Rapport de stage scientifique

Planification non linéaire

Camille Roth
élève-ingénieur de première année

Avril à Juillet 1999

Résumé

Après une brève présentation des enjeux et des possibilités de la recherche en planification, ce rapport propose plus particulièrement d'exposer les bases théoriques de la planification non-linéaire, et les concepts qui en découlent : langage de modélisation de l'environnement, notion de plan... On présentera ensuite de façon détaillée le fonctionnement d'une application concrète des résultats précédents, selon un plan à peu près chronologique (suivant la réalisation du programme en `C` au cours du stage). Enfin, on s'interrogera sur la façon dont on pourrait poursuivre et améliorer ce travail de recherche, à travers des idées qui n'auront pas pu être abordées au fil de ces trois mois.

Table des matières

Eléments de contexte

Le stage s'est déroulé au sein du pôle IA (Intelligence Artificielle) du LIP6, Laboratoire d'Informatique de l'université de Paris VI (Université Pierre et Marie Curie), sous la direction de Jean-Marc Labat. Le LIP6, dont le pôle IA est un des plus importants centres de recherche en IA de la région parisienne, est rattaché au CNRS et est évidemment un organisme public. Il réunit la majeure partie de la recherche en informatique à Paris VI et est structuré en neuf thèmes, dont celui dans lequel j'ai effectué mon stage, à savoir le thème SYSDEF, SYStèmes d'aide à la DEcision et à la Formation. L'aide à la décision est une discipline qui tente de représenter informatiquement des situations décisionnelles et d'y apporter des solutions satisfaisantes. L'aide à la formation consiste en des applications qui tentent de dialoguer avec l'utilisateur à la manière d'un véritable professeur : l'ordinateur doit expliquer ce qui n'apparaît pas clairement, et surtout comprendre ce qui est faux dans la démarche de l'élève.

Introduction

L'objectif de ce stage, que j'ai effectué en monôme, est la conception d'un planificateur non-linéaire. Il convient bien entendu de préciser d'abord ce qu'on entend par *planificateur*, et d'une manière plus générale, préciser ce qu'est la *planification* en intelligence artificielle. La non-linéarité est une notion que j'éclaircirai par la suite, et qui décrit la manière de concevoir le planificateur.

Planifier n'est en rien de l'ordonnancement. Il s'agit, d'abord, d'un problème d'interaction. A partir d'une description d'un environnement dans un état initial, en fournissant une description de l'état final désiré et des opérateurs qui permettent d'agir sur ce même environnement, le planificateur doit trouver un enchaînement d'actions qui permettent de passer de l'état initial à l'état final. L'exemple de la figure 1 représente une Table, trois cubes A, B et C. Au début, C est posé sur A, et A et B reposent tous deux sur la Table. On désire aboutir à la situation finale, empilement successif de C sur B, sur A. On se donne un opérateur qui permet de dépiler un cube sur la Table, et un autre qui permet d'empiler un cube sur un autre. Le problème de l'interaction est alors posé : le planificateur doit trouver une successions d'actions qui permette d'obtenir le résultat voulu (il doit ainsi trouver, par exemple, "*dépiler C*, *empiler B sur C*, puis *empiler A sur B*").

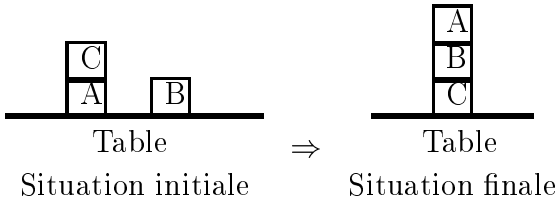


Fig. 1 – Exemple de problème d'interaction.

Le problème de l'interaction est une composante essentielle de la planification, puisque trouver intelligemment un plan qui fonctionne (sans explorer toutes les possibilités, ce qui deviendrait de toutes façons rapidement irréalisable) n'est jamais aisé, et parfois impossible.

Il est à présent possible de préciser les enjeux : planifier, c'est non seulement résoudre le problème de l'interaction, mais aussi se donner des contraintes dont le plan construit par le planificateur tienne compte. Ainsi, il peut exister plusieurs plans qui permettent de passer d'un état à un autre (ce sont des solutions du problème de l'interaction), mais on peut vouloir que le planificateur nous donne celui qui est par exemple le plus simple à mettre en œuvre (c'est la contrainte). Un problème de planification est donc à la fois un problème d'interaction et une manière de résoudre ce problème d'interaction.

Maintenant que le problème est posé, il s'agit de trouver comment un ordinateur pourrait modéliser une situation, se représenter des actions et leurs effets, "raisonner" à partir de toutes ces données, et enfin nous répondre que pour arriver à l'état final, il suffit d'exécuter telle ou telle succession d'actions.

Applications

Le domaine de cette recherche se situe clairement du côté de l'aide à la décision, puisqu'on fait intéragir une situation (l'environnement) et un décideur (le planificateur). Les applications sont multiples, par exemple dans la conception de jeux intelligents, comme le bridge. Des chercheurs américains [5] ont ainsi mis au point un algorithme de bridge s'appuyant sur des techniques de planification (il s'agit d'un planificateur un peu particulier où les situations sont incomplètement décrites, puisque l'ordinateur n'est pas censé connaître le jeu de l'adversaire). Pour tester ses performances, ils ont décidé de le faire jouer contre un des meilleurs jeux de bridges du marché, *Bridge Baron*, qui utilise des algorithmes classiques (base de données, $\alpha\beta$, ...). Ils ont alors constaté après 1000 parties que leur algorithme avait gagné 254 matchs contre 202 (avec 544 matchs nuls), sans même avoir d'autres parties déjà jouées en mémoire. Le gain de performance est tel que la firme qui commercialise *Bridge Baron* a décidé d'y incorporer cet algorithme de planification.

Dans un domaine plus vaste encore, la planification constitue un enjeu majeur de la toute jeune industrie cybernétique. Avec TRIPS [2], des chercheurs de l'université américaine de Rochester ont créé un système intelligent intégré. *Intégré* signifie que TRIPS réunit et fait interagir divers systèmes intelligents, diverses applications de la recherche en IA. Ainsi, il contient un interpréteur de langage naturel[1] qui va permettre d'obtenir des informations de l'utilisateur dans son langage courant (cf. figure 2) puis de les traiter pour que d'autres systèmes intelligents, par exemple un planificateur, puissent aider l'utilisateur. TRIPS contient aussi un générateur de parole, et un algorithme de reconnaissance de voix, de fait, on peut directement dialoguer avec l'ordinateur, comme avec un être humain. Dans l'exemple, il s'agit d'une application d'optimisation de déplacements de personnes, sur une île. On voit que l'homme intéragit de façon tout à

[1]Par opposition au langage informatique, constitué de commandes et d'arguments, le langage naturel est le langage de tous les jours, avec ses nuances, ses ellipses, ses exceptions, etc...

fait naturelle avec la machine, et si l'exemple reste basique, rien n'empêche d'utiliser un planificateur sur des problèmes bien plus complexes, pour peu que celui que l'on inclura dans TRIPS soit suffisamment performant pour les résoudre. A terme, il sera possible de trouver une multitude d'applications fondées sur une telle intégration de "services" d'intelligence artificielle. On pourra alors peut-être arriver à la quasi-ultime réalisation en intelligence artificielle : un robot capable de comprendre, de parler, et de décider ; bref, de se comporter à peu près comme un être humain, du moins d'un point de vue décisionnel.

Homme : Montre-moi une carte de l'île.
Machine : D'accord.

(affiche la carte, avec les cinq points A, B, C, D, E et les différentes routes les reliant)

Il y a deux groupes en A, deux en B, deux en C et un en D.

(affiche une fenêtre avec le nombre de personnes en chaque point)

Il y a deux camions et un helicoptère en E.

(affiche une fenêtre avec les véhicules et leur utilisation)

Homme : Comment déplacer les groupes de B en E le plus efficacement possible ?
Machine : *(affiche les déplacements)*
Il faudra six heures et 42 minutes, et il faut passer le premier groupe par la route n°2, puis n°4, puis utiliser l'hélicoptère pour ...

etc...

FIG. 2 – Dialogue avec TRIPS (oral ou écrit)

Chapitre 1

Données théoriques

1.1 Conceptualisation de l'environnement

Avant toute chose, il faut adopter une méthode de représentation de l'état de l'environnement, qui soit à la fois simple et complète. J'ai décidé d'utiliser la description de type STRIPS[1], d'une complexité largement suffisante pour réaliser un premier planificateur.

Définition 1. ***Un langage de type STRIPS*** *est un langage constitué de :*

- *constantes, qui seront les objets de l'environnement.*
- *classes, qui permettront de différencier les différents types d'objet (on ne peut pas effectuer les mêmes actions avec un cube ou avec une table)*
- *prédicats, qui permettront de décrire les états des objets ; chaque prédicat admet un certain nombre de variables, qui sont le ou les objets dont il décrit l'état (et les relations, dans le cas de plusieurs variables). On notera P (class C_1, class $C_2, \ldots,$ class C_n) un prédicat P admettant n variables respectivement de classes $C_1, C_2, \ldots, C_n$.*

Ainsi, un langage STRIPS qu'il sera possible de choisir dans le cas de la figure 1 sera par exemple constitué de deux classes, *cube* et *support*, quatre constantes, les *cubes* A, B et C et le *support* $Table$, et trois prédicats :

1. *sur* (class *cube*, class *cube*), tel que *sur* (x, y) signifie "le *cube* x est sur le *cube* y".
2. *libre* (class *cube*), tel que *libre* (x) signifie "la face supérieure du *cube* x est libre".
3. *sur_s* (class *cube*, class *support*), tel que *sur_s* (x, y) signifie "le *cube* x est sur le *support* y".

Ces outils - constantes, classes et prédicats - constituent le fondement de STRIPS et permettent de décrire l'état de l'environnement.

Définition 2. *En* STRIPS, ***un état de l'environnement*** *(ou* ***situation****) est défini par un ensemble de prédicats instanciés.*

[1]STRIPS : STanford Research Institute Problem Solver, du nom du programme pour lequel il a été conçu.

Un prédicat instancié est un prédicat qui est explicitement appliqué à des constantes : *sur* (A, B) est un prédicat instancié, qui signifie que le *cube* A de notre environnement est situé sur B. Ce n'est plus une notion abstraite comme *sur* (x, y), qui se réfère à tout x et à tout y, de la même manière que pour une fonction f, $f(2)$ n'est plus une notion abstraite comme $f(x)$. Dans notre exemple, la situation initiale est décrite par :

$$\{sur\_s(A, Table), sur(C, A), libre(C), sur\_s(B, Table), libre(B)\}$$

et la situation finale par :

$$\{sur\_s(C, Table), sur(B, C), sur(A, B), libre(A)\}$$

1.2 Action

Il faut ensuite définir la façon dont on va représenter les actions. Qu'est-ce qui les caractérise ? D'abord, une action met en jeu des objets de l'environnement, il faut donc qu'elle admette des variables, qui apparaîtront dans sa description. Ensuite, on ne peut pas appliquer une action n'importe comment, n'importe quand : il faut une liste de préconditions qui devront être vérifiées pour que l'on puisse appliquer l'action. Cette liste sera un ensemble de prédicats exprimés en fonction des variables de l'action. Enfin, il faut décrire les effets de l'action : on a aussi besoin d'une liste des prédicats qui seront ajoutés à la situation, et d'une liste de prédicats qui seront retirés de la situation.

Définition 3. ***Une action de type STRIPS*** *est un quadruplet* (Var, Pre, Add, Del), *où :*

- Var *est un n-uplet* $(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$, *où* $C_1, C_2, \ldots, C_n$ *sont les classes des variables de l'action, suivies chacune des labels* $x_1, x_2, \ldots, x_n$ *qui seront utilisés pour les désigner dans la suite de la définition.*
- Pre, Add *et* Del *sont des ensembles de prédicats exprimés en fonction de* $x_1, x_2, \ldots, x_n$, *qui désignent respectivement les préconditions de l'action, les ajouts et les retraits à la situation impliqués par l'action. Il faut bien entendu que* $Add \cap Del = \emptyset$.

On appellera action instanciée $\sigma(s)$ *où* σ *est une substitution, une action dont les variables auront été remplacées par des constantes. Pour appliquer une action instanciée à une situation* s, *il faudra que l'on ait* $\sigma(Pre) \subset s$, *et alors, l'action aura pour effet de transformer* s *en* $(s \setminus \sigma(Del)) \cup \sigma(Add)$.

On peut de cette façon définir deux actions pour le monde des cubes :

- $empiler(\{cube\ \ x, support\ \ y, cube\ \ z\}, \{libre(x), libre(z), sur\_s(x, y)\}, \{sur\_s(x, y), sur(x, z)\}, \{sur\_s(x, y), libre(z)\})$, qui empile le *cube* x, reposant sur le *support* y, sur le *cube* z ;
- $depiler(\{cube\ \ x, cube\ \ y, support\ \ z\}, \{libre(x), sur(x, y)\}, \{sur\_s(x, z), libre(y)\}, \{sur(x, y)\})$, qui dépile le *cube* x, reposant sur le *cube* y, sur le *support* z.

L'action instanciée $depiler(C, A, Table)$ représente donc l'action d'enlever le cube C du cube A pour le poser sur la $Table$. Il faut, pour pouvoir l'appliquer, que ses préconditions soient dans la situation actuelle s, ce qui se traduit mathématiquement par $Pre(depiler(C, A, Table)) \subset s$. En d'autres termes, $libre(C)$ et $sur(C, A)$ doivent appartenir à s. Une fois l'action appliquée, on ôtera $sur(C, A)$ à la situation courante s et on lui ajoutera $sur\_s(C, Table)$ et $libre(y)$. On passera ainsi de :

$$s = \{\ldots, libre(C), sur(C, A), \ldots\}$$

à

$$s_{après} = \{\ldots, libre(C), sur\_s(C, Table), libre(A), \ldots\}$$

1.3 Problématique

Ces outils permettent d'obtenir une modélisation suffisamment complète et informatiquement exploitable de l'environnement, dans un très grand nombre de cas. On conçoit par exemple aisément qu'une fois les prédicats *sur*, *sur_s* et *libre* donnés, une fois les opérateurs *depiler* et *empiler* définis, il est possible de poser le problème de l'interaction pour le cas des cubes (figure 1.1). Le rôle du planificateur sera alors d'y répondre en utilisant la modélisation STRIPS (figure 1.2).

Définissons avec STRIPS les notions de problèmes d'interaction et de planification :

Définition 4. ***Problème d'interaction :*** *Un problème d'interaction est un quadruplet* $(L_{STRIPS}, S_i, S_f, L_{actions})$ *où :*

- L_{STRIPS} *est un langage de type* STRIPS *permettant de décrire l'environnement présent,*
- S_i *est la situation initiale, décrite avec* L_{STRIPS}*,*
- S_f *est la situation finale, décrite avec* L_{STRIPS}*,*
- $L_{actions}$ *est l'ensemble des actions disponibles, décrites avec* L_{STRIPS}*.*

Définition 5. ***Problème de planification :*** *Un problème de planification est un couple* $(P_i, L_{contraintes})$ *où :*

- P_i *est un problème d'interaction,*
- $L_{contraintes}$ *est l'ensemble de contraintes exprimées dans un langage* $L \supset L_{STRIPS}$ *que l'on doit tenter de satisfaire.*

La figure 1.3 montre un exemple de problème de planification s'appuyant sur le problème d'interaction de la figure 1.1.

1.4 Notion de plan

Quels types de solutions ces problèmes peuvent-ils avoir ? Une succession d'actions instanciées, qui, appliquées dans l'ordre, nous permettraient d'atteindre l'état final tant

Trouver un plan exécutable permettant de passer de
$\{sur\_s(A, Table), sur(C, A), libre(C), sur\_s(B, Table), libre(B)\}$
à
$\{sur\_s(C, Table), sur(B, C), sur(A, B), libre(A)\}$
à l'aide des opérateurs *empiler* et *depiler*.

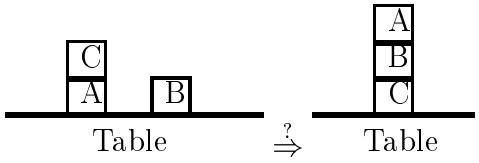


FIG. 1.1 – Problème d'interaction en STRIPS

`situation initiale`
↓
$depiler(C, A, Table)$
↓
$empiler(B, Table, C)$
↓
$empiler(A, Table, B)$
↓
`situation finale`

FIG. 1.2 – Une réponse parmi d'autres au problème de la figure 1.1

Trouver un plan exécutable permettant de passer de
$\{sur\_s(A, Table), sur(C, A), libre(C), sur\_s(B, Table), libre(B)\}$ à
$\{sur\_s(C, Table), sur(B, C), sur(A, B), libre(A)\}$ à l'aide des opérateurs *empiler* et *depiler*, en un temps minimum, sachant qu'*empiler* prend deux fois plus de temps que *depiler*.

FIG. 1.3 – Problème de planification

recherché ? Introduisons la notion de plan : un problème d'interaction a pour solution ce que l'on appelle un plan, description ordonnée des actions à effectuer pour passer de l'état initial à l'état final.

$$\text{Plan :} \quad \texttt{Début} \rightarrow \textit{Action}\ 1 \rightarrow \cdots \rightarrow \textit{Action}\ N \rightarrow \texttt{Fin}$$

L'exemple de la figure 1.2 est aussi un plan. Un tel plan est en réalité le type de plan le plus simple auquel on puisse être confronté, puisqu'il n'envisage qu'un seul ordre d'exécution, et à chaque fois qu'une seule alternative : "après une étape, je dois appliquer la suivante, et ainsi de suite jusqu'à la situation finale". Il s'agit d'un *plan linéaire*. Cette représentation n'est non seulement pas exhaustive (ce qui pourra gêner lors de la résolution du problème de planification) mais elle induit aussi des problèmes lors de la recherche même d'un plan d'interaction, puisqu'en ne cherchant qu'un plan linéaire, on s'oblige à négliger certaines opportunités. Même si, à la fin, on cherche à obtenir un plan linéaire (puisqu'il est impossible d'effectuer plusieurs actions à la fois), on doit s'obliger à travailler sur un type de plan plus ouvert, à savoir un *plan non-linéaire* (voir figure 1.4). La véritable supériorité de ce type de plan apparaîtra bien plus clairement par la suite.

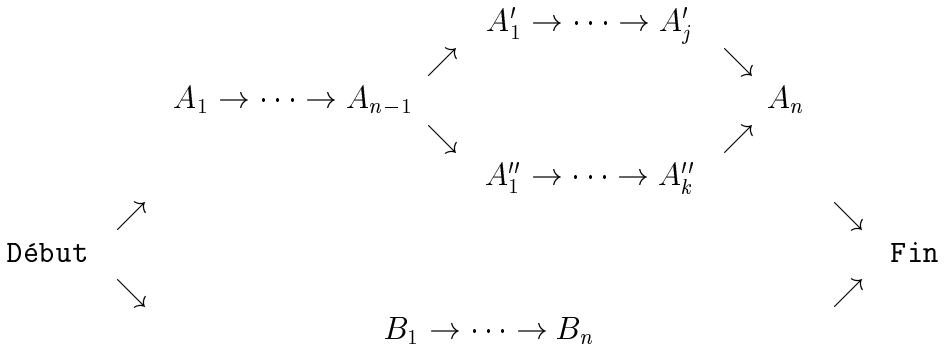


FIG. 1.4 – Plan non-linéaire

On peut toutefois facilement réaliser l'intérêt immédiat d'une telle représentation à travers l'exemple de la figure 1.5. On cherche à éteindre les lumières dans une pièce contenant une lampe de chevet et une lampe halogène toutes deux allumées. On peut commencer par éteindre la lampe de chevet, puis la lampe halogène, ou l'inverse : aucune voie n'est à priori préférable à l'autre. Certes, il faut bien finir par linéariser ce plan pour pouvoir

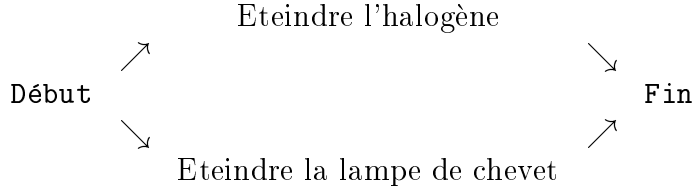


FIG. 1.5 – Comment éteindre les lumières dans une pièce ?

véritablement agir, i.e. le transformer par exemple en :

`Début` → Eteindre l'halogène → Eteindre la lampe de chevet → `Fin`

Mais si l'on se contentait de travailler sur le plan linéaire final, on perdrait une information ("possibilité de commencer indifféremment par l'un ou par l'autre"). Supposons qu'ensuite, par exemple, on pose la contrainte "d'abord éteindre l'halogène", rien ne nous permet d'affirmer à partir du plan linéarisé que l'on peut effectivement inverser l'ordre.

Définition 6. ***Plan :*** *Un plan est un couple* $(L_{actions}, L_{relations})$ *où :*

- $L_{actions}$ *est un ensemble d'actions instanciées,*
- $L_{relations}$ *est un ensemble de relations d'ordre (ou encore de relations de précédence) entre les actions de* $L_{actions}$*. Par exemple, pour deux actions* A *et* B*, si* A *doit avoir lieu avant* B*, on note* $A \prec B$*, et alors* $(A \prec B) \in L_{relations}$*.*

Le plan ainsi obtenu contient implicitement un ordre partiel. D'autre part, on représente la situation initiale *par une action qui ajoute les prédicats instanciés décrivant cette situation :* $A_{initiale}(Pre = \emptyset, Add = s_{initiale}, Del = \emptyset)$*. La* situation finale *est quant à elle décrite par une action qui a pour ensemble de préconditions les prédicats instanciés décrivant cette situation, et qui n'ajoute ni ne retire rien à l'environnement :* $A_{finale}(Pre = s_{finale}, Add = \emptyset, Del = \emptyset)$*.*
Dans un plan, on a donc toujours :

$$\forall A \in L_{actions}, (\{A_{initiale} \prec A\} \cup \{A \prec A_{finale}\}) \subset L_{relations}$$

En schématisant un peu, le plan décrit par la figure 1.5 a ainsi pour ensemble d'actions :

$$\{A_{initiale} = \text{"Allumer les deux lampes"}, A_{finale} = \text{"Les lampes doivent être éteintes"},$$
$$B = \text{"Eteindre la lampe de chevet"}, A = \text{"Eteindre l'halogène"}\}$$

et pour ensemble de relations :

$$\{A_{initiale} \prec A, A_{initiale} \prec B,$$
$$A \prec A_{finale}, B \prec A_{finale}, \quad A_{initiale} \prec A_{finale}\}$$

L'ordre est bien partiel car rien ne décrit les relations entre A et B.

Définition 7. ***Plan linéaire :*** *Un plan linéaire est un plan totalement ordonné, c'est-à-dire que :*

$$\forall (A, B) \in (L_{actions})^2, B \neq A, \quad \begin{cases} soit\ A \prec B, \\ soit\ B \prec A. \end{cases}$$

En d'autres termes, chaque action est toujours suivie d'une seule autre action. On pourrait représenter le plan sur une ligne (d'où le nom), comme sur la figure 1.2. Le plan linéarisé est utile au moment de l'exécution, puisqu'il décrit précisément les actions à

effectuer et l'ordre dans lequel on doit agir. D'un plan non-linéaire (par exemple celui de la figure 1.5), on peut bien entendu construire plusieurs plans linéarisés :

`Début` → Eteindre l'halogène → Eteindre la lampe de chevet → `Fin`

ou

`Début` → Eteindre la lampe de chevet → Eteindre l'halogène → `Fin`

qui sont tous deux exécutables.

Définition 8. ***Linéariser un plan,*** *c'est créer un plan linéaire à partir d'un plan non-linéaire.*

Il n'est cependant pas toujours possible de linéariser un plan, si par exemple dans le plan non-linéaire on a une action A qui doit avoir lieu avant une autre action B, laquelle doit elle-même être effectuée avant l'action A ($A \prec B$ et $B \prec A$). Dans ce cas, il n'existe pas de *réalisation exécutable* du plan non-linéaire.

1.5 Planification

Le problème est donc à présent entièrement posé : le rôle du planificateur est de nous fournir un plan non-linéaire, linéarisable (voire de nous fournir directement un des plans linéarisés), solution d'un problème de planification, ou au moins au problème de l'interaction. Il reste à définir comment le planificateur va procéder, sachant qu'il dispose des descriptions des situations initiale et finale et d'actions pour modifier l'environnement.

1.5.1 Chaînage avant, chaînage arrière

Puisque le planificateur va devoir trouver une ou plusieurs manières de passer de $s_{initiale}$ à s_{finale}, il va déjà falloir savoir par où il doit commencer. On a en effet le choix entre deux orientations : soit on va chercher à modifier la situation initiale pour arriver à la situation finale, ce que l'on appelle *chaînage avant* ; soit on va chercher, en partant de la situation finale, à retrouver les actions qui ont permis d'arriver jusque là, c'est le *chaînage arrière*[2]. Le chaînage arrière paraît néanmoins plus intuitif, puisqu'on part du but à atteindre et qu'on cherche à retrouver le cheminement (plutôt que de partir des hypothèses et de chercher à atteindre le but en essayant toutes les possibilités - il suffit d'imaginer un problème suffisamment complexe pour constater que l'on adopte soi-même rarement une attitude de "chaînage avant", mais plutôt qu'on s'aide du but en se disant : "comment en arrive-t-on

[2]Il existe un méthode hybride, appelée *bigression*, utilisant simultanément les chaînages avant et arrière. On peut montrer, purement algorithmiquement (c'est-à-dire sans se soucier des particularités de la planification), que cette méthode induit une réduction considérable des besoins en mémoire (pour un arbre de recherche de profondeur n avec k fils à chaque fois, en notant M_b et M_c la mémoire requise respectivement par la bigression et par le chaînage (avant ou arrière), on a $\frac{M_b}{M_c} \sim_{n\to\infty} \frac{\lambda}{k^n}$, où λ est une constante). Il n'y a par contre à priori aucun gain de temps de calcul.

là ?"). Puisqu'il s'agit du comportement à priori le plus intuitif et donc le plus humain, et qu'il s'agit d'intelligence artificielle, adoptons-le.

Le chaînage arrière implique de raisonner en termes de *buts* et d'*établisseurs*. Il y a des buts à atteindre, et pour cela, on a besoin d'établisseurs.

Définition 9. ***But***

Une précondition b d'une action A est dite but de A, noté b_A si elle n'apparaît pas dans la liste d'ajout de l'action $A_{initiale}$.

Définition 10. ***Etablisseur***

Un action A' est dite établisseur du but b_A d'une action A si elle possède b_A en liste d'ajout ($b_A \in Add(A')$), et on note $A' \rightarrow b_A$.

De manière générale, on va donc considérer que toutes les préconditions d'une action qui ne sont pas "ajoutées" (ou *établies*) par l'action initiale $A_{initiale}$ (cf définition 6) sont des buts à atteindre. Ainsi, pour l'action finale $A_{finale} = \{s_{finale}, \emptyset, \emptyset\}$, chacun des prédicats instanciés de s_{finale} sont des buts à atteindre (sauf évidemment ceux que $A_{initiale}$ établit déjà). On comprend maintenant l'intérêt de cette représentation (assimiler les situations finale et initiale à des actions), et après avoir défini la notion d'établisseur, il devient possible d'ébaucher un premier algorithme :

Algorithme 1. ***Planification en chaînage arrière***

Au début, on crée la première ramification $\mathcal{R}_0$ correspondant à A_{finale} et on lui associe la liste $L_{buts}(\mathcal{R}_0)$ des buts en attente qui est :

$$L_{buts}(\mathcal{R}_0) = s_{finale} \setminus s_{initiale} = Pre(A_{finale}) \setminus Add(A_{initiale})$$

Ensuite, il s'agit d'une récurrence. Pour toute ramification $\mathcal{R}$ où il reste des buts à atteindre, trouver une ou plusieurs actions A_i établissant chaque but b_i, créer une ramification $\mathcal{R}_i$ pour chaque action A_i (établisseur du but en question), y associer une nouvelle liste de buts $L_{buts}(\mathcal{R}_i)$ et ajouter dans $L_{buts}(\mathcal{R}_i)$ les préconditions de A_i qui ne sont pas établies par $A_{initiale}$. On enlève enfin b_i de $L_{buts}(\mathcal{R})$, et on recommence jusqu'à ce que toutes les listes $L_{buts}(\mathcal{R})$ soient vides, $\forall \mathcal{R}$.

Au début il n'y a donc que les buts induits par l'action finale A_{finale}. C'est progressivement que l'on devra retrouver ce qui a permis d'arriver à A_{finale}, par chaînage arrière, c'est-à-dire retrouver les établisseurs qui permettent de satisfaire les préconditions de A_{finale}, et ainsi de suite, comme le montre l'exemple de la figure 1.6 (A_{finale} possède ici deux buts, but 1 et but 2).

1.5.2 Arbre de plans

Grâce à l'algorithme 1, le planificateur va donc pouvoir trouver un plan solution du problème. Mais qu'est-ce qu'un plan pour le programme ? On choisit de représenter un plan sous la forme d'un graphe d'actions instanciées, graphe dont une extrémité est l'action

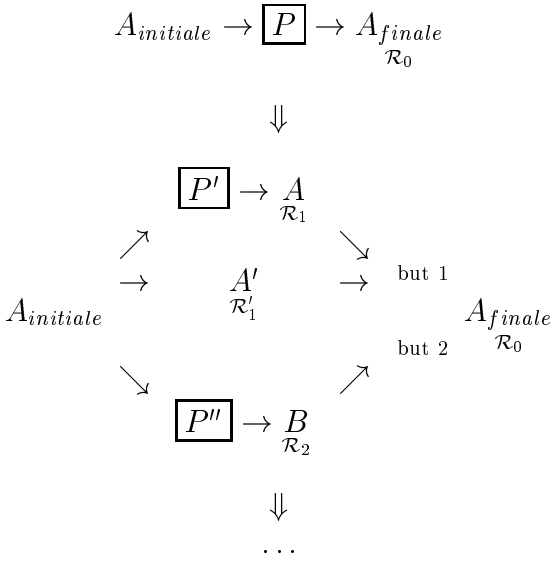
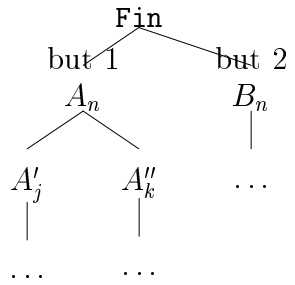


FIG. 1.6 – Progression de l'algorithme 1

finale, et dont l'autre est l'action initiale. A chaque but en attente est associé un fils qui est précisément un établisseur de ce but : chaque action a ainsi autant de fils que de buts. Un graphe possède d'autre part la hiérarchie dont on a besoin pour décrire un plan ($L_{relations}$). Le plan non-linéaire représenté sur la figure 1.4 a ainsi la structure de graphe suivante :



A_n et B_n sont respectivement les établisseurs du premier et du deuxième but de `Fin`. Si l'on reprend l'exemple concret des cubes, le graphe de la figure 1.7 représente un plan non-linéaire solution. Le prédicat instancié $sur\_s(C, Table)$ est un but de l'action finale, l'action instanciée $depiler(C, A, Table)$ en est un établisseur donc c'est un fils de cette action finale, au niveau du but $sur\_s(C, Table)$. Quant aux buts de $depiler(C, A, Table)$ ($libre(C)$ et $sur(C, A)$), on constate qu'ils sont établis par l'action initiale, la recherche est donc terminée pour cette branche du plan-graphe.

Cependant, le planificateur ne se contente pas seulement de chercher un seul graphe-plan. Il peut en effet toujours y avoir plusieurs établisseurs pour un but, ce qui crée autant de possibilités de plans différents. Dans la figure 1.6, on voit que le but 1 de l'action A_{finale} peut être établi aussi bien par A que par A'. Pour gérer toutes ces possibilités, il faut recourir à la notion d'arbre de plans : dans un tel arbre, un nœud - et donc une

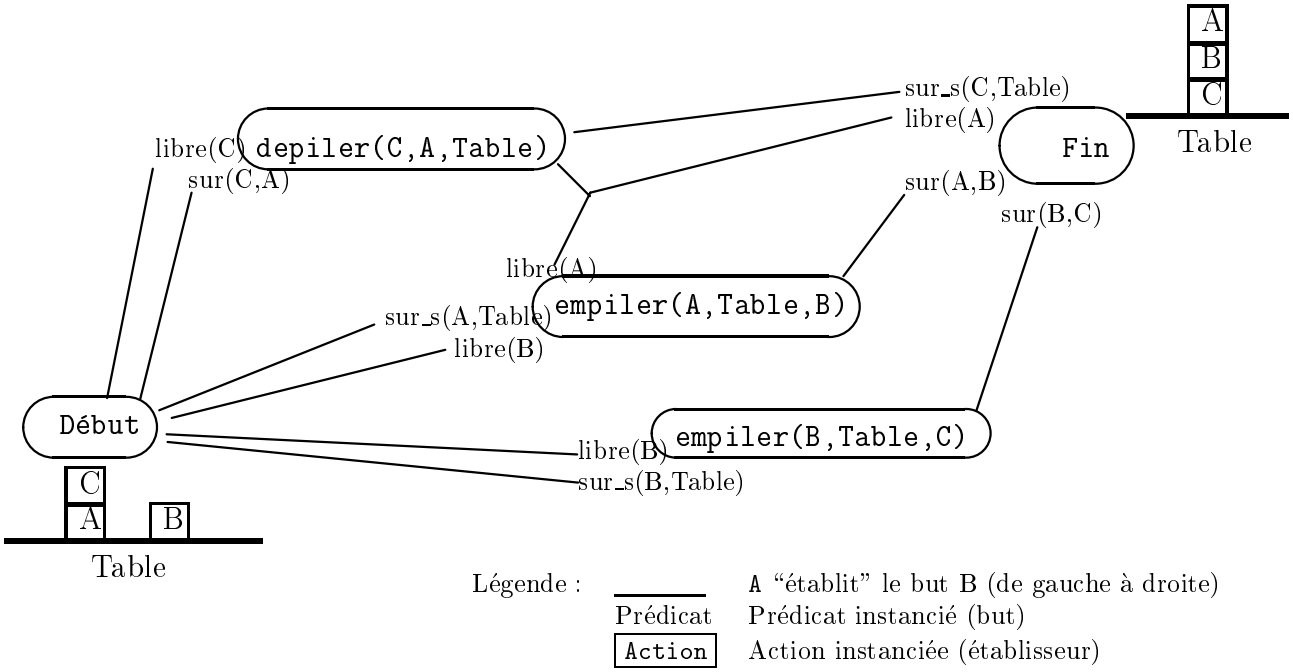
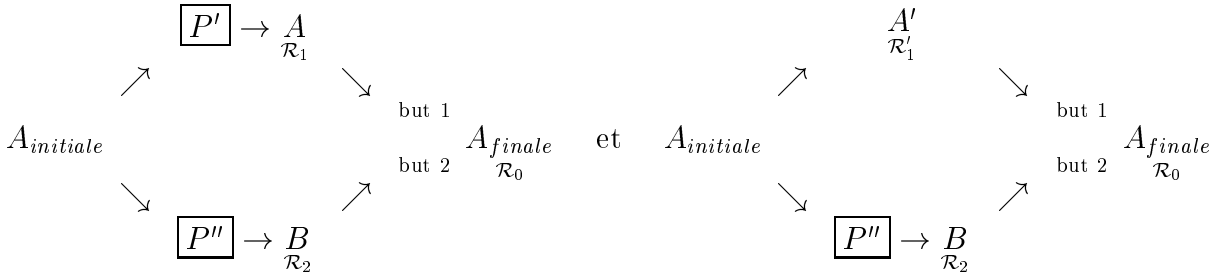


FIG. 1.7 – Un plan non-linéaire pour le problème de la figure 1.1

ou plusieurs bifurcations - apparaît dès qu'il y a plusieurs manières d'atteindre un but. Par définition, chaque lignée de l'arbre de plans est ainsi un plan, qui est lui-même un graphe d'actions instanciées. Le schéma de la figure 1.6 représente un tel arbre de plans, qui contient en réalité 2 plans :



dont chacun est un graphe d'actions instanciées (P' et P'' sont des macro-blocs représentant des portions du plan que l'algorithme 1 doit encore trouver).

1.6 Limitations de l'algorithme 1

1.6.1 Cas de bouclage

L'algorithme 1 est loin d'être parfait. Son principal défaut est de ne pas prendre en compte les cas de bouclage. Effectivement, donnons-nous le problème suivant (d'après [3], p. 119) :

– trois prédicats instanciés a, b et c,

- deux actions $A = (Pre = \{b\}, Add = \{a\}, Del = \{b\})$ et $B = (Pre = \{a\}, Add = \{b\}, Del = \{a\})$,
- les situations $s_{initiale} = \{c\}$ et $s_{finale} = \{a\}$.

L'algorithme 1 est incapable de le résoudre, il va boucler indéfiniment :

$$A_{initiale} \rightarrow \boxed{P} \rightarrow A_{finale}$$
$$A_{initiale} \rightarrow \boxed{P'} \rightarrow B \rightarrow A_{finale}$$
$$A_{initiale} \rightarrow \boxed{P''} \rightarrow A \rightarrow B \rightarrow A_{finale}$$
$$A_{initiale} \rightarrow \boxed{P'''} \rightarrow B \rightarrow A \rightarrow B \rightarrow A_{finale}$$
$$\ldots$$

Puisqu'on peut trouver ce genre d'exceptions dans des problèmes très simples, il est donc impératif de s'en prémunir. Rajoutons ainsi la clause suivante à l'algorithme 1 :

Algorithme 2. ***Planification en chaînage arrière sans bouclage***

Appliquer l'algorithme 1, mais vérifier pour chaque but que l'on tente d'établir si l'on ne le retrouve pas dans l'ascendance directe de la ramification où l'on se trouve. Auquel cas, éviter de l'établir de la même façon qu'auparavant (i.e. avec le même établisseur), sinon on aurait une boucle infinie.

En l'occurrence, dans l'exemple précédent, l'algorithme 2 s'arrête à :

$$A_{initiale} \rightarrow \boxed{P'} \rightarrow B \rightarrow A_{finale}$$

puisque B ajoute comme but le prédicat a, qui est déjà un but de A_{finale}.

Cette clause n'est cependant pas idéale, puisqu'on peut imaginer un plan où on rencontre le même but plusieurs fois, où on l'établit de la même façon, mais pas au même moment (par exemple, dans un problème de tours de Hanoï, où on peut mettre un anneau plusieurs fois sur le même piquet). En fait, l'algorithme 2 constitue déjà une très bonne optimisation de l'algorithme 1, en ce sens qu'il permet d'éviter le bouclage au moment de la construction de l'arbre de plans. Il reste à gérer le cas particulier où il est nécessaire d'avoir le même but établi plusieurs fois par le même établisseur. Ce problème est résolu - et on va le voir, dans un cadre bien plus général - par la technique dite du *"white knight"*, ou "chevalier blanc" ([3], p. 113).

1.6.2 Interrogations sur la linéarisation, cas du chevalier blanc

Avant de décrire plus avant ce qu'est le *white knight*, revenons un peu sur le processus de linéarisation d'un plan. En appliquant l'algorithme 2 à un problème d'interaction, on obtient donc de façon certaine un arbre de plans (sauf s'il y a un but pour lequel aucun établisseur n'existe).

Définition 11. ***Exactitude d'un plan***

Un plan non-linéaire P est exact si et seulement si pour chacun des buts des actions A qui constituent P, il existe une autre action A' qui en soit l'établisseur.

$$P \text{ est exact} \Longleftrightarrow \forall A \in L_{actions}, \forall b_A \in (Pre(A) \setminus s_{initiale}), \exists A' \text{ tel que } A' \rightarrow b_A$$

Avec cette définition, l'algorithme 2 produit donc un arbre de plans exacts, s'il en existe.

Theorème 1. *Pour tout problème d'interaction, l'algorithme 2 trouve un arbre de plans non-linéaires exacts, en l'absence de buts non établissables.*

Il faut cependant rester méfiant, car ceci ne nous assure absolument pas qu'on peut linéariser chacun des plans trouvés en un plan linéaire *exécutable*. Dans la forme actuelle, on a juste acquis la certitude que pour chaque plan de l'arbre, chacune de ses actions était bien résolue (i.e. chacun de ses buts était établi) par d'autres actions, elles-mêmes résolues, et ainsi de suite... D'où la notion d'*exactitude* : un plan peut tout à fait être exact, mais ne pas être linéarisable. Par exemple, dans le cas d'un plan P où $A \prec B$ et $B \prec A$, on ne peut pas linéariser P. On peut par contre très bien avoir $Pre(A) \subset s_{initiale}$ et $Pre(B) \subset s_{initiale}$, ce qui assure l'*exactitude* du plan. C'est un plan non-linéaire que l'algorithme 2 pourrait très bien fournir. Que faire alors ?

Se dire que, pour le moment, on sait obtenir un arbre de plans exacts, et que c'est déjà très bien. Comme dans l'arbre de plans exacts, chacun des plans est *une* façon de "résoudre le problème"[3], on peut d'ores et déjà ne s'intéresser qu'à un plan extrait. Supposons donc à partir de maintenant que l'on travaille sur un des plans de l'arbre (une lignée en quelque sorte), il s'agira évidemment d'un plan non-linéaire exact. Il reste à le linéariser, ou du moins, à essayer de le linéariser. En réalité, c'est à ce moment que les relations d'ordre vont poser problème. Un plan exact a toujours un ordre partiel implicite, le plus faible et le plus immédiat qui soit : tout établisseur A' est forcément avant l'action A dont il établit un des buts b_A. C'est l'ordre *zéro-partiel* qui se définit par :

Définition 12. ***Ordre zéro-partiel***
Un plan non-linéaire P est zéro-partiellement ordonné s'il vérifie la condition suivante :

$$\forall A \in L_{actions}, \forall b_A \in (Pre(A) \setminus s_{initiale}), \forall A' \text{ tel que } A' \rightarrow b_A, A' \prec A$$

Theorème 2. *Un plan non-linéaire est* zéro-partiellement *ordonné s'il est exact.*

Le plan non-linéaire de la figure 1.7 est un plan exact, constitué de trois actions ($A_1 = depiler(C, A, Table)$, $A_2 = empiler(A, Table, B)$, $A_3 = empiler(B, Table, C)$), $A_{initiale} =$ `Debut` et $A_{finale} =$ `Fin`). Il est zéro-partiellement ordonné d'après la hiérarchie des établisseurs :

$$A_1 \rightarrow libre(A)_{A_2} \Rightarrow \qquad\qquad A_1 \prec A_2$$

[3]Rappelons-nous, une nouvelle bifurcation apparaît dans l'arbre des plans chaque fois qu'il y a plusieurs établisseurs pour un but donné, c'est-à-dire plusieurs manières de procéder.

et évidemment

$$\begin{aligned}
\texttt{Début} \rightarrow libre(C)_{A_1}, sur(C,A)_{A_1} &\Rightarrow & \texttt{Début} &\prec A_1\\
\texttt{Début} \rightarrow sur_s(A,Table)_{A_2}, libre(B)_{A_2} &\Rightarrow & \texttt{Début} &\prec A_2\\
\texttt{Début} \rightarrow libre(B)_{A_3}, sur_s(B,Table)_{A_3} &\Rightarrow & \texttt{Début} &\prec A_3\\
A_1 \rightarrow sur_s(C,Table)_{\texttt{Fin}}, libre(A)_{\texttt{Fin}} &\Rightarrow & A_1 &\prec \texttt{Fin}\\
A_2 \rightarrow sur(A,B)_{\texttt{Fin}} &\Rightarrow & A_2 &\prec \texttt{Fin}\\
A_3 \rightarrow sur(B,C)_{\texttt{Fin}} &\Rightarrow & A_3 &\prec \texttt{Fin}
\end{aligned}$$

L'ordre est bien partiel car on ne sait pas à priori où se situe A_3 par rapport à A_1 et à A_2. C'est l'ordre le plus primaire dont on puisse parler.

Ensuite, il faut se rappeler qu'une action établit certes des buts, mais qu'elle peut aussi en détruire, c'est alors à la fois un établisseur de certains buts (correspondant à la liste d'ajout *Add*) et un casseur d'autres (liste de retrait *Del*).

Définition 13. ***Casseur***

Une action C est dite casseur du but b_A d'une action A si elle possède b_A en liste de retrait ($b_A \in Del(C)$), et on note $C \dashv b_A$.

On peut alors se donner une nouvelle notion relative aux plans, de portée supérieure à celle de l'exactitude :

Définition 14. ***Cohérence d'un plan***

Un plan non-linéaire P est dit cohérent *si et seulement si pour toute action A' établissant un but b_A de A, il n'existe aucune action C qui soit située entre A' et A et qui casse b_A.*

$$P \text{ cohérent} \Longleftrightarrow \forall A' \rightarrow b_A, \nexists C \text{ tel que } A' \prec C \prec A \text{ et } C \dashv b_A$$

Définition 15. ***Ordre un-partiel***

Un plan non-linéaire est un-partiellement *ordonné s'il est zéro-partiellement ordonné et s'il est* cohérent.

Ces définitions permettent à présent de préciser clairement les limites de l'algorithme 2 : il est incapable de nous fournir un plan un-partiellement ordonné, il va donc falloir - encore - l'améliorer.

Quand peut-on être en présence d'un casseur ? Quand on cherche à linéariser le plan zéro-partiellement ordonné, il faut d'abord le un-ordonner. A ce moment là, des casseurs peuvent apparaître, et deux alternatives s'offrent à nous pour s'en débarrasser :

– soit on le déplace après,

– soit on le déplace avant.

Parfois, il n'est cependant pas possible de déplacer le casseur, parce qu'il doit absolument rester après l'établisseur et avant l'action dont on cherche à établir le but. Quand on a épuisé toutes les possibilités de déplacement (il peut y en avoir jusqu'à $n!$ dans un plan à n actions), il faut recourir à une autre solution. David Chapman, dans son planificateur non-linéaire baptisé TWEAK, a proposé une troisième voie : le chevalier blanc. Puisque le casseur est inamovible, adjoignons-lui juste après un autre établisseur, "le chevalier blanc"[4], chargé de rétablir le but cassé. Le rétablisseur ne doit bien sûr pas détruire le travail du casseur (quoiqu'à la rigueur, ce soit sans gravité : il suffira de trouver un autre rétablisseur juste après pour réparer une fois de plus les dégâts).

Algorithme 3. ***Création d'un plan cohérent***

*Pour tout casseur C d'un but b_A, tenter de le déplacer, et en cas d'échec, trouver un rétablisseur de b_A à placer après C (*chevalier blanc*).*

Appliquons-le au cas des cubes, où on a pour le moment comme unique contrainte $A_1 \prec A_2$. On sait que :

- A_1 détruit $sur(C, A)$,
- A_2 détruit $sur\_s(A, Table)$ et $libre(B)$,
- A_3 détruit $sur\_s(B, Table)$ et $libre(C)$.

On peut rapidement écarter $sur\_s(C, A)$, $sur\_s(A, Table)$ et $sur\_s(B, Table)$, car ils ne sont détruits que par les établisseurs dont ils sont les buts, donc leur destruction ne gêne personne. Par contre, A_2 détruit $libre(B)$, or $A_{initiale}$ est un établisseur de $libre(B)_{A_3}$, donc A_2 est forcément soit avant $A_{initiale}$ (impossible) soit après A_3, ou bien il va falloir ajouter un chevalier blanc entre A_2 et A_3 pour rétablir $libre(B)$. Il faut alors imaginer plusieurs hypothèses de travail : soit A_2 est après A_3, soit il faut ajouter un rétablisseur. L'algorithme va d'abord travailler sur la première solution, en gardant la seconde pour plus tard, si jamais il échoue (il est toujours plus compliqué de rajouter un chevalier blanc que d'essayer de résoudre le problème en déplaçant le casseur). Donc, cas n°1 : $A_3 \prec A_2$ (et cas n°2, mis de côté : *trouver un rétablisseur entre A_2 et A_3*). Ensuite, A_3 détruit $libre(C)$, donc c'est un casseur de $A_{initiale} \rightarrow libre(C)_{A_1}$. D'où, A_3 est soit avant $A_{initiale}$ (impossible), soit après A_1, soit il faudra recourir à un chevalier blanc. Ainsi, cas n°(1,1) : $A_1 \prec A_3$. Ceci nous donne :

$$A_1 \prec A_3, A_1 \prec A_2 \text{ et } A_3 \prec A_2$$

soit forcément :

$$A_{initiale} \prec A_1 \prec A_3 \prec A_2 \prec A_{finale}$$

Il est aisé de vérifier que ce plan est cohérent, de plus, il est linéaire. C'est d'ailleurs une solution (la meilleure, ici) au problème d'interaction de la figure 1.1. Ici, l'algorithme 3 a réussi sans recourir au *white knight*, même s'il n'a pas oublié de prendre note des opportunités qu'il a temporairement laissées de côté. Il faut cependant voir que dans le cas général l'on n'aboutit pas toujours du premier coup à un plan cohérent, auquel cas

[4]Allusion financière : en bourse, on qualifie de *chevalier blanc* une entreprise qui protège une autre entreprise en difficulté (OPA agressive, etc...)

l'algorithme va essayer une des autres possibilités mises de côté précédemment. Un exemple concret nécessitant le recours au *chevalier blanc* serait trop lourd à détailler ici, c'est pourquoi je me borne à n'esquisser que le comportement de l'algorithme 3 dans une telle situation.

L'algorithme révèle toute sa force lors de la linéarisation, puisque c'est à ce moment précis que les problèmes de précédence, d'établisseurs et de casseurs font surface : il faut rendre le plan *cohérent*, et donc l'ordonner beaucoup plus fortement que lorsqu'on veut un plan *exact*. Si l'on s'aperçoit qu'il est impossible de trouver un ordre dans lequel exécuter les actions dont on dispose grâce au plan exact fourni par l'algorithme 2, il faut souvent rajouter des actions[5]. Ce nouvel algorithme permet aussi de résoudre l'interrogation de la fin du paragraphe 1.6.1 (p. 21), puisqu'à force de rajouter des rétablisseurs en réponse à des casseurs, le plan va bien finir par être trouvé, du moins s'il est trouvable.

L'algorithme 3 a effectivement un unique mais important handicap, puisqu'il a deux issues possibles : soit il forme un plan cohérent (tout va bien), soit il boucle indéfiniment (voire au moins très longtemps, et il faut aussi tenir compte des limitations matérielles). Dans le deuxième cas, on pourrait adjoindre une fois encore un correctif anti-boucle, mais de quel type ? La méthode utilisée dans l'algorithme 2 protège certes du bouclage, mais il faut l'y associer un "rétablisseur", qui est lui-même le *white knight*, qui peut donc encore boucler. On tourne en rond... Ce pourrait être l'enjeu d'un algorithme suivant.

[5]Ce qui paraît logique, car après tout, un plan exact ne contient pas forcément toutes les actions - loin de là - qui seront nécessaires à l'élaboration du plan linéaire solution. Un plan exact a pour seul mérite d'être exact, c'est-à-dire qu'il va du début à la fin, mais de façon partielle. Dans un tel plan, un but a toujours un établisseur pour le servir, de $A_{initiale}$ à A_{finale}, mais on ne peut pas en conclure qu'on a tout trouvé.

Chapitre 2

Conception du planificateur

Les bases théoriques étant posées, il reste à créer le planificateur, c'est-à-dire mettre en œuvre la structure STRIPS, les différents algorithmes présentés, etc... Mais avant tout, il faut décider du langage il sera programmé.

2.1 Et si on le codait en C ?

La question du choix du langage de programmation n'est pas vaine. Opter pour un langage plutôt que pour un autre, c'est aussi adopter une manière particulière de concevoir le planificateur, et donc de répondre au problème de la planification.

A priori, le langage le plus indiqué est le PROLOG[1], car il permet de gérer facilement les opérations sur les ensembles, les implications logiques, etc... Cependant, une fois les outils de manipulation de STRIPS créés, le C présente l'avantage d'être bien plus rapide et bien moins gourmand en mémoire que le PROLOG, ce qui n'est pas négligeable lorsque l'on considère la puissance de calcul et de stockage qu'exige un planificateur, dès que le problème à résoudre n'est plus trivial. J'ai donc opté pour le C.

2.2 Outils de base

2.2.1 Modélisation des concepts algébriques

A présent, on doit choisir la manière dont on va représenter les objets STRIPS et les actions. Il ne faut surtout pas se tromper, car le comportement et la philosophie du planificateur seront déterminés par cette étape. Ceci précède tout le reste, puisque tout élément du programme devra précisément manipuler les objets que l'on aura définis au tout début. Pour autant, il faut rester conscient du fait que la conception de ces objets devra être retouchée (ce qui est d'ailleurs arrivé à maintes reprises). Il devient ainsi nécessaire de

[1] PROLOG *(PROgrammation LOGique)*, langage logique créé en 1970 par Robert Kowalski (Edimburgh University) et Alain Colmerauer (Université d'Aix-Marseille), intensivement utilisé dans la recherche en Intelligence Artificielle.

prévoir une architecture efficace (i.e. qui permette de représenter facilement et fidèlement et de façon exhaustive des situations abstraites, notamment ici de représenter fidèlement le formalisme STRIPS) et flexible (les modifications ultérieures de la structure des objets doivent engager un minimum de travail sur le code déjà créé ; on ne doit pas à chaque fois avoir à reprogrammer intégralement les fonctions qui s'appuient sur ces objets).

Il faut commencer par le plus fondamental, à savoir les *classes* d'objets. Ce sont des structures contenant une chaîne de caractères, le `label`, ou étiquette (leur nom), et un pointeur vers une autre structure de même type, au cas où la classe en question serait elle-même une sous-classe[2] (cf. figure 2.1). De même, on va définir les *objets* comme des structures contenant un `label` et un pointeur vers la classe à laquelle ils appartiennent (fig. 2.2). On peut ensuite définir les *prédicats*, structures contenant un `label`, un entier `nVar` décrivant le nombre de variables, et un tableau dynamique `**classVar` de pointeurs de classe, décrivant la classe de chacune des variables (fig. 2.3).

Dans l'exemple des cubes, on aurait donc :

- une classe `cube` (`label "cube"`, `class=NULL`),
- une classe `support` (`label "support"`, `class=NULL`),
- trois objets `A`, `B` et `C` (`label "A"`, `*class=cube`, `label "B"`, `*class=cube`, `label "C"`, `*class=cube`),
- un prédicat `sur` (`label "sur"`, `nVar=2`, `*classVar[0]=*classVar[1]=cube`),
- un prédicat `sur_s` (`label "sur_s"`, `nVar=2`, `*classVar[0]=cube`, `*classVar[1]=support`),
- un prédicat `libre` (`label "libre"`, `nVar=1`, `*classVar[0]=cube`)

Enfin, il reste à définir les *opérateurs* (cf. fig 2.4 et déf. 3 p. 12), constitués comme les prédicats d'un `label`, d'un entier `nVar` donnant le nombre de variables et d'un tableau dynamique `**classVar` de pointeurs de classe. Ils contiennent d'autre part trois entiers `nPre`, `nAdd` et `nDel` auxquels correspondent respectivement les nombres de prédicats dans la liste des préconditions Pre, dans la liste d'ajout Add et dans la liste de retrait Del. Sont ensuite définis trois tableaux dynamiques `**Pre`, `**Add` et `**Del` qui sont autant de pointeurs vers chacun des prédicats de chacune des listes. Ainsi, `Pre[0]` est un pointeur vers le premier prédicat de la liste des préconditions. Il faut enfin une liste de *réindexation* pour chaque prédicat, puisque les variables de l'opérateur ne sont pas forcément dans le même ordre que celles du prédicat, on attache alors à chacune des listes une liste de réindexation `lPre`, `lAdd` et `lDel`, telle que par exemple, la j-ème variable du prédicat i `Pre[i]` de la liste des préconditions soit la `lPre[i][j]`-ème variable de l'opérateur (cette dernière variable est désignée par `*classVar[lPre[i][j]]`). Dans l'exemple des cubes, l'opérateur $depiler(x, y, z)$[3] serait ainsi déclaré :

- `label "depiler"`,
- `nVar=3`,
- `*classVar[0]=cube, *classVar[1]=cube` et `*classVar[2]=support`,

[2] Si la classe n'est pas une sous-classe, le pointeur `class` vaudra `NULL`, i.e. il ne pointe vers rien.

[3] Pour rappel, cette action est ainsi mathématiquement définie : $depiler(\{cube\ x, cube\ y, support\ z\}, \{libre(x), sur(x, y)\}, \{sur\_s(x, z), libre(y)\}, \{sur(x, y)\})$, qui dépile le *cube* x, reposant sur le *cube* y, sur le *support* z.

```
typedef struct Type
{
  char *label;
  struct Type *class;
} Type;
```

FIG. 2.1 – Structure d'une *classe* d'objets

```
typedef struct
{
  char *label;
  Type *class;
} Object;
```

FIG. 2.2 – Structure d'un *objet*

```
typedef struct
{
  char *label;
  int nVar;
  Type **classVar;
} Predicate;
```

FIG. 2.3 – Structure d'un *prédicat*

```
typedef struct
{
  char *label;
  int nVar;
  Type **classVar;
  int nPre, nAdd, nDel;
  Predicate **Pre, **Add, **Del;
  int lPre[N][N], lAdd[N][N], lDel[N][N];
} Operator;
```

FIG. 2.4 – Structure d'une *action*, ou opérateur

- nPre=2, nAdd=2 et nDel=1,
- *Pre[0]=libre, *Pre[1]=sur ;
  *Add[0]=sur_s, *Add[1]=libre ;
  *Del[0]=sur,
- lPre[0]={0}, lPre[1]={0,1} ;
  lAdd[0]={0,2}, lAdd[1]={1} ;
  lDel[0]={0,1}.

A partir d'ici, le programme est capable de représenter intégralement et efficacement le formalisme STRIPS. La deuxième étape consiste alors à permettre la saisie des informations relatives au problème, et ce, au moyen d'un langage qui reste à définir et qui permettra de communiquer précisément et facilement les données à l'ordinateur : ce sera l'interface du planificateur.

2.2.2 L'interface

Cette partie constitue une portion exclusivement technique et certes assez peu intéressante du planificateur, il faut néanmoins rester très attentif à ce qu'elle reste ergonomique (pas de manœuvres alambiquées pour une opération basique) et qu'elle soit impécablement programmée (aucun *bogue* ne peut être toléré, puisqu'elle va constituer, avec la modélisation des objets, la clef de voûte du programme). On trouvera sur le tableau 2.1 les principales commandes de cette interface, exclusivement textuelle. Outre celles qui servent à saisir les données du problème, il a été ajouté des commandes de contrôle (permettant de savoir ce qui a déjà été défini) et des commandes d'exécution (lancer la résolution, etc...). En utilisant ce langage, on peut entrer le problème des cubes grâce au programme décrit sur la figure 2.5.

Il a fallu d'autre part créer quelques algorithmes de saisie qui, à partir d'une commande entrée anarchiquement par l'utilisateur (des espaces un peu n'importe où) et d'un seul coup (toutes les informations sont données simultanément), devaient permettre une juste interprétation, quelques soient le nombre et le type des arguments entrés. On a donc eu besoin de créer un algorithme de nettoyage de la chaîne saisie (suppression des espaces superflus, séparation des arguments par un seul espace) et d'un algorithme d'extraction des arguments (fonction qui renvoie d'une part le premier argument *et* d'autre part le reste de la commande), dont je ne détaillerai pas ici le fonctionnement.

2.3 Planifier...

2.3.1 *Strips* en chaînage arrière, la fausse piste

Tout est maintenant prêt pour pouvoir s'attaquer à la planification en elle-même, et donc faire fonctionner la commande plannify, c'est-à-dire programmer l'algorithme 3, ou du moins débuter par l'algorithme 2.

Je n'ai pas immédiatement conçu un tel programme. En effet, j'ai commencé par faire

- class C [class SC], définit la classe C [qui, optionnellement, est une sous-classe de SC].
- object O class C, définit l'objet O de classe C.
- predicate P (class C1, class C2, ..., class Cn), définit le prédicat P à n variables de classes C1, C2, ..., Cn.
- operator Op (C1 x1, C2 x2, ..., Cn xn) {PP1 (xi, ...), ...} {PA1 (xj, ...), ...} {PD1 (xk, ...), ...}, définit l'opérateur Op à n variables x1, x2, ..., xn de classes C1, C2, ..., Cn, de liste de préconditions {PP1 (xi, ...), ...}, de liste d'ajout {PA1 ...} et de liste de retrait {PD1 ...}.
- init P (O1, O2, ..., On), ajoute le prédicat instancié P (O1, O2, ..., On) à la situation initiale, de même pour goal P (O1, O2, ..., On) avec la situation finale.
- list [class/object/predicate/operator/init/goal], permet d'afficher la liste des classes, objets, prédicats, opérateurs, la situation initiale ou la situation finale.
- plannify, lance la résolution du problème.
- exit, quitte le planificateur.
- # texte, met texte en commentaire.

TAB. 2.1 – Commandes de l'interface

```
class support
class cube
object A class cube
object B class cube
object C class cube
object Table class support
predicate libre (class cube)
predicate sur (class cube, class cube)
predicate sur_s (class cube, class support)
operator empiler (cube x, support y, cube z) {libre (x), libre(z), sur_s(x,y)} {sur (x,\
   z)} {sur_s(x,y),libre(z)}
operator depiler (cube x, cube y, support z) {libre (x), sur (x,y)} {sur_s (x,z), libre\
   (y)} {sur(x,y)}
init sur_s (A,Table)
init sur (C, A)
init sur_s (B,Table)
init libre (C)
init libre (B)
goal libre(A)
goal sur(A,B)
goal sur(B,C)
goal sur_s(C,Table)
```

FIG. 2.5 – Programme définissant le problème des cubes

un algorithme que l'on pourrait qualifier de "STRIPS en chaînage arrière". Il s'agissait en réalité de prendre la situation finale S_{finale}, de créer autant de ramifications dans l'arbre des plans qu'il y a d'actions instanciées A qui auraient pu permettre d'aboutir à S_{finale}, créer chaque situation S antérieure à S_{finale} et recommencer avec chacune des nouvelles situations jusqu'à ce qu'on tombe sur une situation $S \subset S_{initiale}$.

Algorithme 4. STRIPS ***en chaînage arrière***

Pour toute situation S, créer autant de ramifications qu'il existe d'actions A telles que $Add(A) \subset S$, $Del(A) \cap S = \emptyset$ et $Pre(A) \setminus Del(A) \subset S$, créer chaque situation $S_{anterieure}$ associée à chaque ramification telle que $S_{anterieure} \xrightarrow{A} S$. L'algorithme s'arrête lorsqu'on trouve $S \subset S_{initiale}$, alors, on a trouvé un plan linéaire exécutable.

L'erreur apparaît ainsi clairement : on construit bien un arbre de plans, mais un arbre de plans linéaires. D'autre part, ce procédé est très peu intelligent puisqu'il teste à chaque fois toutes les façons d'aboutir à une situation donnée, créant toutes les situations correspondantes, surtout celles qui ne servent à rien, contrairement à l'autre méthode (rechercher les établisseurs pour chaque but, ce qui finit assez rapidement par aboutir à des buts qui sont dans la situation initiale, plutôt que de vouloir aboutir à une situation toute entière incluse dans $S_{initiale}$). Enfin, l'algorithme 4 ne s'arrête que lorsqu'il a trouvé un plan, linéaire, ce qui n'est absolument pas certain, alors qu'avec la planification non-linéaire, on est au moins sûr d'aboutir à un arbre, *exact* qui plus est. Un rapide calcul nous montre finalement que pour un problème qui mettrait en jeu seulement 10 opérateurs instanciés, avec à chaque fois 5 antécédents (cas d'un problème relativement simple) l'arbre de plans pourrait contenir $5^{10} \approx 10\,000\,000$ de cellules et nécessiter pour sa construction un nombre au moins équivalent d'exécutions de l'algorithme.

Cependant, l'ambiguïté réside dans le fait que ce genre d'algorithme résolve assez bien le problème de la figure 1, mais pas de la bonne façon. Tout problème d'interaction qui possède une solution peut ainsi être résolu par l'algorithme 4 (si l'on adjoint une protection anti-bouclage et si l'on postule une puissance de calcul suffisante) :

Theorème 3. *$\forall P$ problème d'interaction, $\forall T$ durée, $\exists (C, M)$ couple (puissance de Calcul, capacité Mémoire) tel qu'un ordinateur de caractéristique (C,M) et utilisant l'algorithme 4 assorti d'une protection anti-bouclage résolve P avant T.*

Pourtant, dès que le problème devient un peu compliqué, pour un T raisonnable, on a besoin d'un (C, M) trop élevé pour ne pas rendre cet algorithme obsolète. D'autre part, on ne peut pas améliorer beaucoup la méthode (bigression, optimisation technique, ...? : les idées manquent vite). Même si j'ai certainement perdu beaucoup de temps dans cette mauvaise voie et que je n'ai pu récupérer qu'une faible partie de ce que j'avais déjà programmé[4], mon erreur ne m'a finalement pas été tant préjudiciable, puisque j'ai pris conscience du véritable enjeu de la planification non-linéaire. D'autre part, j'ai réalisé que le chaînage arrière STRIPS ne préparait en rien un bon algorithme de planification (au mieux, il le complète), contrairement à ce que j'avais cru au début.

[4]Dans la partie planification bien sûr, car l'interface et la gestion de STRIPS sont tout à fait opérationnelles et définitivement efficaces.

2.3.2 L'implémentation de la non-linéarité

Repartant sur de bonnes bases, il a donc finalement fallu programmer l'algorithme 2. Je me reporterai toujours à la figure 2.8 p. 37 pour décrire la progression du programme, en parallèle avec la description algorithmique. On y voit tout d'abord la saisie des données du problème (c'est ici le programme décrit fig. 2.5 p. 31). On lance ensuite la planification, avec plannify.

Avant toutes choses, il faut faire la liste des actions instanciées disponibles, connaissant la liste des objets et la liste des actions (non instanciées) disponibles. Cette liste sera utilisée ensuite tout au long de la planification, car compte tenu des hypothèses, elle ne varie pas. C'est l'opération dénommée :

```
+++ creating available instanciated operators table...
```

Il faut ensuite créer l'arbre de plans partiellement ordonnés. Il y a deux étapes : la création en tant que telle, où l'on applique purement et simplement l'algorithme 2, et le nettoyage de l'arbre, rendu nécessaire par la méthode employée - j'y reviendrai. La figure 2.6 détaille la structure d'une cellule de l'arbre de plans partiels, qui est une situation-action, selon l'amalgame fait jusqu'ici entre les situations et les actions :

- le début de la structure concerne les tableaux décrivant les buts que la situation engendre (`Goal`), ceux qu'elle établit (`Build`) et ceux qu'elle détruit (`Destroyed`).
- la seconde partie de la structure concerne la position dans l'arbre des plans de la situation en question. D'abord, pour chacun des buts i, le nombre d'antécédents `nPrev[i]`, et un pointeur `Prev[i][j]` vers l'antécédent j. Remarquons ici que le *plan* est caractérisé par des ramifications de type `Prev[i]`, i.e. dues aux différents buts qui restent à atteindre, alors que l'*arbre de plan* est défini par des bifurcations de type `Prev[i][j]`, i.e. dues aux diverses méthodes utilisées pour atteindre le même but. Enfin, il y a le nombre de successeurs dans l'arbre `nNext` et un tableau de pointeurs `Next` vers ces situations.
- la dernière partie contient le numéro dans la liste des actions instanciées de l'opérateur auquel correspond la situation-action. Pour finir, il y a le réel `norm` qui correspondra à la pseudo-norme de la cellule dans l'arbre.

Intéressons-nous à la première étape, repérée par :

```
+++ creating partial plans tree...
```

On est là en plein cœur de l'algorithme 2, puisqu'il s'agit d'un procédure récursive qui va, pour chacun des buts de la situation qu'elle étudie, trouver parmi la liste des actions instanciées celles qui ont en liste d'ajout le but en question, créer ensuite les situations correspondantes et réitérer le processus sur chacune des sous-situations (ou *ante*-situations). Il y a deux motifs d'arrêt de la procédure : soit le but est dans la situation initiale (auquel cas cette portion de l'arbre est exacte), soit le but est dans l'ascendance directe de la ramification (auquel cas la protection anti-bouclage impose de couper la branche en cours). Dans le second cas, puisqu'il s'agit d'une procédure récursive, il convient de ne pas se précipiter pour couper la branche, mais de la marquer judicieusement pour pouvoir s'en débarrasser plus tard. Effectivement, si on l'escamote tout de suite, il se peut que la procédure, en

```
typedef struct Situation
{
  int nGoal;
  Predicate *Goal[N];
  Object *vGoal[N][N];
  int nBuild;
  Predicate *Build[];
  Object *vBuild[N][N];
  int nDestroyed;
  Predicate *Destroyed[N];
  Object *vDestroyed[N][N];

  int nPrev[N];
  struct Situation **Prev[N];
  int nNext;
  struct Situation *Next[N];

  int iOpNext;
  double norm;
} Situation;
```

FIG. 2.6 – Structure d'une *situation*

revenant en arrière dans l'arbre (ce qui est le propre de la récurrence), "atterrisse dans du vide". Le rôle de la deuxième étape sera donc de nettoyer l'arbre, à la fin du processus, quand on a un arbre de plans *exacts*, mais comportant des branches "malades" qui restent à couper :

```
+++ cleaning partial plans tree...
```

L'algorithme 2 a donc fourni un arbre de plans exacts, mais comme on l'a vu pour le plan non-linéaire de la figure 1.7 p. 20, de très nombreux établisseurs sont des *multi-établisseurs*, c'est-à-dire qu'ils établissent plusieurs buts à la fois (par exemple, `depiler(C,A,Table)` établit $\texttt{libre(A)}_{\texttt{Fin}}, \texttt{libre(A)}_{\texttt{empiler(A,Table,B)}}$ et $\texttt{sur\_s(C, Table)}_{\texttt{Fin}}$). Pourtant, dans l'arbre dont on dispose en ce moment, une situation (d'action associée `depiler(C,A,Table)`) est reliée à $\texttt{libre(A)}_{\texttt{Fin}}$ et une autre distincte (de même action associée, i.e. de même nombre `iOpNext`) est reliée à $\texttt{sur\_s(C, Table)}_{\texttt{Fin}}$, alors qu'il est évident qu'elles devraient être confondues. Il va donc falloir faire une opération de fusion des établisseurs :

```
+++ fusionning establishers...
```

qui consistera à regrouper dans un tableau les établisseurs potentiellement fusionnables, en procédant par couches successives et en commençant par la partie droite de l'arbre (situation finale), puis à les regrouper. Certes, rien n'assure que l'on ne commette pas d'erreur en fusionnant de la sorte, c'est-à-dire fusionner certains établisseurs qui ne devraient pas l'être. Néanmoins, on pourra toujours corriger cela plus tard, au moment de la linéarisation, avec des *chevaliers blancs*. D'autre part, il est impensable de laisser des multi-établisseurs non fusionnés dans l'arbre puisqu'ils ne pourront jamais être fusionnés après (ce n'est pas le rôle du chevalier blanc) : on passerait ainsi à côté d'une étape essentielle de la recherche

de plans[5]

On a ainsi obtenu un arbre amélioré de plans exacts, prêts à être exploités. Il va maintenant falloir travailler sur des plans en particulier, et donc extraire les plans de l'arbre, pour les traiter ; mais par lequel commencer ? Puisqu'on ne va pas regarder tous les arbres, et qu'il y en a certains qui présentent plus d'intérêt que d'autres, on va devoir se donner des règles pour choisir quel plan on va commencer par extraire. Pour cela, on va donner une pseudo-norme à chaque situation-action, grâce à une heuristique de deux paramètres ($H(g(S), s(S)) = \frac{g(S)}{1+s(S)}$, où $g(S)$ représente le nombre de buts que S ajoute ($card(Pre(S) \setminus S_{initiale})$), et $s(S)$ le nombre de ceux qu'elle établit).

`+++ normizing partial plans tree nodes...`

Lorsqu'on a une bifurcation dans l'arbre des plans, on choisira ainsi la voie qui mène à la situation de plus faible norme. Au final, on garde le plan de plus faible "norme totale" (chemin de moindre poids). Si l'on échoue en utilisant ce plan, on pourra toujours prendre le suivant, et ainsi de suite... L'extraction, marquée par :

`+++ extracting first partially ordonned plan...`

nécessite la création d'un arbre secondaire, indépendant de l'arbre de plans principal. D'autre part, cet arbre va posséder une hiérarchie, ce qui requiert un nouveau type de structure qui pourra contenir des informations sur les relations d'ordre entre les différentes situations-actions : puisque la hiérarchie qu'on va construire sera propre à chaque plan extrait, il va falloir y associer un arbre d'*e-situations* (cf. fig. 2.7 page suivante). Chacun de ces arbres sera donc un plan en soi, mais pour ne pas dupliquer toutes les informations contenues dans l'arbre de plans principal, on inclut dans une *e-situation* un pointeur vers la situation correspondante dans l'arbre des plans (`ref`). Une *e-situation* contient d'autre part une liste de pointeurs dits "avant" (`Before`), pointant vers celles qui sont après elle dans le plan (les *e-situations* E telles que $E \prec A$) , et une liste de pointeurs "après" (`After`), désignant ceux qui sont avant ($A \prec E$).

`+++ creating plan hierarchy...`

Une fois le premier plan extrait, on va pouvoir définir l'ordre zéro-partiel :

`-- zero-partially ordonned plan`

ce qui consiste d'abord à mettre $S_{initiale}$ dans la base des `Before` et S_{finale} dans celle de `After`, puis à regarder pour un situation-action S donnée, qui est après elle (branches `Next`) ou avant elle (branches `Prev`) dans l'arbre. Après avoir examiné toutes les situations, on obtient donc un arbre d'*e-situations*, qui est pour le moment seulement *exact*. On peut ensuite établir l'ordre un-partiel :

[5]Il suffit pour s'en convaincre de reprendre le plan non-linéaire fig. 1.7 : si, au lieu de les fusionner, on laisse les trois établisseurs `depiler(C,A,Table)` indépendants, aucune autre étape ne permettra de s'en débarrasser ou de les réunir après ; il faudra alors trouver un plan linéaire qui comporte ces trois établisseurs, donc une solution au problème des cubes avec 5 actions au moins (ces trois-là, plus `depiler(A,Table,B)` et `empiler(B,Table,C)`). Comme il n'existe pas de plans comportant ces cinq actions et qui soit solution du problème, on devra recourir au *white knight*, qui va encore rajouter pas moins de 2 opérateurs pour corriger, soit un plan linéaire final contenant au minimum 7 actions. Même si on *finit* par trouver une solution, on est très loin des trois actions de départ qui, bien ordonnées, suffisent (cf. fig. 1.2).

```
typedef struct Esituation
{
  Situation* ref;
  struct Esituation* Prev[N];
  int nNext;
  struct Esituation* Next[N];

  int nBefore;
  int nAfter;
  struct Esituation* *Before;
  struct Esituation* *After;
} Esituation;
```

FIG. 2.7 – Structure d'une *e-situation*

```
-- one-partially ordonned plan
```

L'algorithme que j'ai employé ici pour un-ordonner est assez simple, puisqu'il consiste à regarder ce que chaque e-situation E détruit, et qui cela gêne : en l'occurrence, $\forall P \in Destroyed(E)$, si $\exists E'$ telle que $P \in Goal(E')$, alors $E' \prec E$, ce que l'on traduit en mettant E' dans la liste `Before`(E) et E dans la liste `After`(E'). En réitérant ce processus pour toutes les e-situations, on est assuré d'obtenir un plan un-partiellement ordonné (ou *cohérent*). En réalité, la méthode est même un peu brutale, car on déplace tous les casseurs potentiels après les actions dont ils cassent les buts, sans distinguer les cas où l'on pourrait les déplacer avant, en intercalant entre les deux une action qui rétablirait le but cassé, ou même les cas où il y a déjà une telle action entre le casseur et l'action. Cet algorithme serait donc à améliorer dans une future version. Toutefois, disposant de notre plan un-partiel, il ne reste plus qu'à tenter de le linéariser :

```
+++ linearizing one-partially ordonned plan...
```

Cette étape est assez simple compte tenu des hypothèses, si l'on n'essaye pas d'appliquer la méthode du *chevalier blanc*, qui reste aussi à implémenter dans une prochaine version. En effet, il faut transformer l'ordre partiel du plan en ordre total : si c'est possible, la définition de la *cohérence* nous assure que le plan linéaire trouvé sera exécutable. Si c'est impossible, soit le programme doit s'attaquer à un autre plan de l'arbre de plans (extraire le second et le traiter, par exemple), soit il tente d'arranger le problème avec le *white knight*, mais cette dernière solution n'est vraiment efficace que si l'on rend plus fin l'algorithme précédent d'ordonnancement un-partiel. Dans le cas des cubes, on voit sur la figure 2.8 que cette succession d'opérations amène finalement à la solution du problème d'interaction :

```
--- listing possible execution...
initial -> 1/goal -> 3/goal -> 2/goal -> goal.
    in other words:
depiler (C, A, Table), empiler (B, Table, C), empiler (A, Table, B).
```

```
Planning system v.1.0 (interactive mode)
---------------------- non-linear planner (c) Camille ROTH 1999
> class support
+++ added class support.

(...)

> list goal
[ libre (A) ] [ sur (A, B) ] [ sur (B, C) ] [ sur_s (C, Table) ]
> plannify
+++ creating available instanciated operators table... done.
+++ creating partial plans tree... done.
+++ cleaning partial plans tree..... done.
+++ fusionning establishers... done.
+++ normizing partial plans tree nodes... done.
+++ extracting first partially ordonned plan... done.

[goal]
 |- 1. libre (A)
   [1/goal] depiler (C, A, Table) -> {libre (A), sur_s (C, Table)}
    |- 1. libre (C)
    \- 2. sur (C, A)
 |- 2. sur (A, B)
   [2/goal] empiler (A, Table, B) -> {sur (A, B)}
    |- 1. libre (A)
      [1/goal] depiler (C, A, Table) -> {libre (A), sur_s (C, Table)}
       |- 1. libre (C)
       \- 2. sur (C, A)
    |- 2. libre (B)
    \- 3. sur_s (A, Table)
 |- 3. sur (B, C)
   [3/goal] empiler (B, Table, C) -> {sur (B, C)}
    |- 1. libre (B)
    |- 2. libre (C)
    \- 3. sur_s (B, Table)
 \- 4. sur_s (C, Table)
   [1/goal] depiler (C, A, Table) -> {libre (A), sur_s (C, Table)}
    |- 1. libre (C)
    \- 2. sur (C, A)

+++ creating plan hierarchy...
   -- zero-partially ordonned plan
   -- one-partially ordonned plan
+++ linearizing one-partially ordonned plan... done.
--- listing possible execution...
initial -> 1/goal -> 3/goal -> 2/goal -> goal.
    in other words:
depiler (C, A, Table), empiler (B, Table, C), empiler (A, Table, B).
--- planning done.
> exit
--- terminated.
```

FIG. 2.8 – Exécution du planificateur avec le problème des cubes

Chapitre 3

Ouvertures...

Aussi complexe puisse-t-il sembler, ce programme ne s'attaque pourtant que très partiellement au problème même de la planification (cf. déf. 5 p. 13), à savoir instaurer des contraintes dans le problème de l'interaction, c'est-à-dire le résoudre *d'une certaine façon*. Il faudrait pour cela étendre STRIPS et ajouter des commandes permettant de préciser ces contraintes, par exemple permettre de valuer une action (temps nécessaire pour l'effectuer) et avoir comme contrainte la minimisation de cette valeur, à l'échelle globale du plan (le moins de temps possible).

D'autre part, la modélisation STRIPS n'est pas suffisante en soi pour représenter tous les environnements possibles, ni même leur évolution en fonction du temps. Aucune action de type STRIPS ne permet par exemple de modifier intrinsèquement les objets, elles se bornent à modifier les propriétés des objets, via les prédicats. Prenons le cas d'un objet "paire de ciseaux", quel opérateur STRIPS pourrait le transformer en deux objets "ciseau 1" et "ciseau 2" ? Il n'en existe pas. On peut, pour résoudre cette impossibilité, penser à l'ADL d'Edwin Pednault, qui ajoute un champ *Update* à la description STRIPS d'une action, champ qui décrit une modification du langage lui-même, des objets aux opérateurs. Il reste très difficile à mettre en œuvre, puisqu'il faudrait déjà associer à chaque situation-action un état du langage de description (celui-ci étant désormais variable, il est propre à chaque action). Une amélioration aussi faible des outils amène donc très rapidement un très gros travail de programmation, et rien n'assure non plus que ce soit la bonne méthode : d'autres manières de modéliser le problème, radicalement différentes, existent avec des résultats plus ou moins probants (comme les *éco-agents*, que je ne décrirai pas ici).

Enfin, le plus gros travail reste à essayer d'améliorer l'algorithme 3, notamment au niveau de la linéarisation et de l'utilisation du *white knight*. Un certain travail de recherche théorique est nécessaire avant de tenter de programmer concrètement un nouvel algorithme plus efficace, travail en relation avec la notion de décidabilité d'un problème : existe-t-il une solution, et si oui, peut-on la trouver, ou bien, peut-on trouver un algorithme qui résolve à coup sûr le problème si c'est possible, ou qui puisse affirmer assurément que le problème n'a pas de solution ?

Table des figures

Bibliographie

[1] James F. Allen, Neal Lesh et Nathaniel Martin, *Improving Big Plans*, Department of Computer Science, University of Rochester (1997).

[2] James F. Allen et George Ferguson, *TRIPS : An Integrated Intelligent Problem-Solving Assistant*, Department of Computer Science, University of Rochester (1998).

[3] Eric Jacopin, *Algorithmique de l'interaction : le cas de la planification*, Thèse de doctorat de l'Université Paris VI **TH93/11** (Septembre 1993).

[4] Eric Jacopin, *Structures in Partial Plan Space Planning*, in *Structural Issues in Planning and Temporal Reasoning* (13th National Conference on AI, Portland, Oregon, Août 1996).

[5] Dana Nau, Stephen J.J. Smith et Thomas Throop, *AI planning systems in the real world*, IEEE Expert (Décembre 1996).

[6] Steve Woods, *Planning for Conjunctive Goals : The State of the Art and Ahead*, University of Waterloo (Novembre 1989).