COLLOQUE INTELLIGENCE ARTIFICIELLE BERDER, 13-16 SEPTEMBRE 1998

RÉSUMÉ

Ce rapport contient la majeure partie des interventions effectuées lors du colloque Intelligence Artificielle qui s'est tenu, du 13 au 16 septembre 1998, en l'île de Berder. Il fait état d'un certain nombre de travaux qui ont été effectués au cours de l'année 1998 par l'équipe Métaconnaissances du LIP6. Il constitue la livraison pour 1998 du document d'état de l'art dont DER-EDF a passé commande dans le contrat 129/1K9859/IMA 346.

MOTS-CLÉS

Intelligence Artificielle, Analogies, Métamodélisation, Spécialisation de Programmes, Stratégie, Arbitrage, Généralisation.

ABSTRACT

These proceedings contain papers selected from the Artificial Intelligence Workshop that was held September 13-16, 1998, on Berder island. It is a summary of the LIP6 MetaKnowledge Group work in 1998, which constitutes the 1998 state-of-the-art report that was ordered by DER-EDF in the contract 129/1K9859/IMA 346.

KEYWORDS

Artificial Intelligence, Analogies, Metamodelling, Program Specialisation, Strategy, Decision Making, Generalization

SOMMAIRE

INTRODUCTION

L'équipe Métaconnaissance du LIP6 ainsi que plusieurs chercheurs d'EDF se sont réunis dans l'île de Berder du 14 au 16 septembre 1998. Ce rapport contient les textes de quelques unes des interventions qui y ont été faites.

D'abord, trois articles généraux veulent nourrir la réflexion sur des problèmes fondamentaux de l'IA. Le premier fait le point sur un problème important en informatique : comment transformer un programme de façon à ce qu'il s'exécute plus rapidement. La technique étudiée ici est la spécialisation. Beaucoup de programmes ont des niveaux d'interprétation bien qu'ils ne soient pas des interpréteurs ; enlever ces niveaux les accélère considérablement. Cela permet de réunir les avantages de la généralité et de l'efficacité.
Le papier suivant montre l'originalité de l'avant-dernier livre de Douglas Hofstadter et en apprécie les limites. Ce livre porte sur la créativité et contient la description de plusieurs systèmes expérimentés au sein de son équipe. En particulier, un des systèmes fait intervenir une multitude de "codelets", acteurs qui examinent l'espace de travail, le modifient et peuvent éventuellement créer d'autres codelets.
Nous recherchons de plus en plus en IA à réaliser des systèmes génériques, qui peuvent s'appliquer à un grand nombre de domaines. De tels systèmes ont deux niveaux, généraliste et spécialiste, qui communiquent et entre lesquels les tâches sont réparties. Il est important d'examiner les moyens de réaliser de tels systèmes et les difficultés que l'on rencontre en les implémentant.

Les cinq papiers suivants présentent des travaux en cours de réalisation dans l'équipe. Le premier porte sur la généralisation à partir d'un nombre restreint d'observations. Les approches classiques ont des limitations qu'il examine et que le système GENSAM doit surmonter. Ce système va d'abord être expérimenté dans le domaine médical.
Nous avons ensuite une étude sur la reformulation des problèmes. Une partie importante du travail du chercheur en IA est de transformer l'énoncé d'un problème de façon à ce que le système de résolution soit efficace. Mais alors, l'intelligence est surtout dans la tête du chercheur et non dans le système ; les systèmes futurs devraient pouvoir, comme nous, modifier les énoncés des problèmes qui leur sont posés.
Le papier suivant porte sur le système Lombric capable de comprendre ce qui se passe dans une scène réelle où beaucoup d'agents apparaissent, dans le cas présent suivre le comportement des voitures dans un carrefour complexe .Lombric doit pouvoir interpréter dans des temps de l'ordre de la seconde les images successives venant de plusieurs caméras qui n'ont pas une vision totale de toutes les parties du carrefour.
Les administrateurs d'un réseau d'ordinateurs ont une tâche difficile et l'on manque actuellement de spécialistes compétents. Il serait donc souhaitable d'avoir des systèmes qui résolvent automatiquement un certain nombre de problèmes ; même s'ils n'arrivent pas toujours à réparer les dégâts, ils devraient pouvoir donner à l'administrateur toutes les informations qui lui permettront de trouver la cause des difficultés. Ce sont les buts du système Hammourabi.
Enfin, si l'on commence à avoir des systèmes capables d'apprendre des connaissances tactiques dans le domaine des jeux, on est loin de savoir en faire autant pour les connaissances stratégiques. En effet, les conséquences d'une décision stratégique n'apparaîtront que de nombreux coups plus tard, et elles sont souvent cachées par de nombreuses décisions tactiques, bonnes ou mauvaises, des deux camps. Si des programmes tactiques peuvent donner des résultats satisfaisants dans des jeux comme les échecs, la dimension stratégique est capitale dans d'autres jeux comme le Go. Le système présenté ici doit apprendre des connaissances stratégiques avec des méthodes générales, mais il a été appliqué en priorité au Go.

Nous remercions Régis Moneret et Arnaud Fredon pour avoir parfaitement préparé l'organisation matérielle de ce colloque, qui a pu avoir lieu grâce à un contrat EDF. Merci encore à Régis Moneret qui a bien voulu s'occuper de l'édition de ce rapport.

Jacques PITRAT

ETAT DE L'ART SUR LA SPÉCIALISATION DE PROGRAMMES

Tristan CAZENAVE

cazenave@ai.univ-paris8.fr

Laboratoire d'Intelligence Artificielle

Université Paris 8

2 Rue de la Liberté, 93 Saint Denis

Résumé

La spécialisation de programmes est aussi appelée l'évaluation partielle. C'est une technique de transformation de programmes qui consiste à évaluer une partie d'un programme générique sur certaines de ses entrées de façon à ce qu'il s'exécute ensuite plus rapidement pour ces entrées particulières. Nous verrons ses applications actuelles et son intérêt potentiel pour l'Intelligence Artificielle.

1. Introduction

La spécialisation de programmes est aussi appelée l'évaluation partielle. C'est une technique de transformation de programmes qui consiste à évaluer une partie d'un programme générique sur certaines de ses entrées de façon à ce qu'il s'exécute ensuite plus rapidement pour ces entrées particulières. Cette technique a été utilisée avec succès sur de nombreux langages, des langages fonctionnels (Scheme, Escher…), logiques (Prolog, Goedel) et impératifs (C, Pascal, Fortran).
Nous commençons par replacer, dans la section 2, la spécialisation de programme dans le cadre plus général des transformations de programmes, de nombreux exemples sont tiré de l'excellent papier de Alberto Pettorossi et Maurizio Proietti [Pettorossi et al. 98] et aussi de [Pettorossi et al. 96]. La section 3 décrit l'évaluation partielle. La section 4 décrit son application au langage C. Nous concluons avec la section 5 qui donne des pistes de l'utilité potentielle de la spécialisation de programmes pour l'Intelligence Artificielle.

2. Transformations de programmes logiques

2.1. Transformations de base

Les transformations de base utilisées sont :

- Le dépliage
- Le pliage
- La définition
- La généralisation

Le dépliage [Tamaki et al. 84] correspond au remplacement d'un but dans le corps d'une clause par sa définition.

Dépliage : Soit P un programme logique contenant la clause C de la forme H:-F,A,G où A est un littéral positif et F et G des buts pouvant être vides. On suppose :
$D_1,\ldots,D_n$, est l'ensemble de toutes les clauses du programme P telles que A est unifiable avec les Tete(D_i) en utilisant les unificateurs les plus généraux $\theta_1,\ldots,\theta_n$.
Ci est la clause (H:-F, Corps(D_i),G) θ_i, pour i=1,…,n.
Le dépliage de A dans la clause C dans le programme P consiste à remplacer la clause C par les clauses $C_1,\ldots,C_n$.

Soit P le programme suivant :

```
p(X) :- q(X), r(X).
q(a).
r(a).
r(b).
```

Spécialiser le programme P en le dépliant permet d'obtenir le programme :

```
p(a).
q(a).
r(a).
r(b).
```

Soit le programme suivant :

```
grandmere(X,Y) :- mere(X,Z), parent(Z,Y).
grandpere(X,Y) :- pere(X,Z), parent(Z,Y).
parent(X,Y) :- mere(X,Y).
parent(X,Y) :- pere(X,Y).
mere(alexandra,pierre).
mere(alexandra,mathilde).
mere(chantal,alexandra).
```

Lorsqu'on spécialise ce programme sur le but grandmere(X,pierre) en dépliant toujours l'atome le plus à gauche, et en ne dépliant pas le prédicat 'pere' dont les instances ne sont pas connues lors de la spécialisation, on obtient :

```
grandmere(alexandra,pierre) :- pere(pierre, pierre).
grandmere(alexandra,pierre) :- pere(mathilde, pierre).
grandmere(chantal,pierre).
grandmere(chantal,pierre) :- pere(alexandra, pierre).
```

Le pliage est l'opération inverse du dépliage. Elle correspond donc au remplacement d'un ensemble d'atome dans le corps d'une clause par la un seul atome correspondant au prédicat plié.

Pliage : Soit P un programme logique contenant les clauses $C_1,\ldots,C_n$ ainsi que les clauses $D_1,\ldots,D_n$. Supposons qu'il existe un atome A et deux buts F et G tels que pour tout i compris entre 1 et n, il existe une substitution θ_i qui satisfait :
C_i est une variante de la clause H:-F, Corps(D_i) θ_i,G.
A= Tete(D_i) θ_i
Pour chaque clause D de P qui n'est pas dans $D_1,\ldots,D_n$, Tete(D) n'est pas unifiable avec A
Pour toute variable X dans l'ensemble vars (Di)-vars(Tete(Di)), on a :
$X\theta_i$ est une variable qui n'apparaît pas dans (H,F,G)
La variable $X\theta_i$ n'apparaît pas dans le terme $Y\theta_i$, pour toute variable Y de Corps(D_i) et différente de X.
Le pliage des clauses $C_1,\ldots,C_n$ en utilisant les définitions $D_1,\ldots,D_n$ dans P consiste à remplacer les clauses Ci par la clause C = H:-F,A,G.

Définition : On définit une clause newp($X_1, \ldots, X_m$) :- $B_1, \ldots, B_n$. où newp est un nouveau nom de prédicat, où les B_j sont des atomes dont les prédicats appartiennent au programme initial et où les X_i sont un sous ensemble des variables des B_j, en ajoutant la clause au programme.

Soit le programme Palyndrome qui teste (inefficacement) si une liste est un palyndrome :

```
pal([]).
pal([H]).
pal([H|T]) :- append(Y,[H],T), pal(Y).
append([],Y,Y).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

On définit le prédicat newp :

```
newp (L,T) :- append(Y,L,T), pal(Y).
```

Plier le prédicat newp dans le programme Palyndrome consiste à remplacer la ligne 3 par la ligne 3f :

```
3f. pal([H|T]) :- newp([H],T).
```

Généralisation :
La généralisation de deux termes t_1 et t_2 consiste à trouver un terme t et deux substitutions σ_1 et σ_2, t le plus précis possible, tels que $t\sigma_1=t_1$ et $t\sigma_2=t_2$, on dit que t est la généralisation la plus précise des termes t_1 et t_2.
Cela revient à trouver une formule qui généralise deux formules données. La généralisation est toujours possible, ne serait ce qu'avec le terme t correspondant à une variable X et avec $\sigma_1=(X,t_1)$ et $\sigma_2=(X,t_2)$.

Exemples d'applications de l'algorithme de généralisation à des couples de termes :

f(g(a,b),X) et f(a,g(b,X)) donnent σ_1={(Y,g(a,b)),(Z,X)} et σ_2={(Y,a)),(Z,g(b,X))} et t=f(Y,Z)

f(a,a) et f(g(b,b),g(b,b)) donnent σ_1={(X,a)} et σ_2={(X,b)} et t=f(X,X)

f(a,g(a,b)) et f(b,g(b,b)) donnent σ_1={(X,a)} et σ_2={(X,b)} et t=f(X,g(X,b))

2.2. Le tupling

Le tupling est un stratégie de transformation de programmes qui consiste à ne pas faire faire au programme plusieurs fois le même travail. Elle consiste a regrouper les sous buts communs à plusieurs atomes

Soit le programme suivant qui traverse deux fois une liste, une fois pour la construire, une fois pour calculer sa longueur :

```
1. append ( [], L, L).
2. append([H|L1], L, [H|L2]) :- append(L1, L, L2).
3. length([] , 0).
4. length([T|L], X) :- length (L,Y), X is Y+1.
5. append_length(L1, L2, L3, X) :- append(L1,L2,L3), length(L3,X).
```

On obtient par dépliage de append :

```
6. append_length ( [], L, L, X) :- length (L,X).
7. append_length ( [T|R], L1, [T|L2], X) :- append (R, L1, L2), length
([T|L2], X).
```

Puis par dépliage de length :

```
8. append_length ( [T|R], L1, [T|L2], X) :- append (R, L1, L2), length (L2,
Y), X is Y+1.
```

Enfin par pliage de (5) dans (8)

```
9. append_length ( [T|R], L1, [T|L2], X) :- append_length (R, L1, L2, Y), X
is Y+1.
```

D'où le programme final qui ne traverse qu'une fois la liste :

```
1. length([] , 0).
2. length([T|L], X) :- length (L,Y), X is Y+1.
3. append_length ( [], L, L, X) :- length (L,X).
append_length ( [T|R], L1, [T|L2], X) :- append_length (R, L1, L2, Y), X is
Y+1.
```

La transformation que nous avons effectuée sur le programme append_length s'appelle le tupling, elle consiste à remplacer deux parcours de liste par un seul parcours qui fait les deux opérations à la fois quand c'est possible.

Tupling : Soit une clause C de la forme

$$H :- A_1,\ldots,A_m,B_1,\ldots,B_n.$$

On définit le prédicat newp en insérant la clause T :

$$newp(X_1,\ldots,X_k) :- A_1,\ldots,A_n.$$

Où les arguments $X_1,\ldots,X_n$ sont les éléments de $vars(A_1,\ldots,A_m) \cap vars(H, B_1,\ldots,B_n)$. On cherche alors à trouver une définition récursive du prédicat newp par des pliages et des dépliages utilisant la clause T.
Le tupling est souvent utilisé quand $A_1,\ldots,A_m$ ont des variables communes. Les améliorations dues à cette transformation viennent de ce que l'on n'évalue qu'une seule fois des buts qui sont communs aux atomes $A_1,\ldots,A_m$. Le tupling permet aussi d'éviter de visiter plusieurs fois des structures de données et d'éviter la construction de données intermédiaires.

On peut aussi appliquer la stratégie du tupling au programme palyndrome donné précédemment. pal (Y) et append(Y,[H],T) traverse la même liste Y. Le tupling va chercher a éviter cette duplication de la visite de la liste. On définit pour cela le prédicat newp :

```
newp(L,T) :- append(Y,L,T),pal(Y).
```

(l'argument [H] dans le corps de la clause 3 a été généralisé à L), et on insère cette clause comme la clause 6 dans le programme.On plie alors la définition de 6 dans la clause 3, on obtient :

```
3f. pal([H|T]) :- newp([H],T).
```

La double visite n'est toujours pas évitée en remplacant 3 par 3f. On doit pour cela trouver une définition récursive de newp. Pour cela, on commence par déplier pal(Y) dans la clause 6, on obtient:

```
7. newp(L,T) :- append([],L,T).
8. newp(L,T) :- append([H],L,T).
9. newp(L,T) :- append([H|Y],L,T), append(R,[H],Y), pal(R).
```

On déplie alors append dans les clauses 7, 8 et 9 et on obtient :

```
10. newp(L,L).
11. newp(L,[H|L]).
12. newp(L,[H|U]) :- append(Y,L,U), append(R,[H],Y), pal(R).
```

Maintenant pour avoir une définition récursive de newp on va chercher à plier 6 dans 12. Pour cela on commence par remplacer append(Y,L,U), append(R,[H],Y) par append(R,[H|L],U), on obtient :

```
13. newp(L,[H|U]) :- append(R,[H|L],U), pal(R).
```

puis on peut alors plier 6 dans 13f :

```
13f. newp(L,[H|U]) :- newp([H|L],U).
```

Le programme transformé par tupling permet ainsi de ne parcourir qu'une seule fois la liste de caractères à tester.

2.3. La déforestation

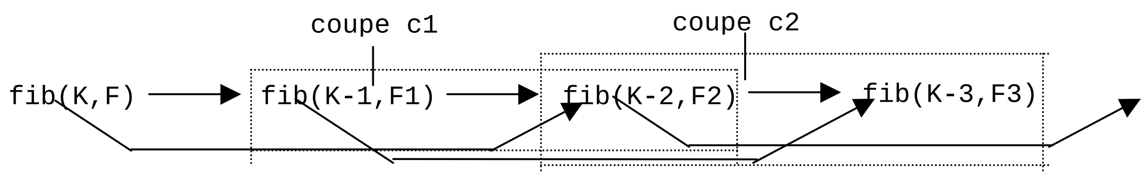
La Déforestation : La déforestation est une stratégie de transformation de programmes qui a pour but d'éliminer les structures de données intermédiaires dans les programmes en introduisant de nouvelles définitions qui sont égales à la composition de prédicats déjà définis.

On veut transformer le programme suivant qui calcule les nombres de Fibonacci :

```
1. fib(0,1).
2. fib(1,1).
3. fib(N,F) :- N1 is N-1,fib(N1,F1),N2 is N1-1,fib(N2,F2),F is F1+F2.
```

Pour un nombre donné, le programme calcule un nombre exponentiel de sommes. Complexité = $O(2^n)$ pour le nième nombre de Fibonacci.

Lorsqu'on représente sur un même schéma pour les nombres de Fibonacci d'indices k, k-1 et k-2 les instances de prédicats auxquels il font appel. On observe les régularités suivantes :



Les régularités sont représentées par les coupes c1, c2 … On introduit un nouveau prédicat t(N,F) pour prendre en compte ces régularités, puis on applique la stratégie du tupling en dépliant cette définition et en repliant pour trouver une définition récursive qui sera appelée pour calculer fib(N,F).

On introduit la définition suivante :

```
4. t(N,[F1,F]) :- N1 is N+1, fib(N1,F1), fib(N,F).
```

On repliant la définition dans fib(N,F) on obtient :

```
3a. fib(N,F) :- N2 is N-2, t(N2,[F1,F2]), F is F1+F2.
```

En dépliant fib(N1,F1), on obtient :

```
5. t(-1,[1,F]) :- fib(-1,F).
6. t(0,[1,1]).
7. t(N,[F1,F]) :- N1 is N+1, N2 is N1-1, fib(N2,F2), N3 is N2-1, fib(N3,F3),
F1 is F2+F3, fib(N,F).
```

Qui devient après les simplification sur les affectations :

```
7a. t(N,[F1,F]):- fib(N,F2), N3 is N-1, fib(N3,F3), F1 is F2+F3, fib(N,F).
```

Qui devient après qu'on ait remarqué que a un N donné ne correspond qu'un seul nombre de Fibonacci, ce qui permet d'unifier F et F2 :

```
7b. t(N,[F1,F]):- N3 is N-1, fib(N3,F3), fib(N,F), F1 is F+F3.
```

On peut rajouter N is N3+1, dans cette clause puisque on a N3 is N-1, En repliant t, on obtient alors :

```
8. t(N,[F1,F]):- N3 is N-1, t(N3,[F,F3]), F1 is F+F3.
```

Le programme final est alors :

```
fib(0,1).
```

```
fib(1,1).
fib(N,F) :- N2 is N-2, t(N2,[F1,F2]), F is F1+F2.
t(-1,[1,F]) :- fib(-1,F).
t(0,[1,1]).
t(N,[F1,F]):- N3 is N-1, t(N3,[F,F3]), F1 is F+F3.
```

Il a une complexité en O(n) pour le nième nombre de Fibonacci.

2.4. Exemple d'application : la compilation du contrôle

La stratégie de Prolog pour contrôler la SLD résolution consiste à évaluer les clauses dans l'ordre dans lequel elles ont été données, de gauche à droite. Cette stratégie simple ne donne pas toujours l'efficacité voulue. Une solution pour améliorer la stratégie de contrôle est de définir une stratégie de contrôle améliorée puis de transformer les programmes utilisant cette stratégie améliorée en des programmes équivalents utilisant la stratégie simple.
Les sous séquences communes : Une séquence est représentée par une liste d'éléments. Pour deux séquences X et Y, on a subseq(X,Y) si et seulement si X est une sous séquence de Y, c'est à dire que X peut être obtenue à partir de Y en lui enlevant des éléments.
Le programme suivant permet de vérifier que X est une sous séquence commune aux deux séquences Y et Z. Pour cela il commence par vérifier que X est une sous séquence de Y, puis il vérifie que X est une sous séquence de Z.

```
1. csub(X,Y,Z) :-subseq(X,Y),subseq(X,Z).
2. subseq([],X).
3. subseq([A|X],[A|Y]) :-subseq(X,Y).
4. subseq([A|X],[B|Y]) :-subseq([A|X],Y).
```

On veut appliquer à ce programme la stratégie de résolution S suivante :

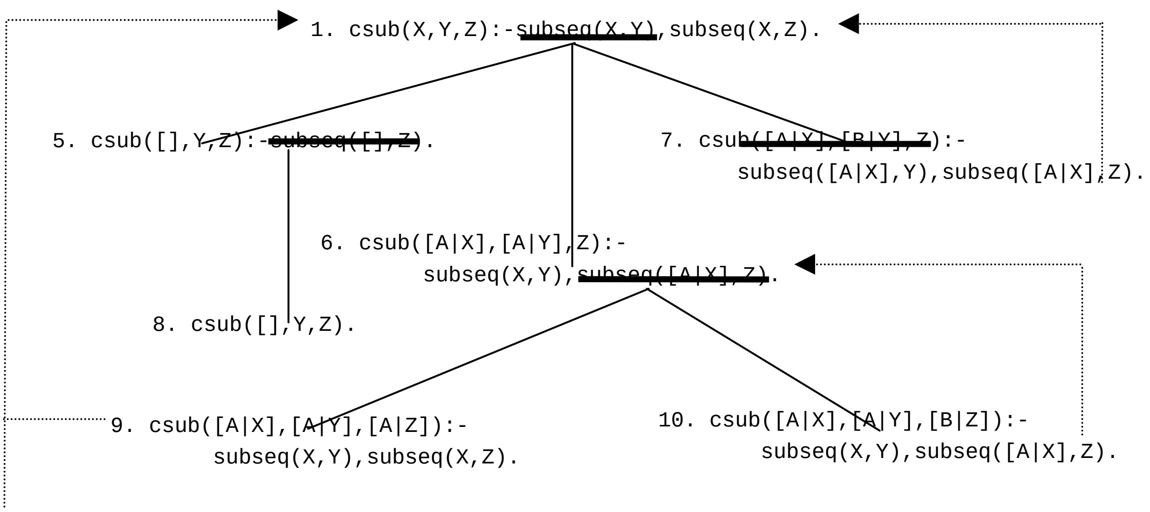
Si le but est 'subseq(w,x),subseq(y,z)' et que w est un sous terme de y
Alors choisir de résoudre l'atome subseq(y,z)
Sinon choisir de résoudre l'atome le plus à gauche dans le but.

L'utilisation de cette stratégie est à priori plus efficace que la stratégie de gauche à droite de Prolog parce que la deuxième occurrence de subseq est choisie dès qu'elle est suffisamment instanciée par l'évaluation de la première occurrence. L'utilisation de cette stratégie permettra donc à un but de la forme 'subseq(...),subseq(...)' d'échouer même si la première occurrence n'a pas été complètement évaluée.

On définit une stratégie de dépliage Sd qui corresponde à la stratégie de résolution S définie précédemment :

Si le corps de la clause à déplier est 'subseq(w,x),subseq(y,z)' et que w est un sous terme de y
Alors choisir de déplier l'atome subseq(y,z)
Sinon choisir de déplier l'atome le plus a gauche dans le corps de la clause.

On déplie alors les prédicats subseq dans la clause 1 en utilisant la stratégie de dépliage définie précédemment :



On définit alors le prédicat newcsub et on plie les clauses définissant csub qui peuvent l'être.

```
11. newcsub(A,X,Y,Z) :-subseq(X,Y),subseq([A|X],Z).
```

On peut plier newcsub dans la clause 6 :

```
6f. csub([A|X],[A|Y],Z) :-newcsub(A,X,Y,Z).
```

On peut aussi plier la clause 7 en utilisant la clause 1

```
7f. csub([A|X],[B|Y],Z) :-csub([A|X],Y,Z).
```

On peut déplier de nouveau la clause newcsub en utilisant la stratégie de dépliage Sd définie.

On applique à la clause 11, les dépliages correspondants à ceux qui amène de la clause 6 aux clauses 9 et 10. On trouve alors les clauses 12 et 13 :

```
12. newcsub(A,X,Y,[A|Z]) :-subseq(X,Y),subseq(X,Z).
13. newcsub(A,X,Y,[B|Z]) :-subseq(X,Y),subseq([A|X],Z).
```

En pliant les clauses newcsub en utilisant les définitions présentes dans le programme, on obtient :

```
12f. newcsub(A,X,Y,[A|Z]) :-csub(X,Y,Z).
13f. newcsub(A,X,Y,[B|Z]) :-newcsub(A,X,Y,Z).
```

Ce qui donne le programme final ainsi obtenu.

```
8. csub([],Y,Z).
6f. csub([A|X],[A|Y],Z) :-newcsub(A,X,Y,Z).
7f. csub([A|X],[B|Y],Z) :-csub([A|X],Y,Z).
12f. newcsub(A,X,Y,[A|Z]) :-csub(X,Y,Z).
13f. newcsub(A,X,Y,[B|Z]) :-newcsub(A,X,Y,Z).
```

Ce programme est plus efficace que l'original. La stratégie de contrôle a été compilée dans le programme original pour donner le programme final.

2.5. Autre exemple : la composition de programmes

Une technique courante en programmation est de diviser un problème en sous problèmes plus faciles, d'écrire des sous programmes pour ces sous problèmes pour finalement composer les sous programmes en un programme qui résout le problème. Toutefois cette façon de procéder amène souvent à des programmes inefficaces parce que les différents sous programmes ne partagent pas des calculs qu'ils pourraient partager. Pour résoudre les problèmes dus à ce type de programmation, deux types de techniques sont utilisées en métaprogrammation :

l'amélioration de l'évaluateur en utilisant par exemple un ramasse-miettes, le memoing, une lazy évaluation ou le coroutinage.
la transformation du programme en un autre programme qui peut être efficacement évalué par un évaluateur standard.

Nous allons voir comment cela est possible sur le programme suivant qui ôte d'une liste X de nombres positifs toutes les occurrences de son maximum :

```
deletemax(Xs,Ys) :- maximal(Xs,M),delete(M,Xs,Ys).
maximal([],0).
maximal([X|Xs],M) :- maximal(Xs,N), M is max(N,X).
```

```
delete(M,[],[]).
delete(M,[M|Xs],Ys) :- delete(M,Xs,Ys).
delete(M,[X|Xs],[X|Ys]) :- M=\=X,delete(M,Xs,Ys).
```

On veut obtenir un programme qui ne traverse la liste Xs qu'une seule fois. Pour cela, on applique la stratégie du tupling aux prédicat maximal et delete qui partagent l'argument Xs.

En dépliant delete dans la clause deletemax, on obtient :

```
7. deletemax([],[]).
8. deletemax([M|Xs],Ys) :-maximal(Xs,N), M is max(N,M), delete(M,Xs,Ys).
9. deletemax([X|Xs],[X|Ys]) :-maximal(Xs,N),M is
max(N,X),M=\=X,delete(M,Xs,Ys).
```

A ce stade, le pliage n'est pas possible, on va donc donner une nouvelle définition qui va permettre le pliage en utilisant la généralisation. On définit un nouveau prédicat gen(Xs,P,Q,Ys) qui généralise les deux buts pour permettre le pliage. On généralise les deux buts 'maximal(Xs,N),delete(M,Xs,Ys)' et 'maximal(Xs,M),delete(M,Xs,Ys)' ce qui donne :

```
10. gen(Xs,P,Q,Ys) :-maximal(Xs,P),delete(Q,Xs,Ys).
```

En appliquant la stratégie du tupling en pliant le prédicat défini dans la clause 1, on obtient :

```
1f. deletemax(Xs,Ys) :-gen(Xs,M,M,Ys).
```

On trouve alors une définition récursive du prédicat nouvellement défini en dépliant le prédicat delete puis en pliant avec la définition du nouveau prédicat :

```
11. gen([],0,Q,[]).
12. gen([X|Xs],P,X,Ys) :-gen(Xs,N,X,Ys),P is max(N,X).
13. gen(([X|Xs],P,X,[X|Ys]) :-gen(Xs,N,Q,Ys),Q=\=X,P is max(N,X).
```

Le programme ainsi obtenu ne parcoure qu'une seule fois la liste comme on le voulait.

2.6. Le changement de représentation des données

Le choix de structures de données appropriées au problème est généralement très important pour écrire des programmes efficaces, on a même introduit une équation qui résume ceci : 'Algorithmes + Structures de Données = Programme'.
Toutefois, il est parfois difficile de connaître les structures de données adaptées à un problème avant d'avoir écrit un programme pour résoudre ce problème. La transformation de programme peut permettre de transformer les structures de données d'un programme pour le rendre plus efficace.
Par exemple, nous allons transformer un programme qui utilise des listes en un programme qui utilise les différence-listes. Une différence-liste est une paire de listes L-R, telle qu'il existe une troisième liste telle que lorsqu'on ajoute X et R, on retrouve L. Une liste peut être représentée par de nombreuses difference-listes.
L'utilisation des difference-listes en transformation de programme se fait souvent en introduisant une définition de la forme :

```
diffp(X,L-R):-p(X,Y),append(Y,R,L).
```

et en trouvant ensuite une définition récursive de diffp qui ne dépend ni de p ni de append.
Nous allons appliquer cette méthode au programme suivant qui retourne une liste :

```
1. reverse([],[]).
2. reverse ([H|T],R) :- reverse(T,V), append(V,[H],R).
3. append([],L,L).
4. append ([H|T],L,[H|S]) :- append(T,L,S).
```

Pour une liste de longueur n, la complexité du programme est en $O(n^2)$. On veut obtenir un programme qui ne traverse la liste Xs qu'une seule fois. Pour cela, on applique la stratégie du tupling aux prédicat maximal et delete qui partagent l'argument Xs.

Ce programme peut être amélioré en utilisant une difference-liste pour le second argument de reverse. Ceci permet d'améliorer le programme parce que la concaténation de difference-listes peut être beaucoup plus efficace que la concaténation de listes.
On introduit pour cela la définition :

```
5. diffrev(X,L-R):-reverse(X,Y),append(Y,R,L).
```

En dépliant diffrev, puis en utilisant sa définition, on trouve une définition récursive de diffrev ne contenant ni reverse ni append.

On déplie reverse :

```
6. diffrev([],L-R) :-append([],R,L).
7. diffrev([H|T],L-R):-reverse(T,V),append(V,[H],Y),append(Y,R,L).
```

On déplie la clause 6, on a :

```
8. diffrev([],R-R).
```

Par pliages et dépliages on peut prouver le remplacement suivant :

```
append(V,[H],Y),append(Y,R,L) ⇔ append(V,[H|R],L)
```

La clause 7 devient alors :

```
9. diffrev([H|T],L-R):-reverse(T,V),append(V,[H|R],L).
```

En pliant la clause 9 avec la clause 5 on obtient :

```
10. diffrev([H|T],L-R):- diffrev (T,L-[H|R]).
```

On définit reverse à l'aide de diffrev :

```
11. reverse(X,Y) :-diffrev(X,Y-[]).
```

Ce qui donne le programme final de complexité O(n) qui utilise les difference-listes:

```
11. reverse(X,Y) :-diffrev(X,Y-[]).
8. diffrev([],R-R).
10. diffrev([H|T],L-R):- diffrev (T,L-[H|R]).
```

2.7. Conclusion sur les transformations de programmes

Nous avons vu sur quelques exemples qu'à partir de quelques transformations de base assez simples, on peut améliorer de façon significative des programmes. Toutefois ces transformations très générales ne sont pas encore automatisables et font appel à ce qu'on appelle des prédicats 'euréka' qui doivent être donnés au programme pour qu'il arrive à effectuer des transformations intéressantes. L'outil est puissant mais encore mal maîtrisé. On peut toutefois l'utiliser dans un cadre plus restreint qui est celui de la spécialisation de programmes proprement dite. Cette restriction permet alors de créer des métaprogrammes de spécialisation entièrement indépendants. C'est ce que nous allons voir avec l'évaluation partielle.

3. L'évaluation partielle

3.1. Introduction

Le but de l'évaluation partielle est de transformer les programmes de façon à ôter les niveaux d'interprétations. Beaucoup de programmes qui ne sont pas des interpréteurs ont tout de même des niveaux d'interprétation, par exemple tous les programme dont le comportement est régit par des données extérieures. Ainsi les programmes suivants peuvent gagner à être évalué partiellement :

Reconnaissance de motifs : un programme général de reconnaissance de motifs peut être spécialisé sur une entrée particulière et reconnaître plus rapidement cette entrée.

Synthèse d'images : un programme de 'Ray tracing' recalcule souvent la façon dont les rayons de lumière traversent une scène donnée depuis plusieurs origines et dans différentes directions. Spécialiser un programme de 'Ray tracing' général sur une scène donnée pour le transforme en un 'ray tracer' spécialisé sur cette scène qui ne fonctionne que pour cette scène, donne un algorithme beaucoup plus rapide. De plus, la spécialisation ne demande que quelque minutes et rend le programme en moyenne 2 fois plus efficace, alors que le programme général peut demander des heures pour calculer une scène.

Bases de données : l'évaluation partielle peut permettre de compiler des requêtes vers des programmes spécialisés de recherche, dont la seule tâche est de répondre à la requête. Une autre application de l'évaluation partielle aux bases de données est la spécialisation d'un programme général de vérification de contraintes d'intégrité sur une base de données particulière.

Réseaux de neurones : le temps d'apprentissage est généralement long. On peut améliorer ce temps d'apprentissage en spécialisant un simulateur général de réseaux de neurones sur une topologie de réseau donnée.

Calculs scientifiques : On peut spécialiser des programmes généraux de calcul scientifique dont on connaît déjà certaines données d'entrée. On peut par exemple spécialiser la FFT sur un nombre de bits donné, ou spécialiser un programme de calcul d'orbites planétaires sur une système planétaire particulier, ou encore spécialiser une simulation générale de circuits électriques sur un circuit particulier.

Une propriété de l'évaluation partielle est qu'elle est automatique, bien que l'on se permette, notamment pour les langages impératifs d'annoter le code pour diriger l'évaluateur partiel.

On définit aussi l'évaluation partielle dans la cadre de la programmation logique :

Evaluation Partielle : Soit P un programme et G un but. L'évaluation partielle produit un nouveau programme P', qui est P-spécialisé pour le but G. L'objectif de cette transformation est que le but G ait la même solution pour les programmes P et P', et que G s'exécute plus rapidement pour P' que pour P.

Lorsque l'évaluation partielle s'applique à des programmes logiques purs (sans cut, sans effets de bords, sans prédicats non-logiques) elle est aussi appelée **déduction partielle**. Nous étudierons principalement la déduction partielle dans le cadre des programmes logiques et l'évaluation partielle dans le cadre des programmes C.

3.2. Un exemple en C

Nous considérons une fonction C qui affiche un texte sous un certain format. Nous appelons cette fonction mini_printf parce qu'elle fait une partie du travail de la fonction printf.

```
void mini_printf(char *chaine,int *valeurs)
{
  int i;
  for (i=0; *chaine != ’\0’; chaine++)
     if(*chaine != ’%’)
       putchar(*chaine);
     else
       switch(*++chaine)
```

```
        {
          case ’d’: putint(valeur[i++]); break;
          case ’%’: putchar(’%’); break;
          default : error();
        }
}
```
On se propose de spécialiser cette fonction sur la chaîne "n = %d".

Avant de spécialiser, on analyse dans le programme ce qui dépend des données sur lesquelles on spécialise. On notera avec des underscore ('_') les instructions, variables et expressions dont on ne connaît pas la valeur lors de la spécialisation dans le programme ci-dessus.

```
void _mini_printf(char *chaine,int *_valeurs)
{
  int i;
  for (i=0; *chaine != ’\0’; chaine++)
     if(*chaine != ’%’)
       _putchar(*chaine);
     else
       switch(*++chaine)
       {
         case ’d’: _putint(_valeur[i++]); break;
         case ’%’: _putchar(’%’); break;
         default : _error();
       }
}
```

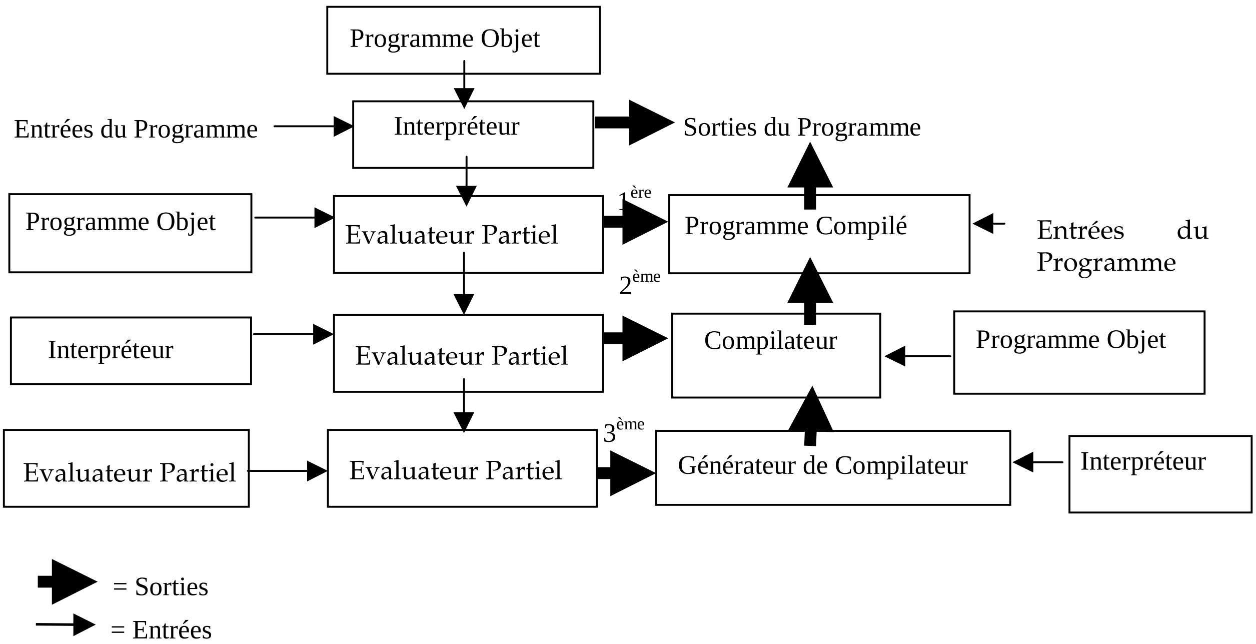
On spécialise alors la fonction mini_printf sur la chaine de caractères "n = %d" :

```
void mini_printf_chaine(int *valeurs)
{
  putchar(’n’);
  putchar(’ ’);
  putchar(’=’);
  putchar(’ ’);
  putchar(value[0]);
}
```

Le programme spécialisé s'éxecute plus rapidement que le programme original qui interprétait la chaîne à chaque appel.

3.3. Les équations de Futamura

Un évaluateur partiel [Consel et al. 93] [Gallagher 93] [Lloyd et al. 91] ou un système de déduction est dit auto-applicable s'il est capable de se spécialiser lui-même efficacement. Les intérêts d'une telle propriété sont multiples. Les plus connus sont les deuxièmes et troisièmes projections de Futamura [Futamura 71]. Les trois projections de Futamura sont données de façon graphique dans la figure suivante :



La première projection de Futamura consiste à spécialiser un interpréteur pour un programme objet particulier, produisant ainsi une version spécialisée de l'interpréteur qui peut être vue comme une compilation du programme objet.

Si l'évaluateur partiel peut s'appliquer à lui-même, on peut alors spécialiser l'évaluateur partiel sur l'interpréteur afin qu'il soit spécialisé dans la première projection de Futamura. On obtient ainsi un compilateur correspondant à l'interpréteur donné en entrée.

La troisième projection de Futamura consiste à spécialiser l'évaluateur partiel pour faire la deuxième projection de Futamura. En spécialisant l'évaluateur partiel sur l'évaluateur partiel, on obtient alors un générateur de compilateur. C'est à dire un programme qui transforme un interpréteur en compilateur.

L'idée principale qui permet l'auto-application est de séparer la spécialisation en deux phases distinctes :

en premier, une analyse du programme à spécialiser qui détermine les valeurs qui pourront être spécialisées, cette phase est appelée Binding-Time Analysis(BTA).
en second, une phase simplifiées de spécialisation, guidée par les résultats de la BTA.

Cette approche de l'évaluation partielle est dite off-line, parce que les décision de contrôle sont prises avant la spécialisation. Elle est mise en contraste par l'approche on-line qui décide au fur et à mesure de la spécialisation ce qui va être spécialisé. Les méthodes de spécialisation on-line sont plus difficiles à auto-appliquer.
Dans le cadre de la programmation logique, le langage Goëdel [Hill et al. 94], [Hill et al. 98] a été développé pour faciliter la métaprogrammation en logique[Barklund 94]. Les trois projections de Futamura ont été effectuées pour le langage Goëdel avec l'évaluateur partiel SAGE [Gurr 93].

3.4. Application au Pattern Matching

Soit le programme suivant qui vérifie si la chaîne de caractères P est présente dans la chaîne S :

```
1. match(P,S):-aux(P,S,P,S).
2. aux([],X,Y,Z).
3. aux([A|Ps],[A|Ss],P,S):-aux(Ps,Ss,P,S).
4. aux([A|Ps],[B|Ss],P,[C|S]):-A=\=B,aux(P,S,P,S).
```

Par exemple, la chaîne [a,b] apparaît dans la chaîne [c,a,b] mais pas dans la chaîne [a,c,b].
Nous allons maintenant évaluer partiellement ce programme sur le but match([a,a,b],X). Pour cela, on commence par introduire une nouvelle définition :

```
5. newp(X):-match([a,a,b],X).
```

On va maintenant déplier ce prédicat newp avec la stratégie de dépliage suivante :

déplier dans les clauses tous les atomes de la forme match(…) ou aux(…).
ne pas déplier une clause dont le corps contient une instance d'un atome qui apparaît dans le corps d'une clause parente dans l'arbre de dépliage.

On obtient l'arbre de dépliage suivant :

```
5. newp(X):-match([a,a,b],X).

6. newp(X):-aux([a,a,b],X,[a,a,b],X).

7. newp([a|S]):-
      aux([a,b],S,[a,a,b],[a|S]).
8. newp([H|S]):-a=\=H,
      aux([a,a,b],S,[a,a,b],S).

9. newp([a,a|S]):-
      aux([b],S,[a,a,b],[a,a|S]).
10. newp([a,H|S]):- a=\=H,
      aux([a,a,b],[H|S],[a,a,b],S).

11. newp([a,a,b|S]):-
      aux([],S,[a,a,b],[a,a,b|S]).
12. newp([a,a,H|S]):- b=\=H,
      aux([a,a,b],[a,H|S],[a,a,b],[a,H|S]).

13. newp([a,a,b|S]).
```

Les atomes des clauses 8, 10 et 12 sont des instances du corps de la clause 6. Le dépliage est donc arrêté.
On définit alors le prédicat newq qui permet de rendre les clauses récursives terminales :

```
14. newq(S):-aux([a,a,b],S,[a,a,b],S).
```

En pliant la clause 6 avec la clause 14, on obtient :

```
6f. newp(X):-newq(X).
```

Quand on déplie le programme sur la clause newq, On obtient le même arbre que précédemment avec les newp remplacés par des newq :

```
13q. newq([a,a,b|S]).
12q. newq([a,a,H|S]):-b=\=H,aux([a,a,b],[a,H|S],[a,a,b],[a,H|S]).
10q. newq([a,H|S]):-a=\=H,aux([a,a,b],[H|S],[a,a,b],[H|S]).
```

```
8q. newq([a,H|S]):-a=\=H,aux([a,a,b],S,[a,a,b],S).
```

Ce qui permet d'obtenir lorsqu'on plie les clauses du programme ainsi obtenu avec la définition de newq.

```
6f. newp(X):-newq(X).
13qf. newq([a,a,b|S]).
12qf. newq([a,a,H|S]):-b=\=H,newq([a,H|S]).
10qf. newq([a,H|S]):-a=\=H, newq ([H|S]).
8qf. newq([a,H|S]):-a=\=H, newq (S).
```

C'est le programme qui est obtenu avec Mixtus [Sahlin 91], l'évaluateur partiel standard pour Prolog.

3.5. Déduction partielle d'un parser

Soit le programme suivant qui parse les mots d'un langage non contextuel :

```
1. parse(Grammar,[Symb],[Symb|X]-X):-terminal(Symb).
2. parse(Grammar,[Symb],Word):-nonterminal(Symb),
                               member(Symb->Symbs,Grammar),
                               parse(Grammar,Symbs,Word).
3. parse(Grammar,[Symb1,Symb2|Symbs],WordX-X):-
                               parse(Grammar,[Symb1],WordX-Y).
                               parse(Grammar,[Symb2],Y-X).
4. terminal(0).
5. terminal(1).
6. nonterminal(s).
7. nonterminal(u).
```

Le premier argument de parse est une grammaire représentée par une liste de productions de la forme Symb→Symbs, où Symb est un symbole non terminal et Symbs est une séquence de symboles terminaux et non terminaux. Le second argument de parse est la chaîne générée par parse représentée par une liste. Le troisième élément est le mot à parser.

On va spécialiser ce programme général sur la grammaire suivante :

```
s→0u
u→1
u→0uu
u→1s
```

On veut donc spécialiser le programme parse sur le but

```
parse([s→[0,u],u→[1],u→[0,u,u],u→[1,s]],[s],X-[]).
```

on notera désormais Γ la liste [s→[0,u],u→[1],u→[0,u,u],u→[1,s]].

En appliquant la stratégie de dépliage définie au paragraphe précédent, On obtient :

```
parse(Γ,[s],[0|Z]-Y):-parse(Γ,[u],Z-Y).
parse(Γ,[u],[1|Y]-Y).
parse(Γ,[u],[0|Z]-Y):-parse(Γ,[u],Z-V),parse(Γ,[u],V-Y).
parse(Γ,[u],[1|Z]-Y):-parse(Γ,[s],Z-Y).
```

La spécialisation du parser général sur une grammaire particulière permet de le rendre bien plus efficace.

4. Spécialisation de programmes C

Il existe actuellement deux évaluateurs partiels pour le langage C : C-Mix a DIKU au Danemark et Tempo à l'IRISA à Rennes. L'évaluation partielle de langages impératifs est aussi décrite dans [Gomard et al. 93]. C-Mix a été appliqué avec succès à la spécialisation d'un ray tracer sur une scène donnée. Il est actuellement en accès libre et peut être utilisé pour un grand nombre de programmes C. Une autre de ses applications a été la spécialisation de fonctions mathématiques sur certaines données d'entrées. Par exemple la spécialisation d'une fonction calculant une FFT sur un nombre d'entrées égal à 512, ou encore la spécialisation du calcul de l'intégrale de Romberg ou le calcul des polynômes de Tchebyshev.

Tempo quant à lui a été appliqué avec succès à la spécialisation de la couche RPC du système Unix ainsi qu'à d'autres types de programmes C.

5. Applications pour l'intelligence artificielle

Le parallèle entre l'évaluation partielle et l'apprentissage par explication a été effectué par Franck Van Harmelen [Van Harmelen et al. 88]. L'apprentissage par explication [Mitchell et al. 86] est une spécialisation dynamique de programme qui tend à rendre les programmes plus efficace lorsqu'elle est utilisée convenablement, on peut par exemple apprendre à atteindre des buts aux échecs [Pitrat 76] ou au jeu de Go [Cazenave 98].

Il existe aussi des liens importants entre spécialisation de programmes, métaprogrammation et métaconnaissances [Pitrat 90]. D'une façon générale, la spécialisation de programmes peut être utilisée par les chercheurs en Intelligence Artificielle pour engendrer des programmes efficaces à partir d'un programme générique et simple.

Il peut y avoir des applications en langage naturel comme le montre l'exemple de la section 3 sur la déduction partielle d'un parser, mais aussi en planification. Ainsi Augustsson utilise l'évaluation partielle pour accélérer la planification des équipages d'avions [Augustsson 97]. Une autre utilisation déjà citée est la spécialisation d'un simulateur de réseaux de neurones sur une architecture de réseau particulière.

On peut aussi noter la similarité des idées sur lesquelles est basée l'évaluation partielle avec des idées qui ont donné des résultats particulièrement bons en Intelligence Artificielle. Ainsi, le programme de K. Thompson de calcul des bases de données de finales d'Echecs, de même que le programme de J. Schaeffer pour les finales de Checkers peuvent être vus comme des évaluations partielles de la définition du gain aux Echecs ou aux Checkers en utilisant les règles du jeu. Ces programmes précalculent des positions au lieu d'avoir à les recalculer à chaque fois, ce qui leur fait gagner du temps.

6. Références

[Augustsson 97] Augustsson L. 1997. *Partial Evaluation in aircraft crew planning*. http://www.carlstedt.se/~augustss.

[Barklund 94] Barklund J. 1994. *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.

[Consel et al. 93] Consel C., Danvy O. 1993. *Tutorial Notes on Partial Evaluation*. Proceedings of the ACM SIGPLAN Symposium on POPL'93. Charleston, 1993.

[Cazenave 98] Cazenave T. 1998. *Metaprogramming Forced Moves*, Proceedings ECAI-98, pp. 645-649. Brighton, 1998.

[Futamura 71] Futamura Y. 1971. *Partial Evaluation of computation process – an approach to a compiler compiler.* Systems, Computers, Controls 2 (5), 45-50.

[Gallagher 93] Gallagher J. 1993. *Specialization of Logic Programs.* Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.

[Gomard et al. 93] Gomard, Jones, Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall 1993

[Gurr 93] Gurr C. 1993. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Dept. of Computer Science, University of Bristol.

[Hill et al. 94] Hill P. M. and Lloyd J. W. 1994. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

[Hill et al. 98] Hill P. M. and Gallagher J. G. 1998. *Meta-Programming in Logic Programming*. Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Oxford University Press, 1998.

[Lloyd et al. 91] Lloyd J. W. and Shepherdson J. C. 1991. *Partial Evaluation in Logic Programming.* J. Logic Programming, 11 :217-242., 1991.

[Mitchell et al. 86] Mitchell, T. M.; Keller, R. M. and Kedar-Kabelli S. T. 1986. Explanation-based Generalization : A unifying view. *Machine Learning* 1 (1), 1986.

[Pitrat 76] Pitrat J. 1976. Realization of a Program Learning to Find Combinations at Chess. Computer Oriented Learning Processes, J. C. Simon editor. NATO Advanced Study Institutes Series. Series E: Applied Science - N° 14. Noordhoff, Leyden, 1976.

[Pitrat 90] Pitrat, J. 1990. *Métaconnaissance - Futur de l'Intelligence Artificielle*. Hermès, Paris.

[Pettorossi et al. 96] Pettorossi A., Proietti M. 1996. *A Comparative Revisitation of Some Program Transformation Techniques*. LNCS 1110, pp. 355-385.

[Pettorossi et al. 98] Pettorossi A., Proietti M. 1998. *Transformation of Logic Programs*. Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Oxford University Press, 1998, pp. 697-787.

[Sahlin 91] Sahlin D. 1991. An Automatic Partial Evaluator for Full Prolog. PhD thesis, Dept. of Telecommunication and Computer Systems, Royal Institute of Technology, Stockholm, Sweden.

[Tamaki et al. 84] Tamaki H. and Sato T. 1984. *Unfold/Fold Transformations of Logic Programs*. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.

[Van Harmelen et al. 88] Van Harmelen F. and Bundy A. 1988. *Explanation based generalisation = partial evaluation*. Artificial Intelligence 36:401-412, 1988.

DOUGLAS HOFSTADTER ET LES CONCEPTS FLUIDES

Jacques PITRAT

pitrat@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé

D. Hofstadter et son équipe ont proposé une approche originale de l'analogie. Après avoir décrit rapidement les quatre systèmes qui ont été implémentés, on étudie plus en détail le système Copycat qui résout une famille de problèmes souvent posés dans les tests psychologiques. Nous verrons ensuite quelques propositions faites pour améliorer le système en passant au niveau méta. Nous ferons enfin un bilan de ces expérimentations.

Mots-clefs : analogie, résolution de problèmes, méta.

1. L'analogie

Depuis les débuts de l'IA, les chercheurs se sont intéressés au problème de l'analogie qui semble essentiel dans un comportement intelligent. Dans l'analogie, on résout un problème différent du problème posé en espérant que cette solution donnera des idées pour trouver la solution du problème initial. On a donc à passer par trois étapes :

- Trouver un problème analogue
- Résoudre ce nouveau problème
- Utiliser cette solution pour résoudre le problème initial.

Les psychologues se sont beaucoup intéressés à l'analogie. Par exemple, ils ont étudié [Thagard et al. 1990] des étudiants auxquels on commençait par raconter l'histoire du général qui devait attaquer une forteresse. Les routes d'accès étaient minées de telle sorte qu'il ne pouvait faire passer qu'une petite troupe par chaque route. Pour attaquer la forteresse, il a divisé son armée en petits groupes convergeant vers la forteresse autour de laquelle son armée s'est finalement trouvée concentrée. Les auteurs de cette étude ont constaté que ceux qui avaient écouté cette histoire avaient plus de chance de résoudre un autre problème, qui est cette fois du domaine médical : un patient a une tumeur et un rayonnement permet de la détruire. Malheureusement, ce rayonnement détruit aussi les tissus sains qui sont entre la source et la tumeur. Il faut là aussi opérer par concentration avec plusieurs sources de moindre intensité convergeant vers la tumeur. Celle-ci recevra ainsi une dose suffisance pour la détruire tandis que les autres tissus recevront une dose insuffisante pour être trop affectés par le rayonnement. Les étudiants ayant lu auparavant l'histoire de la forteresse réussissent plus souvent la tâche médicale que ceux qui ne la connaissent pas. D'autres études psychologiques [Clement 1988] ont montré que les experts utilisent très souvent l'analogie et qu'elle leur permet d'avoir rapidement d'excellentes approximations de la solution des problèmes. De même, [Schoenfeld 1985] a montré l'importance de l'analogie pour le mathématicien.

Bien que de nombreux chercheurs en IA soient convaincus de l'importance de l'analogie, peu d'essais ont été faits sur ce sujet. En effet, il est difficile de l'expérimenter dans un système d'IA, car elle demande de faire appel à des connaissances dans des domaines très variés. Il faut donc donner au système une quantité énorme de connaissances. Celui-ci doit alors déterminer l'analogie qu'il fera, c'est à dire prendre parmi toutes ces connaissances celles qui seront utiles pour le problème à résoudre. Certaines analogies sont classiques comme celle entre l'électricité et l'hydraulique, mais pourquoi pense-t-on à une analogie entre un ressort à boudin et une voiture qui monte le long d'une route autour d'une montagne ? [Hall 1989] donne une bonne synthèse de la plupart des travaux qui avaient été faits à cette date.

Un mécanisme proche de l'analogie, la métaphore, intervient dans la compréhension de textes exprimés dans une langue naturelle. On utilise un rapprochement entre deux mondes différents pour mieux faire passer son message. La métaphore permet d'augmenter le vocabulaire d'un domaine en empruntant des termes à d'autres domaines. Il n'y a peut-être pas plus de dix mots du français qui sont spécifiques aux échecs. Les commentateurs de parties ont largement emprunté à d'autres domaines, en particulier au vocabulaire de la guerre. Mais ils ont aussi puisé dans celui de l'alimentation (capturer=manger), de la mort (perdre ou être capturé=mourir), du théâtre, des sports de combat, etc. Mais un autre intérêt de la métaphore nous rapproche de l'analogie utilisée en résolution de problèmes : elle aide à faire comprendre une situation en la rapprochant d'une autre situation où le lecteur a plus d'expérience. C'est le but de F. Le Lionnais quand il écrit dans son livre sur les prix de beauté aux échecs : «Ce Fou rend à peu près autant de services qu'une lunette astronomique dont on a oublié d'ouvrir l'obturateur.» Comme en résolution de problèmes, l'auteur de la métaphore doit chercher un domaine où le lecteur pourra apprécier la comparaison et l'utiliser pour comprendre la situation exposée par l'auteur. La principale différence est qu'ici la première étape, choix d'un autre domaine, est faite par un individu, l'auteur, alors que les deux autres étapes sont faites par une autre personne, le lecteur. En résolution de problèmes, le résolveur doit traiter les trois étapes.

Depuis de nombreuses années, Douglas Hofstadter et son équipe du Fluid Analogies Research Group [Hofstadter et al. 1995] s'est intéressé à la réalisation de systèmes d'IA capables d'utiliser des analogies pour être créatifs. Le but de ces analogies est la créativité, le système doit trouver des solutions originales et aussi satisfaisantes que possible à des problèmes créatifs. Ces problèmes n'ont pas une solution unique, on peut simplement estimer qu'une solution est meilleure qu'une autre. Il commence par critiquer, avec raison, la plupart des travaux qui ont été faits dans ce domaine. En effet, les quelques systèmes déjà réalisés trouvent bien les analogies que leur créateur veut leur faire trouver, mais il est douteux qu'ils soient capables de trouver des analogies originales et utiles. La démarche d'Hofstadter est tellement différente de ce qui se fait ailleurs que Hall l'a exclu délibérément, quoique semble-t-il un peu à regret, de sa synthèse sur l'analogie. Il l'a d'ailleurs classé à tort parmi les travaux relevant du connexionnisme.

Pour faire leurs expériences, les chercheurs de l'équipe d'Hofstadter ont choisi de prendre des "toy problems", c'est à dire des problèmes dont l'énoncé est très simple et dont la solution ne pose pas de grandes difficultés à un expert humain. Nous commencerons par passer en revue ces problèmes et nous détaillerons la méthode utilisée pour résoudre l'un d'entre eux. Nous verrons ensuite l'intérêt de passer au niveau méta, ce que Hofstadter examine rapidement. Nous ferons enfin le bilan de ces expériences.

2. Les problèmes résolus par le groupe d'Hofstadter

Quatre familles de problèmes ont été expérimentées avec des systèmes créés par des chercheurs travaillant en collaboration avec D. Hofstadter : Jumbo, Numbo, Copycat et Tabletop. Une cinquième famille, Letter Spirit, dont le but est de créer un ensemble de caractères typographiques pour les 26 lettres de l'alphabet, n'a pas encore donné lieu à la réalisation d'un système ; aussi n'en parlerai-je pas.

2.1. Jumbo

Jumbo a été implémenté par D. Hofstadter. Il reçoit un ensemble de lettres dans le désordre et il doit composer un mot anglais utilisant toutes ces lettres. Par exemple, à partir de

T A R F D

il va composer le mot : DRAFT.

A première vue, cela ressemble au Scrabble, mais il y a une différence essentielle : le système n'a pas de dictionnaire. En réalité, Jumbo ne va pas engendrer des mots anglais, mais des mots qui ont l'air d'être des mots anglais. Certains d'entre eux, comme GLINCED ou KNOODLER, n'appartiennent pas à l'anglais et le système n'a aucun moyen de le savoir.

Pour arriver à ce résultat, Jumbo dispose de connaissances sur les lettres qui vont bien ensemble à tel emplacement d'un mot. Par exemple, il affecte un poids de 4 pour le groupe SP en début de mot et un poids de 2 pour le même groupe en fin de mot. De même, le groupe OA a un poids de 2 en début de mot et un poids de 4 en milieu de mot. Il essayera de faire, parmi les lettres données, des regroupements qui satisfont ces préférences. Une lettre mal placée ou non-placée appellera au secours pour que l'on améliore son sort jusqu'à ce que l'on obtienne un maximum local. A partir de

R B H O C A

le système trouve que BR-OA-CH est un bon choix, car BR est bien en début du mot, OA en milieu et CH en fin.

2.2. Numbo

Numbo a été réalisé par D. Defays. Il est basé sur le jeu "Le compte est bon". Il reçoit plusieurs nombres compris entre 1 et 25 et un but à atteindre. Il s'agit d'arriver à créer une suite d'opérations sur les nombres donnés dont le résultat est le but donné. Ces opérations peuvent être l'addition, la soustraction et la multiplication ; on utilise une fois au plus chacun des nombres donnés (mais on peut ne pas les utiliser tous). Il peut y avoir plusieurs solutions. Si les nombres et le but sont choisis au hasard, il est également possible qu'il n'y ait pas de solution. Avec les nombres 11, 20, 7, 1, 6 et un but de 114, il existe deux solutions :

$$114 = (20-1)*6$$

$$114 = (20*6) -7 + 1$$

On a constaté que, pour ce problème, les humains donnent presque toujours la deuxième solution.
Le système dispose de règles heuristiques pour le guider dans sa recherche. Par exemple :

Si le but est grand

Alors il est bon d’essayer la multiplication.

En effet, avec des multiplications on peut arriver plus vite à une valeur proche du but. Dans le cas présent, il est naturel d'essayer des multiplications avec les plus grands nombres, par exemple 11*7 qui donne 77 et 6*20 qui donne 120. On préfère la seconde qui est proche du but. A partir des chiffres restants (11, 7, 1), il reste à trouver 6 que l'on soustraira de 120 pour arriver à 114. Il est facile de voir que 7-1 donne le 6 cherché et l'on trouve la deuxième solution.

2.3. Copycat

Copycat a été réalisé par D. Hofstadter et M. Mitchell. Son but est de résoudre des problèmes analogues à certains tests psychotechniques. On indique au sujet qu'à une première suite de symboles correspond une deuxième suite de symboles. On lui demande quelle suite correspond à une troisième suite de symboles. Les symboles sont ici des lettres de l'alphabet. Un exemple de problème est :

A "abc" correspond "abd".

A quoi correspond "ijk" ?

qui est écrit plus brièvement :

abc -> abd ; ijk -> ?

Le système doit trouver une loi vraie pour le premier couple et l'appliquer au second pour avoir la réponse. Avec l'exemple précédent, une loi possible remplace la troisième lettre par son successeur sans modifier les autres lettres, ce qui donne la réponse "ijl". Dans cet exemple, cette loi est tellement évidente qu'elle surpasse toutes les autres, mais pour d'autres problèmes, plusieurs lois peuvent être des candidats sérieux. Aussi Copycat ne donne-t-il pas toujours la même réponse pour le même problème. Ses auteurs sont satisfaits de son comportement s'il donne souvent les lois les plus fréquemment trouvées par les humains et s'il donne rarement des réponses que nous trouvons tordues. Dans l'exemple précédent, on pourrait avoir comme autre loi : on a la même suite de symboles que dans le résultat du premier couple, ce qui donnerait la réponse "abd". Bien évidemment, cette réponse n'est pas satisfaisante et le comportement de Copycat n'est jugé correct que s'il ne donne cette réponse qu'exceptionnellement. Copycat donne également une estimation de la satisfaction qu'il a d'un résultat par un nombre appelé la "température" du système, prise au moment où il a trouvé cette solution. Cette température est d'autant plus faible qu'il estime le résultat bon. On pardonnera ainsi une mauvaise réponse à Copycat si lui-même l'estime mauvaise.
Prenons un autre exemple un peu plus complexe :

abc -> abd ; xyz -> ?

Les auteurs donnent la statistique suivante alors que ce problème a été proposé 1000 fois :

Résultat	nombre d'obtentions	température
xyd	771	22°
wyz	137	14°
yyz	78	44°
dyz	7	33°
xyy	3	33°
xyz	3	74°
yzz	1	42°

Expliquons quelques unes des lois ainsi trouvées par le système. Notons d'abord que Copycat considère l'alphabet comme une séquence linéaire de lettres : pour lui, il n'existe rien après le 'z' et rien non plus avant le 'a' ; cela exclut donc la réponse que nous aurions tendance à donner : "xya". La réponse "xyd" vient de la loi : la dernière lettre est remplacée par 'd'. Elle est fréquemment obtenue, mais sa température n'est pas la plus basse, ce qui montre que le système ne l'apprécie que modérément. Il préfère par contre la réponse "wyz". Elle est obtenue en remplaçant la loi du premier couple : le successeur de l'élément le plus à droite, par : le prédécesseur de l'élément le plus à gauche. Nous avons là un double "glissement", d'une part de 'prédécesseur' à 'successeur' et d'autre part de 'droite' à 'gauche'. Ce mécanisme de glissement est essentiel pour trouver des analogies. La réponse "yyz" est obtenue par un simple glissement : on passe du successeur de l'élément le plus à droite au successeur de l'élément le plus à gauche. La réponse "xyy" est obtenue aussi par le glissement du successeur de l'élément le plus à droite au prédécesseur de l'élément le plus à droite.

2.4. Tabletop

Le but de ce système est de permettre l'étude des analogies. Pendant les années Thatcher, la question suivante montrait bien les difficultés de l'analogie : Qui est la "first lady" de la Grande Bretagne ? Quatre candidats étaient possibles selon que l'on regardait l'aspect fonction ou l'aspect représentation du président des Etats-Unis, et selon que l'on accordait de l'importance au sexe ou au rôle de l'individu cherché : la reine Elizabeth, le prince Philip, Madame Thatcher et enfin Monsieur Thatcher. Là encore, nous devions avoir un glissement ; plusieurs glissements étaient possibles et les réponses différaient selon celui qui était choisi.

D. Hofstadter propose une autre question qui soulève aussi des réflexions intéressantes : Quel est le East St. Louis de l'Illinois ? Cette question est diabolique, parce que la ville East St. Louis est dans l'Illinois. Aussi une réponse évidente, mais guère satisfaisante, est-elle de répondre : East St. Louis. En réalité, il vaut mieux analyser le problème plus sérieusement et pour cela nous avons besoin de quelques informations sur la géographie des Etats-Unis. East St. Louis est une ville d'environ 50.000 habitants qui est un faubourg pauvre de St. Louis dont elle est séparée par le Mississipi. Mais St. Louis se trouve dans l'état du Missouri, le Mississipi délimitant une des frontières de l'état. Aussi, si nous voulons une bonne analogie, nous devons chercher une situation analogue dans l'Illinois. La plus grande ville du Missouri est St. Louis, la plus grande ville de l'Illinois est Chicago. Essayons de retrouver une ville qui ait les caractéristiques essentielles de East St.Louis, c'est à dire une ville pauvre d'environ 50.000 habitants qui est un faubourg de Chicago et qui se trouve pourtant dans un autre état que Chicago. La réponse est East Chicago qui se trouve dans l'état de l'Indiana et qui est de plus elle aussi séparée de Chicago par une rivière, la Chicago River. Là encore la frontière des états passe entre une ville et un de ses faubourgs. De plus, le nom de chaque ville est formé du préfixe 'East' suivi du nom de la ville principale dont elle est voisine. Nous sommes convaincus de la qualité de l'analogie ainsi trouvée ; nous avons pourtant trouvé pour analogue dans l'Illinois de East St. Louis, elle-même ville de l'Illinois, une ville qui est pour sa part dans l'Indiana. Le nombre de ressemblances permet d'accepter tous ces glissements.

Pour étudier comment un système pourrait faire des analogies analogues dans un domaine moins complexe que la géographie ou la politique, Hofstadter et French ont réalisé le système Tabletop. On suppose que deux personnes sont à table et que des couverts, assiettes, tasses, plats, etc. sont sur cette table. L'un des personnages pointe sur un des éléments et l'autre, représenté par Tabletop, doit désigner un élément analogue. Supposons que, devant un convive, il y ait une tasse entre une cuillère et une fourchette et devant l'autre convive une assiette entre une tasse et un verre. Le premier convive montre la tasse qui est devant lui. Tabletop peut choisir de montrer la même tasse, ce qui revient à dire que le East St. Louis de l'Illinois est East St. Louis. Il peut aussi montrer sa propre tasse, mais pour Tabletop une meilleure réponse est de montrer son assiette. A une tasse entre deux objets de même nature (une fourchette et une cuillère), on fait correspondre un objet situé entre deux objets de même nature (une tasse et un verre). Pour trouver une telle analogie, le système doit travailler à un niveau abstrait qui ne tient plus compte de la nature exacte de chaque objet, mais plutôt de son rôle.

3. Le système Copycat

Le système Copycat est basé sur trois éléments. D'abord, un réseau conceptuel indique les relations entre les divers concepts connus du système ; les liens entre les concepts peuvent changer de valeur, ce qui permettra un glissement éventuel entre deux concepts, comme un passage de "successeur" à "prédécesseur". Ensuite, l'espace de travail est le lieu où l'on construit progressivement des structures qui peuvent se regrouper ; l'une d'entre elles donnera finalement une solution au problème posé. Enfin un ensemble de codelets est un vivier d'acteurs qui examinent ou agissent sur l'espace de travail ; certains d'entre eux peuvent créer de nouveaux codelets.
Je ne donnerai pas une description complète du système qui est complexe, et que l'on peut trouver dans le livre de Hofstadter. Je veux seulement donner une idée des principes sur lesquels ce système est basé.

3.1. Le réseau de concepts

Le système dispose au départ d'une soixantaine de concepts ; il n'a pas la possibilité d'en créer de nouveaux. Certains concepts sont élémentaires, par exemple la lettre 'e' et les 25 autres lettres de l'alphabet. D'autres concepts sont un peu plus profonds et portent sur la relation entre des lettres dans l'alphabet, par exemple la relation "prédécesseur" qui existe entre les lettres 't' et 's'. D'autres concepts de profondeur analogue portent sur des relations entre les lettres d'une suite de symboles, par exemple la fonction "le plus à droite" qui aura pour valeur 'z' si on l'applique à la suite "twkz". Enfin des éléments encore plus profonds portent sur des concepts eux-mêmes, comme la relation "inverse" qui existe entre les concepts "successeur" et "prédécesseur" ou encore entre les concepts "le plus à gauche" et "le plus à droite". De même, le concept "identique" est vrai entre les concepts que sont les lettres 'k' et 'k' ou entre les concepts "successeur" et "successeur".

Ces concepts sont structurés en un réseau où chaque concept peut être relié à un autre concept par un lien donné au départ par le créateur du système. Ce lien a un nom qui est lui-même un concept. Par exemple, le concept "droite" est relié au concept "gauche" par un lien qui a pour nom "inverse". Cette structure ne change pas au cours de la résolution d'un problème, et elle est la même pour chaque problème.

Par contre deux caractéristiques du réseau peuvent changer au cours de la résolution. La première est le degré d'activation d'un concept du réseau. Son activation va augmenter si l'on découvre dans l'espace de travail qu'il est intéressant. Elle va se propager vers les concepts voisins du réseau. Supposons que l'on découvre sur le problème traité que le concept de "successeur " est important (ce qui a des chances d'arriver si l'on traite le problème abc->ghi ; def->?). Le concept "successeur" va être activé et, comme il est relié par un lien "inverse" au concept "prédécesseur", il transmettra son activation à ce dernier concept. L'activation décroît régulièrement, ce qui fait que le réseau est très "élastique" : si un nœud n'est plus stimulé, son activation redescendra à sa valeur de départ. Cela permet au système de repartir dans une nouvelle direction si une voie qu'il estimait prometteuse ne donne finalement rien.

La deuxième caractéristique qui peut changer est la force des liens entre deux concepts. La nature du lien ne peut changer, mais la valeur qui lui est affectée change en fonction de l'activation du concept lié à ce lien. Plus un lien est fort et plus il transmettra l'activation de son origine vers sa terminaison. Supposons que le concept "inverse" ait un degré d'activation élevé ; alors, tous les liens caractérisés par ce concept transmettront facilement l'activation. Si l'on a par exemple une forte activation pour le concept "prédécesseur", elle induira une forte augmentation de l'activation pour le concept "successeur" qui est lié au précédent par justement un lien "inverse". Si par contre le concept "inverse" est peu activé, une forte augmentation de l'activation du concept "prédécesseur" aura très peu de répercussion sur celle du concept "successeur".

Selon l'expression de Hofstadter, il existe un "halo" autour de chaque concept. Si le halo est très fort, le système sera tenté d'essayer un glissement entre deux concepts proches dans ce halo. Par exemple, si l'on a le problème aabc->aabd ; ijkk->? il y a de grandes chances pour que le concept "inverse" soit très activé. En effet, le système découvrira en général l'existence de deux groupes de deux lettres identiques 'aa' et 'kk', l'un situé à droite, l'autre à gauche (première relation "inverse"), l'un étant le prédécesseur et l'autre le successeur des autres lettres (deuxième relation "inverse"). Cela amènera alors à considérer un double glissement entre le successeur de la dernière lettre ('c' qui donne 'd' dans la première partie) vers le prédécesseur de la première lettre qui fera alors passer de 'i' à 'h'. Le système proposera alors la solution "hjkk".

Insistons encore sur le fait qu'il n'y a aucun apprentissage ; il ne s'agit pas d'un réseau connexionniste. Pour chaque nouveau problème, le réseau part de sa valeur par défaut, sans aucune mémoire des changements qui se sont produits lors de la résolution des problèmes précédents.

3.2. L'espace de travail

Diverses tentatives de solution du problème posé sont progressivement construites, puis éventuellement détruites ou modifiées dans l'espace de travail. Au départ, l'espace de travail contient les données du problème. Les acteurs qui agissent dans l'espace de travail sont les codelets. Ils construisent des groupes reliant des éléments de départ, ou des groupes déjà existants, jusqu'à obtenir une structure jugée satisfaisante que l'on n'arrive plus à améliorer. Par exemple, "aabc" peut être considéré comme un groupe de type successeur formé du groupe de type identique "aa" suivi de la lettre 'b' puis de la lettre 'c'. De même, "ijkk" est un groupe de type successeur formé de la lettre 'i', de la lettre 'j' et du groupe de type identique "kk".

On peut aussi créer des correspondances entre des structures déjà construites, et ce sont ces correspondances qui donneront la solution du problème posé. Supposons que l'on doive résoudre : aabc->aabd , ijkk->? Nous venons de voir une structure possible pour les deux premiers éléments de chaque couple. Ces structures ont beaucoup de ressemblances : dans les deux cas, ce sont des groupes successeurs, ils comportent tous les deux un groupe identique de deux lettres. La différence est que dans un cas le groupe identique est au début et dans l'autre à la fin. Si le glissement de "le plus à gauche" vers "le plus à droite" est favorisé par le réseau de concepts, on aura une assez grande probabilité qu'une correspondance soit établie. Dans ce cas, le système arrivera à la solution "hjkk" où l'on remplace le successeur de l'élément le plus à droite par le prédécesseur de l'élément le plus à gauche. Ce double glissement aura d'autant plus de chance de se produire que le concept "inverse" sera activé dans le réseau de concepts. S'il est moins activé, on aura en général un seul glissement, qui donnera "jjkk", où l'on prend le successeur de l'élément le plus à gauche. S'il est très peu activé, le système aura de grandes chances de répondre "ijll", où il estime qu'il faut prendre le successeur du groupe le plus à droite.

L'espace de travail a une température qui estime son niveau d'organisation. Plus il est organisé, et plus la température est basse. Une des fonctions de la température est de mesurer si le système est satisfait de la solution qu'il a trouvée. Il arrive souvent que la solution la plus fréquemment trouvée n'est pas celle qui a la température la plus basse. Nous en avons déjà donné un exemple dans la sous-section précédente. Cela se produit aussi avec le nouvel exemple : sur 1000 essais, on a trouvé 612 fois "ijll" avec une température de 29° et seulement 47 fois "hjkk" , mais avec une température de 19°. Avec "jjkk", la température monte à 47°, mais il est quand même obtenu 121 fois. Nous verrons avec les codelets une autre utilisation de la température.

3.3. Les codelets

Les codelets sont les acteurs qui agissent dans l'espace de travail. Au départ, le système commence avec une population standard de codelets. Un codelet a une priorité ; à chaque étape un codelet est choisi au hasard en tenant compte de sa priorité. Il existe deux groupes de codelets, les acteurs et les explorateurs.

- Les codelets acteurs agissent dans l'espace de travail. Ils construisent de nouvelles structures en créant des groupes ou des correspondances entre groupes. Mais ils peuvent aussi agir en sens contraire en détruisant des structures qu'ils ont déjà créées.

- Les codelets explorateurs se divisent en deux sous-catégories :

  - Les codelets remarqueurs regardent sans idée préconçue ce qui existe dans l'espace de travail. Ce sont eux qui forment la population initiale des codelets. Ils sont la cause de la diversité des réponses que le système trouvera pour le même problème. En effet, la première caractéristique importante qui est découverte peut avoir une conséquence énorme, car elle va conduite à une première structure dans l'espace de travail et l'activation de certains concepts dans le réseau de concepts. Une fois cela établi, le système tendra à favoriser ce qu'il a déjà fait. La direction prise dès le début peut être déterminante à condition qu'elle soit confortée ensuite par d'autres découvertes. Si un problème a deux solutions plausibles, le choix de celle qui sera trouvée dépend très fortement du hasard qui fait que l'on a tiré rapidement un codelet explorateur détectant un élément caractéristique de la première ou de la seconde solution.

  - Les codelets chercheurs ont par contre une mission précise : ils doivent chercher si telle caractéristique est présente, par exemple s'il existe un groupe identique formé de deux lettres en fin d'un groupe. Ils sont créés par les codelets explorateurs qui ont trouvé une caractéristique intéressante et qui cherchent à la confirmer. Ils sont aussi créés par les codelets acteurs qui veulent vérifier une condition nécessaire pour bâtir une structure. Ils peuvent enfin être créés par d'autres codelets chercheurs qui veulent poursuivre une recherche prometteuse dans une certaine direction.

Supposons que nous ayons la suite "mrrjjj". Un codelet explorateur spécialisé dans la recherche de lettres voisines identiques va remarquer qu'il y a deux 'r' consécutifs et il proposera de créer un lien entre eux. Un codelet chercheur va évaluer l'intérêt de créer ce lien en liaison avec les autres liens déjà existants et l'activation des concepts liés à ce lien. Si cet intérêt est jugé suffisant, un codelet acteur va effectivement créer une structure dans l'espace de travail.

Toutes ces opérations ne se produisent pas forcément en séquence, les codelets créés sont mis dans un pool de réserve ; le système choisit au hasard parmi ceux-ci en tenant compte de leur priorité. Cette priorité est calculée au moment de la création du codelet en fonction de l'importance estimée de la tâche qu'il doit accomplir. Mais l'importance de la valeur donnée à une priorité change en fonction de l'état de structuration de l'espace de travail qui est mesuré par sa température. En effet, s'il est peu structuré, il vaut mieux explorer largement en espérant trouver quelque chose de plus intéressant que ce qui a déjà été observé ou construit. Supposons par exemple que le codelet A ait une priorité deux fois plus élevée que celle du codelet B. Cela ne signifie pas que A aura toujours deux fois plus de chances d'être choisi que B. Considérons trois valeurs de la température :

- La température est élevée, l'espace de travail est peu structuré ; le codelet A est à peine plus attractif que B.
- La température est moyenne, il existe déjà des structures prometteuses qu'il faut développer, mais peut-être d'autres pistes se révéleront encore plus intéressantes. A a deux fois plus de chances d'être choisi que B.
- La température est basse, on est près d'avoir une solution satisfaisante ; il ne faut pas que le système disperse son attention. Dans ce cas, il y a une très grande chance que A soit choisi.

Il est impossible de prévoir le fonctionnement d'une telle organisation de codelets pour un essai particulier. Par contre, on peut prévoir l'allure du comportement du système pour un grand nombre d'essais : il découvre effectivement le plus souvent des réponses qui nous paraissent normales et rarement des réponses bizarres. Toutefois, nous avons déjà remarqué qu'il arrive que les réponses les plus intéressantes sont obtenues moins fréquemment que des réponses plus banales.

3.4. Les autres systèmes de l'équipe de Hofstadter

Je ne décrirait pas le fonctionnement des trois autres systèmes réalisés par l'équipe de Hofstadter, mais ils présentent des points communs avec Copycat. D'abord par l'importance du hasard qui permet d'obtenir des résultats différents quand on propose plusieurs fois le même problème. Le hasard n'a pas un rôle mystérieux (Pitrat 1964), c'est la façon la plus simple d'assurer un balayage de l'espace des solutions. On est sûr de faire des essais un peu partout ; plus on fait des essais, et plus la densité de ces essais augmente. En contrepartie de cette simplicité de l'implémentation, on risque de faire des essais trop voisins et de négliger trop longtemps une zone intéressante.
Une autre caractéristique de ces systèmes est l'apparition d'une discontinuité à partir du moment où l'on a construit une sous-structure cohérente ; souvent, on se dirige alors rapidement vers la solution. Par contre, il est également possible de voir disparaître une sous-structure importante parce qu'elle n'est pas renforcée suffisamment par le reste de l'énoncé du problème.

4. Metacat

Metacat n'a pas été réalisé, mais, dans un court chapitre, Hofstadter donne quelques indications sur ce qu'un travail au niveau méta pourrait apporter à Copycat. Ces idées sont excellentes, malheureusement elles ne sont pas faciles à mettre en œuvre ; d'ailleurs, il ne donne guère d'indices sur la façon dont cela pourrait être fait.

La première difficulté pouvant être réglée par un passage au méta est celle des boucles mentales. Le système passe parfois son temps à faire et à refaire les mêmes essais qui ne mèneront à rien. Il ne garde aucun souvenir de ce qu'il vient de faire, il n'a aucune "conscience réflexive" qui lui permettrait de s'observer pendant qu'il fait ses essais. Il faudrait qu'il monitore mieux la recherche de la solution qui est trop laissée au hasard. Pour des problèmes faciles comme ceux posés à Copycat, ce n'est pas trop gênant ; mais pour des problèmes qui ne seraient plus des "toy problems", ce défaut risquerait de devenir rédhibitoire.
Par ailleurs, il serait intéressant que Metacat applique au niveau méta les méthodes qui font le succès de Copycat au niveau de base. Cela signifie qu'il devrait découvrir des régularités dans ses actions pendant la résolution d'un problème. Il peut passer du concept "successeur" au concept "prédécesseur" parce que ces concepts sont les inverses l'un de l'autre, mais il ne fait pas une réflexion sur ce concept de "inverse". Il ne sait pas ce qu'il fait, il se contente de le faire. Il devrait pouvoir déceler non seulement des régularités dans l'énoncé du problème, mais aussi dans la suite des actions qu'il entreprend pour résoudre le problème. Là encore, cela lui permettrait de mieux monitorer la recherche de la solution.

Enfin, le système manque de mémoire épisodique, c'est à dire qu'il ne se souvient pas de la façon dont il a trouvé auparavant des solutions. Il ne devrait pas se contenter de déceler des analogies pour résoudre un problème, il faudrait qu'il puisse aussi voir des analogies entre problèmes. Il pourrait ainsi utiliser les méthodes qu'il a déjà utilisées avec succès dans un problème analogue. Cela conduirait à une forme intéressante d'apprentissage. Si Copycat est capable de trouver des analogies entre des suites de symboles, il est totalement incapable de découvrir des méta-analogies entre la recherche de la solution de deux problèmes d'analogie. Hofstadter va même jusqu'à suggérer de donner au système la possibilité de faire des méta-méta-analogies. Par exemple, il examinerait comment il a trouvé des analogies qui lui ont permis de découvrir une méta-analogie entre deux historiques de recherche pour la solution de deux problèmes de recherche d'analogie. Mais il ne va pas jusqu'à dire comment il tirerait parti de ces méta-méta-analogies.

5. Conclusion

Le travail de l'équipe d'Hofstadter est remarquable. Un de ses intérêts est d'avoir réalisé des systèmes que l'on peut critiquer pour pouvoir faire mieux à l'avenir. C'est pourquoi nous allons voir maintenant un point où je ne suis pas d'accord avec la démarche suivie par Hofstadter et un point sur lequel je pense qu'il n'a pas assez insisté.

5.1. L'utilisation de "toy problems"

Hofstadter insiste sur l'avantage des "toys problems" pour tester des idées d'IA. Mais l'expression "toy problems" est ambiguë. Un premier sens est de considérer qu'il s'agit de problèmes qu'il nous est facile de résoudre, comme les mondes de blocs ou le tic-tac-toe. Pour un deuxième sens, il s'agit de problèmes qui n'ont pas d'application pratique. Dans ce cadre là rentrent les échecs, le go, beaucoup de casse-tête et de façon générale tous les problèmes que l'on appelle "académiques". Mais certains de ces problèmes posent de sérieuses difficultés même à des experts humains.

A mon avis, les problèmes relevant du premier sens ne sont guère intéressants pour l'IA, les méthodes utilisées pour les résoudre se révélant trop souvent inadaptées aux problèmes réels. Par contre, il est utile de créer des systèmes capables de résoudre des "toy problems" du deuxième type. Cela permet en effet d'étudier des points essentiels de l'IA sans devoir en même temps compliquer la tâche pour tenir compte de particularités importantes dans un problème réel, mais sans conséquences sur le point que l'on veut étudier. Cela ne veut pas dire qu'il ne faut jamais s'attaquer à des problèmes réels ; bien au contraire, ils permettent de découvrir des difficultés que l'on n'avait pas pensé inclure dans les problème que l'on fabrique. Mais l'étude de problèmes créés de toute pièce par l'homme a beaucoup d'intérêt pour faire avancer l'IA sur des points fondamentaux, mais très difficiles, comme l'apprentissage, l'utilisation de métaconnaissances et naturellement l'analogie.
Malheureusement, les problèmes présentés ici relèvent davantage de la première catégorie. Bien que je sente l'intérêt des idées présentées, je ne vois pas très bien comment je pourrais les utiliser pour résoudre des problèmes plus complexes.

5.2. Ces méthodes s'appliquent dans d'autres cas que l'analogie créatrice

On peut penser que l'analogie créative est un but en soi quand elle est à la base de la résolution d'un problème, comme ceux posés à Copycat et à Tabletop. Mais les méthodes proposées pourraient être utiles dans tout système de résolution de problème. Hofstadter en est d'ailleurs conscient puisque pour deux des systèmes, Jumbo et Numbo, on cherche la solution d'un problème, sans que l'analogie joue un rôle essentiel. Mais, dans ce livre, Hofstadter met surtout en avant l'analogie, et pas assez l'intérêt beaucoup plus général des méthodes qu'il propose. D'ailleurs, dans son nouveau livre [Hofstadter 1997], il montre bien l'importance du glissement quand nous nous exprimons dans une langue naturelle et surtout quand nous devons traduire un texte d'une langue dans une autre. Il faut en effet découvrir dans la nouvelle langue des expressions analogues à celles qui sont exprimées dans la langue de départ. Près de cent traductions du même sonnet de Marot témoignent de la variété des résultats que l'on peut obtenir à partir du moment où l'on permet des glissements. Les divers glissements que l'on s'autorise de faire conduisent à des résultats fidèles à l'original d'une certaine manière, mais en même temps radicalement différents les uns des autres.

Quand on résout un problème, un grand nombre d'actions et de plans sont en compétition. Des méthodes proches de celles préconisées par Hofstadter semblent intéressantes pour choisir entre eux. Dans son premier chapitre, Hofstadter met bien en évidence l'importance à la fois de découvrir des patterns dans l'activité du mathématicien et de savoir contrôler la recherche de la solution. Il n'a pas appliqué ses idées à des problèmes complexes, mais cela semble faisable. Au début, des codelets explorateurs évalueraient les caractéristiques du problème, et aussi lanceraient des codelets chercheurs pour voir si d'autres caractéristiques, qui peuvent être intéressantes en raison de celles déjà décelées, sont présentes. Ensuite, des actions seraient tentées, avec d'autant plus de persistance qu'elles conduisent à des étapes prometteuses. Mais si elles s'avèrent conduire à une impasse, on pourrait aussi avoir une révolution où l'on repartirait dans une direction complètement différente. Nous sentons que nous utilisons des méthodes de ce type quand nous résolvons un problème, étant entendu que cela ne suffit pas. Il faudrait en particulier développer les idées suggérées dans Metacat pour mieux monitorer la recherche de la solution.

Par ailleurs, l'aspect aléatoire de Copycat pourrait être utilisé dans un système d'apprentissage. Il peut en effet découvrir ainsi plusieurs solutions que l'on peut comparer et surtout trouver des solutions auxquelles l'auteur du système ne pensait pas. Il devrait alors pouvoir remarquer une caractéristique imprévue pour en tirer parti en la comprenant et en la généralisant. Mais en dehors de cet intérêt pour l'apprentissage, l'aléatoire est surtout utile au démarrage d'un problème inhabituel, pour mieux le cerner et avoir la chance de trouver une idée conductrice à partir d'un aspect que l'on aurait ainsi remarqué. Par contre, je ne suis pas convaincu de l'utilité de l'aléatoire quand la solution est avancée ; Hofstadter en limite d'ailleurs l'importance quand la température devient basse, en augmentant l'influence des priorités.

J'ai beaucoup apprécié ce travail d'IA où, pour une fois, des idées nouvelles et prometteuses sont exposées et expérimentées. De plus, son but est l'étude de l'analogie dont la solution est essentielle si nous voulons réaliser des intelligences artificielles. Enfin les méthodes proposées dépassent le cadre de l'analogie créative. En effet, elles peuvent s'appliquer de façon plus générale à la compréhension et à la génération de textes dans une langue naturelle et aussi à la recherche de la solution de n'importe quel problème.

6. Références

[Clement 1988] Clement J., Observed Methods for Generating Analogies in Scientific Problem Solving, *Cognitive Science 12*, 563-586.

[Hall 1989] Hall R., Computational Approaches to Analogical Reasoning: A Comparative Analysis, *Artificial Intelligence 39*, 39-120.

[Hofstadter 1995] Hofstadter D. and the Fluid Analogies Research Group, *Fluid Concepts and Creative Analogies*, Basic Books.

[Hofstadter 1997] Hofstadter D., *Le Ton beau de Marot*, Basic Books.

[Pitrat 1964] Pitrat J. Utilisation des nombres aléatoires dans les problèmes d'Intelligence Artificielle, *Revue Française de Traitement de l'Information 7*, 323-336.

[Schoenfeld 1985] Schoenfeld A., *Mathematical Problem Solving*, Academic Press.

[Thagard 1990] Thagard P., Holyoak K., Nelson G. and Gochfeld D., Analog Retrieval by Constraint Satisfaction, *Artificial Intelligence 46*, 259-310

.

SYSTÈMES GÉNÉRIQUES

Gérard TISSEAU

Gerard.Tisseau@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé

Certains systèmes d'Intelligence Artificielle sont décrits comme étant "génériques" ou "indépendants de tout domaine d'application". Cet article amorce une réflexion sur ces notions, en précisant quel sens on peut leur donner, quelles difficultés elles présentent et quels mécanismes peuvent être utilisés pour les mettre en œuvre.

Mots-clés : systèmes génériques, métamodélisation.

1. Introduction

Deux thèses récentes ([Guin 97] [Pécégo 98]) proposent des systèmes "possédant un caractère générique" ou "indépendants du domaine d'application", ou "applicables à plusieurs domaines". Syrclad [Guin 97] est un système résolveur de problèmes ("l'architecture Syrclad et son application à quatre domaines") et Sygep [Pécégo 98] est un système générateur de problèmes ("SYGEP, un système générateur de problèmes dans des domaines variés"). Dans les deux cas, le système se présente sous la forme d'un noyau indépendant du domaine d'application et de bases de connaissances spécifiques à chaque domaine. La réunion du noyau et d'une base de connaissances constitue un système spécialisé dans un domaine. Par exemple, la réunion du noyau de Syrclad et d'une base de connaissances sur les dénombrements constitue Syrclad-dénombrements, un système résolveur de problèmes de dénombrements.
Dans cet article, nous décrivons les caractéristiques des deux niveaux (généraliste et spécialiste) et les types de communication qui existent entre eux. Nous donnons des exemples de mécanismes de base utilisables et des exemples de types d'architectures possibles. Nous décrivons plus en détail un exemple particulier, l'architecture de Metagen [Revault 96].

2. Les deux types d'agent

L'aspect essentiel de ces systèmes est la séparation en deux niveaux : un niveau générique et un niveau domaine. Chaque niveau peut comporter des connaissances procédurales (un "moteur", un programme, des méthodes) et des connaissances déclaratives (des faits, des règles, des données). Pour parler de ces niveaux, il est pratique de les voir comme des "agents". Un niveau correspond à un type d'agent : "agent généraliste" d'une part, "agent spécialiste d'un domaine" d'autre part. Le problème consiste à étudier les caractéristiques de chacun des deux types d'agents, la répartition des tâches entre les deux et les modalités de leur interaction.

Nous adopterons ce point de vue dans les exemples que nous donnerons, même si les auteurs des travaux cités n'ont pas présenté ou implémenté leurs systèmes de cette façon.

2.1. L'agent généraliste

La généricité se traduit d'abord par le fait qu'il n'y a qu'un seul agent généraliste, censé posséder des compétences générales pour effectuer une ***tâche*** donnée. La tâche de l'agent généraliste de Syrclad est : "résoudre des problèmes", celle de l'agent généraliste de Sygep est : "engendrer des problèmes". Le système formé par l'agent généraliste de Syrclad et l'agent spécialiste des dénombrements permet d'effectuer la tâche spécialisée : "résoudre des problèmes de dénombrement". L'idée de base est qu'il existe des connaissances génériques concernant une tâche donnée, indépendamment du domaine. Par exemple, l'agent généraliste de Syrclad possède l'heuristique suivante : pour résoudre un problème, il est avantageux d'essayer de déterminer sa classe dans une classification des problèmes du domaine. A elle seule, cette idée n'est pas opérationnelle : pour la mettre effectivement en pratique dans un domaine donné, il faut disposer d'une classification des problèmes de ce domaine et de méthodes pour déterminer la position d'un problème dans cette classification. Ces dernières connaissances sont du ressort d'un agent spécialiste. Mais cela n'empêche pas l'agent généraliste de posséder des concepts et des méthodes concernant l'exécution de la tâche.

Par exemple, l'agent généraliste de Syrclad sait que dans la résolution d'un problème par classification, on part d'une certaine formulation du problème, appelé "modèle descriptif", qu'on applique ensuite une expertise dite de "reformulation" pour engendrer une autre formulation du problème appelée "modèle opérationnel". Au cours de ce processus, il sait qu'il doit déterminer les "attributs discriminants" du problème en utilisant des "connaissances de reformulation". L'ordre dans lequel il doit déterminer ces attributs est défini par un "graphe de classification" des problèmes du domaine. Il sait également que le modèle opérationnel possède une certaine "classe" à laquelle est associée une "méthode de résolution" applicable au problème. Tous les concepts précédents (cités entre guillemets) ont un sens au niveau de l'agent généraliste, mais ce sens reste abstrait, indépendant du domaine. L'agent généraliste sait par exemple ce qu'est un "graphe de classification" (comment il est structuré, comment on peut le parcourir), mais il ne connaît à l'avance aucun graphe de classification particulier.

Pour parler en termes de conception orientée objet, l'agent généraliste connaît des *classes* (la classe GrapheDeClassification pour Syrclad par exemple), mais pas les instances de ces classes. De même, il connaît des ***rôles** :* l'agent générique de Syrclad sait que dans la résolution on n'aura besoin que d'une seule instance de la classe GrapheDeClassification, et que cette instance joue le rôle de "l'unique graphe de classification des problèmes du domaine". Mais il ne sait pas à l'avance quels objets particuliers vont remplir ces rôles.

L'agent généraliste connaît également une méthode abstraite (qu'on appellera une ***stratégie***) pour effectuer la tâche demandée : Syrclad effectue sa tâche de résolution de problèmes par la méthode de "reformulation-classification" et Sygep effectue sa tâche de génération de problème par la méthode du "chaînage arrière". Cette méthode peut nécessiter l'exécution de ***sous-tâches*** pour lesquelles l'agent généraliste ne possède pas de méthode. Il délègue alors l'exécution de ces sous-tâches à l'agent spécialiste du domaine considéré. Par exemple, dans la méthode de résolution de l'agent généraliste de Syrclad figure la sous-tâche "calculer la valeur d'un attribut discriminant". Ce calcul étant très lié au domaine d'application, il n'est pas effectué par l'agent généraliste, mais délégué à l'agent spécialiste.

Par l'intermédiaire des rôles et des sous-tâches, l'agent généraliste définit des conventions d'interface avec les agents spécialistes : ceux-ci doivent fournir des valeurs pour les rôles et des méthodes pour les sous-tâches. Cela se traduit concrètement par une syntaxe de communication, des contraintes sur les valeurs attendues et plus généralement par un contrat sur les droits et les devoirs de chacun des agents. Ces conventions sont importantes car les agents spécialistes sont prévus pour être implémentés par des experts des différents domaines, indépendamment du fonctionnement interne de l'agent généraliste.

2.2. L'agent spécialiste

Un agent spécialiste d'un domaine donné possède des connaissances sur ce domaine. Il définit d'abord des concepts et un vocabulaire propres au domaine, qu'on désigne souvent sous le nom d'***ontologie*** du domaine, ce qui se traduit par exemple par des définitions de classes, de relations et de constantes. Il n'y a *a priori* pas de restriction sur l'ontologie du domaine. Chaque domaine définit les concepts qui lui sont nécessaires.
Ensuite il définit des ***méthodes*** pour les sous-tâches que l'agent généraliste ne sait pas effectuer. Ces méthodes font appel à des connaissances spécifiques au domaine (sinon elles auraient été placées dans l'agent généraliste).
Et enfin il affecte aux rôles prédéfinis dans l'agent généraliste des objets particuliers. Il autorise ainsi l'agent généraliste à examiner et à utiliser ces données propres au domaine.

Un système opérationnel dans un domaine donné est formé d'un agent généraliste et d'un agent spécialiste qui collaborent. Un tel système est destiné à être offert comme outil à un utilisateur pour effectuer une tâche particulière, instance de la tâche générique définie dans l'agent généraliste. Par exemple, un système Syrclad opérationnel en dénombrements permet à un utilisateur d'entrer un problème particulier de dénombrements pour que le système trouve la solution. Les conventions de communication avec l'utilisateur dépendent du domaine, car les données qu'il introduit s'expriment dans l'ontologie du domaine. Ces conventions sont donc définies dans l'agent spécialiste. Pour Syrclad-dénombrements, par exemple, l'agent spécialiste définit une syntaxe et une sémantique pour les problèmes entrés par l'utilisateur sous forme de "modèle descriptif". L'utilisateur doit connaître et respecter ces conventions.

3. Collaboration entre les agents

Examinons trois grand types de collaboration entre l'agent généraliste et un agent spécialiste : la délégation, le paramétrage et le parasitage. Dans cette section, nous décrivons ces mécanismes au niveau des principes, sous une forme imagée. Nous décrirons dans la suite des exemples de mécanismes permettant de réaliser ces types de collaboration.

3.1. Délégation

Pour résoudre la tâche principale, l'agent généraliste exécute une méthode qui peut exiger la résolution de sous-tâches qu'on ne peut traiter qu'avec des connaissances du domaine. Il appelle alors l'agent spécialiste et lui délègue l'exécution de la sous-tâche. Le spécialiste doit posséder une méthode pour exécuter cette sous-tâche.
Dans ce mécanisme de collaboration, l'agent généraliste abandonne toute responsabilité concernant l'exécution de la sous-tâche. Pour lui, l'agent spécialiste joue le rôle d'une boîte noire. La sous-tâche n'est pas résolue en commun par les deux agents, mais seulement par l'agent spécialiste. Cette façon de faire possède l'avantage de la modularité, mais supprime la possibilité qu'aurait l'agent généraliste d'intervenir pendant l'exécution de la sous-tâche. S'il pouvait observer le spécialiste en train d'effectuer la sous-tâche, il pourrait par exemple détecter une situation particulière qui lui permettrait de reconnaître que la tâche principale est en fait terminée et qu'il n'est plus nécessaire de poursuivre la sous-tâche.

Poussée à l'extrême, la délégation peut ôter tout intérêt à l'idée de généricité. Par exemple, on peut imaginer un agent généraliste dont la méthode d'exécution de la tâche principale soit :

Demander à l'utilisateur le domaine D
si D = D1 alors envoyer le message Résoudre au spécialiste de D1
si D = D2 alors envoyer le message Résoudre au spécialiste de D2
si D = D3 alors envoyer le message Résoudre au spécialiste de D3
etc.

Ici toute la tâche est déléguée au spécialiste, et la généricité est juste un aiguillage suivant le domaine. De plus cet aiguillage est prévu statiquement pour un ensemble prédéfini de spécialistes.

Les inconvénients potentiels de la délégation viennent de son aspect procédural, qui l'assimile à un appel de procédure.

3.2. Paramétrage

Une façon de faire en sorte que l'agent généraliste garde l'initiative tout en profitant des connaissances de l'agent spécialiste est d'introduire dans la méthode générique d'exécution de la tâche des *paramètres* dont la valeur est demandée à l'agent spécialiste. Ces paramètres peuvent être des données, des faits ou même des règles. L'avantage est maintenant que la communication est déclarative et que l'agent générique peut raisonner sur les paramètres en question, et adapter son comportement en conséquence.

Cette adaptation peut être dynamique ou statique. Dans une adaptation dynamique, les données utiles sont demandées par nécessité lors du déroulement de la méthode générique sur un problème particulier. Dans une adaptation statique, l'agent généraliste commence par examiner les paramètres de l'agent spécialiste avant toute exécution, pour en extraire des règles qu'il pourra appliquer lors des différentes résolutions. Cette fonction de génération de règles est ainsi assimilable à une compilation et elle nécessite des métaconnaissances.

3.3. Parasitage

Un agent spécialiste peut "parasiter" un agent généraliste en profitant de certaines actions du généraliste pour lancer des procédures dont le généraliste n'a pas "conscience". Cela signifie que le spécialiste détecte des événements se produisant dans le processus du généraliste et qu'il réagit à ces événements par le lancement de procédures. Le généraliste ne sait pas a priori quelles procédures seront lancées ni ce qu'elles feront et n'utilise aucun résultat fourni par ces procédures.
Le parasitage est envisagé dans ce sens (après tout, il pourrait aussi être dans l'autre sens : le généraliste pourrait parasiter le spécialiste) parce qu'il est alors possible de définir une fois pour toutes les événements susceptibles de servir de déclencheurs. Ces événements sont définis au niveau générique, et chaque spécialiste en a connaissance.

4. Quelques mécanismes de base

En informatique et en IA, plusieurs mécanismes de base permettent de définir une fois pour toutes un agent (ou module, ou niveau) ayant un comportement générique et d'introduire par la suite d'autres agents capables d'exploiter ce comportement, de le modifier ou de le spécialiser.

4.1. Classes abstraites et méthodes virtuelles

Dans la programmation orientée objet, une possibilité de généricité est offerte avec les classes abstraites et les méthodes virtuelles. Une classe est dite abstraite s'il est interdit d'en créer des instances (les classes “concrètes” sont des sous-classes de la classe abstraite). Une méthode est virtuelle si elle est déclarée dans une classe mais pas programmée dans cette classe ni dans ses superclasses. Généralement les deux concepts vont de pair : une méthode virtuelle est déclarée dans une classe abstraite. L'utilisation habituelle de ce couple de concepts est la suivante : dans la classe abstraite ***C***, on définit une méthode virtuelle ***v*** et on programme une méthode d'exploitation ***m*** qui utilise la méthode ***v*** (c'est-à-dire qui envoie le message ***v*** à l'objet récepteur). Lors de l'exécution de la méthode ***m***, l'objet récepteur ne sera pas de la classe ***C*** (puisqu'elle est abstraite) mais d'une sous classe ***S*** dans laquelle existe une implémentation de la méthode ***v***. La méthode ***m*** est donc générique : elle est écrite pour pouvoir s'appliquer à toutes les sous-classes de ***C*** (actuelles et surtout futures, c'est cela qui en fait l'intérêt), sans savoir à l'avance comment sera implémentée la méthode ***v*** dans ces différentes sous-classes. La classe ***C*** se place donc au niveau du généraliste et la sous-classe ***S*** au niveau du spécialiste. La classe ***C*** possède une méthode ***m*** pour effectuer une tâche générique, et chaque sous-classe possède une méthode ***v*** pour effectuer une sous-tâche de ***m.***

On pourrait implémenter Syrclad et Sygep de cette façon. Pour Syrclad, il existerait par exemple une classe abstraite `ResolveurGenerique` qui implémenterait une méthode générique `resoudre: unModeleDescriptif` permettant d'effectuer la tâche principale de Syrclad par le procédé de “reformulation-classement”. Cette tâche principale contient une sous-tâche `employerResolution: uneClasseDeProbleme`, qui est appelée lorsque la classification du problème est terminée et qu'on cherche à employer une méthode de résolution adaptée à la classe. Elle serait déclarée comme méthode virtuelle dans `ResolveurGenerique` et elle serait implémentée dans chaque sous-classe `ResolveurDenombrements`, `ResolveurThermodynamique`, etc.

4.2. Fonctions du deuxième ordre

Une fonction du deuxième ordre est une fonction acceptant comme argument une autre fonction et pouvant appliquer cette fonction dans sa définition. Un exemple classique de fonction du deuxième ordre est celle qui admet deux arguments, une fonction ***f*** (à un argument) et une liste ***l***, et qui construit la liste formée des images des éléments de ***l*** par ***f***. Appelons-la `distribuer`. Alors l'appel `distribuer(carre, [1,2,3,4])` fournira le résultat `[1,4,9,16]`, liste des carrés des valeurs de la liste initiale. Un autre exemple est la fonction `accumuler`, qui admet deux arguments, une opération (binaire) et une liste, et qui applique l'opération au deux premiers éléments de la liste, puis au résultat et au troisième élément, etc. L'appel `accumuler(addition, [1,2,3,4])` fournira comme résultat ((1 + 2) + 3) + 4, soit 10.

La possibilité d'introduire des fonctions du deuxième ordre permet de séparer un calcul en une partie générique (fonctions du deuxième ordre) et une partie spécialisée (les fonctions passées en arguments).
Considérons par exemple la fonction `calcul`, à un argument `liste`, définie par :

```
calcul(liste) = accumuler(accumulateur, distribuer(transformateur, liste)).
```

`accumulateur` et `transformateur` sont des variables globales qui n'ont pas encore reçu de valeur au moment de cette définition. On supposera que le langage utilisé permet ce genre de définition. Cette fonction est stockée dans un module G, considéré comme module généraliste.

Un spécialiste d'un domaine n°1 peut alors affecter des valeurs aux variables globales, par exemple :

```
accumulateur := addition
transformateur := carre
```

Il stocke ces valeurs dans un module S1 contenant également G.

Un utilisateur travaillant dans le domaine n°1 peut ensuite utiliser le module S1 :
`calcul([1, 2, 3, 4])` fournira $((1^2 + 2^2) + 3^2) + 4^2$, soit 30.

Mais un spécialiste d'un autre domaine n°2 pourrait affecter d'autres valeurs :

```
accumulateur := maximum
transformateur := identite
```

et les stocker dans un autre module S2 contenant également G.

Un utilisateur travaillant dans le domaine n°2 peut ensuite utiliser le module S2 :
`calcul([1, 2, 3, 4])` fournira 4.

Cette séparation des tâches permet de factoriser les connaissances communes au niveau générique : la structure du calcul est définie dans le module généraliste G. Les différents spécialistes exploitent cette même structure prédéfinie. De plus, il n'est même pas nécessaire qu'ils sachent exactement quelle est cette structure. Il leur suffit de savoir qu'ils devront fournir deux paramètres, `accumulateur` et `transformateur`, et de connaître leurs rôles. Ils peuvent ainsi s'exprimer de manière plus déclarative.

4.3. Hooks et démons

Dans certains environnements, on appelle "*hook*" (crochet, hameçon) une variable où on peut stocker une fonction qui sera appelée à certains moments bien définis par un programme existant. L'intention est de permettre à un utilisateur de personnaliser ("customize") un programme existant en ajoutant des fonctionnalités, en modifiant des réglages ou parfois même en modifiant totalement le comportement du programme de base. L'idée est que le programme de base fournit une architecture et une structure de contrôle tout prêts avec des comportements par défaut raisonnables. En général, le code ajouté dans les "hooks" ne modifie pas l'architecture ni la structure de contrôle, il greffe simplement des comportements locaux qui seront activés "au bon moment". Le programmeur fait confiance au programme de base pour reconnaître ce moment et pour appeler le code greffé.

L'éditeur de texte Emacs, programmé en Lisp, offre de nombreux hooks (près d'une centaine), comme par exemple before-init-hook, after-init-hook, text-mode-hook et edit-picture-hook [GNU Emacs 98].

Dans le langage de programmation CommonLisp [Steele 90], les fonctions d'évaluation de base (eval et apply) contiennent des appels à des fonctions auxiliaires evalhook et applyhook qui se trouvent dans les variables *evalhook* et *applyhook*. Dans l'implémentation par défaut, ces variables ne contiennent rien et l'évaluation se déroule d'une manière prédéfinie. Le programmeur peut décider d'affecter à ces variables des fonctions, auquel cas l'évaluation ordinaire est remplacée par un appel à ces fonctions auxiliaires. Cette possibilité permet de redéfinir complètement le comportement de l'interpréteur Lisp, et finalement d'implémenter de nouveaux langages. Dans l'environnement de programmation, elle est utilisée pour la mise au point, par exemple pour l'évaluation pas à pas et l'affichage d'une trace. Ces variables sont appelées des "crochets" (hooks) pour suggérer l'idée qu'il s'agit d'emplacements prévus pour que le programmeur puisse y accrocher son code. Cette possibilité est dynamique, car on peut changer la valeur des variables *evalhook* et *applyhook* au cours de l'exécution. Certaines précautions doivent être prises dans l'implémentation car le code de la fonction evalhook du programmeur, lui, est évalué avec l'évaluateur prédéfini (à moins de spécifier qu'on veut évaluer certaines parties avec evalhook).

La notion de *démon* (ou de *réflexe*) a été introduite en IA à propos des *frames* et a été reprise entre autres dans le domaine des bases de données et dans celui des interfaces graphiques. Un démon est souvent décrit comme une entité autonome qui surveille un processus et qui réagit lorsque certains événements se produisent. Classiquement, les événements en question sont du type : "tel attribut de tel objet vient d'être modifié", "telle procédure vient d'être appelée mais son exécution n'a pas encore commencé" ou "l'exécution de telle procédure vient de se terminer". Cette description imagée d'un démon recouvre en fait un mécanisme de type hook : dans le code du processus figurent à des endroits prédéfinis des appels à des fonctions auxiliaires que le programmeur peut définir comme il veut. L'idée est que le programmeur n'effectue jamais d'appels explicites à ces fonctions, il se contente de les définir.

4.4. Callbacks et abonnements

Une notion voisine de celle de hook est celle de "*callback*" utilisée dans la programmation par composants. En anglais, *to call back* signifie rappeler (quelqu'un) par téléphone. L'idée est de faire coopérer un agent qui détecte des événements et un autre

qui a demandé à être prévenu de l'apparition de ces événements. La mise en coopération des deux agents peut se faire sous forme de *contrat d'abonnement*. Un client peut demander à un agent détecteur : “chaque fois que tu détecteras un événement de type T, tu me préviendras en m'envoyant le message m”. La différence avec un hook vient de ce que le comportement greffé est vu comme un envoi de message à un destinataire particulier plutôt que comme un simple appel de procédure.
L'intérêt de ce type d'abonnement vient de ce qu'on peut programmer un agent détecteur sans savoir à l'avance quels seront ses clients ni comment ils réagiront aux événements. Le détecteur n'attend pas de réponse de la part des clients et il ne doit rien supposer concernant les effets éventuels des réactions des clients. Son seul travail est de détecter des événements, d'enregistrer les contrats d'abonnement et de servir les clients abonnés lors de chaque événement. Les abonnements peuvent se faire et se défaire de manière dynamique lors de l'exécution.

Ce mécanisme peut être utile pour réaliser un système générique : l'agent généraliste peut servir de détecteur d'événements, et chaque agent spécialiste peut réagir aux événements d'une manière appropriée au domaine. Remarquons que les mécanismes de hook et de callback reposent sur la possibilité de stocker une fonction dans une variable et peuvent être considérés pour cette raison comme des mécanismes d'ordre 2. La fonction est transmise et stockée comme une donnée passive, mais elle peut être appelée comme un objet actif.

5. Quelques exemples d'architecture

Les mécanismes précédents sont des outils de base. Certaines architectures ont été proposées pour réaliser des systèmes génériques en utilisant ces outils et éventuellement d'autres.

5.1. Architectures génériques (frameworks)

Nous ne donnerons ici que quelques grandes lignes. Pour une revue plus détaillée, voir par exemple [Revault 96] pages 13 à 40.
Un “framework”, au sens utilisé ici, est une sorte de squelette de programme comportant un choix d'architecture et de concepts (des modules, des classes et leurs relations) et un schéma de contrôle d'exécution spécifiant qui est responsable de quelle tâche, qui appelle qui et quand. Pour obtenir une système opérationnel dans un domaine donné, un programmeur doit “adapter” le framework. Cela revient en général à fournir des définitions de classes concrètes sous-classes de classes abstraites, à programmer dans ces sous-classes les méthodes qui n'étaient que virtuelles dans les classes abstraites et à créer, initialiser et interconnecter des instances des classes concrètes. Le programmeur fait confiance au schéma d'exécution du framework pour appeler au bon moment les procédures qu'il fournit. Il se repose également sur le fait que les concepts et l'architecture proposés forment un tout cohérent et suffisent pour réaliser une application complète et correcte. Il économise ainsi la plus grande partie du travail de conception d'une application.
Remarquons que les procédures du programmeur ont un rôle de serveur et non pas de client : elles n'appellent pas le framework, mais c'est le framework qui les appelle. C'est l'opposé de ce qui se passe dans une programmation utilisant les primitives d'une boîte à outils. Dans ce dernier cas, le programmeur définit son architecture et son schéma d'exécution, et appelle explicitement les primitives. Le framework joue donc bien le rôle d'agent généraliste.

5.2. Métamodélisation

Dans les approches précédentes, on considère comme une règle de bonne programmation le fait que les deux agents ou niveaux soient conçus indépendamment l'un de l'autre, pour permettre de connecter un composant quelconque au niveau générique. Mais cette règle se traduit bien souvent par l'exigence que le niveau généraliste ne doit rien connaître du niveau spécialiste et ce n'est pas toujours une bonne chose. Il est bon effectivement de ne pas inscrire dans l'agent généraliste des connaissances sur les agents spécialistes car ce serait sinon perdre le bénéfice de la factorisation et de l'abstraction. Mais cela n'oblige pas l'agent généraliste à ignorer à quel agent spécialiste particulier il est connecté dans un système spécialisé lors du fonctionnement de ce système. Ce qui est bon lors de la conception des classes n'est pas forcément bon lors du fonctionnement des instances. On peut très bien imaginer que l'agent généraliste (l'instance) examine l'agent spécialiste (l'instance) auquel il est connecté lors d'une exécution particulière et qu'il retire de cet examen des connaissances qu'il puisse exploiter dans sa tâche. Cela sous-entend que l'agent spécialiste soit examinable, c'est-à-dire qu'il rende explicites ses connaissances et son fonctionnement. Pratiquement, ce n'est possible que si ces connaissances et ce fonctionnement sont spécifiées de manière déclarative. L'agent spécialiste doit ainsi contenir une auto-description déclarative exploitable par l'agent généraliste.

La question se pose alors du formalisme dans lequel cette spécification peut être donnée. Souvent, un spécialiste introduit des notations et une syntaxe propre à son domaine. On ne peut pas prévoir dans la conception de l'agent généraliste tous les formalismes de tous les domaines. La solution est alors de représenter explicitement le formalisme utilisé par le spécialiste grâce à un métaformalisme, suffisamment général celui-là pour qu'il puisse être connu du généraliste et partagé par tous les spécialistes.

Le spécialiste contiendrait ainsi deux niveaux d'auto-description : une description de son contenu, exprimée dans un formalisme spécialisé F adapté au domaine, et une description du formalisme F exprimée dans un métaformalisme M. Cette représentation à deux niveaux est souvent qualifiée de *métamodélisation* : la modélisation consiste à spécifier le contenu et la métamodélisation consiste à spécifier le formalisme qui a servi à spécifier le contenu.

L'idée de métamodélisation (ou plus généralement d'utilisation de plusieurs niveaux méta) a été utilisée dans plusieurs disciplines, comme par exemple l'intelligence artificielle [Pitrat 90] [Kornman 93] [Parchemal 88], le génie logiciel, les bases de données (voir une revue dans [Revault 96] pages 43 à 59). Le développement d'Internet et du World Wide Web concourt à la popularisation de cette idée : les moteurs de recherche et les agents intelligents doivent pouvoir examiner des pages Web qui traitent de domaines très divers dans des formalismes très divers. Pour pouvoir mieux analyser et indexer ces pages, il serait souhaitable qu'elles contiennent une auto-description de leur contenu, ainsi qu'une auto-description du formalisme dans lequel ce contenu est exprimé. Dans la version la plus simple, l'auto-description du contenu peut se faire par des mots-clés (ou par le texte lui-même) et l'auto-description du formalisme peut se faire par des méta-données indiquant par exemple en quelle langue est écrit le texte et dans quel thesaurus sont choisis les mots-clés.

6. Un exemple de métamodélisation : Metagen

Nous donnons ici un exemple d'architecture utilisant la métamodélisation, directement inspirée du système Metagen [Revault 96]. Cette architecture étant complexe, avec plusieurs niveaux méta, nous l'introduirons "par le bas" en ajoutant progressivement des niveaux méta. Le lecteur souhaitant avoir tout de suite une vision d'ensemble peut consulter directement la présentation descendante (6.2).

6.1. Présentation ascendante

Imaginons qu'un utilisateur souhaite disposer d'un système pour résoudre des problèmes de thermodynamique. Une démarche possible est de demander à un informaticien-thermodynamicien d'implémenter un tel système, qui comporterait une interface InterfaceThermo permettant à l'utilisateur d'introduire des problèmes et un résolveur ResolveurThermo capable de résoudre les problèmes. Les problèmes introduits seraient exprimés dans le formalisme FormalismeThermo.
De même, si un utilisateur voulait résoudre des problèmes de mécanique, on créerait un ResolveurMeca et une InterfaceMeca utilisant un formalisme FormalismeMeca. Cela suggère un schéma général : pour offrir à un utilisateur d'un domaine D un système de résolution de problèmes, il faut définir Formalisme(D), Interface(D) et Resolveur(D). Les mots "formalisme", "interface" et "résolveur" font référence à des concepts génériques qu'on spécialise pour chaque domaine D. On a donc l'architecture suivante :

Interface(D)
 entrées : interaction avec un utilisateur du domaine D
 sortie : un problème du domaine D, exprimé dans Formalisme(D)

Resolveur(D)
 entrée : un problème du domaine D, provenant de la sortie de Interface(D)
 sortie : la solution du problème

Dans la réalisation de l'interface et du résolveur, deux types de compétences entrent en jeu : des compétences informatiques (quelles sont les techniques utilisables pour programmer des résolveurs et des interfaces) et des compétences du domaine (qu'est-ce qu'un problème du domaine et quelles sont les connaissances qui permettent de résoudre des problèmes du domaine). Il serait alors souhaitable de séparer explicitement ces compétences sous forme de bases de connaissances : l'une, ConnaissancesDomaine(D), contiendrait les connaissances du domaine exprimées dans le formalisme Formalisme(D) et l'autre, ConnaissancesProgrammation, contiendrait les connaissances de programmation nécessaires à la réalisation d'une interface et d'un résolveur.
Au lieu de demander à un informaticien-intégrateur-ingénieur-de-la-connaissance de programmer à la main une interface et un résolveur à partir des deux bases de connaissances, on peut lui demander de réaliser un *générateur d'application* qui effectue automatiquement cette tâche. Ce générateur a également besoin de recevoir en entrée une spécification de Formalisme(D) pour pouvoir lire et analyser ConnaissancesDomaine(D).
Nous disposons alors d'un nouvel élément de l'architecture :

GenerateurApplication
entrées : Formalisme(D), ConnaissancesDomaine(D), ConnaissancesProgrammation
sorties : Interface(D), Resolveur(D)

Pour utiliser une telle architecture, il est nécessaire que quelqu'un fournisse des connaissances liées au domaine D : Formalisme(D) et ConnaissancesDomaine(D). ConnaissancesDomaine(D) est du ressort d'un expert du domaine D. Formalisme(D) est plutôt du ressort d'un *méta-expert* du domaine D, c'est-à-dire d'un spécialiste du *formalisme* utilisé dans le domaine D, par opposition à un spécialiste du *contenu* du domaine.
Le codage d'une base de connaissances dans un formalisme donné n'est pas une tâche facile, et il serait souhaitable qu'il soit assisté par un système d'aide à l'édition. Ce système doit permettre de soulager la mémoire de l'utilisateur pour ce qui est de la syntaxe du formalisme et d'effectuer des contrôles de cohérence sémantique. On est ainsi amené à introduire un nouveau programme EditeurConnaissances(Formalisme(D)) permettant à un expert du domaine D d'écrire une base de connaissances dans le formalisme Formalisme(D) :

EditeurConnaissances(Formalisme(D))
entrées : interaction avec un expert du domaine D
sortie : ConnaissancesDomaine(D)

La sortie ConnaissancesDomaine(D) de cet éditeur sera reliée à l'entrée de même nom de GenerateurApplication.

Mais le concept d'éditeur est un concept générique : c'est un système qui permet de produire une représentation exprimée dans un formalisme F. On peut voir l'éditeur précédent sous la forme plus générale suivante, qui ne parle plus du contenu mais seulement du formalisme :

Editeur(F)
entrées : interaction avec un écrivain qui s'exprime dans le formalisme F
sortie : une représentation exprimée dans le formalisme F

On peut alors envisager d'engendrer automatiquement cet éditeur à partir d'un *générateur d'éditeur* qui utiliserait une description du formalisme F. Pour décrire un formalisme, on a besoin d'un métaformalisme. Fixons-en un une fois pour toutes et appelons-le MetaFormalisme (dans Metagen, il s'agit d'un formalisme de graphe appelé IR3, variante du formalisme entité-association). On introduit ainsi un nouveau composant dans l'architecture :

GenerateurEditeur
entrée : la description d'un formalisme F, écrite dans le MetaFormalisme
sortie : un éditeur permettant d'introduire des représentations écrites en F

Dans l'architecture générale, cette dernière sortie produira le composant que nous avons appelé EditeurConnaissances(Formalisme(D)) si on applique GenerateurEditeur avec comme formalisme F le Formalisme(D) propre au domaine D. En fait, dans le système, le générateur d'éditeur demande également comme entrée des paramètres de présentation graphique et de vérification sémantique.

Mais introduire la description d'un formalisme est aussi une tâche d'édition qui peut être effectuée à l'aide d'un éditeur : EditeurFormalisme. Cet éditeur sera utilisé par un méta-expert du domaine D pour produire la description du formalisme Formalisme(D), exprimée dans le MetaFormalisme. Cet ultime éditeur (nous nous arrêterons là pour l'instant !) pourra être écrit directement à la main ou pourra provenir d'un bootstrap dans lequel il serait engendré par GenerateurEditeur. Dans ce cas un *méta-expert générique* (ou peut-être doit-on dire un métaméta-expert) fournirait à GenerateurEditeur la description du MetaFormalisme dans le MetaFormalisme lui-même (si c'est possible).

EditeurFormalisme
entrées : interaction avec un spécialiste d'un formalisme F
sortie : une description d'un formalisme F, écrite dans le MetaFormalisme

Précisons maintenant ce qui concerne le module ConnaissancesProgrammation et son utilisation par GenerateurApplication. C'est en fait bien souvent la partie la plus difficile à réaliser. Si l'on disposait de techniques générales et efficaces permettant d'engendrer un programme à partir de "connaissances de programmation" simples à exprimer, une grande partie des problèmes de l'informatique disparaîtrait. C'est tout le problème de la "programmation automatique". Cependant, il est envisageable d'engendrer automatiquement des programmes si l'on se restreint à certains formalismes de spécification de systèmes informatiques pour lesquels il existe des techniques de génération. Pour des raisons de faisabilité et d'efficacité, ces formalismes ont pour la plupart un pouvoir d'expression limité, mais ils sont utilisables pour certains types de problèmes.

La démarche Metagen propose de décomposer GenerateurApplication en deux étapes : engendrer d'abord une spécification de système à partir de ConnaissancesDomaine(D), puis engendrer un programme à partir de cette spécification. Elle n'envisage pas en fait d'engendrer séparément une interface de saisie de problème et un résolveur. On introduit alors les deux composants suivants :

GenerateurSpecification(Formalisme(D),S)
entrée : une base de connaissances de résolution ConnaissancesDomaine(D) écrite dans Formalisme(D)
sortie : une spécification de système SpecificationSysteme(S) écrite dans le formalisme de spécification S

GenerateurProgramme(S)
entrée : une spécification de système SpecificationSysteme(S) écrite dans le formalisme de spécification S
sortie : un programme exécutable répondant aux spécifications.

GenerateurSpecification traduit des connaissances d'un domaine en une spécification d'un système résolvant des problèmes du domaine. Celui qui écrit ce générateur doit connaître à la fois le formalisme Formalisme(D) du domaine et le formalisme de spécification S. Metagen propose d'implémenter GenerateurSpecification à l'aide d'un système à base de règles (NeOpus). Ces règles doivent travailler sur des structures informatiques représentant respectivement une base de connaissances ConnaissancesDomaine(D) (donnée en entrée) et une spécification de système SpecificationSysteme(S) (construite comme sortie). Le concepteur des règles doit donc disposer d'un langage lui permettant d'accéder à ces structures, aussi bien pour extraire des éléments et tester des conditions (dans les prémisses des règles) que pour créer, modifier et relier des structures dans les actions des règles. Une partie de ce langage est générique, indépendante des formalismes particuliers, car tous les formalismes qu'on peut définir avec le métaformalisme ont des types génériques d'éléments prédéfinis (des entités, des relations, des rubriques). Une autre partie de ce langage est engendrée automatiquement par un *générateur de langage de règles* à partir de la description d'un formalisme F, elle-même exprimée à l'aide du métaformalisme. Les primitives de langage ainsi engendrées sont surtout des raccourcis d'écriture pour abréger des expressions qu'on pourrait écrire avec la partie générique du langage de règles. Elles se placent en fait au niveau du métalangage propre à F.

GenerateurLangageRegles
entrée : la description d'un formalisme F, écrite dans le MetaFormalisme
sorties : des primitives de langage pour parcourir et créer des représentations en F

Ce générateur de langage de règles doit pouvoir s'appliquer aussi bien au formalisme du domaine Formalisme(D) qu'au formalisme de spécification de systèmes informatiques S. Cela introduit encore une métamodélisation : étant donné qu'il existe plusieurs formalismes de spécification envisageables, il faut d'abord définir quel formalisme va être utilisé. Pour cela, on peut utiliser le même métaformalisme MetaFormalisme et le même éditeur EditeurFormalisme que précédemment, pour produire la définition d'un formalisme de spécification. Cette édition est effectuée par un humain *méta-expert en spécification* (un expert qui maîtrise un formalisme S de spécification de systèmes informatiques).

6.2. Présentation descendante

L'architecture est maintenant complète. En voici tous ses éléments, cette fois-ci en partant du niveau le plus haut (pour chaque élément, on a précisé son langage d'implémentation et son créateur) :

EditeurFormalisme
entrées : interaction avec un spécialiste d'un formalisme F
sortie : une description d'un formalisme F, écrite dans le MetaFormalisme
langage : Smalltalk
créateur : méta-expert générique

GenerateurEditeur
entrée : la description d'un formalisme F, écrite dans le MetaFormalisme
sortie : un éditeur permettant d'introduire des représentations écrites en F
langage : Smalltalk
créateur : méta-expert générique

GenerateurLangageRegles

entrée : la description d'un formalisme F, écrite dans le MetaFormalisme
sorties : des primitives de langage pour parcourir et créer des représentations en F
langage : Smalltalk
créateur : méta-expert générique

EditeurConnaissances(Formalisme(D))
entrées : interaction avec un expert du domaine D
sortie : ConnaissancesDomaine(D)
langage : Smalltalk
créateur : GenerateurEditeur, à partir de Formalisme(D) créé par méta-expert D avec EditeurFormalisme.

GenerateurSpecification(Formalisme(D),S)
entrée : une base de connaissances de résolution ConnaissancesDomaine(D) écrite dans Formalisme(D)
sortie : une spécification de système SpecificationSysteme(S) écrite dans le formalisme de spécification S
langage : neOpus (surcouche de Smalltalk) augmenté du langage de règles associé à Formalisme(D) et à S
créateur : ingénieur de la connaissance, traducteur de Formalisme(D) en S

GenerateurProgramme(S)
entrée : une spécification de système SpecificationSysteme(S) écrite dans le formalisme de spécification S
sortie : un programme répondant aux spécifications, écrit dans un langage cible.
langage : Smalltalk
créateur : Expert en systèmes informatiques spécifiés en langage S

Resolveur(D,S)
entrée : un problème du domaine D
sortie : la solution du problème
langage : langage cible de GenerateurProgramme(S)
créateur : GenerateurProgramme(S)

6.3. Les niveaux de généricité et les experts

Finalement, la généricité dans Metagen s'applique à trois aspects : le domaine D, le formalisme F du domaine D et le formalisme de spécification informatique S. Il y a cependant certaines dépendances : on ne peut pas changer D quand on a choisi F, et le choix de D restreint les possibilités pour F. La démarche est séduisante en raison de cette triple généricité et aussi parce que plusieurs éléments sont engendrés automatiquement à partir de connaissances déclaratives. Les éléments totalement génériques programmés à la main sont EditeurFormalisme, GenerateurEditeur et GenerateurLangageRegles. Un élément programmé à la main et générique par rapport à D et à F, mais dépendant de S est GenerateurProgramme(S). Les éléments contenant l'essentiel des connaissances sont ConnaissancesDomaine(D) et GenerateurSpecification(Formalisme(D),S).

L'architecture Metagen demande la participation de plusieurs experts humains. Précisons qui sont ces experts, leurs compétences et leurs rôles dans l'architecture.

• Le *méta-expert générique* sait ce qu'est un formalisme et il connaît le MetaFormalisme. Il sait en quoi consiste la tâche "décrire un formalisme en utilisant le MetaFormalisme" et il est capable d'implémenter un éditeur permettant à un utilisateur d'effectuer cette tâche. Il sait en quoi consiste la tâche "décrire des connaissances en utilisant un formalisme F" et il est capable d'implémenter un programme permettant, à partir d'une description de F, d'engendrer un éditeur permettant à un utilisateur d'effectuer cette tâche. Il sait en quoi consiste la tâche "parcourir et créer des représentations écrites dans le formalisme F" et il est capable d'implémenter un programme permettant, à partir d'une description de F, d'engendrer des primitives de langage pouvant servir d'outils à un utilisateur pour effectuer cette tâche.

• Le *méta-expert du domaine D* connaît le MetaFormalisme et le formalisme Formalisme(D) attaché au domaine D. Il sait décrire Formalisme(D) à l'aide du MetaFormalisme. Il doit être capable d'utiliser l'éditeur de formalisme mis à sa disposition pour cela.

• L'*expert du domaine D* connaît les concepts du domaine D et possède des connaissances pour résoudre des problèmes du domaine D. Il connaît le formalisme Formalisme(D) propre au domaine et il sait exprimer les concepts et les connaissances dans ce formalisme. Il doit être capable d'utiliser l'éditeur de connaissances mis à sa disposition pour introduire ses connaissances dans le formalisme Formalisme(D).

• L'*ingénieur de la connaissance* spécialisé dans le domaine D et dans la spécification de systèmes informatiques dans le formalisme S connaît le MetaFormalisme, le Formalisme(D) et le formalisme S. Il doit être capable d'écrire des règles permettant d'opérationnaliser les connaissances fournies par l'expert du domaine D, c'est-à-dire de les traduire en une spécification d'un système informatique écrite dans le formalisme S. Il peut pour cela utiliser les primitives de langage mises à sa disposition pour parcourir et créer des représentations écrites dans Formalisme(D) et en S. Il est très probable que, pour écrire ces règles, il doit posséder des connaissances sur le domaine D (la traduction n'est pas purement syntaxique) ainsi que des métaconnaissances lui permettant d'évaluer des caractéristiques des connaissances comme leur importance, leur rôle, leur certitude, etc. Il doit de même posséder des connaissances et métaconnaissances sur les concepts liés à la spécification de systèmes dans le formalisme S.
Par rapport aux autres experts, c'est lui qui a la tâche la plus difficile et nécessitant le plus de créativité. Il doit faire le lien entre l'aspect déclaratif des connaissances et l'aspect opérationnel de la spécification d'un système, et ceci en passant parfois au niveau méta. Remarquons qu'il est présenté comme un spécialiste et non pas comme un généraliste : il est spécialisé dans un formalisme et un domaine d'entrée (D) et dans un formalisme et des concepts de sortie (S). Il y a cependant de fortes chances que, pour mener à bien sa tâche spécialisée, il possède aussi des connaissances sur la tâche générique consistant à transformer des connaissances en une spécification de système. Mais un tel niveau de généricité n'a pas été introduit explicitement dans l'architecture.

• L'*expert en systèmes informatiques spécifiés en S* connaît le formalisme de spécification S et est capable d'implémenter un générateur de programme permettant de produire un programme exécutable à partir d'une spécification écrite en S. Sa tâche n'est a priori pas facile mais il peut bénéficier de toute l'expérience accumulée en informatique dans ce domaine. Les principes de tels générateurs ont été mis au point, du moins pour certains formalismes.

En quoi Metagen se rapproche-t-il du modèle à deux agents que nous avons décrit plus haut ? Notre agent généraliste est indépendant du domaine mais il met en oeuvre une certaine stratégie fixée liée à un certain type de tâche fixé (résolution de problèmes par reformulation-classification pour Syrclad, génération de problèmes par chaînage arrière pour Sygep). Autrement dit, il n'est pas entièrement générique vis-à-vis du domaine. Apparemment, Metagen ne contient pas de tâche prédéfinie ni de stratégie prédéfinie. Cependant, choisir le formalisme de spécification S peut fixer implicitement une tâche et une stratégie, en particulier s'il s'agit d'un formalisme lié à un framework mettant en oeuvre une certaine stratégie pour effectuer un certain type de tâche. Le formalisme du domaine peut ensuite être défini en fonction de cette tâche et de cette stratégie, donc finalement en fonction de S. Par exemple, pour Syrclad, le formalisme de spécification S peut contenir les notions de “graphe de classification” et d'“attributs discriminants” qui sont liées à la stratégie mais indépendantes du domaine. L'agent généraliste serait alors la réunion de GenerateurProgramme et de certaines règles de GenerateurSpecification (celles qui sont indépendantes du domaine D). L'agent spécialiste, quant à lui, serait la réunion de ConnaissancesDomaine(D) et aux règles de GenerateurSpecification qui dépendent du domaine.

7. Conclusion

Nous avons décrit un système opérationnel capable d'effectuer une tâche T dans un domaine D en utilisant une stratégie S comme la réunion d'un agent généraliste possédant des connaissances sur la façon d'effectuer la tâche T avec la stratégie S et d'un agent spécialiste possédant des connaissances sur le domaine D et sur la réalisation particulière de la stratégie S dans le domaine D.
Nous avons envisagé trois types de communication entre les deux agents (délégation, paramétrage, parasitage) et donné des exemples de mécanismes de base utilisables (classes abstraites et méthodes virtuelles, fonctions du deuxième ordre, hooks, callbacks). Nous avons donné des exemples de types d'architectures possibles pour exploiter des systèmes génériques (frameworks, métamodélisation) et nous avons donné l'exemple de l'architecture de Metagen, fondée sur la métamodélisation.
Cette revue est loin d'être exhaustive, car le thème de la généricité est présent sous de nombreux aspects : la réutilisabilité, la programmation par composants, les méthodes de résolution de problèmes, les “design patterns”, les ateliers et méta-ateliers de génie logiciel, la programmation automatique à partir de spécifications, les générateurs d'interfaces utilisateur, les outils et méta-outils d'acquisition de connaissances. Voir une revue détaillée dans [Revault 96].
Le concept de généricité est lié au concept de niveau méta. Les deux concepts ne sont cependant pas confondus. Un concept générique comporte des éléments indéterminés destinés à être instanciés par un spécialiste (comme des paramètres ou des méthodes virtuelles). Il résulte d'une abstraction et d'une factorisation des concepts communs à plusieurs domaines. Le concept de niveau méta est plus difficile à cerner [Pitrat 90]. Il est présent ici par deux aspects : d'une part la représentation explicite d'un formalisme dans un métaformalisme et l'exploitation de cette représentation par des méta-outils capables d'engendrer des outils, et d'autre part l'utilisation de métaconnaissances pour opérationnaliser les connaissances d'un domaine. L'utilisation de techniques “méta” dans la réalisation d'un système générique a l'avantage de rendre les connaissances plus

explicites et plus déclaratives (donc plus faciles à exprimer et mieux exploitables), et de mieux répartir “l’intelligence” entre l’agent généraliste et l’agent spécialiste.

8. Références

[GNU Emacs 98] GNU Emacs Lisp Reference Manual - Hooks,
URL http://www.tac.nyc.ny.us/manuals/elisp/elisp_292.html

[Guin 97] Guin N., Reformuler et classer un problème pour le résoudre, l’architecture Syrclad et son application à quatre domaines, Thèse de doctorat, Université Paris 6, 12 décembre 1997.

[Kornman 93] Kornman S., Sade : un système de surveillance réflexif à base de connaissances. Thèse de l’Université Paris 6, 1993.

[Parchemal 88] SEPIAR: un systeme a base de connaissances qui apprend a utiliser efficacement une expertise. Thèse de doctorat, Université Paris 6, 22 décembre 1988.

[Pécégo 98] Pecego G., SYGEP, Un système de génération d’énoncés de problèmes dans des domaines variés, Thèse de doctorat, Université Paris 6, 12 juin 1998.

[Pitrat 90] Pitrat J., Métaconnaissance : Futur de l’intelligence artificielle. Paris, Hermes, 1990.

[Revault 96] Revault N., Principes de méta-modélisation pour l’utilisation de canevas d’applications à objets (Metagen et les Frameworks), Thèse de doctorat, Université Paris 6, 19 novembre 1996.

[Steele 90] Steele G.L.Jr, Common Lisp the Language, second edition, Digital Press, 1990.

GÉNÉRALISATION ET PETITS ÉCHANTILLONS

Michel MASSON

masson@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé:

Ce papier aborde un problème fréquemment rencontré: comment généraliser un ensemble d'observations lorsque celui-ci est petit? Après avoir parcouru plusieurs familles de solutions, nous décrivons le système GENSAM basé sur l'explicitation des connaissances nécessaires à la généralisation. Bien que réalisé dans un cadre particulier, la démarche utilisée est générale: ayant la possibilité de prendre en compte les connaissances du domaine de l'utilisateur, elle fournit une caractérisation des observations de départ adapté aux besoins de ce dernier.

Mots-clés: échantillon, généralisation, méta-connaissances.

1. Introduction

De nombreuses expérimentations peuvent être décrites comme un ensemble réduit d'observations; l'échantillon ainsi obtenu permet difficilement de généraliser l'expérience réalisée. La recherche médicale clinique abonde en situations semblables, un échantillon regroupe pour chaque patient quelques dizaines de paramètres (description du patient, traitement suivi) assortis d'un classement a posteriori (ex. l'efficacité du traitement suivi avec un recul de 6 mois). Lorsque l'échantillon de départ est important, les méthodes statistiques classiques permettent de mettre en évidence un processus prédictif: la seule connaissance des données initiales permet de prévoir l'efficacité du traitement avec une bonne fiabilité. Or bien souvent, des contraintes spécifiques à l'application limitent la taille de l'échantillon expérimental (nombre limité d'observations possibles, nécessité de mener l'expérimentation dans un cadre contraignant), les méthodes statistiques perdent alors de leur efficacité. L'expérience montre qu'un opérateur humain perspicace peut dépasser les limites de ces méthodes (en les combinant ou en affinant les conditions de validité) pour obtenir un résultat exploitable. Après avoir examiné les limites des méthodes classiques, nous proposons une approche basée sur plusieurs niveaux d'analyse de l'échantillon de départ; à chaque niveau l'ensemble des connaissances est explicité.

2. Approche méthodologique.

Cette étude a pour point de départ l'analyse a posteriori d'un ensemble de dossiers médicaux concernant l'intérêt du lavage et/ou brossage bronchoalvéolaire en l'absence de contrôle fibroscopique chez l'enfant sous assistance respiratoire pour l'aide au diagnostic des infections respiratoires [Labenne & Goldfarb 98].

Le lavage et/ou brossage bronchoalvéolaire sont des techniques de prélèvement bronchique en cas d'infection respiratoire contractée pendant une phase d'assistance respiratoire. Chez l'adulte, ces examens sont normalement réalisés sous contrôle endoscopique; le diamètre minimal de l'appareil d'observation (3,6 mm) implique l'usage d'une sonde endo-trachéale d'un diamètre minimal (4,5 mm). Une sonde d'un tel diamètre ne peut être utilisée chez l'enfant en bas âge pour des raisons anatomiques. En fait, ces caractéristiques anatomiques varient d'un individu à un autre: l'âge a partir duquel l'examen peut être réalisé sous contrôle endoscopique varie donc.

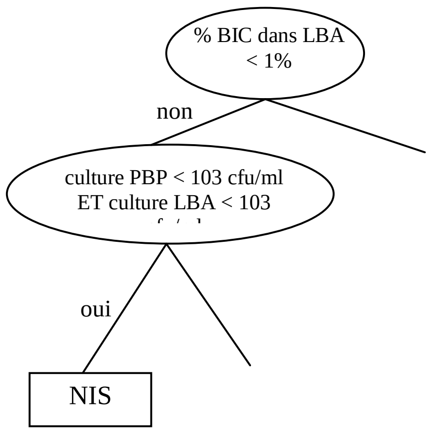
L'étude regroupe 33 observations: pour chacune d'elles on dispose de données morphologiques, du résultat d'examens biologiques, du résultat des analyses des prélèvements bronchiques et de l'évolution de la maladie avec un recul de 6 mois. Deux groupes décrivent l'évolution possible:

- Groupe NIS (Non Infecté Sûr)
- Groupe IS (Infecté Sûr)

La très grande sensibilité de ces examens impose un protocole très strict, d'autant mieux respecté qu'une même équipe assure l'ensemble des opérations définissant le protocole. Ces contraintes expliquent le petit nombre de cas observés.
Chaque observation est décrite par 36 variables, chacune pouvant prendre une valeur numérique, qualitative ou booléenne:

V_1: diamètre_sonde_intubation (en cm)
V_2: age (en mois)
V_3: poids (en gramme)
V_4: nombre_jours_intubation (entier)
V_5: cathéter (oui/non)
V_6: température_arrivée (réel)
V_7: aggravation_respiratoire (oui/non)
V_{11}: température_H+6 (réel)
V_{12}: aspiration sanglante (oui/non)
V_{13}: test_saturation_oxygène (oui/non)
V_{14}: pneumothorax (oui/non)
V_{15}: ponction_pleurale (positive/négative)
etc.

Le but est d'obtenir à partir de l'échantillon un questionnaire (ou arbre de décision) permettant de discriminer les observations initiales vers les 2 groupes définis précédemment avec une marge d'erreur aussi faible que possible. De plus, chaque sommet de l'arbre doit correspondre à une condition ou à une combinaison simple de conditions portant sur les observations initiales afin d'éviter toute difficulté de manipulation et/ou d'interprétation. La figure ci-dessus montre une partie du questionnaire (LBA : Lavage BronchoAlvéolaire, BIC: pourcentage de bactéries intracellulaires, PBP: Prélèvement Bronchique Protégé).



3. Limites des méthodes classiques.

Plusieurs méthodes permettent d'aborder ce problème et les contraintes afférentes; malheureusement aucune ne couvre de façon satisfaisante l'ensemble des contraintes. La plupart des travaux concernant ce domaine utilisent l'outil statistique.

3.1. La méthodologie statistique.

Cette méthodologie fournit un ensemble d'outils standardisés, facilement utilisables pour lesquels il peut être tentant de s'affranchir des conditions de validité.

La **régression** [Tomassone 88a] permet d'expliquer une variable Y (dite variable expliquée) à l'aide de variables explicatives $X_1, X_2, \ldots, X_n$; une correction statistique est introduite grâce au coefficient ε:

$$Y = \Sigma\, b_i X_i + \varepsilon$$

Les X_i correspondent aux variables décrivant l'échantillon. Lorsque la variable Y est discontinue (dans notre étude, elle ne peut prendre que 2 valeurs correspondant au diagnostic confirmé à 6 mois: IS et NIS), il faut introduire une régression logistique. Soit p (resp. 1-p) la probabilité d'évoluer vers IS (resp. NIS), la variable expliquée est alors de la forme $Y = \mathrm{Log}\ (p / 1-p)$. La seule connaissance des X_i permet de connaître le rapport des probabilités associées à IS et NIS. Les variables X_i doivent être indépendantes, ce qui est rarement le cas dans la réalité (ainsi la présence d'un pneumothorax implique une aggravation respiratoire). De plus, si k désigne le taille de l'échantillon et n le nombre de variables explicatives, il est conseillé de respecter la contrainte $k \geq 2 \times n$ (ce qui n'est pas le cas dans l'exemple étudié).

L'**analyse discriminante** [Tomassone 88b] correspond mieux au problème posé: elle permet d'obtenir une classification des diagnostics à 6 mois à l'aide de combinaisons linéaires des signes observés (qualitatifs ou quantitatifs). Le choix d'une distance mesurant les écarts observés sur P permet d'obtenir un classement des diagnostics à 6 mois pour tout nouvel individu additionnel.

L'inconvénient majeur de ces méthodes réside dans leur absence de prise en compte de certaines des connaissances du domaine, cette absence pouvant intervenir à plusieurs niveaux:
la taille et le poids sont à priori corrélés et ceci est pris en compte dans l'analyse discriminante, mais la connaissance a priori par le système d'une telle information permettrait d'examiner (par exemple) si un trouble de la croissance (absence de corrélation poids / taille chez certains individus de l'échantillon) peut influer le diagnostic à 6 mois.
certains résultats sont totalement inférables à partir d'autres (ex. l'aggravation respiratoire à partir de la présence d'un pneumothorax). Utiliser une information semblable permet de réduire la combinatoire des signes.

3.2. L'algorithme ID3.

Cet algorithme développé initialement par J. R. Quinlan [Quinlan 79], [Quinlan 83], [Quinlan 86], et ses dérivés [Utgoff 89] correspondent parfaitement au problème posé: on dispose au départ d'un ensemble d'éléments caractérisés par une classe d'appartenance (pour l'exemple étudié, il s'agira du pronostic à 6 mois) et une liste de paires < attribut valeur >. L'algorithme sélectionne un attribut et utilise sa valeur pour discriminer l'ensemble des éléments de départ, il recommence pour chaque sous-ensemble ainsi obtenu jusqu'à ce que les sous-ensembles ainsi construits ne contiennent que des éléments appartenant à une des classes de départ. L'arbre obtenu est de taille minimale. Cette optimisation est basée sur un principe de la théorie de l'information: il sélectionne chaque attribut en fonction de son aptitude à diminuer l'entropie des sous-ensembles construits à chaque étape.
Malheureusement le questionnaire obtenu ne correspond pas obligatoirement à ce qui est attendu: l'obtention d'un questionnaire discriminant parfaitement chaque cas, peut conduire à surspécifier un groupe (ainsi, de façon abusive le groupe NIS peut être défini par la disjonction de l'ensemble des cas évoluant 6 mois plus tard vers NIS) , l'objectif de généralisation n'est pas réalisé.

La méthodologie statistique des arbres de régression, connue sous le nom de méthode CART [Breiman & al 84] est très proche du principe développé dans l'algorithme ID3. Dans le paradigme de l'arbre, le premier niveau de l'arbre est formé par un noeud unique, la racine, contenant l'ensemble des observations; les niveaux suivants sont formés de noeuds disjoints et dont l'union forme le noeud parent. Les noeuds donnant lieu à subdivision sont dits intermédiaires, les autres sont appelés noeuds terminaux. Comme dans l'algorithme ID3, les étapes de division correspondent à l'application d'un critère, le plus souvent basé sur l'entropie. Le point crucial de la méthode est celui de l'arrêt de la division. Cette méthodologie est fréquemment utilisée dans les contextes de classification et/ou de reconnaissances des formes.
L'arrêt de la décomposition permet d'éviter la surspécification des groupes étudiés, mais la prise en compte des connaissances du domaine ne peut être assurée.

3.3. Le raisonnement par cas.

Le raisonnement par cas est une technique développée en Intelligence Artificielle pour la résolution de problèmes ou le diagnostic [Kolodner 93]. La problématique semble quantitativement voisine de celle concernant la généralisation à partir d'un échantillon de petite taille: résoudre un nouveau problème à l'aide d'un petit nombre (quelques dizaines) d'exemples. En fait, si un cas représente un fragment d'expertise, un élément d'échantillon ne représente rien. Seul l'échantillon est représentatif de l'expérimentation en cours. Plusieurs techniques de raisonnement sont possibles.

Dans les systèmes PROTOS [Porter & al 90] et MEDIATOR [Kolodner & Simpson 89], une première étape sélectionne les *cas* candidats à l'appariement avec le *nouveau cas*; chaque coïncidence est notée, le *cas* assorti du meilleur score fournit la solution.

Dans le système CASEY [Koton 88], des règles de coïncidence permettent d'apparier un cas répertorié avec tout nouveau cas; lorsque plusieurs règles sont en conflit, une expertise de résolution de conflit permet d'isoler le cas le plus représentatif. CASEY dispose pour cela de connaissances sur la valeur d'un attribut, sur le degré de coïncidence exigé pour un attribut (un même écart de valeur - par exemple 2 - concernant l'âge ou la tension artérielle ne sera pas traité de la même façon).
Cette dernière méthode est intéressante car elle permet de regrouper des éléments d'échantillon aux valeurs très proches et de limiter la combinatoire.

4. Une solution basée sur l'explicitation des connaissances: le système GENSAM (GENeralization based on small SAMples).

4.1. Principes de base.

Nous utiliserons les notations suivantes:
Un échantillon E est défini de la façon suivante: $E = \{e_1, \ldots , e_n\}$, où e_i désigne un individu caractérisé par une classe d'appartenance et une liste de paires < attribut valeur >. Nous appellerons **sous-population** tout sous-ensemble de E.
Ex. l'ensemble des individus dont le diagnostic à 6 mois (i.e. la classe d'appartenance) est NIS, l'ensemble des individus dont le poids est inférieur à 3kg et la durée d'intubation est inférieure à 3 jours constituent deux sous-populations de E.
GENSAM doit pouvoir juger ses résultats, pour cela nous introduisons deux mesures (indépendantes du domaine étudié). Soient $E_1, E_2 \subset E$; nous pouvons alors définir la **sensibilité** et la **spécificité** de E_1 par rapport à E_2:

sens $(E_1, E_2) = \text{card } (E_1 \cap E_2) / \text{card } (E_2)$
spec $(E_1, E_2) =$ **sens** $(E_2, E_1) = \text{card } (E_1 \cap E_2) / \text{card } (E_1)$

Si S_i désigne la sous-population où le signe s_i est présent, si NIS désigne la population dont le diagnostic confirmé à 6 mois est Non Infecté Sûr, **sens** (S_i , NIS) mesure la prévalence de S_i chez la sous-population NIS; elle est d'autant plus proche de 1 que le signe est fréquemment rencontré. **spec** (S_i , NIS) indique si le signe est fréquemment rencontré en dehors de la sous-population NIS. Ainsi S_i caractérise d'autant mieux la sous-population NIS que l'on a:

sens (S_i , NIS) et **spec** (S_i , NIS) proches de 1.

4.2. Aperçu général.

GENSAM comporte 4 niveaux. Le niveau DONNEES décrit les éléments de l'échantillon ainsi que les résultats produits par le système. Le niveau OBSERVATIONS assure une classification grossière des éléments du niveau DONNEES, on peut ainsi identifier des sous-populations susceptibles d'être obtenues par intersection d'autres sous-populations. Le niveau CONTRAINTES définit les règles permettant d'obtenir de nouveaux éléments intéressants. Le niveau CONTROLE gère les niveaux précédents; il est totalement figé et ne peut donc être modifié par l'utilisateur.

4.3. Le niveau DONNÉES:

C'est le niveau de base, il décrit les sous-populations manipulées par le système. Soit un échantillon défini de la façon suivante:

$$E = \{ \text{ classe } (var_1 \;\; val_1) \ldots (var_p \ldots val_p) \}$$

Pour chaque variable prenant une valeur qualitative ou booléenne, est associée une sous-population constituée des éléments de E ayant cette valeur pour cette variable. En reprenant les variables utilisées dans l'étude, on obtient:

E1 = E {pneumothorax = oui}
E2 = E {pneumothorax = non}
....

Il importe de savoir comment une sous-population caractérise un des groupes prédéfinis (i.e. NIS, IS); chaque sous-population est décrite par un ensemble de paires <attribut valeur >

E1 :	priorité	peut prendre les valeurs 0, 1, 2 (cf. § 4.6)
	sensNIS = x_{11}	où x_{11} est le résultat de **sens** (E_1, NIS)
	specNIS = x_{12}	où x_{12} est le résultat de **spec** (E_1, NIS)
	sensNISmin = x_{13}	où x_{13} est un minorant de **sens** (E_1, NIS)
	sensNISmaj = x_{14}	où x_{14} est un majorant de **sens** (E_1, NIS)
	sensNISheur = x_{15}	où x_{15} est une estimation heuristique de **sens** (E_1, NIS)
	.....	

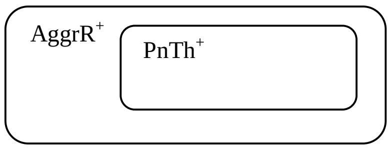
Au départ, seules les sous-populations correspondant aux éléments de l'échantillon sont présentes; GENSAM construit au fur et à mesure de nouveaux éléments qui sont alors décrits par une liste <attribut valeur> semblable à la précédente. Il serait pénalisant en temps de devoir calculer pour chaque nouvelle sous-population les grandeurs sensibilité et spécificité, aussi GENSAM précise ces valeurs par des limites et/ou des estimations. Ainsi, pour une sous-population donnée, seule une partie de ces attributs peut être définie.

C'est à ce niveau que sont prises en compte les connaissances du domaine. Elles sont de la forme *(objet lien objet)* où *lien* peut désigner un terme comme *implique, évoque, varie comme*. Ainsi la dépendance évoquée au § 3.1 peut se traduire par:

"la présence d'un pneumothorax implique une aggravation respiratoire"

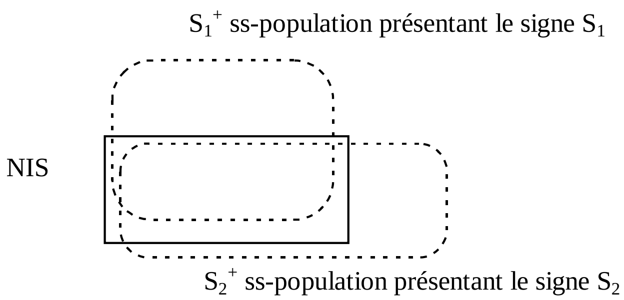
Posons $PnTh^+$ = E {présence_pneumothorax = oui}, $AggrR^+$ = E {aggravation_respiratoire = oui}, la connaissance précédente peut se traduire par:

spec ($PnTh^+$, $AggrR^+$) = 1



4.4. Le niveau OBSERVATIONS:

Ce niveau regroupe les sous-populations en fonction de leur aptitude à participer à la construction d'ensembles plus complexes. L'exemple suivant montrera l'intérêt de ce niveau. Soient NIS la sous-population correspondant au pronostic à 6 mois "Non Infecté Sûr", et S_1^+ , S_2^+ deux sous-populations correspondant à la présence des signes S_1, S_2 dans l'échantillon observé:



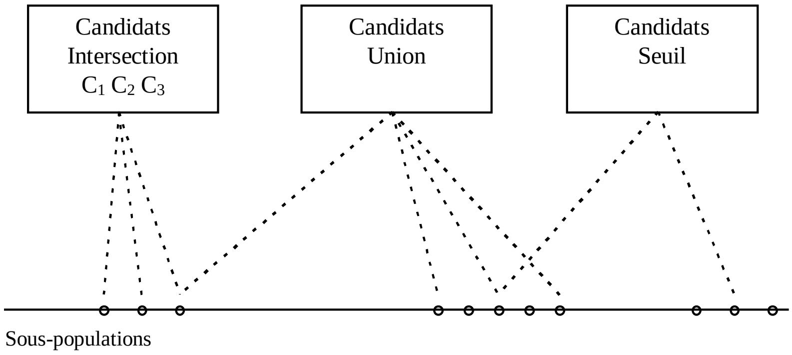
Cela correspond à	**sens** (S_1^+, NIS) = 0.8	**sens** (S_2^+, NIS) = 0.9
	spec (S_1^+, NIS) = 0.5	**spec** (S_2^+, NIS) = 0.6

On observe que la sous-population $S_1^+ \cap S_2^+$ caractérise mieux NIS puisque:

sens ($S_1^+ \cap S_2^+$, NIS) = 0.75   spec ($S_1^+ \cap S_2^+$, NIS) = 1

Les sous-populations candidates à l'intersection correspondent à des éléments du niveau DONNEES ayant une forte sensibilité et une spécificité moyenne par rapport à un des 2 groupes prédéfinis (NIS IS). Les objets de ce niveau sont définis par un ensemble de contraintes. Ainsi la classe des "Candidats à l'Intersection" est définie par les contraintes:

C_1:	**sens** (*, G) < Seuil-faible	* désigne un élément quelconque du niveau DONNEES
C_2:	G = NIS | IS	G désigne une variable pouvant prendre une des 2 val.
C_3:	**spec** (*, G) > Seuil-moyen	



Tout nouvel élément du niveau DONNEES est automatiquement apparié à une ou plusieurs classes en fonction des contraintes vérifiées.

4.5. Le niveau CONTRAINTES.

GENSAM parvient à caractériser les groupes prédéfinis (NIS IS) en agrégeant les sous-populations entre elles: il utilise pour cela les opérateurs suivants:

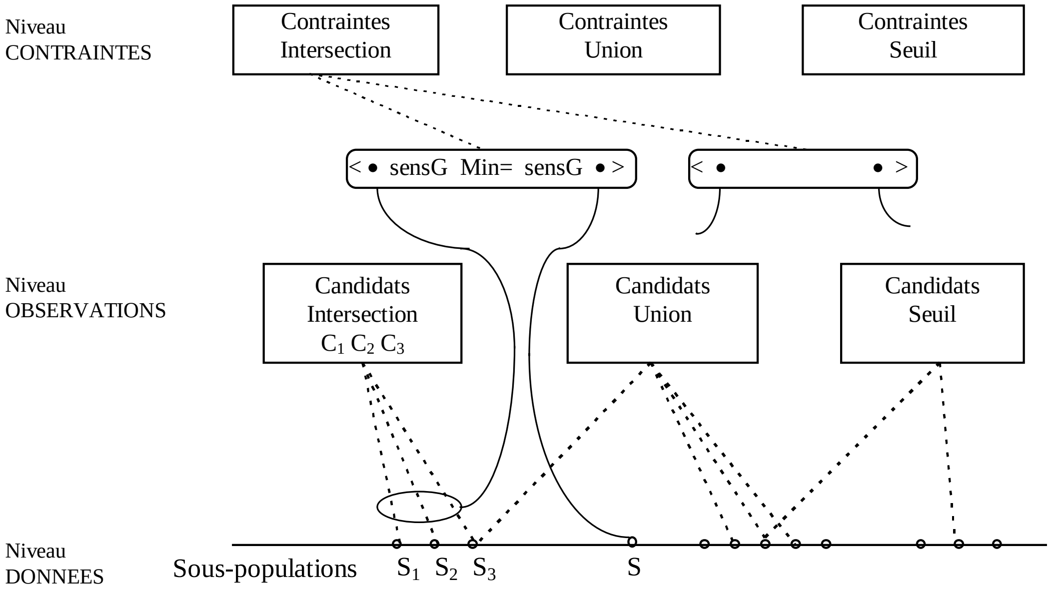
intersection $S = \cap \{S_1 ... S_n\}$, la sous-population S est obtenue par intersection des sous-populations $S_1 ... S_n$
réunion $S = \cup \{S_1 ... S_n\}$, la sous-population S est obtenue par réunionion des sous-populations $S_1 ... S_n$
seuil S = S' [var comp val], la sous-population S obtenue en contraignant la variable **var** pour les éléments de la sous-population S'

A chaque opérateur correspond un ensemble de contraintes portant sue les éléments du niveau DONNEES. Ainsi pour l'opérateur intersection, on a:

(*) **sens** (S, G) < Min $_{i = 1, n}$ {**sens** (S_i , G)} avec G = IS, NIS

La forme générale d'une contrainte est < objet$_1$ attribut$_1$ RELATION attribut$_2$ objet$_2$> ; elle relie les attributs de deux objets par une relation qui peut être:

ISO	les attributs associés aux objets objet1 et objet2 ont même valeur
=SOM	attribut$_1$ prend pour valeur la somme des valeurs prises par attribut$_2$
SOM=	attribut$_2$ prend pour valeur la somme des valeurs prises par attribut$_1$
Max=	attribut$_2$ prend pour valeur la plus grande des valeurs prises par attribut$_1$
=Max	attribut$_1$ prend pour valeur la plus grande des valeurs prises par attribut$_2$
Min=	attribut$_2$ prend pour valeur la plus petite des valeurs prises par attribut$_1$
ESTIM	la valeur de attribut$_1$ est une estimation de celle de attribut$_2$
etc.	



La contrainte (*) peut s'écrire:

< S **sens**G Min= **sens**G S_i > ou encore < S_i **sens**G =Min **sens**G S >

où **sens**G désigne la sensibilité calculée par rapport à un des groupes prédéfinis IS ou NIS.

4.6. Le niveau CONTRÔLE.

Ce niveau assure la propagation des contraintes du niveau précédent et l'actualisation des éléments concernés par la propagation. Selon la nature des contraintes précédemment décrites, plusieurs modes de propagation sont à envisager:

< o_1 $attr_1$ **ISO** $attr_2$ o_2 > et < o_2 $attr_2$ **ISO** $attr_3$ o_3 > donnent la nouvelle contrainte:
< o_1 $attr_1$ **ISO** $attr_3$ o_3 >
(transitivité du lien **ISO**)
< o_1 $attr_1$ **ESTIM** $attr_2$ o_2 > et < o_2 $attr_2$ **ISO** $attr_3$ o_3 > donnent la nouvelle contrainte:
< o_1 $attr_1$ **ESTIM** $attr_3$ o_3 >
(si la valeur d'un attribut d'un objet est estimée à partir de celle d'un autre objet, il en est de même pour tout objet lié à un des précédents par une relation **ISO)**
etc**.**

Une fois les contraintes propagées, GENSAM cherche à calculer les critères **spec** et **sens** associés aux objets du niveau DONNEES. Lorsque cela est possible (i.e. le nombre d'éléments à calculer est petit), le calcul est réalisé pour tous les éléments du niveau. Dans le cas contraire, GENSAM détermine les éléments prioritairement traités.
Notations:
E_i désigne un élément du niveau DONNEES, C_i désigne une classe du niveau OBSERVATIONS.
Tout élément E_i du niveau DONNEES possède un attribut **priorité** défini de la façon suivante:
priorité (E_i) = 2 si E_i a été calculé à l'étape précédente (les éléments récemment calculés sont prioritaires)
= 1 si **sens** (E_i, G) = 1 ou **spec** (E_i, G) = 1 (E_i correspond à une sous-population qui inclut ou est incluse dans un des groupes prédéfinis G, c'est donc une sous-population intéressante)
= 0 sinon
On peut alors définir les critères suivant pour les éléments du niveau OBSERVATIONS:

Elts-calc (C_i) = Card{ E_i / E_i apparié à C_i et **sens** (E_i, G) et/ou **spec** (E_i, G) non définis}
Priorité$^+$ (C_i) = Card{ E_i / E_i apparié à C_i et **priorité** (E_i) = 2 } / Card (C_i)

La stratégie est basée sur le degré d'actualisation des éléments du niveau DONNEES (i.e. le ratio des éléments dont on connaît un seul ou aucun des critères **spec** et **sens** par rapport au total des éléments) et la priorité des éléments de ce niveau (si le ratio d'éléments prioritaires est semblable dans toutes les classes du niveau OBSERVATIONS, alors on actualise les éléments prioritaires de la classe la moins actualisée, sinon les éléments prioritaires de la classe prioritaire sont actualisés). Elle correspond aux étapes suivantes:

(i) Si [Σ Elts-Calc(C_i)] / [Σ Card (C_i)] > 0.8 alors Calculer tous les éléments
Si $\exists$ C_i telle que Elts-Calc(Ci) / Card (C_i) > 0.8 alors Calculer les éléments de C_i
Si $\forall$ i, j | Priorité$^+$ (C_i) - Priorité$^+$ (C_j) | < 0.2 alors choisir C_i telle que **Elts-calc** (C_i) / Card (Ci) soit minimal et actualiser les éléments de C_i de **priorité** = 1, 2

(iii) Sinon, choisir C_i telle que **Priorité$^+$** (C_i) soit maximale et actualiser les éléments appariés à C_i de priorité = 2

(iv)

5. Résultats et Perspectives.

La stratégie du niveau CONTROLE ainsi que les calculs des attributs des éléments du niveau DONNEES sont réalisés en langage C. Les éléments des niveaux CONTRAINTES et OBSERVATIONS sont des frames auxquels sont associées des règles CLIPS (Système à Base de Connaissances construit autour de l'algorithme RETE). Avec un échantillon de 33 éléments, chacun étant décrit par 36 variables et un classement en deux groupes (IS et NIS), on obtient une caractérisation de ces groupes en 25 minutes, ce temps augmente si on accroît la taille de l'échantillon et/ou le nombre de groupes caractérisés: par ex. IS, NIS + IP (Infecté Probable), NIP (Non Infecté Probable). Ces performances pourraient être améliorées en procéduralisant la propagation des contraintes et en utilisant au maximum les connaissances du domaine. Il importe alors de permettre l'introduction de ces connaissances grâce à un langage simple, le langage LCD [Masson 96] adapté à la manipulation de connaissances sur le diagnostic en est une illustration.

Ce travail a été réalisé à partir d'une étude médicale conduite par B. Goldfarb [Laboratoire de Biostatistique et Informatique Médicale, Hôpital Necker & LISE-Ceremade, Université Paris-Dauphine] que je remercie par ailleurs pour les enrichissements concernant la méthodologie statistique et pour m'avoir permis de dégager les éléments essentiels nécessaires au jugement médical.

6. Références

[Breiman & al 84] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone. *Classification and Regression Trees*, Chapman et Hall, 1984.
[Kolodner 93] J. Kolodner. *Case-Based Reasoning*, Morgan Kaufmann Publishers, San Mateo, Californie, 1993.
[Kolodner & Simpson 89] J. Kolodner, R. L. Simpson. The MEDIATOR: Analysis of an early case-based problem solver, Cognitive Science 13, 507-549, 1989.
[Koton 88] P. Koton. Integrating case-based and causal reasoning, *Proceedings of the Xth Annual Conference of the Cognitive Science Society*, Northvale, 1988.
[Labenne & Goldfarb 98] M. Labenne, B. Goldfarb. Blind-protected specimen brush and brochoalveolar lavage in ventilated children. *Critical Care*, 1998.
[Masson 96] M. Masson. Réifier le raisonnement pour améliorer les explications dans les systèmes à base de connaissances: une application au diagnostic médical, *RJC-IA'96*, Nantes, 1996.
[Porter & al 90] B. W. Porter, R. Bareiss, R. C. Holte. Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligece*, 45, 229-263, 1990.
[Quinlan 79] J. R. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the Micro-electronic Age*, 168-201, Michie, D., Editor, Edinburgh University Press, Edinburgh, Scotland, 1979.
[Quinlan 83] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In *Machine Learning, An Artificial Intelligence Approach*, Pages 463-482, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., Editors, Tioga Publishing Company, Palo Alto, CA, 1983.
[Quinlan 86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81-106, 1986.
[Tomassone 88a] R. Tomassone. *Comment interpréter les résultats d'une régression linéaire ?* ITCF, Paris, 1988.
[Tomassone 88b] R. Tomassone. *Comment interpréter les résultats d'une analyse factorielle discriminante ?* ITCF, Paris, 1988.
[Utgoff 89] P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161-186, 1989.

EXEMPLES DE TECHNIQUES DE REFORMULATION APPLIQUÉES AU PROBLÈME DU TOURNOI D'ÉCHECS

Arnaud FREDON

Arnaud.Fredon@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé :

Nous donnons dans ce papier quelques exemples de techniques de reformulations. Ces techniques sont appliquées et commentées de façon concrète en s'appuyant sur un problème dit du tournoi d'échec. Le lecteur pourra ainsi voir facilement l'intérêt de méthodes de reformulation en suivant l'évolution de l'énoncé du problème au fur et à mesure de l'article.

Mots clefs : Reformulation, CSP

1. Introduction à la Problématique de la Reformulation

Depuis les débuts de l'intelligence artificielle, de nombreux efforts ont été faits pour rendre les ordinateurs capables de résoudre des problèmes complexes. Cependant, dans tous les travaux réalisés, on constate que les problèmes sont souvent donnés sous une forme particulièrement adaptée aux capacités du programme de résolution. Ce sont alors les programmeurs qui font montre d'une grand astuce dans leurs façons de représenter les problèmes. En modifiant ainsi l'énoncé d'un problème, les programmeurs d'intelligence artificielle effectuent des reformulations.

Un exemple simple (voire simpliste) permet de montrer l'intérêt des reformulations. Ainsi, si l'on dispose d'un système plus efficace pour effectuer des additions que des multiplications, on peut être tenté de remplacer la forme 3*4 par 4+4+4. Cette opération de réécriture est simple mais permet d'adapter un énoncé de problème aux capacités d'un système cible.

L'idée de reformuler n'est bien sûr pas nouvelle. Déjà en 1966 Alan Newell écrit que résoudre un problème, c'est le reformuler jusqu'à ce qu'il énonce sa propre solution. A sa suite, Saul Amarel, Richard E. Korf et bien d'autres confirment l'intérêt de la reformulation pour la résolution de problèmes.

On peut également utiliser des techniques de reformulations dans une autre intention. Par exemple, on peut souhaiter reformuler un programme pour diminuer son temps d'exécution et l'on pratique alors une optimisation du programme. On peut également vouloir reformuler un énoncé de problème, ou sa résolution, dans un but pédagogique pour en faciliter la compréhension par un être humain.

2. Le problème du tournoi d'échecs

Ce problème fait partie des problèmes simples que l'on peut rencontrer régulièrement dans les articles d'intelligence artificielle. Son énoncé est le suivant :

Trois personnes disputent entres elles un tournoi d'échecs. Chacune joue 7 parties contre chacune des deux autres. A l'issu du tournoi, l'un des joueurs dit : « Je suis content car j'ai gagné le plus de partie ». Un autre joueur dit : « Je suis content car j'ai perdu le moins de partie ». Enfin, le dernier joueur dit « Je suis content car j'ai gagné le tournoi ». Sachant que les échecs sont un jeu à 2 joueurs qui se termine soit sur une victoire d'un des joueurs et une défaite de l'autre, soit sur un match nul entre les deux adversaires, et sachant également que le vainqueur d'un tournoi est le joueur ayant le plus de points, une victoire valant 1 point et une partie nulle 0,5 point, quelles sont les nombres de victoires, parties nulles et défaites de chacun des joueurs qui ont participé à ce tournoi ?

2.1. Formalisation isomorphe du texte de l'énoncé

La première étape de notre travail sur ce problème va consister en la formalisation du texte de l'énoncé en langage naturelle. Nous allons nous efforcer de présenter une formalisation isomorphe au texte de l'énoncé, c'est-à-dire qui contienne les mêmes informations, et avec la même organisation. En effet, nous avons tous tendance lors d'une étape de formalisation à effectuer des reformulations, souvent sans nous en rendre compte.

Trois personnes disputent entre elles un tournoi d'échecs.
Nous avons déjà une première information sur les données que nous aurons à manipuler lors de la résolution du problème : il nous faut en effet déclarer ces 3 personnes dans notre énoncé.

2.1.i. 1ère digression : la nature des données

Avant d'attribuer un type à nos données, il nous faut d'abord donner quelques explications sur la nature des données. Une donnée contient tout d'abord toujours une information d'identité. Cette information permet de distinguer deux données. Par exemple, cette information nous permet d'affirmer que A = A ou $1 \neq 3$. Les données qui ne comportent que cette information sont appelées des symboles.
Une données peut également porter une information d'ordre. Cette information est associée aux opérateurs de comparaison d'ordre, « > » et « < ». Ainsi, les nombres comportent-ils également cette information : $1 < 3$. Les lettres, si on leur attribue l'existence d'un ordre alphabétique, comportent également cette information : FE < FY.
Une données contient donc toujours une information d'identité, et parfois une information d'ordre. Nous pouvons reformuler facilement la nature d'une donnée si la donnée d'origine et celle obtenue à l'issue de la reformulation comportent les mêmes informations. Ainsi, on peut transformer des lettres en nombres, ou vice-versa, mais pas des lettres en symboles.

Nous pouvons maintenant discuter de la nature de donnée à attribuer aux joueurs de notre tournoi d'échecs. Comme nous n'avons pas de raison de leur attribuer un ordre, seule l'information d'identité sera retenue : nous allons donc déclarer les joueurs sous forme de symboles.
Reste à choisir la structure organisatrice de ces symboles.

2.1.ii. 2ème digression : la structure des données

La façon dont les données sont organisées est également porteuse d'information. Il est donc important d'y prêter attention lorsqu'on formalise un énoncé. Nous définissons les structures des données en fonction de 2 critères : leur degré de répétition maximal et leur degré d'ordre. La description complète de ces 2 critères n'étant pas l'objet de cet article, nous allons simplifier leur usage en groupant les structures selon 2 axes : avec ou sans répétition et avec ou sans ordre interne. Une structure avec répétition peut contenir plusieurs fois le même éléments, et le nombre d'occurrences de cet élément est donc important. Une structure avec ordre possède une relation d'ordre implicite. Cette relation d'ordre est indépendante de l'information d'ordre éventuellement portée par la données elle-même. Ainsi, la liste 1.8.2.42.12 définit un ordre entre les nombres qu'elles contient, indépendamment de l'ordre inhérent aux nombres en général.

Le tableau ci-dessous définit les termes choisis pour les 4 cas possibles :

	Sans ordre	Avec ordre
Sans Répétition	ensemble (set)	collier (necklace)
Avec Répétition	sac (bag)	liste (list)

Quelle structure choisir pour nos joueurs d'échecs ? L'ordre ne nous paraît pas important, et nous ne voulons pas autoriser de répétition. Nous allons donc utiliser un ensemble. Nous pouvons donc donner la déclaration des joueurs :

```
PLAYERS is set of symbols
card(PLAYERS) = 3
```

Chacune joue 7 parties contre chacune des deux autres.
Nous allons devoir maintenant formaliser la notion de partie. Plusieurs choix de formalisation sont possibles. Celui que nous prenons n'est pas le meilleur, mais le but de notre papier est justement de montrer comment on peut obtenir une meilleure formalisation par l'emploi des reformulations.
Nous allons donc considérer qu'une partie est une relation faisant intervenir 2 joueurs à un instant donné du tournoi. Le vainqueur d'une partie sera soit un joueur, soit un symbole signifiant que la partie est nulle.

```
GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → PLAYERS ∪ set of symbol =
{◆DRAW}
defined ① ≠ ②
```

Cette dernière ligne signifie simplement que la relation n'a de sens que lorsque le premier et le deuxième paramètre sont différents. Il faut également noter que le caractère ◆ sert à désigner un symbole.
Il nous reste à rajouter des propriétés de cette relation : tout d'abord le joueur vainqueur doit avoir participé à la partie :

```
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}
```

Cette propriété est peu naturelle à concevoir et nous verrons plus tard comment nous aurions pu l'écrire autrement. Cependant, ce qui est important, c'est qu'un système puisse à partir de cette forme obtenir une meilleure formulation grâce à des reformulations.
Enfin, on peut ajouter pour terminer la formalisation de la notion de partie que, puisque nous désignons le vainqueur d'un affrontement entre 2 joueurs, la fonction est symétrique, c'est-à-dire que le vainqueur de la partie entre A et B est le même que le vainqueur de la partie entre B et A.

```
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = GAMES(y,x,z)
```

Il nous reste à déclarer l'ensemble GAME_NUMBER que nous utilisons dans la relation GAMES, et à indiquer que chaque joueur affronte 7 fois chacun des 2 autres.

```
GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}
```

Nous avons défini les instants des parties comme un collier de symboles. D'autres choix étaient également valables. Cependant, tous ces choix sont équivalents à une reformulation près que nous expliquerons dans un article ultérieur sur les structures de données.

Nous avons donc à ce stade l'énoncé suivant :

```
PLAYERS is set of symbols
card(PLAYERS) = 3
GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}
GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → PLAYERS ∪ set of symbol =
{◆DRAW}
defined ① ≠ ②
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = GAMES(y,x,z)
```

l'un des joueurs dit : « Je suis content, car j'ai gagné le plus de parties »
Tout d'abord, il faut noter que nous n'allons pas formaliser la notion de contentement du joueur. Nous aurions pu le faire, mais nous allons montrer plus loin qu'une notion inutile à la résolution du problème peut être détectée et supprimée simplement. Nous allons donc pour l'instant définir la notion de nombre de parties gagnées, et formaliser l'affirmation de ce joueur.

```
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z) = p} )
```

Le nombre de parties gagnées par un joueur correspond au cardinal de l'ensemble des parties ayant pour vainqueur ce joueur.

```
∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
```

Nous introduisons ici un nouveau symbole pour parler du joueur. Ce symbole est obligatoirement rattaché aux symboles des joueurs, sans qu'il soit nécessaire de modifier la définition initiale de PLAYERS. En effet, il est aisé à un système automatique de parcourir le texte pour compléter la définition initiale.

En suivant le même principe, nous introduisons les relations DRAWS, LOSSES et SCORE pour finir de formaliser le problème. La formalisation du but est quant à elle aisée et nous la laisserons de côté par souci de clarté. Nous obtenons donc notre formalisation initiale :

```
PLAYERS is set of symbols
card(PLAYERS) = 3
GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}
GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → PLAYERS ∪ set of symbol =
{◆DRAW}
defined ① ≠ ②
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = GAMES(y,x,z)
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z) = p})
DRAWS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, DRAWS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ;
      GAMES(p,y,z) = ◆DRAW})

LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z) = y})
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)

∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)
```

2.2. 1er exemple de reformulation : référencement des paramètres

La relation GAMES est ce que nous appelons une fonction de pseudosélection. Une explication s'impose donc : une fonction de sélection est une fonction qui a pour résultat la valeur d'un de ses paramètres. Par exemple, la fonction

$$f : \quad A \times A \times B \rightarrow A$$
$$(x,y,z) \rightarrow \quad x \,\Re\, z \Rightarrow x \text{ sinon } y$$

Les seuls résultats possibles de cette fonction sont x ou y, qui sont tous deux des paramètres de la fonction. On l'appelle fonction de sélection parce qu'elle sélectionne l'un des paramètres.
De façon similaire, une fonction de pseudosélection a pour résultat soit l'un de ses paramètres, soit une valeur prise parmi un ensemble de même nature que le paramètre à sélectionner.

Par exemple, f : {1 ;2 ;3} × {1 ;2 ;3 ;4 ;5} → {0 ;1 ;2 ;3}
(x,y) → x < y ⇒ x
sinon 0

GAMES ayant pour résultat soit la valeur d'un de ses deux premiers paramètres, soit le symbole ◆ DRAW, il s'agit d'une fonction de pseudosélection.
Pour une telle fonction, il est intéressant de la reformuler de façon à ce qu'elle ait pour résultat un renvoi sur un paramètre plutôt que leur valeur effective. C'est-à-dire dans l'exemple de GAMES, que cette fonction ait pour résultat soit ◆DRAW, soit ◆① soit ◆② , ces deux derniers symboles se substituant aux valeurs effectives des paramètres pour signifier respectivement 'le premier paramètre' ou 'le deuxième paramètre'
L'intérêt de cette reformulation est de faire disparaître la propriété qui indiquait que la fonction était une fonction de pseudosélection, à savoir :
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}
et également de construire un ensemble d'arrivée plus spécifique à la fonction GAMES.

GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → PLAYERS ∪ set of symbol = {◆DRAW}
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}

Ces 2 lignes sont suffisantes et nécessaires pour détecter la propriété de pseudosélection de la fonction GAMES. La reformulation s'opère en plusieurs temps :

nous déclarons un nouvel ensemble d'arrivée pour la fonction :

GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}

nous substituons ce nouvel ensemble d'arrivée à celui employé précédemment, et la déclaration de la fonction devient donc :

GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → GAMES_RESULT

nous supprimons la propriété déclarant la pseudosélection :

~~∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}~~

nous modifions tous les appels faits à la fonction. Le plus gros travail à faire est de modifier la propriété de symétrie :
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = GAMES(y,x,z)
Pour ce faire, il nous faut décomposer les cas possibles :

GAMES(x,y,z) = ◆① ⇔ GAMES(y,x,z) = ◆②
GAMES(x,y,z) = ◆② ⇔ GAMES(y,x,z) = ◆①
GAMES(x,y,z) ∈ {◆DRAW} ⇔ GAMES(y,x,z) ∈ {◆DRAW}

Comme on peut substituer x par y et y par x, on peut finalement supprimer l'une des deux premières de ces lignes.

Au final, nous obtenons donc désormais :

PLAYERS is set of symbols
card(PLAYERS) = 3
GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS × PLAYERS × GAME_NUMBER → GAMES_RESULT
defined ① ≠ ②
~~∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z)∈ {x ; y ; ◆DRAW}~~
~~∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = GAMES(y,x,z)~~
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = ◆① ⇔ GAMES(y,x,z) = ◆②

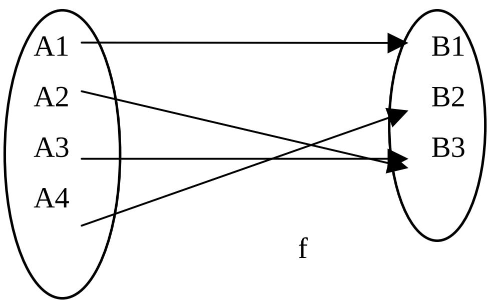
```
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, GAMES(x,y,z) = ◆DRAW ⇔ GAMES(y,x,z) =
◆DRAW
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z) = ◆① })
DRAWS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, DRAWS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ;
      GAMES(p,y,z) = ◆DRAW})

LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z) = ◆② })
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)

∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)
```

2.3. 2ème reformulation : transfert de qualification

Nous allons maintenant montrer une reformulation à rendement négatif, c'est-à-dire que le texte obtenu après cette reformulation sera moins satisfaisant par rapport au but que nous nous fixons, en l'occurrence optimiser la résolution du problème. Une telle reformulation ne peut se faire que si une autre reformulation ne peut s'appliquer sans avoir auparavant appliqué la première. On appelle cette reformulation à rendement négatif une reformulation esclave. Ce type de reformulations est amené à disparaître dans la suite de nos travaux car elles impliquent une planification des reformulations au niveau méta plus complexe que si nous travaillons uniquement avec des reformulations commutatives à rendement positif.

Les principes guidant cette reformulation sont bien connus : une relation fait intervenir des paramètres issus d'un ensemble d'ensembles de départ auxquels elle fait correspondre un résultat dans l'ensemble des ensembles d'arrivée. On peut exprimer la même information en utilisant une fonction booléenne prenant pour ensemble d'ensembles de départ l'union des ensembles de départ et d'arrivée de la relation initiale. Pour chaque n-uplet de paramètres de cette fonction booléenne on associe une valeur de vérité qui indique s'il existe un lien entre eux dans la relation.

$f : \{A1; A2; A3; A4\} \rightarrow \{B1; B2; B3\}$



On peut remplacer la relation f par la relation g :

$g : \{A1; A2; A3; A4\} \times \{B1; B2; B3\} \rightarrow \{Vrai; Faux\}$
$(a,b) \rightarrow f(a)=b$

Nous allons opérer la même méthode sur notre exemple du tournoi d'échecs. La fonction GAMES va ainsi être transformée en une relation booléenne.
Le mode opératoire de la reformulation va être le suivant :
Nous transférons l'ensemble d'arrivée de GAMES dans son ensemble d'ensembles de départ, et nous donnons comme nouvel ensemble d'arrivée l'ensemble des booléens (c'est-à-dire les symboles prédéfinis ◆VRAI et ◆FAUX)

Nous ajoutons une propriété de conservation : comme chaque triplet de la relation précédente avait un seul résultat dans GAMES_RESULT, il ne peut exister qu'un quadruplet de la nouvelle relation construit sur un triplet de l'ancienne relation qui ait un résultat vrai. En clair, il n'existe qu'un lien entre un triplet issu de PLAYERS × PLAYERS × GAME_NUMBER et un élément de GAMES_RESULT.

Nous modifions ensuite toutes les occurrences des paramètres de la fonction pour tenir compte du nouveau paramètre.

Nous obtenons au final le texte suivant :

```
PLAYERS is set of symbols
card(PLAYERS) = 3
GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS × PLAYERS × GAME_NUMBER × GAMES_RESULT → set of all
booleans
defined ① ≠ ②
∀(x,y,z) ∈ PLAYERS × PLAYERS × GAME_NUMBER, card({r}∈GAMES_RESULT ;
GAMES(x,y,z,r)=true)=1
∀(x,y,z,r) ∈ PLAYERS × PLAYERS × GAME_NUMBER × GAMES_RESULT, GAMES(x,y,z,◆①)=◆TRUE
⇔ GAMES(y,x,z,◆②)=◆TRUE
∀(x,y,z,r) ∈ PLAYERS × PLAYERS × GAME_NUMBER × GAMES_RESULT,
GAMES(x,y,z,◆DRAW)=◆TRUE ⇔ GAMES(y,x,z,◆DRAW)=◆TRUE
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z,◆①) =
◆TRUE })
DRAWS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, DRAWS(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ;
      GAMES(p,y,z,◆DRAW) = ◆TRUE})
LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= card( {(y,z) ∈ PLAYERS × GAME_NUMBER ; GAMES(p,y,z,◆②) =
◆TRUE })
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)

∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)
```

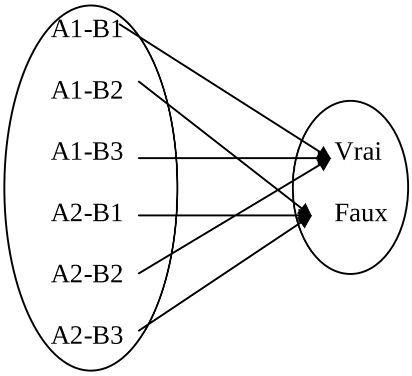
2.4. 3ème reformulation : suppression d'un caractère qualificatif de dénombrement

Nous pouvons maintenant appliquer une nouvelle reformulation. Nous allons supprimer une donnée inutile, en l'occurrence GAME_NUMBER dont le rôle ne consiste qu'à assurer un comptage des parties. Ce paramètre ne sert en effet qu'à avoir plusieurs résultats possibles pour un couple de joueurs donné. Comme nous ne pouvons pour l'instant supprimer de tels paramètres que pour les fonctions booléennes, il devient clair que la reformulation précédente était nécessaire.
Le mode opératoire de la reformulation est cette fois le suivant :

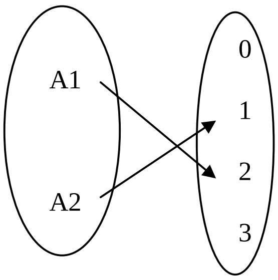
Nous retirons de l'énoncé la déclaration de la donnée à supprimer.

Nous retirons également toutes les occurrences de cette donnée dans les déclarations de relations et leurs usages ultérieurs. Cependant, il faut faire attention à adapter les résultats des fonctions. Puisqu'on supprime le caractère de dénombrement, il va falloir conserver l'information ailleurs. Ceci sera effectué en remplaçant l'ensemble d'arrivée booléen des fonctions modifiées et en le remplaçant par l'ensemble des nombres cardinaux de 0 à cardinal de la donnée supprimée. Sémantiquement, cette opération correspond à compter le nombre de situations vraies pour un n-uplet constitué par les autres paramètres de la relation.

Par exemple, considérons une fonction f : A × B → set of all booleans
Si B est simplement un paramètre de dénombrement, son rôle ne consiste qu'à compter pour chaque valeur de A le nombre de lien Vrai.



On peut donc remplacer cette formulation tout en conservant la même information par :



Puisque la seule information que nous voulions conserver était le nombre de liens vrais pour chaque occurrence de A, cette forme permet de garder cette information et de la rendre plus accessible.

Reprenons donc notre texte et appliquons ces principes :

PLAYERS is set of symbols

card(PLAYERS) = 3

~~GAME_NUMBER is necklace of symbols = {◆1 ; ◆2 ; ◆3 ; ◆4 ; ◆5 ; ◆6 ; ◆7}~~

GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}

GAMES is function PLAYERS × PLAYERS ~~× GAME_NUMBER~~ × GAMES_RESULT → set of cardinals from 0 to 7

defined ① ≠ ②

∀(x,y~~,z~~) ∈ PLAYERS × PLAYERS ~~× GAME_NUMBER~~, $\sum_{r \in GAMES\_RESULT} GAMES(x,y,r) = 7$

∀(x,y~~,z~~,r) ∈ PLAYERS × PLAYERS ~~× GAME_NUMBER~~ × GAMES_RESULT, GAMES(x,y~~,z~~,◆①) = GAMES(y,x~~,z~~,◆②)

∀(x,y~~,z~~,r) ∈ PLAYERS × PLAYERS ~~× GAME_NUMBER~~ × GAMES_RESULT, GAMES(x,y~~,z~~,◆DRAW) = GAMES(y,x~~,z~~,◆DRAW)

WINS is function PLAYERS → set of all cardinals

∀p ∈ PLAYERS, WINS(p)= $\sum_{y \in PLAYERS}$ GAMES(p,y~~,z~~,◆①)

DRAWS is function PLAYERS $\rightarrow$ set of all cardinals

$$\forall p \in \text{PLAYERS},\ \text{DRAWS}(p)=\sum_{y\in PLAYERS} \text{GAMES}(p,y\cancel{,z},\blacklozenge\text{DRAW})$$

LOSSES is function PLAYERS $\rightarrow$ set of all cardinals

$$\forall p \in \text{PLAYERS},\ \text{LOSSES}(p)=\sum_{y\in PLAYERS} \text{GAMES}(p,y\cancel{,z},\blacklozenge②)$$

SCORE is function PLAYERS $\rightarrow$ set of all reals

$\forall$p $\in$ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)

$\forall$p $\in$ PLAYERS, p $\neq$ ◆A, WINS(◆A)>WINS(p)

$\forall$p $\in$ PLAYERS, p $\neq$ ◆B, LOSSES(◆B)<LOSSES(p)

$\forall$p $\in$ PLAYERS, p $\neq$ ◆C, SCORE(◆C)>SCORE(p)

Une explication s'impose pour les modifications apportées à la ligne :

$\forall$(x,y,z) $\in$ PLAYERS $\times$ PLAYERS $\times$ GAME_NUMBER, card({r}$\in$GAMES_RESULT ; GAMES(x,y,z,r)=true)=1

Cette ligne avait pour signification : quel que soit le couple de joueurs, ils n'ont joué qu'une fois ensemble à un instant donné. La nouvelle formulation signifie donc que quel que soit le couple de joueurs considérés, ils ont joué ensemble 7 fois, car 7 est le cardinal de la donnée de dénombrement supprimée multiplié par 1 qui est le nombre de fois que ces joueurs ont disputé un match à un instant donné.

2.5. 4ème reformulation : suppression d'une inconnue entièrement définie en fonction d'autres inconnues.

Nous avons à ce stade la propriété $\forall$(x,y) $\in$ PLAYERS $\times$ PLAYERS, $\sum_{r\in GAMES\_RESULT} GAMES(x,y,r) = 7$.

De plus, comme GAMES_RESULT = {◆① ; ◆② ; ◆DRAW}, on a :

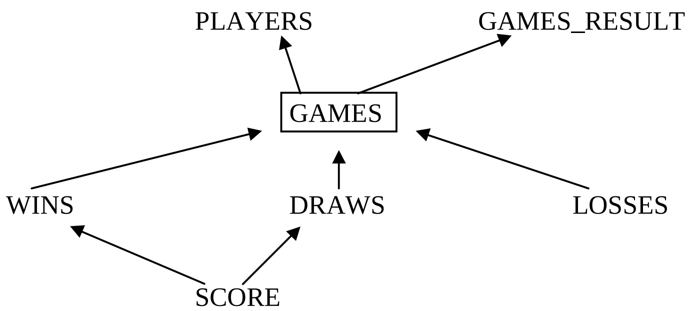
$\forall$(x,y) $\in$ PLAYERS $\times$ PLAYERS, GAMES(x,y,◆①) + GAMES(x,y,◆②) + GAMES(x,y,◆DRAW) = 7

ce qui peut également s'écrire :

$\forall$(x) $\in$ PLAYERS, WINS(x) + DRAWS(x) + LOSSES(x) = 7

Nous allons donc pouvoir réécrire l'une de ces 3 fonctions comme étant dépendante des deux autres, ce qui diminuera d'autant les recherches à effectuer lors de la résolution du problème.

Comment choisir l'inconnue à écrire en fonction des 2 autres ? si l'on trace un schéma de dépendance des données et des relations pour le problème que nous étudions, nous obtenons le schéma suivant :



Il apparaît alors que la réécriture de LOSSES est celle qui entraînera le moins de modification car aucune relation ne dépend d'elle, contrairement à WINS et DRAWS.

On peut donc supprimer la définition de LOSSES et la remplacer dans le texte par LOSSES(x) = 7 - GAGNES(x) - NULS(x)

PLAYERS is set of symbols
card(PLAYERS) = 3
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS × PLAYERS × GAMES_RESULT → set of cardinals from 0 to 7
defined ① ≠ ②
∀(x,y) ∈ PLAYERS × PLAYERS, $\sum_{r \in GAMES\_RESULT} GAMES(x,y,r) = 7$
∀(x,y,r) ∈ PLAYERS × PLAYERS × GAMES_RESULT, GAMES(x,y,◆①)= GAMES(y,x,◆②)
∀(x,y,r)∈PLAYERS×PLAYERS×GAMES_RESULT,GAMES(x,y,◆DRAW) = GAMES(y,x,◆DRAW)
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= $\sum_{y \in PLAYERS}$ GAMES(p,y,◆①)
DRAWS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, DRAWS(p)= $\sum_{y \in PLAYERS}$ GAMES(p,y,◆DRAW)
LOSSES is function PLAYERS → set of all cardinals
~~∀p ∈ PLAYERS, LOSSES(p)= $\sum_{y \in PLAYERS}$ GAMES(p,y,◆②)~~
∀p ∈ PLAYERS, LOSSES(p)= 7 - WINS(p) - DRAWS(p)
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)
∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)

LOSSES(p) sera toujours à valeur entre 0 et 7. En effet, si LOSSES(p) était inférieur à 7, cela signifierait que WINS(p)+DRAWS(p) serait supérieur à 7, ce qui contredirait le fait que la somme de GAMES doit rester égale à 7.

2.6. 5ème reformulation : reformulation des fonctions à symétrie symbolique.

Soit une fonction de la forme $f(x,y,\alpha)$ où x et y sont 2 paramètres de même type, et où α représente un ensemble quelconque de paramètres. Cette fonction est dite symétrique si elle présente la propriété :

$$\forall(x,y)\ f(x,y,\alpha)=f(y,x,\alpha)$$

Soit maintenant une fonction de la forme $f(x,y,\alpha,s)$ où x et y sont deux paramètres du même type, α un ensemble quelconque de paramètres et s un paramètre de type symbole issu d'un ensemble de départ D_S. La fonction f sera dite fonction à symétrie symbolique si on a un ensemble de propriétés $f(x,y,\alpha,s_1)=f(y,x,\alpha,s_2)$ tel que :

$\{s_1\}\cup\{s_2\}=D_S$

et card($\{s_1\}\cap\{s_2\}$)=0
ou $\forall s_i \in \{s_1\}\cap\{s_2\}$, on a la propriété $f(x,y,\alpha,s_i)=f(y,x,\alpha,s_i)$)

Plus clairement, une fonction est à symétrie symbolique si elle est symétrique en excluant le symbole représenté par s, et si ce symbole a un comportement symétrique lié à la symétrie de la fonction, ou indépendant de la symétrie de la fonction en certains points neutres appelés centres de la symétrie symbolique.

Dans le texte que nous traitons, la fonction GAMES est une fonction symbolique symétrique, comme l'indique les 2 propriétés :

∀(x,y,r) ∈ PLAYERS × PLAYERS × GAMES_RESULT, GAMES(x,y,◆①)= GAMES(y,x,◆②)
∀(x,y,r)∈PLAYERS×PLAYERS×GAMES_RESULT,GAMES(x,y,◆DRAW) = GAMES(y,x,◆DRAW)

Nous allons reformuler cette relation de façon à tirer parti de la symétrie. Imaginons une fonction f(x,y) qui soit une fonction symétrique. Plutôt que de représenter tous les couples (x,y) possibles, il suffit de représenter les couples (x,y) avec x ≤ y. Le même principe est applicable ici, et nous allons donc modifier le texte en conséquence pour obtenir :

PLAYERS is set of symbols
card(PLAYERS) = 3
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS × PLAYERS × GAMES_RESULT → set of cardinals from 0 to 7
defined ① < ②
∀(x,y) ∈ PLAYERS × PLAYERS; x < y, $\sum_{r \in GAMES\_RESULT} GAMES(x,y,r) = 7$
~~∀(x,y,r) ∈ PLAYERS × PLAYERS × GAMES_RESULT, GAMES(x,y,◆①)= GAMES(y,x,◆②)~~
~~∀(x,y,r)∈PLAYERS×PLAYERS×GAMES_RESULT,GAMES(x,y,◆DRAW) = GAMES(y,x,◆DRAW)~~
WINS is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, WINS(p)= $\sum_{y \in PLAYERS(y>p)}$ GAMES(p,y,◆①) + $\sum_{x \in PLAYERS(x<p)}$ GAMES(x,p,◆②)
DRAWS is function PLAYERS → set of all cardinals
∀p∈PLAYERS, DRAWS(p)= $\sum_{y \in PLAYERS(y>p)}$ GAMES(p,y,◆DRAW)+ $\sum_{x \in PLAYERS(x<p)}$ GAMES(x,p,◆DRAW)
LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= 7 - WINS(p) - DRAWS(p)
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)
∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)

2.7. 6ème reformulation : accrétion de variables

Nous allons maintenant procéder à une accrétion de variables pour la relation GAMES. Ce procédé est également appelé passage à la forme duale. Le principe en est de remplacer un couple de variables de même type par une variable unique représentant ce couple.
Cette méthode utilise des fonctions de conversion pour passer d’un couple donné vers sa valeur dans la nouvelle variable et réciproquement. L’usage de l’accrétion de variables peut alourdir considérablement le texte de l’énoncé pour un humain, mais simplifie sa manipulation pour un système informatique.
Dans notre exemple, l’intérêt de cette reformulation va être d’intégrer le gain d’information obtenu précédemment par le traitement des propriétés de symétrie dans une variable unique tenant compte des contraintes exprimées. C’est-à-dire que nous allons conserver une quantité d’information strictement identique tout en supprimant les tests d’existence comme par exemple ①<② qui sera traité au niveau de la conversion en couples. Ainsi, une partie de la complexité du problème est traité une fois pour toutes avant la résolution à proprement parler.

Dans notre exemple, nous avons 3 couples de PLAYERS × PLAYERS autorisés comme paramètre de la fonction GAMES. Le passage à la forme duale va permettre d’avoir réellement une variable à 3 valeurs, au lieu d’un tableau 3 par 3 restreint à 3 de ses valeurs.

PLAYERS is set of symbols
card(PLAYERS) = 3
PLAYERS_COUPLES is set of 3 symbols = {◆C1 ; ◆C2 ; ◆C3}
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS_COUPLES × GAMES_RESULT → set of cardinals from 0 to 7
defined always
∀(x) ∈ PLAYERS_COUPLES, $\sum_{r \in GAMES\_RESULT} GAMES(x,r) = 7$
WINS is function PLAYERS → set of all cardinals

~~∀p ∈ PLAYERS, WINS(p)= $\sum_{y \in PLAYERS(y>p)}$ GAMES(p,y,◆①) + $\sum_{x \in PLAYERS(x<p)}$ GAMES(x,p,◆②)~~
WINS(◆A) = GAMES(◆C1,◆①) + GAMES(◆C2,◆①)
WINS(◆B) = GAMES(◆C3,◆①) + GAMES(◆C1,◆②)
WINS(◆C) = GAMES(◆C2,◆②) + GAMES(◆C3,◆②)
DRAWS is function PLAYERS → set of all cardinals
~~∀p∈PLAYERS, DRAWS(p)= $\sum_{y \in PLAYERS(y>p)}$ GAMES(p,y,◆DRAW)+ $\sum_{x \in PLAYERS(x<p)}$ GAMES(x,p,◆DRAW)~~
DRAWS(◆A) = GAMES(◆C1,◆DRAW) + GAMES(◆C2,◆DRAW)
DRAWS(◆B) = GAMES(◆C3,◆DRAW) + GAMES(◆C1,◆DRAW)
DRAWS(◆C) = GAMES(◆C2,◆DRAW) + GAMES(◆C3,◆DRAW)
LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= 7 - WINS(p) - DRAWS(p)
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)
∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)

Une remarque s'impose suit à cette transformation. On remarque que l'emploi de ◆① a été conservé alors que l'on n'a plus affaire à un couple de variables. C'est l'intérêt d'avoir représenté le concept « premier paramètre » sous forme d'un symbole. On peut maintenant le manipuler comme tel sans avoir à se soucier de la sémantique initiale. La sémantique globale du problème est conservée, pas forcément celle attachée à chacun des symboles.
On voit également l'intérêt d'avoir un système pouvant faire automatiquement le passage à la forme duale. En effet, établir celle-ci soi-même peut s'avérer fastidieux et long à taper lorsque le nombre de valeurs possibles pour chaque variable augmente.

2.8. Forme finale

Au final, nous obtenons donc le texte suivant :

PLAYERS is set of symbols
card(PLAYERS) = 3
PLAYERS_COUPLES is set of 3 symbols = {◆C1 ; ◆C2 ; ◆C3}
GAMES_RESULT is set of symbols = {◆① ; ◆② ; ◆DRAW}
GAMES is function PLAYERS_COUPLES × GAMES_RESULT → set of cardinals from 0 to 7 defined always
∀(x) ∈ PLAYERS_COUPLES, $\sum_{r \in GAMES\_RESULT} GAMES(x,r) = 7$
WINS is function PLAYERS → set of all cardinals
WINS(◆A) = GAMES(◆C1,◆①) + GAMES(◆C2,◆①)
WINS(◆B) = GAMES(◆C3,◆①) + GAMES(◆C1,◆②)
WINS(◆C) = GAMES(◆C2,◆②) + GAMES(◆C3,◆②)
DRAWS is function PLAYERS → set of all cardinals
DRAWS(◆A) = GAMES(◆C1,◆DRAW) + GAMES(◆C2,◆DRAW)
DRAWS(◆B) = GAMES(◆C3,◆DRAW) + GAMES(◆C1,◆DRAW)
DRAWS(◆C) = GAMES(◆C2,◆DRAW) + GAMES(◆C3,◆DRAW)
LOSSES is function PLAYERS → set of all cardinals
∀p ∈ PLAYERS, LOSSES(p)= 7 - WINS(p) - DRAWS(p)
SCORE is function PLAYERS → set of all reals
∀p ∈ PLAYERS, SCORE(p) = WINS(p) + 0.5*DRAWS(p)
∀p ∈ PLAYERS, p ≠ ◆A, WINS(◆A)>WINS(p)
∀p ∈ PLAYERS, p ≠ ◆B, LOSSES(◆B)<LOSSES(p)
∀p ∈ PLAYERS, p ≠ ◆C, SCORE(◆C)>SCORE(p)

3. Conclusion

Le texte final que nous avons obtenu a été soumis (après traduction isomorphe dans le langage approprié) au système MALICE. En mode de résolution sous forme interprétée, le temps de résolution a été divisé par 4 par rapport au texte précédemment employé. Nos reformulations ont donc bien satisfait au but de l'optimisation du texte en vue de sa résolution par un système indépendant.

Le but de nos travaux, dans cet article donne un exemple, est de permettre de décharger les opérateurs de système de résolution du besoin actuel de bien connaître ce système et de reformuler soi-même par avance le problème pour optimiser sa résolution. Comme nous l'avons montré, un traitement basé sur la recherche syntaxique de propriétés du texte est suffisant pour effectuer cette opération. Contrairement à une croyance répandue, une analyse sémantique n'est pas nécessaire : il suffit de savoir où se trouve la sémantique et non de la comprendre.
Notre exemple a ainsi montré que des opérations fastidieuses peuvent être exécutées automatiquement, et qu'un système disposant des métaconnaissances nécessaires (celles-ci n'ont pas encore été établies) pourrait appliquer des méthodes de reformulation là où on ne penserait pas à les faire intervenir. Enfin, les informations données par le programmeur n'ont pas besoin d'être sélectionnées au strict nécessaire puisqu'il est possible de détecter par une analyse syntaxique quelles informations sont inintéressantes en vue de la résolution du problème.

CHOIX DE DÉPLACEMENT D'AGENTS MOBILES DANS UN ENVIRONNEMENT SPATIO-TEMPOREL

Vincent LE CERF

lecerf@inrets.fr

INRETS

2, avenue Malleret joinville

94 114 ARCUEIL Cedex

Résumé

Dans cet article nous allons nous intéresser au cœur du système Lombric [*Le Cerf 97*] constitué d'un module évaluant les possibilités de déplacements de chaque acteur présent dans la scène. Le système utilise un graphe spatio-temporel, l'historique de chaque acteur et un ensemble de règles contextuelles pour déterminer les lieux probables de déplacement. Des "conflits" de choix peuvent apparaître lorsque plusieurs acteurs sont susceptibles d'aller dans le même lieu. Les conflits entre les acteurs seront gérés par l'intermédiaire de résolveurs de conflits spécialisés selon le contexte et la complexité des conflits. Une phase de gestion des parcours des acteurs est réalisée par Lombric pour résoudre certaines incohérences et indéterminations.

Mots-clefs : Hypothèses de déplacement - Arbitrage de décision - Représentation spatio-temporelle - Niveau de connaissances - Reconstitution de scène.

1. Introduction

L'objectif de nos travaux est l'étude et le développement de méthodes permettant d'élaborer et de mettre à jour une représentation d'une scène réelle à partir de capteurs. Le système doit être capable de suivre les déplacements des acteurs dans cette scène et de s'adapter aux changements pouvant y survenir. Notre système fonctionne en permanence et doit répondre en un temps relativement court (1 seconde). L'un des domaines d'application est celui de la gestion du trafic à l'intérieur et aux abords d'un carrefour urbain, afin de répondre aux besoins opérationnels de surveillance et d'informations sur le trafic. Les données réelles sont issues du laboratoire "Carrefour Intelligent" [*Sellam 94*] (GRETIA) à l'INRETS. Un carrefour surveillé par 8 caméras fournit en continu des images du trafic urbain. Ce laboratoire nous transmet alors des données prétraitées par un logiciel de traitement d'images [*Aubert 96*].
Après une brève présentation de l'architecture de Lombric et de son mode de fonctionnement, nous étudierons comment le système reconstitue les parcours de chaque véhicule présent dans la scène réelle.

2. Reconstitution, Scène et acteurs

Les données réelles sont fournies par un logiciel de traitement *d'images* [*Blosseville 88*][*Aubert 96*] en terme d'occupation spatiale, résultant de présence d'objets animés sur la scène. A partir de ces données, l'objectif de notre système est de reconstituer le déplacement de ces agents en temps réel. Pour cela nous avons utilisé une approche symbolique permettant de prendre en compte l'incertitude des données, la représentation du temps et l'intégration de données issues simultanément de plusieurs capteurs. Cette approche est mise en œuvre en premier lieu par la construction d'un graphe spatio-temporel, périodiquement actualisé à partir des données réelles. Ensuite, chaque agent, une fois détecté en entrée de la scène observée, se déplace sur ce graphe en conformité avec les données obtenues régulièrement par les capteurs. C'est cette opération que nous allons décrire dans la suite de cet article. Nous nous arrêterons plus spécifiquement sur la manière dont Lombric choisit parmi les hypothèses de déplacement celle correspondant à chaque i-acteur présent.

☞ *Dans la suite de cet article nous utiliserons le terme de **i-acteur** pour désigner la représentation informatique d'un acteur réel.*
Nous utiliserons le terme de i-véhicule quand on voudra se référer à l'exemple de la reconstitution de scène de trafic urbain et le terme de véhicule sera réservé pour les véhicules réels.

2.1. Reconstitution

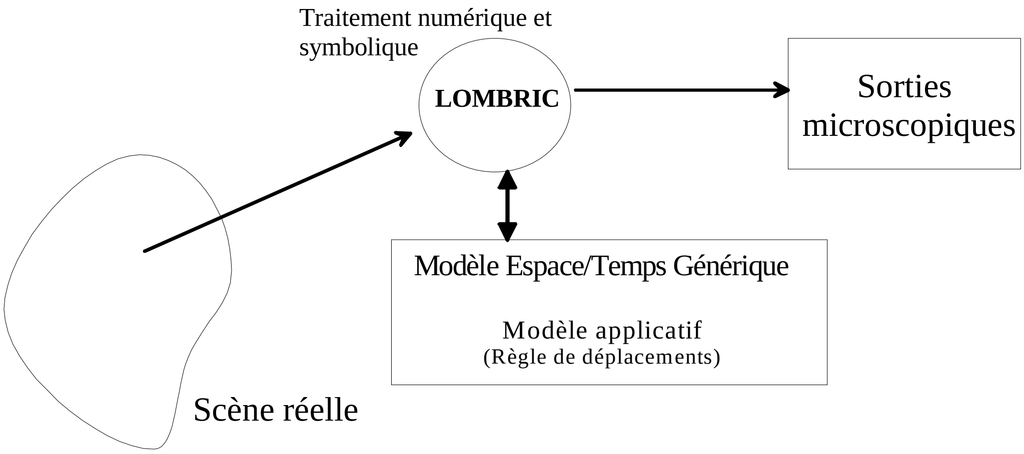


Figure 1 : L'utilisation des modèles par Lombric

Le système Lombric utilise un modèle double, comportant un modèle générique de l'espace et du temps et un modèle de la dynamique spécifique au type d'application. La difficulté est de déplacer le i-acteur (positionné sur un état-zone du graphe spatio-temporel) de manière à obtenir un itinéraire cohérent et correspondant à celui de l'acteur réel qu'il représente.

2.2. La scène observée

2.2.i. Les caractéristiques de la scène observée

Les mondes de notre étude sont les mondes dynamiques où l'espace est délimité, le temps d'étude non fini, avec un nombre variable d'acteurs dont les actions possibles sont généralement connues.
La complexité de l'étude du trafic routier est essentiellement due à deux paramètres : la variabilité du trafic accentuée par les différences entre les déplacements de chaque véhicule (vitesse, direction, sens, arrêts,...), et l'ordre d'apparition de ces événements qui n'est pas prévisible. On peut alors parler de système complexe non déterministe dans la prévision.

2.2.ii. L'observation : des capteurs géographiquement répartis

Pour analyser une scène il faut des moyens d'observation. Lors de l'acquisition d'images de scènes extérieures, de nombreuses perturbations, telles que les conditions météorologiques, les problèmes liés à la lumière (ombre, éclairage, jour, nuit), ainsi que le manque de fiabilité des capteurs vidéo [*Le Cerf 98*] perturbent les données et augmentent leurs imprécisions. Les informations issues des capteurs, peuvent être incomplètes, quelquefois manquantes, et souvent bruitées. Les données perçues issues des capteurs vidéo ne permettent pas directement de reconstituer le trafic. Il va falloir les qualifier, les interpréter, poser des hypothèses pour qu'elles correspondent à des données intelligibles (i.e. véhicules).

2.2.iii. D'une perception instantanée à une représentation spatio-temporelle continue

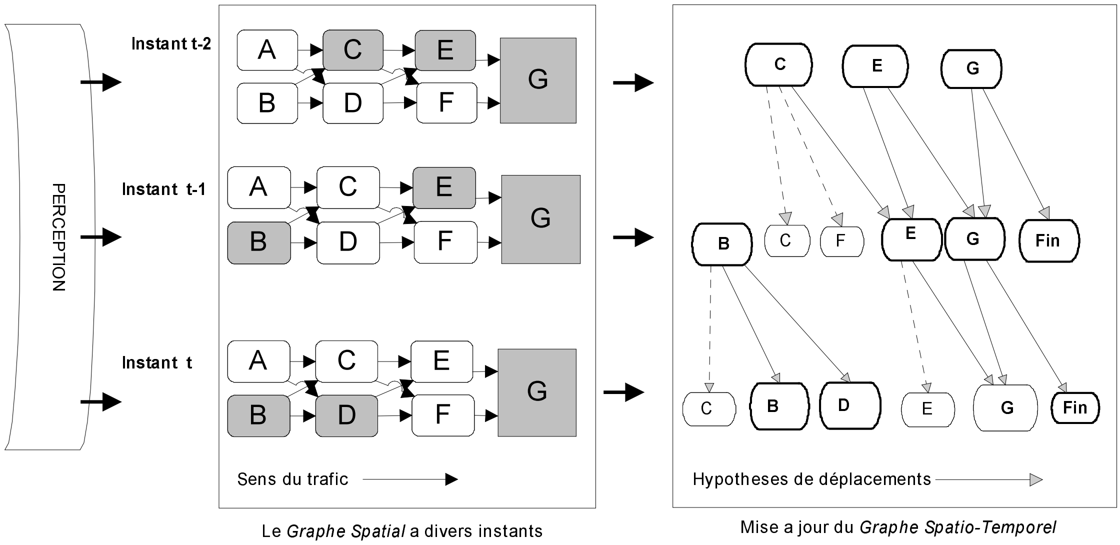


Figure 2 : Traitement de l'information à chaque cycle

L'environnement du i-acteur est un graphe spatio-temporel [*Le Cerf 97*]. Son environnement évolue donc au cours du temps et ses hypothèses de déplacement ne sont connues qu'avec quelques secondes d'avance. Après chaque construction d'un nouvel instant dans le graphe spatio-temporel, le système déplace les i-acteurs avec un retard de N secondes (N compris entre 2 et 5). Ce délai permet de simplifier le graphe spatio-temporel par propagation «arrière» des informations [*Le Cerf 98*].

2.3. Les acteurs

2.3.i. Individualisme des acteurs

Les i-acteurs sont des objets ayant un minimum de permanence dans l'espace et dans le temps. Ils sont mobiles et peuvent être déformables dans la perception (i.e. forme évolutive dans le temps selon la position et la vue de la caméra). Chaque i-acteur possède une base de connaissances qui évolue selon son contexte "environnemental" et garde un historique de son déplacement.

2.3.ii. Un modèle de déplacement

Le i-acteur est positionné à chaque instant sur un nœud etat-zone représentant l'état d'une surface au sol (appelé zone) à un instant donné. De cette manière il est situé dans l'espace et le temps. La principale connaissance construite par le i-acteur est un **parcours** appartenant à la scène observée, au cours du temps. Un parcours est un chemin élémentaire dans le graphe spatio-temporel, d'un nœud de départ *nzd* à un nœud de sortie *nzs* en passant par 1 seul nœud à chaque instant. A chaque étape du parcours (chaque nœud d'état), le i-acteur est qualifié par le système LOMBRIC par les caractéristiques suivantes :
une position (état-zone) : localisation spatiale et temporelle
une position relative aux autres i-acteurs (relation de précédence),
une qualification comportementale (arrêt, mouvement lent, mouvement rapide),
des paramètres estimés : taille, vitesse, direction et orientation.

I-acteur	Exemple
Nom (ou numéro)	i-acteur n°6
Heure de création	12:27:09
Lieu de création	Z08
Heure de disparition	12:27:49
Lieu de disparition	Z37

Parcours (liste d'étapes)

Etape 1 Z08 12:27:09
Etape 2 Z08 12:27:10
…
Etape 8 Z09 12:27:16
Etape 9 Z10 12:27:17
Etape 10Z11 12:27:18
Etape 11Z11 12:27:19
…
Etape 40Z37 12:27:48
Etape 41Z37 12:27:49

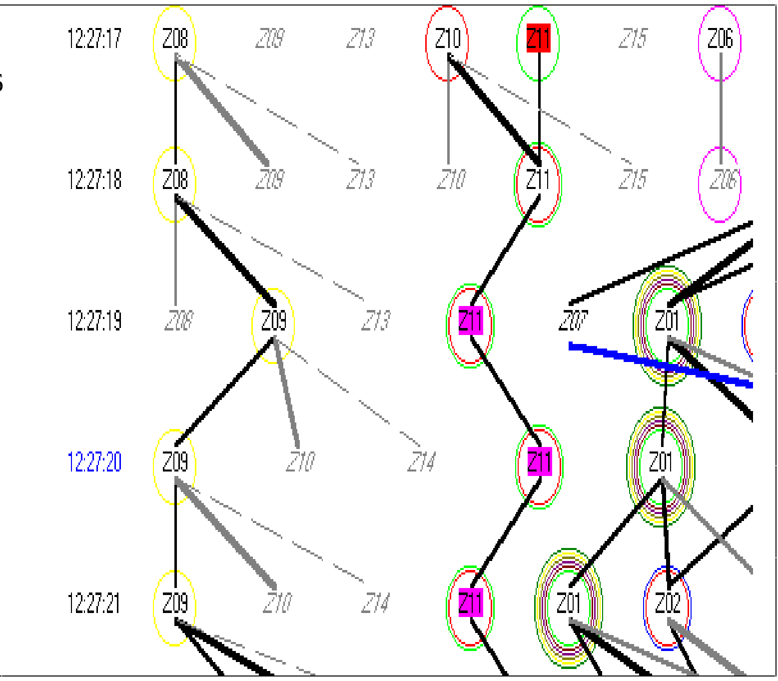


Figure 3 : Exemple de parcours d'un i-acteur

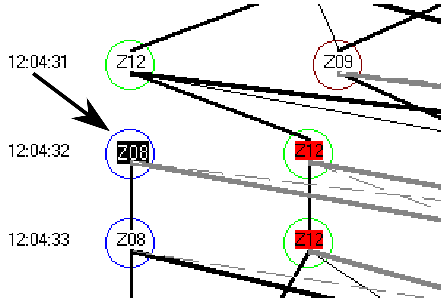
A la facette topologique des i-acteurs, s'ajoute un point de vue fonctionnel exprimé par des relations temporelles (arc du graphe spatio-temporel) d'accessibilité entre les différentes zones. Le i-acteur possède donc l'ensemble des états futurs où il peut se déplacer. Cela permet d'avoir une notion d'un déterminisme local contraignant l'ensemble des avenirs possibles de notre agent dans l'espace et le temps.
Le niveau de détails auxquels nous travaillons ne nous permet pas d'obtenir, des informations caractérisant le véhicule de manière sûr et précise. Contrairement au modèle de simulation [*Abours 88*] [*Espié 93*], nous n'avons pas, par exemple, l'estimation de la "vitesse préférentielle" du conducteur qui nous permettrait "d'asseoir" certain comportement et plus particulièrement le changement de voie [*El Hadouaj 98*].

3. Création et Destruction des acteurs de la scène

Le système LOMBRIC dirige la création et la destruction des i-acteurs dans le graphe spatio-temporel au niveau des zones d'entrées et des zones de sorties. La régulation des i-acteurs est aussi contrôlée par le module d'adaptation lors de la gestion des incohérences.

3.1. Création des i-acteurs

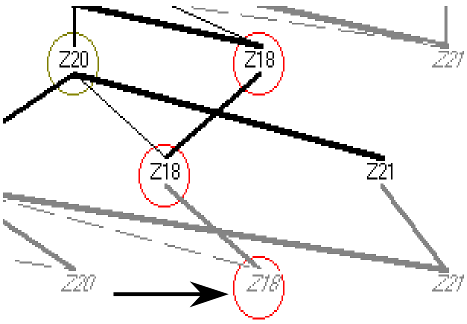
Les i-acteurs ne sont introduits dans le graphe spatio-temporel que sur les états-zones d'entrée. Si une zone d'entrée d'un tronçon d'entrée possède à un instant t, de l'occupation et qu'aucun i-acteur ne se trouve sur cette zone, alors le système crée un nouvel i-acteur.
Le respect de la contrainte liée au nombre maximum d'acteurs pouvant entrer sur une zone entre 2 cycles d'analyse régule le nombre de créations abusives. (Ex : le débit est limité à 1 i-véhicule maximum par cycle)
Critère de création
Zone proche (+3) ou est (+5) une zone d'entré
Persistance de l'occupation dans le futur
Contrainte de la capacité de la zone et des flux vérifiés
Aucun autre i-véhicule n'est présent ou très proche



Par exemple, la zone 08 est une zone du début d'un tronçon d'entrée dans le carrefour. La zone ne possède aucun i-véhicule et aucun autre i-véhicule ne peut y aller. Le système crée alors un nouveau i-véhicule.

3.2. Destruction des i-acteurs

Les i-acteurs sont supprimés généralement du graphe spatio-temporel sur les états-zones de sortie du carrefour. Si une zone de sortie est une feuille à un instant t, et si un i-acteur se trouve sur cet état-zone, alors le système va le supprimer.
Le critère de destruction d'un i-acteur est vrai quand il se positionne sur une *feuille* du graphe représentant une *zone de sortie* à un instant t. La nature du i-acteur n'a pas d'importance (réel ou virtuel)
Critère de destruction
Zone est proche (+3) ou est (+5) une zone de sortie
Etat-zone est une feuille
Non persistance de l'occupation dans le futur
Contrainte de la capacité de la zone et des flux vérifiés



Par exemple, la zone 18 est une zone de fin d'un tronçon de sortie du carrefour. Le i-véhicule n'a plus de choix dans le futur, le système le supprime du graphe Spatio-temporel.

Une destruction progressive des informations possédées par chaque i-acteur est alors déclenchée.

4. Les configurations du GST : du plus simple au plus complexe

4.1. Résolution de plusieurs problèmes dépendant

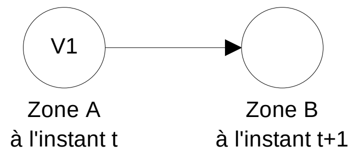
Le problème à résoudre est la reconstitution d'un itinéraire pour chaque i-acteur. La résolution d'un problème peut être vue comme le déplacement à l'intérieur d'un espace de recherche qu'on peut représenter sous la forme d'un graphe [*Nilsson 80*]. Les nœuds du graphe représentent les états que peuvent prendre successivement les i-acteurs. Les arcs représentent les actions d'une durée de 1 seconde (déplacement dans la zone suivante ou séjour dans la même zone) qui permettent de changer d'état. Dans notre cas le graphe spatio-temporel représente notre espace de recherche. Celui-ci est parcouru par plusieurs i-acteurs qui se partagent les possibilités et interagissent ensemble au cours de la résolution. Nous devons alors résoudre plusieurs problèmes dépendants les uns des autres.
Souvent lors du parcours d'arbre de décision ou de graphe d'états, le programme possède une heuristique de choix de déplacement (ou une stratégie de parcours). Dans notre cas, les heuristiques dépendent fortement des i-acteurs et de leur contexte (paramètres spatial et temporel). Les règles dans notre modèle évaluent ces heuristiques de déplacements.
Le graphe spatio-temporel propose des ressources spatiales (surfaces et transitions) et temporelles limitées. Les i-acteurs vont négocier ces ressources avec l'environnement et les autres i-acteurs. Lors de plusieurs choix de déplacements et lors de conflits avec les autres i-acteurs, le système va créer des **résolveurs de conflits spécialisés**. Nous allons voir comment les résolveurs de conflits vont résoudre les conflits existants lors du déplacement des i-acteurs.

4.2. Les cas simples

Certaines configurations n’apportent aucun conflit, ni de difficulté de choix de déplacement. C’est le cas où l'i-acteur a au plus un choix de déplacement dans le graphe spatio-temporel et n’a pas d'influence directe sur la dynamique des autres acteurs. Nous allons voir ces cas *sans conflit*.

4.2.i. L'agent ne possède qu'une possibilité de déplacement

Un i-acteur se trouve sur un état-zone (Zone A à l'intant t) et une seule solution de déplacement lui est proposée (la Zone B à l'instant t+1).

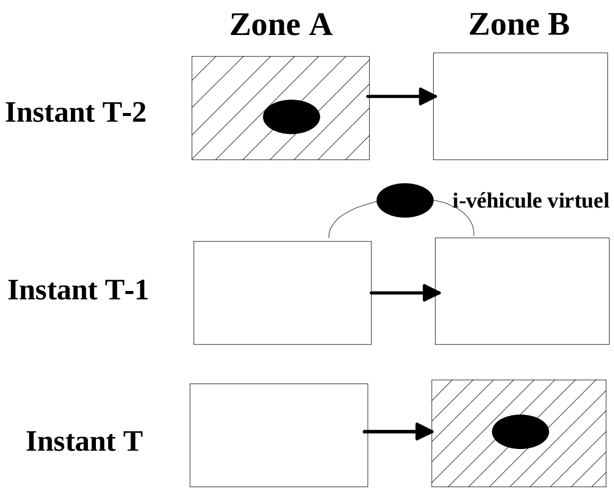


Si zone A = zone B alors le i-acteur reste sur la zone où il se trouve. Il peut soit être en arrêt, soit en mouvement.
Si zone A ≠ zone B alors le i-acteur se déplace de la zone A dans la zone suivante B.

Dans le cas où il y a plusieurs i-acteurs dans la zone A, le système LOMBRIC déplacera les i-acteurs dans la zone B en respectant l'ordre de la file d’attente existant dans la zone A.

4.2.ii. L'agent n'a pas de possibilité de déplacement

Le i-acteur se trouve alors sur un état-zone feuille dans le graphe. Cela se produit par exemple lorsqu'un véhicule n'est plus visible par les caméras ou lors de données nulles ou négligeables issues du traitement d'images. Aucun état appartenant à l'instant suivant ne lui est donc accessible. Le i-acteur devient alors **virtuel** et se met en attente.
Les i-acteurs virtuels existants ont des hypothèses de position et de déplacement mais ne peuvent pas se déplacer au regard des informations fournies par les capteurs. Chaque **i-acteur virtuel** remet en question ses hypothèses de déplacement et cherche une autre possibilité de déplacement à l'instant précédent. Notre processus de génération / vérification d'hypothèses a lieu durant la construction du graphe spatio-temporel et lors du déplacement des autres i-acteurs. Chaque i-acteur doit être cohérent avec lui-même, avec les autres i-acteurs, et avec l’environnement. Cette cohérence doit s’effectuer dans le temps. Un i-acteur peut ne pas être cohérent pendant un instant, mais peut le devenir ultérieurement.
Le système tentera de «réintroduire » les i-acteurs virtuels dans le graphe spatio-temporel. De cette manière on va pallier l’absence ou à l’inexactitude des mesures.



Le i-véhicule à l’instant T-2 est sur la zone A qui possède de l'occupation (zone rayée).
A l'instant T-1, le traitement d'image ne nous fournis pas d'occupation ni dans la zone A, ni dans la zone B. Le système Lombric rend le i-véhicule virtuel.
A l'instant T la zone B, possède de l'occupation mais n'a pas de i-véhicule "justifiant" cette occupation.
Le système Lombric introduit le i-véhicule virtuel, qui redevient réel à l'instant T dans la zone B.

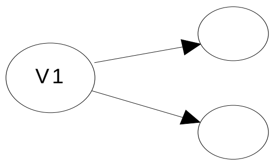
Voyons maintenant les cas plus complexe.

4.3. Définition des cas plus complexes

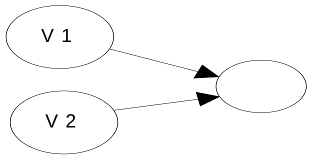
On distingue plusieurs types de cas complexes : un i-acteur a plusieurs choix de déplacement, et/ou le i-acteur se trouve en concurrence avec un autre pour un déplacement donné.

4.3.i. Plusieurs situations complexes

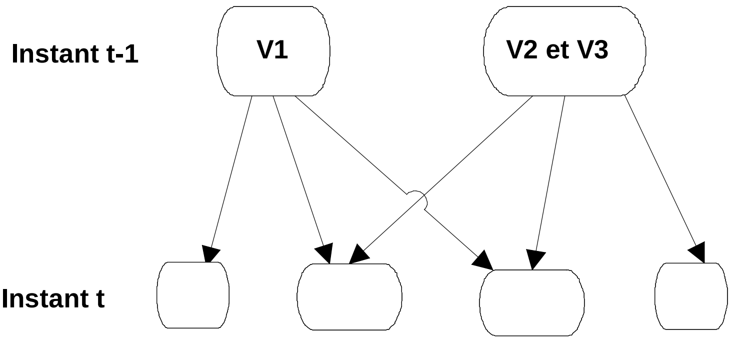
Cas A1 : Le i-véhicule possède plusieurs choix de déplacement possibles (sans concurrence avec un autre i-véhicule).



Cas A2 : Le i-véhicule est en concurrence avec d'autres i-véhicules pour entrer dans le même état-zone.



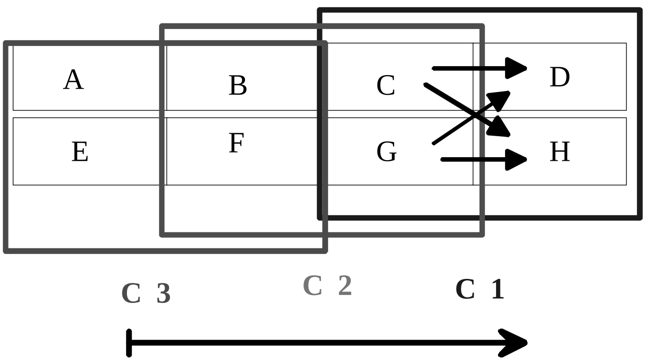
Cas A3 : Le cas le plus courant est la combinaison du cas A1 et du cas A2



Les situations présentées ci-dessous sont sensiblement les mêmes que celles présentées Par F. brémond dans [*Brémond 97*] qui cependant travaille à un niveau plus numérique.

4.3.ii. Représentation spatial des conflits

Un conflit est de véhicule ayant un ou des objectifs de déplacements en communs.



C1 : Conflits entre les i-véhicules se trouvant sur C et G et désirant se déplacer sur C, D, G ou H

Nous allons étudier comment le système LOMBRIC construit les résolveurs de conflits et l'ordre dans lequel il les active. Un superviseur va remplir ce rôle de gestionnaire de résolveurs de conflits. Celui-ci active tous les résolveurs de conflits qu'il construit toutes les secondes.

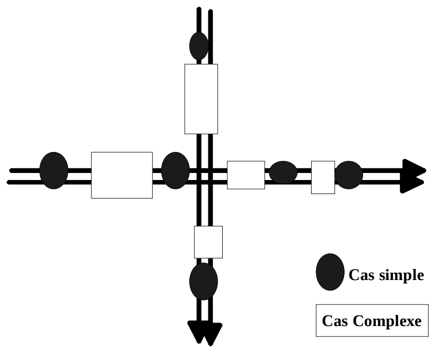
Nous allons maintenant voir comment le système résout les conflits entre i-véhicules.

5. Résolution des cas complexes

Après la description des conflits pouvant survenir lors du déplacement des i-acteurs, nous allons voir comment Lombric les résous.

5.1. Isolement des conflits par résolution progressive (du plus simple au plus complexe)

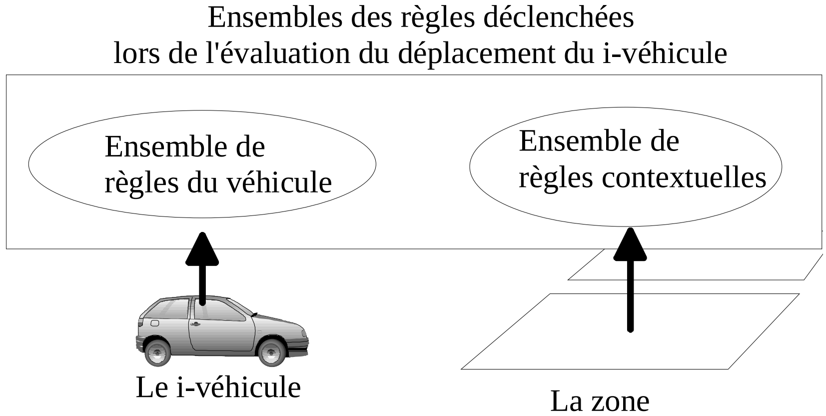
La résolution des cas simples va isoler les cas complexes qui seront alors localisés et non dépendants.



Sur ce schéma nous avons un carrefour urbain (relativement simple) où la résolution des cas simples (Rond) va isoler des zones de conflits constitués de cas complexe (Carré). Cela permet au système de traité chaque conflit de manière indépendante

5.2. Evaluation de plusieurs choix de déplacements

Pour évaluer ces choix de déplacement, le système Lombric va déclencher des règles. L'**ensemble de ces règles représente un modèle de déplacement** sous-jacent du système LOMBRIC.



Le i-véhicule récupère les règles liées à son contexte, qu'il regroupe avec ses propres règles à la manière d'une ardoise contextuelle [*Trannois 96*].

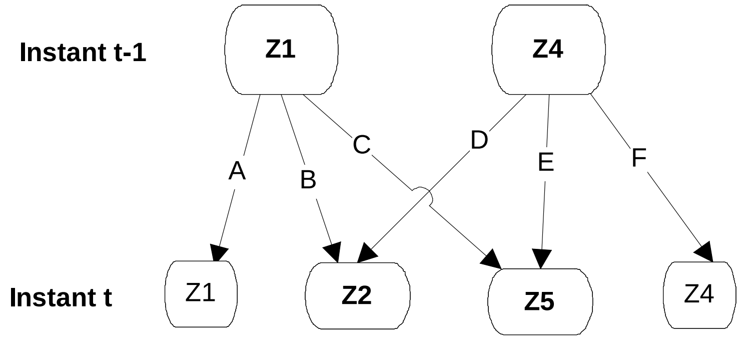
5.2.i. Quelques mots sur les règles utilisées par Lombric

Ces règles dite de modification de pondérations sont de la forme : ***Si*** *[Ensembles de prémisses]* ***Alors*** *[Incrémentation ou décrémentation d'une valeur de pondération pouvant être calculée par une fonction]*. Les prémisses n'utilisent pas les valeurs de pondération. Ces règles sont de cette manière indépendantes les unes des autres et peuvent donc être déclenchés en parallèles. Les pondérations ajoutés ou soustraites peuvent être elles-mêmes pondérées d'un indice de fiabilité, et/ou d'un indice d'inhibition. Cela permet d'inhiber l'évaluation de certaines règles lors de conditions exceptionnelles (i.e. saturation d'une suite de zones).

On distingue deux classes de règles. Les règles topologiques qui évalue la realtion de déplacement entre la position courante du i-acteur et sa position hypothétique à l'instant suivant. Les règles de la dynamique des agents tiennent compte des autres i-acteurs et des conséquences de leur déplacement sur la capacité des zones et des flux de déplacement.

5.2.ii. Pondération "topologique" des hypothèses de déplacements

Lors de la construction du graphe spatio-temporel, le système LOMBRIC pondère chaque arc reliant l'état présent aux états futurs en déclenchant des règles dites "topologiques". Ces règles utilisent les caractéristiques des zones (place libre, type de zone : entrée ou sortie,…), mais aussi des caractéristiques du graphe spatio-temporel (feuille ou nœud, pondération sur les arcs). Ces pondérations ne tiennent pas compte des i-acteurs eux-mêmes, ni de leur nombre, ni de leurs interactions. Les règles vont matcher avec certaines configurations du graphe et modifier les pondérations des arcs concernés. Ces pondérations seront ensuite utilisées lors de l'évaluation des déplacements des i-véhicules.



Entre parenthèses nous avons mis les pondérations les plus courantes données par les règles. Par exemple (-3) signifie que l'on soustrait 3 à la valeur de l'arc pour la zone concernée. Ces pondérations peuvent être modifiées selon la fiabilité des mesures affectées à la zone concernée.

Règles topologiques

Le système privilégie les nœuds (+ 3) plutôt que les feuilles (- 3). Si Z2 et Z5 sont des nœuds et Z1 et Z4 sont des feuilles, les arcs B, C, D et E seront privilégiés.

Les états dont les zones ont un taux d'occupation nul ne sont pas privilégiés (-15).

Le système privilégie les nœuds (dans les tronçons uniquement) atteint par un seul arc. Les arcs A et F sont ici privilégiés (+15)

Certaines règles privilégient les entrées ou les sorties dans des zones spécifiques :

La sortie d'une zone de conflit	(+5)
La sortie d'une zone d'entrée	(+3)
L'entrée dans une zone de sortie	(+3)

S'il y a une diminution de l'occupation de la zone de départ à l'instant d'arrivée alors l'arc réflexif est pondéré de (-15). Par exemple, si occupation de z4 à l'instant t << occupation de z4 à l'instant t-1 alors ont diminuera la valeur de l'arc F.

Si l'occupation de l'état futur est en augmentation, alors le système pondère en fonction de cette augmentation (4* hausse de l'occupation). La hausse du taux d'occupation est calculée par rapport à la taille moyenne d'un véhicule dans cette zone.

Si la zone est au centre et l'occupation reste stable et l'état futur n'est atteint que par 1 seul arc alors on augmente la valeur de cet arc (+15).

Pour les arcs réflexifs, si l'occupation et l'arrêt sont stable alors le système pondère positivement cet arc selon son remplissage actuel (+50 si tout est rempli).

Si l'occupation et l'arrêt sont stables dans l'état futur et la zone de l'état présent est différent de la zone de l'état futur alors le système pose qu'il est impossible d'entrer dans cette zone (-30)

5.2.iii. Règles de déplacement du i-acteur

Pour chaque possibilité de déplacement l'i-acteur récupère les pondérations topologiques (Cf. Paragraphe précédent). Cette valeur topologique est complétée par des pondérations issues de règles associées aux caractéristiques des i-véhicules : leurs comportements, leurs historiques et leurs positionnements. Ces règles étudient entre autre les interactions intra et inter zones avec les autres i-véhicules.

Le système décide qu'un état-zone possède de la place libre si le nombre de i-véhicule présent sur cet état est inférieur au minimum de véhicule estimé à partir du taux d'occupation spatial.

Le système va privilégier les états-zones futurs n'ayant aucun autre i-véhicule (+2).

Si l'état futur possède de la place libre alors (+20) sinon (-15)

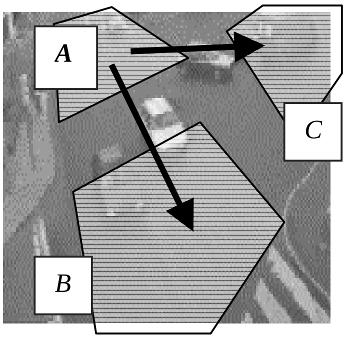
Si le débit est saturé alors le système soustrait une pondération calculée selon les débits fixé dans le graphe spatial

Si l'état futur possède de la place libre et le i-véhicule est le dernier de la file d'attente alors (+10)

Si l'on n'est pas au centre et si le i-véhicule se trouve le premier de sa file d'attente et si il reste de la place libre alors (+5) pour ce choix.

5.2.iv. Exemple d'évaluation de déplacement d'un i-véhicule

Soient le i-véhicule V1 positionné à l'instant t-1 sur la zone A. Il est confronté à un choix de déplacement (zone A, zone B ou zone C).



i-véhicule V1			
A* → A = *3		***A*→*C* = *40***	
R1 : pondération de base	**+8**	R1 bis : pondération de base	**+12**
R2 : diminution dans la même zone	**-5**	R3 : *C* est un nœud (GST)	**+3**
		R4 : Sortir d'une zone de conflit	**+5**
		R5 : *C* possède de la place libre	**+15**
		R6 : premier file d'attente	**+5**
A* → *B* = *35	**+35**		
R1 bis, R3, R5, R6			

Le système va commencer par déclencher les règles topologiques puis celle liés à la dynamique. L'hypothèse de déplacement (A→ C) ayant la pondération finale la plus élevés sera priviligié.

5.3. Gestion des conflits

Nous allons maintenant étudier comment Lombric résout les conflits.

5.3.i. Déclenchement ordonné (Aval vers Amont) des résolveurs de conflits

Le système déplace d'abord les i-acteurs n'ayant pas de conflit. Ensuite les résolveurs de conflits sont déclenchés en respectant un ordre déterminé par les dépendances entre résolveurs de conflits (dans notre cas cet ordre respecte les précédences spatiales et temporelles). L'activation des résolveurs de conflits se propage des zones aval (sorties du carrefour) vers les zones amont (entrées du carrefour). Dans le modèle CASIMIR [*Cohen 89*], les lois de poursuite sont traité de manière séquentielle et respecte aussi une activation d'aval en amont. Le mouvement du véhicule est donc défini par rapport au véhicule qui le précède. La nouvelle position du véhicule en aval de la file est alors calculée en premier. Par la suite le système calcul la position des véhicules d'aval vers l'amont. Le dernier véhicule traité est donc celui qui est entré le dernier dans le carrefour.

L'activation des zones centrales (le lien Amont-Aval est inexistant) pose un problème d'ordonnancement. Cependant, **aucun cycle ne peut avoir lieu grâce** à la couleur des feux. En effet, le non-franchissement d'une ligne de feu dont la couleur est rouge, ordonne implicitement les zones au centre du carrefour. Et dans le centre du carrefour il y a toujours au moins 1 feu de rouge. Pour un carrefour sans feux, on prendra la zone arrêtée (ayant un taux d'arrêt > 0) comme "ligne de non-franchissement virtuelle".

Remarque : dans la réalité, un conducteur agit (en grande partie) en fonction des véhicules qui se trouvent devant lui. Par exemple, si un véhicule devant freine, après un temps de perception et de réaction, le conducteur ralenti son véhicule. Dans l'ordre d'activation (Aval-Amont) on respecte l'ordre de réactions des conducteurs. Par ailleurs, il est logique de résoudre en premier les conflits qui sont sur le point de sortir des tronçons.

5.3.ii. Algorithme de résolutions de conflits

L'agent résolveurs de conflits peut avoir des difficultés à choisir les déplacements locaux de ses i-véhicules. Deux possibilités A, B s'offrent à lui pour décider :

Cas A : Attendre plus d'information pour décider

Cas B : Evaluer des solutions avec un algorithme ordonnant les i-véhicules de manière arbitraire et déplace les i-acteurs en conséquence.

Cas A : Le système attend plus d'information pour déplacer les i-acteurs et gèle pendant un temps limité (2 cycles) le processus de déplacement. Au cycle suivant, lors de la mise à jour, certaines des nouvelles informations (passage au rouge d'un feu par exemple) sont propagées dans le graphe spatio-temporel. Certains arcs présents lors de l'analyse précédente vont disparaître ou être différemment pondérés. Dés que le résolveurs de conflits va pouvoir prendre une décision, le système essaye de rattraper le "temps perdu". On limite cette possibilité (le cas A) que pour une courte durée et lors d'analyse du trafic très fluide.

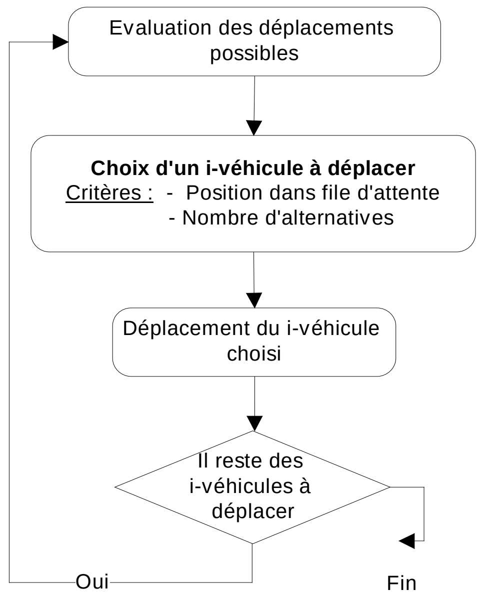
Dans certains des cas suivant, le système force la résolution (Cas B) :

Qualité des informations capteurs : La fiabilité des capteurs n'est pas bonne (ex : zones éloignées des capteurs)

Déplacements peu contraints : Moins le déplacements est contraint (plusieurs orientations et/ou plusieurs directions) plus on peut retarder l'évaluation

Faible nombre de i -acteurs.

Cas B : Le résolveur de conflits utilise un algorithme de résolution de conflits basé sur l'ordonnancement (arbitraire) des i-véhicules et le nombre de choix possibles pour ces i-véhicules.



Algorithme de résolution de conflits :

Le résolveurs de conflits estime la place disponible dans les zones visées. La deuxième étape consiste à lister les hypothèses de déplacement de chaque i-véhicule en déclenchant des règles liées au contexte. Par exemple toutes les règles pour les zones d'entrée vont être déclenchées pour un i-véhicule se trouvant sur un état-zone d'entrée. Ensuite, le résolveur de conflits ordonne les i-véhicules selon les critères suivants :

Position dans la file d'attente pour chaque zone

Nombre de choix de déplacements pour les i-véhicules (celui qui possède le moins de choix sera activé en premier).

Le résolveurs de conflits déplace un à un les i-véhicules dans les états choisis (pondération du choix le plus élevé) et met à jour au fur et à mesure les hypothèses de déplacement.

6. *Validation et cohérence du parcours du i-acteur*

6.1. « Historisation » des parcours des i-acteurs

Pour résoudre lors de l'évaluation des déplacements, des choix de même valeur (ayant la même pondération après le déclenchement des règles) le système utilise les parcours précédemment reconstitués et stockés. Le système garde deux niveaux de détails pour chaque parcours.

La description la plus riche est composée de variables spatiales, temporelles, comportementales et des justifications des choix des i-véhicules à chaque étape (toutes les secondes). Ces parcours seront utilisés pour des traitements à court terme, pour reconstituer des pelotons ou pour découvrir des incidents ponctuels par exemple. Ils vont permettre des réponses précises à des interrogations à partir d'un langage de requêtes : LOMSCAN.

Le deuxième type de parcours correspond à la définition de la trajectoire. Seule les informations spatiales sont gardées. Ce type de parcours est souvent utilisé pour arbitrer entre deux choix équivalents. Lors d'indétermination le système choisira le parcours le plus utilisé précédemment reconstitué.

Les parcours sont factorisés dans des arbres (pour éviter les redondances d'information) et pour les trouver et les utiliser plus rapidement lors de la recherche d'un parcours similaire.

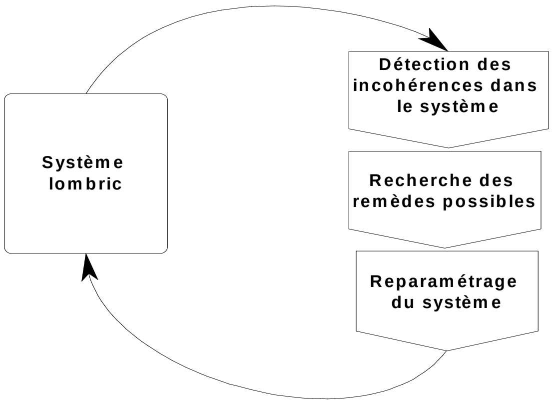
6.2. Gestion de la cohérence dans les parcours

Un module utilise les connaissances et les résultats de LOMBRIC afin d'améliorer son fonctionnement. Pour y parvenir, ce module analyse l'ensemble des connaissances construites par le système Lombric de façon plus globale. Son rôle est en partie de réguler le nombre de i-véhicules présents dans la scène.

L'adaptation du système à ce qu'il observe est un point essentiel et nécessaire pour comprendre et anticiper le comportement des véhicules. Face à la présence ou l’absence supposée d’un événement, le module d'adaptation modifie les paramètres du système. Cela permet de palier à une absence partielle d’information (défaillance momentanée des capteurs...) et de tenir compte des données bruitées.

Le module d’adaptation a pour objectif de vérifier la cohérence des résultats obtenus et le bon fonctionnement du système.

Ces 3 étapes permettent de contrôler la validité des flux reconstitués par le système lors des étapes précédentes et palier l’absence d’information fournie par les capteurs.



Ce module fonctionne en 3 étapes :

La première étape est consacrée à la *détection des incohérences,*

la seconde *cherche des remèdes applicables* et

la troisième étape *applique ces remèdes*.

Le module d'adaptation distingue 2 sortes d'incohérences dans le graphe spatio-temporel. :

6.2.i. Ressources libres

Il manque des i-véhicules sur une zone donnée au regard du taux d'occupation fourni par les caméras. Lors de la détection de ces **ressources libres** : Si la zone est à l'entrée du carrefour, le système crée un nouveau i-véhicule, sinon le module d'adaptation recherche un i-véhicule virtuel qui pourrait physiquement se trouver dans la zone concernée. Il va prendre le i-véhicule virtuel le plus proche de cette zone et respectant le flux de déplacement.

6.2.ii. Dépassement de capacité

Trop de i-véhicules se trouvent dans la zone au regard du taux d'occupation fourni par les caméras. Lors de la détection d'un **dépassement de capacité** : Si les i-véhicules se trouvent sur une zone de sortie, alors le module d'adaptation supprime les i-véhicules. Sinon il modifie les seuils (nombre minimum et maximum de i-véhicules) pour privilégier la sortie de la zone. Pour les zones centrales, le seuil de saturation de la zone est régulièrement modifié. Lors de zones complètement saturées des règles "topologiques" sont inhibées. Par exemple, la règle d'augmentation de l'occupation dans la zone suivante n'est pas déclenchable.

6.2.iii. Trop de i-véhicule virtuels

Si cela se produit toujours sur la même zone, on modifie le seuil d'occupation minimum représentant un véhicule. Si c'est tout le tronçon qui est concerné, on peut déclencher une procédure de redécoupage de la surface au sol en augmentant le nombres de zones par exemples.

7. Conclusion

Actuellement, notre système est capable de suivre les déplacements des véhicules à l'intérieur et aux abords d'un carrefour urbain. L'interprétation du graphe spatio-temporel évoluant en temps réel permet d'être robuste au regard des données imparfaites fournies par le traitement d'images. La prise en compte des conflits et des indéterminations lors du déplacement des i-acteurs permet de construire des parcours cohérents. Lombric analyse la reconstitution des parcours pour chaque acteur et en détecte les incohérences. Des remèdes sont alors appliqués en cours d'exécution et vont permettre d'adapter des paramètres de Lombric.

8. Références

[Aubert 96] Aubert D., Bouzar S., Lenoir F. and Blosseville J.M, - *Automatic vehicle queue measurement at intersections using image-processing*. Eighth international conference on road traffic monitoring & control, London, 1996.

[Abours 88] Abours S et Al - *Les modèles INRETS de simulation* - Synthèse INRETS N°12, 1988.

[Blosseville 88] Blosseville JM., Lenoir F., Beucher S., *Image processing applied to trafic measurement: The Titan system* - RTS – English issue – N°4, 1988.

[Brémond 97] Brémond F. - *Environnement de résolution de problèmes pour l'interprétation de séquences d'images*, Thèse de l'université de Nice Sophia-antipolis, Octobre 1997

[Cohen 90] Cohen S. - *Ingénierie du trafic routier* , Presse Ponts et chaussées; 1990

[Espié 93] Espié S. - *Architecture parallèle multi-acteurs pour la simulation microscopique du trafic* - Colloque "Automatique pour les véhicules terrestre", Amiens 1993.

[El Hadouaj 98] El Hadouaj S. - *Prise en compte des phénomènes d'anticipation dans le cas de conduite en file*, Stage de DEA, 1998

[Le Cerf 97] Le Cerf V, Pintado M, - *An Adaptive Model of Camera-Driven Urban Intersections Observation*, Workshop on dynamic scene recognition, juin 1997, Toulouse.

[Le Cerf 98] Le Cerf V, - *Reconstitution de déplacements de véhicules en milieu urbain à partir de données bruitées*, In Information Processing and Management of Uncertainty in Knowledge-Based (IPMU), juillet 1998, Paris.

[Nilsson 80] Nilsson N. - *Principles of Artificial Intelligence* - Springer-Verlag, 1980

[Sellam 94] Sellam S. , Boulmakoul A, - *Intelligent intersection : artificial intelligent and computer vision techniques for Automatic Incident Detection*, in Artifical Intelligent Applications to Trafic Engineering, 1994

[Trannois 94] Trannois H. - *Utilisation du contexte pour la distribution des connaissances dans un système Multi-Agents ; application à la simulation du trafic routier* - Thèse, 1994

LE SYSTÈME *HAMMOURABI* : DÉVELOPPEMENTS ACTUELS ET TRAVAUX FUTURS

Henri LESOURD

Henri.Lesourd@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé

Dans ce papier, nous présentons notre agent *Hammourabi*, conçu au départ pour la maintenance et au diagnostic d'un environnement UNIX en mode semi autonome. Nous nous intéressons d'une part dans ce papier aux *connaissances* que possède cet agent, et nous discutons aussi certains points à notre avis importants au sujet de son architecture, en resituant ceci dans un cadre plus général.

Mots-clés: Systèmes dynamiques, perception, abstraction, diagnostic.

1. Introduction

Que peut apporter aujourd'hui l'IA à l'utilisateur d'un ordinateur ? Nous savons que cet utilisateur est souvent gêné par le comportement des programmes qu'il manipule. De façon générale, l'environnement informatique est perçu comme quelque chose de complexe, qui demande des connaissances, de la maîtrise et du savoir-faire. Ceci nous étonne au premier abord, car il nous semble que l'informatique est basée sur quelques principes très simples, et à notre avis rapidement assimilables. Nous connaissons d'autre part certaines des difficultés rencontrées lors de la rédaction et de la mise au point de programmes, et nous sommes amenés à prendre conscience du temps que nous y passons ; nous nous apercevons enfin qu'en réalité, il nous arrive de passer énormément de temps à simplement configurer la machine, et à tenter d'imaginer ce qui peut bien avoir lieu en son sein. Comment tout ceci est-il possible ? Apparemment, la connaissance de principes généraux, bien qu'utile, apparaît être insuffisante : elle ne nous permet pas de maîtriser l'outil comme nous le souhaiterions, et nous sommes toujours forcés de nous atteler à des tâches répétitives et consommatrices de temps. Ceci est à notre avis un vrai problème, dont nous avons analysé certaines des causes dans notre travail, et pour lequel nous proposons une solution : l'agent *Hammourabi*, dont la fonction est justement d'analyser le comportement du logiciel dans la machine, dans le but de mettre en œuvre – ou d'aider à la mise en œuvre - de solutions aux problèmes rencontrés.

2. Le problème posé

Nous pouvons d'abord concevoir les problèmes liés à la manipulation et à la maîtrise des machines par un utilisateur comme des problèmes liés à l'absence de *déclarativité* des langages utilisés pour communiquer avec ces machines. C'est par exemple l'approche adoptée par Oren Etzioni, qui plaide pour la réalisation d'agents logiciels (ou *Softbots*) destinés à aider l'utilisateur lors de son interaction avec la machine. De tels *Softbots* ont d'abord été mis en œuvre dans le cadre d'un environnement UNIX [ETZ92], et appliqués ensuite dans le domaine de l'*Internet* [ETZ94, KWO96]. Dans de telles approches, il est sous-entendu que l'environnement met ses ressources à la disposition de l'utilisateur sous la forme d'un certain nombre de *commandes*, que l'on modélise ensuite de façon logique sous la forme d'opérateurs. Il est alors possible de faciliter le travail de l'utilisateur en lui fournissant un moteur dans lequel on peut passer des commandes énoncées sous la forme de buts logiques déclaratifs. Le moteur décompose ensuite ces buts logiques et construit un programme (un *plan*) en utilisant les opérateurs qu'il connaît, ce qui évite en particulier à l'utilisateur de devoir connaître la syntaxe particulière des commandes sous-jacentes, et d'avoir simplement même à connaître ces commandes. De cette manière, il est possible d'encapsuler des systèmes particuliers et de donner accès à des environnements hétérogènes à travers un langage unique. Il reste que la faisabilité d'une telle encapsulation de systèmes n'est pas discutée dans les travaux mentionnés ci-dessus, bien qu'elle se rattache en réalité au difficile problème de l'*intégration* des systèmes. Cette intégration est elle possible ? Si nous observons ce qui se passe dans le monde de l'informatique, il apparaît que la réponse à la question n'est pas du tout évidente : c'est un peu la raison d'être de toutes les techniques, méthodes de conception et de modélisation, tentatives de normalisation etc. existantes à ce jour (Méthodes Merise, Kads, conception par objets, aspects, langages dérivés d'SGML, protocoles, CORBA, etc.). A notre avis, la conception des systèmes que l'on désire intégrer doit prendre en compte un minimum le besoin d'intégration, ce qui est malheureusement le cas de très peu d'environnements en service aujourd'hui. Notons enfin que l'approche « basée sur la déclarativité » est déjà ancienne puisqu'en 1987 par exemple, B. Roger [ROG87] conçoit et met en œuvre une surcouche d'UNIX dotée de connaissances sur les application utilisées, capable d'aider un utilisateur en fournissant une interface en langage naturel : l'utilisateur peut poser des questions et obtenir des réponses en français sur ce qu'il voit, et est d'autre part suivi par la machine, qui lui propose des écrans d'aide contextuels en tenant compte de son comportement antérieur. Ce type d'approche a été repris plus tard [PAC94] dans l'environnement Smalltalk.

Nous nous inspirons pour notre part dans une certaine mesure de l'approche décrite ci-dessus, mais il est aussi une raison importante au manque d'intelligibilité des machines : vouloir rendre le langage de communication déclaratif est certes un but souhaitable, mais il nous semble que pour être en mesure de vraiment *comprendre* ce que l'on dit à cette machine, il est tout d'abord nécessaire de pouvoir simplement *observer* ce qui s'y déroule ; il est ensuite nécessaire de pouvoir *modifier* le comportement de cette machine. Il ne suffit pas de communiquer avec cette machine en utilisant un langage commode et intuitif : encore faut-il que le langage en question *utilise les bons concepts*. Nous nous souvenons par exemple des problèmes rencontrées par nous lors de la consultation de la documentations de divers systèmes : cette documentation a beau être déclarative, puisqu'énoncée en langue naturelle, elle *ne contient souvent <u>pas</u>* l'information dont nous avons besoin. Cette information peut être indisponible pour diverses raisons : d'abord parce que la documentation est mal organisée, ce qui oblige l'utilisateur à chercher au hasard de ses associations d'idées ; d'autre part, un certain nombre d'informations sont dissimulées. C'est le principe de l'encapsulation, qui consiste à dissimuler les détails internes au système, voire même à en interdire totalement l'accès, dans le but de favoriser l'évolutivité des systèmes et la réutilisabilité des codes sources. Malheureusement, c'est surtout de ce type d'informations dont nous avons besoin pour pouvoir comprendre ce qui se passe dans l'ordinateur et résoudre des problèmes quand une fonctionnalité n'est pas prévue. Nous sommes pour notre part partisans d'une encapsulation modérée dont la définition exacte reste encore à trouver, qui tiendrait en particulier compte de l'*exhaustivité* de l'ensemble des fonctions fournies dans une API. Le besoin d'améliorer l'approche classique en conception de systèmes se fait jour de plus en plus actuellement et doit mener, selon nous, à une révision des principes aujourd'hui acceptés, dans le but d'éliminer leur beaucoup trop grande rigidité [KIC92]. En attendant, il demeure difficile de réaliser des logiciels intégrés de façon suffisamment profonde avec l'environnement qui les héberge. Comment faire ?

Dans la suite de ce papier, nous décrivons d'abord notre agent *Hammourabi* ainsi que le domaine des problèmes qu'il doit pouvoir traiter, en donnant plusieurs exemples de tels problèmes et des moyens envisagés pour les résoudre (partie 2). Dans la partie 3, nous donnons un aperçu plus varié des connaissances que dont nous souhaiterions doter *Hammourabi* dans la suite de nos travaux. Dans la partie 4, nous réfléchissons de manière un peu plus théorique à ce que nous a appris la conception d'*Hammourabi*.

3. La solution envisagée

Dans cette partie, nous présentons d'abord deux exemples (paragraphe 2.1), puis nous donnons des détails sur le fonctionnement de l'environnement que nous avons bâti (paragraphe 2.2), et nous montrons le fonctionnement d'*Hammourabi* sur le second exemple auparavant présenté (paragraphe 2.3).

3.1. Présentation

Les problèmes auxquels nous nous intéressons sont souvent des problèmes dans lesquels interagissent plusieurs composants logiciels, et où l'on assiste à la propagation d'erreurs dans l'environnement. De façon générale, nous pensons que la capacité à extraire des informations issues du *comportement* du système est tout à fait fondamentale, et que sans elle, on n'a qu'une vision statique et très partielle de la réalité, ce qui diminue ensuite forcément la capacité d'interaction que pourrait posséder un agent (humain ou artificiel) dans le système. Voici un premier exemple du type de problème auxquels nous nous intéressons :

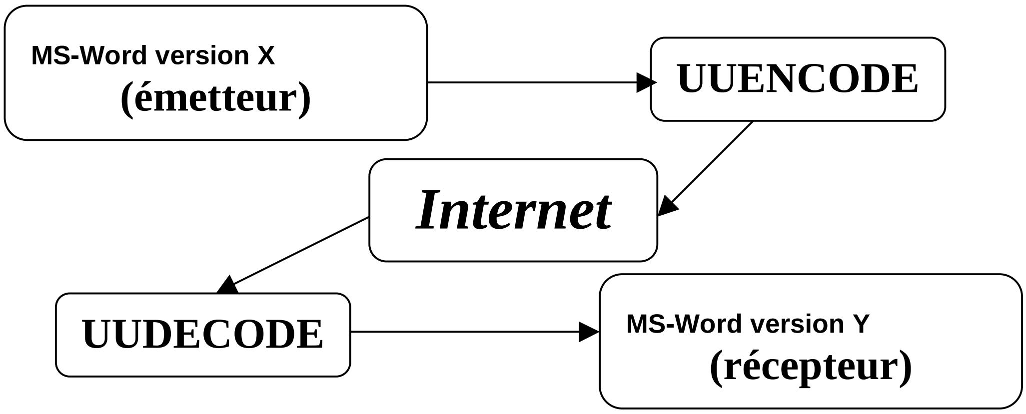


Figure 1 : exemple 1

Dans cet exemple, l'utilisateur d'un logiciel de courrier électronique envoie un document Word à son correspondant. Le document est d'abord recodé avec `UUENCODE`, un composant qui traduit un fichier binaire en un format compatible avec l'*Internet* ; le document `UUENCOD`é est ensuite transmis à travers le réseau, puis décodé après réception par le composant `UUDECODE` ; le destinataire du message consulte ensuite ce document avec le logiciel Word. Les problèmes observés par nous dans l'environnement décrit sont :

- Défaut de compatibilité entre les versions de Word de l'émetteur et du récepteur ;
- Défaut de compatibilité entre les versions des composants `UUENCODE`/`UUDECODE` de l'émetteur et du récepteur ; bugs dans `UUENCODE` ;
- Erreurs de transmission réseau.

Une remarque importante que nous pouvons faire est que ces trois types de causes possibles ne sont pas exclusifs les uns des autres, contrairement à ce qu'on imagine naïvement (on se dit que quand même, on ne peut pas être suffisamment malchanceux pour avoir *à la fois* la mauvaise version de Word et un `UUENCODE` bogué… C'est néanmoins ce qui est arrivé au sujet que nous interrogions : nous nous sommes justement aperçus de ça après avoir résolu le premier problème (le `UUENCODE` bogué), et observé qu'*en plus*, notre correspondant nous envoyait des fichiers Word 97 et non des fichiers Word 95).

Ce que nous apprend cet exemple, c'est que quand on se trouve dans une situation où le logiciel est constitué d'un ensemble de composants coopérants, il peut être facile de décrire les causes possibles des problèmes rencontrés en terme d'erreurs localisées à un endroit de la chaîne de traitement de l'information, qui se propagent ensuite dans le reste du système. Notons aussi que la possibilité d'acquérir certaines informations est extrêmement utile : si nous avions par exemple eu accès aux numéros de version des logiciels employés de part et d'autre, il aurait été facile de diagnostiquer les problèmes de compatibilité, au lieu d'être obligé de simplement les *supposer* (de façon générale, remarquons que la capacité à supposer et à imaginer est probablement assez corrélée à la capacité d'accès aux informations pertinentes).

En pratique, nous n'avons souvent pas accès à de telles informations dans les environnements que nous connaissons (à part peut-être en Smalltalk, mais dans ce cas-là, nous manquons d'applications) ; nous avions d'abord procédé en réalisant un espion au sein d'UNIX, mais cette solution, bien qu'utile, ne nous donne pas la finesse d'observation souhaitée : les exemples de problèmes collectés par nous étaient beaucoup trop divers, et nous avons abouti à la réalisation d'une plate-forme de simulation, dans laquelle évolue notre agent *Hammourabi* et dans laquelle nous construisons des *modèles* des environnements logiciels antérieurement observés par nous. Il se pourrait qu'une telle plate-forme de simulation finisse par se révéler un outil extrêmement commode pour la réalisation d'applications véritables ; d'autre part, une fois que nous y aurons complètement intégré notre espion UNIX, nous espérons obtenir un outil prometteur dans le domaine de l'administration système. Dans ce qui suit, nous allons à présent développer une description de notre agent *Hammourabi* et de l'environnement de développement que nous avons réalisé spécialement pour lui, à travers un second exemple qui s'inspire d'un cas réel mettant en jeu le service de gestion du courrier électronique du système UNIX. Grâce à ce service, les utilisateurs possédant un compte sur la machine ont à leur disposition une boîte aux lettres, dans laquelle sont stockés les messages que leur envoient d'autres utilisateurs situés sur d'autres machines, à travers le réseau *Internet*. Le service de gestion du courrier électronique du système UNIX met en jeu les entités suivantes :

- Les *boîtes aux lettres* des différents utilisateurs, concrétisées par des fichiers texte. Il existe une boîte aux lettres par utilisateur ; ces boîtes aux lettres sont toutes situées dans le répertoire `/var/mail/` de l'arborescence UNIX, et chacune porte le nom de l'utilisateur auquel elle appartient.
- L'*application* `mail`, *dédiée au courrier électronique*, qu'emploient les utilisateurs pour consulter les messages que contient leur boîte aux lettres.
- *Le démon* `inetd`, qui tourne en tâche de fond, et réceptionne le courrier arrivant sur la machine, puis le stocke dans la boîte aux lettres de l'utilisateur auquel il est destiné. Ce démon `inetd` assure aussi la gestion d'autres services *Internet*, tels que la gestion des liaisons `telnet` et `ftp`, par exemple.

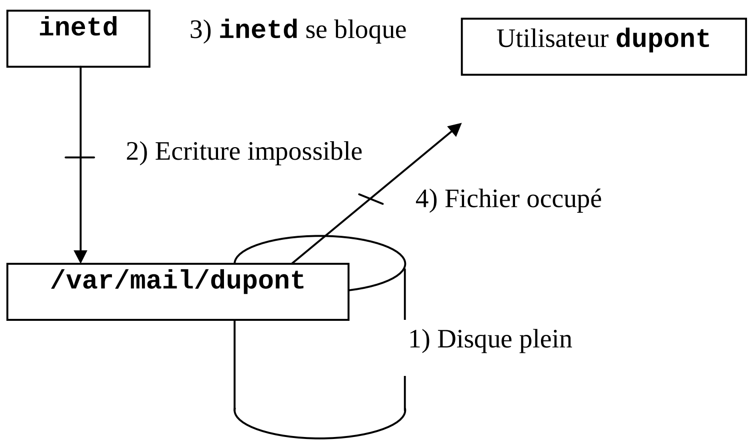


Figure 2 : exemple 2

Le scénario d'erreur que nous présentons se déroule comme suit :

- Le disque contenant les boîtes aux lettres est plein ;
- Un peu plus tard, un message arrive sur la machine ;
- Le démon `inetd` réceptionne le message, puis ouvre la boîte aux lettres de l'utilisateur `dupont` auquel le message est destiné (accès en lecture/écriture) ;
- Il tente alors de stocker le message sur le disque ; survient alors l'erreur "Ecriture impossible", à la suite de laquelle le démon `inetd` se plante ;
- Un peu plus tard, notre utilisateur `dupont` tente d'utiliser l'application de courrier électronique, qui échoue sur une erreur "Fichier occupé" au moment où elle tente d'accéder à la boîte aux lettres de M. Dupont. Cette dernière est effectivement toujours occupée par le processus démon `inetd`, qui s'est planté en ne libérant pas l'accès à la ressource.

Le problème sera résolu successivement en stoppant et en relançant le démon `inetd`, puis en nettoyant la boîte à lettres de l'utilisateur `dupont`, qui se trouve à présent dans un état corrompu à la suite de l'erreur « écriture impossible » qui a planté le démon `inetd`. Par la suite, d'autres utilisateurs seront confrontés à des problèmes semblables à celui rencontré par l'utilisateur `dupont`, jusqu'à ce qu'on s'aperçoive que la cause de tous ces problèmes est l'exiguïté du disque destiné à stocker le courrier électronique. Il faudra alors entreprendre de faire de la place sur ce disque.

3.2. L'environnement expérimental et le langage *V*

Le langage que nous avons bâti pour réaliser nos maquettes de logiciels (que nous appelons le langage *V*) est inspiré du langage *Prolog*, ainsi que de *Snark* [LAU], mais nous l'avons en particulier étendu dans les directions suivantes :
Gestion des processus : exécution de plusieurs processus de résolution concurrents ;

Espionnage : possibilité d'observer les processus au moment où ils interagissent avec d'autres entités, y compris des processus.

Dans cet article, nous rappelons simplement les points principaux pour permettre au lecteur non initié de comprendre ce dont il est question, mais nous le renvoyons à de précédentes publications pour une information plus détaillée [LES98, LES98-2]. En substance, nous modélisons les entités présentes dans la machine en considérant que certaines de ces entités sont des entités *actives* (ou acteurs) ; ces entités actives effectuent des *actions* portant sur d'autres entités, et ce sont ces actions que nous sommes à même d'observer dans notre environnement. Espionner une action consiste à fournir au système un triplet qui peut être partiellement instancié, qui mentionne :
L'*acteur* que nous désirons espionner ;
L'*action* que nous cherchons à espionner ;
L'*objet* sur lequel a lieu cet action.

Les programmes sont constitués par des suites d'instructions *à la Prolog*, qui effectuent des lectures en énumérant tous les cas possibles, par exemple :

Programme P1 :

```
$P:(? TYPE:PERSONNE NOM:DUPONT)
$P:(? PRENOM:$N)
```

Ce programme est équivalent à l'énoncé du but *Prolog* suivant :

```
?- personne(dupont,N).
```

Dans notre système, les objets sont représentés sous la forme de listes attribut-valeur dans lesquelles il est possible de supprimer et de rajouter des attributs ; d'autre part, le langage nous permet de référencer les objets eux mêmes avec des variables. Dans l'exemple ci-dessus, lorsqu'on a passé la première instruction, la variable `$P` référence l'objet matché courant, par exemple : `(TYPE:PERSONNE NOM:DUPONT PRENOM:JEAN-PIERRE)`.

Pour *espionner* tous les processus lisant des informations au sujet de personnes dont le nom est `DUPONT`, ce qui inclut le programme P1 donné plus haut, nous écrivons alors :

Programme P2 :

```
$P:(¿? TYPE:PERSONNE NOM:DUPONT)
$P:(? ID:$ID)
   (_PRINT $ID)(_CR)
```

Cette fois-ci, au moment où le programme P1 exécute sa première instruction de lecture, le programme P2 se déclenche, et à l'issue de la première instruction, la variable `$P` du programme P2 est instanciée et contient alors l'objet représentant le processus instance du programme P1 ayant effectué l'action. L'instruction qui suit lit l'identificateur de processus de l'instance de P1 qu'on espionne, qu'on affiche ensuite à l'écran dans l'instruction suivante.

Pour faire ce qui est décrit ci-dessus, il faut qu'existe un autre type d'instruction, servant à l'espionnage. L'instruction de lecture est préfixée par un '`?`', alors que l'instruction d'espionnage est préfixée par un caractère '`¿`'; comme il existe d'autres types d'instructions que la lecture et l'espionnage, il faut spécifier ensuite le type d'action qu'on espionne, par exemple :
`¿?` : espionner les lectures ;
`¿!` : espionner les écritures ;
`¿¿` : espionner les espions.

Il est possible aussi d'espionner et d'interdire tel ou tel type de pattern d'action, ce qui nous donne la possibilité d'écrire des programmes contrôlant d'autres programmes :

Programme P3 :

```
$P:(¿?■ NOM:MOTS-DE-PASSE)
$P:(? TYPE:SUPER-USER)
$P:(↑)
```

Ce programme P3 espionne et stoppe tous les processus consultant l'entité contenant les mots de passe des utilisateurs, teste si ces programmes ont le droit de consulter ce fichier (s'ils ont le type `SUPER-USER`), et dans ce cas-là seulement, les fait redémarrer. C'est sur ce genre d'exemple qu'il est possible d'utiliser les espionnages dans UNIX précédemment mis au point par nous. Avec ces routines, il est possible d'espionner et de stopper les primitives du noyau, donc l'on peut espionner les ouvertures du fichier des mots de passe UNIX (fichier `/etc/passwd`) exactement de la façon qui vient d'être décrite. Il est par contre impossible de reproduire le comportement des programmes P1 et P2, parce que dans ces cas-là, la manipulation des données concernant `DUPONT` par une application inconnue de nous serait inobservable dans le cas général. Ceci nécessiterait de doter notre agent de capacités liées à la reconnaissance de forme, et nous éloignerait sensiblement de notre problématique.

3.3. Un exemple détaillé

Nous allons à présent reprendre l'exemple numéro 2 sur la gestion du courrier électronique que nous avons donné plus haut, et montrer de façon détaillée comment nous le traitons. Nous modélisons tout d'abord l'API système UNIX de façon à reproduire le comportement du *filesystem*, en particulier en ce qui concerne les verrous positionnés sur les fichiers par le système quand un programme a ouvert ce fichier. Nous modélisons l'occurrence d'erreur par l'émission d'un message dans notre API simulée (dans notre environnement de simulation, l'écriture (préfixe `!`) correspond à l'émission d'un message avec mémorisation ; l'émission de message sans mémorisation est la forme la plus basique d'action possible (préfixe `¡`)). Voici par exemple la façon dont nous écrivons la procédure `OUVRIR` (le `fopen()` du langage C) :

Procédure `OUVRIR` :

```
(CLASSE:ACTEUR TYPE:FONCTION ♀:OUVRIR
  ENVIR:($REP:¤    $NFIC:¤    $RESU:¤    $ETAT:¤
$MOIMEME:¤ $U:¤)
   CODE:(
        (¿>    OUVRIR    REP:$REP    NFIC:$NFIC
RESU:$RESU)
  $REP:(? FIC:$RESU)
 $RESU:(? ♀:$NFIC)
 $RESU:(? ETAT:$ETAT)
       (_§ $MOIMEME)
       (▼/
          ($RESU:(? ETAT:LIBRE)
           $RESU:(- ETAT:LIBRE)
           $RESU:(! ETAT:OCCUPE)
           $RESU:(? UTILISATEURS:$U)
              $U:(! UTILISATEUR:$MOIMEME)
          )
          (       (¡ ERREUR ACTEUR:$MOIMEME  <--
Instruction I3
                           ACTION:OUVRIR
                           OBJET:$RESU
                           RESULTAT:$ETAT
                 )
          )
       )
   )
)
```

Si nous nous plaçons dans le cadre du scénario de l'exemple 2, au moment où l'utilisateur `DUPONT` tente de lire sa boîte aux lettres et reçoit le message d'erreur "Fichier occupé", l'instruction I3 ci-dessus se déclenche et émet un pattern d'erreur. Ce pattern est ensuite détecté par les procédures de diagnostic qui constituent *Hammourabi*. Par abus de langage, nous appelons ces procédures de diagnostic des *situations d'erreur*. Dans le cas qui nous occupe, nous déclenchons la situation d'erreur `FICHIER-OCCUPE` qui traite ce problème :

```
(CLASSE:ACTEUR TYPE:DEMON ♀:FICHIER-OCCUPE
  ENVIR:($A:¤ $O:¤ $US:¤ $PA:¤ $PU:¤ $MTU:¤
             $CTU:¤ $CON:¤ $TEST:¤ $MOIMEME:¤)
    CODE:(
       (¿   PERCEPT   ACTEUR:$A   ACTION:OUVRIR
RESULTAT:OCCUPE)
   $A:(? PROGRAMME:$PA)       <-- Instruction 1
   $O:(? UTILISATEURS:$US)    <-- Instruction 2
```

```
  $US:(? UTILISATEUR:$U)
   $U:(? PROGRAMME:$PU)       <-- Instruction 4
  $PU:(? META:$MTU)           <-- Instruction 5
 $MTU:(? CONTEXTE:$CTU)       <-- Instruction 6
      (▼/                     <-- ▼/
sert à exprimer une suite de choix exclusifs
         §ACCES NON AUTORISE§
          ($CTU:(?  CONNEXION:$CON)         <--
Choix 1
           $CON:(?  LIEU:$TEST  DUREE:PERMANENT)
<-- Instr 7
                (? $TEST OBJ:$O)
                (? ESPIONNER ACTEUR:$PA ACTION:*
OBJET:*)
                (_§ $MOIMEME)
                (? NOTIFIER-ADMINISTRATEUR
                   SITUATION:$MOIMEME)
          )
          ((?  PROCESSUS-BLOQUE  ACTEUR:$U))  <--
Choix 2
       )
    )
)
```

Nous ne détectons pas directement les événements liés aux erreurs parce que leur traitement est centralisé pour éviter que l’on n’observe tout à la fois ; en utilisant la fonction `ESPIONNER`, *Hammourabi* peut ainsi guider la focalisation de son système perceptif. Lorsque la situation `FICHIER-OCCUPE` se déclenche, il commence d’abord par se renseigner sur les processus utilisant actuellement l’objet où a eu lieu l’erreur (instructions 2 à 4), puis consulte les informations qu’il possède sur ces processus (instructions 5 et 6). Il sera donc amené à consulter les informations concernant le démon `inetd` (TSVP) :

```
(CLASSE:ACTEUR TYPE:DEMON ♀:INETD
 ................................
   META:(
      VERSION:1
      CONTEXTE:(
         CONNEXION:(
            CLASSE:CONNEXION
            LIEU:(FICHIER NOM:"/dev/tcpip")
            MODE:(LECTURE ECRITURE)
            DUREE:100
         )
         CONNEXION:(
            CLASSE:CONNEXION
            LIEU:(FICHIER NOM:"/var/mail/*")
            MODE:(LECTURE ECRITURE)
            DUREE:100
         )
      )
   )
)
```

Hammourabi s’apercevra alors qu’il n’est pas prévu que le démon `inetd` accède à une boîte aux lettres de façon permanente, et supposera alors que le processus est bloqué, ce qui l’amènera à se dérouter vers la procédure de diagnostic `PROCESSUS-BLOQUE` (choix 2), qui décidera alors de réinitialiser le démon `inetd` et d’espionner son comportement. Par la suite, on observera une erreur d’écriture sur le disque provoquée par le démon `inetd`, ce qui rendra possible la découverte de l’origine de tous les problèmes, à savoir le manque d’espace disque.

4. Les connaissances du système

Nous avons d'autre part réfléchi au type de connaissances dont on pourrait ultérieurement doter *Hammourabi*. Nous allons en dire quelques mots ici. Nous classons les problèmes du type de ceux décrits plus haut dans la catégorie des problèmes liés à la gestion d'erreur quand un programme possédant des ressources termine sans libérer ces ressources ; c'est le module 1 de l'agent *Hammourabi*. Le module 2 examine une série de problèmes connexes, ceux qui surviennent quand le format des données est corrompu à la suite d'une erreur ayant entraîné l'arrêt d'un programme, corruption des données qui entraîne par la suite des blocages de la part d'autres programmes tentant de manipuler ces données. Dans le contexte de l'exemple donné ci-dessus, un tel comportement apparaît nettement quand l'en-tête des messages électroniques est corrompu à la suite de l'erreur affectant le comportement du démon `inetd`, ce qui empêche ensuite leur consultation par l'application de consultation du courrier électronique, et donc aussi leur suppression et le retour des fichiers boîtes à lettres dans un état cohérent. Nous avons observé exactement le même type de comportement avec le logiciel de gestion des partitions de Linux RedHat (*Disk druid*) : une fois qu'on a fabriqué une partition étendue *Linux* avec *Disk druid*, il devient impossible de la supprimer avec le `FDISK` MS-DOS, parce que d'une part si on essaie de supprimer la partition, `FDISK` annonce que la partition étendue contient des « lecteurs logiques », et que d'autre part, quand on consulte le contenu de cette partition, `FDISK` annonce qu'il n'existe « aucun lecteur logique défini » ! Une fois complétée, notre simulation exhibera certainement ce type de comportement, ce qui renforce la confiance que nous avons en notre approche empirico-simulatoire. Nous pourrions aussi nous intéresser aux actions anormalement longues (module 3), qui sont souvent synonymes d'erreurs ou de mauvais choix d'une opération. Le module 4 d'*Hammourabi* est là pour résoudre les problèmes causés par des débordements dus au calcul avec des nombres de précision fixe dans les ordinateurs (problèmes de type « an 2000 »). Un autre aspect important de la gestion des informations (module 5) dans un ordinateur concerne le suivi des duplications, fusions et destructions d'informations dans la machine, qui permettrait d'avertir intelligemment l'utilisateur lors de la destruction d'une information, par exemple au cas où il n'existe plus qu'un exemplaire de cette information. Le suivi de l'*utilisation* d'une information peut aussi avoir son utilité, par exemple si l'on constate que cette information n'a été manipulée que par le processus qui l'a créée. Dans un tel cas, nous pourrions éventuellement conclure que le travail effectué par le processus en question ne sert à rien, et diagnostiquer ainsi un problème lié à un *calcul inutile*. Comme nous l'avons dit, le suivi des versions des informations peut permettre de détecter le moment où une information est détruite irréversiblement ; de façon plus générale, il s'agit là d'une évaluation portant sur la *dangerosité* des opérations effectuées. Dans le module 6 d'*Hammourabi*, nous nous intéresserons peut-être de façon plus spécifique à ces questions, en nous plaçant du point de vue de la *sécurité* d'un système informatique, et des problèmes liés au *détournement de service* dans un système. Le module 7 d'*Hammourabi*, enfin, concerne un aspect important de la gestion du système, qui est la constitution d'un *historique à long terme*, dans lequel on note les actions administratives effectuées ; ceci est important car beaucoup de problèmes trouvent leur origine dans de telles actions d'administration fautives. Par exemple, dans le cas réel qui inspire notre exemple 2 au sujet du courrier électronique, nous nous sommes trouvés à court de place disque parce que nous avions auparavant installé une application (un gestionnaire de cache web) stockant beaucoup de données sur ce même disque, alors même qu'un second disque avait été prévu pour stocker ces informations là, qui resta inutilisé pendant toute la période précédant l'apparition des problèmes.

5. Problématique

Dans cette partie, nous réfléchissons à l'architecture que nous avons été amenés à donner au système ; nous nous apercevons que ce système, dédié à la perception et à l'interprétation d'un environnement dynamique complexe, présente certains aspects récurrents dans d'autres travaux que le nôtre. Nous considérons d'abord que la première couche (bien que ce terme ne soit pas véritablement adapté ; nous ne pensons pas qu'il soit possible, ou même souhaitable de concevoir un système d'IA en utilisant une architecture en couches, parce que dans la réalité, les relations de dépendance entre ces couches ne correspondent pas à un modèle hiérarchique où la couche supérieure contrôlerait totalement celle du niveau d'en dessous ; il faut rechercher à notre avis plutôt une interaction harmonieuse entre des composants, mais c'est plus difficile à faire) de notre système effectue un traitement de nature *abstractive* ; le composant situé au niveau supérieur assure l'étiquetage, c'est à dire la reconnaissance des informations dont on dispose : nous appelons ceci la *qualification*. La dernière couche est celle où a normalement lieu l'émergence du *sens* des informations ainsi étiquetées, et le traitement correspondant est un traitement alors lié à la *cohérence* que l'on peut associer à certaines manières de regrouper les informations.

5.1. Abstraction

Notre système *Hammourabi* est confronté à un univers changeant, dans lequel l'acte de percevoir comporte un aspect *actif*, ceci parce que la quantité d'informations traitée par la machine à un instant donné doit forcément être limitée ; pour ce faire, il nous faut trouver un moyen de *focaliser* l'attention du système sur les aspects « intéressants » de la scène observée ; le mécanisme de focalisation doit utiliser un *modèle* du monde, une grille à partir de laquelle on associe des significations aux percepts entrants dans la mémoire du système. La perception telle que nous la décrivons peut donc être vue comme une forme d'*abstraction*, puisqu'elle nous amène à *ne pas* prendre en compte certains éléments dont nous avons connaissance. Cet aspect des choses est connu depuis longtemps déjà. C'est par exemple l'opinion de Rudolf Arnheim [ARN69], pour qui l'origine de la pensée abstraite n'est pas à chercher dans le mécanisme de formation des mots, mais se situe au contraire bel et bien dans les mécanismes de nos organes sensoriels : le processus de perception se définit en réalité par la collaboration de deux aspects, l'un de nature abstractive, l'autre de nature généralisante. Des travaux plus récents sont venus étayer ce point de vue ; pour l'étude de la compréhension dans le domaine du jeu de Go, par exemple, Patrick Ricaud [RIC95] donne un modèle cognitif du joueur basé sur l'abstraction, alors que Tristan Cazenave [CAZ97] conçoit pour sa part la manière de jouer au Go comme un processus de généralisation basé sur la construction d'explications logiques.

5.2. Qualification

Une fois que la machine a été en mesure de diriger son attention vers les éléments de l'univers utiles à son raisonnement, elle entame ensuite une phase de reconnaissance de ces éléments que nous appelons la *qualification*. Qualifier un percept consiste à évaluer les relations qui le lient avec le reste des éléments perçus, à lui assigner un *rôle* dans l'ensemble de ce qui est vu. Ce traitement qualificatif a pour objet de structurer ce qui figure sur la « rétine » de l'ordinateur, et d'extraire un *modèle de travail* à partir de ceci, ce qui est un aspect bien connu du fonctionnement de la perception (voir par exemple [SOW84], [DEL94]).
Dans *Hammourabi*, c'est le mécanisme d'activation des situations d'erreur qui assure la qualification : une situation d'erreur contient en particulier un pattern servant à détecter la présence de certaines configurations d'événements, à tester l'état de certains objets, etc. Nous retrouvons très souvent un tel mécanisme de qualification dans divers systèmes, bien que ceci ne soit pas toujours apparent au premier abord ; dans le système *Lombric* [LEC98] par exemple, qui est un système dédié à la reconstitution de scènes de trafic dans un carrefour routier, il n'existe pas à proprement parler de composant destiné à prendre en charge la qualification : à cause des contraintes temps réel que doit assurer *Lombric*, la qualification est intégrée de façon procédurale au fonctionnement du moteur.

5.3. Interprétation d'un environnement dynamique complexe

De façon générale, le modèle perceptif dont nous avons discuté est supposé basé sur la *mémoire à court terme* du système ; quand il s'agit par la suite d'interpréter les données perçues et le modèle de travail dont il a été question ci-dessus, le fonctionnement a alors lieu dans la mémoire à *moyen terme* de ce même système. Dans notre système, nous avons choisi de définir le mécanisme de compréhension et de production d'explications par la donnée d'un processus de construction d'interprétations cohérentes : quand un nombre suffisant d'éléments issus de la première couche perceptive ont été amenés dans la mémoire à moyen terme du système, il s'agit alors de regrouper certains de ces éléments en des sous ensembles cohérents par rapport à certains critères. Par abus de langage, nous appelons de tels sous ensembles cohérents des *interprétations*, ou encore des *explications*. Les critères de cohérence peuvent être diversement choisis ; chez Thagard [THA98], le critère de cohérence est la non contradiction logique d'un ensemble d'hypothèses formelles ; chez Grandjean et al. [GRA92], une approche basée sur les mêmes principes est adoptée, qui utilise en sus une logique possibiliste ; de façon générale, il nous semble que la compréhension présente un aspect *émergent* absolument irréductible ; mais encore faut-il que cette émergence se base sur quelque chose, ce qui signifie définir une propriété désirée pour les ensembles obtenus par un tel processus d'émergence.

Dans *Hammourabi*, le traitement de la cohérence est lié au fait que les alarmes et événements observés sont des conséquences d'autres problèmes situés ailleurs dans l'univers, et actuellement inconnus d'*Hammourabi*. Il lui faut donc résoudre les problèmes dès qu'ils se présentent (quand c'est possible), mémoriser ce qui a eu lieu au cours de la procédure de résolution (c'est à dire enregistrer une *trace* de ce processus de résolution) puis éventuellement positionner des espions (qui eux aussi génèrent des traces lors de leur déclenchement, associées cette fois-ci aux entités espionnées, et non au résultat des actions effectuées par l'agent *Hammourabi* lui-même). Ce n'est qu'ensuite, peut-être longtemps après, qu'*Hammourabi* aura la possibilité de lier entre elles plusieurs de ces traces de façon cohérente, construisant ainsi des explications recherchées des phénomènes observés. En conclusion, un traitement lié à la cohérence est apparu nettement dans *Hammourabi* parce que dans une situation donnée les informations dont on a besoin pour trouver la source des phénomènes observés ne sont pas toutes disponibles en même temps (elles peuvent même ne l'être jamais), ce qui nous oblige à les stocker dans une mémoire à long terme, et à construire donc peu à peu une *représentation* de l'univers dans cette mémoire à long terme.

6. Conclusion

Dans ce papier, nous avons décrit nos travaux dans le domaine de la réalisation de systèmes d'IA dotés de capacités perceptives. Bien que nous ne nous en démarquions pas totalement, l'approche majoritairement adoptée en IA est plutôt d'un type logique dans lequel on considère des propriétés, des phrases émises au sujet du monde, et où le problème principal consiste à étudier des procédures de décision capables d'évaluer la véracité de tels prédicats portant sur ce monde [NSI63]. Depuis les travaux de la fin du dix-neuvième siècle dans le domaine de la logique formelle (paradoxe de Russell), prolongés au vingtième siècle par Gödel, puis Turing (qui répond par la négative à la question de Hilbert concernant l'existence d'une procédure mécanique capable de résoudre un ensemble de problèmes suffisamment vaste), nous connaissons mieux les limites de telles procédures de décision mécaniques. Notons aussi que le courant formaliste n'est pas le seul a avoir donné son interprétation du fonctionnement de l'esprit ; Henri Poincaré [POI02], par exemple, conçoit l'émergence d'un concept comme un processus de confrontation itératif entre une théorie et la réalité, ce qui est d'ailleurs l'essence de la méthode scientifique. Ce que nous pouvons dire en tout état de cause, c'est qu'il est au moins une différence fondamentale entre un automate enchaînant des règles logiques et un être intelligent : ce dernier est capable de *prendre en compte la réalité* [ARN69, DEL94], ce qui donne à penser que les mécanismes liés à l'appréhension de cette réalité sont ceux qui jouent le plus grand rôle. Nous pensons qu'aujourd'hui de plus en plus, avec l'augmentation de la puissance des machines, il deviendra possible de mettre en œuvre des expériences d'IA mettant en jeu des machines capables d'interagir de façon complexe avec leur environnement, et de s'automodifier de façon cohérente.

Le système *Hammourabi* est encore pour sa part bien loin d'être doté de telles capacités ; nous devrions en avoir terminé une première version opérationnelle dans le courant du mois de Janvier 98, date à partir de laquelle nous entamerons la mise en œuvre d'expérimentations selon le plan esquissé au paragraphe 3 de cet article. Nous projetons aussi de mettre en œuvre une version d'*Hammourabi* téléopérable à travers le réseau *Internet*, et de mettre en place un site pour du travail coopératif utilisant la plate-forme *V*. Notre but à moyen terme est de réaliser des applications sur cette plate-forme, et de faire travailler *Hammourabi* à partir du comportement d'utilisateurs réels, avec des applications réelles, dans un environnement réel.

7. Références

[ARN69] Arnheim R. (1969) *La pensée visuelle*, Flammarion, Paris, 1974.
[CAZ97] Cazenave T. (1997) *Système d'apprentissage par auto-observation. Application au jeu de Go*, Thèse de l'université Paris 6, 1997.
[DEL94] Delacour J. (1994) *Biologie de la conscience*, PUF (Que sais-je ?), Paris.
[ETZ92] O. Etzioni, N. Lesh, R. Segal : Building Softbots for UNIX – *Working paper, University of Wahington*, 1992. Voir `http://www.cs.washington.edu/rese arch/softbots`.
[ETZ94] Oren Etzioni - *A Softbot-based interface to the internet* – CACM, Juillet 94.
[GRA92] Ghallab M., Grandjean P., Lacroix S., Thibault J.P. (1992) *Représentation et raisonnement pour une machine de perception multi-sensorielle*, PRC-GDR, Marseille, 19-21 Octobre 1992.
[KIC92] G. Kiczales - *Towards a New Model Of Abstraction in Software Egineering*. In Proceedings of the International Workshop on New Models for Software Architecture '92 ; Reflection and Meta-Level Architectures, 1992, Tokyo, Japan.
[KWO96] C.T. Kwok, D.S. Weld – *Planning to Gather Information*. AAAI'96 proceedings, 1996.
[LAU] Pour une information exhaustive sur *Snark*, le meilleur document est la thèse de Vialatte, soutenue à Paris 6 sous la direction de J.L. Laurière (je n'en ai malheureusement pas la référence exacte au moment où j'écris ceci).
[LEC98] V. Le Cerf, communication personnelle ; voir aussi V. Le Cerf (1998) *Reconstitution de déplacements de véhicules en milieu urbain à partir de données bruitées*, IPMU, Paris.
[LES98] H. Lesourd – *Le système* Hammourabi *: une intelligence artificielle semi-autonome opérant dans l'environnement système UNIX* – Rapport de présoutenance de thèse, LIP6, Université Paris 6, 1998.
[LES98-2] H. Lesourd – *Le système* Hammourabi *: un agent semi-autonome opérant dans l'environnement système UNIX* – RJCIA'98, 1998.
[NSI63] Newell A. & Simon H. (1963) *GPS : a program that simulates human thought*, in Computers & Thought, E. A. Feigenbaum, J. Feldman (eds.), McGraw Hill.
[PAC94] *François Pachet* - Pluggable advisors as epiphyte systems – *Rapport LAFORIA*, 1994.
[POI02] Poincaré H. (1902) *La science et l'hypothèse*, Flammarion, Paris, 1968.
[RIC95] Ricaud P. (1995) *Une approche pragmatique de l'abstraction appliquée à la modélisation de la stratégie élémentaire du jeu de Go*, Thèse de l'université Paris 6, 1995.
[ROG87] B. Roger – *Un système de dialogue intelligent avec un interlocuteur à la découverte du monde simulé par un logiciel quelconque*. Thèse de l'université Paris 6, 1987.
[SOW84] Sowa J.F. (1984) *Conceptual structures*, Addison-Westley, p. 69 (Percepts and concepts).

[THA98] Thagard P. & Verbeurgt K. (1998) *Coherence as constraint satisfaction*, Cognitive Science, vol. 22, January-March, pp 1-24.

RÉALISATION D'UN SYSTÈME MULTI-JEUX DE PLANIFICATION STRATÉGIQUE : APPLICATION AU JEU DE GO

Régis MONERET

Regis.Moneret@lip6.fr

Laboratoire d'Informatique de Paris VI (LIP6)

4, Place Jussieu 75005 - PARIS -

Résumé

Des expériences réalisées en Sciences Cognitives permettent de penser que les êtres humains utilisent des connaissances sous la forme de patterns complexes, pour analyser, classifier et enfin trouver des stratégies dans des jeux tels les Echecs et le Go. Dans cet article, nous présentons un système de planification à base de patterns logiques, capable de représenter des stratégies complexes pour de tels jeux : Après un pré-traitement de la position courante par des règles de modélisations fournies au système, une bibliothèque de plans est appelée afin d'une part, de raffiner l'évaluation de la position ébauchée par les règles de modélisation, et d'autre part, de déterminer le meilleur plan à appliquer dans la situation présente. Les plans se présentent comme des arbres de sous-plans récursifs de la forme attaque_amie - ripostes_adverses. A chaque plan est associé un pattern logique qui indique quand le plan peut-être testé. La connaissance du système réside de ce fait dans la bibliothèque des plans, qui est mise à jour grâce à un algorithme d'apprentissage automatique de type EBL.

Mots-clefs : Planification, Apprentissage Automatique, Jeux, Stratégies

1. Introduction

[Charness, 1977] et [De Groot, 1965] furent sans doute les premiers à suggérer qu'aux Echecs, les Grands Maîtres analysaient les positions du jeu à l'aide de patterns complexes tels que *les menaces*, *les clouages*, *les échecs à la découverte*, *les blocages*… [Berliner, 1977], [Pitrat, 1977], [Wilkins, 1980] montrèrent par la suite comment construire des programmes pour ce jeu en s'appuyant sur des patterns. Néanmoins, leurs systèmes étaient tous basés sur la recherche de **gains tactiques visibles** (capture de pièces, mats en *n* coups…). De même, au Go, [Cazenave, 1996], tente d'apprendre des théorèmes du jeu liés à des événements tactiques précis (connexion/déconnexion de chaînes, vie/mort de groupes, formations d'yeux…). Par contre, il n'existe pas à ma connaissance de programme **d'apprentissage automatique de stratégies**, ce qui est dû notamment à la difficulté de reconnaître des événements stratégiques (choix entre plusieurs tactiques par exemples). Il s'agit pourtant là d'un aspect essentiel pour un programme de jeu. C'est ce manque de vision stratégique qui explique en partie la faiblesse des programmes de Go actuels, qui gaspillent fréquemment leurs ressources dans des combats inutiles. Dans cet article, nous décrivons un système de planification destiné à servir de support à un programme capable d'apprendre automatiquement des stratégies. Bien que le système ait été conçu dans le cadre de jeux non coopératifs à deux joueurs, nous nous limiterons ici au cas du Go, jeu de plateau inventé en Chine il y a près de 4000 ans. Ce jeu a été choisi d'une part pour sa complexité, supérieure à celle des Echecs, qui le rend de ce fait, inattaquable (pour l'instant en tout cas) par des méthodes combinatoires, et d'autre part, pour sa richesse sur le plan stratégique (la réussite au jeu de Go dépend pour beaucoup de la capacité à mener à bien en parallèle des buts souvent contradictoires). Enfin, c'est un jeu positionnel, qui requiert de nombreuses connaissances, ce qui en fait un domaine idéal pour l'Intelligence Artificielle.

2. La Représentation des Jeux

Notre système a été conçu afin d'être applicable pour une classe de jeux bien particulier, à savoir les jeux : déterministes, à information complète, à deux joueurs, à plateau de jeu fini et à tour de jeu alterné.
Pour définir un nouveau jeu, nous fournissons au système :

une description du plateau de jeu et des pièces utilisées
des règles décrivant les coups légaux
des règles décrivant comment jouer un coup légal donné (par exemple, comment jouer une prise de pierres au Go)
des règles de terminaison de la partie (mat aux Echecs, règles chinoises au Go), qui vont permettre au système, par régression, de calculer un arbre des buts pour le jeu.
Des règles de modélisation de la position en cours.

REGLE contacts (COLOR C) :
  SI (i1,j1) est une intersection.
      SI Couleur(i1,j1) = C.
  SI (i2,j2) est une intersection.
  SI (i1,j1) <> (i2,j2).
  SI Couleur(i2,j2) = C.
  SI |i1-i2| + |j1-j2| < 3.
  ALORS (i1,j1,i2,j2) est une instance de LIENS(C).

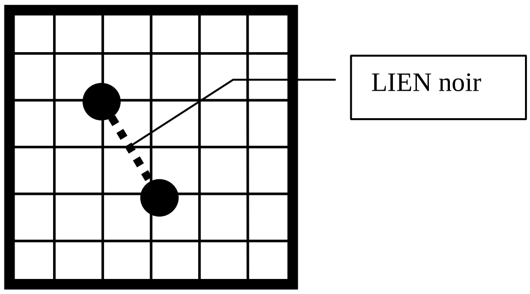


Figure 1 : Exemple de Règle de Modélisation

Ces dernières règles ne sont à vrai dire pas vraiment nécessaires pour décrire un jeu, mais elles permettent de donner au système des concepts (tels les "yeux" au Go ou les "fourchettes" aux Echecs), qui sont censés accélérer l'apprentissage de nouvelles connaissances par le système. Nous ne nous intéressons pas ici à la création de nouveaux concepts ; nous considérons en effet que les principaux concepts nécessaires pour bien jouer à ces jeux sont déjà connus des joueurs humains, et sont largement disponibles dans la littérature. Par contre, la seule connaissance livresque des concepts nous semble en soi insuffisante pour bien jouer. Ce qui nous intéresse, c'est l'apprentissage des stratégies/tactiques utilisant ces concepts. Le lecteur intéressé par la génération automatique de concepts pour les jeux pourra consulter [Utgoff, 1996] à ce sujet.

L'idée ici, est d'effectuer grâce aux règles de modélisation, un pré-traitement sur la position en cours. Aucune "lecture" de la position (= une recherche dans l'arbre de jeu) n'est effectuée. Ainsi par exemple, la règle énoncée dans la Figure 1 ne teste pas si les deux intersections (i1,j1) et (i2,j2) peuvent effectivement être connectées par noir. Cela implique que le programme devra par la suite faire ces calculs s'il le juge nécessaire, c'est à dire, si un de ses plans le demande. En résumé, ce pré-traitement est une modélisation grossière de la position, qui doit être raffinée plus tard par le système, qui "choisira" le niveau de certitude dont il a besoin en fonction de la "sensibilité" de la zone (les "zones de conflits" nécessitant plus de calculs que les zones calmes). Ceci doit permettre au système de concentrer ses ressources là où il en a besoin, c'est-à-dire en général, dans les "zones de conflits", qui sont déterminées par les plans stratégiques.

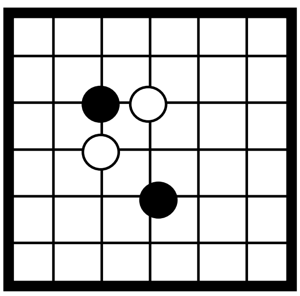


Figure 2 : Position dans laquelle la Règle de Modélisation de la Figure 1 est mise en défaut

3. Représentation des Stratégies

3.1. Opposition Tactique /Stratégie

S'il est déjà difficile de définir avec précision ce qu'on entend par stratégie, il est peut-être encore plus difficile de distinguer entre "tactiques" et "stratégie". Selon l'expérience du joueur, le niveau de description où l'on se place, telle manœuvre sera classée dans l'une où l'autre classe. Ainsi, par exemple, au Go, capturer un groupe ennemi est une manœuvre stratégique pour un débutant : il s'agit de coordonner la pose des pierres. Au contraire, cette même capture sera perçue par un bon joueur comme un simple outil tactique à sa disposition en vue de satisfaire des buts "plus élevés", comme faire du territoire par exemple. C'est la raison pour laquelle nous n'avons pas souhaité faire de distinction dans notre système entre les niveaux tactiques et stratégiques : les deux sont représentées par des plans. Ainsi, toute stratégie peut être vue comme une tactique par un plan plus élevé qu'elle et réciproquement, tout plan "tactique" est du point de vue de ses sous-plans, un plan stratégique. Ceci peut sembler en contradiction avec ce que nous avions dit en introduction sur les programmes de jeux actuels, pourtant, il n'en est rien : les systèmes d'apprentissage actuels se basent tous sur la capacité de reconnaître *dans l'environnement* quand une tactique/stratégie adverse a réussi (mat, prise d'une pièce, d'un groupe de pierres…). A ma connaissance, aucun d'eux n'est en mesure d'apprendre des stratégies au sens militaire du terme, c'est-à-dire **un choix entre plusieurs options tactiques**, qui ne se manifestera donc pas de manière *visible* sur l'échiquier/ le Goban.

3.2. Représentation des Plans Stratégiques

Nous avons mis en œuvre une nouvelle méthode de planification assez éloignée des modèles traditionnels tels que STRIPS ou HTN ([Willmott, 1997] et [Hu et al, 1997]) afin de représenter les tactiques/stratégies dans ce type de jeux. Chaque plan est constitué :

d'un "pattern logique" exprimé dans le même langage que les règles définissant le jeu, qui spécifie les conditions devant être vérifiées pour "essayer" le plan en mode simulation. Ainsi, dans le plan de la Figure3, le pattern logique exige qu'un groupe de pierres ennemies soit entouré par deux groupes amis plus "forts" que lui (la force d'un groupe étant mesurée par des règles de modélisation non présentées ici), pour que le plan soit essayé.

d'un arbre des sous-plans précisant les calculs devant être effectués en mode simulation lorsque le pattern logique a matché.

d'un but représenté sous **la forme d'un échange de ressources**. Ainsi, dans le plan de la Figure 3, le joueur qui déclenche le plan échange NIL (= rien dans ce cas précis) contre du territoire ou de l'influence. Le but du plan sert à juger en mode simulation si le plan a réussi ou non en comparant la position initiale avec les positions terminales après simulation. Dans le cas du plan de la Figure 3, la simulation est considérée comme réussie si le territoire ou l'influence des positions terminales est supérieure à celle de la position initiale. Ceci permettra par la suite de distinguer les plans qui n'ont pas pu être correctement appliqués (les buts du plan n'ont pas été atteints) des plans faux (les buts du plan n'ont été atteints, mais la position après l'application du plan est une position perdante pour le joueur qui a appliqué le plan)

de diverses informations permettant de comparer les plans entre eux (risque/valeur d'un plan…)

de conséquences de l'échec d'un plan en mode simulation : en effet, si le pattern logique d'un plan a été vérifié, mais que le système est incapable de faire aboutir le plan en mode simulation, c'est que peut-être la modélisation de la position est en cause. Dans ce cas, le système applique l'action décrite par le champ SI_ECHEC (dans le plan Figure 3, si la simulation de la capture de G a échoué, c'est que G devait être plus fort que prévu initialement. En conséquence de quoi, on renforce son statut).

```
PLAN (COLOR C) capturer_un_groupe_ennemi_faible_coincé_entre_deux_groupes_amis_forts
{
        BUT : NIL >> territoire ou influence
        PATTERN LOGIQUE:
                SI G ∈ groupe(~C).                  G est un groupe ennemi
                SI statut(G) < 2 .                  G n’est pas inconditionnellement vivant
                IF G1 ∈ groupe(C).                  G1 est un groupe ami
                IF géométrie(G1,G) = 1              G1 est à gauche de G
                IF statut (G) < statut (G1) .       G1 est plus fort que G
                IF G2 ∈ groupe(C) .                 G2 est un groupe ami
                IF géométrie(G2,G) = (1,0).         G2 est à droite de G
                IF statut (G) < statut(G2) .        G2 est plus fort que G
        SCHEMAS :
                ENTOURER (G) {
                                || FUIR(G)  { ENTOURER(G) }
                                || VIVRE(G) { CONNECTER (G1,G2) }
                                || AUTRE{ TUER(G) }
}
        VALEUR : #G .                               la valeur du plan est proportionnelle au
                                                    cardinal de G
        RISQUE : statut (G) .                       le risque est proportionnel à la force de G
SI_ECHEC : statut (G) := statut (G) + 1 .           G est plus fort que prévu
}
```

Figure 3 : Exemple de Plan

3.3. Simulation d’un Plan :

L'idée de base de la représentation des tactiques/stratégies est illustrée par la Figure 4 :
Une stratégie est définie comme un chemin d'une position initiale vérifiant le pattern logique vers une ou plusieurs positions terminales satisfaisant le but du plan. Pour tester un plan dans une situation donnée, le système de planification appelle récursivement les sous-plans définis dans le schéma du plan qu'il essaye. Ainsi, dans le plan de la Figure 3, le système commence par appeler le sous-plan "ENTOURER" (non donné ici) pour savoir s'il peut parvenir à entourer le groupe G. Dans l'affirmative, le plan précise que seules 3 options sont ouvertes à l'ennemi :

- FUIR (= briser l'encerclement)
- VIVRE sur place (en faisant deux yeux)
- AUTRE (réponse ennemie non liée à l'attaque du groupe G, ce qui revient à sacrifier G)

Le système commence par appeler le sous-plan FUIR. Si ce sous-plan échoue (du point de vue de l'ennemi), le système considère que l'option FUIR n'est pas possible et passe à l'étude des options restantes. Si, au contraire, FUIR retourne un ou plusieurs coups susceptibles d'aider G à fuir, le système choisit le meilleur (grâce aux paramètre valeur/risque retournés par le sous-plan FUIR), joue ledit coup, et, après avoir appelé à nouveau les règles de modélisation sur la position ainsi obtenue, appelle le sous-plan ENTOURER à nouveau. Si ce dernier échoue, c'est que l'ennemi peut s'enfuir, et que le plan **capturer_un_groupe_ennemi_faible_coincé_entre_deux_groupes_amis_forts** a échoué. Le système passe alors à l'étude des plans restants de la bibliothèque. Si ENTOURER réussit, le système conclut comme précédemment que l'option FUIR n'est pas réalisable par l'ennemi, et passe à l'étude de l'option VIVRE sur place. Finalement, si cette option est également mauvaise pour l'adversaire, il passe à l'option AUTRE. Ce dernier sous-plan est un peu particulier, car il suppose que l'ennemi va passer son tour.
Lorsque la simulation dans le monde des plans est achevée, et que le plan est jugé applicable, le résultat (= l'arbre des coups joués de part et d'autre par les deux joueurs) de cette simulation est ensuite placé dans la liste des plans applicables.

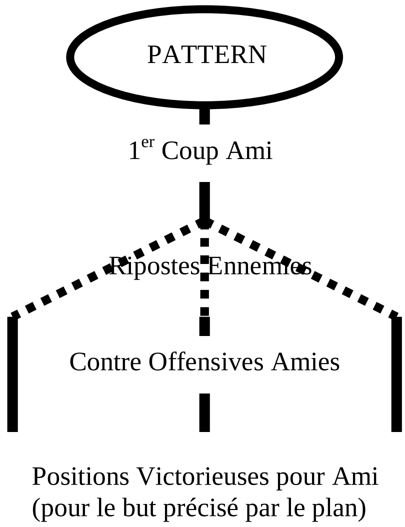


Figure 4 : Modèle de Représentation d'une Stratégie

3.4. Langage des Plans et Métaplans:

Afin de représenter des stratégies complexes, un langage de plan a été mis au point. Celui-ci permet en autre de faire des tests et des branchements à l'issue de l'appel d'un sous-plan. Ainsi, par exemple, le plan Figure 3 suppose qu'une fois que le sous-plan ENTOURER a été appelé avec succès pour la deuxième fois, le groupe G n'est plus en mesure de FUIR à nouveau. En fait, le plan véritable est :

```
        SCHEMAS :
                ENTOURER (G) {
                              || LABEL $1 : FUIR(G)  { ENTOURER(G)  { $1} }
                              || VIVRE(G) { CONNECTER (G1,G2) }
                              || AUTRE{ TUER(G) }
}
```

Les sous-plans FUIR et ENTOURER sont ainsi appelés successivement jusqu'à ce que l'un d'eux ne soit plus réalisable. Ceci augmente considérablement la puissance d'expression des plans, mais peut engendrer des boucles infinies, en cas d'erreur dans un sous-plan. Pour le moment, le problème est géré manuellement.

Dans le paragraphe 2.1, nous avons expliqué pourquoi nous avions choisi de représenter dans un même formalisme tactiques et stratégies. Pourtant, notre intérêt premier est d'être en mesure d'apprendre des stratégies au sens militaire du terme, c'est à dire, des fonctions de choix entre plusieurs tactiques/stratégies. Pour cela, nous avons besoin d'être en mesure de représenter ces options dans le pattern logique. En d'autres termes, nous allons chercher à faire apprendre au système des Métaplans, c'est à dire, des plans qui vont coordonner d'autres plans, du type " Si on vient de jouer le plan 'Ouverture1' au coup précédent, il faut maintenant jouer de préférence le plan 'Créer_du_territoire' ", ou encore "Si plusieurs plans conseillent de jouer un même coup, alors, il faut accorder plus d'importance à ce coup"… Pour cela, **nous avons introduit un aspect réflexif dans le système de planification** : La simulation des plans elle-même, modifie la modélisation de la position en introduisant notamment :
l'historique des coups joués
l'historique des plans joués par le système (et d'une manière générale de tous les plans considérés comme réalisables pour une position donnée)
des informations sur les plans de la bibliothèque (leurs buts/valeurs/risques…)
Toutes ces informations pouvant être utilisées dans le pattern logique de nouvelles stratégies.

4. Vision d'ensemble du Système de Planification

La Figure 5 montre le schéma directeur du système : à partir d'une position de départ (en bas à gauche), le système, par un simple matchage des règles de modélisations qui lui ont été fournies, construit une représentation "enrichie" de la dite position (en haut à gauche). Cette représentation est ensuite utilisée pour rechercher les plans de la bibliothèque susceptibles d'être appliqués (par matchage de leur pattern logique associé). Une fois qu'un plan a passé cette étape, il est ensuite "essayé" dans le "monde des plans", c'est à dire qu'il appelle récursivement ses sous-plan comme expliqué précédemment, afin de savoir s'il peut effectivement réussir dans cette situation précise. Si le test dans cet univers virtuel a été couronné de succès, le plan est alors noté, puis le résultat est ensuite stocké dans une liste de résultats. A la fin de cette étape, le système dispose de la liste des plans qu'il juge applicables. Si aucun plan n'est applicable dans le cas précis, la procédure d'apprentissage est alors appelée pour créer un nouveau plan. Si au contraire, la liste des résultats n'est pas vide, le système choisit alors le plan qui a la meilleure note, et joue les coups prévus par la simulation. Si un hiatus se produit entre ce que prévoyait le plan et la réalité, la procédure d'apprentissage est à nouveau appelée afin de corriger le plan existant. Sinon, si les résultats réels concordent avec la simulation, le plan est exécuté jusqu'à la fin, puis le système recommence le tout.

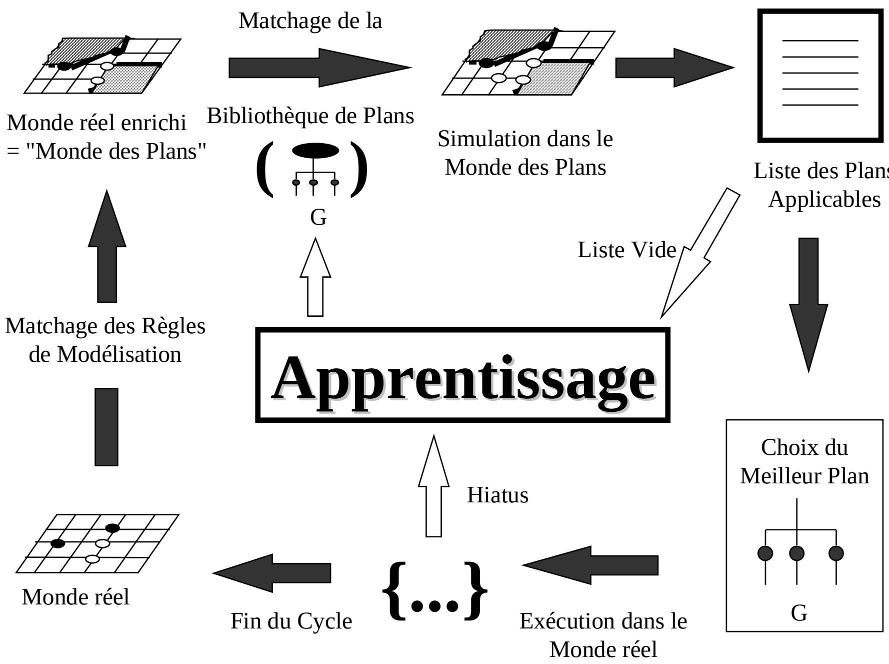


Figure 5 : Vue d'ensemble du système

5. Conclusions / Perspectives

Dans cet article, nous avons présenté un système de planification capable de représenter des stratégies complexes dans une classe de jeux contenant les Echecs et le Go. Ce système a été conçu afin de servir de support à un programme d'apprentissage automatique qui sera chargé de corriger les plans existants ainsi que d'en créer de nouveaux lorsque des situations encore inconnues se présenteront à lui.

6. Références

[Berliner, 1977] H.J. BERLINER « A Representation and some Mechanism for a Problem-Solving Chess Program. » in *Advances in Computer Chess I*, M.R.B. Clarke ed., Edinburgh, 1977.

[Cazenave, 1996] T. CAZENAVE « Explanation-Based Learning in Games : an Application to the Game of Go. » in *Proceedings of the 8th International Conference on Artificial Intelligence Applications* (EXPERSYS-96), Paris, 1996.

[Charness, 1977] N. CHARNESS « Chess Skill in Man and Machine. » in *Human Chess Skill*, Springer-Verlag, 1977.

[De Groot, 1965] A. De GROOT « Thought and Choice in Chess. » The Hague, 1965.

[Hu et al, 1997] Shui HU and Paul E. LEHNER « Multipurpose Strategic Planning in the Game of Go. » IEEE Transactions on Pattern Analysis and Machine Learning, vol.19, N° 9, 1997.

[Pitrat, 1977] J. PITRAT « A Chess Combination Program Which Uses Plans. » Artificial Intelligence n°8, pp 275-321, 1977.

[Wilkins, 1980] D. WILKINS « Using Patterns and Plans in Chess. » Artificial Intelligence n°14, pp 165-203, 1980.

[Willmott, 1997] Steven WILLMOTT « *Adversarial Planning Techniques and the Game of Go*. » MS of Edinburgh, 1997

[Utgoff, 1996] P.E. UTGOFF « Feature Function Learning for Value Function Approximation. » Technical Report 96-09, Department of Computer Science, University of Massachusetts, 1996.
(disponible via ftp at ftp://ftp.cs.umass.edu/pub/techrept/techreport/1996/UM-CS-1996-009.ps)