `prioQuestion (question)`

```
1 (defun prioQuestion (question)
2   (cadr (assoc 'priorite question)))
```

2.1.5 Fonction`poserQuestion (question)`

Dans le but de poser une question a l'utilisateur et de récupérer sa réponse, la fonction `poserQuestion (question)` a été implémentée. Elle prend en paramètre d'entrée une question, l'affiche à l'écran, convertie la réponse en symbole et la retourne.

Tant que la réponse de l'utilisateur n'est pas dans la liste des modalités de réponse, la fonction repose la question.

Listing 2.5 – Fonction `poserQuestion (question)`

```
1  (defun poserQuestion (question)
2    (let ((reponse nil)
3          (reponseInterface nil)
4          (questionPosee (cadr (assoc 'question question)))
5          (listeReponsePossibles (cadr (assoc 'reponse_question question))))
6
7     (while (not (member reponse listeReponsePossibles))
8       (format t "~%~a (modalités de réponse : ~a)~%" questionPosee
9                                                      listeReponsePossibles)
10      (setq reponseInterface (read-line))
11      (cond
12       ((or (equal reponseInterface "oui") (equal reponseInterface "OUI"))
13       (setq reponse 'oui))
14
15       ((or (equal reponseInterface "non") (equal reponseInterface "NON"))
16       (setq reponse 'non))
17       ((or (equal reponseInterface "je ne sais pas")
18       (equal reponseInterface "JE NE SAIS PAS")) (setq reponse 'jnsp))
19
```

```
20        ((or (equal reponseInterface "beaucoup")
21        (equal reponseInterface "BEAUCOUP")) (setq reponse 'bcp))
22
23        ((or (equal reponseInterface "une") (equal reponseInterface "UNE"))
24        (setq reponse 'une))
25
26        ((or (equal reponseInterface "as") (equal reponseInterface "As"))
27        (setq reponse 'as))
28
29        ((or (equal reponseInterface "2d") (equal reponseInterface "2D"))
30        (setq reponse '2d))
31
32        ((or (equal reponseInterface "bcp") (equal reponseInterface "BCP"))
33        (setq reponse 'bcp))
34
35        (T (setq reponse (parse-integer reponseInterface)))
36       ))
37      (return-from poserQuestion reponse)
38      ))
```

A noter que la conversion de la valeur `reponseInterface` (chine de caractère) en symbole est nécessaire au bon fonctionnement du test de validité de la réponse par rapport à la question posée. En effet, ce sont bien des symboles qui sont dans la liste des réponse possibles dans les questions de la base de questions.

Pour la même raison, on convertie la réponse en nombre s'il s'agit d'un nombre (lorsque tous les autres cas ne sont pas vérifiés).

L'utilisateur peut donc donner ses réponses tout en majuscules ou minuscules s'il doit entrer une chaine de caractère. L'utilisateur répond en un mot : il n'y a donc pas de problème d'espace (que l'interpréteur Lisp interpréterai comme deux symboles).

2.1.6 Fonction `selectionQuestion (liste_question)`

Avant qu'une question puisse être posée, elle doit être recherché selon le principe de priorité minimale défini précédemment. Comme les questions peuvent apparaitre dans n'importe quel ordre, il faut parcourir l'ensemble de la base de questions. La question retournée est la question de priorité minimale présente dans la base.

Listing 2.6 – Fonction `selectionQuestion (liste_question)`

```
1  (defun selectionQuestion (liste_question)
2    (let ((selectionQuestion (car liste_question))
3          (proriteQuestion (cadr (assoc 'priorite (car liste_question)))))
4      (dolist (elem liste_question selectionQuestion)
5        (setq proriteElem (cadr (assoc 'priorite elem)))
6          (if (< proriteElem proriteQuestion)
7            (progn
8              (setq proriteQuestion proriteElem)
9              (setq selectionQuestion elem)))
10       )
11     ))
```

Ce procédé s'apparente à la recherche du minimum dans une liste de nombres. On affecte à la variable locale `proriteQuestion` qui désigne la priorité de la question, la priorité de la première question (priorité du car de la liste). De même `selectionQuestion` est initialisé à la première question (car de la liste). Ensuite, on parcours l'ensemble des questions de la liste. Si la priorité d'une question est plus petite que la valeur de `proriteQuestion` alors :

— on affecte à `proriteQuestion` cette priorité.

— on affecte à `selectionQuestion` cette question

2.1.7 Fonction `supprimerSousQuestion (coupleQuestionReponse liste_question)`

Si le moteur d'inférence supprime la question posée à l'utilisateur à chaque itération, cela ne veut pas dire que la prochaine question posée sera légitime, en particulier dans le cas de sous questions. En effet, si la question supprimée possède de sous question, une seule sous question parmi l'ensemble des sous questions de celle-ci devra être posée (en fonction de la réponse à la question mère).

De plus, la priorité de la sous question qui devrait être posée par la suite n'est pas

forcément minimale. Il y a donc un risque de poser une mauvaise sous question. Pour palier à cela, il faut supprimer les sous questions qui ne doivent pas être posée

Il s'agit de celles dont le numéro de la question mère est le numéro de la question supprimée dans le moteur d'inférence et dont la réponse à la question mère est différente de la réponse de l'utilisateur.

Avant de supprimer la question dans le moteur d'inférence, il est nécessaire de sauvegarder dans une variable le numéro de la question et dans une autre la réponse à cette question. On transmet donc a la fonction `supprimerSousQuestion` une liste des deux variables qui applique le principe précédent.

Listing 2.7 – Fonction `supprimerSousQuestion`

```
(defun supprimerSousQuestion (coupleQuestionReponse liste_question)
  (let ((elemSousQuestion nil)
        (numQuestionMere nil)
        (reponseQuestionMere nil)
        (numSousQuestion nil))

    (dolist (elem liste_question liste_question)
      (setq elemSousQuestion (cdr (assoc 'est_sous_question elem)))
      (if (not (null elemSousQuestion))
          (progn
            (setq numQuestionMere (cadr (assoc 'num_question_mere
            elemSousQuestion)))

            (setq reponseQuestionMere (cadr (assoc 'reponse_question_mere
            elemSousQuestion)))

            (if (and (equal (car coupleQuestionReponse) numQuestionMere)
            (not (equal (cadr coupleQuestionReponse) reponseQuestionMere)))
                (progn

                  (setq numSousQuestion (cadr (assoc 'num_question elem)))
                  (setq liste_question (remove elem liste_question))
                  (supressionToutesSousSousQuestion numSousQuestion)))
)))))
```

C'est le rôle de la fonction `supressionToutesSousSousQuestion`.

2.1.8 Fonction `supressionToutesSousSousQuestion (numSousQuestion)`

La fonction reçoit en entrée le numéro de la sous question. La fonction parcours dans une boucle l'ensemble des question de la base et supprime les questions qui ont pour question mère `numSousQuestion`.

La fonction parcours l'ensemble de la base de question. Si une ou plusieurs sous question correspondent, alors elle sont supprimées de cette liste. Il faut également supprimer l'ensemble de leur sous questions puisque celles-ci ne seront pas non plus soumises à l'utilisateur.

Listing 2.8 – Fonction `supressionToutesSousSousQuestion (numSousQuestion)`

```
1  (defun supressionToutesSousSousQuestion (numSousQuestion)
2    (let ((elemSousQuestion nil)
3          (numQuestionMere nil)
4          (numSousSousQuestion nil))
5
6     (if (not (null numSousQuestion))
7         (dolist (elem liste_question liste_question)
8           (setq elemSousQuestion (cdr (assoc 'est_sous_question elem)))
9           (if (not (null elemSousQuestion))
10              (progn
11                (setq numQuestionMere
12                (cadr (assoc 'num_question_mere elemSousQuestion)))
13
14                (if (equal numSousQuestion numQuestionMere)
15                  (progn
16                    (setq numSousSousQuestion
17                    (cadr (assoc 'num_question elem)))
18
19                    (setq liste_question (remove elem liste_question))
20                    (supressionToutesSousSousQuestion numSousSousQuestion)))
21                 )))
22        )))
```

2.1.9 Fonction `Recuperer_la_reponse(QuestionAPoser ReponseQuestionAPoser)`

Cette fonction permet de transformer une réponse d'une question en une valeur utilisable par le moteur d'inférence. A partir d'une question et de la réponse à cette question, la fonction va donner la valeur T si la réponse vaut oui et NIL si la réponse vaut non, puis va faire une liste de la variable, que change la question, et de la réponse.

Listing 2.9 – Fonction `Recuperer_la_reponse`

```
1  (defun Recuperer_la_reponse (QuestionAPoser ReponseQuestionAPoser)
2  (let (variable)
3    (if (assoc 'variable QuestionAPoser)
4      (progn
5        (setq variable (cadr (assoc 'variable QuestionAPoser)))
6        (cond
7          ((equal ReponseQuestionAPoser 'non) (setq ReponseQuestionAPoser nil))
8          ((equal ReponseQuestionAPoser 'oui) (setq ReponseQuestionAPoser T))
9        )
10
11       (cons variable ReponseQuestionAPoser)))))
```

2.1.10 Fonction `ChangerBaseFait (conclu)`

`ChangerBaseFait` permet d'incrémenter la base de fait. Si la conclusion est une liste et que l'attribut de cette conclusion appartient bien à la base de fait, alors on met la valeur dans le couple attribut-valeur, sinon (si la conclusion n'est pas une liste) on push la conclusion dans la liste.

Listing 2.10 – Fonction `ChangerBaseFait (conclu)`

```
1  (defun ChangerBaseFait (conclu)
2    (if (not (equal conclu nil))
3      (if (and (listp conclu) (assoc (car conclu) *baseFait*))
4          (setf (cdr (assoc (car conclu) *baseFait*)) (cdr conclu))
5          (push conclu *baseFait*))))
```

2.1.11 Fonction `est_dans_buts (*baseFait* buts)`

La fonction vérifie si un tour se trouve dans la base de fait, la fonction va le retourner, sinon la fonction renvoi nil. On vérifie d'abord si un tour ne se trouve pas dans l'attribut tour2 (a l'aide de la fonction assoc), sinon alors on va vérifier si un tour ne se trouve pas dans tour1 (toujours à l'aide de la fonction assoc).

Si un tour est trouvé, la fonction vérifie s'il appartient bien à la base des buts (à l'aide de la fonction member), si oui alors la fonction retourne le tour en question, sinon la fonction renvoie nil.

Listing 2.11 – Fonction `est_dans_buts (*baseFait* buts)`

```
1 (defun est\_{dans\_{buts}} (*baseFait* buts)
2 (let (tour)
3     (setq tour (cdr (assoc 'tour2 *baseFait*)))
4         (if (eq tour nil) (setq tour (cdr (assoc 'tour1 *baseFait*))))
5             (if (member tour buts) tour (return-from est_dans_buts nil))
6     ))
```

2.1.12 Fonction `regleCand (etat baseregle)`

Cette fonction renvoie les règles applicables à partir de la base de fait (état) et de la base de règle. Pour chaque regle, la fonction va vérifier si chaque prémisse de la règle appartient à la base de fait.

Si oui, et si la conclusion de la règle n'appartient pas déjà à la base de fait, alors la contions push le numéro de la règle dans une liste, puis elle renvoie cette liste après avoir vérifié chaque règle.

Listing 2.12 – Fonction `regleCand (etat baseregle)`

```
1 (defun regleCand (etat baseregle)
2 (let ((lbis) (verif))
3     (dolist (elem baseregle lbis)
4         (loop for x in (cadr elem) do
5             (if (not (member x etat :test #'equal)) (prog1 (setq verif t)
6                 (return-from nil))))
7
8                 (if (and (not (eq verif t))
9                          (not (member (conclusion (car elem) baseregle) etat
```

```
                        :test #'equal)))

                (setq lbis (append lbis (list (car elem)))))
        (setq verif nil)
    )))
```

2.1.13 Fonction `Question_1premisse (basefait baseregle questions)`

`Question_1premisse` renvoie la première question qui permet de répondre à une règle à laquelle il manque une prémisse. La fonction va pour chaque règle, prendre la longueur de la liste des prémisses dans une variable `valeur`, puis va vérifier si chaque prémisse de la règle appartient à la base de fait, si oui, on décréments de 1 `valeur`, si non la variable `prem` prend la valeur de la prémisse.

Ensuite, la fonction va vérifié pour chaque question, si la variable que va changer la question correspond à l'attribut de la prémisse et si la priorité de la question est inférieur à la valeur `prio`.

Si oui, la fonction va attribuer à la variable `quest` la question testée, et à `prio` la valeur de la priorité de la question testée.

Listing 2.13 – Fonction `Question_1premisse (basefait baseregle questions)`

```
(defun Question_1premisse (basefait baseregle questions)
    (let ((val 0) (prem) (quest) (prio 16))
        (dolist (regle baseregle quest)
            (setq val (length (premisse (car regle) baseregle)))

            (dolist (p (premisse (car regle) baseregle) val)
                (if (member p basefait :test #'equal) (setq val (- val 1))
                    (setq prem p))
            )

            (if (eq val 1) (progn (dolist (q questions quest)
                (if (and (not (assoc 'est_sous_question q))
                         (and (< (prioQuestion q) prio)
                              (equal (car prem) (variableQuestion q))))
                    (progn
                        (setq quest q)
                        (setq prio (prioQuestion quest))))
```

```
18                          )))
19       )
20         quest
21     )
22 )
```

On définit la variable `prio` à 16, qui est la priorité la plus haute dans nos questions.

À la fin de la fonction, `quest` correspond à la question ayant la priorité la plus faible des questions permettant de répondre à la règle à laquelle il manque une prémisse.

2.2.2 Fonctionnement du moteur

Le moteur développé fonction tant qu'il reste une question a poser et si un tour de magie se trouve dans la base de fait (il a été trouvé). C'est l'invariant de boucle. Le moteur d'inférence comporte deux partis.

La première partie consiste à poser des question dont nous avons défini la priorité (certaines question nous semble être essentielle de les poser en premières pour trouver un tour rapidement), questions que nous posons avec les fonctions définies auparavant.

Ensuite nous récupérons la réponse à cette question pour mettre à jour la base de fait. Tant qu'il existe une règle applicable, nous appliquons la règle pour mettre à jour la base de fait.

Dans la deuxième partie, nous utilisons la fonction `Question_1premisse` qui pose une question dont la réponse permet de répondre à une règle dont il manque uniquement une prémisse. Après avoir posé cette question, nous incrémentons la base de fait avec la réponse, puis tant qu'il existe une règle applicable, nous appliquons cette règle.

2.2.3 Implémentation en Lisp

Voici ci-dessous le code complet de notre moteur d'inférence.

Listing 2.14 – Moteur d'inférence

```
(defun moteurInference ()
  (format t "~%~%")
  (affich)
  (format t "Merci␣d'avoir␣lance␣PRESTO,␣votre␣systeme␣expert␣a␣l'epreuve
␣␣des␣magiciens␣~%")

  (format t "PRESTO␣est␣un␣systeme␣expert␣developpe␣par
␣␣GALASSO␣Guillaume␣et␣MAZGAJ␣Alexandre␣~%")

  (defparameter liste_question *questions*)
  (let ((QuestionAPoser 'wesh) (reponse) (regleAppli))

    (while (and (not (est_dans_buts *baseFait* buts)) (not (equal liste_question
    nil)))

     ;;On recupere la reponse et on change la base de fait

     (setq QuestionAPoser (selectionQuestion liste_question))
     (setq ReponseQuestionAPoser (poserQuestion QuestionAPoser))
     (setq liste_question (remove QuestionAPoser liste_question))
     (setq liste_question (supprimerSousQuestion
                           (list (cadr (assoc 'num_question QuestionAPoser))
                           ReponseQuestionAPoser) liste_question))

     (setq reponse (Recuperer_la_reponse QuestionAPoser
                    ReponseQuestionAPoser))

     (ChangerBaseFait reponse)
     ;On va parcourir les regles applicables avec la base de fait actuelle
     (setq regleAppli (regleCand *baseFait* baseregle))
     (while (not (eq regleAppli nil))
      (dolist (r regleAppli)
          (if (not (member (conclusion r baseregle) *baseFait* :test #'equal))
            (progn
              (ChangerBaseFait (conclusion r baseregle))
```

```
36                  (setq regleAppli (regleCand *baseFait* baseregle))
37          ))))
38
39      ;;On va poser une question permettant de repinde a une regle à laquelle il
40      ;;manque une seule premisse
41
42      (setq QuestionAPoser (Question_1premisse *baseFait* baseregle
43                                liste_question))
44
45      (if (not (eq QuestionAPoser nil))
46          (progn
47            (setq ReponseQuestionAPoser (poserQuestion QuestionAPoser))
48            (setq liste_question (remove QuestionAPoser liste_question))
49            (setq liste_question (supprimerSousQuestion
50                                    (list (cadr (assoc 'num_question QuestionAPoser))
51                                    ReponseQuestionAPoser) liste_question))
52
53            (setq reponse (Recuperer_la_reponse QuestionAPoser
54                                                  ReponseQuestionAPoser))
55            (ChangerBaseFait reponse)
56            (setq regleAppli (regleCand *baseFait* baseregle))
57            (while (not (eq regleAppli nil))
58              (dolist (r regleAppli)
59                  (progn
60                    (ChangerBaseFait (conclusion r baseregle))
61                    (setq regleAppli (regleCand *baseFait* baseregle))
62      ))))))
63      (print *baseFait*)
64      (format t "~%Le tour auquel vous avez assisté est ~s~%~%"
65          (est_dans_buts *baseFait* buts))
66
67      (format t "Tutorial :~%~%")
68      (format t "~s~%" (eval (est_dans_buts *baseFait* buts)))
69      (format t "~%Merci d'avoir utilisé PRESTO ~%")
70  ))
```

2.2.4 Analyse de la fiabilité du moteur d'inférence

Le moteur développé est fiable. En effet, celui-ci s'arrête et retourne un résultat cohérent et valide.

2.4.1 Le moteur s'arrête

A chaque itération dans le moteur d'inférence, au moins une question est posée à l'utilisateur et est supprimée. En effet, le moteur va rechercher par l'intermédiaire de la fonction `selectionQuestion (liste_question)` puis elle va être supprimée ainsi que ses éventuelles sous question dont la réponse à la question mère ne correspond pas à la réponse de l'utilisateur ainsi que l'ensemble des sous question (et niveaux inférieurs) associées.

Ainsi, même si la fonction `Question_1Premisse` ne revoie aucun résultat (il maque à tous les tours plus d'une règle pour le valider), le nombre de question de la base de questions est décrémenté au minimum d'une unité (cas d'une questions sans sous question ou d'une sous questions sans niveau inférieur).

Par conséquent, même si aucune règle n'est applicable à aucune des itération, le moteur va s'arrêter selon l'invariant de la boucle donné plus haut. A la fin, lorsque le moteur s'arrête il y a deux possibilités :

— Le moteur trouve à trouvé le tour de magie auquel l'utilisateur a assisté et le retourne.

— Le tour n'a pas été trouvé (la base de question est vide. Le moteur revoie NIL.

2.4.2 Le moteur retourne un résultat valide

Pour l'ensemble des tours implémentés et les règles associés, le moteur fonctionne correctement. En effet, si 'application des règles correspondants aux faits résultant des réponses de l'utilisateur, suit un des chemins dans l'arbre des possibilités, au moins une règle est appliquée à chaque itération.

De ce fait, un tour va pouvoir être trouvé car :

— Il n'est pas possible de trouver dans la base deux tours avec les mêmes prémisses (pas de confusion possible)

— Les faits résultants de l'application des questions (faits déduits par le moteur compris) couvrent l'ensemble de la base de faits

— Les règles correspondent à toutes les successions de faits possibles pour les tours implémentés.

3 Tests et résultats

3.1 Batterie de tests effectués

Notre moteur d'inférence se lance avec la commande *moteurInference).*

L'ensemble des tours implémentés dans le système sont retrouvés à partir du dialogue avec l'utilisateur. Le système répond au problèmr posé.

Nous avons donné dans le fichier Lisp joint au rapport les réponse aux questions à nos tours pour "La disparition de l'as", "Le changement de carte" et "le Triomphe"

3.2 Améliorations possibles

Dans le but d'avoir un système plus performant, il est possibles d'ajouter d'autres tours, ce qui contraint à rajouter des règles ainsi que des faits et des questions si de nouveaux paramètres doivent être pris en compte. Par exemple, ce serait le cas si on voulait étendre le système au tours de magie n'incluant pas que des cartes.

De plus, il existe plusieurs manière d'effectuer le même tour de magie. Même si notre système pose des questions sur les faits principaux permettant d'identifier le tour, si on voulait une gestion plus fine des manipulations du magicien (c'est à dire pouvoir donner le tour et sa manière de procéder, au lieu de quelques explications), un système expert ne semble pas approprié pour cette tache.

Le raisonnement à partir de cas permettrait de répondre en parte à cette problématique car on pourrait rattacher le problème de l'utilisateur à un problème déjà

rencontré mais comme il y a plusieurs façons d'effectuer un tour, le calcul de similarité s'avarierait relativement complexe d'autant plus qu'il serait fonctions des réponses de l'utilisateur au fur et à mesure que le système lui soumet des questions.

Les alternatives possibles serait d'implémenter de l'apprentissage couplé à du traitement sémantique (ontologies et ingénierie linguistique). Ainsi, on pourrait poser des questions laissant plus de libertés à l'utilisateur (notamment avec les questions portant sur le détournement d'attention) tout en pouvant les rattacher à des cas précédemment rencontrés.

Conclusion

Pour conclure, nous pouvons dire que ce projet a été particulièrement instructif quant à l'application des notions théoriques étudiés dans l'UV à un cas concret mais aussi quant à conception de la démarche de résolution du problème.

La représentation des connaissance s'est avérée primordiale pour un traitement efficace des informations à postériori. Cela nous a permis de revenir sur les différentes étapes de résolution de ce type de problèmes, que ce soit par rapport à la modélisation du problème (représentation des connaissances, choix du moteur), implémentation en langage Lisp (fonctions de service, moteur d'inférence) ou l'analyse des résultats.