COLLOQUE METACONNAISSANCE DE BERDER, 19-21 SEPTEMBRE 2001

Résumé :

Ce rapport contient la majeure partie des interventions effectuées lors du colloque Métaconnaissances qui s'est tenu du 19 au 21 septembre 2001, sur l'île de Berder. Il fait état d'un certain nombre de travaux qui ont été réalisés au cours de l'année 2001 par l'équipe Métaconnaissance du LIP6.

Mots-clés :

Intelligence Artificielle, Métaconnaissances, Monitoring, EIAO, Jeux

---

Abstract :

These proceedings contain selected papers from the Artificial Intelligence Workshop that was held September, 19-21 2001, on Berder island. It is a summary of the LIP6 Meta-knowledge Group work in 2001.

Keywords :

Artificial Intelligence, Meta-knowledge, Monitoring, ITS, Games

SOMMAIRE

INTRODUCTION

L'équipe Métaconnaissance du LIP6 ainsi que des chercheurs d'EDF, de Paris 5, de Paris 8 et de Dauphine se sont réunis du 19 au 21 septembre 2001 dans l'île de Berder. Ce rapport contient quelques unes des interventions qui y ont été faites.

Les deux premiers textes sont liés au système MACISTE ; Jacques Pitrat montre que MALICE, le résolveur général de problèmes faisant partie de MACISTE, devrait devenir un chercheur en IA à part entière. En effet, la plus grande part du succès des systèmes actuels vient de l'intelligence de leur concepteur. Il faudrait réalise des systèmes capables de plus d'autonomie, et en particulier qui pourraient créer eux-mêmes des systèmes d'IA à notre place. Basile Starynkevitch raconte comment il a vécu son expérience d'utilisateur du système MACISTE et indique quelques améliorations qui seraient souhaitables.

Nous avons ensuite la description d'un aspect de deux systèmes en cours de réalisation. Tristan Pannérec montre comment son système général, MARECHAL, résout un problème. Ce papier prend pour exemple la solution d'emplois du temps afin de mieux montrer les mécanismes du système. Par ailleurs, l'équipe qui travaille sur le système d'EIAO Combien ? traite ici d'un problème essentiel pour ces systèmes : comment peut-on détecter les erreurs des élèves ?

Tristan Cazenave nous montre comment le puzzle Eternity a fini par être résolu. Michel Masson présente une synthèse sur les méthodes qui permettent de généraliser à partir d'un petit nombre d'exemples ; elles s'appliquent donc à des cas où les méthodes statistiques classiques sont peu efficaces.

Nous remercions Fabrice Kocik qui a bien voulu se charger de l'organisation matérielle de ce colloque, qui s'est parfaitement déroulé, et Tristan Pannérec qui est l'éditeur de ces actes.

Jacques Pitrat

MALICE, NOTRE COLLEGUE

Jacques Pitrat
CNRS-LIP6
Université Pierre et Marie Curie

Résumé : L'étude de papiers récents d'IA montre qu'une grande partie du travail fait par les chercheurs qui conçoivent un système pourrait être automatisé ; nous examinerons deux articles dans cette optique. Nous présenterons ensuite les travaux faits dans cette direction avec le résolveur général de problèmes MALICE. Après avoir décrit rapidement la méthode de résolution utilisée par MALICE, nous verrons comment un tel système peut effectuer de façon autonome une partie du travail de conception actuellement fait par des chercheurs humains : il analyse les résultats qu'il obtient et il définit de nouvelles expériences afin de récolter les éléments qui lui permettront de découvrir de nouveaux principes, d'améliorer les méthodes de résolution et de vérifier que les modifications apportées sont bien satisfaisantes.

Mots-clés : métaconnaissance, résolution de problèmes, amorçage,

1. Un collaborateur indispensable

La conception des systèmes actuels d'IA revient entièrement au chercheur qui crée le système. Cela présente un double inconvénient. Nous pouvons d'abord nous demander où est l'intelligence : est-ce celle du système ou celle de celui qui l'a conçu et mis au point ? Mais aussi, en réalisant tout nous-mêmes, nous n'avons pas la possibilité de tout bien faire, faute de temps et de patience. Il vaudrait mieux nous faire aider dans cette tâche. Plutôt que d'avoir deux phases, une où nous créons seuls un système et une où nous lui faisons résoudre des problèmes, il vaudrait mieux que le système intervienne aussi dans la première phase. [Newell 1960] avait déjà proposé que GPS trouve des différences pour chaque application de GPS, mais ce projet n'a malheureusement jamais été implémenté.

Les capacités des systèmes d'IA et des chercheurs sont complémentaires. Nous savons découvrir de nouveaux concepts, mais nous savons mal, non seulement faire une recherche combinatoire, mais aussi effectuer de nombreuses expériences, les analyser pour en déduire des modifications ainsi que l'idée d'autres expériences qui auront encore plus de chance de donner des informations utiles. Dans la création d'un système, la phase d'adaptation des paramètres (le "tuning") qui régissent le comportement de ce système, est très délicate. Cela prend beaucoup de temps au chercheur qui, de plus, le fait mal parce qu'il n'a pas assez de temps pour faire toutes les expériences qui seraient nécessaires et pour les analyser. Faute de place, il ne peut en indiquer tous les résultats dans ses articles ; c'est d'ailleurs pour cela qu'il est difficile de reproduire les performances d'un système créé par un autre chercheur. En effet, si nous pouvons reconstituer l'ossature d'un système, nous ne connaissons avec assez de précision la valeur, et même la nature, de tous les paramètres qui interviennent.

La raison fondamentale du succès d'un amorçage est la collaboration d'un système avec son auteur, chacun apportant ses qualités ; les défauts de chaque élément de ce couple est compensés par les qualités de l'autre. Dans le développement de MALICE, le résolveur général de problèmes du système MACISTE, j'essaye actuellement de me faire aider par le système lui-même. Nous verrons ce qui a été fait et les étapes qui sont en cours de réalisation. Mais avant de rentrer dans la description de MALICE, je vais présenter deux excellentes recherches menées récemment en IA pour montrer qu'une grande partie du travail des chercheurs aurait pu être automatisé.

2. Deux exemples de développement classique de système d'IA

Je prends comme premier exemple de développement un système récent, Rolling Stone, dû à A. Junghanns et J. Schaeffer [Junghanns 2001,]. J'ai choisi ce système pour deux raisons : il a obtenu d'excellents résultats et le papier contient une description détaillée des étapes de sa mise en œuvre. Loin de moi l'idée de

critiquer les auteurs de ce travail remarquable qui est caractéristique des réalisations qui ont les meilleurs résultats actuellement en IA. Je l'ai choisi pour ses qualités afin de montrer qu'il est possible d'automatiser une partie du travail qui a été fait par les auteurs de ce système. De telles réalisations sont d'ailleurs indispensables pour nous donner l'idée des métaconnaissances nécessaires pour construire justement des métasystèmes qui créent des systèmes analogues à ceux que ces auteurs ont créés. J'isolerai à chaque étape de leur démarche les principes que j'en ai tirés ; on pourrait les donner à un système général de résolution de problèmes. J'ai aussi énoncé des métaprincipes dans le cas où la gestion des principes n'est pas évidente ; ils sont en italique.

Rolling Stone résout des problèmes de Sokoban. Un tel problème est défini par une grille 20X20 dont certaines cases sont occupées par des murs infranchissables, qui peuvent se trouver même au milieu de la grille. Au départ, un certain nombre de pierres sont dans ce labyrinthe ainsi qu'un acteur, le pousseur, qui peut pousser une pierre sur la case située derrière si elle est vide. Le pousseur ne peut, ni passer à travers un mur ou à travers une pierre, ni tirer une pierre : si le seul passage, qui lui permettrait d'aller se positionner derrière, est bloqué, le problème est insoluble. Pour réussir le problème, le pousseur doit collecter toutes les pierres dans une zone de la grille donnée au départ. Dans les solutions, on ne fait pas figurer les mouvements du pousseur et on ne tient pas compte du nombre de ses déplacements pour évaluer la qualité d'une solution : il suffit de s'assurer que le pousseur dispose d'au moins un chemin pour se mettre derrière la pierre qu'il va pousser. La description d'une solution comporte uniquement la séquence des déplacements de pierres qui ont été effectués. Ces solutions peuvent être très longues, un des 90 problèmes posés au système a une solution comportant 674 poussées de pierre.

Les auteurs ont naturellement commencé par examiner si un résolveur général existant pouvait résoudre ce type de problème. Ils ont pris Blackbox qui a résolu en quelques secondes les problèmes avec une ou deux pierres, mais qui s'est révélé incapable de résoudre les problèmes de l'ensemble de test qui comportent déjà six pierres pour les plus faciles.

> Commencer par examiner le comportement des systèmes déjà existants.

Il faut donc réaliser un système nouveau qui soit plus performant que les systèmes généraux existants. L'idée est de choisir un algorithme général de recherche combinatoire auquel les auteurs ajouteront par la suite des méthodes pour mieux sélectionner les essais et pour que la fonction inclue dans l'algorithme soit plus précise. L'algorithme choisi est le Iterative Deepening A* (IDA*). Ils ne donnent pas les raisons de ce choix ; cet algorithme classique demande pour fonctionner un estimateur de la borne inférieure de la distance à la solution. L'estimateur de départ a été pris très simple, en cumulant les distances de chaque pierre à la case but la plus proche ; c'est directement inspiré de l'analyse means-ends qui examine les différences entre la situation actuelle et le but à atteindre.

> Choisir un algorithme qui sera la base du fonctionnement du système. Il vaut mieux que l'on puisse améliorer facilement cet algorithme, soit en modifiant les fonctions dont il a besoin, soit en ajoutant des mécanismes de sélection plus stricts.
>
> Au départ, il faut choisir les fonctions figurant dans l'algorithme aussi simples que possible. On peut s'inspirer de la différence entre la situation actuelle et le but à atteindre pour chaque élément sans tenir compte des contraintes apportées par les interférences entre les buts de chaque élément.

Aucun des 90 problèmes n'a été résolu avec cet version initiale de l'algorithme. La première idée est d'améliorer l'estimateur. Quand on mesure la distance d'une pierre à la case but la plus proche, on peut affecter toutes les pierres à la même case ; aussi, maintenant exige-t-on qu'une case but ne puisse être affectée qu'à une pierre. Aucun problème n'est encore résolu.

> Si les résultats sont mauvais, on peut affiner les fonctions figurant dans l'algorithme en tenant compte de contraintes apportées par les liens entre les buts de chaque élément.

Il y a en général beaucoup de possibilités de commuter les actions : en poussant d'abord la pierre P1 puis la pierre P2 ou au contraire la pierre P2 puis la pierre P1. D'où l'idée d'introduire une table de transposition pour éviter de considérer plusieurs fois la même situation. 5 problèmes sont maintenant résolus.

Si les actions peuvent commuter, donnant souvent lieu à des situations identiques à des emplacements différents de l'arborescence, il faut introduire un dispositif pour repérer les cas où l'on retrouve plusieurs fois la même situation afin de ne la développer qu'une fois.

Les tables de transposition sont un excellent moyen pour reconnaître si l'on a déjà des informations sur une situation.

Il vaut mieux considérer en priorité les situations qui ont le plus de chance de mener à une solution. Pour cela on ordonne les actions. Les auteurs utilisent le principe d'inertie en privilégiant les coups qui déplacent la même pierre que les précédents. Il y a cette fois baisse des performances : plus que quatre problèmes résolus.

Il est important de trouver un moyen d'ordonner les actions de façon à ce que l'on considère en premier les actions qui ont le plus de chance de mener à une solution.

Il est souvent meilleur d'assurer une certaine continuité dans la suite des actions, c'est à dire d'envisager en premier les actions qui ont des liens en commun (par exemple même acteur) avec les actions qui viennent d'être faites. (Principe d'inertie).

Un problème de monitoring s'est posé ici : bien que la dernière modification amène une baisse des performances, les auteurs décident de la garder parce que cette baisse est faible. Ils pensent qu'elle est due à un manque de chance et que son principe est sain. Ils vérifient sur le programme final que ce mécanisme est efficace et qu'il améliore sensiblement le comportement du système. Nous rencontrons là un métaprincipe qui porte sur le monitoring de la construction de la solution, et non plus sur la construction elle-même.

Si l'application d'un principe n'amène pas une amélioration des performances, mais une légère baisse, si ce principe est sain, il ne faut pas abandonner son application, car cela peut être dû à un manque de chance. Mais il convient de vérifier plus tard qu'il apporte bien finalement une amélioration effective.

Dans certains cas, il est facile de voir rapidement qu'une situation ne peut pas mener à une solution. Dans le cas du Sokoban par exemple, le pousseur ne peut que pousser. Si une pierre se trouve contre un mur, il ne peut que la faire glisser contre ce mur. S'il n'y a pas de creux qui lui permettrait de passer derrière et s'il n'y a pas de case but le long de ce mur, il est impossible de l'amener à une case but. Aussi, il faut arrêter immédiatement la recherche dans cette branche de l'arborescence, sans perdre son temps à essayer d'amener les autres pierres au but. Les auteurs donnent à Rolling Stone divers types de situations de ce type qu'il est facile de vérifier sur une position. On revient à 5 problèmes résolus.

S'il existe des situations où il existe des patterns simples tels qu'il est impossible d'avoir une solution si un de ces patterns est présent, les donner au système qui arrêtera le développement d'une branche dès qu'il en détecte un.

Il existe souvent des séquences de coups sur la même pierre qui peuvent être considérées comme un seul macro-coup : par exemple quand une pierre est en vue directe de sa case but, il suffit de la pousser par une série d'actions qui vont l'y amener tout droit. Il en est de même si elle rentre dans un tunnel où l'on ne peut qu'avancer tout droit : on va directement à la sortie du tunnel. On peut donc considérer que l'on exécute un seul macro-coup, ce qui diminue la taille de l'arborescence. De façon analogue, dans [Pitrat 1966], le système pouvait découvrir des métathéorèmes qui résumaient en une seule étape l'application de toute une série de règles. 17 problèmes sont maintenant résolus.

Etablir des macro-actions qui résument en une seule étape une séquence d'actions qu'il est en général judicieux de faire en entier.

Les macro-actions sont très puissantes, nous observons une discontinuité dans les performances : on passe de 5 à 17 problèmes résolus. Il faut donc en favoriser l'application : on va privilégier les coups qui mènent à une situation où l'on peut appliquer une macro-action.24 problèmes résolus.

Favoriser les coups qui mènent à des situations où une méthode efficace de résolution a des chances de s'appliquer.

Une situation ne peut parfois mener à la solution sans que cela puisse se voir en utilisant uniquement les patterns locaux montrant une impossibilité. Par exemple, une série de pierres fait barrage, empêchant le pousseur d'atteindre la zone située derrière, car ces pierres sont trop près d'un mur pour pouvoir livrer passage si on les pousse. Pour voir cela, on essaye de résoudre des problèmes sur une partie restreinte du Sokoban. Le système tente de comprendre les raisons d'un échec, il en déduit un pattern minimal en éliminant tout ce qui n'est pas lié à l'impossibilité de résoudre le problème ; il l'ajoute aux patterns dont il teste la présence pour décider l'impossibilité d'atteindre la solution à partir de certaines situations. De plus, quand il arrive à trouver la solution du problème local, mais en plus d'étapes que prévu, il retient le pattern pour augmenter d'autant la fonction évaluant la proximité de la solution dans les cas où ce pattern est présent. On y gagne donc sur deux tableaux : on élimine des situations impossibles et on évalue les autres avec plus de précision. C'est là le point le plus délicat et le plus important car, après cette amélioration, on saute à 48 problèmes résolus.

Utiliser des méthodes de "Explanation Based Learning" afin de comprendre pourquoi une situation ne peut mener à la solution. On extrait des patterns qui généralisent cette situation et on les ajoute aux patterns d'échec évidents déjà trouvés.

Si la combinatoire locale montre que l'estimation de la distance au but est incorrecte si certains patterns sont présents, ajouter une pénalité égale à cette erreur d'estimation quand le pattern est présent.

Il vaut mieux avoir une certaine continuité dans la recherche et ne pas sauter continuellement d'un coin du Sokoban à un autre. On définit l'influence d'une action sur une autre et on privilégie les actions sur lesquelles les actions précédentes ont une forte influence. En liant ainsi les coups, on crée ainsi des sortes de plans implicites. On passe à 50 problème résolus.

Favoriser les actions dont l'existence dépend le plus des actions précédentes.

La présence d'un pattern apportant des pénalités à une position indique que cette situation est complexe ; elle est sans doute encore insuffisamment pénalisée. 0n multiplie donc par un facteur de près de 2 la valeur de cette pénalité. Cela n'élimine pas l'examen d'une action, mais le retarde. Ceci est très heuristique ! Toutefois, le nombre de problèmes résolus passe à 54.

Si un élément indique une difficulté à résoudre un problème, augmenter la valeur de la pénalité due à cet élément davantage que ce qui a été strictement calculé. Ce principe est douteux.

Quand un principe est douteux, vérifier avec soin qu'il améliore effectivement les performances.

Il arrive que la solution d'un problème dépende beaucoup des paramètres de départ donnés au système : pour un ensemble de valeurs de ces paramètres, on a la solution en 2 minutes et pour un autre on n'a aucune solution au bout de 2 heures. Un mécanisme de redémarrage avec de nouveaux paramètres est donc fort utile : il s'était en particulier montré très efficace dans la recherche de phrases réflexives [Pitrat 1996]. Aussi Rolling Stone repart avec un autre mécanisme de sélection des ex aequos au bout d'un certain temps. On passe à 57 problèmes résolus sur les 90 de l'ensemble test.

Quand on est assez loin de la solution bien que l'on ait déjà dépensé pas mal de temps, repartir avec un autre ensemble de paramètres plutôt que de continuer jusqu'à l'épuisement du temps alloué.

Ce travail montre clairement la démarche de ses auteurs, de sorte que l'on peut en tirer des idées pour un système général qui construirait, en s'aidant d'expérimentations, un système résolvant une certaine famille de problèmes. La description d'autres systèmes est intéressante pour ce même point de vue, par exemple, D. Schuurmans et F. Southey [Schuurmans 2002] ont réalisé un système pour résoudre des problèmes SAT. Ils ont établi des principes que je généralise à un cadre plus vaste :

Privilégier une action qui minimise le nombre de contraintes qui ne sont pas satisfaites.

Favoriser une action qui augmente la mobilité, c'est à dire qui crée une différence importante entre la valeur des variables actuelles et celles qu'elles avaient dans des situations plus haut dans l'arborescence.

Favoriser des actions qui augmentent la couverture de l'espace de recherche, c'est à dire qui évitent de laisser de larges zones non explorées.

Avec tous ces principes, il faudrait réaliser un système qui créerait des systèmes adaptés à chaque problème particulier. Il est caractéristique que nous connaissons bien la plupart de ces principes et que nous les avons souvent utilisés dans des domaines complètement différents du Sokoban ou des problèmes SAT. Il existe un panier de méthodes qui peuvent s'appliquer à de nombreux problèmes ; un système automatique devrait pouvoir faire comme nous et piocher dans ce panier pour construire un résolveur performant pour chaque nouveau problème. Je vais maintenant indiquer comment je compte donner à MALICE la possibilité de faire des expérimentations et d'en déduire des méthodes de résolution adaptées à chaque famille de problèmes.

3. Présentation rapide de MALICE

MALICE est la partie résolveur de problèmes de MACISTE. Il est basé au départ sur le formalisme de description de problèmes de ALICE [Laurière 1976] ; ses méthodes sont pour beaucoup inspirées de celles utilisées par ALICE. Il contient cinq parties : un ensemble de règles qui font progresser le système vers la solution, des séquences d'actions qui indiquent quelles règles considérer quand un événement se produit, des connaissances pour faire un choix quand il faut développer une arborescence, une expertise de création de programmes spécifiques résolvant chacun un problème de façon combinatoire et enfin un mécanisme général qui gère le tout.

Dans le formalisme d'ALICE, on résout un problème en trouvant des liens entre les éléments de l'ensemble de départ et des éléments de l'ensemble d'arrivée d'un certain nombre de correspondances. Ces correspondances sont caractérisées par la valeur des degrés maximaux et minimaux des ensembles de départ et d'arrivée : par exemple, pour une fonction, le degré minimal et le degré maximal de l'ensemble de départ valent 1 alors que ceux de l'ensemble d'arrivée ne sont pas définis. Les éléments des correspondances doivent souvent satisfaire aussi un certain nombre de contraintes, en général algébriques.

3.1. Les règles

Une règle indique que si certaines conditions sont remplies, on a le droit d'accomplir un certain ensemble d'actions. C'est une façon déclarative d'exprimer des connaissances car, à la différence des actions conditionnelles, on n'est pas obligé d'appliquer une règle quand ses conditions sont satisfaites. Une règle ne contient pas d'informations sur les situations où l'on doit l'appliquer. Certaines des actions d'une règle font progresser le système vers la solution, par exemple en enlevant un lien possible entre un élément de l'ensemble de départ et un élément de l'ensemble d'arrivée ou en indiquant qu'il y a une contradiction. D'autres actions ont pour effet de changer la formulation du problème, donc agissent sur sa représentation, par exemple en créant de nouvelles contraintes ou en déterminant la valeur d'un degré minimum ou maximum pour un ensemble de départ ou d'arrivée d'une des correspondances. Une règle contient seulement les conditions nécessaires pour que l'on ait le droit de l'appliquer. Il est possible qu'une certaine règle puisse être appliquée de façon différente selon les variables y figurant qui sont connues. Pour préciser cela, donnons un exemple, la règle R6 :

R6 contient les variables suivantes : X(expression), Y(expression), R(contrainte), DX(nœud départ), DY(nœud départ). Elles sont typées, X est une expression algébrique, R une contrainte (donc un type particulier d'expression), DX est un nœud d'un ensemble de départ d'une correspondance.

Ces variables sont reliées par les relations suivantes :

$$X \in \text{UNDESFILS}(R)$$

$$Y = \text{AUTREFILS}(R,X)$$

$$R \in \text{CONTRAINTES}(DX)$$

$$DX \in \text{VARNŒUDS}(X)$$

$$DY \in \text{VARNŒUDS}(Y)$$

$$R \in \text{CONTRAINTES}(DY)$$

R est une contrainte et la fonction UNDESFILS a pour valeur l'ensemble des arguments de la connective principale, deux si c'est une connective binaire comme l'égalité. La fonction AUTREFILS(R,X) a valeur l'argument de la connective principale de R qui n'est pas X quand cette connective est binaire. DX est un nœud d'un ensemble de départ et CONTRAINTES(DX) est l'ensemble des contraintes qui contiennent la variable correspondant à ce nœud. VARNŒUDS(X) est l'ensemble des nœuds qui correspondent aux variables contenues dans l'expression X.

Nous avons maintenant le cœur de la règle qui est une production (et non une action conditionnelle) : si les prémisses sont vraies, on a le droit (mais pas le devoir) d'établir les faits qui sont en partie résultat :

$$\text{CORPI}(R) = @EQ$$

$$\text{SUP}(X) < \text{INF}(Y)$$

$$\Rightarrow \text{CONTRADICTION}$$

Cette règle exprime que si R est une contrainte d'égalité, et si une borne supérieure d'un des termes est inférieure à une borne inférieure de l'autre, alors on a une contradiction. La fonction COPRI a pour valeur la connective principale de l'expression qui est son argument, et l'objet @EQ représente l'égalité. La fonction SUP a pour valeur une borne supérieure de l'expression argument et la fonction INF une borne inférieure. Notons qu'il n'est pas nécessaire d'avoir la borne supérieure ou inférieure précise : si les fonctions SUP et INF ne donnent qu'une valeur approchée, moins difficile à calculer, la règle est correcte, quoiqu'elle ne s'appliquera pas aussi souvent qu'il le faudrait. La seule condition est que la borne inférieure soit sûrement une borne inférieure. Il peut être intéressant dans un premier temps d'utiliser des fonctions peu coûteuses qui peuvent donner des valeurs éloignées de la valeur optimale, pour n'utiliser des fonctions plus précises que si les deux termes de l'inégalité sont proches l'un de l'autre ; c'est une expertise de monitoring qui devrait gérer de telles décisions.

Une règle est compilée en autant de programmes C qu'il y a de façons de l'utiliser. Pour la règle R6, on peut être amené à s'en servir :

1. quand on a une nouvelle contrainte R ; X et Y seront définis par les deux premières relations, les autres seront inutilisées, car inutiles. Remarquons qu'elle peut être appliquée deux fois, X prenant successivement comme valeur le premier et le deuxième terme de la contrainte.
2. quand un nœud de départ DX a vu diminuer la valeur maximale de ses valeurs possibles, ce qui peut diminuer la valeur de SUP(X) si la variable correspondant à DX apparaît dans le terme X ; en effet, dans ce cas, SUP(X) < INF(Y) a plus de chances d'être vrai. MALICE ajoute alors les quatre premières relations, trois pour définir R, X et Y qui sont nécessaires pour appliquer la règle et la quatrième pour vérifier que DX se trouve bien dans X.
3. quand un nœud de départ DY a vu augmenter la valeur minimale de ses valeurs possibles, ce qui peut augmenter la valeur de INF(Y) si la variable correspondant à DY apparaît dans le terme DY. MALICE ajoute alors les deux premières et les deux dernières contraintes.

Tout ce travail de compilation est fait automatiquement à partir du moment où un nouvel appel à une règle figure dans une séquence d'actions.

3.2. Les séquences d'actions

Une grande partie de l'expertise d'un expert vient de ce qu'il a compilé ses connaissances : il sait dans chaque situation ce qu'il faut faire et il ne perd pas de temps à effectuer des essais inutiles. J'ai essayé de donner à MALICE une possibilité analogue avec les séquences d'actions.

MALICE connaît un certain nombre d'événements. Quand un événement se produit, il déclenche la séquence d'actions correspondante. Dans des contextes différents, les séquences d'actions peuvent être aussi différentes. Les événements connus actuellement par le système sont :

1. Le début de la résolution d'un problème
2. Une contradiction
3. La création d'une contrainte
4. La détermination de la valeur d'un degré maximum ou minimum pour un ensemble de départ ou d'arrivée
5. L'enlèvement d'un lien possible entre un élément d'un ensemble de départ et un élément d'un ensemble d'arrivée
6. La détermination d'un lien certain entre un élément d'un ensemble de départ et un élément d'un ensemble d'arrivée
7. Le changement de la valeur minimale ou maximale possible d'un élément d'un ensemble de départ

Pour chacun d'entre eux, j'ai défini une séquence d'actions. Dans un stade ultérieur, ces séquences devront être créées, puis modifiées, automatiquement.

Un élément d'une séquence d'actions peut être :

1. Une condition. Si elle est vraie, on exécutera la sous-séquence d'actions qui lui est liée. Il existe deux types de conditions : dans l'un on s'arrête à la fin de la sous-séquence ; dans l'autre on revient à la séquence principale. Cela permet par exemple d'indiquer un traitement spécifique pour les contraintes qui ne contiennent qu'une seule variable.
2. L'ordre d'envisager une règle. Elle est mise en attente avec la priorité calculée par la formule attachée à l'élément.
3. L'ordre d'envisager d'envisager une règle. On met cette fois en attente qu'il faudra envisager une règle ; deux définitions de priorité sont attachées à cet élément, l'une pour la priorité avec laquelle on envisagera d'appliquer la règle et l'autre celle avec laquelle on appliquera la règle si l'on s'est décidé à l'envisager.

La catégorie « envisager d'envisager » peut paraître superflue. Elle a été introduite pour des raisons d'efficacité, car envisager d'appliquer une règle peut se révéler fort coûteux en temps et peut charger de façon inutile les listes d'attente de règles à appliquer. Par exemple, si on a une contrainte de la forme :

$$a_1+a_2+.....+a_i+...+a_n = b_1+b_2+.....+b_j+.....+b_p$$

on peut être amené à considérer d'appliquer n+p fois une règle qui créera de nouvelles contraintes du type :

$$b_j <= SUP(a_1+a_2+.....+a_i+...+a_n) - INF(b_1+b_2+.....+b_{j-1}+b_{j+1}+.....+b_p)$$

Dans le cas où il y a des actions plus prometteuses à faire que d'appliquer un aussi grand nombre de fois une règle, on préfère garder en attente le fait que l'on devra l'envisager peut-être ultérieurement.

Reprenons la règle R6 que nous avons vu plus haut. Elle apparaît actuellement dans les séquences d'actions liées à trois événements :

1. On l'envisage quand on crée une nouvelle contrainte R.
2. On envisage de l'envisager quand on diminue la valeur maximum possible d'un nœud DX qui figure dans la partie X d'une contrainte R.
3. On envisage de l'envisager quand on augmente la valeur minimum possible d'un nœud DY qui figure dans la partie Y d'une contrainte R.

Dans le premier cas, la priorité de l'appliquer est donnée par la formule :

LOG CARVAR(R) [2 : @ELEVE] [15 : @TRESBAS]

Cela signifie que la valeur de la priorité sera définit par une courbe logarithmique qui prend la valeur @ELEVE si le nombre de variables figurant dans R (défini par la fonction CARVAR) est égal à 2 et @TRESBAS si ce nombre est égal à 15. Avec LOG, la valeur est non définie si elle est en dehors de

l'intervalle |2,15]. En effet, moins il y a de variables, plus la règle R6 a de chances d'être satisfaite et moins de temps elle demande pour être évaluée. Au delà d'un certain nombre de variables, il y a une chance infime que ses conditions soient satisfaites.

Dans le deuxième cas, on a pour priorité d'envisager de l'envisager :

ULOG DMAX(DX) [1 : @TRESBAS] [14 : @TRESELEVE]

La valeur de la priorité dépend de la différence DMAX entre l'ancienne et la nouvelle valeur maximum possible (qui vient d'être modifiée) du nœud DX : plus cette différence est grande et plus on a de chances que la règle R6 s'applique puisque DX intervient dans une fonction SUP. À ce stade, on n'a pas encore défini la contrainte R, d'où un gain de temps appréciable si cette priorité est faible et qu'il aurait fallu envisager d'appliquer R6 à un très grand nombre de contraintes. ULOG indique toujours une progression logarithmique entre les extrêmes, mais si la valeur est supérieure à 14, ce qui est encore plus prometteur, on l'envisagera, tout en prenant la valeur qui était associée à 14.

Toujours dans le deuxième cas, une fois que l'on a décidé de l'envisager et que l'on regarde l'intérêt de son application à une contrainte R, on a la priorité :

PROG (LOG CARVAR(R) [2 : @ELEVE] [15 : @TRESBAS] ,
ULOG DMAX(DX) [1 : 0] [14 : 3])

La fonction PROG permet de cumuler deux aspects : son premier argument a pour valeur une priorité P et son deuxième argument un entier qui va augmenter, s'il est positif, cette priorité P et la diminuer s'il est négatif. Cette augmentation est d'autant plus grande que l'entier est grand : PROG(@TRESBAS,2) vaut @ASSEZBAS et PROG(@TRESELEVE,-1) vaut @ELEVE. Dans le cas présent, on tient compte à la fois du fait qu'il vaut mieux avoir des contraintes avec peu de variables et qu'il vaut mieux que la diminution de la valeur maximum possible de DX soit aussi grande que possible.

Les priorités sont définies par des courbes comportant peu de paramètres de façon à ce que le système puisse les créer, les modifier et les examiner facilement.

3.3. Le backtrack

Il n'est pas toujours possible de résoudre un problème directement. Aussi est-il en général nécessaire de faire des choix, par exemple on considère successivement toutes les valeurs possibles d'une variable. On crée ainsi une arborescence. Quand on rencontre une contradiction, on ne conclut plus à une impossibilité de résoudre le problème, mais on remonte au dernier choix et on examine la possibilité suivante s'il en reste. Le système doit avoir des connaissances pour choisir le choix qu'il va faire : toutes les valeurs d'une certaine variable ou toutes les contraintes issues d'une certaine contrainte disjonctive.

MALICE dispose d'un ensemble de connaissances pour faire ce choix. Il applique ces connaissances à tous les choix possibles et cela associe finalement une valeur à chaque choix. Il prend le choix possible qui a la valeur la plus élevée. Il existe une grand nombre de connaissances indépendantes les unes des autres. Elles indiquent d'abord le type de choix : nœud de départ, nœud d'arrivée, contrainte disjonctive. Nous avons ensuite le degré d'intérêt qui est lié à cette connaissance avec son importance. Par exemple :

DEPART =>ULOG CARD(CONTRAINTES(ND)) [1 : @BAS] [20 : @TRESELEVE]

IMPORTANCE : @ASSEZBAS

Cela signifie que pour chaque nœud de départ, appelé par convention ND, on ajoutera à l'ensemble des couples un couple dont l'intérêt est d'autant plus élevé qu'est élevé le cardinal de l'ensemble des contraintes où ce nœud apparaît. L'importance de chacun de ces éléments sera assez basse. On favorise la considération de nœuds qui apparaissent dans beaucoup de contraintes, sans tenir compte ici du type de contrainte.

La connaissance peut éventuellement contenir des conditions qui limitent son application ou qui définissent des variables. Si une variable ainsi définie balaye un ensemble, la connaissance pourra donner naissance à plusieurs couples (intérêt, importance) :

DEPART S∈ CONTRAINTES(ND), COPRI(S)=@EQ =>

LOG CARVAR(S) [2 : @TRESELEVE] [15 : @TRESBAS]

IMPORTANCE : @ASSEZELEVE

Avec la connaissance précédente, on ajoute autant de couples (intérêt, importance) qu'il y a de contraintes contenant la variable ND qui représente le nœud et dont la connective principale est l'égalité. Moins une de ces contraintes contient de variables et plus l'intérêt correspondant sera grand.

À partir de cet ensemble de couples (intérêt, importance), une méthode générale, inspirée de [Pinson 1987] détermine l'élément dont on va examiner successivement toutes les valeurs possibles. L'avantage de donner déclarativement les connaissances de choix de l'élément sur lequel on va backtracker est que MALICE pourra lui-même les créer ou les modifier. Comme pour les séquences d'actions, un ensemble de telles connaissances est actuellement donné au système au départ.

3.4. La création de programmes combinatoires

Une des méthodes de résolution de MALICE est la création d'un programme combinatoire suivie de son exécution. [Lucas 1989] et [Laurière 1996] ont déjà bien montré l'intérêt de cette approche, à condition d'engendrer un programme efficace. Un tel système doit être méta-intelligent, c'est à dire être assez intelligent pour se rendre compte que cela ne sert parfois à rien d'être intelligent. Il est inutile de réduire la combinatoire par des méthodes subtiles si le gain espéré par cette démarche est trop faible : on a plus vite fait d'examiner toutes les possibilités, à condition d'avoir pris la peine d'éliminer auparavant des essais manifestement inutiles et de définir un bon ordre pour faire des choix. La création d'un programme combinatoire se fait toujours après une phase de fonctionnement normal de MALICE qui a créé de nouvelles contraintes et éliminé des valeurs possibles pour certaines variables : cela diminue souvent de façon sensible la combinatoire et permet au système de mieux ordonner, dans le programme créé, les variables dont on examine successivement toutes les valeurs possibles.

Une méta-expertise de MALICE est capable d'engendrer un programme efficace. Elle est pour le moment exprimée dans le formalisme de MACISTE. Comme ce formalisme n'est pas assez déclaratif, il va falloir donner ces métaconnaissances sous une forme plus déclarative de façon à ce que le système puisse les modifier quand il se rend compter que les programmes engendrés ne sont pas suffisamment performants.

3.5. La gestion des (méta)connaissances

L'utilisation des séquences d'actions et des règles est très simple. Quand un événement se produit (ceci est initialisé par l'événement : début de résolution de problème), on exécute la séquence d'actions correspondante, ce qui donne des essais à faire avec, pour chacun, leur priorité : les actions sont soit des règles à appliquer avec leurs arguments (ce qui est fait en exécutant le programme C correspondant), soit des actions à envisager (parce qu'il y avait une action demandant d'envisager d'envisager). Les essais à faire sont donc stockés et l'on exécute chaque fois l'essai avec la plus haute priorité. S'il ne donne lieu à aucun événement, on prend le suivant. S'il se produit des événements, on exécute les séquences d'actions correspondantes, on ajoute éventuellement de nouveaux essais à faire et on prend l'essai qui a maintenant la plus haute priorité.

Auparavant, le système doit faire un travail important au niveau méta pour écrire les programmes qui correspondent aux règles et aussi ceux qui correspondent aux connaissances permettant de choisir la variable dont on considérera successivement les valeurs. Cette traduction est assez facile, parce que l'on passe par une étape partiellement déclarative, qui est le formalisme de MACISTE. Celui-ci a une méta-expertise performante pour traduire des ensembles d'actions conditionnelles en programmes C. Il peut être également amené à écrire, compiler et exécuter un programme combinatoire qui résout le problème.

4. Expérimenter et analyser

MALICE devra devenir complètement autonome : il résoudra des problèmes tout en monitorant la recherche de la solution, analysera des résultats, en tirera des lois, décidera de faire des expériences, créera et modifiera ses méthodes de résolution. Pour le moment, j'ai développé essentiellement l'analyse des résultats et la détermination de nouvelles expériences. Nous allons commencer par voir que cette approche entraîne plusieurs contraintes pour le système.

Deux éléments interviennent constamment dans ces recherches : les paramètres qui régissent le comportement du système et les grandeurs qui caractérisent la qualité d'une solution. Un certain nombre de paramètres ont une grande influence sur les résultats d'un système, et une tâche importante pour le

chercheur est de définir un bon ensemble de leurs valeurs ; MALICE doit faire lui aussi cet ajustement. Ces paramètres sont dans le cas présent :

1. Le niveau minimum de priorité accepté. Par exemple, on ne tente que les essais qui ont une priorité supérieure ou égale à @ASSEZBAS. Cela signifie que tout essai qui a la priorité @BAS ou @TRESBAS est éliminé. On augmente la métacombinatoire, c'est à dire la combinatoire sur les méthodes envisagées, quand on diminue la priorité minimum acceptée. On a des solutions plus élégantes, mais il faut souvent plus de temps pour les trouver. En contrepartie, on diminue en général la combinatoire, les arborescences engendrées sont plus petites.

2. L'application de certaines règles peut être inhibée, par exemple même si une séquence d'actions dit de considérer la règle R6, cet essai sera éliminé.

3. Certaines priorités dans les séquences d'actions sont changées, par exemple on donnera à un essai la priorité @MOYEN alors qu'auparavant on ne lui donnait que la priorité @ASSEZBAS parce que l'on a décidé d'être plus généreux pour les essais qui font intervenir une contrainte qui contient beaucoup de variables.

4. On peut changer les importances des différents facteurs qui déterminent la variable ou la contrainte disjonctive que l'on choisit pour backtracker. Par exemple, on considérera que le nombre de contraintes où elle figure est un facteur d'importance @ELEVE au lieu de @ASSEZELEVE, ce qui va conduire à préférer davantage les variables qui apparaissent dans beaucoup de contraintes.

À côté des paramètres, des grandeurs permettent d'évaluer la qualité d'une solution. Ce sont dans tous les cas : le temps CPU nécessaire pour l'obtenir ; dans le cas où l'on utilise les méthodes d'interprétation de type ALICE : le nombre de feuilles et le nombre de nœuds de l'arborescence, le nombre de contraintes créées, le nombre d'étapes ; dans le cas où le système crée un programme : le nombre de nœuds considérés. De plus, si l'on n'a pas eu le temps d'achever la résolution, on évalue une grandeur imprécise, le taux d'achèvement qui est le rapport entre la taille de l'arborescence effectivement développée et l'estimation de sa taille totale. Grâce à ces grandeurs, MALICE peut comparer les avantages et les inconvénients de divers jeux de paramètres.

4.1. Contraintes sur le système

De façon générale, MALICE doit avoir beaucoup des propriétés d'un système d'exploitation, en particulier être stable et, en cas de difficulté, garder les informations sur ce qui s'est passé et pourvoir repartir. Le système ne doit jamais s'arrêter, quoiqu'il arrive. Il attrape donc toutes les erreurs conduisant à une interruption, mémorise l'anomalie et repart pour continuer ses expériences en tenant compte de ces incidents. De même, s'il n'y a plus de mémoire disponible, il doit faire le ménage pour pouvoir poursuivre.

MALICE doit contrôler le temps des différentes tâches qu'il lance. Quand il fonctionne en mode interprétatif, c'est facile puisqu'il garde le contrôle. Par contre, quand il exécute un programme qu'il vient d'écrire et de compiler, il ne faut pas qu'il l'arrête brutalement en cas de dépassement du temps qui est alloué à ce programme. Il inclut dans les programmes qu'il crée des points où il passera certainement régulièrement et où il trouvera éventuellement l'ordre de s'arrêter. Dans ce cas, avant de s'arrêter il transmet à MALICE toutes les informations indiquant où il en est dans sa recherche. Cela permet d'évaluer les difficultés rencontrées par ce programme et de les corriger éventuellement.

MALICE mémorise les expériences qu'il a faites et leurs résultats : il tient un journal qu'il peut consulter, ce qui lui évite de refaire des expériences qui ont déjà été faites, de comparer en quoi une modification de stratégie change les performances, et surtout de métamonitorer sa recherche. Le monitoring se fait en cours de résolution de problème et il permet de voir par exemple que l'on est en train de tourner quasiment en rond, car l'on fait de nombreux essais analogues sans progresser. De même, le métamonitoring se place au niveau supérieur et voit que, dans la progression vers une meilleure capacité à résoudre des problèmes, on fait toujours le même type d'expériences ou que l'on oublie d'expérimenter avec certaines familles de problèmes. Pour se rendre compte des faiblesses du métamonitoring, un journal des essais faits est bien indispensable.

4.2. Analyser les résultats

MALICE analyse les résultats qu'il obtient en résolvant des problèmes pour en déduire quelles nouvelles expériences faire afin d'éclaircir des points douteux et de détecter des surprises, c'est à dire des résultats en contradiction avec ce qui était prévu. Il s'en servira ultérieurement pour trouver des lois sur les conséquences de la modification de paramètres et pour modifier les séquences d'actions ou en créer de nouvelles.

L'analyse des résultats porte sur quatre éléments : la trace créée en cours de résolution, l'arborescence des essais combinatoires, les prédictions et les conséquences des variations des paramètres. Les deux premières analyses peuvent être faites sur une seule résolution, les autres sont étroitement liées à l'expérimentation et demandent plusieurs résolutions du même problème avec des paramètres différents.

4.2.1. Analyse de la trace

En examinant la trace, MALICE découvre un grand nombre de faits intéressants. Le plus important est l'étude de l'utilisation des règles. Pour cela, le système examine tous les faits qu'il a établis et qui sont manifestement nécessaires pour la solution, comme l'affectation d'une valeur à une variable ou la découverte d'une contradiction. Il retient aussi des événements dont il n'est pas sûr qu'ils ont servi à établir la solution, et en particulier l'élimination d'un lien possible entre une variable et sa valeur. En effet, il est difficile de voir toutes les conséquences de cette élimination, qui peut intervenir de façon cachée dans la détermination de la borne supérieure d'une expression algébrique. À partir du moment où l'on connaît tous ces essais utiles, on ajoute tous les essais qui ont participé à la préparation d'un d'entre eux, et ceci de façon récursive. On arrive ainsi à un sous-ensemble des essais faits tel que l'on a certainement la même solution en ne faisant que ces essais. Il est possible que certains de ces essais soient quand même superflus, mais on ne fait jamais l'erreur inverse, on n'omet jamais d'essai indispensable. La méthode est efficace, car le nombre des essais ainsi gardés ne représente qu'une faible fraction de tous les essais faits. On engendre ainsi une explication de la solution.

MALICE se rend compte alors par exemple que certaines règles ont été souvent utilisées et ont rarement servi à l'avancement de la solution ; cela peut venir de ce que la règle ne produit rien parce que ses conditions ne sont pas remplies ou de ce que la règle s'est appliquée, mais a conduit à une nouvelle contrainte qui n'a absolument pas servi à trouver la solution. Il faudra alors prévoir de limiter plus sévèrement l'utilisation de telles règles. Inversement, il voit que des règles qui ont une très basse priorité ont pourtant été essentielles dans la découverte de la solution, ce qui est anormal et qui devrait conduire à une modification de la priorité de ces règles.

Le système peut aussi remarquer une longue période pendant laquelle aucun essai faisant progresser le système vers la solution n'a été fait. Une telle remarque montre que le monitoring est sans doute mal fait. Cela se produit souvent quand MALICE engendre une grande quantité de contraintes qui ne conduisent à rien.

Le résultats de l'utilisation des règles sont récapitulés avec chaque règle. On peut ainsi savoir si des règles sont souvent ou rarement utilisées et également si, quand elles sont utilisées, c'est avec ou sans succès. On peut aussi voir qu'une certaine règle est très utile pour certains problèmes et totalement inutiles pour d'autres. Ce sera utile pour construire des séquences d'actions adaptées à des familles de problèmes.

4.2.2. Analyse de l'arborescence

Dans certains cas, MALICE trouve directement la solution du problème et montre son unicité (ou son impossibilité) sans faire aucun backtrack. Mais en général il est amené à développer une arborescence de choix. Ces choix peuvent être d'affecter une valeur à une variable, de dichotomiser les valeurs possibles d'une variable(cas X<=5 et cas X>5) ou de considérer autant de branches qu'il y a de termes dans une contrainte disjonctive. MALICE découvre souvent des anomalies dans cette arborescence.

Une première anomalie se produit quand, une certaine variable a toujours la même valeur dans toutes les sous-arborescences qui suivent un choix. Par exemple, que X vaille 1,2,3 4,ou 5, la variable Y a ensuite toujours la valeur 8. Il est naturel alors de se demander s'il n'aurait pas été possible de prouver que Y vaut 8 avant de considérer toutes les valeurs de X.

Une deuxième anomalie se produit quand on trouve une contradiction immédiatement après avoir fait un choix. Par exemple, on affecte la valeur 2 à X et on trouve tout de suite que cela donne une contradiction.

On peut se demander s'il n'aurait pas été possible de trouver des règles pour éliminer cette valeur 2 pour X avant de lui affecter cette valeur ; cela limiterait la taille de l'arborescence engendrée.

Les anomalies détectées par examen de l'arborescence incitent à faire des expériences en reprenant ces problèmes, mais en permettant d'essayer beaucoup plus de règles juste avant les choix litigieux qui ont été faits. Avec ces règles, on pourra dans certains cas diminuer l'éventail des choix, ou même on les rendra tous inutiles ; si l'on arrive à faire mieux, il y a matière à apprendre afin de pouvoir définir par la suite les essais utiles qui avaient été omis.

4.2.3. Les prédictions

Dans une première résolution d'un problème, on trouve des faiblesses que l'on pense corriger en refaisant un nouvel essai après avoir changé la valeur de certains paramètres. Par exemple, on a observé qu'aucun essai fait à la priorité la plus basse n'avait été utile et on recommence en interdisant tout essai à cette très basse priorité. On prédit que la taille de l'arborescence solution sera la même, mais que cela prendra moins de temps et que l'on fera moins d'essais. Si la prédiction s'avère fausse, il y a une anomalie, qui est une mauvaise surprise ; MALICE essayera de l'interpréter. Si le résultat n'est pas très éloigné de ce qui était prévu, cela peut être normal, par exemple le temps n'est pas mesuré avec une très grande précision. C'est peut-être aussi dû à une faiblesse de sa théorie actuelle, par exemple il croit qu'en éliminant des essais inutiles, on fait moins d'essais alors que ce n'est pas toujours exact : les essais inutiles peuvent conduire à créer des contraintes inutiles, mais dont la présence va inciter le système à faire d'autres choix dans l'arborescence. Par exemple, ces contraintes superflues vont l'inciter à choisir d'autres variables pour backtracker ; s'il se trouve que, par hasard, ce choix était meilleur que celui fait dans la deuxième exécution, il est normal que les résultats soient moins bons. Il faudra alors qu'il fasse des expériences pour vérifier le bien-fondé de cette explication de l'anomalie. Cela peut être enfin inexplicable pour MALICE, par exemple si cela est dû à un bogue du système lui-même ; il faut alors qu'il en informe l'auteur du système.

4.2.4. Comparer plusieurs résolutions d'un même problème

Il est difficile de déterminer la valeur de l'ensemble des paramètres qui régissent le comportement d'un système. Cela fait une partie importante du travail du chercheur en IA, travail qu'il ne fait pas toujours bien faute de temps. Nous pouvons donner au système des méthodes pour faire ces modifications, mais il est plus intéressant que le système découvre lui-même les lois qui lient une modification de la valeur d'un paramètre et ses conséquences sur la résolution d'un problème.

Les modifications que l'on peut apporter sont souvent en séquence, par exemple on a sept valeurs possibles pour une priorité. On peut donc comparer ce qui se passe quand on interdit tout essai dont la priorité est inférieure à l'une de ces valeurs. Nous avons vu qu'un certain nombre de caractéristiques sont observables sur une solution : degré d'achèvement, nombre de solutions, nombre de feuilles de l'arborescence, nombre de nœuds de l'arborescence, nombre total de contraintes engendrées, nombre d'étapes, temps CPU. On peut faire varier un paramètre et comparer ce qui en résulte en gardant les autres paramètres constants. Par exemple, on peut ne considérer que des solutions avec le degré d'achèvement total et comparer l'influence du niveau de priorité accepté sur le nombre d'échecs, le nombre de solutions, le temps CPU, etc. Le système découvre pour chaque problème un lien entre les éléments en succession, par exemple avec le problème P14, le nombre d'échecs croit fortement avec la priorité minimale admise (on admet moins d'essais quand elle croît) alors que le temps d'exécution décroît. De même, pour le problème P22, à temps d'exécution constant, le degré d'achèvement d'un problème diminue quand la priorité minimale augmente, alors que la taille de l'arborescence augmente.

À partir de ces résultats par problèmes, MALICE devra établir des lois générales. Par exemple, si les problèmes sont totalement résolus, le nombre de solutions est toujours constant quand la priorité minimale croit, alors qu'en général le nombre d'échecs ne décroît pas. Ceci est doublement utile : en cas d'insuccès, on peut savoir comment agir pour y remédier, mais aussi, on peut avoir des surprises quand on est dans un cas qui ne correspond pas aux lois établies. Donnons un exemple concret de ce dernier point qui n'est pas évident.

Pour un certain problème, P34, MALICE découvre qu'en augmentant la priorité admise pour les essais et en gardant le temps machine constant, on a une anomalie dans la croissance : au bout de 10 minutes, si la priorité est TRESBAS, l'achèvement est de 1% ; il est autour de 1 à 2% pour toutes les autres valeurs de la priorité minimale admise, sauf pour la valeur intermédiaire @ASSEZELEVE où il est de 31%. MALICE vérifie naturellement qu'il obtient la solution en repartant avec cette valeur et en l'autorisant à dépenser quatre fois plus de temps. Mais on peut aussi remarquer que ce progrès anormal est sans doute dû à un

meilleur choix de l'ordre des variables que l'on considère pour mettre dans l'arborescence. On peut donc refaire les essais avec une autre priorité et ce même choix de variables et voir si l'on n'aura pas des solutions avec une arborescence encore plus petite (les priorités basses conduisent en général à appliquer plus de règles, mais à avoir des arborescences de taille plus réduite). MALICE a ainsi matière à réflexion pour améliorer les connaissances qui choisissent la variable dont on va considérer toutes les valeurs possibles : ce choix était sans doute mal fait avec les autres valeurs des priorités minimales et c'est par pure chance qu'il a été bien fait quand on a refusé tout essai qui n'avait pas au moins la priorité @ASSEZELEVE. Il faut savoir profiter d'un coup de chance.

4.3. Faire des expérimentations

Il est indispensable qu'un système puisse faire des expérimentations ; SEPIAR [Parchemal 1988] en avait d'ailleurs déjà montré et la possibilité et l'intérêt. Cela est utile pour trois raisons : cela permet de vérifier une hypothèse, d'établir des lois sur les conséquences de la modification d'un paramètre et enfin d'explorer l'espace des paramètres pour découvrir des faits imprévus. En effet, si on ne fait que ce qui est conforme à ce que l'on sait, on ne risque pas de progresse en découvrant des faits imprévus qui peuvent modifier nos croyances.

Dès à présent, MALICE est capable de faire un certain nombre d'expériences. La façon normale de traiter un problème est d'interpréter les connaissances en suivant les indications des séquences d'actions et en envisageant tous les essais, y compris, ceux de la priorité la plus basse, @TRESBAS. Voici les expériences qu'il fait actuellement :

1. Si le problème a été résolu avec une arborescence comportant des choix, on résout aussi le problème en engendrant un programme combinatoire que l'on exécute ensuite. Normalement, ces programmes sont très efficaces, car MALICE les engendre après avoir déjà modifié l'énoncé du système, en particulier en ajoutant des contraintes ; de plus, il choisit avec beaucoup de soin l'ordre des choix. En contrepartie, sa solution combinatoire est difficilement compréhensible et explicable. Si le programme combinatoire met plus de temps à résoudre un problème que s'il le résout en utilisant uniquement les règles, le système signale une anomalie.

2. Pour établir les lois sur les liens entre les paramètres et les caractéristiques d'une solution, on essaye systématiquement les problèmes avec des priorités minimales acceptées de plus en plus élevées. On a ainsi pour chaque problème des séries de résultats que l'on analyse ensuite.

3. Si l'explication de la solution d'un problème ne contient pas de règles au-dessous d'une priorité P, on essaye de résoudre le problème en interdisant d'utiliser des règles au-dessous de cette priorité. Normalement, on devrait avoir la même solution en moins de temps et en engendrant moins de contraintes. Il se peut que cela ne se produise pas, parce que, avec la première solution, on a engendré par exemple des contraintes qui n'ont servi à rien, mais qui ont influé sur les choix de variables dont on examine toutes les valeurs. Un changement de l'ordre des variables pour lesquelles on fait des choix peut avoir des conséquences importantes sur les performances. Aussi, si cela se produit, on indique l'existence d'une anomalie, qui conduit à réétudier le choix de la variable dont on va examiner successivement les valeurs possibles.

4. Si une règle a été rarement ou jamais utile, on résout à nouveau le problème en inhibant cette règle. On compare les résultats des deux résolutions et on peut ainsi s'apercevoir qu'il y a une redondance cachée, d'autres règles remplaçant la règle ainsi inhibée (dans le cas "rarement") ou une influence indirecte des contraintes qui n'ont pas servi mais qui ont joué un rôle dans le choix des variables de backtrack.

5. Dans un but exploratoire, on peut résoudre des problèmes en inhibant une règle sans avoir de raison précise pour cela, simplement par curiosité.

6. On peut élargir le champ d'application de certaines règles, par exemple en modifiant leurs priorités dans les séquences d'actions. Cela a deux utilisations : si l'on soupçonne que certaines règles sont appliquées trop souvent ou pas assez, on change leurs priorités en conséquences. Si cette modification s'avère fructueuse, on pourra changer les séquences d'actions de façon définitive. Par ailleurs, en changeant systématiquement les priorités et en comparant l'évolution des résultats pour cette succession d'essais, on peut établir des lois sur l'influence d'un changement sur les performances.

MALICE est dès à présent capable de faire des expériences. Il les utilise pour trouver des lois et détecter des anomalies. Mais il n'est pas encore capable de tirer pleinement partie des anomalies qu'il a observées.

5. Développements futurs

Les développements futurs de MALICE doivent porter sur de nombreuses directions : ajouter de nouvelles règles, créer et modifier les séquences d'actions, modifier la représentation des problèmes, définir une théorie sur les modifications des paramètres, définir dans un formalisme analogue à celui de MALICE toutes les métaconnaissances qui permettent de définir une stratégie de résolution de problèmes, introduire du monitoring et du métamonitoring et enfin lui donner l'équivalent de la conscience morale pour le rendre totalement autonome.

5.1. Ajouter de nouvelles règles

MALICE dispose actuellement de 70 règles, mais ce nombre est insuffisant : un expert humain dispose d'un éventail bien plus large de méthodes. La qualité de certaines solutions est toutefois surprenante, pour certains problèmes elles sont bien meilleures que celles obtenues par des humains entraînés : je ne connais pas beaucoup de personnes capable de trouver la solution de DONALD+GERALD=ROBERT et d'en montrer l'unicité en faisant seulement deux choix binaires. Comme cela a déjà été établi avec ALICE [Laurière 1979], le fait de disposer d'un panier de méthodes, dont l'idée est venue de problèmes variés, est très fructueuse : le système pense à se servir de certaines d'entre elles dans des situations où un humain n'y pense pas. Du fait de sa rapidité, l'ordinateur permet une métacombinatoire bien plus importante que celle d'un chercheur humain et c'est cela qui conduit à des solutions plus élégantes. Il faudrait ajouter de nouvelles règles pour étendre le champ d'application du système à d'autres domaines, et en particulier à diverses théories mathématiques. À très long terme, il serait évidemment souhaitable que MALICE soit capable de créer lui-même ces nouvelles règles.

5.2. Créer et modifier les séquences d'actions

Il existe actuellement une seule famille de séquences d'actions, la même pour tous les problèmes et je l'ai donnée au système. Un expert n'a pas le même comportement selon le type de problèmes qu'il traite et il faudrait que le système établisse une certaine variété de séquences d'actions.

Il faudrait avoir des métarègles capables de créer une séquence d'actions. De telles métarègles sauraient que si un problème a telles caractéristiques, alors telle règle est ou n'est pas nécessaire ; elles devraient de plus déterminer les priorités correspondantes. Le résultat des expériences devrait ensuite permettre d'affiner ces séquences initiales forcément imparfaites, et peut être aussi les métarègles qui créent les séquences.

5.3. Modifier la représentation des problèmes

Modifier la représentation d'un problème est difficile pour les humains, il ne faut donc pas s'attendre à ce que MALICE ait des possibilités très importantes dans ce domaine à court terme. Toutefois, il serait possible d'introduire une recherche des symétries dans l'énoncé des problèmes ; en cas de découverte de symétrie, le système ajouterait des contraintes pour n'engendrer qu'une seule solution. L'existence de symétries peut augmenter considérablement le nombre de solutions pour des problèmes où il existe plusieurs symétries indépendantes : le nombre total de solution équivalentes est alors le produit du nombre d'éléments équivalents pour chacune de ces symétries. Un exemple de symétrie facile à trouver et assez fréquent est celui où deux variables apparaissent dans une seule contrainte et, dans cette contrainte, à l'intérieur d'une même somme : SOMME[A, B, , ,] ; bien évidemment, on peut alors toujours permuter les valeurs attribuées à A et à B. Pour éviter d'avoir deux solutions, il suffit d'ajouter la contrainte $A \leq B$. S'il y a dix contraintes de ce type, il y aurait sans cela plus de mille solutions équivalentes. Il y a bien longtemps, [Gelernter 1959] avait fait une remarquable implémentation de l'utilisation des symétries.

5.4. Définir une théorie de la modification des paramètres

Pour choisir comment modifier la valeur de paramètres en cas d'échec ou d'inefficacité, on a besoin de disposer d'une théorie des conséquences de leur modification sur les résultats obtenus : temps nécessaire, nombre de feuilles de l'arborescence, nombre de nœuds examinés. On établit par exemple qu'en général si l'on accepte des essais ayant une priorité plus faible, on aura moins de feuilles, moins de nœuds dans l'arborescence, moins de temps CPU, mais plus de contraintes seront engendrées. Par contre, si l'on inhibe une règle, on aura en général plus de feuilles, plus de nœuds, mais moins de contraintes.

Ces lois sont utiles pour savoir comment arriver à avoir une solution : si par exemple on est saturé devant un trop grand nombre de contraintes, on peut augmenter le seuil admis pour une priorité. Mais ce peut aussi

être utile pour pouvoir être surpris : la surprise est essentielle, car elle indique qu'il y a quelque chose à comprendre. Une fois que l'on a compris, on aura sans doute quelque chose à apprendre.

On doit pour établir ces lois faire beaucoup d'expériences, donc avoir une théorie de l'expérimentation des paramètres : quelle expériences faire pour établir une loi sur un certain paramètre ? Par exemple, on peut faire un grand nombre d'essais avec des priorités variées et comparer les résultats.

5.5. Définir les métarègles dans le même formalisme que les règles

À côté des règles qui permettent de faire progresser la solution, en créant de nouvelles contraintes ou en découvrant des contradictions, nous devrons avoir des métarègles qui décident des expériences à faire, établissent des lois sur les conséquences de la modification des paramètres, créent de nouvelles règles. Actuellement, certaines de ces métaconnaissances existent, mais elles sont exprimées dans le formalisme de MACISTE qui n'est que partiellement déclaratif. Aussi serait-il souhaitable de les exprimer dans le même formalisme que les règles, en les accompagnant de séquences de méta-actions, ce qui aurait l'avantage d'être plus facilement compréhensible et d'aider le chercheur à donner ces nouvelles métaconnaissances. À très long terme, si le système est devenu réflexif, il pourrait créer des métarègles de ce type à l'aide d'elles-mêmes.

5.6. Introduire plus de monitoring et de métamonitoring

Il faudrait donner à MALICE plus de possibilités de monitoring. En cours de traitement, il se trouve parfois confronté à une floraison de nouvelles contraintes qui seraient intéressantes si elles étaient rares, mais sont sans intérêt par suite de cette concurrence. On ne peut pas raisonner de façon absolue, mais en fonction de l'historique. Mais il faut aussi éviter les mêmes écueils au niveau méta : on peut être amené à faire trop d'expériences proches les unes des autres, expériences qui auraient été intéressantes si elles n'avaient pas été précédées des autres expériences. Avec le métamonitoring, on a l'œil sur les expériences passées pour avoir une vue globale de ce que l'on fait et éviter de s'enliser dans une zone trop restreinte de l'espace des expériences possibles.

5.7. Définir une "conscience morale"

MALICE doit fonctionner de façon de plus en plus autonome, son auteur devant intervenir de moins en moins au fur et à mesure de son élaboration. Pour éviter les dérives, il faut qu'il ait un module robuste qui surveille le tout, que l'on pourrait assimiler à une sorte de "conscience morale". Ce module devrait suivre des principes généraux qui retardent ou interdisent des opérations trop dangereuses comme la suppression définitive d'éléments dont la création et l'adaptation a demandé beaucoup de travail et qui se sont révélées pendant longtemps comme très performants.

6. Conclusion

La construction de MALICE est évidemment une tâche de très longue haleine. Ce doit être un résolveur de problèmes général plus performant que les meilleurs résolveurs spécifiques et aussi que les résolveurs généraux que sont les êtres humains. Il est difficile de savoir dans quelles directions exactement le système va se développer, les idées viennent en expérimentant. Mais je suis persuadé qu'une telle voie de recherche est extrêmement riche, car nous sommes obligés en la prenant d'étudier des problèmes fondamentaux comme l'apprentissage basé sur la compréhension, la définition d'expériences à faire et la construction de systèmes créatifs autonomes.

7. Références

[Gelernter 1959] Gelernter H., *A note on syntactic symmetry and the manipulation formal systems by machine*, Information and Control 2, 1959, 80-89.

[Junghanns 2001] Junghanns A and Schaeffer J., *Sokoban : Enhancing general single-agent serach methods using domain knowledge*, Artificial Intelligence 129 (2001) 219-251.

[Laurière 1976] Laurière J.-L., *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires*, Thèse de l'Université Paris 6, 1976.

[Laurière 1979] Laurière J.-L., *Toward efficiency through generality*, Proceedings of the sixth IJCAI, 1979, 619-621.

[Laurière 1996] Laurière J.-L., *Propagation de contraintes ou programmation automatique ?*, Rapport LAFORIA 96/19, 1996.

[Lucas 1989] Lucas J.-Y., *Génération automatique de programmes par règles et compilation de bases de règles. Application à un système expert de diagnostic de signaux courants de Foucault*, thèse de l'Université Paris 6, 1989.

[Newell 1960] Newell A., Shaw J. and Simon H., *A variety of intelligent learning in a General Problem Solver*, Yovits and Cameron eds, Self-Organizing systems, Pergamon Press, 1960, 153-189.

[Parchemal 1988] Parchemal Y., SEPIAR *: Un système à base de connaissances qui apprend à utiliser efficacement une expertise*, Thèse de l'Université Paris 6, 1988.

[Pinson 1987] Pinson S., *Méta-modèle et heuristique de jugement : le système CREDEX. Application à l'évaluation du risque crédit entreprise*, Thèse de l'Université Paris 6, 1987.

[Pitrat 1966] Pitrat J., *Réalisation de programmes de démonstration de théorèmes utilisant des méthodes heuristiques*, Thèse de l'Université de Paris, 1966.

[Pitrat 1976] Pitrat J., *Ce titre contient quatre 'a', un 'b', cinq 'c', cinq 'd', dix-neuf 'e', deux 'f', un 'g', deux 'h', treize 'i', un 'j', un 'k', un 'l', un 'm', seize 'n', trois 'o', quatre 'p', sept 'q', sept 'r', sept 's', quinze 't', dix-huit 'u', un 'v', un 'w', six 'x', un 'y' et quatre 'z'*, Rapport Laforia 96/26, septembre 1996.

[Schuurmans 2001] Schuurmans D. and Southey F.,*Local search characteristics of incomplete SAT procedures*, Artificial Intelligence 132 (2001) 121-150.

UN REGARD EXTERIEUR SUR LE SYSTEME *MACISTE* DE JACQUES PITRAT

Basile STARYNKEVITCH [1]

basile@starynkevitch.net

http://starynkevitch.net/Basile/

Résumé: Ce papier présente quelques observations sur le système réflexif *Maciste* de Jacques Pitrat. *Maciste* est un système reflexif à base de connaissances, exprimées dans des règles regroupées en expertises. Je présente ici quelques caractèristiques observables du programme *Maciste* et de son code. Des exemples ou extraits d'expertises sont donnés. Leur compilation est esquissée. Après une présentation incomplète des représentations de Maciste, je donne quelques reflexions que ce système m'a inspiré, bien que je n'en ai compris qu'une très petite partie.

Mots-clés: système, réflexivité, méta-connaissances, compilation, *Maciste*

1. Introduction

Jacques Pitrat a eu la gentillesse de me transmettre le système *Maciste* qu'il développe depuis une quinzaine d'années, en me prévenant que j'aurai du mal à appréhender son système, qu'il n'a pas développé pour être facilement utilisable par autrui, mais pour expérimenter quelques idées de recherche sur la méta-connaissance.

Maciste est un système complexe et original, difficile à appréhender. Il faudrait beaucoup de temps (plusieurs mois de travail à temps plein) pour le maîtriser, et le temps m'a manqué pour ce faire. Ce papier ne présente donc que des menues premières impressions d'un système important que j'ai tout juste eu le temps [2] d' effleurer.

2. Présentation du système *Maciste*

Le système *Maciste* est un système *réflexif* à base de [méta-] *connaissances*; ses connaissances sont formulées en des *règles* de production et d'autres formes plus déclaratives, -conseils...-, que je n'ai pas explorées. Ces règles sont regroupées en paquets, appelés *expertises*. En outre, *Maciste* utilise des symboles ou objets nommés, appelés *individus* (par exemple, les expertises et les attributs sont des individus). Les individus contiennent des listes de couple attribut-valeurs.

Le système *Maciste* compile ses connaissances en *des formes exécutables* au moyen d'expertises de compilation. Il est auto-suffisant, à l'exception du noyau Linux, des bibliothèques système `libc` (pour les fonctions système et de base) et `libdl` (pour le chargement dynamique[3]) dont seules 20 fonctions sont

[1] 8, rue de la Faiencerie, 92340 Bourg La Reine – autre mél possible: `basile@tunes.org`

[2] Je n'ai consacré qu'une vingtaine d'heures à étudier *Maciste*, dans une version datant du printemps 2001.

[3] Sous Linux comme dans la plupart des Unix, un programme peut *durant son exécution* charger un module binaire (shared object) -par `module= dlopen(filename, flags)` - puis obtenir dans ce module l' adresse d' un symbole (donc d'une fonction) à partir de son nom par `funptr= dlsym(module, symbolname);` Enfin, il peut décharger

utilisées: `exit` pour s'arrêter, `getc, putc, open, read, write, fopen, fclose, fflush, fputs, puts, fscanf, lseek` pour les entrées-sorties, `dlopen, dlclose, dlsym, dlerror` pour le chargement dynamique de programmes générés à la volée, `signal, alarm` pour gérer les signaux `SIGBUS, SIGFPE, SIGALRM, SIGQUIT, SIGILL, SIGSYS, SIGSEGV` et `system` pour lancer des programmes externes, en particulier le compilateur `gcc` et les commandes `pwd, rm` et `date` (qu'on pourrait remplacer par les fonctions `remove`, `getcwd` et `time`, pour un gain marginal) et `diff` .

Maciste tourne sous Linux sur un PC (j'ai un Athlon 1,2GHz, 256 puis 512 Mo RAM, 40Go disque, Linux Debian/Sid glibc 2.2.4, noyau 2.2.19, Xfree86 4.1, gcc 3.0.1) et sur un portable (Pentium 233MHz, 10Go disque, Linux Mandrake 8.0, gcc-2.96) et devrait tourner tel quel sur tout PC Linux à processeur x86 (Pentium, Athlon, Transmeta, 486, ...). Jacques Pitrat l'a longtemps développé sur une lente SparcStation1 (25MHz?, 64Mo RAM, SunOS4). *Maciste* est probablement portable [4] sur toute machine à mots de 32 (ou 64?) bits disposant d'un compilateur C et la possibilité de chargement dynamique de code (par l'interface `dlopen` ou similaire) et devrait tourner sans modification majeure sur la plupart des machines sous Unix.

Maciste m'a été livré dans un répertoire contenant des fichiers générés par le système *Maciste* lui-même (ainsi que quelques scripts et routines C inutiles) :

- le fichier `dx.h` est un fichier d'entêtes en C pour `#include` et il comprend 1245 lignes
- le fichier `dy` est un fichier binaire d'amorce, lu et produit séquentiellement; il fait 123904 octets
- le fichier `dz` est un fichier binaire accédé et modifié par segments; c'est un gros fichier de 27416576 octets qui est partiellement lu par blocs (on gagnerait à le projeter en mémoire par l'appel système `mmap`) et il contient la totalité des connaissances de *Maciste*
- le fichier `dk` est un fichier binaire servant seulement à l'échange d'information entre deux versions de *Maciste*. Il contenait à la livraison 45 kiloctets et n'est pas indispensable pour installer ou lancer *Maciste*.
- Les 1328 fichiers de code C générés par *Maciste*. Chaque expertise de *Maciste* correspond généralement à **plusieurs** fonctions C (une fonction par mode d'appel de cette expertise). Chaque fonction C a son fichier [5]de code C. Par exemple, l'expertise `LIRB` (sans doute pour lire un bloc?) correspond aux fonctions `LIRB0 LIRB1 LIRB2` donc aux fichiers `LIRB0.c LIRB1.c` et `LIRB2.c`. Ces 1328 fichiers commencent tous par une ligne `#include "dx.h"` et totalisent 160472 lignes de code C. Certains fichiers de code générés semblent temporaires. Certaines expertises sont expansées en ligne à leur compilation par *Maciste* et ne correspondent à aucune fonction C.
- 35 fichiers inutiles -sauf pour déboguer- qui sont des shell-scripts d'une ou quelques lignes ou des petits programmes C

Pour installer *Maciste* il m'a juste fallu compiler à la main et linker l'ensemble des fichiers C par la commande `gcc -O2 [A-Z]*.c -o maciste -ldl` . Cette compilation dure un peu plus de 3minutes et produit un binaire d'un peu plus de 3 megaoctets.

Le système *Maciste* est autonome (à l'exception du noyau du système d'exploitation `Linux`, de la vingtaine de fonctions de `libc` ou `libdl`, et du compilateur `gcc`) et s'utilise interactivement. En particulier, son utilisateur ne fournit pas des fichiers (autres que ceux préalablement générés par *Maciste*, listés ci-dessus) mais interagit uniquement à travers l'expertise d'édition `EDITE`. Jacques Pitrat n'a pas développé *Maciste* pour qu'il soit facile à utiliser par autrui, de sorte que l'interface est trop minimaliste à mon gout, du niveau d'un éditeur ligne à ligne comme `ed` sous Unix ou `edlin` sous MsDos mais suffisante pour lui.

---

ce module par `dlclose`. On peut ainsi accéder dynamiquement à plus d'une centaine de modules. Windows permet aussi cela autrement.

[4] Le portage de *Maciste* sur une autre machine (par exemple un processeur 64 bits – Alpha, IA64, x86-64) ou un autre système (par exemple Windows ou HPUX) nécessiterait une très bonne connaissance de *Maciste* et la disponibilité d'un système *Maciste* opérationnel.

[5] Il serait un peu plus efficace de générer toutes les fonctions correspondant à une même expertise dans le même fichier C, mais le gain ainsi obtenu serait marginal: *Maciste* met un peu plus de temps à compiler une expertise en du code C que le compilateur gcc ne met à compiler ce code produit par *Maciste* en du code machine.

2.1. Un premier exemple de session *Maciste*

Voici (en figure 1 ci-dessous) un exemple simpliste de session d'utilisation de *Maciste*. Les commandes que j'ai tapées sont en `courier oblique` et les réponses du système sont en `courier droit`, tout le reste (en police Times) étant mes commentaires explicatifs. Il s'agit juste d'afficher l'expertise `PGCD` qui calcule le PGCD de deux nombres entiers selon Euclide. Je n'ai tapé que la commande `L PGCD` pour lister cette expertise puis la commande `#` pour quitter *Maciste*. Le processus *Maciste* occupe 8 megaoctets de mémoire. Le temps CPU est négligeable (moins de 0,12s dont 0,09s en appel système, notamment plus de 430 appels à `read` pour lire par bloc d'un kiloctet les fichiers `dy` et `dz`).

Cette expertise `PGCD` est compilée par *Maciste* en deux fichiers `PGCD0.c` (30 lignes) et `PGCD1.c` (20 lignes).

```
./maciste            (lancement de Maciste depuis le shell)
Maciste affiche la quantité de mémoire disponible
TRIPLETS DISPONIBLES 1782289
 L PGCD              je demande le listage de l'expertise de PGCD
******************************        $PGCD
**********************      Maciste affiche le titre (en une ligne)
DATE:1-1-2002       ... et la date d'aujourd'hui
EXPERTISE DE BASE
  voici les règles d'Euclide:
B>1,A>=B=>XX:=A,YY:=B
A>1,A<B=>XX:=B,YY:=A
deux règles sans conditions
=>H:=MOD(XX,YY)
=>R:=COND[H=0:YY,H>1,PGCD YY->XX H->YY LL<-R:LL,:1]

métainformations d'usage
0 APPARAIT DANS [$EVLD,ATOME18T,$CHPGCD,"Z29Z] DONNEES
[`B`,`A`] RESULTATS
[`R`]
1 APPARAIT DANS [$PGCD] DONNEES [`XX`,`YY`] RESULTATS
[`R`]
VARIABLES GLOBALES [`XX`,`YY`,`H`,`R`,`B`,`A`]
#  je tape le dièse qui fait arreter Maciste et me renvoie au shell
```

Figure 1: *une première session avec* Maciste

2.2. Le code généré par *Maciste*

Voici les fichiers `PGCD0.c` et `PGCD1.c` correspondants (indentés par mes soins pour la présentation) – voir figures 2 et 3 ci-dessous.

Le code C généré par *Maciste* est très simple, et utilise une portion réduite des possibilités du langage C:

- pas de passage d'arguments ni de résultats dans les routines générées – *Maciste* utilise sa propre pile pour les arguments (en conservant la pile d'appel du C pour les adresses de retour et les variables locales), le tableau d'entiers `int pile[40000];` déclaré dans `dx.h` et géré par *Maciste*;
- pour faciliter le chargement dynamique, *Maciste* appelle toutes ses fonctions indirectement par un tableau de pointeurs de fonctions `void (*f[3000])();` déclaré dans `dx.h`, rempli à l'initialisation et lors du chargement dynamique de programmes;
- pas d'utilisation des constructions structurées de contrôle du C – boucle `for`[6] et `while`, sélection par `switch`, blocs imbriqués, mais utilisation systématique d'étiquettes avec `goto` sans utiliser l'extension branchement indirect[7] du `gcc`, qui serait pourtant très utile.
- aucun calcul en nombre flottant simple `float` ou `double` !

[6] Il n'apparait qu'un seul genre de boucle `for`, sans doute une espèce de macro interne au générateur de code de *Maciste* – toutes les boucles `for` ressemblent à `for (a=x[jvj+7]; a>0; a=t[a]) if (s[a]==712) goto l70;`

[7] Le compilateur gcc accepte l'affectation d'une étiquette `lab` dans un pointeur `p` par `p= &&lab;` et permet ensuite un branchement *indirect calculé* très efficace par `goto (*p);`

- aucune utilisation des fonctions d'échappements `setjmp` et `longjmp` du langage C, utiles pour les traitements exceptionnels.
- pas d'utilisation explicite de `struct`-ures ou de pointeurs *Maciste* représente ses données dans des triplets stockés dans trois tableaux `int t[195001]; int s[195001]; short r[195001];` déclarés dans `dx.h` ce qui est à mon avis légèrement moins efficace que des pointeurs sur des objets représentés par des structures C. Les listes attribut-valeurs au coeur de *Maciste* sont représentées par un chaînage de tels triplets.
- La mémoire est allouée statiquement -dans des tableaux déclarés dans `dx.h` qui sont dimensionnés par *Maciste* à la compilation; ce serait plus souple de les allouer dynamiquement (par `malloc` ou `mmap`) et il serait beaucoup plus rapide et plus simple de projeter le fichier `dz` en mémoire par `mmap`.
- pas de mesure du temps de calcul par l'appel système standard `clock` ou `times` ou bien la librairie `libhrtime` [Rasmussen00] qui permettrait des mesures fines (microsecondes de CPU), utiles dans un système introspectif.

Le code C généré par *Maciste* est bien évidemment (et volontairement) très proche du code machine; on pourrait évidemment générer directement du code machine (au détriment de la portabilité) en s'aidant de `ccg` [Pium99] ou bien produire du code dans un langage de plus bas niveau, comme `C--` [Peyton99] qui a été explicitement conçu pour servir de langage cible genéré. On pourrait aussi générer du code octet, par exemple celui de la JVM [Lindholm96], malgré ses inconvenients.

Voici (figure 2) le premier fichier C généré par *Maciste* pour le PGCD. Il contient un appel à la fonction PGCD1 codée dans le second fichier généré:

fichier *PGCD0.c*

```
#include "dx.h"
void PGCD0(void )
{int XX,YY,H;
int B,A;  int R;
int WZ1,WZ2;
B=pile[v[22]];A=pile[v[22]+1];v[22]+=3;
WZ2=v[22]+2;WZ1=v[22]+1;
if((B<=1)) goto l1;
if((A<B)) goto l1;
XX=A;  YY=B;
l2: H=XX%YY;
if((H==0)) goto l3;
if((H<=1)) goto l4;
pile[v[22]]=YY;pile[WZ1]=H;
(*f[605])();     /*PGCD1(YY,H,R)*/
R=pile[WZ2];
l5: v[22]-=3;pile[v[22]+2]=R;return;
l1: if((A<=1)) goto l4;
if((A>=B)) goto l4;
XX=B;  YY=A;
goto l2;
l3: R=YY;
goto l5;
l4: R=1;
goto l5;
}
```

Figure 2: *première fonction correspondant à `PGCD`*

Et voici (figure 3) le second fichier `PGCD1.c` généré par *Maciste*. On y remarque que la récursivité terminale qui figure dans l'expertise originelle a été remplacée par une itération interne.

fichier *PGCD1.c*

```
#include "dx.h"
void PGCD1(void )
{int H,V2,V3;
int XX,YY; int R;

XX=pile[v[22]];YY=pile[v[22]+1];v[22]+=3;
V3=YY; V2=XX;
l1: H=V2%V3;
if((H==0)) goto l2;
if((H>1)) goto l4;
R=1;
l3: v[22]-=3;pile[v[22]+2]=R;return;
l2: R=V3;
goto l3;
l4: V2=V3;
V3=H;
goto l1;
}
```

Figure 3: *Seconde fonction correspondant à* `PGCD`

Maciste peut re-générer la totalité de son code (par la commande `u` de l'interface `EDITE`) en 4 minutes20 sec de CPU et 6 minutes à la pendule (y compris la compilation du code C produit par `gcc` **sans** optimisation `-O`), dans un processus occupant 22 Mo de mémoire. En regénérant ainsi *Maciste* dans un répertoire neuf (contenant au départ seulement le programme binaire `a.out` et les fichiers `dy dz`) j'obtiens 1154 fichiers C totalisant 138904 lignes et 4979761 octets. Les trois plus gros fichiers sont `COPWB0.c` (61842 octets, 1089 lignes) `DIAC0.c` (54284 o=983 l) et `COPK0.c` (51381 o= 955 l). Les trois plus petits fichiers sont `SEV0.c` (187o=11 l), `COMB0.c` (181o=12 l) et `SPC0.c` (147o= 10 l). Le fichier d'entête commun `dx.h` compte 1245 lignes (24603 octets) dont 1151 lignes de déclaration de fonctions générées. La plupart des fichiers sont petits (moyenne: 120 lignes ou 4315 octets): le premier quart (288) des fichiers fait au moins 4985 octets ou 117 lignes, le dernier quart des fichiers compte au plus 1243 octets ou 42 lignes; le fichier médian `ATOME38T2.c` a 2340 octets et 68 lignes., et les 159 plus gros fichiers (environ un septième) cumulent la moitié des lignes générées.

Quelques expertises se traduisent en beaucoup de fonctions (donc de fichiers) différents. Au plus, `INTERP` et `TRI` correspondent chacune à 19 fonctions différentes, mais la plupart des 750+ expertises se traduisent en seulement une ou deux fonctions chacune – seules 32 expertises se traduisent en 4 fichiers chacune au moins.

2.3. *Maciste* et d'autres systèmes

Il est difficile de trouver des systèmes (réflexifs, à base de connaissances) similaires à *Maciste*, mais il est remarquable que *Maciste* (développé par Jacques Pitrat tout seul) soit de complexité et de taille comparables à des systèmes réalisés par des équipes entières de plusieurs chercheurs ou développeurs:

- le compilateur du langage logico-fonctionnel *Mercury* fait 1652 milliers de lignes de code C, dont 1535 klignes générées à partir de 300 klignes de code *Mercury*.
- l'implémentation `gcl 2.3` du *Gnu Common Lisp* fait 130 klignes de C dont 35klignes générées
- le système réflexif procédural *Pliant* [Tonneau99] compte 66 klignes de *Pliant* et 15 klignes de C. Il génère du code machine x86 ou du code C.
- le langage Gnu Prolog 1.2.1 est implémenté en 21 klignes de Prolog et 61 klignes de C
- le langage fonctionnel Ocaml (compilateur natif + machine virtuelle) fait 140 klignes dont 46 klignes de C; le binaire produit `ocamlopt.opt` par le compilateur natif fait 1,3Mo.
- Le moteur d'inférences Clips compte 140 klignes.

Le compilateur gcc 3.0.1 (pour C, C++, Fortran) compte 1358klignes (dont certaines sont générées) et le noyau linux 2.2.19 compte 2650 klignes (la plupart sont des pilotes des nombreux matériels supportés par Linux). *Maciste* utilise peu de leurs fonctionalités.

3. Représentations pour l'utilisateur

L'utilisateur interagit avec *Maciste* au travers des expertises d'édition.

3.1. Expertises d'édition de *Maciste*.

L'interface utilisateur de *Maciste* est très frustre, mais suffisante pour J.Pitrat qui en a l'habitude. Étant seulement un utilisateur novice et occasionnel de *Maciste*, je trouve cette interface insuffisante et intimidante. L'utilisateur manipule des expertises et des individus au travers de l'expertise d'édition `EDITE` dont voici (en figure 4) un extrait (où on reconnait la règle qui arrête *Maciste* par l'entrée du caractère #):

extrait des règles d'*EDITE*

```
DABORD L<0,INCONNU(KR)=>LK,LL:=0
DABORD L>=0=>LL:=L
DABORD P=KPT(INT(@TMP)),P>0=>KR:='~',EAP
VALNUM(INT(@TMP))->NR PRED(P)->NK
=>KR:=CR(LL),UR:=CR(SUC(LL))
KR='E',CRR SUC(LL)->I DP<-J,P=CV(R,@OBJ)=>LEXP DP Z<-
EX DX<-DX,EVL Z XX->X
   R<-RES TY<-TY TZ<-TZ,DPG 1 DP ['('],DPG 3 DX
[')'],POURTOUS[TZ>0=>PLUS
   MNS(@TMP)
CV(TZ,@OBJ)],POURTOUS[TZ>sepbase,TZ<=sepcod=>PLUK
CV(TZ,
   @OBJ)->BL @OUI->M @ORDRE->A], ENDERNIER ENLEVE
{RECHER(SUC(LL))}(P)
KR='m'=>SNUM
KR='-',UR=' '=>MAIN
KR='+',UR=' '=>DISQUE,NOUVER(@TMP)=(NIV:0),ZV 14 0
KR='#'=>STOP                          la règle arrêtant Maciste
KR='g'=>VG (DEJA:$MAIN)->X
KR='f'=>FICTIF
KR='+',UR.APP.['#','+']=>DISQUE, ENDERNIER STOP
KR='£'=>OTEINUTILE LL
KR='t'=> DABORD DATE @OUI->P,TABLERASE, ENDERNIER
POURTOUS[=> DABORD DATE
   @OUI->P,DISQUE, ENDERNIER STOP]
KR='r'=>VERIF
etc....
```

Figure 4: *Extrait de l'expertise d'édition principale*

L'expertise `EDITE` est quatre fois plus grosse que cet extrait, et elle est compilée en deux fonctions `EDITE0` et `EDITE1` totalisant 1419 lignes de C. Comme on peut le deviner, l'expertise principale d'édition n'est pas très déclarative (car l'édition est par essence une activité procédurale pour la machine, car dirigée par les interactions de l'utilisateur), et peu modulaire. On aurait pu, par exemple, décrire chaque possibilité ou commande d'édition par un individu ou un objet la décrivant. Mais Jacques Pitrat a choisi de concentrer ses efforts de développement sur d'autres expertises et son éditeur lui suffit.

Les messages d'erreur sont souvent très réduits: une erreur de saisie provoque le plus souvent un simple bip!

Pour manipuler des expertises, l'utilisateur peut lancer l'expertise `EDIBLOC` d'éditions de bloc (par la commande `T` de `EDITE`); l'utilisateur dispose alors d'un éditeur syntaxique frustre (`T` pour aller en haut, `D` pour descendre, `Y` pour visionner l'élément courant, etc...) pour modifier l'expertise ainsi éditée.

Habitué au confort d'`emacs` ou des interfaces graphiques, je trouve l'interface de *Maciste* trop aride pour pouvoir être utilisée agréablement; pour cela, il serait idéalement souhaitable d'apporter l'une des améliorations suivantes à *Maciste* :

- permettre l'édition d'une ligne en cours de saisie [8], par exemple en s'interfaçant à la bibliothèque `readline` (comme le font beaucoup de programmes: `bc, mysql, ncftp, prolog, scm`, etc...); actuellement *Maciste* lit une ligne entière (tamponnée par la gestion des terminaux `tty` du noyau Linux), traitée caractère par caractère par `getc`;
- gérer le terminal en plein écran au moyen de la bibliothèque `ncurses` (comme le font `zsh, gdb, hugs`, et les éditeurs `emacs, vi, joe`...)
- fournir une interface graphique multifenêtrée [9], au moyen d'une bibliothèque toolkit comme `gtk` [Harlow99], `qt`, ... (le code C à générer est verbeux, mais facilement générable)
- s'interfacer -comme le font beaucoup de programmes, dont `pliant-` à un navigateur Web (tel que `netscape` ou `mozilla`...) par le protocole HTTP (avec HTML), faisant de *Maciste* un mini-"serveur" Web spécialisé.

A mon avis, *Maciste* ne fournit pas suffisamment d'informations en retour à l'utilisateur; en particulier, il n'y a pas d'expertise d'aide dans *Maciste*. L'un des intérêts des interfaces graphiques est d'expliciter à l'utilisateur les possibilités d'interaction : par exemple, un menu déroulant affiche l'ensemble de ses choix. Peut-être qu'un modèle de l'utilisateur serait utile à *Maciste*, pour s'adapter aux novices comme aux experts..

3.2. Représentation des connaissances dans *Maciste*

Les connaissances de *Maciste* sont essentiellement regroupées en expertises, chacune pouvant être listée par la commande `L` de l'expertise `EDITE`. Comme J.Pitrat l'a écrit ailleurs, *Maciste* contient des expertises sous plusieurs formes, notamment une forme passive pouvant être modifiée, une forme active interprétable, des formes exécutables, etc... J'ai réussi une fois[10] à obtenir de *Maciste* un listing de 10520 lignes listant apparemment un grand nombre (plusieurs centaines) d'expertises.

Maciste contient aussi des individus, qui sont des objets nommés (semblables aux symboles lispiens). Chaque individu contient une liste de couples attribut-valeurs. Les expertises, les fonctions, les attributs sont des individus. La commande `m` de l'expertise `EDITE` en donne la liste; il y en a plus de 2000.

En mémoire centrale, les expertises sont représentées dans des listes chainées de couples attribut-valeurs, où l'attribut est un individu et sa valeur est un individu, un nombre entier, une liste de telles valeurs, etc... De façon persistante, les informations de *Maciste* sont essentiellement dans le fichier `dz` (de plus de 27Mo) et y sont accédées et écrites par blocs de 1024 octets. Les fichiers binaires de *Maciste* (`dy` et `dz`) semblent portables sur des machines [11] petit- ou grand- indiens (et ont permis à J.Pitrat de passer d'une station Sun/Sparc à un PC/Linux). Leur gestion comporte l'expertise de `MENAGE`.

La gestion de la mémoire se fait par des expertises comme `GARBAG` et `GARBAH`, et semble utiliser un procédé simple de marquage (mark&sweep). A mon avis, il serait facile et avantageux d'implémenter un ramasse-miettes générationnel copieur [Jones96] qui sera beaucoup plus efficace, et permettrait d'utiliser très intensivement l'allocation dynamique [Appel87].

Je n'ai qu'une **très petite compréhension** du système *Maciste*. En effet, les intentions de Jacques Pitrat ne sont pas toujours évidentes. Par exemple, l'expertise `ALCTV` est très petite et son nom ne me parle guère; en voici (figure 5) la liste:

---

[8] On pourrait aussi mettre le terminal en mode raw par les fonctions `tcsetattr` et `cfmakeraw` ou bien la commande `stty raw` mais je recommande nettement les bibliothèques `readline` ou `ncurses`

[9] On pourrait aussi s'interfacer directement à la `libX11` voire au protocole X11 lui-même, mais je recommande l'utilisation d'une bibliothèque d'interface graphique.

[10] Mais je n'arrive plus, au moment où je rédige cet article, à reproduire la séquence de touches qui fait écrire par *Maciste* un grand nombre d'informations.

[11] Les architectures des machines sont dites "little-endian" ou "big-endian" selon que l'octet de poids fort d'un mot est celui d'adresse la plus haute ou la plus basse dans ce mot.

expertise *ALCTV*

```
te ********************************      $ALCTV
*********************
DATE:2-1-2002

LANT DP EXX<-EX DXX<-DF=>EX:=CREE(%EXX),DX:=DXX,PLUS
AVEC(AL) EX
CR(SUC(DX))#' ',SMV DX->I R<-R DXX<-DX,R=`POUR`=>LCANT
EX->R DXX->DP DFF<-DF

0 APPARAIT DANS [$EDIPB] DONNEES [`DP`,`AL`] RESULTATS
VIDE
VARIABLES GLOBALES [`EX`,`DX`,`DP`,`AL`]
```

Figure 5: *listage complet de l'expertise* `ALCTV`

La seule fonction C correspondant à cette expertise est dans le fichier `ALCTV0.c` listé en figure 6:

fichier *ALCTV0.c*

```
#include "dx.h"
void ALCTV0(void )
{int DX,V4,V3,R,V2,DFF;
int DP,AL;
int WZ1,WZ2,WZ3,WZ4;
int jvj;
jvj=v[0];
v[0]+=4;
x[jvj+1]=10127;z[jvj+1]=(-100);
if(v[90]==352&&v[97]==0) {
(*f[2])();v[22]+=2;goto l1;
}
DP=pile[v[22]];AL=pile[v[22]+1];v[22]+=2;
WZ4=v[22]+4;WZ3=v[22]+3;WZ2=v[22]+2;WZ1=v[22]+1;
pile[v[22]]=DP;pile[WZ1]=jvj+2;
(*f[322])();if(v[102]) goto l1;
/*LANT0(DP,jvj+2,DX)*/
DX=pile[WZ2];
pile[v[22]]=jvj+2;pile[WZ1]=jvj+3;
(*f[197])();     /*CPI1(jvj+2,jvj+3)*/
pile[v[22]]=100;pile[WZ1]=455;pile[WZ2]=102;pile[WZ3]=
jvj+3;pile[WZ4]=jvj+4;
(*f[206])();     /*QUADRI4(100,455,102,jvj+3,jvj+4)*/
pile[v[22]]=AL;pile[WZ2]=jvj+4;
(*f[33])();     /*PLUSC0(AL,455,jvj+4)*/
V4=DX+1;
V3=bh[v[1]][V4];
if((V3==32)) goto l1;
pile[v[22]]=DX;
(*f[124])();     /*SMV0(DX,R,V2)*/
R=pile[WZ1];V2=pile[WZ2];
if((R!=(-3049))) goto l1;
pile[v[22]]=V2;pile[WZ1]=jvj+4;
(*f[294])();     /*LCANT0(V2,jvj+4,DFF)*/
DFF=pile[WZ2];
l1:x[jvj+1]=incon;v[0]=jvj;v[22]-=2;return;
}
```

Figure 6: *listage complet de* `ALCTV0.c` *produit par compilation de* `ALCTV`

Je laisse au lecteur le soin de deviner ce que fait (ou que devrait faire) cette expertise `ALCTV` et cette fonction. Je n'en ai malheureusement aucune idée.

A mon avis, il manque dans chaque individu, et en particulier dans chaque expertise, un attribut `DESCRIPTION` qui devrait être associé à une chaine (ou idéalement un commentaire hypertextuel actif) donnant l'intention du développeur (Jacques Pitrat) lorsqu'il a entré cet individu. Il y a beaucoup d'autres expertises de même acabit dans *Maciste*: je les devine pleines de science, mais je n'arrive pas à en

comprendre le sens, l'intention, le role, ou la spécification. Seul Jacques Pitrat pourrait remplir cet attribut (et un commentaire même incomplet serait déjà très précieux!).

Peut-être qu'un espace hiérarchique de noms (comme les packages de *CommonLisp* ou de *Java* par exemple) rendrait aussi plus facile la compréhension et l'extension de *Maciste*.

Les individus de *Maciste* ne semblent pas versionnés (il parait que les deux dernières versions sont sur disque dans `dy`). Jacques Pitrat utilise chez lui des répertoires différents pour en garder des versions plus anciennes. Une commande d'`EDITE` permet de lancer la commande `diff` relativement à un autre répertoire. On pourrait interfacer un versionneur comme CVS (`www.cvshome.org`) ou plutôt `PRCS` (prcs.sourceforge.net)à *Maciste*, en associant à chaque individu le numéro de version de sa dernière modification.

Comme on peut l'imaginer, je n'ai qu'une **vision très floue** du système *Maciste* de Jacques Pitrat. Ce système m'a inspiré quelques reflexions (nécessairement spéculatives) sur les systèmes reflexifs à base de connaissances.

4. Réflexions et interrogations sur les systèmes réflexifs à base de connaissances

Les systèmes réflexifs sont encore très rares, même si l'idée de reflexivité partielle commence à percer, notamment dans l'informatique actuelle (en particulier, Java a quelques menus aspects reflexifs, bien insuffisants). Je donne ici quelques reflexions sur les systèmes d'exploitation, les compilateurs.

Jacques Pitrat dit souvent que le système d'exploitation et le compilateur C sont *des boites noires dont il faudrait se débarasser*. Si je partage ce point de vue dans son principe, je pense que des considérations pratiques devrait nous inciter à utiliser ces boites noires (et d'autres). A contrario, on pourrait même refuser la boîte noire (ou beige) constituée par le matériel informatique lui-même, et vouloir que la machine se bâtisse[12] ou s'assemble elle-même physiquement.

4.1. Place et rôle du système d'exploitation

Pike [Pike00] constate une stagnation dans la recherche sur les OS. Le projet *Tunes* [Rideau99] vise à construire un système d'exploitation reflexif.

De mon point de vue, il faut (particulièrement avec les OS actuels, Unix, Linux, Windows, Plan9, Hurd...) distinguer les systèmes d'exploitation au sens large (comprenant les nombreux logiciels utiles livréa avec le système) de leur noyau au sens strict. Bien sur, le fondement des systèmes actuels est à revoir: les notions de fichiers et de processus sont évidemment obsolètes (datant des années 1970, elles sont inadaptées à nos machines ayant des processeurs d'1GHz, des mémoires de 1Go, des disques de 100Go, en permanence branchées sur l'Internet avec 100millions d'autres machines et une bande passante de l'ordre du 1Mo/sec), la persistance y est inexistante ou insuffisante.

On peut s'intéresser au développement d'un système d'exploitation reflexif; la robotique ou les télécommunications [Stefani99] en serait des applications prometteuses. Dans cette optique, la gestion fine des ressources, permises par la reflexivité, est un objectif principal: il faut alors s'intéresser à gérer finement le temps, la mémoire, les communications, la distribution, la concurrence, en étant l*e plus proche* possible de la machine (physique ou virtuelle, par exemple la JVM). Ce ne semble pas être l'optique de *Maciste* (il faudrait alors renoncer fortement au C, gérer octet par octet la mémoire et sa pagination, dépendre du processeur x86, s'interfacer aux nombreux périphériques physiques...) et *Maciste* me semble loin de ce but (ses représentations physiques, son code généré, sa gestion de la mémoire et du temps ne sont pas localement optimaux). Une optique "système d'exploitation réflexif" doit être très proche, donc **très dépendante**, de la machine (donc difficile à porter initialement [13]sur une autre machine). L'adaptation à une machine physique, en particulier l'écriture des nombreux pilotes des periphériques utiles (compte tenu de la diversité du matériel actuel, notamment les cartes graphiques et les cartes réseau sur les PC), demande un travail (d'ingénierie) colossal – qui ne me paraît pas à la portée d'un homme seul ou d'une petite équipe. On

[12] Ce fantasme était constitutif de la machine Gosseyn du regretté Jean-Marc Fouet [Fouet87]

[13] Un système déclaratif réflexif complet et complexe serait sans doute facilement portable une fois qu'il est complétement réalisé: il suffirait d'y changer la description de la machine physique où il tourne.

pourrait par exemple s'appuyer sur un noyau existant, ou bien développer un noyau au dessus de l'OsKit [Ford97], ou d'autres noyaux existants.

Il me parait plus raisonnable de s'appuyer sur les systèmes d'exploitation actuels -tels que Linux-; A la limite, on pourrait ne s'appuyer que sur le noyau [14](comme le font Pliant ou certains compilateurs Lisp) sans utiliser aucune des bibliothèques usuelles (pas même la `libc`). Cela demande alors de réaliser dans le système certaines fonctionalités utiles pourvues par la `libc` (par exemple la conversion précise -difficile à coder- entre flottants et chaînes de caractères `strtod` et son opposé `sprintf("%g", ..)`ou les fonctions mathématiques de la `libm` et les appels aux services de noms `gethostbyname` qui peuvent utiliser le DNS). Pour le coder rapidement, on pourrait coder en C un petit programme serveur pour ces appels et faire communiquer le système réflexif avec ce serveur ad hoc.

Utiliser les bibiothèques systèmes, telles que la `libc`, `libdl` et la `libm`, est un compromis raisonnable. En effet, elles fournissent des fonctions très utilisées, donc généralement standardisées, stables, robustes et bien déboguées. La disponibilité de leur code source sous Linux est un avantage supplémentaire pour en comprendre, si besoin était, le fonctionnement intime.

En poursuivant cette idée d'utiliser les bibliothèques systèmes, on pourrait accepter l'idée d'utiliser d'autres bibliothèques usuelles sous Linux, par exemple une bibiothèque d'interface graphique comme `gtk` ou d'interface à un SGBD comme `mysql` ou même le ramasse-miettes conservatif de H.Boehm. La difficulté pratique est d'adapter les conventions d'appel et le modèle de données du système reflexif à ceux usuels en C. Du point de vue IA, il est alors important de donner au système les connaissances (qui sont dans la tête du développeur) pour utiliser efficacement ces bibliothèques. Pour des utilitaires complexes comme gtk ou mysql, cela représente un volume certain de connaissances.. D'un point de vue plus social ou politique, faire un système d'IA qui sache profiter des logiciels existants facilitera son acceptation par une large communauté d'utilisateurs. Les systèmes actuels (et notamment Linux) comportent une grosse quantité de logiciels utiles. Pourquoi nos systèmes d'IA n'en profiteraient-ils pas?

Un utilitaire important est bien sur la chaîne de compilation, notamment le compilateur C `gcc`, l'éditeur de liens `ld`, le chargeur dynamique `libdl` voire éventuellement le constructeur de programmes `make` (que Maciste n'utilise pas).

4.2. La chaine de compilation du système d'exploitation

Il s'agit ici de la chaine de compilation du système d'exploitation utilisé, c'est à dire sous Linux du compilateur `gcc` – qui compile un sur-ensemble de C - (et donc aussi du préprocesseur `cpp`, de l'assembleur `as`, de l'éditeur de liens statique `ld`, de l'éditeur de lien dynamique `ld.so` et du chargeur dynamique `libdl`). Il pourrait être aussi utile d'utiliser le constructeur de programme `make` (notamment pour lancer 2 compilations C simultanément par `make -j2`).

Générer du code C est une pratique fréquente, en particulier dans de nombreux compilateurs (Mercury, Haskell, Godel, Eiffel, Ada, les compilateurs C++ d'autrefois, avec `cfront`). C'est aussi dans la lignée originelle d'Unix (qui prone la réutilisation des briques existantes, assemblées par des pipes). C'est aussi une relative facilité (bien que la syntaxe de C soit pénible à générer, car peu orthogonale) pendant le développement du système reflexif générant le code C.

L'avantage mineur de générer du code C est -pour le développeur du système reflexif générateur de C- la possibilité de lire (voire de modifier) les fichiers C produits. Un autre avantage est la relative portabilité du code C généré (qui peut être moins forte [15]qu'on ne le croit): le code C produit est probablement compilable sur une machine similaire. Par ailleurs, le système générant du C n'a pas à se préoccuper des détails de la génération de code machine (choix des instructions machines, ordonnancements, affectation des registres) qui sont de la responsabilité du compilateur C.

Mais générer du code C a aussi des inconvénients; d'abord, le coût de la génération du code C: il faut produire du code dans un fichier textuel selon une syntaxe complexe; ensuite, la compilation par `gcc` du code C produit prend un certain temps (`gcc` met plus d'un vingtième de seconde à compiler l'un des fichiers

[14] C'est à dire les 225 appels systèmes énumérés dans `<asm/unistd.h>` dont le protocole d'appel sur PC/x86 est décrit en [Boldyshev01] ce qui permet d'utiliser un appel système (sans passer par la libc) au moyen de l'instruction machine `int 0x80`

[15] Les compilateurs Mercury et Cfront (implémentation originelle de C++) génèrent du code C qui dépend de la machine cible, notamment de la taille des mots (32 ou 64 bits) et de leur indianité (little or big endian).

C les plus courts produits par *Maciste*). Ceci réduit les possibilités d'experimentation fine du système reflexif (par exemple, il serait inconfortable pour l'utilisateur de générer puis exécuter du code à chaque interaction élémentaire[16] telle qu'un clic de souris et il est impraticable de générer du code qui ne servirait que quelques millisecondes). Enfin et surtout, le langage C a des manques flagrants, et des défauts majeurs (absence d'appel recursifs terminaux indirects, incompatibilité des invariants d'un ramasse-miettes avec l'optimiseur du C [Boehm92], syntaxe difficile à produire).

Pour pallier à ces inconvénients, S.Peyton-Jones & al. proposent [Peyton99] le langage `C--` (dont *QuickC--* est une implémentation prometteuse libre, en cours de développement). Ce langage vise à être un assembleur portable, exclusivement destiné à servir de langage intermédiaire pour les systèmes générant du code.

Il faut noter que générer du code machine de qualité médiocre (c'est à dire deux ou trois fois plus lent que celui produit par un compilateur C optimiseur) est simple; on ne se soucie pas de l'allocation des registres ou bien de la selection fine des instructions, et on produit en séquence les instructions machine par un mécanisme de macro-expansion rudimentaire. Seul le codage des instructions elles-même reste fastidieux, et on peut s'aider pour ce faire du préprocesseur `ccg` [Piumarta99]. Dans ce cas, je pense que la traduction de la représentation interne (quasi-arborescente) de *Maciste* la plus proche du C vers du code machine demanderait tout au plus une centaine de règles simples.

En générant du code machine, on pourrait alors se passer complétement d'interpréteur (comme l'expertise `INTERPRETE` de *Maciste*, qui interprète une forme intermédiaire des expertises existantes); l'introspection caractéristique des systèmes reflexifs pourrait se faire par génération de code verrue appelant l'introspecteur. Les objets contenant du code machine devront être traités spécialement[17]: ils ne doivent pas être déplacés sans relocation ou regeneration. Je pense qu'un système réflexif gagnerait beaucoup à générer du code machine directement (comme le font Squeak pour Smalltalk et les JIT pour Java), même si ce code n'est pas d'excellente qualité. En effet, cela permet de générer rapidement des routines temporaires – un tel système pourrait générer en moins d'une milliseconde une routine spécifique temporaire qui n'aurait d'utilité que pendant une fraction de seconde. Ceci permettrait au système d'être très dynamique et adaptatif. Ces ordres de grandeurs très petits, et très différents de ceux obtenus en générant du code `C` ou `C--`, permettrait des expérimentations nouvelles. Par exemple, le système réflexif pourrait adapter dynamiquement sa façon de ranger les attributs dans les objets en mémoire. Je rejoins alors l'idée de Jacques Pitrat de se débarasser du compilateur C, mais je trouve que ça serait assez facile à faire dans *Maciste*. Il serait judicieux de garder la génération de C et d'y ajouter la génération dynamique de code machine.

Jacques Pitrat m'a brièvement expliqué la génération de code dans *Maciste*. Chaque expertise est compilée différemment selon son contexte d'appel (c'est là où la reflexivité est déterminante: toutes les utilisations d'une expertise sont connues de *Maciste*). L'expertise `C` lance `CB`, qui segmente parfois l'expertise à compiler par `SEGM.` Une autre expertise `ORB` transforme les règles prétraitées en graphes interprétables. `CAN` fait de l'inférence de types, comme dans les langages fonctionnels tels que Caml. Une expertise `CF` fusionne les graphes, et une autre `CE` génère le code en langage C. Hélàs, ces expertises font parties de celles, les plus nombreuses, que je n'ai pas eu le temps de comprendre.

4.3. Ce que je n'ai pas su voir dans *Maciste*

Ce que je n'ai pas su appréhender dans *Maciste*, ce sont les fonctionalités et les expertises les plus importantes de ce remarquable système : le compilateur d'expertise, le résolveur de problème MALICE, le module espion MESPION, le monitoring [Pitrat00], l'extension du langage par bootstrap, etc...

Il eu fallu disposer de beaucoup de temps pour étudier *Maciste*. Je ne suis pas certain qu'une année à temps plein me suffise à comprendre *Maciste*, au point de pouvoir y contribuer comme j'avais initialement rêvé de le faire. J'espère que d'autres que moi auront le privilège d'accéder à *Maciste*.

---

[16] Certaines implémentations de CommonLisp, Smalltalk ou Self généraient du code aussi fréquemment. La vitesse de génération du code, et la possibilité de ramasser (par le GC) le code devenu inutile, sont alors plus importantes que la qualité et la rapidité du code machine généré.

[17] Sur certaines architectures de processeurs, il faut signaler explicitement au cache le code machine nouvellement généré. Ce n'est pas requis sur x86. Le lecteur curieux pourrait regarder le code dépendant machine de `SML/NJ 110` pour les détails. Meme si l'on dispose d'un ramasse-miettes copieur, je déconseille le déplacerment des objets contenant du code machine comme l'a fait SML/NJ

Il est possible qu'il faille, pour comprendre un système reflexif aussi important que *Maciste*, disposer non seulement du système et de ses explications, mais aussi de son historique, c'est à dire de ses versions successives au cours du temps.

4.4. En guise de conclusion : intérêt et acceptation politique des systèmes réflexifs

Un système réflexif est intéressant par les possibilités inégalées d'évolution et d'adaptation qu'il pourrait offrir. Cet avantage se devine dès maintenant dans les systèmes de logiciels libres[18] actuels - comme l'ensemble des logiciels sous Linux, qui constituent ensemble un système aveuglément reflexif, en ce sens qu'il est suffisant pour se générer lui-meme à partir de la totalité de son code source (on pourrait sur un PC Linux disposer du code source et recompiler l'ensemble des programmes qui y sont installés), code qui est librement disponible. D'autre part, le mouvement du logiciel libre est aujourd'hui suffisamment accepté (pour etre encouragé et soutenu, y compris financièrement, par des entreprises privés, petites ou multinationales, l'Etat, la Commission Européenne,[19] ...). Il me semble ressembler, dans son idéal de totale transparence des codes, au paradigme des systèmes reflexifs.

J'y vois une espérance et une attente: l'espoir de voir un jour un système reflexif publié comme un logiciel libre, et l'attente implicite, par la communauté informe du logiciel libre, de systèmes radicalement meilleurs que les Linux d'aujourd'hui. Ce pourrait être des systèmes reflexifs (en logiciels libres) notamment s'ils trouvent des applications porteuses (par exemple autour du Web).

Le remarquable travail de Jacques Pitrat sur *Maciste* montre que le développement d'un système reflexif est une entreprise très vaste, et je pense que le modèle de codage coopératif du logiciel libre pourrait lui être valablement appliqué. En outre, la publication de logiciels réflexifs à base de connaissances permettrait de convaincre ceux qui sont sceptiques de l'intérêt ou de la validité de cette approche à base de méta-connaissances.

Remerciements.

Je remercie beaucoup Jacques Pitrat de m'avoir fait l'honneur de me confier une copie de son système *Maciste* ; merci aussi à Faré Rideau d'avoir relu une première version de ce texte, et à mes filles Anne et Marie d'y avoir corrigé quelques fautes.

Références bibliographiques

[Appel87] Appel, A.: *Garbage Collection can be faster than stack allocation.* Information Processing Letters, 25(4), pp275-279, 1987

[Boehm92] H.Boehm & D.Chase: a proposal for Garbage-Collection Safe C Compilation, Journal of C Translation 4(2), pp126-141 december 1992 et `http://www.hpl.hp.com/personal/Hans_Boehm/`

[Boldyshev01] Boldyshev K.: *The Linux Assembly HOWTO*, `http://www.linuxdoc.org/HOWTO/Assembly-HOWTO` **2001**

[Fouet87] Fouet J-M: *Utilisation de Connaissances pour améliorer l'utilisation de connaissances : la machine Gosseyn,* thèse d'Etat, Univ. Paris 6, septembre 1987

[Ford97] Ford B. & al: *The Flux OSKit: a Substrate for OS and Lauguage Research,* **16**th ACM symposiom on Operating Systems Principles, StMalo, oct.1997 **`http://www.cs.utah.edu/flux/oskit/`**

[Harlow99] Harlow, G**.:** *Developing Linux Applications with GTK+ and GDK***,** NewRiders Publishing, 1999 (isbn 0-7357-0021-4) - voir aussi `www.gtk.org`

[18] Je ne distingue pas ici logiciels libres et opensource. Voir www.aful.org www.opensource.org www.fsf.org www.gnu.org www.april.org www.freshmeat.net etc... On peut aussi trouver des justifications ou explications macro-économiques et sociales du logiciel libre.

[19] Les appels nationaux comme le RNTL ou européens comme PCRD (IST) mentionnent explicitement le logiciel libre et pourraient peut-être soutenir le développement de systèmes reflexifs libres, s'ils ont des applications porteuses.

[Jones96] Jones, R. & Lins, R.: *Garbage collection (algorithms for automatic dynamic memory management)*, Wiley 1996 (isbn 0-471-94148-4)

[Lindholm96] Lindholmm, T et Yellin F**.:** *the Java Virtual Machine Specification*, AddisonWesley 1996 (isbn 0-201-63452-x)

[Puimarta99] Puimarta, I**:** *dynamic code generation for C and C++ http://www-sor.inria.fr/projects/vvm/ccg/ 1999*

[Pitrat96] *Pitrat J.: Implementation of a Reflective System* Future Generation Computing Systems 12, pp235-242, 1996

[Pitrat00] Pitrat J.: *La mise en place d'un monitoring dans Malice* Colloque Berder Septembre 2000 (LIP6)

[Rasmussen00] Rasmussen N. : *high resolution process timing for the linux kernel*. http://www.cs.wisc.edu/~paradyn/libhrtime/ 2000

[Peyton99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig: *C--: a portable assembly language that supports garbage collection* PPDP, October 1999, www.cminusminus.org

[Pike00] Pike R.: **S***ystems Software Research is Irrelevant*, http://www.cs.bell-labs.com/who/rob/ 2000

[Rideau99] Rideau F. & al: le projet Tunes http://www.tunes.org/ 1999-2002

[Stefani99] Stéfani J-B: *On the reflective structure of information networks* (Reflection99, StMalo) Springer LNCS1616 pp93-94

[Tonneau99] Tonneau H. & al: *le système Pliant* http://pliant.cx/ 1999-2002

LA GENERATION DE SOLUTION AVEC LE SYSTEME MARECHAL SUR UN EXEMPLE

Tristan Pannérec
Laboratoire d'Informatique de Paris VI
Tristan.Pannerec@lip6.fr

Résumé :

Le système MARECHAL est un système général pour la résolution de problèmes d'optimisation demandant beaucoup de connaissances. Il fonctionne grâce à une architecture méta : une partie basique permet de générer des solutions pendant qu'une partie méta observe et contrôle la précédente pour obtenir la meilleure solution possible. Dans ce papier, nous allons décrire la partie basique du système et les connaissances qu'elle utilise. Nous nous appuierons pour cela sur l'exemple de la génération d'emploi du temps dans un établissement scolaire.

Mots-clés : Résolution de problème, Problèmes d'optimisation, système expert, emploi du temps.

1. Introduction

Le système MARECHAL est un système général pour la résolution de problèmes d'optimisation complexes demandant beaucoup de connaissances. Le but est de coordonner un ensemble important de décisions pour maximiser un critère. Dans de tels problèmes, les techniques classiques de recherche opérationnelle se révèlent parfois inadaptées à cause de la taille des espaces de recherche, de la complexité des fonctions d'évaluation ou de la structure non discrète des espaces. Il est indispensable, dans de tels cas, d'utiliser des connaissances pour guider la recherche et d'avoir un contrôle intelligent sur cette recherche.

Le système MARECHAL fonctionne grâce à une architecture méta[20] : une partie basique permet de générer des solutions pendant qu'une partie méta observe et contrôle la précédente pour obtenir la meilleure solution possible. Il a été appliqué à un jeu de réflexion complexe où les joueurs programment les mouvements de leurs pièces, ceux-ci s'effectuant ensuite de manière simultanée. Il est également appliqué au placement automatique de composants électroniques sur les circuits imprimés.

Dans ce papier, nous allons décrire la partie basique du système qui permet de générer des solutions. Nous présenterons son architecture, son fonctionnement et les connaissances qu'elle utilise. En parallèle, nous décrirons le langage qui permet de donner ces connaissances au système de manière déclarative. A titre illustratif, nous nous appuierons sur l'exemple de la génération d'emploi du temps dans un établissement scolaire. Le but n'étant pas de résoudre ce problème, nous utiliserons une formulation très simplifiée. Ce problème sera décrit dans la première partie du papier. La partie suivante donnera un aperçu global de l'architecture du système MARECHAL. Les trois autres parties détailleront chacune une des trois couches de la partie basique du système et présenteront en même temps les éléments du langage qui les concernent. Finalement, nous terminerons par une conclusion et une partie annexe qui contient la description complète du langage et l'ensemble des connaissances permettant de résoudre le problème simplifié des emplois du temps.

[20] Voir [Maes & Nardi 88] et [Pitrat 91] pour les architecture méta et [Pitrat 90] pour l'utilisation de métaconnaissances.

2. Un problème pour illustrer

Afin d'illustrer la description qui suit du système, nous allons nous appuyer sur un exemple « fictif ». Les applications réelles auxquelles le système MARECHAL a été confronté sont en effet trop complexes et volumineuses pour servir d'illustration[21]. Nous avons donc choisi de développer ici le problème de la génération d'un emploi du temps pour un établissement scolaire. Naturellement, le but n'est pas de résoudre ce problème, mais de s'en servir pour expliquer comment fonctionne le système MARECHAL. C'est pourquoi, non seulement le problème sera très simplifié, mais en outre, la méthode de résolution qui sera donnée au système ici ne correspondra pas forcément à celle qu'on lui donnerait si on voulait réellement résoudre ce problème. Naturellement, cette simplification fait que ce problème peut être très facilement résolu par des techniques classiques basés sur la combinatoire.

2.1. Description informelle du problème

Nous considèrerons donc simplement un ensemble de M professeurs, un ensemble de N classes et un ensemble de K cours associant chacun un professeur avec une classe. Chaque cours est supposé durer deux heures, un professeur pouvant avoir plusieurs cours avec la même classe et inversement. Nous négligerons les problèmes de salles et supposerons que les classes et les professeurs ont peu de cours par rapport au nombre de créneaux horaires possibles par semaine, de façon à se ramener à un problème sous-contraint (où le nombre de solutions possibles est énorme, le problème étant d'en trouver une bonne). La semaine sera composée de 5 jours ouvrables (lundi, mardi, mercredi, jeudi et vendredi) contenant chacun deux demi-journées. Les demi-journées seront découpées en deux créneaux horaires de deux heures.

Le principe est de placer chaque cours dans la semaine de façon à ce qu'une même classe ou un même professeur n'ai pas deux cours au même moment. En outre, deux cours identiques (même classe et même professeur) ne peuvent être ensembles dans la même demi-journée. Le but est que les professeurs aient un maximum de demi-journées et de journées libres et, dans une moindre mesure, que les classes aient également le maximum de demi-journées libres[22].

2.2. Description formelle du problème

Plus formellement[23], on se donne donc :

- un ensemble P = {p1..pM} de professeurs
- un ensemble C = {c1..cN} de classes
- un ensemble L = {l1..lK} de cours
- un ensemble J = {j1..j5} de jours
- un ensemble H = {h1..h4} de créneaux horaires journaliers
- une fonction FC : L -> C qui associe à chaque cours la classe concernée
- une fonction FP : L -> P qui associe à chaque cours le professeur concerné
- une fonction E, qui pour une solution S calcule son évaluation E(S) définie comme suit : $E(S) = \sum_{p \in P} (f(p) + g(p)) + \sum_{c \in C} \frac{f(c)}{2}$, où f(p) est le nombre de demi-journées libres du professeur p, g(p) est le nombre de journées libres du professeur p et f(c) est le nombre de demi-journées libres de la classe c.

Le problème est alors de trouver une fonction S : Q -> J^H telle que :

---

[21] A titre d'exemple, le jeu de réflexion, qui fut la première application historique, nécessite plusieurs pages pour décrire ses règles et plusieurs heures de jeu pour en saisir la complexité. Plusieurs milliers de lignes de connaissance ont été données au système pour résoudre ce problème et il serait trop long de les décrire ici.

[22] Cette fonction d'évaluation est, une fois encore, uniquement à but pédagogique et il ne faut pas y chercher un sens quelconque.

[23] La lecture de la description mathématique du problème n'est pas indispensable pour la suite.

- $\forall(i,j)\in Q^2$, $(FC(i)=FC(j) \vee FP(i)=FP(j)) \Rightarrow S(i) \neq S(j)$
- $\forall(i,j)\in Q^2$, $j\in J$, $\forall(a,b)\in H^2$ tels que $(j,a) = S(i)$ et $(j,b) = S(j)$, si $(a=h1 \wedge b=h2) \vee (a=h2 \wedge b=h1) \vee (a=h3 \wedge b=h4) \vee (a=h4 \wedge b=h3)$ alors $FC(i) \neq FC(j) \vee FP(i) \neq FP(j)$
- E(S) soit maximale

2.3. Un exemple d'instance de problème et de solution

Considérons maintenant l'instance de problème très simple suivante :
- 3 classes c1, c2 et c3.
- 2 professeurs, p1 et p2.
- 7 cours : {l1=(p1,c1), l2=(p1,c2), l3=(p1,c2), l4=(p1,c3), l5=(p2,c1), l6=(p2,c2), l7=(p2,c2)}

Une solution possible est donnée sur la Figure 7. La valeur de cette solution est de 37 et on démontrerait facilement que cette valeur est optimale dans ce cas.

Emploi du temps du professeur p1				
Lundi	Mardi	Mercredi	Jeudi	Vendredi
c2 (l2)				
c1 (l1)				
c2 (l3)				
c3 (l4)				

Emploi du temps du professeur p2				
Lundi	Mardi	Mercredi	Jeudi	Vendredi
c1 (l5)				
c2 (l6)				
c3 (l7)				

Figure 7 : *Exemple de solution*

Dans la suite de ce papier, nous décrirons comment le système MARECHAL fonctionne pour résoudre un tel problème, quelle est son architecture, quelles sont les connaissances dont il a besoin et comment sont elles définies. Au fur et à mesure de cette présentation, nous donnerons certaines des connaissances qu'il est possible de lui fournir pour résoudre ce problème. La liste complète des connaissances (correspondant à une méthode de résolution possible qui n'est pas forcément la meilleure) se trouve en annexe.

3. Vue générale de l'architecture du système de base

Le système MARECHAL se décompose en deux parties : la partie « basique » dont le but est de générer des solutions et la partie « méta » qui contrôle la précédente et supervise le processus de résolution afin d'obtenir la meilleure solution possible, notamment grâce à un mécanisme d'amélioration. Dans ce papier, nous ne décrirons que la partie basique. Des éléments sur la seconde partie peuvent être trouvés dans [Pannérec 01]. Dans cette partie, nous allons donner une vision globale de la partie « basique » du système et chaque composante sera ensuite détaillée dans les parties suivantes.

L'architecture de la partie « basique » est organisée en trois couches (cf. Figure 8) dont les deux plus basses sont très classiques. La première est une couche de perception permettant de dialoguer avec le monde extérieur (dans notre cas, l'application qui gère des emplois du temps). Elle contient avant tout un ensemble de services utilisables par les couches supérieures pour interroger le monde extérieur. Il s'agira en fait d' accéder à l'énoncé du problème, comme par exemple le nombre de classes ou le professeur associé au $i^{ème}$ cours. Cette couche contient également des fonctions de pré-traitement de ces informations (par exemple, pour compter le nombre de cours pour un professeur donné etc.).

La seconde couche est la couche inférence qui permet d'effectuer les déductions et les décisions, en d'autres termes, de générer des nouvelles données. Elle est principalement centrée sur un moteur d'inférence qui utilise des règles de production écrites dans un langage proche de la logique des prédicats. Dans la plupart des problèmes, la chaîne d'inférences peut être très longue et conduire à une explosion combinatoire si ces règles ne sont pas organisées, ce qui a motivé une vision distribuée de l'ensemble des règles. Les règles ayant le même objectif, concourant à la même opération ou devant être activées au même moment sont ainsi regroupées en bases autonomes. La couche inférence peut également contenir quelques procédures écrites

directement en C++ pour implémenter des raisonnements spécialisés (recherche d'un chemin par descente de gradient, anticipations à court terme etc.) qui ne peuvent être définies simplement par une base de règles.

La troisième couche est la couche de résolution/construction qui permet de contrôler la résolution du problème, c'est-à-dire d'appeler les bases de règles à exécuter et effectuer les choix qui composeront la solution finale. Elle utilise pour cela un ensemble de schémas de résolution que l'on peut voir en première approximation comme des plans de résolution.

Le système dispose enfin d'une mémoire principale sous la forme d'une base de faits qui constitue une zone de données globale. Chaque fait possède une valeur de vérité comprise entre 0 (totalement faux) et 100 (totalement vrai) qui peut avoir des sémantiques différentes. La base de faits permet d'enregistrer les informations variées dont a besoin le système au cours de son raisonnement, comme par exemple « le cours n°8 est placé le Mardi à 10h00 » ou « p2 est un professeur » ou « le professeur p2 a 4 demi-journées libres » ou encore, « le mardi est une journée intéressante (70%) pour placer les cours restant du professeur p2 ». L'interfaçage de la base de faits avec les autres composants du système se fait via un module de gestion mémoire.

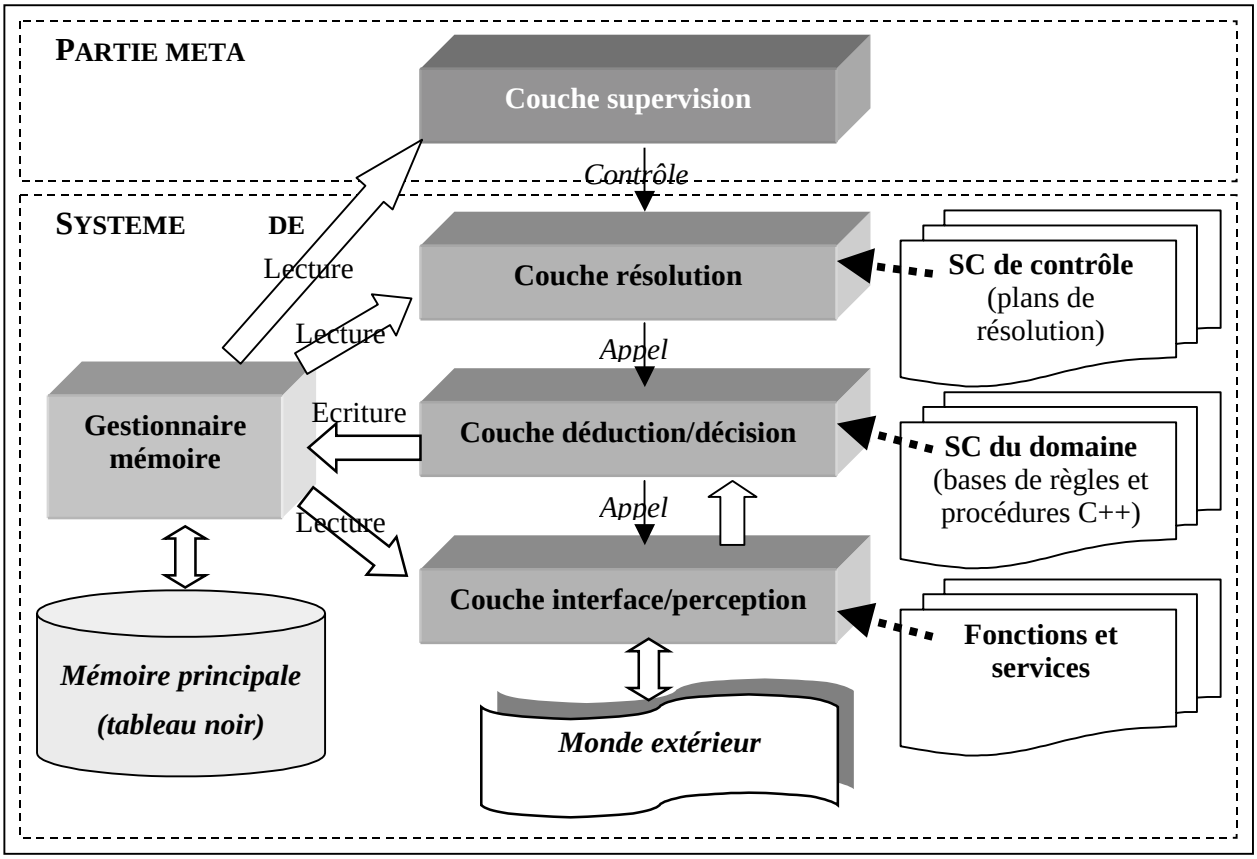


Figure 8 : *Architecture globale du système MARECHAL*

On peut remarquer que l'architecture du système MARECHAL s'apparente largement à un modèle de tableau noir. Ce modèle a été introduit dans les années soixante-dix [Nii 86] [Corkill 91] et a connut ensuite de nombreux développements [Engelmore & Morgan 88] [Jagannathan et al. 89]. Il est issu de la constatation suivante : Pour résoudre des problèmes, un système d'Intelligence Artificielle a souvent besoin de nombreuses connaissances. Lorsque ces problèmes sont complexes, le nombre de connaissances devient tel que l'utilisation d'une seule source globale de connaissances est impossible. Les systèmes ont alors intérêt à adopter une architecture distribuée dans laquelle plusieurs sources de connaissances (SC) spécialisées coopèrent pour résoudre le problème [Davis 80]. Ces SC peuvent être construites de façon indépendante (principe de modularité) et être de nature différente (procédures, moteur d'inférence en chaînage avant ou arrière etc.). Dans le modèle du tableau noir, les SC communiquent via une zone de données globale et partagée appelée « tableau noir » et permettant d'enregistrer les éléments de solution appelés hypothèses. Dans le système MARECHAL, la base de fait peut être vue comme un tableau noir, les bases de règles et procédures de la couche inférence comme des sources de connaissances du domaine et les schémas de résolution comme des sources de connaissance de contrôle.

Le système MARECHAL fonctionnant à partir de connaissances données déclarativement, le langage qui permet de décrire ces connaissances est un élément essentiel que nous allons décrire en parallèle avec le système dans les parties suivantes. En effet, ce langage contient lui-même trois niveaux pouvant être associés aux trois couches précédemment définies, comme le montre la Figure 8. Le premier niveau est celui des fonctions et services définis dans la couche perception. Le second niveau est celui des règles d'inférences qui seront utilisées dans la couche inférence. Le troisième niveau est celui des schémas de résolution qui apparaissent dans la couche résolution. Chaque niveau est défini dans un fichier séparé. Une expertise complète pour un problème donné contient donc trois fichiers : langage.txt, inférence.txt et resolution.txt (cf. Annexes). Les connaissances sont exprimées grâce à un vocabulaire et un ensemble de règles grammaticales. On distingue le vocabulaire inférieur qui est essentiellement dépendant du domaine et est défini dans le fichier langage.txt, du vocabulaire moyen contenant des commandes et prédicats indépendants du domaine et du vocabulaire supérieur permettant de définir des structures comme les règles. Pour définir les mots des vocabulaires, la convention suivante a été utilisée : les mots indépendants du domaine sont en majuscule et les mots dépendant du domaine en minuscule.

4. La couche perception et le vocabulaire inférieur

Dans un système intelligent, la première capacité indispensable est de pouvoir percevoir le monde dans lequel il évolue. Pour le système MARECHAL, ce monde est le problème qu'il doit résoudre. La couche perception permet donc d'assurer la communication avec l'application qui a appelé le système MARECHAL pour une résolution et qui contient l'énoncé du problème à résoudre.

Pour pouvoir utiliser cet interfaçage dans les connaissances, on définit un vocabulaire inférieur pour chaque famille de problème dans un fichier langage.txt. Il s'agit d'une liste de mots auxquels sont associés des arités (nombre d'arguments). On aura par exemple :

```
NbProfesseur      0. // Retourne le nombre de professeur[24]
ClasseDuCours     1. // Retourne la classe associé a un numéro de cours
NbCoursProf       1. // Calcule le nombre total de cours pour un prof
…
```

On peut distinguer quatre catégories de mots inférieurs : les services, les ensembles, les termes et les fonctions.

4.1. Les services

Les services sont des fonctions, au sens informatique du terme, qui permettent d'interroger l'application hôte sur l'énoncé du problème à résoudre. Pour les emplois du temps, on aura simplement :

- NbClasse : Retourne le nombre de classes (N)
- NbProf : Retourne le nombre de professeurs (M)
- NbCours : Retourne le nombre de cours (K)
- ClasseDuCours(k) : Retourne la classe pour le kième cours (FC(k))
- ProfDuCours(k) : Retourne le professeur pour le kième cours (FP(k))

Pour chaque service, la couche perception contient une fonction C++ associée qui interroge l'application hôte et retourne la valeur demandée. Aucune information en dehors du nom et du nombre d'argument n'est nécessaire pour définir un service.

Pour chaque application, un service spécial doit être défini pour envoyer la solution construite par le système à l'application hôte. C'est la seule partie de la couche perception qui écrit des données au lieu de les lire. Pour les emplois du temps, nous aurons donc un service « EnvoiSolution ».

4.2. Les ensembles

Les ensembles permettent de définir des types d'objets pour le problème à traiter. Dans l'exemple des emplois du temps, nous aurons ainsi les ensembles « Classe », « Professeur », « Cours », « Jour » et « Creneau ». Chaque ensemble prend en argument un ou plusieurs entiers permettant de définir un objet.

[24] Comme en C, la séquence « // » dans les fichiers de connaissances indique un commentaire et la fin de la ligne est ignorée par le système.

Ainsi, « Jour(2) » devient une constante correspondant à mardi. Dans d'autres applications, on pourra utiliser plusieurs arguments et écrire « Point(4 5) » pour représenter une coordonnée. Il existe un petit nombre d'ensembles prédéfinis (cf. Annexes).

Quand c'est possible, il est souhaitable de définir pour chaque argument de chaque ensemble la plage de valeur possible. En faisant appel à un service, on aura par exemple « Classe = [1 NbClasse] ». Pour les ensembles à un seul argument, le système crée un fait de la forme « Ensemble(x) » pour chaque valeur de x possible. Il créera ainsi par exemple les faits Prof(1), Prof(2), …, Prof(M). Ces faits seront très utiles pour instancier des variables dans les règles (cf. 5).

4.3. Les termes

Les termes sont des mots inférieurs permettant de construire des faits dans la mémoire. Pour les emplois du temps, nous aurons besoin des termes « Affectation », avec trois arguments, et « Interet », avec un argument.

Ces termes nous permettrons de construire les exemples de faits suivants :

- Affectation(Cours(3) Jour(2) Creneau(3)) : Elément de solution.
- Interet(Affectation(Cours(3) Jour(2) Creneau(3))) : Intérêt d'un choix

4.4. Les fonctions

A partir des éléments définis précédemment, les fonctions permettent de calculer des valeurs. Un certain nombre de fonctions sont prédéfinies dans le système, comme les fonctions arithmétiques usuelles (+, *, - ABSOLU, COS) ou comme les fonctions suivantes que nous utiliserons pour les emplois du temps:

- POSITIF(v) : retourne 1 si v>0 et 0 sinon.
- NUL(v) : retourne 1 si v=0 et 0 sinon.
- TERNAIRE(c v1 v2) : si c est non nul, retourne la valeur de v1, sinon retourne celle de v2

Les fonctions spécifiques à un domaine donné sont définies, quant à elles, par des expressions fonctionnelles. Les arguments sont représentés par des variables.

On pourra ainsi définir :

```
CreneauFrere($t)   // Retourne le créneau qui est dans la même demi-journée que $t
    = TERNAIRE(POSITIF(-(2 $t)) –(3 $t) –(7 $t))
```

Les autres fonctions spécifiques nécessaires à la résolution des emplois du temps sont les suivantes (leur définition peut être trouvée en annexes) :

- NbCoursProf($p) : Calcule le nombre total de cours pour le prof $p
- NbCoursClasse($c) : Calcule le nombre total de cours pour la classe $c
- NbCoursPlacesProf($p) : Calcule le nombre de cours placés pour le prof $p
- NbCoursPlacesProfJour($p $j): Calcule le nombre de cours placés dans la journée $j pour le prof $p
- NbCoursPlacesClasse($c) : Calcule le nombre de cours placés pour la classe $c
- ExisteCoursPourClasse($c $j $t) : Retourne vrai si la classe $c a actuellement un cours le jour $j au créneau $t
- ExisteCoursPourProf($p $j $t) : Retourne vrai si le prof $p a actuellement un cours le jour $j au créneau $t
- ProfPourClasse($c $j $t) : Retourne, si un cours existe pour la classe $c le jour $j et le créneau $t, le prof associé à ce cours (0 sinon)
- ExisteCoursMemeDJ($c $j $t $p): Retourne vrai si la classe $c a actuellement un cours avec le prof $p dans la même demi-journée du jour $j que le créneau $t
- ValSolution() : Retourne la valeur de la solution courante

Pour évaluer ces fonctions, le système dispose d'un interpréteur. Naturellement, l'interprétation est plus coûteuse en temps que l'exécution d'une fonction compilée. C'est pourquoi le système dispose également d'une version C des fonctions. La compilation automatique des fonctions données déclarativement vers du code C ne poserait pas de problème, mais elle n'a pas été réalisée pour le moment.

5. La couche inférence et les règles

Si la couche perception permet principalement de lire des données au besoin prétraitées, la couche inférence a, quant à elle, le rôle de produire de nouvelles données. En effet, la résolution d'un problème passe nécessairement par un ensemble de déductions qui permet d'arriver à un état final (la solution) à partir d'un état de départ (l'énoncé). Mais si, in fine, seule la solution trouvée pour le problème est intéressante, le système, comme les être humains, déduira en général de nombreuses informations intermédiaires et un grand nombre de règles sont souvent nécessaires. La couche inférence est donc centrée sur un moteur d'inférence qui utilise des règles déclaratives. Ces règles travaillent sur des faits stockés dans la mémoire du système.

5.1. La base de faits

La mémoire du système est une base de faits. Comme le système peut avoir à manipuler des informations plus ou moins vraies, certaines ou probables, les faits sont souvent flous et leur valeur de vérité enregistrée dans la base de faits est comprise entre 0 et 100[25].

La base de faits permet également d'enregistrer des données dont la valeur est entière. Par exemple, on pourrait décider d'enregistrer en permanence les nombres de cours placés pour chaque professeur, au lieu d'utiliser une fonction. On pourrait alors demander à la base de fait la valeur de « NbCoursPlacesProf(Prof(2)) » au lieu d'appeler « NbCoursPlacesProf(2) ». Cette possibilité est très intéressante lorsque la valeur est lue un grand nombre de fois car cela évite de la recalculer en permanence comme le ferait une fonction. Ces données sont appelées des données « internalisées », car elles peuvent servir également à transférer en mémoire l'énoncé du problème au début de la résolution, comme par exemple les coordonnées des pièces dans un jeu de stratégie. Les données internalisées peuvent s'utiliser comme des fonctions.

Pour pouvoir faire des requêtes sur la base de faits, on utilisera des propositions, c'est-à-dire des faits dont une ou plusieurs sous-branches sont remplacées par des variables (lettre précédée du symbole « $ » ou « ? »). Les exemples suivants sont des propositions :

- Affectation(Cours($c) Jour(2) Creneau(3))
- Interet($a)

5.2. Les règles

Les règles sont données pour chaque application dans le fichier « Inference.txt ». Elles sont constituées de trois parties : une condition, une liaison et une conclusion[26]. Comme les faits ont des valeurs de vérité floues, les règles sont elles-mêmes floues dans le sens où elles peuvent plus ou moins conclure sur un fait. Les conclusions des règles portant sur le même fait sont agrégées grâce à des formules qui ne seront pas décrites ici.

```
// Règle 1 : on privilégie les profs avec peu d'heures restant à placer
(QUELQUESOIT[Prof($p)]
    SOIT[$g NbCoursPlacesProf($p)]
    SOIT[$h –(NbCoursProf($p) $g)]
    SUPERIEUR[$h 0]
    FAIBLER[$h 0 20 DIMINUTION 100]
)BON <100> FAIRE AJOUTER[Interet(Prof($p))]
```

Figure 9 : *Exemple de règle*

Un exemple de règle est donné dans la Figure 9. Cette règle permet d'évaluer l'intérêt de choisir un professeur pour placer ses cours. La condition commence par considérer tous les professeurs $p, puis stocke dans $g et $h le nombre de cours déjà placés pour ce professeur et le nombre de cours restant à placer. On élimine ensuite les professeurs pour lesquels $h est nul puis on indique que plus $h est élevé, plus l'intérêt doit être grand.

[25] Les faits non flous (booléens) n'ont que deux valeurs de vérité possibles : 0 et 100.
[26] Voir annexe 0 pour une description plus précise.

5.3. Le rôle des règles

Les règles servent à produire des nouvelles informations qui seront stockées dans la base de faits. Mais selon l'utilisation de ces informations, on peut distinguer deux catégories de règle : les règles de production proprement dites et les règles d'évaluation des choix possibles. En ce qui concerne les premières, elles permettent d'ajouter des faits qui serviront dans d'autres règles, c'est-à-dire, en d'autres termes, à définir un « concept ». Par exemple, on peut définir le concept de « ProblemeInsoluble » ou de « EstDeBonneHumeur » pour un professeur grâce aux règles de la Figure 10 :

```
// Règle 1 : test si le pb est insoluble du point de vue d'un professeur
QUELQUESOIT[Prof($p)]
    SOIT[$h NbCoursProf($p)]
    SUPERIEUR[$h 20]
)NECESSAIRE FAIRE AJOUTER[ProblemeInsoluble]

// Règle 2 : Evalue l'humeur d'un professeur
QUELQUESOIT[Prof($p)]
    NON PRESENT[EstDepressif($p)] // depressif => bonne humeur imposs.
    FAIBLER[NbCoursProf($p) 0 20 DIMINUTION 50] // peu d'heure de cours
    FAIBLER[Deplacement($p) 0 100 DIMINUTION 50]// dist. Travail/
                                                // domicile faible
)BON <100> FAIRE AJOUTER[EstDeBonneHumeur(Prof($p))]
```

Figure 10 : *Exemples de règle de production proprement dite*

Les règles d'évaluation des choix possibles sont d'une importance centrale dans le processus de construction d'une solution car elles permettent de guider de façon heuristique la sélection des choix. Contrairement aux règles précédentes, dont le déclenchement est obligatoire, ces règles peuvent être en théorie ignorées[27] par le système et on parlera donc de « conseils ». A chaque fois que le système doit faire des choix, de tels conseils lui permettent d'évaluer *a priori* les différentes options. La règle de la Figure 9 est un exemple de conseil. En général, il faut plusieurs conseils pour évaluer une possibilité, certain concluant positivement et d'autre négativement. Le système agrège alors ces valeurs selon des formules classiques de logique floue.

5.4. Les bases de règles

Afin de simplifier l'appel des règles et d'éviter les tests de déclenchement inutiles, les règles sont regroupées en base de règles, selon leur rôle. Ainsi, tous les conseils pour l'évaluation des possibilités du choix d'un prof à considérer formeront une base « CalculInteretProf » et toutes les règles de production devant être déclenchées au début de la résolution seront regroupées dans une base « PreTraitement ». Chaque base agit comme une source de connaissance capable de réaliser une étape très précise du processus de résolution. Au moment de l'appel d'une base, certaines variables[28] peuvent être pré-instanciées pour configurer la tache à exécuter. Par exemple, la base « CalculInteretJour », qui permet d'évaluer les jours intéressants à étudier pour un professeur donné, prendra en paramètre le numéro du professeur à étudier dans une variable « $p ».

6. La couche résolution et les schémas

Pour construire une solution, il faut appeler les SC de la couche inférence et faire des choix d'après les intérêts calculés. Ce rôle est dévolu à la couche de résolution qui constitue le cœur du processus de construction des solutions. L'utilisation d'un tel niveau fait sortir le système MARECHAL du cadre classique des systèmes experts centrés sur un moteur d'inférence. Pour autant, le problème qui se pose dans cette couche, comme dans toute architecture à base de tableau noir, est celui de la coordination des SC du domaine. Dans cette section, nous faisons donc tout d'abord un rapide historique des diverses formes de contrôle utilisées dans les systèmes à tableau noir. Nous présentons ensuite la solution retenue dans le système MARECHAL, a savoir l'utilisation de schémas de résolution permettant de décomposer le

[27] En pratique, ces règles servent également à définir l'ensembles des choix possibles et il faudrait séparer les deux activités pour pouvoir se passer réellement de l'évaluation.

[28] On parlera des « paramètres » de la base.

problème et de définir des plans de résolution. La couche résolution peut être ainsi perçue comme un module d'exécution de plans permettant de faire les choix qui constitueront la solution.

6.1. Le contrôle dans le modèle de tableau noir

Dès son apparition, le modèle du tableau noir posa le problème du contrôle des diverses sources de connaissances. Un système à base de tableau noir doit en effet déterminer en permanence quelle SC activer et sur quelles données. Le problème se complique lorsque les SC peuvent être à la fois coopératives ou concurrentes. Dans les premiers systèmes, comme HearSay-II [Erman et al. 80] [Lesser et al. 77], le contrôle était implémenté de façon procédurale. Ce type de contrôle était difficile à maintenir et insuffisant pour décrire des stratégies complexes. Il devint donc nécessaire de fournir aux systèmes à base de tableau noir des connaissances de contrôle explicites formant des SC de contrôle par opposition aux SC du domaine. Plusieurs possibilités ont été explorées. Le principe initial était l'opportunisme : chaque SC s'activait « d'elle-même » lorsqu'elle pensait pouvoir entraîner une progression vers la solution. Si cette vision était très souple, elle était parfois assez inefficace et difficile à mettre en place lorsqu'on pouvait déterminer a priori des chaînes d'activations figées pour le problème. Le principe de contrôle hiérarchique (que nous appellerons d'une manière générale « contrôle a priori ») fut donc introduit dans des systèmes comme HASP/SIAP [Nii et al. 82] et CRYSALIS [Terry 83], où les SC sont organisées en une hiérarchie à plusieurs niveaux. Lorsqu'une SC de contrôle est activée, elle choisit une séquence d'activations de SC du niveau inférieur. Ce type de contrôle pose le problème inverse du précédent : il est inadapté lorsque le flux de contrôle ne peut être déterminé a priori. Pour pallier cela, des systèmes hybrides, comme le système ATOME [Lâasri et al. 88] sont apparus. Ils permettent d'allier à la fois la souplesse d'un contrôle opportuniste avec l'efficacité d'un contrôle hiérarchique. C'est dans cette logique que se positionne le système MARECHAL.

6.2. Le contrôle de la construction dans le système MARECHAL

Pour contrôler la construction des solutions, notre système dispose de plans de résolution, appelés schémas de résolution, qui décrivent les différentes étapes d'une résolution. Ces schémas sont fournis au système dans le fichier « Resolution.txt ». Ces sources de connaissances « de contrôle » permettent d'une part, de coordonner l'activité des sources de connaissances « du domaine » de la couche inférence et, d'autre part, de définir les choix à faire pour construire la solution.

6.2.1. Décomposition du problème

Lorsqu'un problème est complexe, il est inconcevable de le résoudre directement dans sa globalité. La méthode fondamentale pour diminuer la complexité est de chercher à décomposer le problème en sous-problèmes pseudo-indépendant. Prenons par exemple le cas d'une personne devant organiser son voyage de Paris à York. Elle conclura tout d'abord qu'elle doit passer par Londres, puis elle considèrera deux sous-problèmes : aller de Paris à Londres et aller de Londres à York. Le système MARECHAL fonctionne de cette manière et dispose de plusieurs schémas de résolutions, chacun étant associé à un sous-problème. Les schémas peuvent alors activer d'autres schémas et forment plus ou moins une hiérarchie de plans, le but étant de découper le processus de résolution en différentes étapes. Le système possède ainsi des méthodes de décomposition du problème mais la décomposition exacte dépend de l'instance du problème (on peut ainsi parler de décomposition dynamique à partir d'une pré-décomposition statique). Pour le problème des emplois du temps, nous avons choisi[29] la pré-décomposition de la Figure 11. Chaque sous-problème est paramétrable, comme une procédure informatique, et peut être appelé plusieurs fois avec des valeurs de paramètre différentes. Par exemple, on appellera le sous-problème « PlacerProf » pour chaque professeur existant. On ne connaît alors pas a priori le nombre d'appel ni l'ordre dans lequel seront étudiés les professeurs, cet ordre étant déterminé au moment de la résolution (cf.). Pour définir de telles séquences d'appel, les schémas de résolution contiennent des structures itératives (mots clés « TANTQUE … FAIRE » et « QUELQUESOIT ») et des structures conditionnelles (« SI…ALORS…SINON… »). La description détaillée des schémas de résolution est donnée en annexe.

[29] Cela n'est pas forcément la meilleure pré-décomposition, mais celle qui sera la plus illustrative pour la suite.

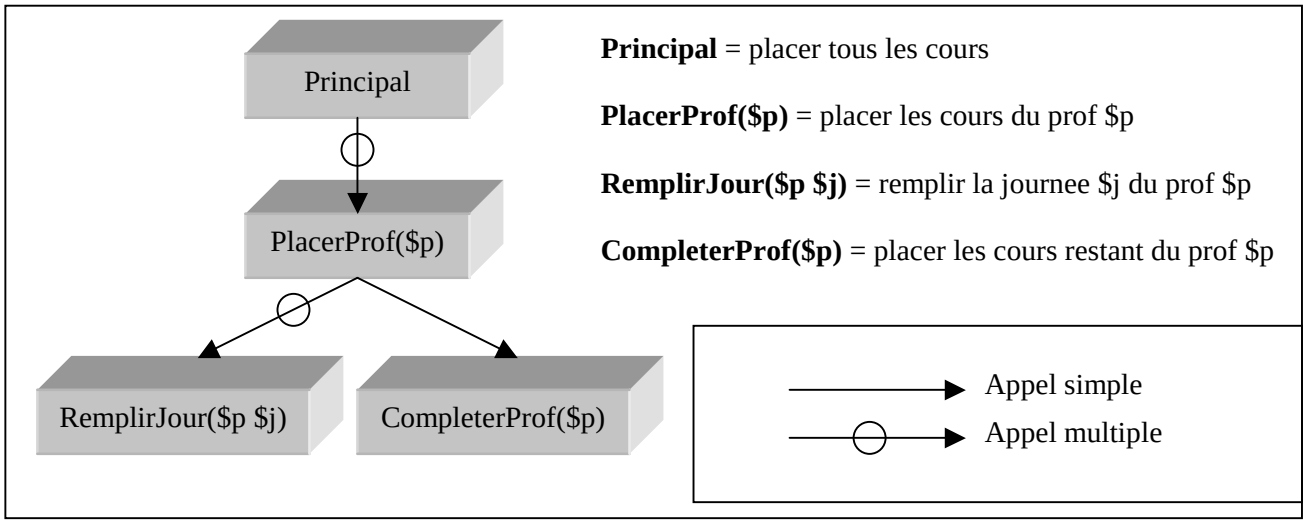


Figure 11 : *Pré-décomposition du problème des emplois du temps*

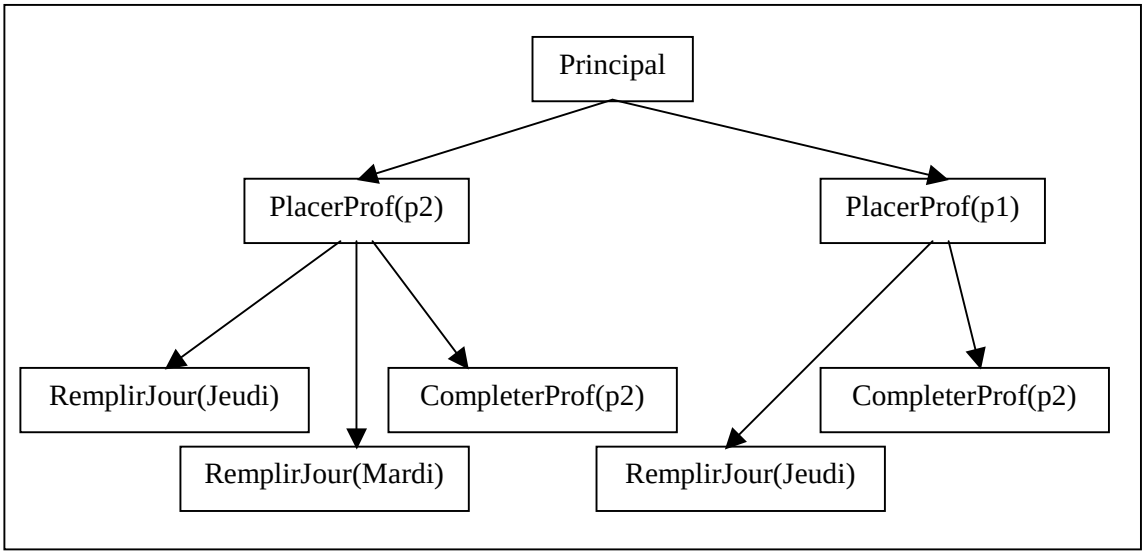


Figure 12 : *Décomposition obtenue en cours de résolution*

D'autre part, plusieurs schémas de résolution peuvent en théorie être définis pour la réalisation d'une même tâche. Un sous-problème peut en effet être résolu de différentes façons selon la situation et les schémas sont alors concurrents (alors que deux schémas résolvant des parties distinctes d'un problème sont coopératifs). Cette possibilité étant actuellement assez marginale, nous ne la décrirons pas dans ce manuscrit et nous renvoyons le lecteur intéressé à [Pannérec 00][30] où elle est décrite en détail.

6.2.2. Les choix concrets et abstraits

Le but des schémas de résolution est de construire une solution au problème posé, c'est-à-dire de déterminer les choix, appelés « choix concrets », constituant cette solution. Mais en fait, face à l'incroyable combinatoire que présente la plupart des problèmes de décision complexe, il n'est en général pas possible de travailler uniquement à ce niveau de résolution. Nous avons vu dans le paragraphe précédent qu'il était nécessaire de décomposer le problème en sous-problèmes et le système doit alors faire des choix sur les sous-problème à considérer (par exemple, choisir le prochain professeur à étudier). Il peut être aussi intéressant de faire des choix stratégiques à priori qui limiteront le nombre de possibilité. Par exemple, dans un jeu de stratégie on pourra se demander tout d'abord s'il faut plutôt attaquer ou défendre. Dans le premier cas, on pourra ensuite réfléchir à la zone la plus prometteuse pour une offensive. Il n'y aura plus alors qu'à

[30] Nous avons en particulier montré dans ce papier que le système MARECHAL disposait alors des quatre opérateurs correspondant aux quatre possibilités de combinaison entre un contrôle de type opportuniste/a priori et un objet d'étude qui pouvait être soit des données soit une méthode de résolution.

considérer les mouvements permettant d'attaquer cette zone et ignorer tous les autres, ce qui restreint énormément les possibilités. Un système d'IA doit donc intégrer de tel choix, d'une part, car cela le rendra plus efficace et, d'autre part, car cela facilitera le transfert d'expertise à partir des êtres humains qui utilisent largement ce principe. Par définition, nous appellerons donc « choix abstrait, un choix qui n'apparaîtra pas dans la solution mais qui conditionne, à un moment donné de la résolution, la génération des choix concrets.

Un choix se fait toujours dans le système grâce à la commande « MEILLEURFAIT(p) » qui recherche, parmi les faits matchant avec la proposition p, celui qui a la plus haute valeur de vérité. Pour donner une priorité à une possibilité de choix, il suffit donc de fixer la valeur de vérité du fait associé, comme nous l'avons vu avec les règles d'évaluation.

6.2.3. Les commandes des schémas de résolution

Les deux principales actions pouvant être effectuées dans un schéma de résolution sont l'appel d'un sous-problème (i.e., d'une SC de contrôle) et l'appel d'une SC du domaine. La première se fait avec la commande « SOUSPROBLEME » et la seconde avec les commandes « ANALYSE » ou « APPEL » selon qu'il s'agit d'une base de règle ou d'une procédure écrite en C.

En théorie, ces trois commandes sont suffisantes. En pratique, il est souvent pénible de devoir écrire une base de règle pour des taches très simples comme la suppression ou l'ajout d'un fait précis dans la base de faits. C'est pourquoi, les commandes suivantes on été rajoutées :

- AJOUTEFAIT(f) : Ajoute le fait f dans la mémoire.
- AJOUTEFAITV(f v) : Ajoute le fait f dans la mémoire avec la valeur v.
- SUPPFAIT(f) : Supprime le faite f de la mémoire
- SUPPFAITTQ(p f) : Supprime parmi les faits en mémoire qui matchent avec la proposition p, ceux pour lesquels l'évaluation de f est non nulle.

6.2.4. Exemples de schémas

Nous ne donnerons ici que trois schémas pour illustrer les différentes possibilités offertes par le langage. L'ensemble des schémas ainsi que les bases de règles utilisées sont données en annexe. On remarquera que chaque schéma fonctionne en général sur le même principe : une boucle « TANTQUE MEILLEURFAIT » permet de faire une succession de choix. Elle est précédée d'un appel à une base de règle pour évaluer a priori l'intérêt des choix.

Schéma « principal » :

```
#implemente sous-probleme Principal[31]
Decomposition:
(   ANALYSE[InteretProf]
    TANTQUE MEILLEURFAIT[Interet(Prof($p))] OPTION(RL(Prof($j)))
    FAIRE
    (   SOUSPROBLEME[PlacerProf]
        ANALYSE[InteretProf]
    )
    APPEL[EnvoiSolution]
)
```

Le schéma principal est celui par lequel commence le système. Ce schéma consiste à appeler pour chaque professeur le schéma « PlacerProf » puis à appeler la procédure d'envoi de la solution construite vers l'application hôte. L'ordre d'étude des professeurs est recalculé après chaque nouveau professeur étudié grâce à la base de règles « InteretProf ». Le choix à chaque étape du prochain professeur à étudier est un choix abstrait.

[31] Les lignes commençant par # dans les fichiers de connaissances constituent des directives de chargement.

Schéma « PlacerProf »

```
#implemente sous-probleme PlacerProf
ParametreS: $p. // prof
Decomposition:
(   ANALYSE[InteretJour]
    TANTQUE MEILLEURFAIT[Interet(Jour($j))] OPTION(RL(Jour($j)))
    FAIRE
    (   SI SUPERIEUR[-(NbCoursProf($p) NbCoursPlacesProf($p))
                -(4 NbCoursPlacesProfJour($p $j))] ALORS
        SOUSPROBLEME[RemplirJour]
    )
    SOUSPROBLEME[CompleterProf]
)
```

Ici, le problème est de placer les cours du professeur $p et la méthode consiste à choisir des jours et les remplir avec les cours de ce professeur. Pour cela, on appel pour chaque jour choisi le sous-problème « RemplirJour ». L'intérêt des jours est calculé a priori. A chaque jour choisi, on teste si le nombre de cours restant à placer est bien supérieur au nombre de créneaux restant pour ce jour, sans quoi le remplissage est forcément impossible. Une fois qu'on a remplit au maximum les journées, on place les cours restant avec le sous-problème « CompléterProf ».

Schéma « RemplirJour »

```
#implemente sous-probleme RemplirJour
ParametreS: $p  // prof
            $j. // jour
Decomposition:
(   ANALYSE[InteretChoix1]
    TANTQUE MEILLEURFAIT[Interet(Affectation(Cours($c) Jour($j) Creneau($h)))]
    FAIRE
    (   AJOUTEFAIT[Affectation(Cours($c) Jour($j) Creneau($h))]
        SUPPFAIT[Interet(Affectation(Cours(?0) Jour($j) Creneau($h)))]
        SOIT[$f CreneauFrere($h)]
        SUPPFAITTQ[Interet(Affectation(Cours(?0) Jour($j) Creneau($f)))
              NUL(-(ClasseCours($c) ClasseCours(?0)))]
    )
)
```

Contrairement aux schémas précédents, on effectue ici des choix concrets pour construire la solution (représentée par l'ensemble des faits « Affectation(Cours($c) Jour($j) Creneau($h))) »). La base « InteretChoix1 » permet de sélectionner et d'évaluer les différentes possibilités. L'instruction « SUPPFAITTQ » permet de supprimer les possibilités non consistantes avec le choix qui vient d'être fait.

6.3. Exemple de construction d'une solution

Nous allons maintenant présenter un exemple de construction de solution à partir de l'expertise décrite précédemment[32] et de l'instance de problème définie en 2. La trace informelle du raisonnement est alors la suivante :

Résolution sous-problème principal
Choix abstrait du professeur p2 (pour lequel il y a moins de cours à placer)
Résolution sous-problème PlacerProf(p2)
Aucun choix de jours car seulement 3 cours à placer
Résolution sous-problème CompleterProf(p2)
Choix concret de mettre l5 le lundi à 8h00
Choix concret de mettre l6 le lundi à 10h00 (pour compléter la demi journée entamée)
Choix concret de mettre l7 le lundi à 14h00 (pour compléter la journée entamée)
Fin résolution Sous-problème courant
Fin résolution Sous-problème courant
Choix abstrait du professeur p1 (le seul qui n'a pas tous ses cours de placés)

[32] Voir les annexes pour la description complète de cette expertise.

Résolution sous-problème PlacerProf(p1)
Choix de Lundi par défaut
Résolution sous-problème RemplirJour(Lundi, p1)
Choix concret de mettre l1 le lundi à 10h00 (pour compléter demi-journée de c1)
Choix concret de mettre l2 le lundi à 8h00 (pour compléter la demi-journée entamée)
Choix concret de mettre l4 le lundi à 16h00 (pour compléter la demi-journée entamée)
Choix concret de mettre l3 le lundi à 14h00 (pour compléter la journée entamée)
Fin résolution Sous-problème courant
Résolution sous-problème CompleterProf(p1)
Rien à faire
Fin résolution Sous-problème courant
Fin résolution Sous-problème courant
Envoi de la solution
Fin résolution Sous-problème courant

La solution obtenue est celle de la Figure 7. Le fait qu'elle soit optimale est un coup de chance. En général, les connaissances qui permettent de faire les choix *a priori* ne peuvent être parfaites et conduire aux bons choix *a posteriori*. C'est pourquoi il est nécessaire, à partir du moment où une solution a été obtenue, de remettre en cause les choix qui s'avèrent mauvais *a posteriori* pour tenter d'améliorer la solution. Cette possibilité est réalisée par la partie méta qui se superpose à la partie basique présentée ici. C'est bien sûr cette seconde partie qui fait la force du système MARECHAL. Elle est décrite dans [Pannérec 01].

7. Conclusion

Dans ce papier, nous avons présenté la partie basique du système MARECHAL. Celle-ci permet de générer des solutions grâce notamment à un moteur d'inférence et une couche résolution qui contrôle le déroulement du processus d'inférence et effectue les choix. Pour fonctionner, le système nécessite des connaissances, principalement fonctions, des règles et des schémas de résolution qui donne au système des méthodes de construction et des heuristiques pour effectuer les choix. La partie basique se comporte finalement comme un ensemble d'interpréteurs (de fonctions, de règles, de schémas etc.).

Naturellement, le fonctionnement par interprétation de connaissances déclaratives est intéressant parce que les connaissances sont plus simples à fournir. Mais il est naturellement moins efficace que ne le serait un fonctionnement en mode compilé. L'une des améliorations possibles concernant cette partie consisterait donc à compiler automatiquement les connaissances déclaratives en code C, comme le font de nombreux systèmes généraux d'IA.

8. Bibliographie

[Corkill 91] Corkill D. D. : *Blackboard Systems* – AI Expert 6 (9), p. 40-47, 1991.

[Davis 80] Davis R. : *Report on the Workshop on Distributed Artificial Intelligence* - SIGART Newsletter, n°73, p. 42-52, 1980.

[Engelmore & Morgan 88] Engelmore R. et Morgan T. : *Blackboard Systems* – Addisson-Wesley, Reading, Massachussets, 1988.

[Erman et al. 80] Erman L. D., Hayes-Roth F., Lesser V. R. et Reddy D. R. : *The HearSay-II Speech-Understanding System : Integrating Knowledge to Resolve Uncertainty* - ACM computing Surveys 12 (2), p. 213-53, 1980.

[Howarth 98] Howarth R.J. : *Interpreting a dynamic and uncertain world : task-based control*. Artificial Intelligence vol. 100 n°1-2, p 5-65, Avril 1998.

[Jagannathan et al. 89] Jagannathan V., Dodhiawala R. et Baum L. S. : *Blackboard Architectures and Applications* – Academic Press, New-York, 1989.

[Lâasri et al. 88] Lâasri H., Maître B., Mondot T., Chapillet F. et Haton J.P. : *ATOME : A Blackboard Architecture with Temporal and Hypothetical Reasoning* – Proceedings of the 8th ECAI, Munich, p. 1-5, 1988.

[Lesser et al. 77] Lesser V. R. et Erman L. D. : *A Retrospective view of the HearSay-II Architecture*. In Proceedings of IJCAI-77, p790-800, 1977.

[Maes & Nardi 88] Maes P. et Nardi D. : *Meta-level architectures and reflection*. North-Holland.

[Nii et al. 82] Nii H. P., Feigenbaum E. A. Anton J. J. et Rockmore A. J. : *Signal-to-Symbol Transformation : HASP/SIAP Case Study* – AI Magazine 3 (2), p. 23-35, AAAI Press, 1982.

[Nii 86] Nii H. P. : *Blackboard Systems : The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures* – AI Magazine 7 (2), AAAI press, 1986.

[Pannérec 00] Pannérec T. : *Gestion implicite de la concurrence dans un système à base de tableau noir*. Actes du colloque Intelligence Artificielle de Berder, pages 42-53, rapport interne LIP6 n°2000/002, 2000.

[Pannerec 01] Pannérec T. : *Using Meta-level Knowledge to Improve Solutions in Coordination Problems*, Research and Development in Intelligent Systems XVIII, Proceedings of the 21st SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence, Springer, Cambridge, December 2001.

[Pitrat 90] Pitrat J. : *Métaconnaissance, futur de l'intelligence artificielle*. Hermès, 1990.

[Pitrat 91] Pitrat J. : *An intelligent system must and can observe his own behavior*. Cognitiva 90, pages 119-128, Elsevier Science Publishers, 1991.

[Terry 83] Terry A. : *The CRYSALIS Project : Hierarchical Control of Production Systems* – Tech. Report HPP-83-19, Université de Standford, Califonie, 1983.

9. Annexes

9.1. Annexes I : Le vocabulaire

Dans cette annexe, nous donnons l'ensemble des mots prédéfinis des trois vocabulaires. Naturellement, cette description est partielle : de nombreux mots, qui ne sont plus utilisés ou le sont très rarement, ont été omis.

9.1.1. Le vocabulaire inférieur

Le vocabulaire inférieur permet de définir les services, les termes, les ensembles et les fonctions. Chaque élément y est défini par une chaîne de caractère et un entier pour le nombre d'arguments.

Pour définir des termes et des services, aucune information supplémentaire n'est nécessaire. Pour les ensembles, il faut préciser le domaine de variation de chaque argument. Pour les fonctions non prédéfinies, il faut associer une expression fonctionnelle qui définit la fonction.

Termes prédéfinis

Quelques termes ont été prédéfinis, notamment en ce qui concerne la gestion des méta-informations.

- ATTENTE : Pour les faits représentant ce à quoi s'attend le système.
- ECHECVERIF : Pour les faits décrivant un échec de vérification.
- ANOMALIE : Utilisée en cas d'attente violée.
- VERROUILLAGE : Pour verrouiller une ressource.
- METAINFO : Pour définir une méta-information sur la résolution effectuée.
- SOUSPBFILS : Pour enregistrer dans la mémoire la hiérarchie des instances de sous-problème étudiées.
- CHOIX : Pour parler d'un choix.
- ROLECHOIX : Pour définir le rôle d'un choix (pourquoi il a été fait).

Les ensembles prédéfinis

Un petit nombre d'ensembles prédéfinis existe :

- PR : Pas de raisonnement (permet d'accéder à la trace du système)
- SR : Segment de raisonnement (permet d'accéder à la trace du système)
- SIT : Situation (pour la gestion de plusieurs solutions)
- POINT : Pour définir une coordonnée dans un espace discret à deux dimensions
- VAL : Type générique

Fonctions prédéfinies

Les fonctions prédéfinies peuvent être classées en trois catégories. La première concerne les fonctions usuelles, comme les fonctions arithmétiques et trigonométriques (+, *, -, /, ABSOLU, MODULO, COS, SIN, ATAN2…) ou comme les fonctions suivantes :

- POSITIF(v) : retourne 1 si v>0 et 0 sinon.
- NUL(v) : retourne 1 si v=0 et 0 sinon.
- TERNAIRE(c v1 v2) : si c est non nul, retourne la valeur de v1, sinon retourne celle de v2
- MIN(a b), MAX(a b) : Retourne la valeur min ou max entre a et b.
- VERITE(f) : Retourne le degré de vérité du fait f.
- NBFAITS(p) : Retourne le nombre de faits qui matchent avec la proposition p.

La seconde catégorie concerne les fonctions qui prennent en paramètre une condition[33] et qui permettent d'effectuer des opérations sur des ensembles de faits, comme par exemple :

- CARDINAL(CONDITION(c)) : Compte le nombre d'instanciation de variables qui vérifie la condition c
- SOMME(CONDITION(c) f) : Retourne la somme des valeurs de f sur les instanciations vérifiant la condition c.

[33] Les conditions utilisées ici s'écrivent de la même façon que celles pour les règles (cf. 0). Il s'agit de la seule exception où du vocabulaire moyen peut se trouver en-dessous de vocabulaire inférieur dans une expression.

- MOYENNE(CONDITION(c) f) : Retourne la moyenne des valeurs de f sur les instanciations vérifiant la condition c.
- VMIN(CONDITION(c) f) : Retourne la valeur minimum de f sur les instanciations vérifiant la condition c.
- VMAX(CONDITION(c) f) : Retourne la valeur maximum de f sur les instanciations vérifiant la condition c.

La dernière catégorie contient les fonctions qui permettent d'accéder à des données internes du système pour permettre la réflexivité. Elles sont utilisées surtout au niveau méta du système.

- PRCHOIXRESP(f) : Retourne le numéro du pas de raisonnement associé au choix qui a provoqué la modification du fait f.
- PRRESPAJOUT(f) : Retourne le numéro du pas de raisonnement qui a provoqué la modification du fait f.
- APPARTIENTSP(a b) : Retourne 1 si le segment ou le pas a appartient au sous-problème de segment b.
- SPCOURANT : Retourne le segment correspondant au sous-problème en cours de résolution.
- SESSIONCOURANTE : Retourne le segment correspondant à la session de résolution courante.
- SPRESPPAS(a) : Retourne le segment du sous-problème responsable du pas de raisonnement a
- CODEERREURSP : Retourne le code de retour du dernier sous-problème dont la résolution s'est terminée.

9.1.2. Le vocabulaire moyen

Le vocabulaire moyen permet de représenter des commandes ou des prédicats indépendants du domaine. Les commandes peuvent se répartir en deux groupes : celles utilisées dans les actions des règles et celles utilisées dans les plans de résolution. Les prédicats peuvent, quant à eux, se classer en trois catégories : les prédicats d'instanciation, les prédicats de filtrage et les prédicats d'évaluation.

Les commandes pour les actions

Les commandes sont utilisées principalement dans les plans de résolution, mais on les trouve aussi dans les parties conclusion des règles où les deux commandes principales sont :

- AJOUTER[f] : permet d'ajouter un fait dans la mémoire
- SUPPRIMER[p] : permet de supprimer tous les faits qui matchent avec la proposition « p »

Les commandes pour les schémas de résolution

- SOUSPROBLEME : Appel d'un sous-problème.
- ANALYSE : Appel du moteur d'inférence sur une base de règles.
- APPEL : Appel d'une procédure écrite en C.
- AJOUTEFAIT(f) : Ajoute le fait f dans la mémoire.
- AJOUTEFAITV(f v) : Ajoute le fait f dans la mémoire avec la valeur v.
- SUPPFAIT(f) : Supprime le fait f de la mémoire
- SUPPFAITTQ(p f) : Supprime parmi les faits en mémoire qui matchent avec la proposition p, ceux pour lesquels l'évaluation de f est non nulle.
- SAUVECONTEXT : Sauvegarde des variables pour préparer un appel récursif de schéma.
- RESTAURECONTEXT : Restaure l'état des variables après un appel récursif de schéma.
- RETOURNE : Permet de définir la valeur retournée par un sous-problème.
- MEILLEURFAIT : Instancie des variables avec le meilleur fait matchant une proposition. C'est en fait un prédicat.
- INSTMATCH : Instancie des variables avec le premier fait trouvé matchant une proposition. C'est en fait un prédicat.
- INSTANCIE(p f) : Instancie les variables de p pour obtenir le fait f.
- DETRUITVAR : Supprime l'instanciation d'une variable.
- BREAKPOINT : Stoppe l'exécution pour débuguer.
- AFFICHEFAIT : Affiche un fait pour débuguer.

Les prédicats d'instanciation

Les prédicats d'instanciation permettent de définir des ensembles de variation pour des variables de façon à ce qu'une condition puisse passer en revue un certain nombre de possibilités. Ces prédicats permettent donc

d ‘affecter des valeurs à des variables en itérant si plusieurs valeurs sont possibles. Naturellement, les variables instanciées le sont pour la suite de la condition et pour la partie conclusion de la règle. Les trois principaux prédicats d’instanciation sont :

- QUELQUESOIT[p] : Permet d’instancier les variables de la proposition p avec les jeux d’instanciation issus du matchage de p avec la base de faits.
- QQUESOITENTIER[$i a b] : Permet de faire varier une variable entière « $i » de la valeur « a » à la valeur « b ».
- SOIT[$i f] : Permet d’affecter la valeur de « f » à la variable « $i ». Il n’y a pas d’itération ici, puisqu’une seule valeur est possible.

Les prédicats de filtrage

Les prédicats de filtrage permettent de refuser les jeux d’instanciation qui ne vérifient pas une certaine condition. Ils peuvent être précédés du mot clé « NON » pour indiquer la négation de la condition. Les principaux prédicats de ce type sont :

- SUPERIEUR[a b], INFERIEUR[a b], EGAL[a b], DIFFERENT[a b] : indique que la valeur de l’expression fonctionnelle « a » doit être supérieure, inférieure, égale ou différente de celle de « b ».
- PRESENT[f] : indique que le fait « f » doit être présent dans la base de faits.
- EGALARBRE[a b] : test l’égalité de deux arbres.

Les prédicats d’évaluation

Comme le système peut avoir à manipuler des informations plus ou moins vraies, certaines ou probables, les faits sont souvent flous et leur valeur de vérité enregistrée dans la base de faits est comprise entre 0 et 100[34]. Les règles doivent donc être capables de gérer ce flou, notamment en contenant des relations telles que « plus p est vraie, plus la conclusion c sera vraie ». Les prédicats d’évaluation autorisent cela en permettant de calculer une valeur floue pour chaque jeu d’instanciation qui vérifie la totalité d’une condition. Les principaux prédicats de ce type sont les suivants :

- ELEVER[v a b t p] : plus « v » est proche de « b » par rapport à « a » (a doit être inférieur à b), plus la condition (et donc la conclusion de la règle) sera vraie. « t » est une constante parmi « AUGMENTATION », « DIMINUTION » et « CENTRAGE » qui permet de choisir une formule d’agrégation et « p » est un poids intervenant dans les formules d’agrégation.
- FAIBLER[v a b t p] : négation de la précédente.
- VRAI[f t p] : plus le fait « f » est vrai, plus la condition est vraie. « t » et « p » ont le même rôle que précédemment.

9.1.3. Le vocabulaire supérieur

- BON, MAUVAIS, NECESSAIRE, IMPOSSIBLE : utilisés dans les règles (cf. 0).
- FAIRE, AVOIR : idem
- NON : négation d’une prémisse
- SI … ALORS … SINON … : structure conditionnelle dans les plans de résolution
- TANTQUE … FAIRE … : structure itérative dans les plans de résolution
- QUELQUESOIT … FAIRE … : structure itérative dans les plans de résolution

9.2. Annexes II : La grammaire du langage

La grammaire du langage est basée sur les principes suivants : le vocabulaire supérieur encadre le vocabulaire par des parenthèses et est récursif. Le vocabulaire moyen n’est pas récursif et encadre le vocabulaire inférieur par des crochets. Finalement, le vocabulaire inférieur est récursif et utilise les parenthèses.

9.2.1. Description informelle de la grammaire du langage

Les atomes

- Les variables sont des lettres précédées du symbole « $ » ou « ? ».
- Les entiers sont des suites de chiffres

[34] Les faits non flous (booléens) n’ont que deux valeurs de vérité possibles : 0 et 100.

Les expressions fonctionnelles

Les expressions fonctionnelles sont des arbres dont chaque nœud est une fonction et chaque feuille une valeur entière, une variable ou une fonction d'arité 0. Chaque nœud doit évidemment avoir autant de fils que l'arité de la fonction de ce nœud. Les expressions fonctionnelles sont écrites selon une notation fonctionnelle préfixée.

Les faits et les propositions

Un fait est un arbre dont chaque nœud correspond soit à un ensemble, soit à une valeur entière, soit à un terme. Les faits s'écrivent selon une notation fonctionnelle préfixée. Les règles grammaticales de formation des faits sont les suivantes :

- Les arguments d'un ensemble sont forcément des valeurs entières
- Un terme ne peut avoir pour argument une valeur entière

Une proposition est un fait dont une ou plusieurs sous-branches sont remplacées par des variables.

Les règles

Les règles sont constituées de trois parties, une condition, une liaison et une conclusion, et s'écrivent « <condition> <liaison> <conclusion> ». La condition est une liste conjonctive de pseudo-prémisses, chaque pseudo-prémisse pouvant être une prémisse ou une liste disjonctive de prémisses. La conclusion est en générale une liste d'actions. Une prémisse ou une action est définie par un mot du vocabulaire moyen plus une expression fonctionnelle ou une proposition pour chaque argument nécessaire. Les liaisons possibles entre une condition et une conclusion sont les suivantes (ou x est un entier compris entre 0 et 100 exprimant le poids de la règle) :

- BON <x> FAIRE[35] : Plus la condition est vraie et plus x est élevé, plus la conclusion est recommandée.
- MAUVAIS <x> FAIRE : Plus la condition est vraie et plus x est élevé, plus la conclusion est déconseillée.
- IMPOSSIBLE FAIRE : Si la condition n'est pas totalement fausse, alors la conclusion ne pourra être effectuée.
- NECESSAIRE FAIRE : Si la condition n'est pas totalement fausse, alors la conclusion sera forcément effectuée.

Les schémas de résolution

Les schémas de résolution consistent en une séquence d'élément, chaque élément étant soit une structure, soit une action (cf. actions des règles en 0 et commandes des schémas en 0). Les structures possibles sont les suivantes :

- SI <condition> ALORS <séquence> : structure conditionnelle
- SI <condition> ALORS <séquence1> SINON <séquence2> : structure conditionnelle
- TANTQUE <commande> FAIRE <plan> : structure itérative utilisée généralement avec la commande MEILLEURFAIT.
- QUELQUESOIT <commande> FAIRE <séquence> : structure itérative utilisée généralement avec la commande INSTMATCH.

9.2.2. Description formelle de la grammaire du langage

Ce paragraphe contient une tentative de description formelle du langage utilisé pour définir des connaissances au système MARECHAL (règles et schémas de résolution). Il ne définit pas la syntaxe complète des fichiers contenant ces connaissances.

Dans la suite, « x* » signifie une liste d'au moins deux éléments « x » séparés par des espaces. « x^n » signifie une liste d'exactement n éléments « x » séparés par des espaces.

<variable> = $<lettre>
<entier> = <chiffre> | <chiffre><entier>

[35] Le mot clé « FAIRE » peut être remplacé dans certaine règle par « AVOIR » et la conclusion est alors une condition. Ces règles spéciales permettent de gérer des contraintes.

<fonc-0> = mot du vocabulaire inférieur correspondant à une fonction d'arité 0
<fonc-n> = mot du vocabulaire inférieur correspondant à une fonction d'arité n > 0
<atome> = <variable> | <entier> | <fonc-0>
<ExpFonctionelle> = <atome> | <fonc-n>(<ExpFonctionelle>^{n})
| CONDITION(<Condition>)
<Fait> = <ensemble-n>(<entier>^{n}) | <terme-n>(<Fait>^{n})
<atome-prop> = <variable> | <entier>
<Proposition> = <ensemble-n>(<atome-prop>^{n})
| <terme-n>(<Proposition>^{n})
<modificateur> = DIMINUTION | AUGMENTATION | CENTRAGE
<Prémisse> = QUELQUESOIT[<Proposition>]
| QQUESOITENTIER[<variable> <ExpFonctionelle> <ExpFonctionelle>]
| SOIT[<variable> <ExpFonctionelle>]
| SUPERIEUR[<ExpFonctionelle> <ExpFonctionelle>]
| INFERIEUR[<ExpFonctionelle> <ExpFonctionelle>]
| EGAL[<ExpFonctionelle> <ExpFonctionelle>]
| DIFFERENT[<ExpFonctionelle> <ExpFonctionelle>]
| PRESENT[<Fait>]
| EGALARBRE[<Fait> <Fait>]
| ELEVER[<ExpFonctionelle> <ExpFonctionelle> <ExpFonctionelle> <modificateur>
<ExpFonctionelle>]
| FAIBLER[<ExpFonctionelle> <ExpFonctionelle> <ExpFonctionelle> <modificateur>
<ExpFonctionelle>]
| VRAI[<Fait> <modificateur> <ExpFonctionelle>]
| MEILLEURFAIT[<Proposition>]
| INSTMATCH[<Proposition>]
<PrePrémisse0> = <Prémisse> | NON <Prémisse>
<PrePrémisse> = <PrePrémisse0> | (<PrePrémisse0>*)
<Condition> = (<PrePrémisse>*) // peut ne contenir qu'un seul élément
<ActionRègle> = AJOUTER[<Proposition>] | SUPPRIMER[<Proposition>]
<Conclusion> = <ActionRègle> | (<ActionRègle>*)
<Liaison> = BON <<entier>> | MAUVAIS <<entier>> | NECESSAIRE | IMPOSSIBLE
<Règle> = <Condition> <Liaison> FAIRE <Conclusion> | <Condition> <Liaison> Avoir <Condition>
<Commande> = SOUSPROBLEME[<nomsouspb>]
| ANALYSE[<nombase>]
| APPEL[<nomprocedure-0>]
| APPEL[<nomprocedure-n>(<ExpFonctionnelle>^{n})]
| AJOUTEFAIT[<Fait>]
| AJOUTEFAITV[<Fait> <ExpFonctionnelle>]
| SUPPFAIT[<Proposition>]
| SUPPFAITTQ[<Proposition> <ExpFonctionnelle>]
| SAUVECONTEXT
| RESTAURECONTEXT
| RETOURNE[<ExpFonctionnelle>]
| INSTANCIE[<Proposition> <Fait>]
| DETRUITVAR[<variable>]
| BREAKPOINT
| AFFICHEFAIT[<Fait>]

<Instruction> = <Commande>
| SI <PrePrémisse0> ALORS <Schéma>
| SI <PrePrémisse0> ALORS <Schéma> SINON <Schéma>
| TANTQUE <PrePrémisse0> FAIRE <Schéma>
| QUELQUESOIT <PrePrémisse0> FAIRE <Schéma>
<Schéma> = <Instruction> | (<Instruction>*)

9.3. Annexe III : Expertise de construction pour les emplois du temps

La première partie de cette section examine les schémas de résolution nécessaires, la seconde présente les bases de règles qui sont utilisées et la troisième, les fonctions qui sont définies. Les connaissances apparaissent ici telles qu'elles sont écrites dans les fichiers fournis au système.

9.3.1. Les schémas de résolution

```
// schéma principal
#implemente sous-probleme Principal
Decomposition:
(   ANALYSE[InteretProf]
    TANTQUE MEILLEURFAIT[Interet(Prof($p))] OPTION(RL(Prof($j)))
    FAIRE
    (  SOUSPROBLEME[PlacerProf]
       ANALYSE[InteretProf]
    )
    APPEL[EnvoiSolution]
)


// Placement des cours du professeur $p
#implemente sous-probleme PlacerProf
ParametreS: $p. // prof
Decomposition:
(   ANALYSE[InteretJour]
    TANTQUE MEILLEURFAIT[Interet(Jour($j))] OPTION(RL(Jour($j)))
    FAIRE
    (  SI SUPERIEUR[-(NbCoursProf($p) NbCoursPlacesProf($p))
              -(4 NbCoursPlacesProfJour($p $j))] ALORS
       SOUSPROBLEME[RemplirJour]
    )
    SOUSPROBLEME[CompleterProf]
)


// Placement des cours du professeur $p dans le jour $j
#implemente sous-probleme RemplirJour
ParametreS: $p  // prof
            $j. // jour
Decomposition:
(   ANALYSE[InteretChoix1]
    TANTQUE MEILLEURFAIT[Interet(Affectation(Cours($c) Jour($j) Creneau($h)))]
    FAIRE
    (  AJOUTEFAIT[Affectation(Cours($c) Jour($j) Creneau($h))]
       SUPPFAIT[Interet(Affectation(Cours(?0) Jour($j) Creneau($h)))]
       SOIT[$f CreneauFrere($h)]
       SUPPFAITTQ[Interet(Affectation(Cours(?0) Jour($j) Creneau($f)))
             NUL(-(ClasseCours($c) ClasseCours(?0)))]
    )
)
```

```
// Placement des cours restant du professeur $p
# implemente sous-probleme CompleterProf
PARAMETRES:       $p. // prof
DECOMPOSITION:
(  ANALYSE[InteretJour]
   ANALYSE[InteretChoix2]
   TANTQUE MEILLEURFAIT[Interet(Affectation(Cours($c) Jour($j) Creneau($h)))]
   FAIRE
   (  AJOUTEFAIT[Affectation(Cours($c) Jour($j) Creneau($h))]
      SUPPFAIT[Interet(Affectation(Cours(?0) Jour($j) Creneau($h)))]
      SOIT[$f CreneauFrere($h)]
      SUPPFAITTQ[Interet(Affectation(Cours(?0) Jour($j) Creneau($f)))
            NUL(-(ClasseCours($c) ClasseCours(?0)))]
   )
)
```

9.3.2. Les bases de règles

```
#implemente base de conseils InteretProf

// Règle 1 : on privilégie les profs avec peu d'heures restant à placer
(QUELQUESOIT[Prof($p)]
    SOIT[$g NbCoursPlacesProf($p)]
    SOIT[$h –(NbCoursProf($p) $g)]
    SUPERIEUR[$h 0]
    FAIBLER[$h 0 20 DIMINUTION 100]
)BON <100> FAIRE AJOUTER[Interet(Prof($p))]


#implemente base de conseils InteretJour
#PARAMETRE $p // prof

// Règle 1 : on privilégie les jours avec peu de créneaux libres
(QUELQUESOIT[Jour($j)]
    SOIT[$g NbCoursPlaceJour($p $j)]
    SOIT[$h –(4 $g)]
    SUPERIEUR[$h 0]
    FAIBLER[$h 0 4 DIMINUTION 80]
)BON <100> FAIRE AJOUTER[Interet(Jour($j))]


#implemente base de conseils InteretChoix1
#PARAMETRE $p // prof
#PARAMETRE $j // jour

// Règle 1 : on privilégie les affectations sur les jours intéressants
(QUELQUESOIT[Cours($c)]
    EGAL[ProfDuCours($c) $p]
    NON PRESENT[Affectation(Cours($c) Jour(?0) Creneau(?1))]
    SOIT[$a ClasseDuCours($c)]
    QUELQUESOIT[Creneau($t)]
        NON PRESENT[Interdit(Affectation(Cours($c) Jour($j) Creneau($t)))]
        EGAL[ExisteCoursPourProf($p $j $t) 0]
        EGAL[ExisteCoursPourClasse($a $j $t) 0]
        EGAL[ExisteCoursMemeDJ($a $j $t $p) 0]
        ELEVER[ExisteCoursPourProf($p $j CreneauFrere($t)) 0 1
           DIMINUTION 30]
        ELEVER[ExisteCoursPourClasse($a $j CreneauFrere($t)) 0 1
           DIMINUTION 50]
)BON <100> FAIRE AJOUTER[Interet(Affectation(Cours($c) Jour($j) Creneau($t)))]

// Règle 2 : on privilégie les choix obligatoires
(QUELQUESOIT[Obligatoire(Affectation(Cours($c) Jour($j) Creneau($t)))]
    EGAL[ProfDuCours($c) $p]
)NECESSAIRE FAIRE AJOUTER[Interet(Affectation(Cours($c) Jour($j) Creneau($t)))]
```

```
#implemente base de conseils InteretChoix2
#PARAMETRE $p // prof

// Règle 1 : on privilégie les affectations sur les jours intéressants
(QUELQUESOIT[Cours($c)]
    EGAL[ProfDuCours($c) $p]
    NON PRESENT[Affectation(Cours($c) Jour(?0) Creneau(?1))]
    SOIT[$a ClasseDuCours($c)]
    QUELQUESOIT[Jour($j)]
        SOIT[$h VERITE([Interet(Jour($j)))]
        SUPERIEUR[$h 0]
        ELEVER[$h 0 100 DIMINUTION 100]
        QUELQUESOIT[Creneau($t)]
          NON PRESENT[Interdit(Affectation(Cours($c) Jour($j) Creneau($t)))]
          EGAL[ExisteCoursPourProf($p $j $t) 0]
          EGAL[ExisteCoursPourClasse($a $j $t) 0]
          EGAL[ExisteCoursMemeDJ($a $j $t $p) 0]
          ELEVER[ExisteCoursPourProf($p $j CreneauFrere($t)) 0 1
            DIMINUTION 30]
          ELEVER[ExisteCoursPourClasse($a $j CreneauFrere($t)) 0 1
            DIMINUTION 30]
          ELEVER[NbCoursJourneeProf($p $j) 0 4 DIMINUTION 30]
)BON <100> FAIRE AJOUTER[Interet(Affectation(Cours($c) Jour($j) Creneau($t)))]

// Règle 2 : on privilégie les choix obligatoires
(QUELQUESOIT[Obligatoire(Affectation(Cours($c) Jour($j) Creneau($t)))]
    EGAL[ProfDuCours($c) $p]
)NECESSAIRE FAIRE AJOUTER[Interet(Affectation(Cours($c) Jour($j) Creneau($t)))]
```

9.3.3. Les fonctions

```
// Retourne le créneau qui est dans la même demi-journée que $t
CreneauFrere($t)
   = TERNAIRE(POSITIF(-(2 $t)) –(3 $t) –(7 $t))


NbCoursProf($p) // Calcule le nombre total de cours pour le prof $p
   = CARDINAL(CONDITION(
      (QUELQUESOIT[Cours($l)]
       EGAL[ProfDuCours($l) $p]
      ))


// Calcule le nombre total de cours pour la classe $c
NbCoursClasse($c)
   = CARDINAL(CONDITION(
      (QUELQUESOIT[Cours($l)]
       EGAL[ClasseDuCours($l) $c]
      ))


// Calcule le nombre de cours placés pour le prof $p
NbCoursPlacesProf($p)
   = CARDINAL(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) $a $b)]
       EGAL[ProfDuCours($l) $p]
      ))


// Calcule le nombre de cours placés dans la journée $j pour le prof $p
NbCoursPlacesProfJour($p $j)
   = CARDINAL(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) Jour($j) $b)]
       EGAL[ProfDuCours($l) $p]
      ))
```

```
// Calcule le nombre de cours placés pour la classe $c
NbCoursPlacesClasse($c)
   = CARDINAL(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) $a $b)]
       EGAL[ClasseDuCours($l) $c]
      ))


// Retourne vrai si classe $c a actuellement un cours le jour $j au créneau $t
ExisteCoursPourClasse($c $j $t)
   = POSITIF(CARDINAL(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) Jour($j) Creneau($t))]
       EGAL[ClasseDuCours($l) $c]
      )))


// Retourne vrai si le prof $p a actuellement un cours le jour $j au créneau $t
ExisteCoursPourProf($p $j $t)
   = POSITIF(CARDINAL(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) Jour($j) Creneau($t))]
       EGAL[ProfDuCours($l) $p]
      )))


// Retourne, si un cours existe pour la classe $c le jour $j et le créneau $t,
// le prof associé à ce cours (0 sinon)
ProfPourClasse($c $j $t)
   = VMAX(CONDITION(
      (QUELQUESOIT[Affectation(Cours($l) Jour($j) Creneau($t))]
       EGAL[ClasseDuCours($l) $c]
      )
      ProfDuCours($l))


// Retourne vrai si la classe $c a actuellement un cours avec le prof $p dans la
// même demi-journée du jour $j que le créneau $t
ExisteCoursMemeDJ($c $j $t $p):
   = NUL(–($p ProfPourClasse($c $j CreneauFrere($t)))
```

DETECTION INCREMENTALE D'ERREUR

Groupe COMBIEN?
Duma J.[36], Giroire H.[37], Le Calvez F.[38], Tisseau G.[37], Urtasun M.[38]

Résumé

Dans un système interactif, le problème de la détection des erreurs commises par l'utilisateur se pose de manière incrémentale, au fur et à mesure de la saisie d'informations. Dans le projet Combien? nous avons défini pour l'apprentissage des dénombrements une quinzaine d'interfaces de même facture. Cela nous a conduit à rechercher une méthode générale de détection des erreurs de l'élève pour ces interfaces. Celles-ci ont pour caractéristique d'éditer une solution sous forme d'une structure arborescente. Nous proposons un modèle de mécanisme de détection incrémentale d'erreurs que nous appelons « détection ciblée », utilisant un appariement dans des arborescences. La notion de cible (nouvel élément introduit qu'on veut analyser) permet de prendre en compte l'aspect interactif. Les connaissances nécessaires sont organisées sous forme de base de schémas d'erreurs. Nous montrons ensuite comment nous avons appliqué ce mécanisme dans le projet Combien?

Mots Clef

EIAH, interfaces pédagogiques, diagnostic, représentation des connaissances, dénombrement.

1. Introduction

Dans un logiciel à but pédagogique, les erreurs commises par l'utilisateur ont un rôle important, parce qu'elles constituent des moments privilégiés d'apprentissage [1]. De nombreuses recherches autour des erreurs (détection et/ou explication) ont été entreprises. Dans les ITS (Intelligent Tutoring System), cette recherche était centrée sur l'amélioration du modèle de l'élève en vue de l'explication [7]. Il semble que ces dernières années, un certain nombre de recherches se soient orientées plus particulièrement sur la modélisation du domaine d'apprentissage pour pouvoir réaliser un inventaire des erreurs le plus exhaustif possible et à partir de là apporter des explications. C'est dans ce cadre que se place le travail que nous allons exposer. Le mécanisme de détection d'erreurs est primordial. Il doit pouvoir faire référence à des connaissances portant sur le domaine d'étude et s'adapter à différentes stratégies pédagogiques.

Nous proposons ici un mécanisme de ce type. Nous en donnons d'abord les principes généraux, fondés sur une approche que nous appelons *détection ciblée*.

Cet article présente d'abord le projet Combien? [6] [9] qui a pour but de créer un ITS de dénombrement. Dans ce logiciel nous avons défini un certain nombre d'interfaces permettant à l'élève de donner sa solution. Nous nous sommes posés le problème de gestion des erreurs dans ces interfaces qui ont pour particularité d'éditer des structures arborescentes. Nous présentons le mécanisme de détection ciblée. Nous montrons ensuite la pertinence de son utilisation pour la détection d'erreurs dans un système interactif. Nous décrivons ensuite comment nous l'avons implémenté dans nos interfaces.

[36] Lycée Jacquard, Paris
[37] LIP6, Pôle IA, SYSDEF, Université Pierre et Marie Curie, Paris VI
[38] CRIP5, SBC, Université René Descartes, ParisV

2. Cadre de l'étude

2.1. Le projet Combien?

Le projet "Combien ?" a pour but de définir une méthodologie de conception de différents composants d'un EIAH (Environnement Interactif d'Apprentissage Humain). Pour valider nos réflexions, nous réalisons un système pédagogique d'aide à l'apprentissage humain dans le domaine mathématiques des dénombrements. Les exercices correspondant sont de la forme : "Etant donnés des ensembles servant de référentiels, compter dans un certain univers les éléments vérifiant des contraintes de sélection". Nous avons défini les fondements mathématiques d'une méthode de résolution (la *méthode constructive*) adaptée aux conceptions usuelles des élèves et permettant d'accéder à la théorie mathématique du domaine [6]. Cette méthode consiste à définir une *construction* d'une *configuration solution*, c'est-à-dire d'un élément de l'ensemble à dénombrer. Cette construction est formée d'*étapes*, chaque étape construisant une partie d'une configuration et permettant de calculer le nombre de possibilités qui lui est associé.

Nous avons défini une classification des problèmes du domaine et les schémas de résolution associés aux différentes classes. Nous avons introduit pour chaque classe une "machine à construire une solution".

Chaque machine se présente pour l'élève sous forme d'une interface pédagogique qui conduit l'élève à construire la solution d'un exercice de la classe considérée et lui permet de compter [8]. Nous avons défini un modèle conceptuel objet du domaine[9] adapté à une utilisation interactive. Il permet de représenter cette solution sous forme arborescente.

2.2. Un exemple

Soit l'exercice E1 : « Avec un jeu de 32 cartes, combien peut-on former de mains de 5 cartes contenant exactement 3 dames et exactement 2 cœurs » cet exercice fait partie de la classe Construction_Ensemble_par_Cas. En effet, deux cas sont possibles : la dame de cœur fait ou non partie des deux cœurs. L'ensemble des configurations solutions CS (mains de 5 cartes vérifiant les contraintes) est formé de l'union de l'ensemble des configurations solutions contenant la dame de cœur CS1 et de l'ensemble des configurations solutions ne contenant pas la dame de cœur CS2. Ces deux ensembles sont disjoints donc Card CS = Card CS1 + Card CS2. C'est l'application du *principe d'addition* pour les dénombrements. Chacun de ces deux cas appartient à la classe de problème Construction_Ensemble.

Nous allons décrire dans la suite de l'article, la résolution du cas 1 dont l'énoncé E2 pourrait s'écrire :" Avec un jeu de 32 cartes, combien peut-on former de mains de 5 cartes contenant exactement la dame de cœur, 2 autres dames et exactement 1 cœur non dame. L'élève commence par énoncer une description constructive d'un élément de l'ensemble à dénombrer qui peut s'exprimer ainsi :

On veut former une main de 5 cartes prises dans un jeu de 32 cartes

On choisit la dame de cœur

On choisit 2 dames non cœur

On choisit 1carte de cœur non dame

On choisit 1 carte ni cœur, ni dame.

Pour déterminer le nombre de mains de 5 cartes correspondant aux contraintes de l'énoncé, il suffit alors de compter le nombre de choix possibles à chaque étape et d'en faire le produit. On utilise ici le *principe multiplicatif* enseigné aux élèves pour résoudre. Pour que ce principe puisse être utilisé il faut que les sous ensembles décrits à chaque étape soient disjoints. Par exemple, le sous ensemble décrit par "carte de cœur non dame" {7♥, 8♥, 9♥, 10♥, J♥, K♥, A♥} est bien disjoint du sous ensemble décrit par "dame non cœur" {Q♠, Q♦, Q♣}. La méthode constructive ne permet donc à l'élève que de donner une solution utilisant des ensembles disjoints. Les solutions obtenues par cette méthode ne font apparaître que des ensembles disjoints et permettent une justification du nombre calculé.

Pour introduire une description constructive, l'élève a à sa disposition une interface nommée "machine à construire une solution". La figure 1 représente la machine Construction_Ensemble utilisée successivement dans les deux cas de l'exercice traité, instanciée sur l'énoncé E2 .

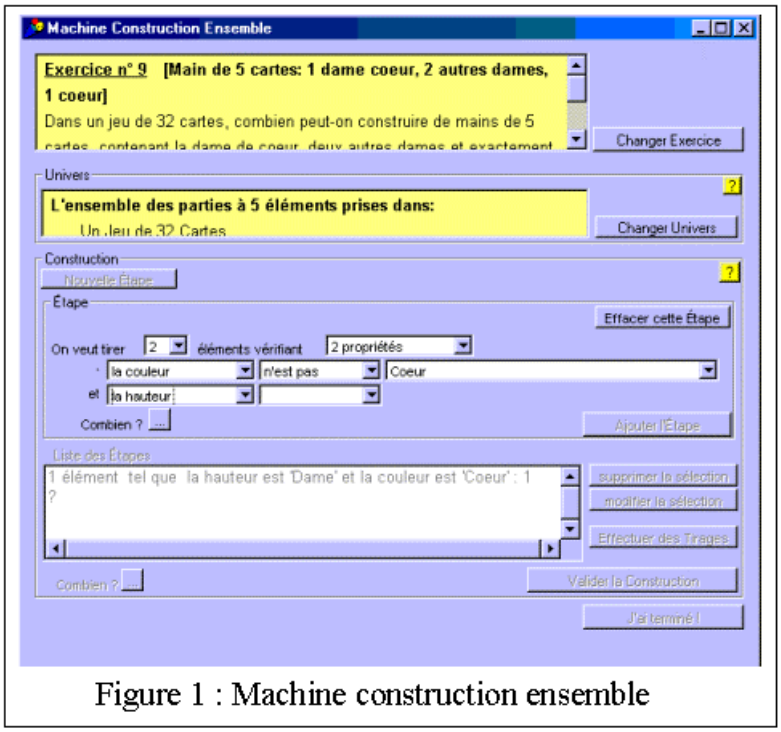


Figure 1 : Machine construction ensemble

Cette interface permet à l'utilisateur d'éditer sa solution sous forme de l'arborescence de la figure 2[*]. En fait, l'élève ne voit pas l'arborescence qu'il édite. Chaque nœud de l'arborescence a pour valeur un objet du modèle conceptuel du domaine. La solution est formée de l'*univers* (ensemble des mains de 5 cartes) et de la *construction* formée des 4 *sous-constructions* ou *étapes* (on choisit la dame de cœur, puis on choisit 2 dames non cœur, puis on choisit 1 carte de cœur autre que la dame enfin 1 carte ni cœur ni dame).

Pour toutes les classes d'exercice, l'élève énoncera sa solution sous forme constructive à l'aide d'une interface spécifique à la classe. Quelle que soit la complexité de l'exercice, une solution est composée d'un *univers* et d'une ou plusieurs *construction*s elles-mêmes formées de plusieurs sous-constructions : les *étapes.*

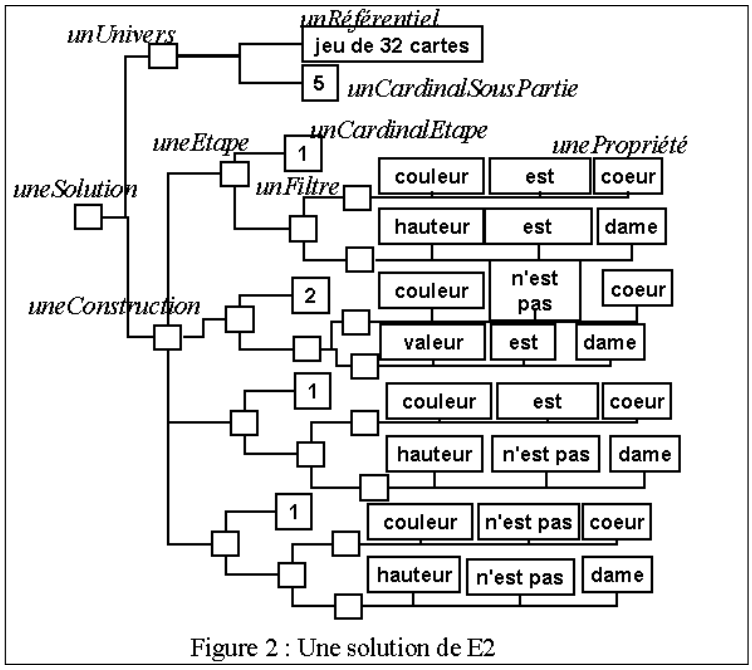


Figure 2 : Une solution de E2

Nous nous sommes attachés à définir les machines à construire comme des interfaces pédagogiques qui guident l'élève dans la construction de sa solution.

[*] Dans la figure, chaque rectangle ou carré représente un nœud de l'arborescence, quels que soient sa taille et son contenu. Les différences de taille et de contenu n'indiquent pas de différence de nature entre les nœuds.

3. Les erreurs dans une interface pédagogique

Une interface pédagogique de résolution de problèmes doit permettre à un élève utilisateur de formuler une solution à un problème qui lui est posé, mais surtout d'exploiter au mieux cette activité pour augmenter ses connaissances et ses capacités. Dans une telle interface la gestion des erreurs commises par l'élève ne concerne pas uniquement l'aspect ergonomique et fonctionnel, comme dans toute interface, mais relève en premier lieu d'une décision pédagogique. Généralement on essaie dans une interface de canaliser l'activité de l'intervenant pour qu'il effectue le moins d'erreurs possibles [3]. Il est en effet techniquement possible de concevoir l'interface pour que l'utilisateur ne puisse pas commettre certaines erreurs et cela dynamiquement en fonction de la situation (rendre indisponibles les boutons qui provoqueraient des erreurs ou ne faire figurer dans un menu que les items qui ne provoquent pas d'erreur)[4]. Dans une interface pédagogique, il faut au contraire pouvoir laisser à l'élève la possibilité d'exprimer sa solution et donc de commettre des erreurs mais seulement celles qui sont "intéressantes" du point de vue pédagogique.

3.1. Les erreurs dans le projet Combien ?

Pour construire des interfaces pédagogiques on est amené à rechercher quelles sont les erreurs pédagogiquement intéressantes et en faire une liste. Le problème n'est pas "comment faire la liste de toutes les erreurs dans un domaine", mais "comment restreindre suffisamment un domaine pour pouvoir déterminer une liste exhaustive des erreurs possibles dans la restriction choisie ?".

Nous pensons que de telles restrictions de domaines sont pédagogiquement intéressantes, parce qu'elles permettent de se concentrer sur certaines difficultés bien précises sans s'occuper des autres.

Dans le projet Combien?, nous avons restreint le domaine en contraignant l'élève à l'utilisation d'une méthode de résolution (la méthode constructive). Nous avons pour chaque classe de problèmes construit une machine correspondant à la modélisation spécialisée de la méthode pour cette classe. Nous avons éliminé une famille d'erreurs inintéressantes en contrôlant l'interface (menus déroulants dynamiquement construits, autorisation d'utilisation de boutons, séquencement...).

Nous avons distingué quatre catégories d'erreurs : la première catégorie concerne des erreurs qui sont détectables sans prendre en compte une solution de l'exercice, et ne font pas intervenir la méthode constructive. Nous avons utilisé pour elles les termes "impossible" ou "incohérent". On peut considérer que ce sont des erreurs d'inattention. Par exemple, on demande de tirer 5 as dans un jeu de 32 cartes, ou on construit une main de 6 cartes pour répondre à un tirage de 5 cartes. La deuxième catégorie concerne des erreurs dans l'utilisation de la méthode constructive. C'est, par exemple, l'utilisation de sous ensembles non disjoints pour la machine ConstructionEnsemble. Ces erreurs sont délicates à gérer car on n'en voit la raison d'être que plus tard, au moment de compter. Pour l'élève elles peuvent présenter un caractère arbitraire. Dans la troisième catégorie d'erreurs, la solution est correcte vis à vis de la machine mais elle est incomplète ou ne correspond pas au problème traité. Ces erreurs sont détectées par comparaison avec une solution correcte connue par le système. La dernière catégorie concerne les erreurs de calcul du nombre de possibilités à chacune des étapes.

Ces catégories d'erreurs se retrouvent dans chaque machine. Pour les détecter et les traiter nous avons cherché à mettre au point une méthode générale. Un problème est représenté par une structure arborescente qui comporte la représentation de l'exercice choisi et une solution correcte. Nos machines permettent d'ajouter à celle-ci la structure arborescente correspondant à la solution de l'élève au fur et à mesure de sa construction. Une construction se fait donc en ajoutant, à chaque événement validation, une sous-arborescence à la structure en cours de création. La détection des erreurs revient alors à rechercher si la structure ainsi modifiée vérifie ou non les contraintes de cohérence sémantique qui définissent ce qu'est une représentation correcte.

3.2. Le problème de la détection incrémentale d'erreurs

Nous nous plaçons dans le cadre général d'un système interactif où l'utilisateur édite une représentation conceptuelle structurée en arborescence. Le problème est de détecter les erreurs commises au cours de cette édition. Les erreurs considérées sont relatives à un modèle conceptuel du domaine d'application : la représentation éditée doit vérifier des contraintes de cohérence sémantique qui définissent ce qu'est une représentation correcte. Le fait que la saisie de l'arborescence soit interactive introduit le problème de la

détection *incrémentale* des erreurs. La structure n'est pas saisie en une fois, mais par morceaux, l'interface décidant de la taille des morceaux et dans une certaine mesure de leur ordre. Il s'agit d'une sous-arborescence parfois réduite à un nœud unique. A chaque saisie d'un morceau de la structure, un contrôle de validité est possible, permettant de détecter des erreurs avant même que la structure ne soit complète.

On examine alors les erreurs éventuelles provoquées par l'insertion de ce morceau que nous appellerons dans la suite *cible*. Dans une détection incrémentale d'erreurs, on suppose qu'on a déjà déterminé toutes les erreurs commises avant l'insertion. On s'intéresse uniquement aux nouvelles erreurs introduites par l'insertion de la cible. Elles font donc nécessairement intervenir des nœuds appartenant à la cible. Elles peuvent provenir d'incohérences internes à la cible, ou de violations de contraintes faisant intervenir la cible et d'autres éléments de la structure.

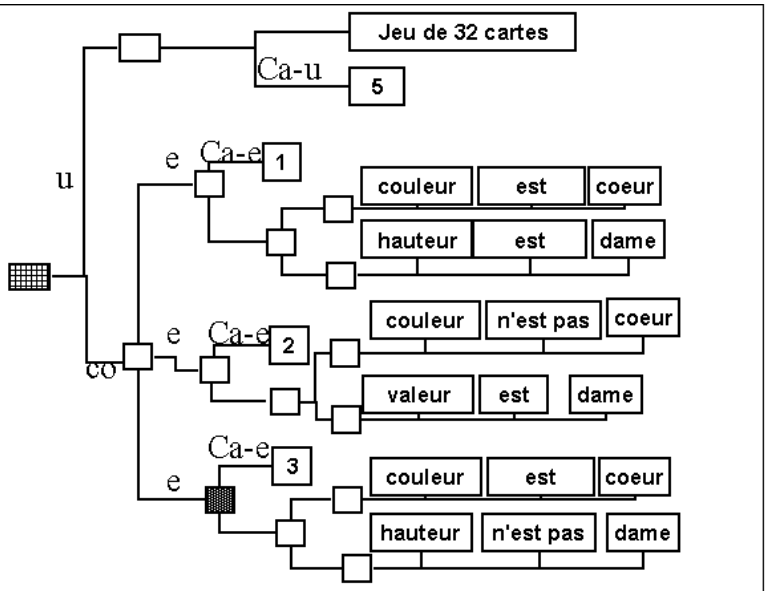


Figure 3 : Elaboration de solution de E2 comportant une erreur

La figure 3 illustre une situation prise dans l'élaboration de la solution de l'exercice E2 (cf. §2.2) où l'on détecte une erreur provenant d'une cible qu'on vient d'insérer. Avec cette proposition de l'élève, les mains construites auraient plus de 5 cartes, ce qui est contradictoire avec la contrainte initiale qu'il a lui-même énoncée. Les éléments qui participent à la violation d'une contrainte sont les noeuds étiquetés 5, 1, 2 et 3 représentant ici des nombres de cartes. La contrainte non respectée est la suivante : à tout moment, la somme des cardinaux des différentes étapes (ici 1 + 2 + 3) doit être inférieure ou égale au cardinal des parties imposé par l'univers (ici 5).

Le non respect de cette contrainte correspond à un certain type d'erreur : celle qui consiste à ajouter une étape rendant le cardinal total trop grand. Cette description de l'erreur comporte en fait deux aspects : un aspect théorique concernant le domaine (la somme est trop grande) et un aspect interactif (cette erreur se produit lorsqu'on ajoute l'étape).

Le problème est de donner une représentation formelle et opérationnelle de cette erreur. Pour cela, la représentation doit indiquer *où* il faut chercher l'erreur, c'est-à-dire à quels éléments s'intéresser (ici il faut considérer l'élément 5 et la construction de l'élève en position co), *comment* on va reconnaître qu'il y a bien une erreur (ici on le reconnaît à la propriété 1 + 2 + 3> 5) et *quand* on doit chercher à détecter cette erreur (ici, lors de l'insertion de la nouvelle étape « choisir 3 cœurs non dames »).

Pour indiquer *où* chercher les éléments intéressants, on va donner leurs rôles et leurs positions dans la structure. Par exemple, le rôle de l'élément 5 est d'être le cardinal des parties défini dans l'univers. Cela peut se traduire par un chemin d'attributs à partir de la racine : u/Ca-u.

Pour indiquer *quand* on doit chercher à détecter l'erreur, on va donner le rôle et la position de la cible par un chemin d'attributs à partir de la racine : co/e.

Pour indiquer *comment* reconnaître qu'il y a une erreur, on va donner une *condition d'erreur*, formalisée comme l'application d'un prédicat aux éléments intéressants. Pour cela, il faut pouvoir discerner et désigner ces éléments intéressants, par exemple en les nommant. Soit X un élément se trouvant au bout d'un chemin u/Ca-u, soit C la construction (chemin co). Alors la condition d'erreur peut s'exprimer par : sommeCardinauxPlusGrande(C, X), traduisant la propriété : « la somme des cardinaux des étapes de C est supérieure à X ». Dans notre implémentation, ce prédicat est testé par une procédure programmée.

Nous aboutissons ainsi à la représentation suivante de cette erreur (ou plutôt de ce type d'erreur) :

dans le contexte d'une solution proposée par l'élève, lorsqu'on ajoute une cible à la position co/e, il y a une erreur si la condition sommeCardinauxPlusGrande(C, X) est vérifiée, avec C à la position co et X à la position u/Ca-u.

Les indications de position peuvent se noter graphiquement, en dessinant un *patron* (pattern) d'une sous-arborescence faisant apparaître uniquement les éléments intéressants, leurs positions et leurs noms. Sur ce même patron, on peut indiquer la position de la cible. Décrire un type d'erreur revient donc à fournir deux choses : d'abord un patron et ensuite une condition d'erreur s'appliquant à des éléments nommés du patron. Il faut de plus traduire l'expression « dans le contexte d'une solution ». En effet, la solution de l'élève n'est en fait qu'une sous-arborescence de l'arborescence complète qui est construite (celle-ci contient aussi, par exemple, l'exercice choisi par l'élève et une solution correcte de cet exercice). Pour traduire cela, on indique la position de la racine du patron par rapport à la racine de l'arborescence complète. La figure 3bis montre la description du type d'erreur correspondant à l'exemple de la figure 3.

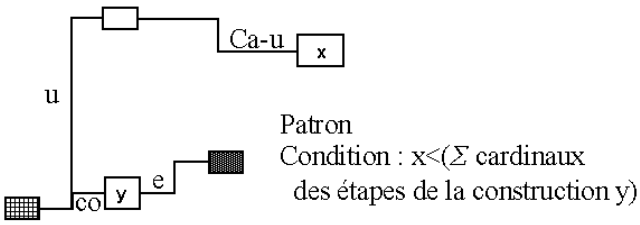


Figure 3bis : Description d'un type d'erreur

Nous avons ainsi les éléments d'un modèle que nous appelons *détection ciblée* : une représentation arborescente d'objets d'un domaine, un patron d'erreur avec des variables (éléments nommés), une condition d'erreur, une cible, un contexte. Ce modèle a été décrit dans [5]. Nous en rappelons les grands traits dans le paragraphe suivant.

4. Détection ciblée

4.1. Le modèle de la détection ciblée

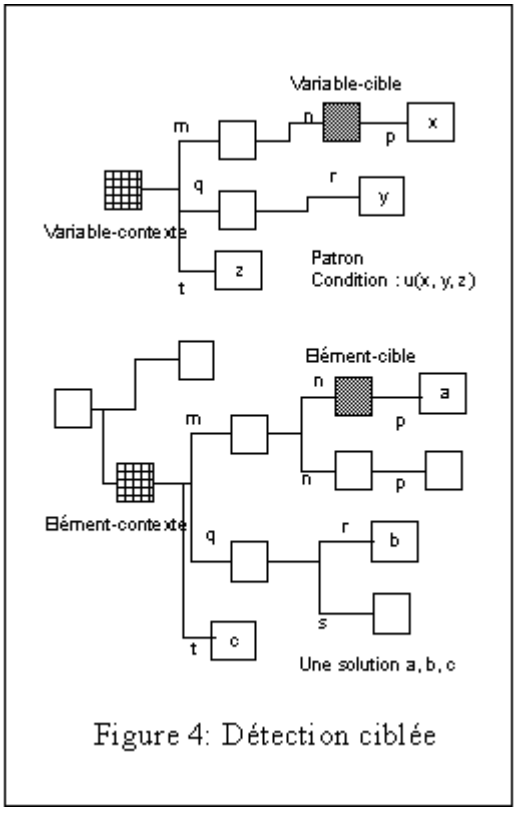


Figure 4: Détection ciblée

Le problème de la détection incrémentale d'erreurs, tel que nous l'avons présenté ci-dessus, s'inscrit dans un cadre plus large : celui de la recherche d'instanciations de variables d'un patron (*pattern*) arborescent dans une structure de données arborescente, certaines variables étant assujetties à vérifier une certaine condition qui les lie. A ce thème, nous ajoutons l'idée de *ciblage* : on distingue un nœud particulier du patron appelé *variable cible*, et on se limite à chercher les instanciations des variables du patron pour lesquelles la variable cible s'apparie à un élément particulier de la structure, appelé *élément cible*. La racine du patron n'est pas contrainte à s'apparier avec la racine de la structure. Elle peut s'apparier avec un nœud quelconque, du moment que les autres conditions sont respectées. Nous appellerons *élément contexte* l'élément apparié avec la racine du patron, celle-ci étant appelée *variable contexte*. Ce type de problème s'apparente à la résolution de requêtes dans une base de données ou dans un langage de programmation logique. Nous l'appelons ici *détection ciblée*.

La figure 4 illustre une détection ciblée : en haut, on voit un patron d'arborescence comportant des variables x, y, z, une variable contexte, une variable cible et une condition u(x,y,z) ; en bas, on voit la structure dans laquelle on cherche et une solution instanciation du patron dans laquelle les variables a, b, c vérifient la condition u.

4.2. Les connaissances et leur utilisation

Pour pouvoir détecter et expliquer des erreurs lors de l'insertion d'une cible, le système a besoin de connaissances particulières. Ces connaissances sont regroupées dans des structures de données appelées *schémas d'erreurs*. Chaque schéma d'erreur est décrit par :

- Un patron arborescent. Chaque variable du patron est repérée par son chemin d'attribut à partir de la variable contexte.

- Une condition d'erreur liant certaines variables
- Un chemin d'attributs indiquant comment le contexte (la racine du patron) doit être relié à la racine de la structure étudiée. C'est la contrainte d'*ancrage* du contexte. Dans l'explication de l'erreur, seule l'arborescence ayant pour racine le contexte est significative : elle contient toutes les variables intervenant dans la condition d'erreur. On peut d'ailleurs toujours se ramener au cas où le contexte est le premier ancêtre commun à toutes ces variables.
- Un concept dont le contexte doit être instance. C'est la contrainte de *typage* du contexte. Elle pourrait être incluse dans la condition d'erreur, mais il est pratique de la séparer pour mieux catégoriser et indexer l'erreur.
- Un nœud du patron jouant le rôle de variable cible

La forme des arborescences dans la structure ("représentation conceptuelle") dépend du modèle conceptuel choisi (quelles classes, quels attributs ou relations) et la forme des arborescences dans les patrons dépend en plus d'une décision d'implémentation propre à chaque schéma d'erreur : que met-on dans le patron, et que met-on dans la condition d'erreur procédurale ? Un patron ne définit pas entièrement une erreur. Il faut lui ajouter une condition d'erreur procédurale. Le patron permet de proposer des candidats éventuels comme occurrences d'erreurs, et la condition permet de décider lesquels de ces candidats sont vraiment des occurrences d'erreurs. Il y a une sorte d'équilibre à trouver entre complexité du patron et complexité de la condition en fonction de l'implémentation choisie.

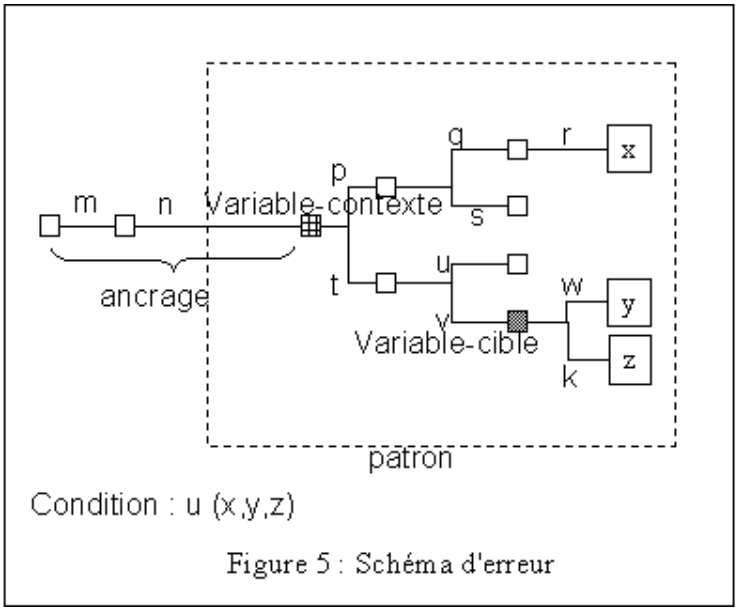


Figure 5 : Schéma d'erreur

La figure 5 résume les connaissances contenues dans un schéma d'erreur. Suivant l'application visée, un schéma d'erreur peut être augmenté d'autres connaissances, en particulier celles qui sont nécessaires à l'explication de l'erreur dans le domaine considéré.

Lorsqu'on a fait un catalogue de types d'erreurs pour un domaine donné dans une application interactive, on dispose d'une base de schémas d'erreurs. Le problème est alors de détecter toutes les occurrences d'erreurs introduites par l'insertion d'une nouvelle cible dans la structure globale en cours d'édition, une occurrence d'erreur étant définie par un schéma d'erreur avec une instanciation de ses variables.

Pour cette détection, nous proposons *le principe de l'algorithme* suivant qui s'appuie sur la structure arborescente du modèle du problème :

On examine un par un les ancêtres de la cible, en commençant par la cible elle-même. Chacun d'eux est un contexte potentiel d'un schéma d'erreur. Pour chacun d'eux, on examine chaque schéma d'erreur et on ne s'intéresse qu'à ceux dont les contraintes d'ancrage et de typage du contexte sont réalisées. Pour ces schémas, on essaye d'apparier l'élément cible de la structure à la variable cible. On ne retient que les schémas pour lesquels c'est possible. Pour chaque schéma retenu, on cherche alors toutes les instanciations des variables qui vérifient la condition d'erreur.

Ce principe général peut être amélioré en indexant les schémas d'erreur dans la base. Pour chaque cible et chaque contexte candidat dans la structure, il est facile de calculer les chemins qui relient ces éléments à la racine de l'arborescence. On peut alors utiliser le couple formé par ces deux chemins comme clé d'indexation des schémas d'erreur, ce qui permet pour un couple donné de sélectionner directement les schémas vérifiant la contrainte d'ancrage du contexte et l'appariement correct de la cible.

4.3. Adéquation de cette représentation

Cette représentation des connaissances nous permet de traiter le problème de la détection incrémentale d'erreurs dans un système interactif.

Les connaissances sont données de façon modulaire : les schémas d'erreur sont indépendants, on peut les éditer séparément.

Le degré de déclarativité est élevé : les connaissances données dans un schéma d'erreur ne sont pas mélangées à la façon de les utiliser. La seule partie pouvant faire intervenir des éléments procéduraux est la condition d'erreur, dont la forme dépend de la formalisation du domaine. Un schéma d'erreur sert de ressources à la disposition de tout mécanisme voulant l'utiliser. Cela permet entre autres de bien séparer la détection des erreurs et leur signalement : le schéma d'erreur ne prend pas l'initiative de signaler l'erreur lorsqu'elle est détectée, c'est un mécanisme client qui s'en charge. On peut changer la stratégie de signalement sans changer les schémas d'erreur.

Dans un schéma, on sépare bien ce qui ressort du domaine (le patron, la condition d'erreur, les contraintes d'ancrage et de typage du contexte) et ce qui ressort de l'interactivité du système (la cible). Du point de vue de l'expertise du domaine, un type d'erreur ne fait pas intervenir la notion de cible. Un expert du domaine peut énoncer les connaissances correspondantes sans se préoccuper de la façon dont elles seront utilisées interactivement. De son côté, le concepteur du système interactif peut, pour chaque type d'erreur défini de cette façon, ajouter une indication de cible. Pour lui, la cible correspond à un morceau de la structure que l'interface permet de saisir et de valider. Décider quelles cibles l'interface doit faire apparaître correspond à un choix ergonomique (et, dans le cas de notre application, didactique).

L'algorithme utilisé permet de faire apparaître les erreurs dans un ordre pertinent pour l'utilisateur : d'abord les plus locales (les plus proches de la cible), censées être plus visibles et plus compréhensibles, puis ensuite des erreurs plus globales (lorsqu'on remonte dans des contextes de plus en plus larges), plus complexes à appréhender.

La réification de chaque type d'erreur sous forme de schéma permet d'y accrocher d'autres connaissances (des explications, des conseils, des suggestions de correction) qui trouvent là un support naturel.

Les schémas étant des connaissances déclaratives, il est possible à un mécanisme de niveau méta de les analyser et de les traiter, par exemple pour évaluer leur difficulté (en fonction par exemple de la complexité du patron, des concepts apparaissant dans la condition, etc.).

5. Détection ciblée et Combien?

5.1. Environnement de recherche d'erreurs

Le but d'une interface pédagogique est de faire prendre conscience aux élèves d'un certain nombre de concepts du domaine. Le concepteur de l'interface choisit les concepts qui lui semblent importants, et propose à l'élève des outils d'édition pour les objets correspondant à ces concepts. Notre système cherche à promouvoir une activité de modélisation de la part de l'élève. Dans ce cadre certaines de ses erreurs proviennent de contradictions internes au modèle qu'il propose. Prendre conscience de ces contradictions peut le motiver pour approfondir les concepts sous-jacents. Le système TAPS [2] suit une démarche analogue.

Nous avons défini les concepts à retenir et construit nos interfaces de façon à faire apparaître les *cibles* correspondantes. Dans notre modèle nous voulons que l'élève retienne qu'une solution est composée d'un *univers* et d'une *construction* elle-même formée de plusieurs *sous-constructions*. Nous définissons les machines de façon que l'élève n'ait à valider que les éditions de concepts à retenir. A chaque validation de l'élève (bouton Ok, ou valider), le système construit la cible et la rajoute à la solution en construction. Il recherche les erreurs éventuelles que cela provoque. Les principes de la détection ciblée s'appliquent ici et nous permettent de faire une analyse fine des erreurs.

Dans la machine constructionEnsemble présentée §2.2 nous avons déterminé quinze types différents d'erreurs pédagogiquement intéressantes. Par exemple, lorsque l'élève valide l'univers (main de x cartes pris dans un jeu de 32 cartes cf. Figure 1) une erreur possible est : le cardinal des éléments de l'univers n'est pas le bon. Pour pouvoir l'expliquer la définition de l'erreur doit être très précise.

En fait l'erreur ci-dessus peut provenir de deux types d'erreurs. Soit le cardinal d'un élément de l'univers est supérieur au cardinal du référentiel (x vaut par exemple 33), soit il ne correspond pas au cardinal correspondant de la solution-système (x vaut par exemple 4). Le premier type d'erreur est contenu dans le deuxième, nous pensons qu'il est important de les distinguer car elles ne correspondent pas au même type d'erreur au niveau de la connaissance. L'une est une incohérence au niveau de la connaissance d'un jeu de 32 cartes, elle relève le plus souvent d'une erreur d'inattention ou d'une faute de frappe. Pour la détecter, il suffit de savoir ce qu'est un jeu de cartes. L'autre est une erreur de résolution. Pour la détecter, il faut connaître la solution-système et l'explication de l'erreur fera intervenir les concepts de la solution. Dans notre représentation ces deux erreurs correspondent à deux *contextes* différents de l'arborescence. Pour la première erreur, le *contexte* est l'univers lui-même. Pour la deuxième erreur, le *contexte* est la racine de l'arborescence composée de l'*exercice*, la *solution système* et la *solution-élève*.. Nous avons nommé cette racine *environnement*,

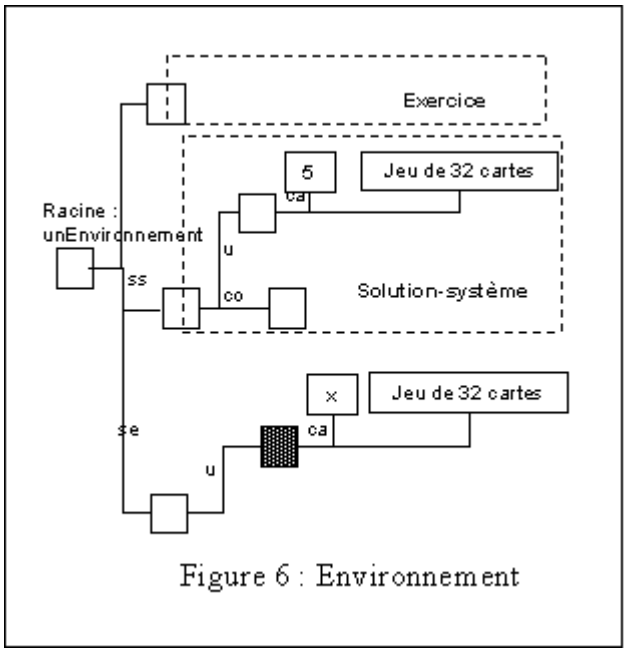


Figure 6 : Environnement

La figure 6 montre l'environnement de recherche des erreurs dans une machine. Les rectangles en pointillés indiquent que les sous arborescences ne sont pas détaillées. La solution-élève est en cours d'élaboration. Les arcs sont étiquetés par les attributs du modèle objet, par exemple *se* pour solution-élève, *u* pour univers et *ca* pour cardinal.

L'élève vient de définir l'univers (chemin se/u) et le système va pouvoir rechercher si une des erreurs présentées ci-dessus est présente.

5.2. Schémas d'erreur

Nous avons associé à tous les types d'erreur répertoriés des schémas d'erreur. Ils sont tous définis dans l'arborescence de l'*environnement* qui sera donc le nœud par rapport auquel on *ancrera* le *contexte*. Notre système est bâti sur un modèle conceptuel du domaine permettant d'exprimer aussi les solutions. Les nœuds de l'arborescence correspondent donc à des objets du domaine (instance des classes) et les étiquettes sur les arcs à des attributs de ces objets. Pour définir le patron il nous suffit donc de donner les chemins du contexte aux variables. De même le chemin d'ancrage du contexte donne en même temps le typage du contexte. Actuellement la condition d'erreur du schéma est donnée par un nom, qui spécifie une procédure avec un ou deux arguments.

Le schéma d'erreur correspondant à "cardinal d'un élément de l'univers supérieur au cardinal du référentiel" est le suivant :

cheminContexte : se/u cheminCible : vide Relation : cardUniversPlusGrandCardReferentiel Argument1 : ca

Les notations sont celles utilisées dans la figure 6, se est l'arc de l'environnement vers la solution-élève, u est l'arc de la solution-élève vers l'univers. Le contexte est l'univers. Dans ce type d'erreur, le contexte et la cible sont les mêmes donc le chemin du contexte à la cible est vide. L'univers est composé de deux attributs : un cardinal et un référentiel. L'erreur porte sur l'aspect cardinal de l'univers.

Pour le deuxième type d'erreur, le cardinal de l'univers de la solution-élève n'est pas égal à celui de la solution-système, nous avons le schéma suivant :

cheminContexte : vide cheminCible : se/u Relation : estDifferentDe Argument1 : ca Argument2 : ss/u/ca

Ici, le contexte est la racine de l'arborescence, ss est l'arc de l'environnement vers la solution-système. L'argument2 est le cardinal de l'univers de la solution-système.

Les schémas d'erreurs sont mémorisés dans une base de données dans laquelle ils sont indexés par le couple (contexte, cible).

5.3. Détection d'erreurs

La recherche d'erreurs s'effectue lors de l'insertion d'une cible. L'algorithme utilisé est une implémentation de celui qui est décrit au §4.2. Les schémas d'erreurs sont indexés par le couple (contexte, cible). Lorsque l'élève édite l'univers au moyen de l'interface, il définit les deux aspects de cet univers puis clique sur le bouton "valider". L'événement valider provoque l'envoi du message "vérifier (univers)". Nous déroulons l'algorithme en ne prenant en compte que les deux schémas d'erreur définis au paragraphe précédent. La cible est l'univers et le premier contexte est l'univers. Un schéma d'erreur est recherché à ce niveau. C'est le premier schéma décrit qui est trouvé, et la procédure *cardUniversPlusGrandCardReferentiel* est déclenchée. Si la condition d'erreur est vérifiée, une occurrence de l'erreur est créée. Le même travail est ensuite réalisé au niveau du contexte précédent (la solution-élève), pour lequel nous n'avons pas décrit dans cet article de schéma d'erreur. Le contexte suivant est la racine Environnement, au niveau duquel le deuxième schéma décrit est trouvé. La procédure relative à la relation *cardUniversSEDifferentcardUniversSS* est déclenchée. Si la condition d'erreur est vérifiée, une occurrence de cette erreur est créée. La détection des erreurs correspondant à l'accrochage de la cible univers à la solution-élève en cours, est réalisée et la machine possède la liste des occurrences d'erreurs créées. Le processus se poursuit tant que la solution élève n'est pas terminée. Les deux schémas d'erreur précisés ci-dessus ne sont qu'une partie de l'ensemble des schémas d'erreurs associés à la classe de problèmes dont fait partie l'exercice E2 traité dans cet article. D'autres schémas d'erreurs concernent la partie construction de la solution et correspondent à des types d'erreurs plus complexes. Par exemple, lorsque l'élève donne les étapes successives de sa construction, il définit en fait des sous-ensembles. Dans l'étape : "je choisis deux cartes dont la hauteur est dame et dont la couleur n'est pas cœur", il détermine le sous-ensemble {Q♠, Q♦, Q♣}. La méthode de construction nécessite la disjonction de ces sous-ensembles [6]. La non-disjonction se traite comme les erreurs précédemment évoquées. Le schéma d'erreur correspondant est associé au contexte solution de l'élève car les connaissances nécessaires à la reconnaissance de la disjonction demandent des connaissances sur l'univers.

En fonction des options pédagogiques choisies, les moments de signalement et/ou d'explication des erreurs seront ou non les mêmes que les moments de détection des erreurs [4]. En effet, dans une interface pédagogique, il faut quelquefois laisser l'élève s'exprimer même si l'on a détecté une erreur, soit pour qu'il s'en aperçoive tout seul plus tard, soit pour lui montrer qu'il arrive à une impasse et lui fournir l'occasion de progresser dans son apprentissage [10].

6. Réalisation

Les exercices de dénombrement de la classe de terminale S que nous traitons, correspondent à une quinzaine de classes de résolution différentes. Nous avons particulièrement travaillé sur quatre d'entre elles que nous avons nommées constructionEnsemble, constructionEnsembleParCas, constructionListe et constructionAssociation.

L'exercice E1 fait partie de la classe constructionEnsembleParCas. Les deux sous-exercices qui en découlent font partie de la classe constructionEnsemble. Un autre exemple de cette classe est : "Avec un jeu de 32 cartes, combien peut-on former de mains de 12 cartes contenant **au moins** 7 cœurs, **exactement** 4 as, 2 piques et **rien** d'**autre**". Cet exercice est plus difficile à résoudre pour l'élève bien que la structure de la solution soit la même. Une solution correcte serait :

On veut former des mains de 12 cartes prises dans un jeu de 32
On choisit 4 as
On choisit 1 pique non as
On choisit 7 cœurs non as.

La difficulté de l'exercice provient du fait que pour n'avoir que 12 cartes en suivant les contraintes, il faut s'apercevoir que tous les cœurs font partie de chaque configuration-solution. Si la première étape de la partie construction de la solution de l'élève est : je choisis 7 cœurs, l'ajout de la cible 7 cœurs provoque une erreur dans le contexte *environnement* (connaissance de la solution -système nécessaire). En effet, quelle que soit la suite de la solution de l'élève, elle ne pourra pas conduire à une solution correcte car il ne pourra pas compléter avec le dernier cœur sans utiliser un sous-ensemble non disjoint de celui choisi à la première étape.

Un exemple de la classe constructionListe est : "Combien peut-on former de mots de 5 lettres contenant exactement deux fois la lettre a". Un mot est en fait une application de l'ensemble des 5 places (référentiel-source) dans l'alphabet (référentiel-but). Cette application peut être injective ou non, dans la machine l'injectivité est exprimée comme dans les énoncés des dénombrements en utilisant la notion de "avec ou sans remise". L'univers pour cet exemple est l'ensemble des mots de 5 lettres avec remise. Dans cette classe, on retrouve certains des types d'erreurs de la classe précédente, par exemple : le cardinal de l'univers de la solution-élève n'est pas égal à celui de la solution-système. Il existe de plus des types d'erreurs spécifiques.

La classe constructionAssociation permet de résoudre des exercices comme : "Une association sportive a un bureau composé de 8 personnes, combien y a-t-il de façons d'élire un trésorier et deux secrétaires, une même personne pouvant être à la fois trésorier et secrétaire". Dans cette classe, il y a deux référentiels en jeu : un référentiel-source (dans l'exemple c'est l'ensemble des rôles à attribuer aux personnes) et un référentiel-but (l'ensemble des personnes). L'univers est alors soit l'ensemble des relations du référentiel-source vers le référentiel-but, soit l'ensemble des fonctions du référentiel-source vers le référentiel-but, soit l'ensemble des applications du référentiel-source vers le référentiel-but, ces fonctions ou applications pouvant être injectives ou non. Dans l'exemple cité, l'univers est l'ensemble des relations quelconques de l'ensemble des rôles {secrétaire, trésorier} vers l'ensemble des personnes. Les exercices de la classe constructionListe peuvent aussi être résolus par la machine relative à la classe constructionAssociation. Dans ce cas l'ensemble des places est le référentiel source et la notion de remise dans le référentiel but s'exprime par l'injectivité des fonctions ou des applications. On retrouve de nouveau le type d'erreurs : le cardinal de l'univers de la solution-élève n'est pas égal à celui de la solution-système, associé à des types d'erreur spécifiques.

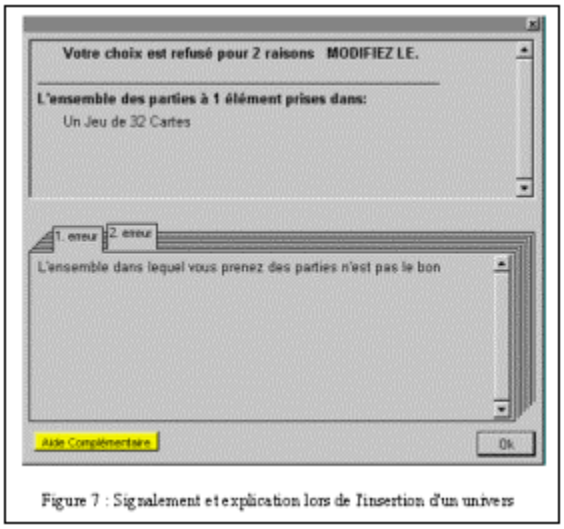


Figure 7 : Signalement et explication lors de l'insertion d'un univers

Nous avons implémenté les principes décrits dans cet article en utilisant les schémas d'erreurs pour les 3 classes de problèmes de base citées ci-dessus. Nous avons réalisé un éditeur de schémas d'erreur pour créer nos bases de schémas. Chacune d'entre elles contient au moins une quinzaine de schémas. Actuellement, certaines des interfaces réalisées permettent de dialoguer avec l'utilisateur au niveau de chaque attache de cible. Elles indiquent les différents schémas d'erreurs à chacun des niveaux.

La machine constructionEnsemble signale les erreurs au moment de leur détection et donne à l'élève une explication qui est attachée au schéma d'erreur. Cette explication est à deux niveaux. Un premier message signale les symptômes, sans expliciter la raison de l'erreur ni le remède. Un bouton Aide supplémentaire permet d'obtenir des précisions sur cette erreur (cf. figure 7).

7. Conclusion

Dans cet article, nous avons défini un mécanisme de détection ciblée qui permet d'identifier un patron associé à une condition liant certaines de ses variables dans une structure arborescente conceptuelle. Nous

avons montré comment il peut être utilisé pour la détection incrémentale d'erreurs. Ce mécanisme permet de donner déclarativement les types d'erreur et d'avoir un algorithme systématique de détection d'erreurs pour un système interactif à l'aide duquel l'utilisateur édite une représentation conceptuelle structurée en arborescence.

Dans le projet Combien?, l'élève a à sa disposition des interfaces lui permettant d'éditer sa solution et de compter à chaque étape le nombre de possibilités pour obtenir le nombre d'éléments demandés. Dans la modélisation que nous avons faite, la solution que l'élève doit éditer est une arborescence faisant intervenir les concepts du domaine. Lors de la création de l'interface, nous avons défini les concepts que l'élève doit apprendre. Un moment de validation correspond à l'édition d'une sous arborescence correspondant à un de ces concepts. Nous avons implémenté dans ces interfaces le mécanisme de détection ciblée pour la recherche incrémentale des erreurs commises par l'élève en prenant comme cibles ces sous arborescences. Nous avons défini pour chaque interface une base de schémas d'erreurs indexée par le couple contexte-cible. A l'ajout d'une cible nous détectons toutes les erreurs possibles pour cette cible dans les différents contextes. Le traitement des erreurs (signalement et explication) pourra se faire sur le champ ou être différé selon des choix pédagogiques. Les schémas d'erreurs sont des structures déclaratives qui contiennent toutes les connaissances nécessaires pour ce traitement.

8. Bibliographie

[1] Astolfi J.-P., *L'erreur, un outil pour enseigner*, *Paris*, ESF, 1997

[2] Aziz N., Pain H. and Brna P. (1995). *Handling contradictions in student modelling in Translating Algebraic Problems Systems (TAPS)*, Proceedings of the International Conference on Computers in Education (ICCE96), Singapore, December 1995, pp 614-621. AACE, Virginia, USA. (Nominated for best paper prize)

[3] Choplin H., Gallisson A., Lemarchand S. : *Hypermédias et pédagogie : comment promouvoir l'activité de l'élève ?* - Actes du quatrième colloque Hypermédias et apprentissages, pp 87-98, J.F.Rouet et B. de la Passardière eds, 1998.

[4] Duma J, Giroire H, Le Calvez F., Tisseau G., Urtasun M., *Gestion des erreurs dans une interface pédagogique.*Colloque métaconnaissances Berder 2000, rapport interne Lip6 2001 n°14, pp.22-37, 2001.

[5] Giroire H., Le Calvez F., Tisseau G., Urtasun M., Duma J., *Un mécanisme de détection incrémentale d'erreur et son application à un logiciel pédagogique.* RFIA 2002, pp. 1063-1072.

[6] Le Calvez F., Urtasun M. , Tisseau G., Giroire H., Duma J. *Les machines à construire : des interfaces pour apprendre une méthode constructive de dénombrement*. Actes des 5èmes Journées francophones EIAO, pp 49-60, M. Baron, P. Mendelsohn, J.F. Nicaud eds, Hermès, 1997.

[7] Py D., Quelques méthodes d'intelligence artificielle pour la modélisation de l'élève. STE, Vol 5-2, pp. 123-140, Hermes, 1998.

[8] Tisseau G., Giroire H., Le Calvez F., Urtasun M., Duma J., *Principes de conception d'un système pour enseigner la résolution des problèmes par la modélisation*. RFIA'2000, pp.121-130, Paris, 2000.

[9] Tisseau G., Giroire H., Le Calvez F., Urtasun M., and Duma J., *Design principles for a system to teach problem solving by modelling.* Lecture Notes in Computer Science N° 1839, ITS'2000, pp. 393-402, Springer-Verlag, Montréal, 2000.

[10] Tsukasa Hirashima, Tomoya Horiguchi, Akihiro Kashihara, Junichi Toyoda, *Error-Based Simulation for Error-Visualization and Its Management.* International Journal of Artificial Intelligence in Education, 9, pp. 17-31, 1998.

IA ET ETERNITY

Tristan Cazenave

Laboratoire d'Intelligence Artificielle
Département Informatique, Université Paris 8,
2 rue de la Liberté, 93526 Saint Denis, France.

cazenave@ai.univ-paris8.fr

Résumé: Dans ce papier, nous exposons la façon dont le puzzle Eternity a été résolu par AlexSelby et Olivier Riordan. Nous mettons en perspective les méthodes qu'ils ont utilisé avec des techniques d'Intelligence Artificielle similaires.

Mots Clés: Recherche heuristique, puzzle, Eternity.

Introduction

Eternity est un puzzle. Il a été commercialisé en Angleterre en 1999 au prix de 30 Livres. Ce fut un des jeux qui y fut le plus vendus en 1999/2000 : le nombre de puzzles vendus dépassa le nombre des ventes de Trivial Pursuit. Il est vrai que son auteur, Christopher Monckton un ancien conseiller de Mrs Tatcher, avait offert un prix de 1 million de Livres à la première personne qui trouverait une solution. Ce n'est pas la peine de vous précipiter pour acheter le puzzle, il a été résolu en Mai 2000 par Alex Selby (70%), Dr en Mathématiques de Cambridge, 3ème Dan au jeu de Go, sans emploi, programmeur de Go potentiel... et Olivier Riordan (30%), Dr en Mathématiques de Cambridge, chercheur au Trinity College. Ils ont devancé de peu le grand maître d'Echecs par correspondance Guenter Stertenbrink, qui a trouvé une deuxième solution en Juillet 2000.

Nous allons décrire les méthodes qu'Alex Selby et Olivier Riordan ont employé pour résoudre ce problème, à partir de notes d'Alex Selby [Selby].

Les pièces et le puzzle

Eternity compte 209 pièces. Le but est de remplir complètement un dodécagone (forme régulière à douze cotés égaux) avec les 209 pièces qui sont des polydrafters. Les polydrafters sont des pièces composées de n triangles rectangles (30-60-90). Il y a 6 didrafters, 14 tri, 64 tetra, 237 penta, 1014 hexa, 4124 hepta et 17705 octadrafters. Aucune des pièces d'Eternity n'a d'angle de 30 degrés ou de triangles vides. Les polygones qui ont cette propriété s'appellent des polydudes. Avec ces contraintes, il y a 3 didrafters, 1 tridrafter, 9 tetradrafters, 15 pentadrafters, 59 hexadrafters, 152 heptadrafters, 517 octadrafters, 1547 nonadrafters, and 5064 decadrafters, 16123 hendecadrafters, and 52630 dodecadrafters.

Eternity utilise des dodécadudes (pièces composées de 12 triangles rectangles sans angle de 30 degrés ou de triangle vide).

La création du puzzle est facile, pas sa résolution

Contrairement à sa résolution, la création du puzzle est facile. On peut tout d'abord remarquer que toutes les pièces ont la même aire (6 triangles équilatéraux ou 12 triangles rectangles). De plus, il existe 770 pièces qui ont des propriétés convenables pour faire partie du puzzle. Il est assez facile de remplir une figure de taille 209 avec 770 pièces. Il suffit de choisir les pièces qui correspondent aux aires vides à la fin. En revanche, il

est beaucoup plus dur de remplir un figure de taille 209 avec 209 pièces. A la fin, il reste très peu de choix pour les aires restées vides.

Régions, aires, états et sites.

On définit la taille d'une région comme l'aire de cette région divisée par l'aire d'une pièce. Un état est une région associée à un ensemble de pièces. S'il y a plus de pièces que la taille de la région, il restera des pièces lorsque la région sera remplie, et il peut y avoir plus d'une solution. Un site est un espace vide connexe sur une région.

La force brute

La solution la plus simple qui vient à l'esprit pour résoudre Eternity est la force brute : une recherche exhaustive de toutes les possibilités de remplir une région. Cela ne marche pas pour trouver la solution du puzzle complet, mais cela est utile pour résoudre des puzzles de tailles plus petites.

On peut utiliser l'algorithme de force brute suivant pour paver une région : Etant donné un mécanisme pour choisir un site, on considère une branche par façon de poser une pièce sur le site choisi. Aucune solution ne manque puisque le site doit être couvert pour toutes les solutions. Aucune solution n'est comptée deux fois puisque chaque pièce placée amène à un ensemble disjoint de solutions. L'algorithme marche toujours si on a plus de pièces que la taille de la région.

Cette méthode de remplissage est plus adaptée à Eternity que de choisir la prochaine pièce et de backtracker sur ses placements possibles. On peut avec cette méthode choisir le site à paver. Intuitivement, on voit bien qu'il faut choisir le site pour lequel on a le moins de façons possibles de placer une pièce. Si il y a une impossibilité dans la région, il faut la détecter le plus tôt possible, et ne pas perdre de temps à backtracker sur des régions faciles alors qu'il n'y a pas de solutions. Cette propriété d'Eternity se rencontre aussi dans d'autres problèmes. Par exemple pour le coloriage de cartes, il est inutile de continuer de colorier une partie facile de l carte si on est en présence d'une impossibilité sur une autre partie de la carte. On peut gagner beaucoup de temps en détectant rapidement les impossibilités. Le parallèle est aussi frappant avec l'algorithme de recherche avec partition de M. Ginsberg [Ginsberg 1996]. Son algorithme consiste à mémoriser des sous ensembles importants de positions déjà cherchées et à les réutiliser ensuite pour évaluer les nouvelles positions similaires. Il obtient ainsi des gains très importants pour l'Alpha-Béta sur la résolution de donnes ouvertes au Bridge. Un autre problème pour lequel de telles méthodes sont utiles est Sokoban [Junhganns and Schaeffer 2001], dans lequel on peut détecter à l'avance des "deadlocks" qui empêchent de trouver une solution.

L'heuristique de commencer par le choix le plus contraint, est aussi une heuristique bien connue des systèmes de résolution de contraintes [Lauriere 1978], [Gent et al. 1996]. Alex Selby propose de valider théoriquement cette intuition en faisant les hypothèses suivantes : le facteur de branchement étant bp=f(p), p étant la profondeur, le nombre de nœuds de la recherche est égal à $N=1+b_0(1+b_1(1+b_2(1+...)))$. Pour un n et un r donnés, N peut se réécrire $N=b_n(K+b_{n+r}.L)$ avec K et L>0 dépendants des autres b_i. Si on peut faire varier b_n et b_{n+r} en gardant $b_n.b_{n+r}$ constant et les autres b_i inchangés (ce qui correspond a intervertir le choix de deux sites) alors, puisque $b_n(K+b_{n+r}.L)=K.b_n+$constante, pour minimiser N, on doit minimiser b_n (maximiser b_{n+r}). Ce raisonnement est bien conforme à l'heuristique qui consiste à choisir le site qui à le plus petit facteur de branchement en premier.

Le problème avec cette modélisation de la recherche est que l'arbre de recherche n'est pas aussi uniforme. Et que suivant le coup joué, les arbres de recherche suivants diffèrent beaucoup. De plus les sites d'Eternity ne sont pas vraiment indépendants. Par exemple, étant donné deux sites A et B. On a 3 façons de remplir A et 4 façons de remplir B. D'après l'heuristique précédente A est meilleur. Mais si après chaque remplissage de A on a encore plusieurs remplissages possibles alors qu'après 2 des façons de remplir B on est bloqué... alors B est peut être meilleur que A d'un facteur 2 ou 3 ! Il est tout de même coûteux d'avoir cette information puisqu'on doit examiner ce qui se passe après le pavage de A et de B. Toutefois c'est parfois utile de passer du temps à avoir une bonne approximation du "vrai" branchement (ça ne marche pas pour Eternity mais ça peut être utile pour d'autres problèmes...).

Il peut exister une meilleur façon de calculer le facteur de branchement bd que de faire une recherche à profondeur d et de compter les feuilles. C'est ce que nous allons voir maintenant.

Améliorations sur la force brute

L'idée consiste à associer à chaque pièce le nombre p de façons de paver chaque site (pour Eternity p≈0.03). Si on note b_i, le nombre de façons de paver un site après avoir placé i pièces. Pour un puzzle de N pièces (sur un région de taille N) avec des pièces similaires on a b_i =(N-i)p. Le nombre de solutions du puzzle est le produit pour i allant 0 à N-1 soit $N!t^N$. Il existe N tel que $N!t^N > 1$, ce qui nous permet de calculer la taille critique du puzzle au dessus de laquelle il y a des solutions. Soit s la taille critique du puzzle. Pour Eternity s ≈75. Or Eternity a 209 pièces ce qui est très supérieur à 75. Le nombre de solutions d'Eternity est proche de 10^{95} avec cette modélisation. On peut noter que pour N<s on a un puzzle sur-contraint. La meilleure méthode est alors la force brute. Alors que pour N>s le puzzle est sous contraint et on peut alors s'intéresser à des méthodes de type recherche locale ou métaheuristiques qui font une recherche plus ou moins aléatoire de l'espace de recherche.

Pour une taille N>s, le problème n'est pas plus dur que pour s, puisqu'on peut poser le N-s premières pièces et résoudre un problème de taille s. De plus on peut poser les N-s premières pour se mettre en position favorable. Reste a définir ce qu'est une position favorable. Ici on peut faire un analogie avec les problèmes de binpacking (remplissage d'un coffre de voiture par exemple) : on cherche à mettre en premier les formes bizarres , et à mettre les formes faciles à placer en dernier. De plus, à tout moment l'espace qui reste ne doit pas être trop fragmenté… (garder une bonne forme). Les mauvaises configurations auront beaucoup de trous alors que les bonnes configurations auront frontière convexe. L'algorithme de recherche consiste donc en deux phases. Une phase de remplissage qui tend à amener à des états favorables de taille relativement petite (≈ 30) aussi vite et bien que possible. Une seconde phase qui consiste à utiliser la force brute sur ces états. Pour des états suffisamment petits, il est payant de les explorer complètement. Le temps passé à les explorer est plus petit que le temps passé à les créer.

Pour Eternity un état favorable est :

- un état ayant de bonne pièces (facilement plaçables)
- un état ayant une belle frontière.

Une fois cette observation heuristique faite, il reste à la représenter avec des nombres. De plus, des améliorations portant sur des facteurs du second ordre comme par exemples les pièces qui marchent bien ensemble peuvent être ajoutées.

Estimation de la difficulté des pièces

Pour estimer la difficulté de trouver une solution avec une pièce, on peut trouver toutes les solutions à des problèmes de petite taille, et compter le nombre de fois où la pièce est présente dans les solutions. Reste à choisir la taille des petits problèmes à résoudre. Pour N=100, la recherche brute ne peut pas marcher car le puzzle est trop grand et impossible à explorer. Pour N=40, c'est encore trop difficile, de plus il n'y a pratiquement pas de solutions à cause de la taille critique d'Eternity. Il n'y a donc pas de taille vraiment utilisable avec cette méthode. On doit donc faire des approximations.

Pour approximer la difficulté des pièces, on peut faire un recherche exhaustive sur des états qui ont plus de pièces que la taille de la région. Ils auront beaucoup plus de solution, et cela donnera un estimation de la difficulté réelle de poser les différentes pièces. Par exemple, on peut essayer de paver des régions de taille 24 avec 90 pièces. La probabilité de pouvoir paver un région de taille 24 avec 24 pièces est de 10^{-17}. C'est donc en pratique impossible. En revanche, il y a $4x10^{21}$ façons de choisir 24 pièces parmi 90. On arrive donc à une résultat de 40000 solutions (en fait 40000/4=10000 solutions à cause de la parité). De plus le temps de recherche exhaustif pour placer 90 pièces sur une région de taille 24 est raisonnable (10 minutes pour des milliers de solutions). En répétant cette recherche des milliers de fois, on arrive en quelques semaines à approximer la difficulté des pièces.

Estimation de la difficulté des régions

Pour calculer la probabilité qu'une certaine région peut être remplie avec un certain ensemble de pièces, on utilise les probabilités calculées pour chaque pièce. La probabilité que la région soit pavée est le produit des probabilités des pièces pour un état (en fait A. Selby utilise les logarithmes des probabilités, ce qui lui permet de les additionner ce qui est moins coûteux que les multiplications). A ces probabilités associées aux pièces composant un état, on peut ajouter des paramètres mi sur la forme de la région. On a alors une évaluation de la probabilité de paver un état.

Algorithme de recherche

Il ne reste plus alors qu'à utiliser un algorithme de recherche qui va explorer en premier les états les plus prometteurs donnés par l'évaluation des états décrite aux sections précédentes. L'algorithme retenu par A. Selby est simple :

1) Décider du site le plus contraint
2) Engendrer tous les pavages de ce site
3) Prendre les n meilleurs (largeur=n)
4) Goto 1

La solution d'Eternity a été trouvée avec une largeur de 10 000 et un lookahead de 2.

Epilogue

Alex Selby et Olivier Riordan se sont donc partagé un million de Livres pour avoir été les premiers a avoir résolu Eternity. Il est possible que Christopher Monckton édite un deuxième puzzle de type Eternity, qui pourrait être doté d'un prix de 5 millions de £. Il est étonnant que la communauté IA ne se soit pas intéressée à ce problème qui rentre pourtant tout a fait dans son champ de recherche lorsque le premier puzzle Eternity est sorti. Il serait probablement intéressant de constituer un petit groupe de chercheurs en IA intéressés par le sujet si un autre puzzle de ce type est édité…

Bibliographie

[Gent et al. 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith and Toby Walsh: An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. CP 96, LNCS. pp 179-193, 1996.

[Ginsberg 1996] Ginsberg M. L. *Partition Search*. Proceedings de AAAI96.

[Junghanns and Schaeffer 2001] A. Junghanns and J. Schaeffer. Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge, *Artificial Intelligence 129 (1-2)*: 219-251 (2001).

[Lauriere 1978] J.-L. Lauriere, *A Language and a Program for Stating and Solving Combinatorial Problems*. Artificial Intelligence 10 (1978) 29-127.

[Selby 2001] Selby A. Notes from a talk. http://www.archduke.demon.co.uk/eternity/talk/notes.html. January 2001.

PROBLEMES LIES A L'ANALYSE D'UN ECHANTILLON DE PETITE TAILLE

Michel Masson
LIP6, Université Pierre et Marie Curie
masson@lip6.fr

Résumé:

Cet article analyse la validation des résultats produits par un système généralisant des échantillons de petite taille (gensam). Plusieurs méthodes appliquées dans des domaines voisins sont passées en revue; après avoir mis en évidence leur limite dans le cadre d'une transposition stricte, une méthodologie de la validation est proposée.

Mots-clés: validation, échantillon de petite taille, fouille de données, généralisation.

1. Introduction

Nous examinons dans cet article le problème posé par la validation du système GENSAM, ce système cherche à généraliser des observations obtenues à partir d'échantillons de petite taille [Masson 00]. Dans beaucoup de domaines, les cas observés sont en nombre limité, la méthode statistique est alors de portée limitée. GENSAM repose sur une analyse à plusieurs niveaux et sur l'utilisation d'heuristiques de généralisation. Construire des règles de pronostic illustre parfaitement cette approche comme le montre l'exemple suivant:

> On cherche à établir l'intérêt prédictif du prélèvement par lavage bronchoalvéolaire chez l'enfant souffrant d'infections respiratoires. Dans ce cas, le contrôle visuel ne peut être effectué en raison du faible diamètre de la trachée chez l'enfant. On dispose d'un ensemble de dossiers médicaux rassemblant des résultats d'examens biologiques, des données cliniques ainsi que des données techniques (diamètre de la sonde utilisée, durée de l'intubation, etc.). Chaque élément de l'échantillon est assorti d'un classement a posteriori (évolution du patient avec un recul de 6 mois). Il s'agit alors d'obtenir à l'aide des seules données de départ un diagnostic à 6 mois en utilisant une combinaison simple des informations de départ avec une marge d'erreur aussi faible que possible.

Dans ce type d'étude, on dispose d'un petit nombre de cas (environ 50) car les résultats sont liés aux techniques utilisées et à l'équipe qui les a mis en oeuvre. Il convient alors de valider l'ensemble des résultats obtenus : choix des critères d'études, comparaison avec d'autres méthodes. La spécificité du problème abordé - échantillon de petite taille - rend cette tâche délicate.

2. Généraliser un ensemble de données

Généraliser une petite série d'observations est à la croisée de plusieurs disciplines. GENSAM s'apparente à un problème de classification semblable à ceux étudiés en Analyse de Données mais avec des contraintes particulières. On a coutume de distinguer l'Analyse de Données supervisée - les données étudiées sont à repartir dans des classes connues à l'avance, en particulier on connaît une description des éléments de celles-ci - et l'Analyse de Données non supervisée. Dans ce dernier cas, la tâche de l'algorithme est de trouver des régularités ou d'extraire les règles les plus générales, le résultat de l'analyse pouvant être visualisé. Nous distinguerons un algorithme de classification et son résultat: une classification (i.e. la répartition par l'algorithme des éléments de l'échantillon dans les classes). D'autre part, il est usuel dans ce domaine d'appeler *variable* toute paire attribut-valeur.

2.1. Le problème de départ

Le terme échantillon évoque en premier lieu la méthode statistique: on veut établir l'effet d'un traitement, on examine alors deux échantillons parmi une population observée au travers d'une variable distribuée normalement. Le premier échantillon - échantillon témoin - recense des individus tirés au hasard dans la population; le deuxième, des individus auxquels le traitement est administré. Plusieurs techniques permettent de conclure sur l'efficacité du traitement à partir des différences observées sur les deux échantillons (ex. test de STUDENT).

L'approche suivie par le système GENSAM repose sur un seul échantillon: il ne peut prétendre généraliser la population concernée mais peut offrir un outil d'aide à la décision en synthétisant une série d'observations. Le risque de surspécification s'ajoute alors à la difficulté du problème.

GENSAM ne peut être comparé un système d'apprentissage par cas, car chaque cas représente alors un fragment de connaissances du domaine, or les éléments de l'échantillon soumis à GENSAM ne peuvent prétendre à aucune représentation, ce sont seulement des observations.

Une autre difficulté dans l'étape de validation provient des données bruitées: idéalement l'ensemble d'apprentissage doit contenir des éléments dont l'appartenance à une classe donnée peut être déterminée sans équivoque. Or il peut arriver qu'un ou plusieurs attributs des éléments de l'échantillon ne correspondent pas à leur classe d'appartenance (cf. § 5.3.1 - SPONGIA est un système de classification des éponges pour lequel aucune référence de classification incontestable n'existe), il faut alors déterminer dans quelle mesure cette situation perturbe la phase d'apprentissage.

Le domaine d'application joue un rôle important. Le système GENSAM, conçu au départ pour généraliser des échantillons liés au domaine médical est totalement indépendant du domaine d'application. Or la validation d'un tel système est étroitement liée à son champ d'application : la validation consistant à convaincre l'utilisateur de la pertinence des résultats obtenus, celle-ci va dépendre du type de lectorat dans de nombreux domaines.

Il apparaît que les travaux publiés dans des revues appliquées à la médecine (Artificial Intelligence in Medicine, JAMA, BJM, ...) ne font jamais apparaître les données brutes, d'autre part le traitement a priori (suppressions d'éléments jugés non pertinents dans le cadre de l'étude) est important, or ce dernier dépend du domaine et de l'expérience de celui qui conduit l'étude. Par contre, les articles publiés dans les revues scientifiques non spécifiquement médicales (Machine Learning, Expert Systems with Applications, Decision Systems,...) justifient longuement les procédures de validation utilisées.

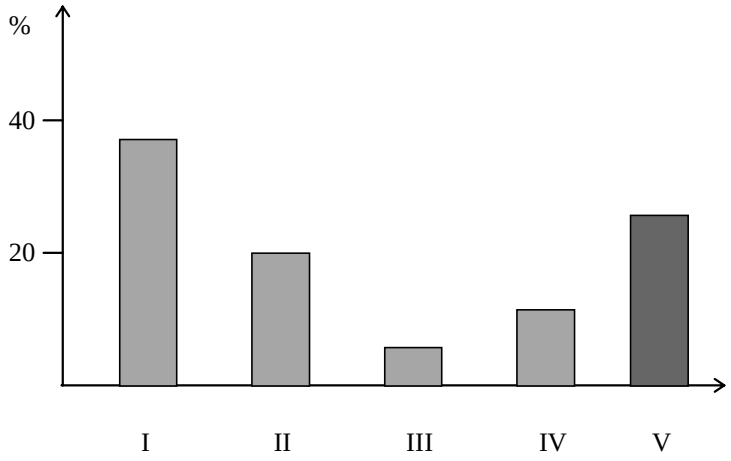
2.2. La validation des résultats médicaux

Dans un premier temps, il peut être fructueux d'étudier la mise en oeuvre de l'étape de validation des résultats obtenus dans le domaine médical en raison d'une littérature abondante. La validation repose alors sur un ou deux tests statistiques courants souvent mal utilisés; l'absence des données brutes (destinées à empêcher l'appropriation détournée des résultats) est alors un frein à l'adhésion du lecteur. Plusieurs travaux [Ross 51], [Badgley 61], [Schorr & Karten 66], [Gore & al. 77] ont montré que le nombre d'utilisations incorrectes de la méthode statistique a décru depuis les années 50 mais reste élevé.

Plusieurs études menées entre 1950 et 1977 ont étudié la pertinence des statistiques produites dans la

littérature médicale. Ces études n'ont examiné qu'une partie des articles publiés dans la période de référence, les barres verticales de la figure ci-dessus donnant l'intervalle de confiance, le nombre d'articles contenant des erreurs d'utilisation de la méthode statistique est très probablement compris dans ces intervalles.

Le test t est la procédure la plus utilisée dans la littérature médicale (cf. § 3), or il apparaît qu'il est souvent utilisé de façon inappropriée. La figure ci-après montre les résultats d'une analyse de son utilisation dans le numéro 56 de la revue circulation [Glantz 80], 142 articles ont été examinés (excluant les cas ne relevant pas de la méthode statistique), il apparaît un usage incorrect de la statistique t dans 39% des articles.



I Aucune analyse statistique II Analyse de la variance
II Test t appropriés IV Utilisation appropriée d'autres tests
V Utilisation incorrecte du test t pour des hypothèses multigroupes

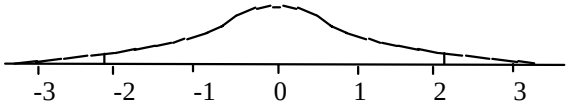
La nécessité de prendre en compte des données imparfaites, l'émergence de méthodes non statistiques (génération de règles, apprentissage par cas, réseau de neurones,...) ont conduit à mener différemment les procédures de validation.

3. Exemple d'utilisation de la méthode statistique

Le rôle de la méthode statistique est de permettent la validation d'une hypothèse de travail. Supposons que nous voulions mesurer l'effet d'un diurétique, pour cela nous prenons deux groupes parmi une population de référence (i.e. ses paramètres cliniques sont aléatoirement répartis); le premier groupe reçoit un placebo tandis que le second recevra le médicament. Il s'agit alors de comparer la production d'urine entre les deux groupes, l'écart entre les valeurs moyennes observées permet-il de conclure sur l'efficacité du traitement? Dans ce cas précis, le test de STUDENT (appelé encore statistique *t*) permet de répondre; pour cela on mesure le rapport :

$$t = \frac{\text{différences entre les moyennes des échantillons}}{\text{écart type de différences entre les moyennes des échantillons}}$$

Si le médicament n'a pas d'effet, on peut considérer que les deux échantillons sont tirés d'une même population et la valeur de *t* est proche de 1. Dans le cas contraire, une valeur élevée de *t* suggère que les deux échantillons ne sont pas extraits d'une même population (pour le critère concernant la production d'urine). En fait la valeur de *t* dépend du choix des échantillons dans la population, une valeur de *t* élevée peut être observé même si le médicament n'a pas d'effet; quelle est alors la probabilité d'une telle situation? Pour une population de 200 personnes, il y a 10^{27} façons différentes de construire des échantillons de 10 personnes, la répartition des valeurs de *t* est donnée par la courbe suivante :



Nous pouvons définir le pourcentage des valeurs de *t* élevées comme le pourcentage des surfaces grisées sur la figure : ainsi 5% des valeurs possibles de *t* sont supérieures à +2,086 et inférieures à -2,086. Autrement dit, en posant P = 0.05, P mesure la probabilité d'avoir tort en affirmant que la moyenne des échantillons sont différents alors qu'en fait ils sont tirés de la même population (on parle d'hypothèse nulle). En fonction de la taille des échantillons, le tableau suivant donne pour P donné, la valeur de *t* au-delà de laquelle l'hypothèse nulle est fausse avec une probabilité d'erreur P.

Table des valeurs critiques de *t*

v	0.50	0.20	0.10	0.05	0.02	0.01
1	1.000	3.078	6.314	12.706	31.821	63.657
2	0.816	1.886	2.920	4.303	6.965	9.925
3	0.765	1.638	2.353	3.182	4.541	5.841
4	0.741	1.533	2.132	2.776	3.747	4.604
5	0.727	1.476	2.015	2.571	3.365	4.032
. . .	. . .	. . .	. . .	. . .	. . .	. . .
20	0.687	1.325	1.725	2.08	2.528	2.845
. . .	. . .	. . .	. . .	. . .	. . .	. . .

4. Les techniques de base pour l'extraction de connaissances :

L'objet de GENSAM étant l'extraction des connaissances dans un échantillon, il semble intéressant d'examiner les techniques de validation utilisées dans les domaines voisins. La découverte de connaissances dans les bases de données (Knowledge Discovery in Data Bases) s'apparente en partie au fonctionnement de GENSAM, elle se compose de six étapes : délimitation du domaine, construction de l'ensemble des données, traitement a priori, fouille de données (data-mining), traitement a posteriori et validation. L'étape data-mining peut être résolue par la construction d'arbres de décision, la génération de règles symboliques, l'utilisation de méthodes statistiques ou les réseaux de neurones.

4.1. Construction d'arbres

L'algorithme ID3 [Quinlan 86] est le premier d'une famille d'algorithmes générant un arbre de classification: C4.5 [Quinlan 93], ASSISTANT [Cestnik & al. 87], ASSISTANT-R [Kononenko 94]. 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.

4.2. Génération de règles

Elle consiste à construire incrémentalement un ensemble de règles permettant de classifier un ensemble de données. Soit $E = \{e_1, ... ,e_n\}$chaque e_i étant défini par un ensemble de paires attribut-valeur et d'un attribut particulier C dont la valeur est une classe d'appartenance. On cherche à construire des règles de la forme :

R_k Ex.

SI	**IF**
A_{k1} **a** v_{k1}	**Age > 46**
.....	**Number_of_painful_joints > 3**
A_{kp} **a** v_{kp}	**Skin_manifestations = psoriasis**
ALORS	**THEN**
C = c_k	**Diagnosis=Crystal_induced_synovitis**

α : =, <, >

A l'issue de ce processus, on obtient un ensemble R de règles. Etant donné un élément à classifier, on teste successivement les règles de R, la première règle dont les conditions sont satisfaites donne la classe

d'appartenance de l'élément. Si aucune règle ne convient, une règle supplémentaire classe l'élément (en général dans la classe comportant le plus d'éléments). Les algorithmes ainsi développés - AQ15 [Michalski & al. 86], CN2 [Clark 89] - opèrent règle par règle. Ils construisent incrémentalement la règle jusqu'à ce que sa précision soit correcte.

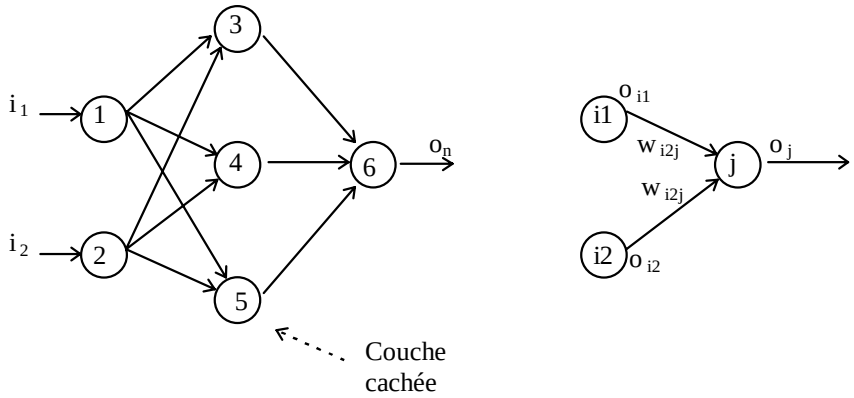
4.3. Les méthodes statistiques

Elles sont nombreuses et leur usage est bien codifié. Parmi les plus simples, on trouve la régression [Tomassone 88a], l'analyse discriminante [Tomassone 88b]. La classification bayésienne est très répandue [Kononenko 01], pour tout élément à classer dont les valeurs des attributs sont v_1, ... , v_p , sa probabilité d'appartenance à la classe C est donnée par $P(C \mid v_1 \mathrm{K}\ v_p) = P(C) \prod_i \frac{P(C \mid v_i)}{P(C)}$ où P(C) désigne la probabilité a priori (prévalence de la classe C) et P(C | v_i) la probabilité d'appartenir à C si le $i^{ème}$ attribut vaut v_i. La méthode impose une contrainte forte: l'indépendance des attributs.

La méthode des k plus proches voisins (k-nearest neighbors method) [Kononenko 01] est également utilisée pour obtenir une classification. Pour tout nouvel élément à classer, la méthode consiste à calculer les k plus proches voisins, la classe du nouvel élément correspond à la classe la plus fréquente parmi ces k éléments (ce choix peut être pondéré par la distance). En fonction des valeurs prises par les variables, l'utilisateur choisit la métrique la plus appropriée: distance euclidienne, distance des blocs, distance de Mahalanobis (cf. § 5.3.1).

4.4. Les réseaux de neurones

Un réseau de neurones est constitué d'un ensemble de noeuds (neurones) reliés entre eux, chaque connexion est affectée d'un poids w_{ij} renseignant sur la force du lien entre les neurones i et j (une valeur négative correspond à une inhibition). La plupart des auteurs utilisent des réseaux multicouches - les neurones d'une couche sont reliés aux neurones des couches adjacentes - en limitant leur nombre à trois [Bhattacharyya & Pendharkar 98]. Le nombre de neurones de la couche d'entrée est égal au nombre d'attributs utilisés pour décrire l'échantillon, le nombre de neurones de la couche intermédiaire comprend 2 à 3 fois le nombre de neurones de la couche précédente, le nombre de neurones de la couche de sortie correspond au nombre de classes à identifier. Un neurone dispose d'une ou plusieurs entrées - correspondant aux sorties des neurones auxquels il est relié - et d'une sortie déterminée par: $o_k = f(\sum_i w_{ik} o_i + \theta_k)$, f est la fonction logistique, sa forme usuelle est $f(x) = 1/(1+e^{-x})$ car sa dérivée peut être exprimée à l'aide d'elle-même.



Au départ les poids w_{ij} sont répartis aléatoirement, un algorithme - la rétropropagation - ajuste progressivement ces poids en cherchant à diminuer l'écart E entre les sorties attendues t_n et les sorties observées o_n (n parcourant l'ensemble des noeuds de la couche sortie): $E = \frac{1}{2} \sum_n \| t_n - o_n \|^2$. Un premier ensemble - ensemble d'apprentissage - permet de fixer les poids, le réseau est ensuite testé sur un second ensemble de cas - ensemble test - pour la validation des résultats.

4.5. Critères pour évaluer la performance d'un algorithme:

Pour simplifier, nous supposerons que l'échantillon à analyser comprend m éléments répartis en deux classes. Posons c_{ij} = nombre d'éléments de la classe C_i classés dans la classe C_j par l'algorithme (i,j = 1, 2).

Le critère le plus fréquemment utilisé pour évaluer un algorithme est le critère **précision**, il mesure le nombre d'éléments correctement classés: **précision** = $(c_{11} + c_{22}) / m$

Un supplément d'information peut être apporté par les critères **sensibilité** et **spécificité**:

$$\textbf{sensibilité}\ (C_i) = c_{ii} / \sum_j c_{ij} \qquad \textbf{spécificité}\ (C_i) = c_{ii} / \sum_j c_{ji}$$

La sensibilité mesure l'aptitude de la classification à isoler les éléments de C_i tandis que la spécificité renseigne sur le proportion d'éléments mal classés.

On peut vouloir prendre en compte la prévalence des classes grâce au **score informationnel** d'une classification. Soit e_i un élément de l'échantillon E, soit C sa classe d'appartenance, P_a (C) la probabilité a priori de C et P(C) la probabilité donnée par l'algorithme de classification (ex. classification bayésienne), alors le score informationnel de e_i est:

$I(e_i) = - \log P_a(C) + \log P(C)$ si $P_a(C) \leq P(C)$

$I(e_i) = \log (1- P_a(C)) - \log (1 - P(C))$ si $P_a(C) > P(C)$

(il mesure le gain d'information apporté par l'algorithme de classification comparativement à la probabilité à priori)

Le **score informationnel** de l'échantillon E est donné par: $I(E) = \frac{1}{m} \times \sum_{k=1}^{m} I(e_k)$

Un autre critère important est la mesure de la dispersion des résultats en fonction de la taille des classes: si une classe est de taille relative importante, les chances de classer correctement un élément dans cette classe sont élevées, cela augmente donc artificiellement la qualité de la classification. Cette remarque est prise en compte avec la statistique KAPPA qui est analysée en détail au § 5.3.2.

5. Méthodes de validation

Si aucune méthode de validation n'est directement transposable à GENSAM, il existe un moyen - la procédure leave-one-out - permettant de valider les données qui lui sont soumises: le leave-one-out permet de tester la sensibilité de l'algorithme par rapport à chaque observation.

5.1. La procédure leave-one-out

Soit n le nombre de classes ventilées par l'algorithme, soit m le nombre d'éléments de l'échantillon. La procédure leave-one-out consiste à travailler sur m échantillons de m - 1 éléments et à comparer les résultats obtenus. L'algorithme comprend donc m étapes:

étape 1: on teste l'algorithme sur e_2 e_3 .... e_m

étape 2: on teste l'algorithme sur e_1 e_3 .... e_m

.....

étape m: on teste l'algorithme sur e_1 e_2 e_3 .... e_{m-1}

E_{-k} désigne la classification correspondant au traitement de l'échantillon privé de l'élément k, on utilise un indicateur de score (par exemple la précision) mesurant le pourcentage d'éléments bien classés pour comparer ces m classifications. Cet indicateur est obtenu à partir d'un tableau de contingences c_{ij} (i,j ∈ [1, n]), c_{ij} ayant pour valeur le nombre d'éléments classés par l'algorithme dans la classe j et appartenant en fait à la classe i. Il peut être calculé pour chaque E_{-k} et pour l'ensemble des éléments de l'échantillon:

$$P(E_{-k}) = \sum_{i=1}^{n} c_{ii} / (m-1) \qquad P(E) = \sum_{i=1}^{n} c_{ii} / m$$

alors la quantité $\sum_{k=1}^{m} [P(E_{-k}) - P(E)]^2$ renseigne sur la dispersion des résultats obtenus.

Si la procédure leave-one-out permet de tester la sensibilité de l'algorithme de classification et l'homogénéité de l'échantillon étudié, elle n'est en aucun une aide à la validation de la démarche empruntée.

5.2. La validation par comparaison de méthodes:

Il peut être intéressant de valider une méthode nouvelle ou une amélioration de méthode existante en la comparant à des méthodes éprouvées. Beaucoup d'articles comparent ainsi la performance des réseaux neurones avec la méthode d'extraction de règles, l'analyse discriminante et/ou des techniques plus originales [Lavrac 99]. Outre les difficultés liées à l'harmonisation des données, se pose le problème de l'extrapolation des résultats comme les exemples suivants vont le montrer.

5.2.1. Comparaison réseaux de neurones, analyse discriminante, programmation mathématique:

Les auteurs de cette étude [Pendharkar & al. 99], ont voulu comparer les résultats de ces méthodes dans la prédiction du cancer du sein à partir d'un échantillon regroupant 454 patientes. La validation repose sur la comparaison du critère **précision** dans les trois méthodes en s'appuyant sur un partitionement de l'échantillon de départ : échantillon d'apprentissage / échantillon test. La taille de l'échantillon d'apprentissage pouvant influencer la qualité des résultats, il a été procédé à 3 décompositions apprentissage / test de l'échantillon de départ: 25% / 75%, 50% / 50% et 75% / 25%, chaque décomposition étant effectuée aléatoirement.

La programmation mathématique fait appel à un algorithme d'enveloppement (à chaque classe est associée une courbe enveloppant les éléments de celle-ci). Le tableau suivant donne les résultats sur la **précision** en fonction de la répartition échantillon de travail / échantillon test:

	25/75(%)	50/50(%)	75/25(%)
Réseaux neurones	81.6	81.5	81.5
DEA	62.1	66.5	67
Analyse Discriminante	60.5	66.1	65.6

L'approche par réseaux neurones donne les meilleurs résultats (on remarquera que la répartition a peu d'incidence sur les résultats et que la validation utilise le seul critère **précision**).

5.2.2. Comparaison réseaux de neurones et régression

Dans cette étude, les auteurs [SubbaNarasimha & al. 00], partis d'un a priori en faveur des réseaux de neurones, ont voulu vérifier cette hypothèse dans des domaines liés à la gestion des ressources humaines. Ils ont comparé les résultats obtenus à partir d'un réseau de neurones avec ceux obtenus à l'aide des outils statistiques classiques (en particulier la régression) en tenant compte de la présence de données biaisées. Les modèles basés sur la régression imposent aux variables, outre leur indépendance, de suivre une distribution normale. Sachant que le non respect de ces conditions altèrent les résultats [Marquez & al. 91], les auteurs ont voulu tester l'efficacité des réseaux de neurones sur ce point.

La procédure opératoire est classique: après sélection de quatre échantillons de 50 éléments chacun, deux sont utilisés pour l'apprentissage (régression et réseau de neurones), les deux autres permettent de tester les modèles ainsi obtenus. Les variables non distribués normalement étaient corrigées (élévation au carré, inverse, logarithme, etc.), la correction la plus efficace était alors appliquée. La comparaison des performances utilisait le pourcentage d'erreurs de classement (une variante du critère précision). Les résultats (cf. tableau) sont en faveur de la régression, l'écart étant statistiquement significatif à hauteur de 2% pour l'un des exemples.

Mesure du pourcentage d'erreur			
	Régression	Réseau de Neurones	Signification Stat.
Exemple 1	16.72	23.61	< 0.10
Exemple 2	19.68	25.23	< 0.02

En fait, il ressort que les réseaux de neurones, en raison de leur capacité d'auto-adaptation, proposent une alternative à la transformation initiale des données imposée par la méthode statistique.

5.3. Visualiser pour valider

Dans certains cas la visualisation des résultats peut s'avérer être une forme de validation. L'analyse de données a popularisé une méthode de représentation: les dendrogrammes.

5.3.1. Visualisation d'une classification à l'aide de dendrogrammes:

Le système SPONGIA [Domingo & al. 99] est un système à base de connaissances spécialisé dans l'identification des éponges, ce domaine est complexe : identifier une éponge revient à déterminer son espèce à l'intérieur d'un genre, lui-même élément d'une famille, les familles étant regroupées dans un ordre, les ordres étant rattachés à des classes. Un processus classique de validation consisterait à rassembler des espèces aléatoirement sélectionnées dans les zones spécifiques à celles-ci, cette démarche - d'un coût très élevé - a dû être abandonnée. Les auteurs ont choisi une méthode originale : ils ont soumis à un groupe d'experts réputés la description (sous forme de paires attribut-valeur) de chaque spécimen à tester. L'ensemble des informations ainsi collectées constitue une matrice cubique m_{ijk} :

i ∈ {experts} SPONGIA

j ∈ {nom de classification}

k ∈ {numéro du spécimen}

La valeur m_{ijk} exprime la note de confiance (∈ [0,1]) de l'expert i pour l'appartenance du spécimen k à l'espèce j. N désigne le nombre de classes.

L'analyse hiérarchique permet de mesurer les performances des experts en utilisant une distance pour évaluer leurs résultats. Quatre distances sont utilisés pour cette analyse :

- distance euclidienne :

$d_k^e(a,b) = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(m_{ajk} - m_{bjk})^2}$ exprime la distance entre les experts a et b, concernant le spécimen k

- distance des blocs (distance de Manhattan) :

$d_k^M(a,b) = \frac{1}{N}\sum_{j=1}^{N}\left| m_{ajk} - m_{bjk} \right|$ elle permet de mieux séparer 2 experts dont les jugements diffèrent peu

- distance de Mahalanobis: cette distance pondère la distance entre 2 jugements en prenant en compte le niveau hiérarchique de différence : une erreur sur le genre est moins grave qu'une erreur sur l'ordre.
- indice de concordance: cet indice mesure la concordance des réponses fournies par les experts en prenant en compte une concordance due au hasard. Il est basé sur la statistique KAPPA (cf. § suivant).

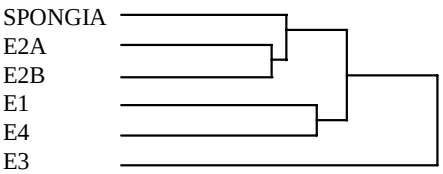
Un algorithme de regroupement permet de constituer des groupes dont les écarts de jugement sont faibles. Pour cela, il faut pouvoir estimer la distance d'un groupe quelconque r à un groupe t obtenu par regroupement des groupes g' et g'':

$$d(t,r) = \frac{n_{g'}}{n_{g'} + n_{g''}} d(g',r) + \frac{n_{g''}}{n_{g'} + n_{g''}} d(g'',r)$$ où $n_{g'}$ ($n_{g''}$) désigne le cardinal de g' (g'')

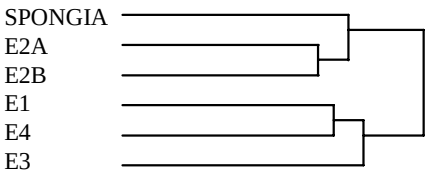
Au départ les groupes sont constitués de singletons, on construit un premier groupe constitué par les 2 éléments les plus proches, puis les regroupements deux par deux sont poursuivis jusqu'à obtenir un seul

groupe. Ces regroupements successifs peuvent être représentés graphiquement à l'aide de dendrogrammes: la distance horizontale entre deux regroupements est d'autant plus faible que ces deux regroupements sont proches. En utilisant les quatre distances définies précédemment, on obtient les figures :

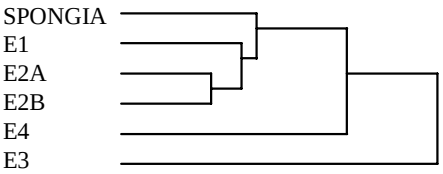
Dist. Euclidienne



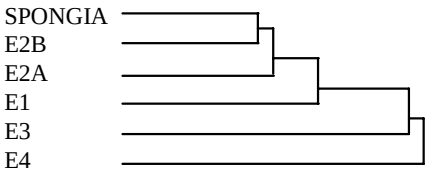
Dist. de Mahalanobis



Dist. des Blocs



Indice de concordance



On constate que SPONGIA occupe la première place devant les experts et que SPONGIA est toujours plus proche d'un expert humain que ceux-ci ne sont proches entre eux. Les auteurs en déduisent la validation du système SPONGIA.

5.3.2. Visualisation de la dispersion des classifications

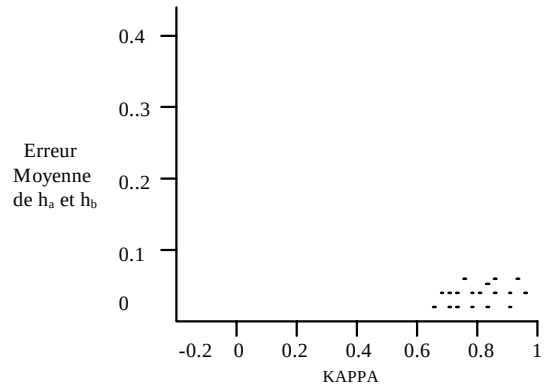
Une autre méthode de comparaison visuelle des performances des algorithmes de classification a été proposée par [Dietterich 00], elle utilise la statistique KAPPA. Chaque algorithme est visualisé par un nuage de points dans un espace à deux dimensions. Comparer deux méthodes revient alors à comparer des figures. Malheureusement, seuls les critères **précision** et **éparpillement** sont pris en compte.

La méthode part d'un algorithme de classification (C4.5, ID3, ...) et analyse la qualité des classifications obtenues. Soient n le nombre de classes et m le nombre d'objets à classer. A chaque paire de classifications h_a et h_b, on fait correspondre un point dont l'ordonnée est la moyenne des précisions obtenues par h_a et h_b et l'abscisse est la valeur d'un indice mesurant la dispersion des résultats de ces classifications (cette mesure de la dispersion est communément appelée statistique KAPPA). Cette statistique peut être définie à partir du tableau de contingences $\{C_{ij}\}$ i,j =1,...,n . C_{ij} représente le nombre d'éléments classés dans C_i par h_a et dans C_j par h_b.

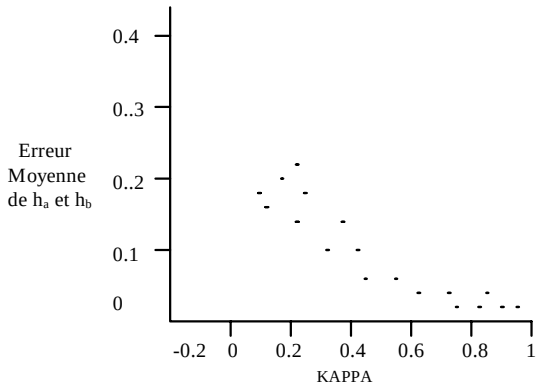
Posons $\Theta_1 = \dfrac{\sum_{i=1}^{n} C_{ii}}{m}$ $\qquad$ $\Theta_2 = \sum_{i=1}^{n}\left(\sum_{j=1}^{n}\frac{C_{ij}}{m}\cdot\sum_{j=1}^{n}\frac{C_{ji}}{m}\right)$

$\Theta_1 \in [0,1]$, il mesure la probabilité que h_a et h_b concordent dans leurs résultats. Θ_2 mesure la probabilité de concordance des classifications h_a et h_b; $\sum_{j=1}^{n} C_{ij}/m$ (resp. $\sum_{j=1}^{n} C_{ji}/m$) représente la probabilité pour un exemple d'être affecté dans la classe i par la première (resp. deuxième) classification. La statistique KAPPA est définie par $K=\dfrac{\Theta_1-\Theta_2}{1-\Theta_2}$, K diminue lorsque la probabilité que les deux classifications coïncident augmente.

Pour un algorithme de classification donné, on fait correspondre un point de l'espace précédemment défini pour chaque paire de classifications opérées par l'algorithme; à ce dernier correspond alors un nuage de points. Les auteurs utilisent cette méthode pour comparer deux variantes de l'algorithme de classification C4.5, on obtient les deux répartitions suivantes:



Bagged C4.5



Adaboosted C4.5

Un nuage de points situé essentiellement en bas et à droite du diagramme indique une bonne précision et une faible dispersion des classifications effectuées par l'algorithme. Les auteurs en concluent que l'algorithme Bagged C4.5 est plus performant que l'algorithme Adaboosted C4.5.

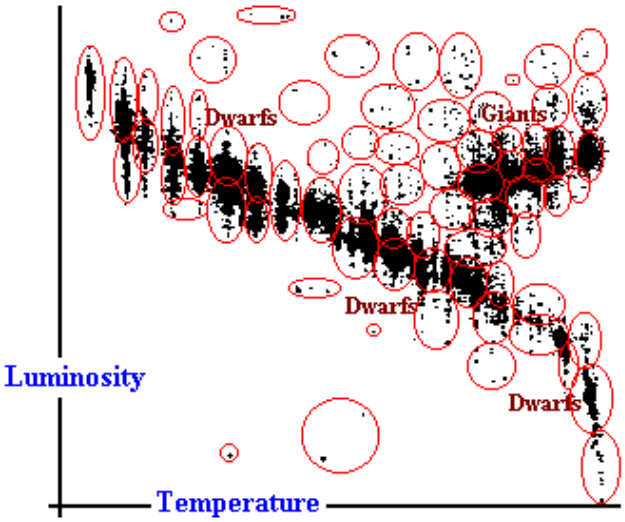
5.3.3. Modifier l'échantillon pour mieux analyser visuellement

La plupart des travaux mentionnés ici font l'objet d'un traitement préalable destiné à supprimer de l'échantillon les éléments non conformes. Cet étape peut être automatisée pour faciliter un affichage par des dendrogrammes (comme ceux utilisés dans le système SPONGIA). D. Wishart [Wishart 86] propose une amélioration de la méthode des k moyennes - une méthode d'analyse hiérarchique - prenant en compte des données discrètes, continues ou manquantes et permettant d'ignorer des éléments isolés. Le logiciel CLUSTAN est basé sur cette technique [www.clustan.com].

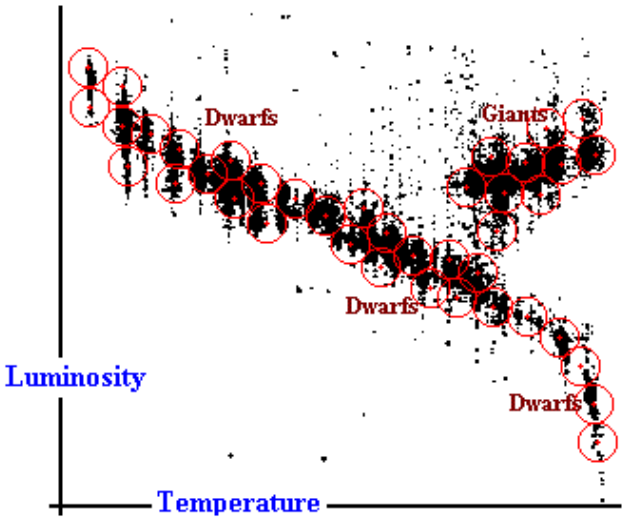
La méthode des k moyennes est une méthode de classification supervisée; elle cherche à répartir les éléments d'un échantillon en k classes (k étant fixé au départ). Chaque élément d'une classe est plus proche des éléments de la classe à laquelle il appartient que de la moyenne des éléments de toute autre classe. Pour cela on définit une distance prenant en compte ces différents types de variables. D. Wishart modifie l'algorithme de construction des classes en introduisant un seuil d'isolement : pour tout élément on calcule la plus petite des distances moyennes aux autres classes, l'élément est supprimé si cette distance est supérieur à δ. L'algorithme opère ainsi:

k donné, fixer un seuil d'isolement δ

1) partitionner les éléments en k regroupements (aléatoirement)
2) pour chaque élément, calculer la plus petite distance d avec la moyenne des regroupements
   Si $d > \delta$, oublier l'élément
   Sinon affecter l'élément au regroupement le plus proche
3) recommencer en 2) tant qu'un changement est intervenu



Sans seuil d'isolement



Avec seuil d'isolement

La figure ci-dessus montre le résultat, tantôt sans utilisation de seuil, tantôt en utilisant un seuil (l'exemple analyse de familles d'étoiles en fonction de leur luminosité et de leur température). La distance ainsi définie peut être utilisée pour construire un dendrogramme séparant parfaitement deux familles: étoiles naines / étoiles géantes.

Quand la validation repose sur la mise en évidence de plusieurs classes, l'élimination (paramètrable grâce à la valeur δ) des éléments isolés facilite la comparaison visuelle de plusieurs solutions.

6. Conclusion

La façon dont ont été résolues les difficultés rencontrées dans toute étape de validation permet de fixer une ligne directrice. L'optimisation des critères de classification (précision, concordance,...) est moins importante que les connaissances mises à jour. La pertinence de ces connaissances est décisive dans l'adhésion de l'utilisateur. L'échantillon, en raison de sa taille, peut ne pas être représentatif du problème de départ; la visualisation des résultats en offrant une grande quantité d'informations, facilite l'interactivité du processus de validation. Il en résulte la possibilité de pouvoir confronter plusieurs types de connaissances à partir d'un même ensemble de données.

Bibligraphie

[AuteurA1 99] AuteurA1 P., AuteurA2 P. : *Titre de l'article* – titre recueil, numéro recueil, pages, éditeur, année.

[Badgley 61] Badgey R.F.: *An Assessment of Research Methods in 103 Scientific Articles from Two Canadian Medical Journal* - Can M.A.J. , 85, 256-260, 1961.

[Bhattacharyya & Pendharkar 98] Bhattacharyya S., Pendharkar P.C., *Inductive, evolutionary and neural techniques for discrimination: a comparative study* - Decision Science, 28-4, 1998.

[Cestnik & al. 87] Cestnik B., Kononenko I., Bratko I.: *ASSISTANT 86: A knowledge elicitation tool for sophisticated users* - Progress in machine learning, Sigma Press, 1987.

[Clark 89] Clark P., Niblett T.: *The CN2 Induction Algorithm* - Machine Learning, 3, 261-283, 1989.

[Dietterich 00] Dietterich Th., *An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting and Randomization* - Machine Learning, 40, 139-157, 2000.

[Domingo & al. 99] Domingo M., Martin-Baranera M., Sanz F., Sierra C., Uriz M.J.*: Validating SPONGIA, an expert system for sponge identification* - Expert Systems with Applications, 16, 379-384, 1999.

[Glantz 80] Glantz S.A.: *How to Detect, Correct and Prevent Errors in the Medical Literature* - Circulation, 61, 1-7, 1980.

[Gore & al. 77] Gore S., Jones I.G., Rytter E.C.: *Misuses of Statistical Methods: Critical Assessment of Articles in B.M.J. from January to March, 1976* - B.M.J., 1-6053, 85-87, 1977.

[Kononenko 94] Kononenko I.: *Estimating attributes: analysis and extensions of RELIEF* - Proceedings of the Europeab Conference on Machine Learning, Springer, 171-182, 1994.

[Kononenko 01] Kononenko I., *Machine learning for medical diagnosis: history, state of the art and perspective* - Artificial Intelligence in Medicine, 23, 89-109, 2001.

[Labenne & Goldfarb 98] Labenne M., Goldfarb B,. *Blind-protected specimen brush and brochoalveolar lavage in ventilated children* - Critical Care, 1998.

[Lavrac 99] Lavrac N., *Selected techniques for data mining in medicine* - Artificial Intelligence in Medicine, 16, 3-23, 1999.

[Margineantu] Margineantu D., Dietterich Th., *Pruning Adaptative Boosting* - ,211-218,

[Marquez & al. 91] Marquez L., Hill T., Worthley R., Remus W.*: Neural networks as an alternative to regression* - Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences, 4, 129-135, 1991.

[Masson 00] Masson M.: *GENSAM: un système généralisant les petits échantillons* - Colloque sur la Métaconnaissance, Cahiers du LIP6, Berder, 2000.

[Michalski & al. 86] Michalski R.S., Mozetic I., Hong J., Lavrac N.: The multipurpose incremental learning system AQ15 and its testing application to three medical domains - Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, 1041-1045, 1986.

[Pendharkar & al. 99] Pendharkar P.C., Rodger G.J., Herman N., Benner M.: *Association, statistical, mathematical and neural approaches for mining breast cancer patterns* - Expert Systems with Applications, 17, 223-232, 1999.

[Quinlan 86] J. R. Quinlan: *Induction of decision trees* - Machine Learning, 1:81-106, 1986.

[Quinlan 93] J. R. Quinlan: *Programs for Machine Learning* - Morgan Kaufmann, 1993.

[Ross 51] Ross O.B.: *Use of Controls in Medical Research* - JAMA, 145, 72-75, 1951.

[Schorr & Karten 66] Schorr S., Karten I.: *Statistical Evaluation of Medical Journal Manuscripts* - JAMA, 195, 1123-1128, 1966.

[SubbaNarasimha & al. 00] SubbaNarasimha P.N., Arinze B., Anandarajan M. *: The predictive accuracy of artificial neural networks and multiple regression in the case of skewed data: exploration of some issues* - Expert Systems with Applications, 19, 117-123, 2000.

[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.

[Wishart 86] Wishart D.: *Hierarchical cluster analysis with messy data* - Classification as a Tool of Research, 453-460, Gaul Schader, Amsterdam, 1986.