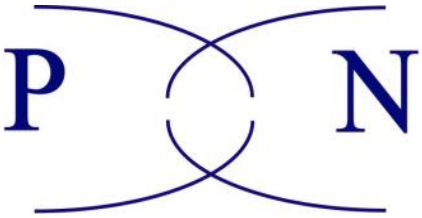


PACTE NOVATION
42, rue du Dr. Lombard
92441 Issy-Les-Moulineaux

EPITA
14/16, Rue Voltaire
94276 Le Kremlin Bicetre

Jacques Couvreur,
Promotion SCIA 2000,
Stage de Fin d'Etudes
Du 3 janvier au 30 juin 2000

Generation de Plans d'Actions

Evaluer l'apport du nouvel algorithme de génération de plans d'actions *Graphplan*, par rapport aux algorithmes existants, et l'implémenter en C++ afin de le tester sur des cas simples de planification

«

Il y a ceux qui font quelque chose:
Ils sont trois qui font quelque chose.

Il y a ceux qui ne font rien:
Il sont dix qui font des conférences.

Il y a ceux qui croient faire quelque chose
Et ils sont cent qui font des conférences
Sur ce que disent les dix
De ce que font les trois qui font quelque chose.

Et il arrive que l'un des cent dix vienne expliquer
La manière de faire à l'un des trois qui font quelque chose.

Alors l'un des trois intérieurement s'exaspère,
Et extérieurement sourit,
Mais il se tait car il n'a pas la parole.
D'ailleurs, il a quelque chose à faire...

Que les trois qui font me pardonnent… »

Je tiens à remercier,
Cécile, pour sa visite guidée d'Issy,
Sylviane, pour sa relecture consciencieuse et efficace,
Philippe Morignot, pour sa confiance et son soutien,
Fréderic Leroy, pour la qualité de ses cours,
Et Akli Adjaoute, pour ses conseils sur l'existence qui se résument en « être pour soi, et non pour les autres » et « *think different* ».

De plus, ce rapport étant dans l'immédiat ma dernière obligation scolaire, je saisis l'opportunité de remercier également tous les enseignants qui m'ont accompagné durant ma scolarité, et qui ont contribué à ce que je suis aujourd'hui, donc à ce rapport…

I.

Résumé

Résumé

Dans le cadre de la dernière année de mon cursus scolaire à l'EPITA (Ecole Pour l'Informatique et les Techniques Avancées) je devais effectuer un stage de six mois qui soit en rapport avec ma spécialisation, SCIA (Sciences Cognitives et Intelligence Artificielle). Il s'agit en effet, du moyen idéal de mettre en pratique les connaissances acquises tout au long de la formation dispensée à l'Ecole. De plus, cette expérience permet de nous familiariser davantage avec le monde du travail en entreprise.

Le présent document a pour intention de rapporter ce qui a été fait durant ce stage, qui s'est déroulé dans la société Pacte Novation du 3 janvier au 30 juin 2000, société avec laquelle j'ai pris contact au salon ProSearch à la Défense (92).

Cette société d'Issy-Les-Moulineaux (92), créée en avril 1994, est une société de Conseil et d'Ingénierie en Informatique spécialisée dans l'utilisation et l'intégration des **Techniques d'Informatique Avancée**. Elle est actuellement composée d'une cinquantaine d'ingénieurs.

Ses différents clients lui permettent de couvrir de nombreux domaines : l'industrie métallurgique (SOLLAC…), l'énergie (EDF…), le transport (GEC Alstom…), l'industrie de l'informatique et des télécommunications (Alcatel, Bull…) ou encore la finance des salles de marché (Crédit Agricole Indosuez…).

Afin de répondre aux attentes de ses clients, Pacte Novation a diversifié ses compétences réparties sur trois axes :

- la communication homme/machine, incluant l'ergonomie et la réalisation d'IHM,
- les technologies orientées objets et distribuées,
- et enfin l'aide à la décision, basée sur des systèmes à base de connaissances, et **l'optimisation** en allocation de ressources et dans l'ordonnancement, notamment, avec l'utilisation de Programmation Par Contrainte.

C'est au niveau de ce dernier point que se situe mon stage, et plus précisément dans la **problématique de la planification**.

La génération de plans d'actions constitue un domaine de l'Intelligence Artificielle en pleine expansion, comme l'attestent les nombreuses publications parues depuis 30 ans dans les conférences générales et dédiées. Vous en aurez un aperçu à la fin de ce rapport en consultant la bibliographie et la présentation non exhaustive des acteurs de la planification.

La problématique peut se résumer ainsi : étant donné un ensemble d'actions décrites dans un langage proche de la logique des prédicats du premier ordre, étant donné une description d'un état initial et d'une description de buts finaux, trouver un plan non linéaire (une

séquence de séquences) d'actions amenant l'état initial dans un état où les buts finaux sont satisfaits. Ses applications pratiques sont par exemple l'organisation d'opérations militaires de type « force déterminée », l'organisation embarquée des actions de propulsion et radar d'une sonde spatiale passant devant Jupiter ou la planification des tâches de grands projets industriels.
Ce problème a été montré NP-complet.
Ce domaine a connu un renouveau depuis deux ans par la découverte d'un algorithme de planification très performant, GRAPHPLAN. Par exemple, dans la meilleure implémentation connue à ce jour de cet algorithme, un plan d'une centaine d'actions est généré en 6 minutes sur une station SUN.

De manière concrète, **mon stage** s'est réalisé comme suit.
Les deux premiers mois, j'ai tout d'abord évalué l'apport de ce nouvel algorithme par rapport aux algorithmes existants jusqu'alors.
Cette **étude bibliographique** et comparative m'a permis d'établir un modèle objet, saisi à l'aide de *Rational rose 98*, de ce que sera mon implémentation. J'ai ensuite exploré la voie des algorithmes de recherche locale (Algorithme A*, Recuit Simulé et Recherche Tabou), afin de voir leur éventuelle utilité dans cette problématique.
Ce qui m'a permis de conclure qu'il était préférable de s'orienter dans un premier temps vers une approche **Programmation Par Contraintes**, et qu'une implémentation ultérieure des méta-heuristiques serait intéressante à titre comparatif.
Le choix de la Programmation Par Contraintes ayant été fait, je me suis donc penché sur l'étude d'un moteur de propagation de contraintes. N'ayant le temps durant mon stage de six mois d'implémenter également ce moteur, je devais en trouver un préexistant. Pacte Novation étant partenaire de la société française *ILOG*, elle a déjà éprouvé son moteur de PPC, c'est donc tout naturellement que mon choix s'est porté sur ce dernier, *ILOG Solver* version 4.31. C'est en fait un composant C++ qui s'intègre sous forme de librairie statique au projet, que j'ai développé sous *Microsoft Visual C++ 6.0*. avec une approche des *Microsoft Fundation Class* (*MFC*).
La fin de mon stage a donc consisté à **écrire l'algorithme de génération de plans d'actions**, qui fonctionnait sur l'exemple du Monde Des Cubes. Les entrées sont alors une configuration initiale de cubes et une configuration finale (à savoir les buts à atteindre), exprimées toutes deux sous forme d'un ensemble de propositions : « (CubeA on CubeB) (CubeC Clear) … ». Les actions possibles, de la forme « move_CubeA_from_CubeB_to_CubeC », font aussi partie des entrées. En sortie, on doit obtenir un plan d'actions, à savoir une séquence d'actions.

Les deux premières semaines de juin on été consacrées à la rédaction de ce rapport, et à la réalisation du site Web associé et imposé par l'Ecole.

Les jours restants jusqu'à la fin du mois et donc du stage, seront employés :

- à l'amélioration des résultats, notamment par l'implémentation de fonctionnalités supplémentaires,
- à la préparation de ma soutenance du 6 juillet à l'Ecole,
- et enfin, à l'élaboration d'un papier en vue de le présenter à la conférence « The 19th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group », qui se déroulera les 14 et 15 décembre 2000 à The Open University, Milton Keynes, Buckinghamshire (UK). Cela implique des résultats de qualité et un rapport soumis avant le 17 septembre 2000.

La conclusion de ce stage est plutôt positive.

Seules ombres au tableau,

- l'absence de travail en groupe, avec un chef de projet omniprésent, ne m'ont pas permis d'améliorer ma méthodologie, apparemment faible, comme l'a souligné mon maître de stage,
- et la non implémentation de la totalité des fonctionnalités qu'il était prévu au départ.

Sinon, pour le reste, que ce soit pour ce qui est du stage en lui même,

- algorithme terminé,
- résultats satisfaisants,
- apprentissage continu sur le plan théorique (bibliographie)
- ou technique (Rational Rose, ILOG Solver et Microsoft Visual C++),
- prise de contact du travail en entreprise,
- développement du sens de l'autonomie…

ou des débouchés,

- opportunité de soumettre mes résultats dans une conférence internationale sur le domaine,
- proposition d'un contrat d'embauche par la société,

ce stage est une réussite.

Abstract

For my final year of computer engineering at EPITA (Computing School For Advanced Technologies) specialization SCIA (Cognitive Sciences and Artificial Intelligence) promotion 2000, I carried out a 6 months internship, from January to June 2000 at PACTE NOVATION.
The object of this document is a summary of what I have done during this internship.

This company located at Issy-Les-Moulineaux (92) and created in April 1994, is a Consulting and Engineering company, specialized in the use and the integration of the techniques of Advanced Data processing. It is currently made up of about sixty engineers. PACTE NOVATION's various customers allow it to cover many fields: metallurgical industry (SOLLAC...), energy (EDF...), transport (GEC Alstom...), telecommunications (Alcatel...), hi-tech industries and services (Bull...) or financial banking (Credit Agricole Indosuez...). In order to answer the needs of its customers, PACTE NOVATION diversified its competences distributed on three axes:

- the man/machine communication, including ergonomics and realization of MMI,
- objects oriented and distributed technologies,
- and finally Decision Support software, based on knowledge based systems, and optimization in resource allocation and scheduling, in particular, with the use of Constraint Programming.

My internship is based mainly on this last point, and more precisely on planning problems. The generation of action plans constitutes a field of Artificial Intelligence in full expansion, as have attested it many publications published for 30 years in the general and dedicated conferences. **The problems can be summarized as follows**:

- being given a set of actions described in a language close to the logic of the first order predicates,
- being given a description of an initial state and a description of final goals,
- to find a plan nonlinear (a sequence of sequences) of actions bringing the initial state in a state where the final goals are satisfied.

Practical applications are for example the organization of military operations of type " forces given ", the embarked organization of the actions of propulsion and radar of a space probe passing in front of Jupiter or the planning of the tasks of great industrial projects.

This problem is showed as being NP-complete.

This domain has known a revival since two years by the discovery of a very powerful algorithm of planning, GRAPHPLAN. For example, in the best known implementation of this algorithm, a plan of a hundred actions is generated in 6 minutes on a SUN station.

In a practical way, **my internship has been carried out as follows**.

The first two months, I first evaluated the contribution of this new algorithm compared to the existing ones.

This bibliographical and comparative study enabled me to establish a model object, using *Rational Rose 98*, of what will be my implementation. I then explored the way of local search algorithms (A* algorithm, Simulated Annealing and Taboo Search), in order to see their possible utility in this problematic. That enabled me to conclude that it was preferable to initially focus the approach on Constraint Programming, and that a later implementation of méta-heuristic would be interesting a purely comparative basis.

Once the choice of Constraints Programming done, I focus the study of an constraint propagation engine. Due to my six months internship, I did not have enough time to implement this engine. Moreover this implementation was not the subject of my internship. As to the partnershiop between PACTE NOVATION and ILOG, the Constraints Propagation Engine developped by this latter, named *Ilog Solver 4.31*, already used for many other projects in-house was my natural choice. It is in fact a C++ component which is integrated as a static library into the project, that I developed under *Microsoft Visual C++ 6.0*.

Thus the end of my internship consisted in writing **the action plans generation algorithm**, which functioned on the example of the « World Of Cubes ».

The inputs are :

- an initial configuration of cubes
- and a final configuration (namely the goals to reach), expressed both as a set of proposals: " (CubeA one CubeB) (CubeC Clear)... ".
- The possible actions, of the form " move_CubeA_from_CubeB_to_CubeC ", are also part of the inputs.

As output, one must obtain an action plan, namely a sequence of actions. The algorithm operates as follows.

It takes the whole of the proposals which define the current state (at the beginning it is about the initial state), applies all the possible actions to the latter, in order to obtain a new state, which can possibly contain contradictory propositions. If this new state does not contain at least all the final goals, it starts again this process. If not, it creates a Solver variable for each proposal of each generated state, and the Solver constraints generating by the constraints of the problem. Then, it launches the Solver solution generation, which seeks

an action to support each proposal. If all actions are found, then it is solution ; if not it starts again the process.

The first two weeks of June were devoted to the drafting of this report, and the realization of a Web site associated to it.

The remaining days until the end of the month and thereafter of the internship, will be devoted:

- to improving the results, in particular by implementing additional functionality,
- to preparing my orals on the 6th of July,
- and finally, to writing an another report in order to present it to « The 19th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group » which will be held at The Open University, Milton Keynes, Buckinghamshire (UK), on the 14th-15th December 2000. That implies results of quality and a report submitted before the 17th September 2000.

The conclusion of my internship contain more positive aspects than negatives ones.

The positives points are :

- finished algorithm,
- satisfactory results,
- continuous training on the theoretical level (bibliography),
- technical training (Rational Rose, ILOG Solver and Microsoft Visual C++),
- acquainting to company working constraints,
- and development of my autonomy.

The only shades in the table, to leave for improves :

- the absence of group working and of an omnipresent project leader did not enable me to improve my methodology, apparently weak, as underlined it my internship master,
- and not all the functionality requested in the beginning were implemented.

As a result, my work may be submitted to an international conference on planification, and finally I have been proposed a working contract.

II.

Sommaire

Sommaire

Table des Matières

Table des figures

III.

Introduction

Dans le cadre de la dernière année de mon cursus scolaire à l'EPITA (Ecole Pour l'Informatique et les Techniques Avancées) je devais effectuer un stage de six mois en rapport avec ma spécialisation, SCIA (Sciences Cognitives et Intelligence Artificielle). Il s'agit en effet, du moyen idéal de mettre en pratique les connaissances acquises tout au long de la formation dispensée à l'Ecole. De plus, cette expérience permet de nous familiariser davantage avec le monde du travail en entreprise.

Le présent document a pour intention de rapporter ce qui a été fait durant ce stage qui s'est déroulé dans la société Pacte Novation du 3 janvier au 30 juin 2000, société avec laquelle j'ai pris contact salon ProSearch, à la Défense (92).

Cette société d'Issy-Les-Moulineaux (92), créée en avril 1994 et actuellement composée d'une cinquantaine d'ingénieurs, est une société de Conseil et d'Ingénierie en Informatique spécialisée dans l'utilisation et l'intégration des **Techniques d'Informatique Avancée**.

Afin de répondre aux attentes de ses différents clients, couvrant de nombreux domaines (l'industrie métallurgique, l'énergie, le transport, l'industrie de l'informatique et des télécommunications ou encore la finance des salles de marché), Pacte Novation a diversifié ses compétences réparties sur trois axes :

- la communication homme/machine, incluant l'ergonomie et la réalisation d'IHM,
- les technologies orientées objets et distribuées,
- et enfin l'aide à la décision, basée sur des systèmes à base de connaissances, et **l'optimisation** en allocation de ressources et dans l'ordonnancement, notamment, avec l'utilisation de Programmation Par Contraintes.

C'est dans ce dernier domaine que se situe mon stage, et plus précisément dans le cadre de la planification.

La génération de plans d'actions constitue un domaine de l'Intelligence Artificielle en pleine expansion, comme l'attestent les nombreuses publications parues depuis 30 ans dans les conférences générales et dédiées.

Ses **applications pratiques** sont par exemple l'organisation d'opérations militaires de type « force déterminée », l'organisation embarquée des actions de propulsion et radar d'une sonde spatiale passant devant Jupiter ou la planification des tâches de grands projets industriels.

La **problématique** peut se résumer ainsi :

- étant donné un ensemble d'actions décrites dans un langage proche de la logique des prédicats du premier ordre,
- étant donné une description d'un état initial
- et une description de buts finaux,
- trouver un plan non linéaire (une séquence de séquences) d'actions amenant l'état initial dans un état où les buts finaux sont satisfaits.

Pour y parvenir, il est nécessaire de trouver une structure de donnée capable de représenter un état, les actions, et les buts, mais aussi un algorithme performant capable de les manipuler efficacement, même, et surtout, lorsqu'ils deviennent nombreux.
Pour cela, une étude comparative de ce qui a déjà été fait dans ce domaine, des avancées récentes et de l'intérêt éventuel que peuvent présenter certaines techniques de l'Intelligence Artificielle, est au préalable souhaitable.
C'est précisément les deux étapes de mon stage.

Ce rapport débute par une présentation ciblée de Pacte Novation.
Il reprend ensuite les résultats de l'étude bibliographique initiale et la conclusion déduite.
Il se poursuit par la description point par point de l'algorithme développé et de la structure de données sur laquelle repose ce dernier. Le rapport présente également le composant logiciel *ILOG Solver* qui a servis à implémenter algorithme.
Il se conclue par une discussion sur les résultats obtenus, les prolongements éventuels de ce stage et enfin, ce qu'il m'a apporté.
Ce rapport contient également en annexe,

- une présentation des acteurs jouant un rôle moteur dans le domaine de la planification, afin de vous donner un aperçu de l'importance et de la place que prend actuellement la planification dans la recherche et le développement en informatique,
- et une description des principales classes de *Solver*, extraite de *ILOG Solver 4.3, Reference Manual*.

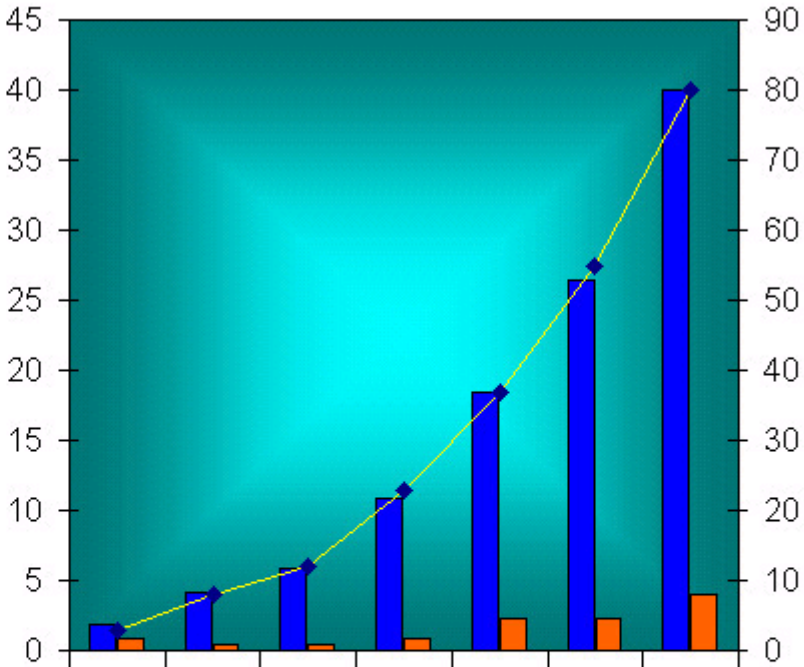
IV.

Présentation de l'entreprise

PACTE NOVATION, entreprise créée en avril 1994, est une Société de Conseil et d'Ingénierie en Informatique spécialisée dans l'utilisation et l'intégration des techniques d'Informatique Avancée. Elle est actuellement composée d'une cinquantaine d'ingénieurs. Son chiffre d'affaire est d'environ 25 millions de francs pour un résultat net de plus de deux millions.

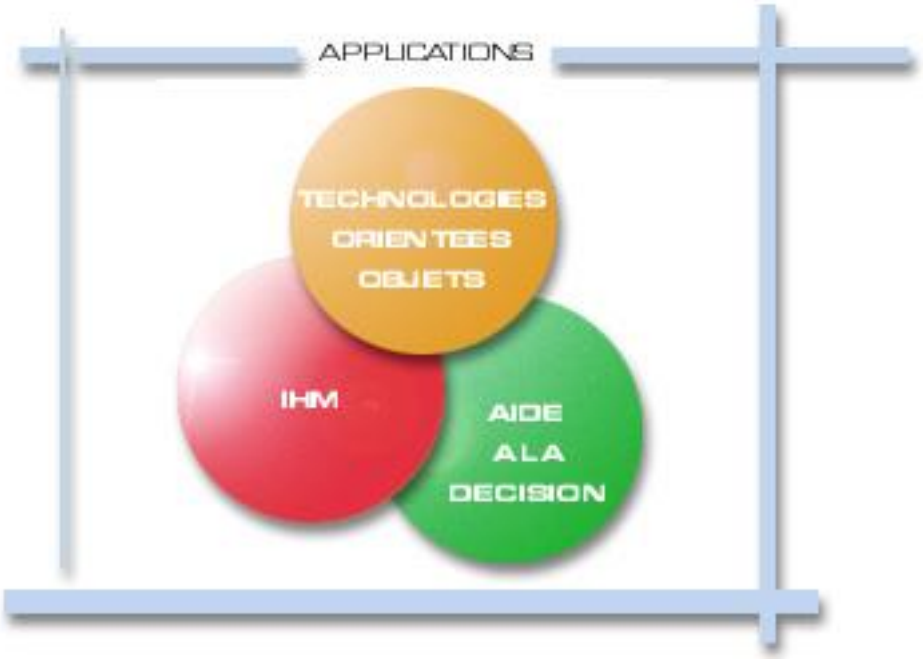
Deux associés sont à la tête de PACTE NOVATION : Christian Tora, Président Directeur Général, et Bruno Gaudinat, Directeur Technique.

Chiffres d'affaire et résultats nets



	94/95	95/96	96/97	97/98	98/99	99/00	00/01
CA (MF)	1,8	4,1	5,8	10,8	18,5	26,5	40
Résultat net (MF)	0,8	0,41	0,45	0,9	2,3	2,3	4
Effectif moyen	3	8	12	23	37	55	80

Domaines de compétences



Les compétences de PACTE NOVATION sont réparties sur trois axes :

- **La communication homme/machine (incluant l'ergonomie et la réalisation d'IHM) :**
  Prise en compte du facteur humain lors de l'informatisation de processus
  Analyse de la tâche et spécification des Interfaces Homme/Machine
  Evaluations et recommandations ergonomiques sur des IHM existantes
  Réalisation d'IHM graphiques et d'outils utilisant le Langage Naturel

- **Les technologies orientées objets et distribuées :**
  Conception et réalisation d'applications à base de technologies orientées objet
  Conseil et pratique des méthodes orientées objets

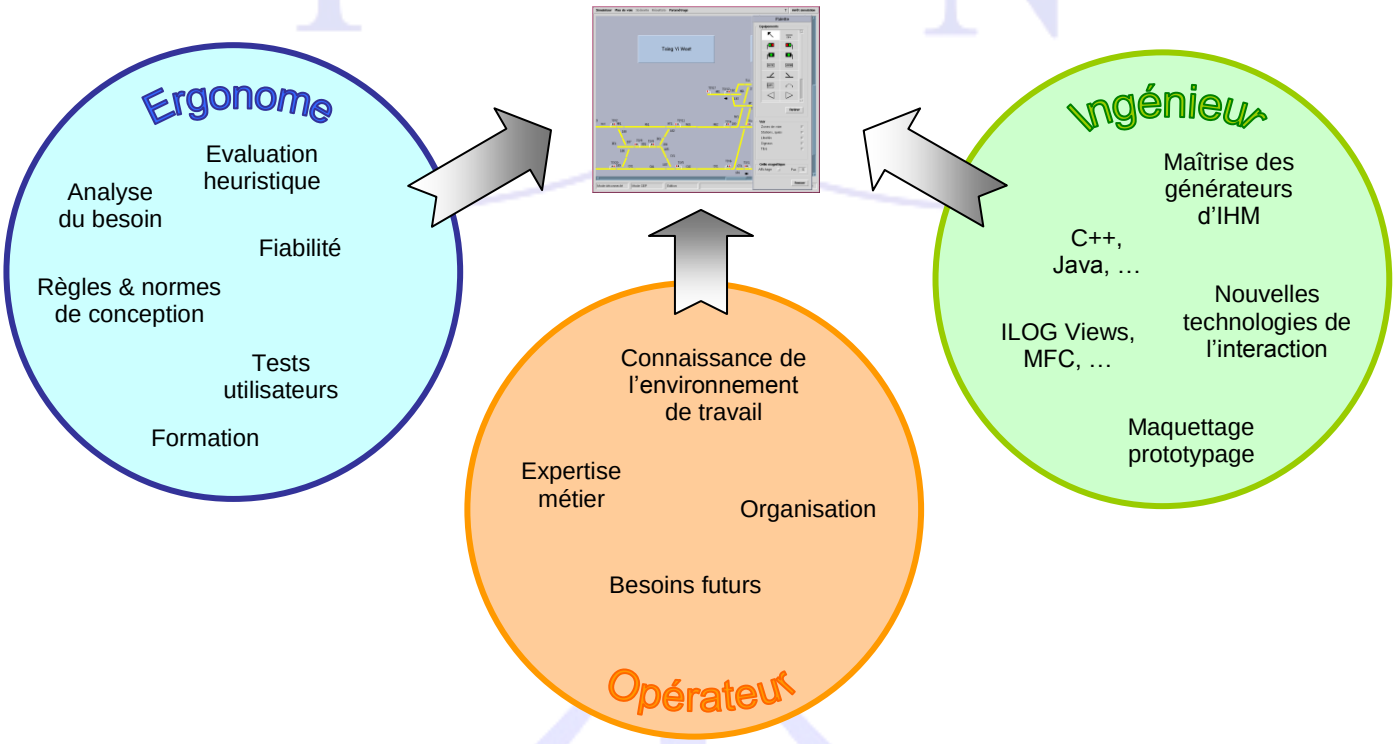
- **L'aide à la décision :** résolution de problèmes intégrant des raisonnements d'experts, ayant une forte complexité ou une grande combinatoire :
  - *Systèmes à Base de Connaissances :* spécification, conception et développement de systèmes à base de connaissances, modélisation et capitalisation de savoir-faire
  - *Systèmes à Base de Contraintes :* allocation de ressources, planification et ordonnancement, optimisation de processus, ...

Signalons enfin que PACTE NOVATION est membre du Comité Richelieu, dont la mission est de promouvoir les petites et moyennes entreprises innovantes, dans le milieu des hautes technologies, au sein de la D.G.A. et d'autres organismes publics.

La Communication Homme / Machine

Le pari de l'intégration …

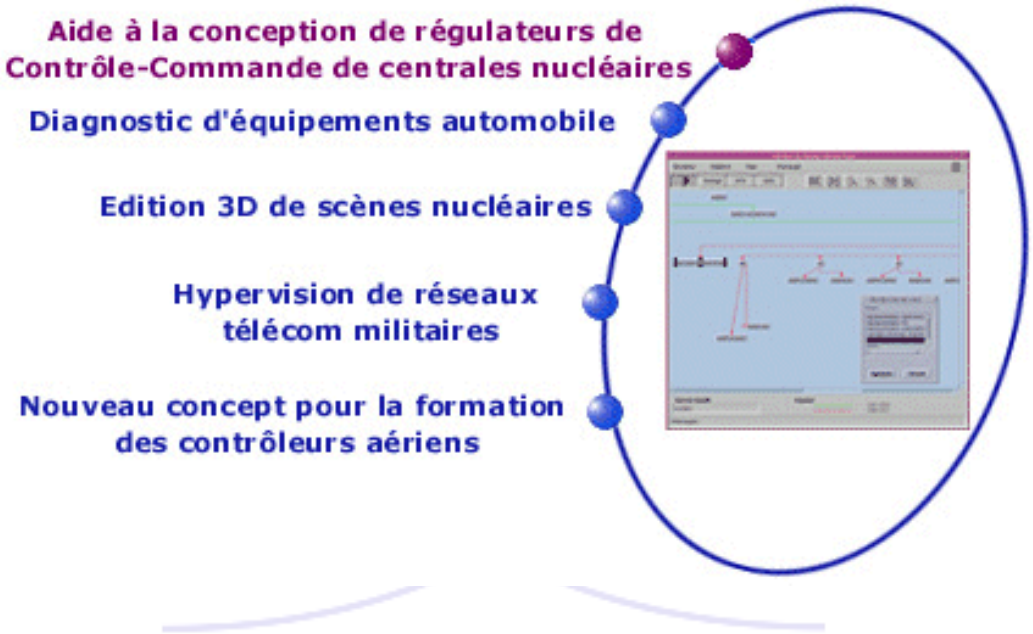
Le domaine des interfaces Homme / Machine est par essence pluridisciplinaire : ergonomes, cogniticiens, graphiste, ingénieurs, opérationnels … Réussir une IHM, c'est réussir l'intégration de ces disciplines.



Les moyens de la réussite …

Sa philosophie de l'intégration a forcément des implications méthodologiques. Celles-ci ont été explorées, expérimentées, formalisées et finalement « outillées » (voir MATOS) pour devenir une réalité de sa pratique quotidienne.

Des réalisations concrètes (*)



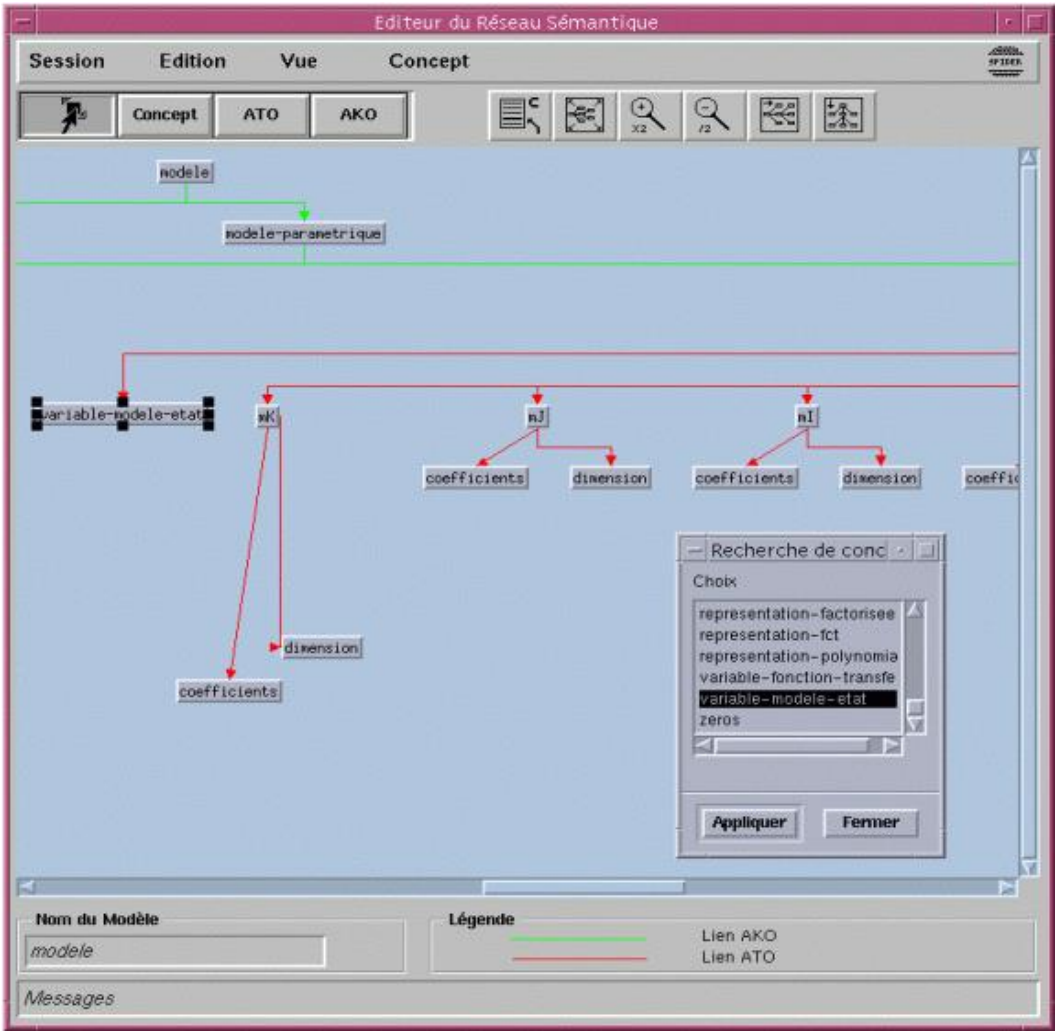


Figure 1: *IHM d'un logiciel d'aide à la conception de régulateurs de Contrôle-Commande de centrales nucléaires.*

Pacte Novation a également implémenté la partie fonctionnelle de ce projet sur la base d'un système expert.

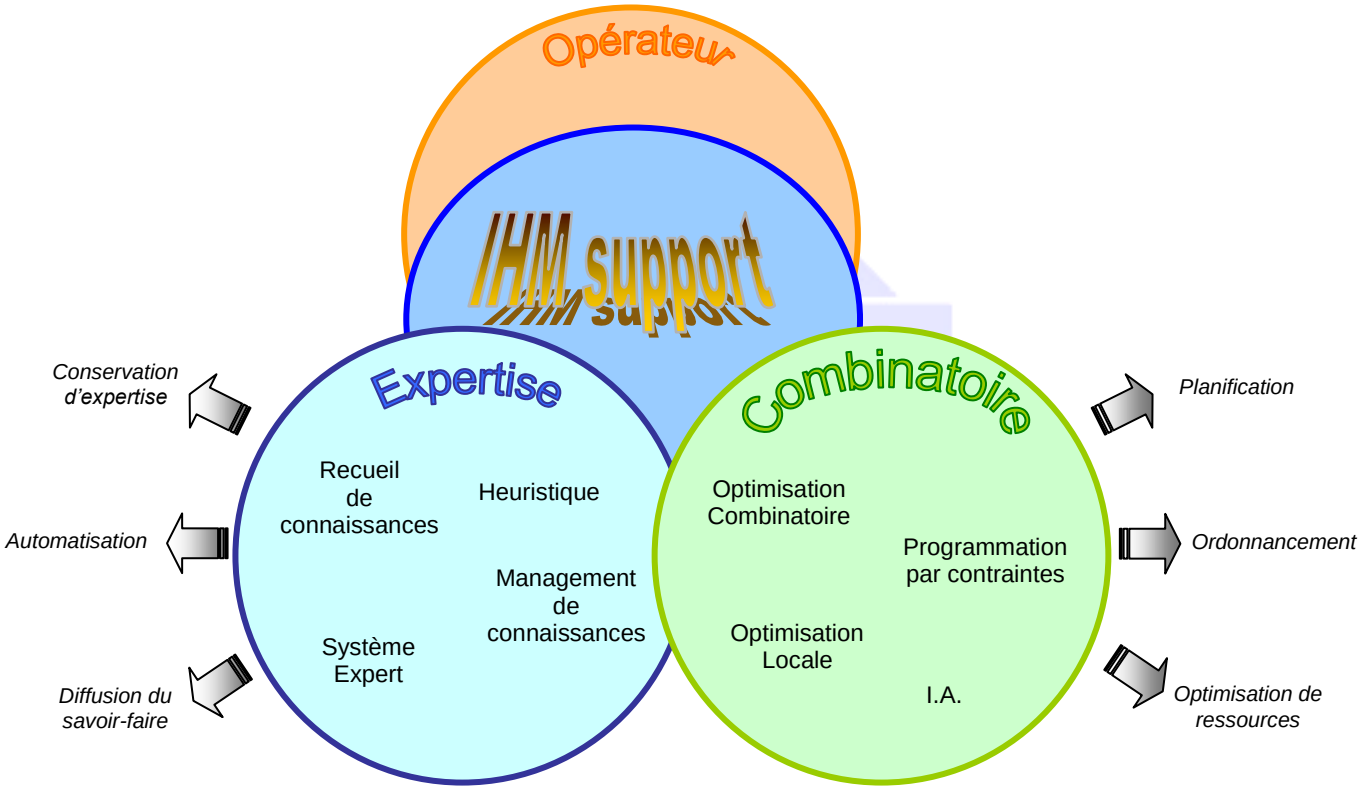
L'aide à la décision

Capitaliser et valoriser l'intelligence de l'entreprise …

Les systèmes d'aide à la décision ont pour objectif de mettre en valeur un savoir-faire pointu de ses clients. Il s'agit d'une véritable coopération entre l'utilisateur qui conserve la décision finale, une IHM support intuitive et des processus complexes producteurs de forte valeur ajoutée.

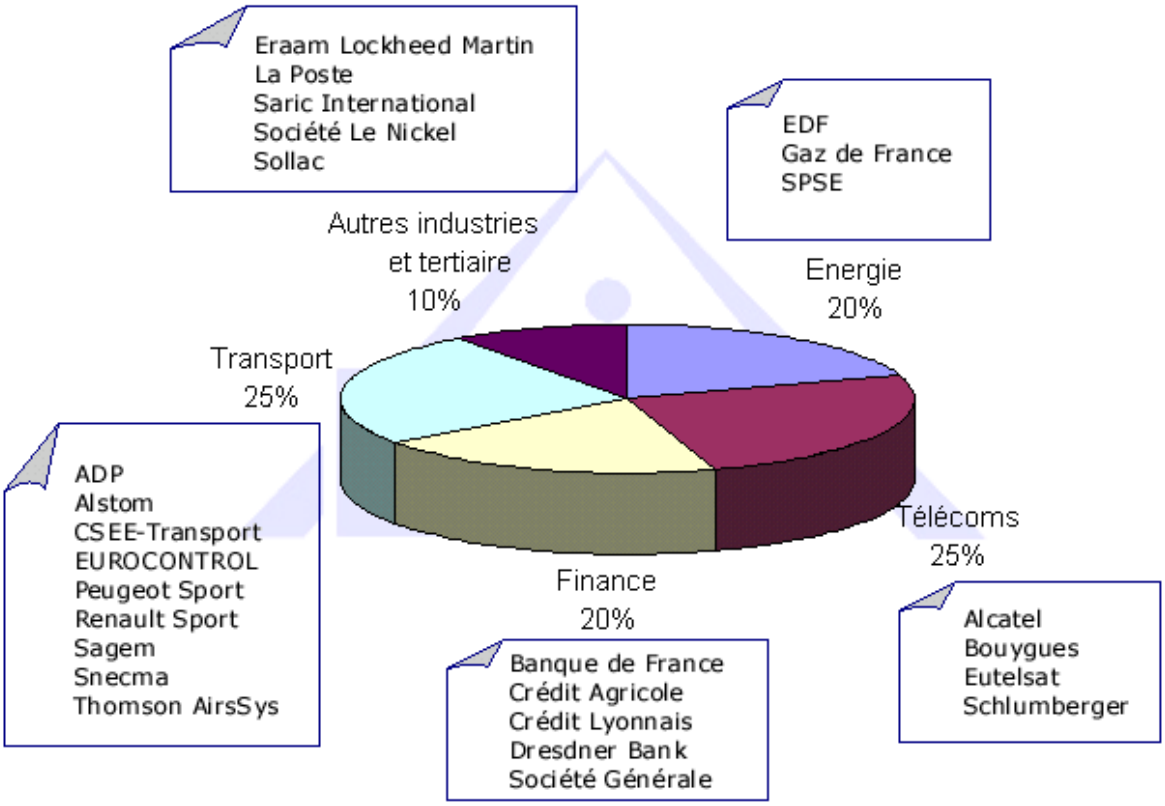
Un savoir-faire né de l'accumulation d'expérience rare ou pointue fera penser à un système expert à base de règles à des fins de conservation, diffusion ou automatisation des connaissances. Cette approche sert aussi pour implanter de manière déclarative des règles de gestion propres à un métier donné.

Un savoir-faire s'appuyant sur un raisonnement fortement combinatoire fera appel aux techniques issues de la Recherche Opérationnelle, la programmation par Contraintes ou de l'Intelligence Artificielle.



Des réalisations concrètes

Partenaires et clients



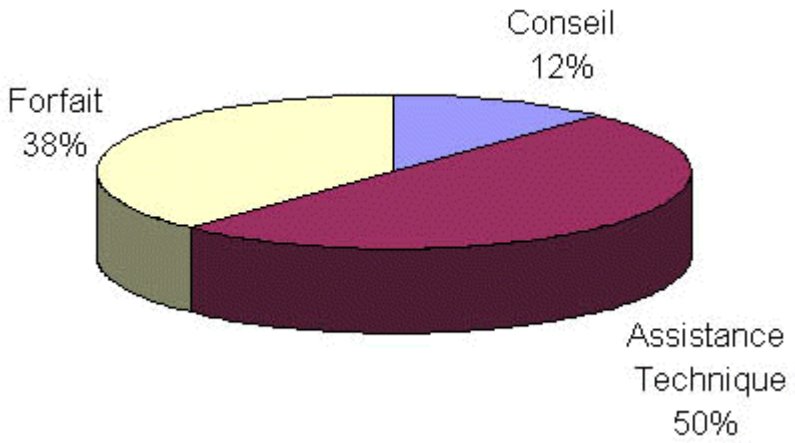
PACTE NOVATION intervient dans de nombreux domaines aussi divers que l'industrie métallurgique (SOLLAC, SPSE, Société Le Nickel), l'énergie (EDF, GDF), le transport (Renault Sport, GEC Alstom, ADP, Eurocontrol, ERAAM, Transnucléaire), l'industrie de l'informatique et des télécommunications (Alcatel, Bull) ou encore la finance des salles de marché (Crédit Agricole Indosuez, Crédit Lyonnais, Dresdner Bank, Société Générale).

La société s'est associée à un partenaire de choix : ILOG. ILOG, société française cotée au NASDAQ, est le premier éditeur français dans le domaine des composants logiciels orientés objets. Le partenariat est fort puisque PACTE NOVATION se voit confier l'évaluation de certains de leurs produits et la sous-traitance d'un certain nombre de leurs formations. Ce partenariat est d'ailleurs mentionné sur le site D'ILOG, à l'adresse : http://www.ilog.fr/html/partners/pacte.htm

Un autre partenariat, dans le domaine de l'ergonomie du logiciel, s'est organisé avec ERGO SOFT France, qui leur permet d'exploiter leurs laboratoires d'utilisabilité.

Enfin, PACTE NOVATION a également réussi à créer des liens scientifiques solides notamment dans le domaine des Sciences cognitives et du langage naturel, avec le C.A.M.S. de Paris.

Types d'activités



PACTE NOVATION exerce trois types d'activités :

- Réalisation de projets au forfait, branche importante de la société puisqu'elle représente 50% de ses activités.
- Conseil, notamment en Ergonomie du Logiciel et en Architectures Orientées Objet.
- Assistance technique, délégation de personnel, dans ses domaines de compétence.

Quelques références

En un peu plus de cinq années d'existence, PACTE NOVATION a participé à des projets ou programmes d'envergure comme :

- Projet ALICE-CARROLL pour Renault Sport F1, un système expert pour la supervision en temps réel des moteurs de Formule 1.
- Un Simulateur de Trafic Ferroviaire pour GEC-Alsthom, vendu dans des affaires à l'export (Hongkong, Athènes, Indonésie...).
- Projet SACHEM pour la SOLLAC, dans le domaine de la supervision des hauts-fourneaux.
- Projet AGATE pour SPSE (Société Pétrolière du Sud Européen), dans le domaine de la planification et l'optimisation d'un pipeline pétrolier.
- Projet MAXIME pour ADP (Aéroports De Paris), qui concerne la planification du personnel de l'escale des aéroports parisiens.

V.

Travail effectué

Les heuristiques: éléments de réponse?

INTRODUCTION

Il faut tout d'abord signaler que la plupart des méthodes que nous allons voir maintenant sont présentées dans la littérature comme des **techniques d'optimisation plus que comme des méthodes d'extraction de la solution**. Cependant il est possible que les avantages qu'elles présentent puissent être exploitables. Y a t il tant de différences entre les techniques qui permettent d'obtenir la meilleure solution à partir d'une bonne, et celles qui fournissent une solution à partir d'un état non solution ? Nous allons tenter d'y répondre.

Afin d'entrevoir d'éventuelles pistes de solution, et en nous appuyant notamment sur [Myna97], nous allons explorer deux grandes familles de méthodes qui permettent de résoudre de manière générale des problèmes d'optimisation en **variables discrètes**:

- les méthodes (d'exploration) globales, en particulier l'énumération implicite heuristiquement ordonnée, que nous illustrerons au travers de **l'algorithme A***,
- et les méthodes (d'exploration) locales, en particulier les méthodes d'exploration par voisinage, pour lesquelles nous étudierons le **recuit simulé** et la **recherche tabou**.

Mais ce chapitre n'aurait d'intérêt sans une orientation dans le sens de la planification : c'est donc dans l'optique de les appliquer à la planification d'actions que nous allons étudier ces différentes techniques, ce qui nous permettra de conclure sur les intérêts éventuels qu'elles sont susceptibles de présenter.

Remarque :

Tout au long de ce chapitre, on utilisera aussi bien les termes d'opérateur, de mouvement ou encore de modification pour parler des actions.

*EXEMPLE D'ENUMERATION IMPLICITE : PARCOURS D'ARBRE AVEC L'ALGORITHME[1] A**

Nous allons ici détailler une première approche qui permet d'obtenir une solution et qui utilise l'**algorithme**[1] **A***. Cet algorithme a été présenté pour la première fois dans [HNR68].

[1] Mots clefs : *A star*, meilleur d'abord, *best first*, stratégie informée ou guidée

Principe général

Il fonctionne sur le principe de la recherche du « **meilleur-d'abord** » (***Best first search***).

On dispose d'une situation initiale `START` bien connue, d'une (ou plusieurs) situation(s) finale(s) déterminée(s) `GOAL`. La préoccupation est alors de trouver comment passer de `START` à `GOAL` avec un coût minimal ou un profit maximum.

On appelle état l'ensemble des informations permettant d'identifier un sous - problème. Pour passer d'un état à un autre, on dispose d'opérateurs apportant un gain additif. Appliquer ces opérateurs à un état s'appelle développer l'état. Réciproquement, on suppose qu'on dispose de marqueurs permettant de retrouver les développements successifs ayant conduit à une solution. Car **c'est davantage le cheminement nécessaire pour identifier une solution que la solution elle même qui compte** dans cette approche : cette dernière étant connue.

Une fonction d'évaluation

En fait l'algorithme parcourt un arbre de recherche et, pour éviter de s'engager vers un noeud par un chemin plus mauvais qu'un précédent, il va stocker des informations sur les nœuds qu'il parcourt. Pour cela, il va évaluer la qualité du nœud auquel il se trouve grâce à une **fonction d'évaluation** f.

Lorsque l'on examine l'état `v`, on connaît le coût de `START` à `v` : c'est la somme des coûts c(`START`, v_1), c(v_1, v_2), …, c(v_k, `GOAL`) des opérateurs nécessaires pour passer de `START` à `v`. Mais, à moins que `v` = `GOAL`, le coût (« futur ») de `v` à `GOAL` est inconnu : on ne peut que l'estimer.

En général, la fonction d'évaluation est donc définie comme la somme de deux autres fonctions notées couramment g et h, telles que pour un nœud donné `v`:

- g donne le coût du chemin déjà parcouru depuis la racine, de `START` à `v`,
- h est une **estimation** du coût du chemin restant à faire jusqu'au nœud final, de `v` à `GOAL`.

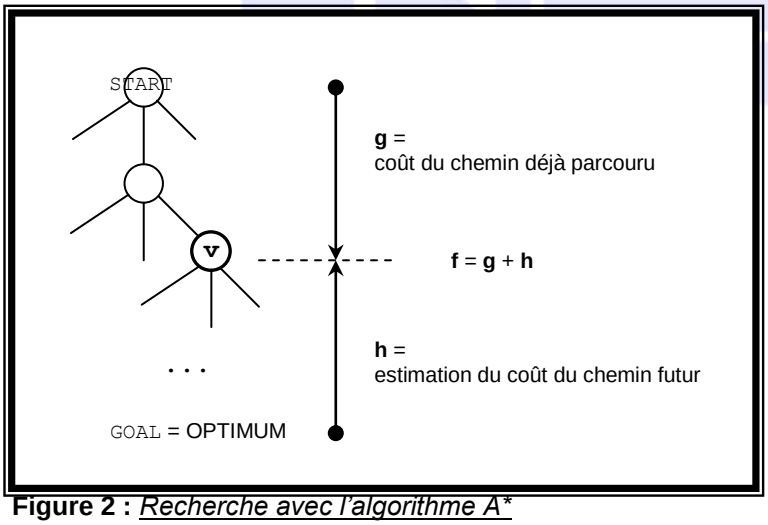


Figure 2 : *Recherche avec l'algorithme A**

Mais on peut aussi pondérer le rapport entre le gain connu g et le gain espéré h par la formule du type f = w.g + (1 - w).h où w∈[0; 1].
L'idéal est d'avoir une estimation exacte de cette distance, ainsi l'effort d'exploration de l'espace de recherche sera minimal. Cependant de telles estimations demandent souvent trop de temps de calcul pour être applicables. Il faut donc trouver un compromis entre le temps de calcul et l'exploration de l'espace de recherche pour rendre l'algorithme pratiquement acceptable [Nils82] [PeKi82].

C'est pour cela qu'on parle de **stratégie informée**. En effet, à tout instant, pour chaque nœud, on doit pouvoir fournir une estimation de la valeur du chemin restant à parcourir.

L'algorithme

Le principe de l'algorithme consiste à chaque étape à choisir un état `u` dans la liste A_VOIR (1°) des états à développer, initialement égale à `START`, et ce jusqu'à ce que `GOAL` ait été identifié, ou que cette liste soit vide (le problème n'admet pas de solution). Cette sélection suit une stratégie « **meilleur d'abord** », dans la mesure où l'élément sélectionné dans A_VOIR est celui dont l'évaluation est la meilleure.
On supprime `u` de A_VOIR pour l'ajouter à la liste DEJA_VU (2°), initialement vide, qui contient l'ensemble de tous les états parcourus pendant la recherche.
Par application d'opérateurs sur l'état sélectionné `u`, on obtient de nouveaux états v_i, les successeurs de `u`, que l'on ajoute à la liste A_VOIR (3°).
Si ces situations v_i ont déjà été envisagées, mais avec un coût moindre, on met les coût à jour, et on considère que l'état doit être réexaminé en tenant compte de cette nouvelle situation.

Cela donne de manière plus formelle :

```
1.  Fonction A_star(START, GOAL : T_Data) : T_Path
2.        A_VOIR ← {START}                          // Liste des éléments à développer
3.        DEJA_VU ← ∅                               // Liste des éléments déjà explorés
4.        u ← ∅
5.        Tant que (A_VOIR ≠ ∅) et (u ≠ GOAL) faire
6.              v ← Choix_dans(A_VOIR)              // En meilleur d'abord
7.              Si (u ≠ G) alors
8.                    A_VOIR ← A_VOIR \ {u}
9.                    DEJA_VU ← DEJA_VU ∪ {u}
10.                   LISTE_FILS ← Developper(u)    // En appliquant les opérateurs
11.                   Pour chaque v dans LISTE_FILS faire
12.                         Si (v ∉ (A_VOIR ∪ DEJA_VU)) alors
13.                               A_VOIR ← A_VOIR ∪ {(v; f(v))}
```

```
14.                              Pred(v) ← u
15.                        Sinon
16.                              w ← Pred(v)
17.                              Si (g(w) + c(w, v) < g(u) + c(u, v)) alors
18.                                    Pred(v) ← u
19.                                    g(v) ← g(u) + c(u, v)
20.                                    f(v) ← g(v) + h(v)
21.                                    Si (v ∈ DEJA_VU) alors
22.                                          DEJA_VU ← DEJA_VU \ {v}
23.                                          A_VOIR ← A_VOIR ∪ {v}
24.                                    Fin_Si
25.                              Fin_Si
26.                        Fin_Si
27.                  Fin_Pour
28.            Fin_Si
29.      Fin_TantQue
30.      Si (u = GOAL) alors
31.            Retourner (START, ..., Pred(Pred(u)), Pred(u), u)
32.      Sinon
33.            Pas de solution
34.      Fin_Si
```

L'**intérêt** de cet algorithme apparaît tout de suite : si un nœud a déjà été marqué précédemment, avec un coût moindre, alors l'algorithme y retourne.
Cela ce fait au détriment d'un **inconvénient** tout aussi flagrant qui pourrait s'avérer limitant avec des domaines étendus: la consommation excessive d'espace mémoire.

Les propriétés de l'algorithme

A* présente les trois propriétés suivantes :

1. **terminaison** :
   si le graphe est fini (nombre fini d'états) et que la fonction d'évaluation f est à valeurs positives ou nulles ($f(v_i) \geq 0$) alors A* s'arrête, soit parce qu'il a trouvé un nœud terminal, soit parce que la liste des nœuds A_VOIR est vide (tous parcourus).
2. **admissible** :
   si f=g+h et $h(v) \geq h^*(v)$ $\forall v$ (pour un problème de maximisation), alors A* est admissible, c'est à dire que l'algorithme trouve toujours une solution optimale du problème.
3. **consistant** :
   pour h si et seulement si quelque soit le successeur v de u, $|h(v) - h(u)| \leq c(u, v)$

La fonction d'estimation

La recherche étant basé sur le coût d'un nœud, la fonction d'évaluation joue un rôle clef. Comme la détermination du coût d'un chemin de la racine au nœud courant n'est pas

difficile, **c'est bien la fonction d'estimation h qui est le point sensible et déterminant de l'algorithme**.

Comme on le constate dans l'algorithme, les « sauts en arrière » peuvent être fréquents. Or, si l'on ne veut pas être obligé de recalculer dans ces cas les valeurs déjà acquises, il faut que **la fonction h soit monotone** : elle n'aurait d'intérêt autrement. Concrètement, à mesure que l'on s'approche du nœud final (optimum), la fonction d'évaluation h doit varier toujours dans le même sens (croître ou décroître), et particulièrement, quand on revient à un même nœud précédemment marqué, cette fonction doit retourner une valeur cohérente avec ce principe : un même état du système en deux endroits différents du l'arbre (deux nœuds ayant la même étiquette) devront avoir la même valeur h(v). Cela permet de gérer la notion d'impasse dans un chemin.

On peut maintenant se demander **comment obtenir ces valeurs de h(v)** pour tous les nœuds v de l'arbre, à savoir les déterminer une seule fois de manière statique au début de l'algorithme, ou les (re)calculer de manière dynamique à chaque fois que l'on arrive à un noeud. Si le nombre d'états, de nœuds, est faible, on peut se permettre de calculer une fois pour toutes la valeur de h pour chacun de ces nœuds, puis de stocker ses valeurs, afin de les utiliser quand nécessaire surtout si le calcul de h n'est pas simple. Mais si le nombre de nœuds est important, voire dénombrable mais infini, cela devient inefficace voire impossible : il faudra alors savoir déterminer pour un nœud quelconque v, à tout moment, la valeur h(v). Cela est d'autant plus efficace, que connaître et donc calculer cette valeur h(v) pour tous les nœuds peut être une perte de temps inutile dans la mesure où il y a très souvent une partie des nœuds qui n'est pas explorée. L'idéal serait de mixer les avantages de ces deux méthodes, à savoir calculer en temps réel la valeur h en un nœud v, que lorsque l'on passe effectivement par ce nœud, puis la stocker pour le cas où l'on viendrait à repasser par un nœud portant la même étiquette.

Une amélioration de A* : Iterative Deepening A*

Iterative Deepening A (IDA*)* [Korf85] est un algorithme d'exploration de graphe dérivé de A*. Son principe est le suivant :

- A chaque itération, on réalise une exploration en profondeur d'abord.
- Dans cette exploration, tous les états dont l'évaluation dépasse un certain seuil sont éliminés de l'exploration.
- Ce seuil est initialement (à la première itération) égal à l'évaluation de l'état initial.
- A chaque nouvelle itération, le seuil est le minimum des évaluations qui dépassaient le seuil dans la dernière itération.
- Chaque état est évalué par une fonction de la forme f = g + h comme dans A*.

- On n'explore à chaque itération que des éléments dont l'évaluation est égale au seuil.

Cela donne de manière plus formelle:

```
1.  Function IDA(START, GOAL : T_Data) : T_Path
2.      SEUIL ← f(START)
3.      NOUVEAU_SEUIL ← -∞
4.      u ← START
5.      Tant que (u ≠ GOAL) ou (f(u) > SEUIL) faire
6.          u ← EnProfondeur(u, START, SEUIL, NOUVEAU_SEUIL)
7.          SEUIL ← NOUVEAU_SEUIL
8.          NOUVEAU_SEUIL ← -∞
9.      Fin_TantQue
10.     Retourner (START, ..., Pred(Pred(u)), Pred(u), u)
11.
12. Function EnProfondeur(u, START, s, ns : T_Data) : T_State
13.     LISTE_FILS ← Developper(u)              // En appliquant les opérateurs
14.     Pour chaque v dans LISTE_FILS tant que (v ≠ GOAL) ou (f(v) < s) faire
15.         Si (f(v) ≤ s) alors
16.             En_Profondeur(v, s, ns)
17.         Sinon
18.             Si (f(v) < ns) alors
19.                 ns ← f(v)
20.             Fin_Si
21.         Fin_Si
22.     Fin_Pour
23.     Si (v = GOAL) et (f(v) ≥ s) alors
24.         Retourner (v)
25.     Sinon
26.         Retourner (START)                   // On recommence depuis START
27.     Fin_Si
```

Cet algorithme **améliore** A* mais possède toujours certains **inconvénients**.

Tout d'abord, cette méthode est intéressante lorsque le graphe d'exploration n'est pas un arbre ou que ou que le nombre d'états à explorer est extrêmement grand. En effet, IDA* est asymptotiquement optimal en terme de temps de résolution et de place mémoire requise en comparaison des autres algorithmes d'énumération. Par contre, lorsque la taille du problème est raisonnable, IDA* est pénalisé par sa répétition de tâches (par exemple faite qu'une seule fois en général dans un algorithme de recherche en profondeur uniquement). Cependant les besoins en mémoire d'IDA* sont minimaux, puisque l'exploration en profondeur d'abord peut se faire de façon récursive avec un minimum de mémorisation.

Limites dans le cadre de la planification

Cette monotonie à propos de la notion d'estimation, garante de l'efficacité de A*, est en fait une limite, dans la mesure où, contrairement à d'autre méthodes comme celles que nous allons voir dans les prochains chapitres, elle empêche d'emprunter un chemin qui serait tout d'abord plus coûteux, mais finalement (sur 2 ou 3 niveaux) bien plus avantageux : **A* exclue les chemins localement défavorables**.
C'est pourtant ce qui permet par exemple à la recherche Tabou et au recuit simulé de ne pas s'enfermer dans un minimum local. Ici, la technique utilisée pour sortir de ce type d'impasse est le retour en arrière (brutal) à des nœuds déjà empruntés par des chemins moins coûteux.

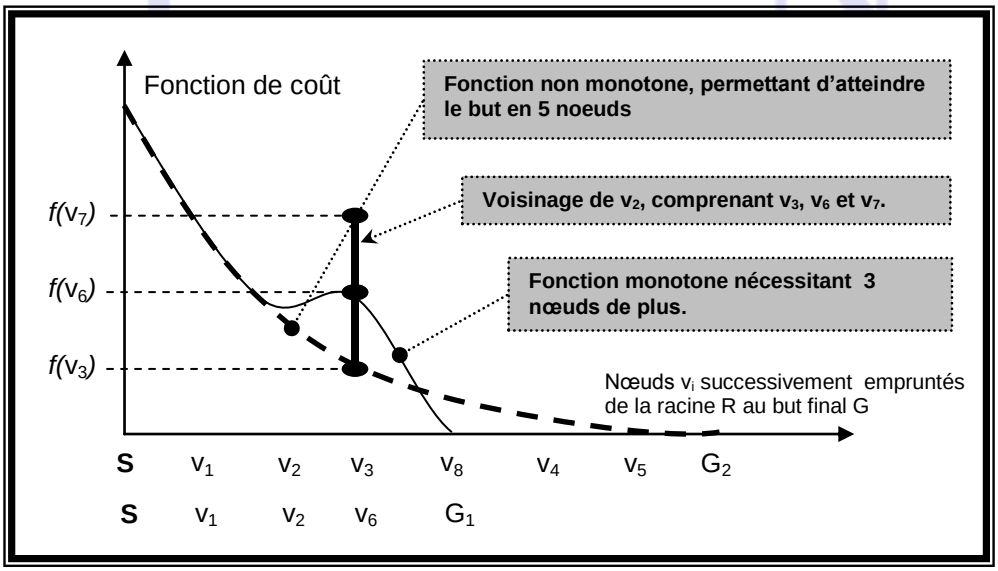


Figure 3 : *Fonctions coût monotone et non monotone*

L'énumération implicite présente l'avantage de fournir des garanties fortes sur les résultats trouvés. La structure de recherche systématique se prête particulièrement bien à l'élaboration de propriétés de la forme théorème/preuve. On a donc des garanties concernant le caractère optimal de la solution trouvée, le nombre d'états explorés, etc.

Cependant, cette sécurité sur le déroulement de l'algorithme et la qualité des résultats a un prix. Ces algorithmes nécessitent un temps de calcul très important, et sont assez peu souples. Dans de nombreux cas, un temps de calcul assez important est nécessaire avant qu'une solution, même de mauvaise qualité, puisse être fournie par l'exploration. En général, l'énumération implicite requiert une place mémoire importante. Il s'agit d'une seconde limitation majeure aux performances de tous ces algorithmes.

Cet algorithme fourni donc un chemin optimum de parcours d'arbre. Dans la figure précédante, avec la fonction monotone, la solution est la séquence de nœuds (S, v_1, v_2, v_3, v_4, v_5, G_2).

…

On obtient donc comme solution, une séquence **totalement ordonnée**, ce qui n'est pas la meilleure des choses, comme nous l'avons précédemment expliqué.

L'EXPLORATION LOCALE

En réponse à cette monotonie, nous allons maintenant explorer les métaheuristiques d'exploration locale.

Les métaheuristiques sont des méthodes caractérisées par l'absence de garantie sur le caractère optimal ou même l'écart à l'optimum des solutions approchées trouvées. Il s'agit d'une logique de recherche d'un élément satisfaisant en parcourant de manière déterminée, orientée, l'ensemble à explorer.

Parmi toutes ces techniques, présentées et étudiées dans [GlGr89], nous allons nous intéresser seulement aux algorithmes d'exploration locale, car ils sont également liés à une représentation du problème sous forme de graphe. Mais ces algorithmes présentent plus de degrés de liberté que ceux de l'énumération implicite.

Principe

Au niveau des composantes, il faut distinguer deux niveaux :

- la détermination du voisinage, le choix d'un élément, et la mémorisation, comme caractéristiques générales présentent sous une forme ou une autre dans les différents algorithmes,
- la diversification, l'intensification et l'oscillation dont l'efficacité est reconnue, mais la présence non obligatoire.

Le principe fondamental de l'exploration locale est, à partir d'une solution totalement instanciée, de passer à un autre élément, du même type que cette solution mais différent, et de recommencer de proche en proche.

Dans un premier temps, nous allons donc approfondir l'expression « élément de même type » ou élément voisin.

Le voisinage

Le **voisinage** défini les éléments proches vers lesquels on peut porter l'exploration, mais il ne contient pas l'élément dont on cherche le voisinage, afin d'éviter une boucle infinie : les constituants de cet ensemble sont appelés *voisin*. Pour un élément donné, il n'existe qu'un seul et unique voisinage.

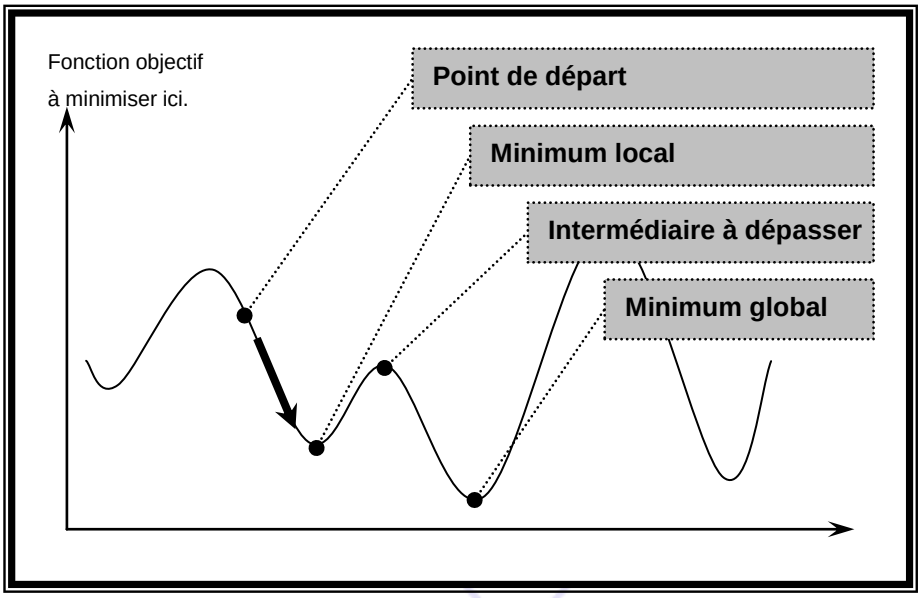


Figure 4 : *Extremum local et global*

La sélection

Une fois que l'on a déterminé le voisinage, il faut savoir vers quel voisin de l'élément courant propager l'exploration. On doit donc **faire un choix** parmi tous ses voisins. Il faut noter que l'élément courant étant quasiment toujours modifié, le voisinage exploré est également différent à chaque itération. Si on prenait de manière systématique le meilleur voisin, à savoir celui optimisant la fonction objectif, on retomberait dans le cas d'algorithmes de descente ou « meilleur-d'abord », avec l'inconvénient cité précédemment : inefficacité face aux extremum locaux. Comme on peut le voir sur la figure, lorsque l'on se trouve en un extremum (minimum sur la figure) local, si l'on accepte pas d'emprunter un chemin intermédiaire non optimum, on reste enfermé dans cet optimum local, avec pour conséquence de ne pas découvrir d'autres optima locaux meilleurs voire l'optimum global.
Cette dégradation, qui consiste à prendre un voisin de « moins bonne qualité », doit cependant être contrôlée, pour éviter de nouveau de boucler en alternant sans arrêt entre l'optimum local, et le moins mauvais des voisins de « moins bonne qualité ».

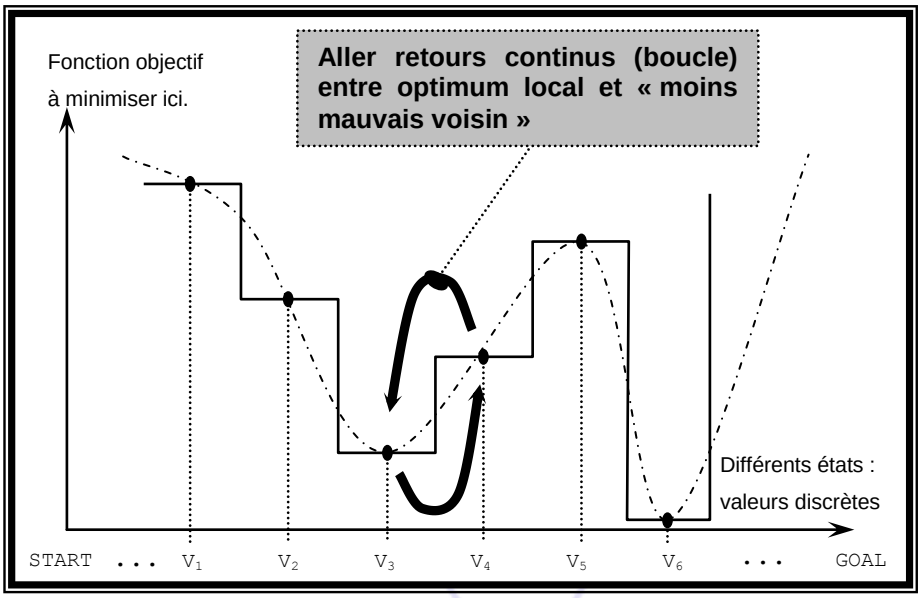


Figure 5 : *Sans mémorisation, risque de « bouclage »*

Pour cela, deux techniques sont utilisées par les métaheuristiques:

- l'introduction du **hasard** à donné naissance aux algorithmes de Monte-Carlo, à GRASP et au **recuit simulé** que nous allons voir dans le chapitre suivant.
- L'introduction d'une **mémoire flexible** est à l'origine de la **recherche tabou**, que nous présenterons dans le chapitre d'après.

La première possibilité consiste donc à choisir un ou plusieurs voisins, **sans examiner** l'ensemble du voisinage. Cette sélection peut se faire soit par une heuristique, soit au hasard.

La seconde possibilité consiste à **explorer l'ensemble du voisinage**, et à sélectionner le (les) meilleur(s) élément(s). Encore une fois, le choix de l'élément peut se baser sur la valeur de la fonction objectif de cet élément, ou bien sur une heuristique.

Le **hasard** peut intervenir à plusieurs niveaux.

Tout d'abord, on peut choisir au hasard le prochain voisin que l'on va considérer. Cependant, l'intérêt du choix aléatoire, est d'éviter d'explorer et donc de générer tous les voisins. Plus que le choix, c'est donc véritablement **la génération** d'un voisin qui devra être aléatoire, sous entendu que le choix portera sur l'individu généré. Mais si les choix successifs étaient purement aléatoires, l'exploration de l'arbre de recherche le serait tout autant (*random walk*), est aurait peut de chance de trouver une solution, du moins de manière optimale : il faudra donc intégrer d'autres mécanismes pour rendre cette technique utilisable.

Mais le hasard peut intervenir à un autre niveau. Nous avons vu précédemment que pour ne pas s'enfermer dans un optimum local, l'algorithme devait être capable de passer par un voisin de « moins bonne qualité ». Mais comment déterminer la valeur de cette « moindre qualité » ?… Justement de manière aléatoire. En effet, on peut s'autoriser à passer par un

voisin dont la valeur de la fonction objectif soit moins bonne que celle du nœud courant, en s'en écartant (au plus ou au moins) d'une valeur petite prise au hasard : c'est sur ce principe que fonctionne le recuit simulé.

La mémorisation

Contrairement aux algorithmes d'énumération implicite comme A*, ceux que nous voyons dans ce chapitre n'utilisent pas de mémorisation systématique.
Le recuit simulé par exemple ne mémorise que l'état courant et le meilleur état trouvé.
Par contre, la recherche Tabou est basée sur une **mémoire flexible**[2] [Glov96], à savoir que l'on ne conserve que quelques informations nécessaires à l'exploration, soit concernant les éléments (cela peut demander beaucoup de ressources mémoires en cas de domaines étendus), soit concernant les mouvements, qui eux sont souvent en nombre déterminé ce qui permet de même de connaître à l'avance l'espace mémoire nécessaire.
Cette quantité d'espace mémoire nécessaire peut donc rapidement exploser. Pour limiter ce besoin, parfois problématique, on se base sur deux critères :

- l'**ancienneté**[3] des éléments : en mémorisant l'instant d'apparition de l'élément on peut ensuite ne conserver que les n plus récents,
- leur **fréquence**[4] : on mémorise ici le nombre de fois qu'intervient un élément ou un mouvement.

La diversification

Elle consiste à générer un nouvel élément, différent de ceux déjà explorés, dans le but de partir dans une nouvelle direction, pour explorer une autre région. Dans les algorithmes génétiques cette diversification porte le nom peut être plus explicite de mutation.
Partant de là, deux questions se posent :

- **quand** lancer la diversification,
- et dans **quelle direction** partir : à quoi doit ressembler le nouvel élément ?

Selon le but recherché et le problème considéré, plusieurs méthodes, permettant de déterminer à quel instant déclencher la diversification, s'offrent à nous.
La diversification peut tout d'abord être déclenchée de manière **automatique**. Le plus simple, consiste à déterminer un nombre de cycles, d'itérations, au bout duquel on déclenche la diversification.

[2] Flexible Memory
[3] recency
[4] frequency

De manière plus dynamique, pour ne pas dire « plus intelligente », on peut mettre en place une diversification basée sur le **gel de l'exploration**. En effet, dés que l'on détecte que les éléments générés demeurent (plus ou moins) constants, on peut lancer la diversification. Le moyen le plus simple là encore, est de se donner un nombre d'itérations de l'exploration, durant lequel cette fois la meilleure solution ne change quasiment plus, voire plus du tout.

Une dernière méthode envisageable, fondée sur la **mémorisation**, serait d'équilibrer l'apparition de diversités. En se basant sur les fréquences mémorisées, on peut percevoir l'apparition de nouveaux éléments, et lancer la diversification lorsque le déséquilibre devient trop grand, afin de garantir une certaine homogénéité : on cherche à explorer l'ensemble des éléments pas encore examinés de manière aussi uniforme que possible. Pour cela, dés qu'un ou plusieurs attributs dépassent une fréquence d'apparition (fixée ?), on détermine un nouvel élément courant excluant ces attributs très utilisés et privilégiant les attributs moins rencontrés jusqu'à présent. Cette dernière méthode apparaît comme un moyen de déterminer aussi bien l'instant où lancer la diversification que la direction dans laquelle s'engager.

Une fois que l'on a ainsi établi que la diversification devenait nécessaire, du moins utile, il reste encore à définir la forme du nouvel élément à générer. En d'autre termes, il faut déterminer la zone à explorée, délaissée jusqu'à là, qui permettrait la diversification de l'exploration. Expliciter cette notion de « zone », nous permettra de mettre en évidence les différentes méthodes envisageables. Une « zone » peut (doit ?) être vue comme un sous-ensemble de celui des éléments, non encore explorés et ayant des caractéristiques communes (mêmes valeurs pour des variables données…).

Si la **méthode de détermination de l'élément initial** n'est pas déterministe, à savoir si elle ne donne pas toujours le même père pour une instance donnée, on peut l'utiliser pour générer un nouvel élément ayant des caractéristiques proches de l'élément considéré, mais différent de lui.

Par contre, lorsqu'elle est déterministe, on peut la **modifier** en y introduisant une part de hasard dans la génération de l'élément, afin de différencier le nouveau successeur de l'ancien.

Une dernière méthode, serait d'utiliser la **mémorisation des fréquences ou de l'ancienneté**, comme déjà vu. Ainsi, au lieu de diverger vers une zone choisie de façon « hasardeuse » parmi celles délaissées, comme dans les deux méthodes précédentes, on considère la zone la plus délaissée. En classant les attributs ou les variables du problème selon les fréquences croissantes, ou les anciennetés décroissantes, on pourra ensuite utiliser une méthode gloutonne basée sur ce classement, c'est à dire construire un élément en privilégiant les attributs ou variables les mieux classés, donc les moins souvent rencontrés.

L'intensification

L'idée générale de **l'intensification**, est pour ainsi dire complémentaire de la diversification, dans la mesure où elle vise à explorer préférentiellement une zone déjà examinée, et qui apparaîtrait comme prometteuse. La meilleure façon d'éviter une voie défavorable, n'est elle pas de s'engager vers une estimée comme prometteuse ? Cependant, cette technique est plus compliquée à mettre en pratique que la précédante. En effet, pour la diversification, il suffit de constater que la « qualité » du résultat est actuellement en deçà d'une valeur seuil pour s'engager vers une autre voie, alors que pour intensifier une zone, il faut être capable de prévoir si elle va être prochainement prometteuse.

Comme précédemment, plusieurs techniques permettent de déterminer **quand** lancer l'intensification.
On peut le faire de manière **automatique**, à savoir par exemple toutes les k itérations. Il s'agirait, au bout d'un nombre d'itérations donné, d'intensifier une zone à déterminer. Cette solution étant présente dans la littérature, je la reporte ici, mais je ne vois pas l'intérêt de considérer l'intensification autrement que de manière dynamique (et non automatique) à savoir par rapport à la qualité de la solution actuellement envisagée. Je pense en effet que si l'on se trouve sur une voie apparemment intéressante, ce serait pure perte de temps, et donc d'efficacité, que d'attendre que ce fameux nombre d'itérations soit atteint avant de décider d'intensifier.
Une autre (la !) solution est donc de se baser sur **l'amélioration**. L'intensification est dans ce cas déclenchée soit lorsqu'une amélioration de la meilleure solution est détectée, soit lorsque l'élément considéré prend la forme d'un schéma initialement défini comme prometteur.

Une fois que l'on a déterminé que la recherche devait être intensifiée, en plus de savoir **quelle zone intensifier** comme dans la diversification, il faut ici **établir la durée** de l'intensification, à savoir pendant combien d'itérations il faudra rester dans cette voie qui peut finalement s'avérer être non optimum.
Pour déterminer la zone à intensifier, la méthode la plus simple est la **mémorisation**. En effet, en comparant entre elles les n meilleures, ou les n dernières, solutions trouvées précédemment, et enregistrées, on peut alors les comparer entre elles pour établir l'ensemble des attributs qu'elles ont en commun et qui définiront la zone à intensifier.
Une autre possibilité est de partir simplement d'une ancienne solution enregistrée, et d'imposer certains mouvements et/ou d'en interdire d'autres, pendant un certain temps, de façon à favoriser l'exploration intensive de la zone ainsi déterminée par contraintes autour de l'ancienne solution.

Reste encore à déterminer le nombre d'itérations pendant lequel on s'autorise à rester dans cette zone soit disant prometteuse avant d'atteindre éventuellement une solution.
Cette étape est particulièrement importante et délicate. En effet, il faut ici établir le nombre d'itérations pendant lequel on va s'investir dans l'intensification. Rappelons que le caractère « prometteur » de la voie empruntées n'est forcement qu'une estimation. Par conséquent, avant d'arriver à la solution, il va falloir réaliser un certain nombre d'itérations. Le problème tient au fait que cette voie empruntée peut finalement s'avérer être une impasse infructueuse, et donc une perte de temps d'autant plus grande que le nombre d'itérations était important. Ainsi il faudra donc faire un compromis entre le degré de certitude avec lequel l'algorithme est capable de déterminer si une solution est prometteuse, et le temps à lui consacrer : si l'algorithme est capable de déterminer avec précision que la voie est prometteuse, on peut s'autoriser un plus grand nombre d'itérations avant d'obtenir la solution, puisqu'on est quasiment sûr d'obtenir une (bonne) solution. La détermination de la durée de l'intensification peut être faite :

- soit de façon préalable et statique : mais déterminer de manière statique quand arrêter l'intensification apparaît tout aussi inefficace que de déterminer de la même façon quand la commencer (Cf. un peu plus haut…),
- soit en considérant le nombre d'itérations ayant été nécessaire pour établir les précédentes solutions,
- soit en tenant compte du degré d'instanciation de la zone à intensifier et du nombre de mouvements imposés,
- soit en étant attentif aux variations de la qualité de la solution considérée,
- soit enfin en mixant tout cela…

En fait, il apparaît que les critères et techniques de détection de la date où commencer, et de celle où terminer l'intensification, sont similaires.

L'oscillation

L'oscillation consiste à s'approcher d'une frontière, à la dépasser, puis à en s'en éloigner, avant de faire demi tour et de recommencer.
L'intérêt de cette technique apparaît lorsque la caractéristique (courbe) de la fonction objectif est fortement indentée. On peut ainsi sauter de proche en proche, d'extrema locaux en extrema, en s'affranchissant des nombreux « mauvais » voisins.
Pour cela :

- soit on modifie la fonction objectif (on « leurre » l'algorithme),
- soit on force la direction d'exploration dans le voisinage, obligeant ainsi à poursuivre la recherche malgré les « mauvais » voisins.

Cette technique dégage **plusieurs avantages**, notamment présentés dans [KGA93] et [Glov89].

- Si le graphe associé aux éléments réalisable n'est pas connexe, à savoir s'il possède au moins deux éléments non reliés entre eux par un chemin, alors l'oscillation permet de **traverser les zones non réalisables** au cours de la recherche. De plus, même si le graphe est connexe, mais que les chemins soient long et tortueux, l'oscillation permet là encore de « prendre comme raccourci » un « mauvais » voisin.
- Pour certains problèmes, le simple fait de trouver une solution est déjà une problématique NP-complet : dans ces conditions ne parlons donc pas de trouver une solution optimum. L'oscillation peut permettre dans ces cas une **définition et une exploitation plus simple**, voire utilisable, du voisinage.
- Enfin, pour couvrir une zone la plus large possible, et favoriser ainsi au maximum la découverte de solutions éventuelles, et donc être d'avantage efficace, les techniques d'exploration locale doivent assurer **une diversité suffisante** en ce qui concerne les éléments observés. L'oscillation fourni cette diversité révélant ainsi des possibilités d'amélioration inaccessibles lorsque l'exploration est restreinte à des zones plus réduites.

EXEMPLE D'EXPLORATION LOCALE : LE RECUIT SIMULE

Principe

Les origines de la méthode du Recuit Simulé[5] remontent aux expériences de *Metropolis et al.* [MRRT53]. Leurs travaux ont abouti à un algorithme simple pour simuler l'évolution d'un système physique instable vers un état d'équilibre thermique à une température T fixée. Un état de ce système est caractérisé par la position exacte de l'ensemble des atomes qui le constituent. Tout nouvel état est obtenu en faisant subir un déplacement infinitésimal et aléatoire à un atome quelconque. Soit ΔE la différence d'énergie occasionnée par une telle perturbation. Le nouvel état est accepté si l'énergie du système diminue ($\Delta E < 0$). Dans le cas contraire, il est accepté avec une certaine probabilité.

De nombreuses années se sont écoulées depuis les travaux de *Metropolis et al.* avant que leur algorithme soit exploité en 1983 par *Kirkpatrick* [KGV83] en vue de définir une nouvelle heuristique pour l'optimisation combinatoire. En faisant l'analogie entre l'énergie du système physique et la fonction objectif du problème d'une part, et entre les états successifs du système et les solutions admissibles d'autre part, en considérant de plus la température d'un système physique ne terme d'agitation thermique, et en s'inspirant de la méthode de

[5] *Mots clefs :* Simulated annealing, algorithme de Monte-Carlo, méthode d'exploration locale.

Monte-Carlo, le recuit simulé consiste donc à chaque itération à **choisir aléatoirement un voisin** dans le voisinage de l'élément courant, et à le considérer comme le nouvel élément courant:

- avec **certitude** s'il est de meilleur qualité,
- avec une certaine **probabilité**, sinon.

Si ***f*** est la fonction d'évaluation ou objectif d'un élément (celle qui détermine sa qualité), ***v*** le voisin d'un élément donné ***u***, et ***T*** la température courante, alors la probabilité associée à ce voisin pour la température *T* est définie comme suit :

$$p(v,T) = \exp\left(\frac{-|f(v) - f(u)|}{T}\right)$$

Dans le cas le voisin choisi est de moindre qualité, **l'acceptation** de ce dernier se fait en générant aléatoirement un nombre compris entre 0 et 1 exclu. Si ce nombre est inférieur ou égale à la probabilité affectée au voisin alors ce dernier est accepté, dans le cas contraire, on maintient l'élément courant.

Toute meilleure solution trouvée est mémorisée, et l'algorithme s'arrête lorsque plus solution n'est détectée parmi les voisins, pendant un pallier de température (cycle complet d'itérations à une température donnée, ou que la température est nulle).

Pour des présentations plus complètes et détaillés du recuit simulé, on pourra se reporter à [KGV83], [JAMS89], [Egle90] et [Dows93].

Température et fonction de refroidissement

Afin que ces détériorations temporaires, mais successives, de la qualité soient contrôlées, le recuit simulé intègre un paramètre supplémentaire, appelé ***température***, qui correspond à un écart maximum à partir de la solution courante, en dehors duquel les valeurs de la fonction objectif ne doivent pas s'aventurer. Les variations de cette température sont imposées de manière précise par une fonction décroissante appelée ***fonction de refroidissement***. En règle générale, la température est diminuée par pallier, à chaque fois que l'on a effectué un nombre donné d'itérations. La nouvelle température est obtenue en multipliant l'ancienne par un coefficient *a* plus petit que 1 : $a \in]0, 1[$.

Se pose alors le problème de **déterminer une valeur pertinente pour *a***. On peut déjà noter que *a* ne doit pas être trop loin de 1. En effet, d'après des travaux théoriques, l'algorithme converge si la température « lui en laisse le temps ». De manière pratique, entre chaque variation de température, il faut que l'algorithme ait l'« autorisation » (intervalle de recherche, ou température, assez grand) et le temps (suffisamment d'itérations pour chaque pallier) de balayer suffisamment de situations. Déterminer une valeur pertinente pour ce

paramètre ne peut donc être fait de manière absolue, mais au contraire en corrélation avec le nombre d'itérations autorisées pendant chaque pallier (paramètre `NB_MAX` à la ligne 14 de l'algorithme).
Lorsque la température est abaissée suffisamment lentement pour que l'équilibre soit maintenu, le processus se traduit par une augmentation du poids des configurations à basse énergie.

Ainsi, à mesure que l'on progresse dans la recherche, la température diminue, et il en est de même pour l'importance des détériorations acceptées : l'exploration évolue d'une stratégie quasiment de type aléatoire au départ, vers une stratégie de type « meilleur d'abord ». Le nouvel élément est toujours pris au hasard dans un voisinage. Cependant, au départ la forte température implique que la dimension de ce voisinage est grande et que l'on choisi au hasard parmi des voisins divers et variés (tous les changements sont acceptés : la probabilité d'acceptation, étant de la forme *exp(-k/T)*, tend vers 1 quand T est grand), alors que la faible température finale, synonyme de voisinage restreint, sous entend que les voisins disponibles sont proches de la solution actuelle (*exp(-k/T)*, et donc la probabilité d'acceptation d'une solution plus mauvaise, tend vers 0 quand T est petit).

Algorithme

Voici l'algorithme du la méthode du Recuit Simulé.

```
1.  Fonction RS(u0 : T_Element) : T_Element                  // u0: élément initial
2.      u ← u0
3.      V(u0) ← Determiner_voisinage_de(u0)
4.      T ← T0                                               // T0: température initiale
5.      CONDITION_D_ARRET ← faux
6.      NB_ITERATION ← 0
7.      Tant que (CONDITION_D_ARRET = faux) faire
8.          CONDITION_D_ARRET ← vrai
9.          Tant que (NB_ITERATION < NB_MAX) faire           // NB_MAX itérations par T
10.             NB_ITERATION ← NB_ITERATION + 1
11.             v ← Choix_aleatoire_dans(V(u))
12.             u_old ← u
13.             Si (f(u) ≥ f(v)) alors                       // Si le voisin est meilleur
14.                 u ← v
15.                 CONDITION_D_ARRET ← faux
16.             Sinon                                        // Sinon calcule probabilité
17.                 p = Tirage_aleatoire_dans([0; 1])
18.                 Si (p < exp(-|f(v)-f(u)|/T)) alors
19.                     u ← v
20.                     CONDITION_D_ARRET ← faux
21.                 Fin_Si
22.             Fin_Si
```

```
23.                  Si (u ≠ u_old) alors                    // cad si u a été modifié
24.                        V(u) ← Determiner_voisinage_de(u)
25.                  Fin_Si
26.            Fin_TantQue
27.            T ← Fonction_de_refroidissement(T)          // souvent: T ← a.T / 0<a<1
28.            CONDITION_D_ARRET ← CONDITION_D_ARRET ou (T = 0)
29.                                                        // Si T=0 on arrête aussi
30.      Fin_TantQue
31.      Retourner u
```

Tant que la condition d'arrêt n'est pas vérifiée (ligne 7), pour chaque pallier de température on fait `NB_MAX` itérations (ligne 9) où :

- on choisit au hasard un élément dans le voisinage `V(u)` de l 'élément courant noté `u` (ligne 11),
- si le voisin est de meilleure qualité (ligne 13) on le prend comme nouvel élément courant (ligne 14),
- sinon, on l'accepte selon les conditions des lignes 16 à 22 de l'algorithme.
- si l'élément courant a été modifié (ligne 23), on calcule alors son voisinage (ligne 24), puis on passe à l'itération suivante.

Une fois le `NB_MAX` d'itérations effectuées, on diminue la température (ligne 27).
Si elle est alors nulle (pose notamment problème à la ligne 18 en divisant par zéro), ou que l'élément courant n'a pas été modifié durant le dernier pallier de température (lignes 28 et aussi 15 et 20), on arrête.

Il est à noter, que l'on peut trouver en lieu et place de « élément `u` » la notion de « solution `s` ». Plus qu'un simple changement de lettre, cela implique que l'on connaît au départ déjà une solution au problème, et qu'au lieu de chercher à **extraire une solution** on souhaite **optimiser une solution** déjà connue mais de « mauvaise » qualité au départ.

Avantages et inconvénients

Tout d'abord, l'utilisation d'une température, qui plus est variable, permet de prendre en compte des **dégradations contrôlées des éléments**, et ainsi de ne pas s'enfermer dans des optima locaux. Du points de vue théorique, le recuit simulé converge vers une solution optimale (il suffit de prendre une décroissance infiniment lente de la température[6]), et en pratique, présente des performance accrues par rapport aux méthodes de recherche du « meilleur d'abord » (algorithme A*) ou basée sur l'exploration purement aléatoire (Monte-Carlo).

[6] Etant donné que la fonction de refroidissement est le plus souvent de type linéaire, une décroissance **infiniment lente** consiste à prendre un coéfficient directeur proche de 1 : ensuite pour choisir entre 0,9 ou 0,999, ça dépend du problème.

Cependant, comme d'habitude, cette efficacité à un prix. En plus de la définition des voisinages, de la tolérance aux dégradations (la fonction donnée dans l'algorithme est la plus répandue, mais on en trouvera d'autres dans [Dows93] p. 43) et de la condition d'arrêt, **il est nécessaire de définir de nombreux autres paramètres**, définitions rendues délicates par les interactions existantes. Il s'agit plus précisément :

- des températures initiale et finale ainsi que de la fonction de refroidissement (Cf. [Dows93] p.44),
- de la longueur de chaque chaîne pour une température donnée (nombre d'itérations à cette même température).

En fait, la définition de ces paramètres correspond aux étapes de diversification, d'intensification et d'oscillation, que nous avons étudiées dans le chapitre précédant.

Elle dépend du type de problème traité, du type de résultat que l'on souhaite obtenir, de la forme des instances envisagées, et traduit le compromis entre la qualité souhaitée pour la solution finale, et la rapidité d'obtention de cette solution qui sera le plus souvent contraignante et imposée.

En effet, contrairement à d'autres heuristiques, **il est impossible d'arrêter l'algorithme du recuit simulé avant la fin**, car le refroidissement perd alors toute signification. Le fait de tolérer la sélection de « mauvais » éléments, qui plus est avec une grande part de hasard certes décroissante, dans le but d'atteindre plus rapidement la solution éventuelle, entraîne que l'algorithme passe par des phases « instables » susceptibles d'être la solution courante si l'on arrête l'exploration avant la fin. Cela confère une importance toute particulière au critère d'arrêt du recuit simulé, contrairement à d'autres techniques, comme les algorithmes génétiques par exemple, où l'évolution de la solution est monotone dans le sens de l'optimum : on peut donc extraire à tout moment une solution significative, même si sa qualité augmente avec la durée de l'exploration.

Mais cet inconvénient n'en est pas forcément un pour nous, selon ce que nous allons considérer comme élément du voisinage. En effet :

- soit cet élément correspond à une séquence d'actions, initialement vide, que l'on construit et complète à mesure de l'exploration,
- soit c'est une séquence qui dés le départ contient n actions, pouvant avoir aucune signification (extraction de solution) ou au contraire être consistante (optimisation de solution), et qui par ajout et/ou modification d'actions va évoluer vers une solution meilleure, voire optimale.

Dans le premier cas, il apparaît évident qu'arreter l'algorithme avant la fin n'a pas d'intérêt. Le problème est donc de savoir si l'on fait simplement de l'extraction ou si l'on part d'une solution existante dans le but de l'optimiser …

Dans la planification, les temps de traitement actuels s'envolent dés que le nombre d'actions possibles dépasse le millier. Dans ces conditions, il apparaît primordial de savoir tout d'abord extraire une séquence solution dans des délais acceptables, à défaut d'être

optimums. A partir de là, étant donné le coût d'une optimisation, on en droit de se demander si elle serait intéressante compte tenu de l'amélioration apportée. Par exemple, si l'optimisation d'une séquence solution initiale de 1500 actions, produit une séquence meilleure (et même optimale) de 1300 actions, pour un temps équivalent à celui de l'extraction de la solution initiale, on peut éventuellement remettre en question son intérêt, et en tout cas certainement le relier au type du problème à planifier.

AUTRE EXEMPLE : LA METHODE TABOU

Principe

Développée par *Glover*, [Glov89] et [Glov89], et indépendamment par *Hansen* sous le nom anglais de « *steepest ascent, mildest descent* », la **Recherche Tabou**[7] est une méthode séquentielle.
A l'inverse de celles présentées précédemment qui ne génèrent qu'un seul voisin, la méthode **Tabou examine un sous-ensemble de solutions** pour ne retenir que la meilleure. Dans certains cas où le voisinage n'est pas trop étendu, il est possible de l'examiner entièrement pour sélectionner effectivement LA meilleure solution, au lieu d'en choisir une aléatoirement parmi les meilleures comme cela se fait en présence de voisinages trop vastes pour être intégralement explorés.

La recherche tabou génère donc de manière générale plusieurs solutions à chaque étape, ce qui risque de la faire boucler dés qu'elle s'éloigne d'un extremum local : la sélection du meilleur voisin provoquant au tour suivant le retour à l'extremum local que l'on va quitter. Pour prévenir cela, elle reprend la notion de diversification vue précédemment, à savoir **l'interdiction de certains mouvements** qualifiés alors de **tabous** afin de ne pas revenir à une solution déjà rencontrée récemment. Le caractère *tabou* d'un mouvement doit être temporaire afin d'accroître la flexibilité de l'algorithme par remise en question des choix effectués une fois les risques de cycle écartés.
Ainsi, à chaque itération, **on mémorise** dans une liste, qu'on appellera par la suite `TABOU`, l'élément `u` (ou solution `s`) courant(e). Désormais, les voisins autorisés d'un élément `u` sont tous ceux présents dans son voisinage privé de ceux obtenus à partir des mouvements contenus dans `TABOU`.

La gestion de ces listes, qui permettent à l'algorithme de mener une exploration aussi large que possible tout en réduisant les risques de bouclage, joue donc un rôle essentiel

[7] Mots clefs : Tabu Search, steepest escent/midest descent.

dans l'algorithme et **la longueur de chacune d'entre elles** doit être définie rigoureusement en fonction du problème considéré.

On pourra obtenir des présentations très complètes de la recherche tabou dans [Glov89], [Glov90], [GTD93] et [GlLa93].

Algorithme

Voici une première version simplifiée de l'algorithme de la Méthode Tabou.

```
1.  Fonction RT(u0 : T_Element) : T_Element
2.      u ← u0                                  // u0: element initial
3.      u* ← u0                                 // u*: meilleur element trouvé
4.      TABOU ← Ø                               // liste(s) TABOU vide(s)
5.      CONDITION_D_ARRET ← false
6.      NB_ITERATIONS ← 0
7.      Tant que (CONDITION_D_ARRET = false) et (V(u; TABOU) ≠ Ø) faire
8.          NB_ITERATIONS ← NB_ITERATIONS + 1
9.          v ← Meilleur_element_de(V(u; TABOU))
10.         Si (f(v) ≥ f(u*)) alors
11.             u* ← v
12.         Fin_Si
13.         Mise_a_jour_de(TABOU)               // voir paragraphe suivant
14.         u ← v
15.     Fin_TantQue
16.     Retourner u*
```

Une amélioration consiste à prendre en compte non pas une mais plusieurs solutions possibles, à intégrer dans les listes tabou des attributs de mouvements, et à utiliser la fonction d'aspiration.

```
17. Fonction RT2(u0 : T_Element) : T_Element
18.     u ← u0                                  // u0: élément initial
19.     u* ← u0                                 // u*: meilleur élément trouvé
20.     TABOUi ← Ø                              // listeS TABOU videS
21.     CONDITION_D_ARRET ← false
22.     NB_ITERATIONS ← 0                       // compteur d'itérations
23.     MEIL_ITERATION ← 0                      // itération de u*
24.     Tant que (CONDITION_D_ARRET = false) faire
25.         NB_ITERATIONS ← NB_ITERATIONS + 1
26.         VOISINAGE ← Determiner_les_solutions(TABOUi, FONC_ASPIRATION)
27.                     // on génère un ensemble de solutions u'=u ⊕ a
28.                     // satisfaisant au moins une des 2 conditions :
29.                     //   * au moins un attribut de a n'est pas tabou
30.                     //   * f(u*) < A(f(u))
31.         u* ← Meilleur_element_de(VOISINAGE)
```

```
32.          Mise_a_jour_de(TABOU_i)              // voir paragraphe suivant
33.          Mise_a_jour_de(FONC_ASPIRATION)      // voir paragraphe d'après
34.          Si (f(u) < f(u*)) alors
35.               u* ← u
36.               MEIL_ITERATION ← NB_ITERATIONS
37.          Fin_Si
38.                    // On s'arrête :
39.          CONDITION_D_ARRET ← (NB_ITERATIONS - MEIL_ITERATION > NB_MAX)
40.                    //     si ca fait au moins NB_MAX itérations
41.                    //     que l'on a pas modifié la meilleure solution
42.          CONDITION_D_ARRET ← CONDITION_D_ARRET ou (NB_ITERATIONS > NB_TOT)
43.                    //  OU si l'on a fait au total + de NB_TOT itérations
44.     Fin_TantQue
45.     Retourner u*
```

Comme pour l'algorithme du recuit simulé, on peut trouver en lieu et place de « élément u » la notion de « solution s », avec les mêmes implications.

Définition et mise à jour de la liste TABOU

Cette mise à jour de TABOU, directement déterminée par sa structure et effectuée à la ligne 13 et 32 de l'algorithme, peut être de complexité variable.

La façon la plus simple de faire, est de la considérer de manière circulaire, de type *First In, First Out (FIFO*). On ajoute en fin de liste l'élément courant, et on retire le premier élément de la liste qui correspond en fait au plus ancien. Cette gestion permet d'éliminer assurément tous les cycles de longueur inférieur ou égale à ||TABOU|| (nombre d'éléments dans TABOU). Ainsi **tous les éléments tabous ont la même longévité** et demeurent interdit pendant le même nombre d'itérations, à savoir ||TABOU||.

Une autre approche serait d'attribuer cette fois à **chaque mouvement tabou un nombre d'itérations qui lui soit propre**. On pourrait donc former TABOU de couples constitués :

- du mouvement que l'on vient de considérer (entre un élément u et son voisin v),
- et d'une valeur n, propre à chaque mouvement, et qui correspond au nombre d'itérations pendant les quelles ce mouvement restera interdit.
- Cette variante n'a réellement d'intérêt par rapport à la précédente que si les n sont différents pour chaque mouvement. La mise à jour de TABOU consiste alors à décrémenter pour chaque mouvement présent dans la liste, le nombre d'itérations associé, et à supprimer les couples dont ce nombre est nul. Il faut maintenant se demander si la souplesse assurée ici est bien intéressante au vu du traitement supplémentaire (parcours de toute la liste…) qu'elle occasionne.

On peut également envisager des gestions de listes `TABOU` plus complexe où les durées d'interdiction, c'est à dire le nombre d'itérations pendant lesquelles un mouvement restera interdit, peuvent être variées comme précédemment, mais aussi variables comme dans les listes dynamiques de [DaVo93]. Ainsi, chaque mouvement possède une durée d'interdiction propre, qui peut cette fois changer pendant l'exécution de l'algorithme. Il faut alors déterminer quel mouvement considérer, puis quelle valeur lui assigner en fonction d'une règle initialement définie, ou en fonction des caractéristiques des éléments considères, ou encore de manière aléatoire dans un intervalle donné.

Cependant, de telles démarches peuvent devenir rapidement très coûteuses en temps de calcul et en place mémoire suivant le type des éléments considérés. Une alternative efficace consiste à **mémoriser certains attributs du mouvement** considéré (pour passer de `u` à son voisin) plutôt que le mouvement lui même. Ainsi, dans les étapes suivantes, un mouvement quelconque sera interdit si tous ses attributs sont présents dans les listes, selon la nature et le nombre de ces mêmes attributs. Ce gain de mémoire ne sera efficace que si l'on est capable de sélectionner de manière pertinente les « bons » attributs .

Mais dans tous les cas, il est nécessaire de déterminer avec précision **des valeurs pertinentes pour la longévité des mouvements tabous** ou de leurs attributs: trop longue elle endommagerait l'efficacité de l'algorithme en rendant éventuellement le problème insoluble (chaque action ne pouvant alors être utilisée qu'une fois, le nombre d'action possibles diminuerait à chaque itération jusqu'à devenir éventuellement nul), et trop courte elle augmenterait les risques de cycle.
Il pourrait donc être utile d'avoir la possibilité de modifier le caractère tabou d'un mouvement…

Degré d'aspiration

Cette notion intervient aux lignes 26 et 33 de l'algorithme.

La méthode Tabou est un cadre souple permettant d'incorporer toutes sortes d'améliorations et de méthodes avancées. Une des plus souvent retenues est le **critère d'aspiration** qui permet de passer outre certains interdits. Son utilisation principale consiste à outrepasser l'interdiction d'un mouvement s'il permet d'obtenir un élément meilleur que la solution trouvée jusqu'à présent.
Les listes tabou, surtout quand elles font intervenir des attributs d'éléments**, peuvent générer des interactions pouvant accroître le pouvoir « tabou »** des listes, et alors restreindre inutilement voire à tort l'ensemble des solutions autorisées à chaque itération. Dans certains cas, elles interdisent non seulement le retour aux dernières solutions visitées

mais également tout un ensemble de solutions dont plusieurs peuvent ne pas encore avoir été visitées, et qui pourtant peuvent très bien être attrayantes voire meilleures que celles rencontrées jusqu'à là.

Cela étant, **il est nécessaire de pouvoir modifier**, et plus précisément supprimer, le caractère tabou d'une action, en particulier quand son application à l'élément courant peut conduire vers une solution estimée intéressante, mais sans augmenter le risque de cyclage dans le processus de recherche. Ce rôle est attribué à une fonction auxiliaire appelée **fonction d'aspiration** et notée ***A()***. En général, son domaine de définition est constitué des valeurs que retourne la fonction objectif, notées $z = f(u)$, pour tous les u déjà connus. Un z donné correspondant à une valeur fixée de *f*, il se peut que ce même z corresponde à plusieurs éléments u, pour peu que ces éléments aient la même valeur de fonction objectif : le nombre de z est donc inférieur ou égal à celui d'éléments u.
Le nombre $A(f(u))$ est appelé **degré d'aspiration** de $f(u)$, et représente un seuil maximum, en dessous (si le but est de MINIMISER la fonction objectif) duquel l'algorithme est assuré de ne pas cycler lorsqu'il considère un élément v telle que $f(v) = f(u)$.
Une action (mouvement) interdite a sera acceptée malgré tout si la condition d'aspiration suivante est satisfaite :

$$f(u \oplus a) < A(f(u)).$$ [8]

La fonction d'aspiration *A* peut être définie de plusieurs manières.
L'une d'entre elles, est de définir pour chacun des éléments $z = f(u)$ de son domaine de définition, la valeur de retour $A(z)$ comme égale à $f(u')$ où u' est la meilleure des solutions parmi toutes celles obtenues précédemment à partir d'un élément ayant la même valeur de la fonction objectif, mais au coup suivant, après qu'il ait subit son action. Pour toute valeur de retour $f(u)$ de la fonction objectif, $A(f(u))$ indique donc la meilleure valeur (de retour de la fonction objectif) rencontrée jusqu'à présent à l'itération suivante, c'est à dire une fois appliquée l'action ou modification.
En posant initialement pour tout z, A(z)=z, on mettra ensuite cette fonction d'aspiration à jour de la manière suivante :

$$A(f(u)) = min(A(f(u)) ; f(u'))$$
$$A(f(u')) = min(A(f(u')) ; f(u))$$

[8] Où $(u \oplus a)$ est l'état obtenu lorsque l'on applique l'action a à partir de l'état u.

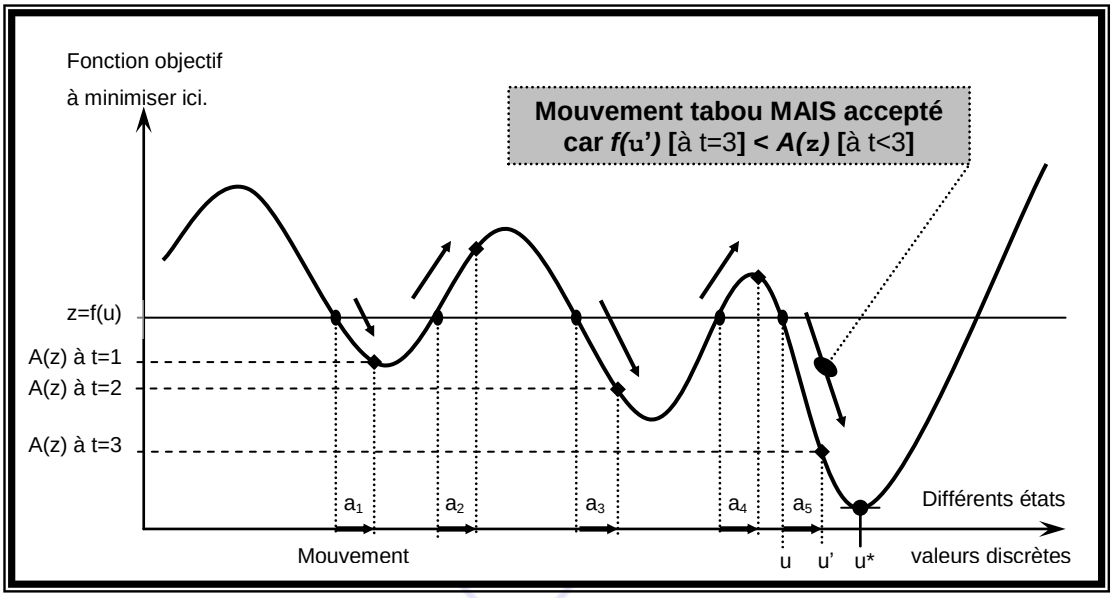


Figure 6 : *Illustration du concept de fonction d'aspiration*

Cette définition de la fonction d'aspiration implique que les valeurs de la fonction objectif soient en nombre fini. Dans le cas contraire, on pourra étendre cette définition en prenant comme paramètre de la fonction *A* non pas des éléments `z`, mais des intervalles $[z_1, z_2]$.

Ainsi, bien qu'on parle de FONCTION d'aspiration, il s'agira dans l'algorithme plus précisément d'une liste contenant les valeurs *A(f(*`u`*))*. Etant donné que nous avons un nombre fini d'éléments `u` (états du système) et que de plus *f()* est surjective (plusieurs antécédents `u` pour une même valeur résultat), on est assuré que notre liste sera de dimension finie et raisonnable.

Critère d'arrêt

Là encore la diversité est reine.

En règle générale, pour interrompre l'algorithme, on prend en compte deux critères. Tout d'abord on vérifie à chaque itération que le **compteur d'itérations totales** n'a pas dépassé un nombre maximal noté `NB_TOT` depuis le début du processus (Cf. ligne 42 de l'algorithme).
On se base sur une autre valeur seuil, `NB_MAX`, qui correspond au nombre maximal **d'itérations que l'on s'autorise entre deux modifications** (améliorations) consécutives de la (meilleure) solution (Cf. ligne 39 de l'algorithme).

Mais au lieu de considérer le nombre d'itérations, on pourrait aussi se baser sur le **temps total**, qui devrait être inférieur à une valeur maximale.

Avantages et inconvénients

Si l'algorithme de base de la méthode tabou est facile à implémenter, le nombre d'études menées sur cet algorithme, ses applications, ses améliorations possibles et les divers mécanismes qui peuvent lui être adjoints sont tels qu'il devient **difficile de savoir quelles sont les options à retenir pour une implémentation particulière**. En envisageant seulement le problème de la gestion de la liste tabou, des critères d'aspiration et des processus d'intensification et de diversification vus plus haut (sans compter la détermination du voisinage et de la condition d'arrêt), on se retrouve face à une grande combinaison de choix dans laquelle il est difficile de faire le tri.
Le plus souvent, il s'agit de principes très généraux qui laissent donc beaucoup de liberté dans leur application, mais qui nécessitent un important travail de mise au point de l'implémentation et de paramétrage pour obtenir des performances satisfaisantes. Ce travail n'est donc réellement effectué que lorsque l'implémentation de la méthode de base ne sonne pas de résultats satisfaisants ;

De plus, contrairement au recuit simulé, il n'y **a pas de résultats théoriques qui garantissent la convergence** de la méthode Tabou vers une solution optimale. Celle ci étant en effet hautement adaptative et modulable, son analyse par des outils mathématiques traditionnels est malaisée. Cependant, à condition de modifier la fonction objectif et les probabilités de génération, certains résultats montrent que la méthode Tabou converge, mais après un temps infini ce qui serait justement ce qu'on cherche à éviter ! De toute façon l'important n'est pas de trouver la solution optimale, mais de visiter au moins une solution de bonne qualité au cours de la recherche.

CONCLUSION

L'avantage de ces méthodes est assurément leur **généricité**. En effet, elles peuvent être appliquées à un grand nombre de problèmes réduisant ainsi la tache de développer entièrement un algorithme au « simple » fait d'adapter l'heuristique (existante) au problème considéré. Cependant, cet avantage a son **revers de la médaille** : ces méthodes ne sont **pas optimales**, puisque non adaptées structurellement au problème. Surtout que pour certaines d'entre elles, les adapter de façon efficace au problème considéré peut finalement s'avérer au moins compliqué: leur structure repose souvent sur l'utilisation de paramètres supplémentaires (par exemple la température pour le Recuit Simulé ou encore le degré d'aspiration pour la méthode tabou) pour lesquels il est nécessaire de déterminer des valeurs pertinentes, afin de rendre l'algorithme efficace et donc justifié, ce qui n'est pas forcément d'une grande simplicité !

De plus, certaines de ces techniques ont un véritable intérêt lorsque les arrêtes reliant les nœuds ou états ont des **valuations distinctes**.

Par exemple dans le cas du voyageur de commerce, les arrêtes représentant la distance entre deux villes (les deux nœuds extrémités), le but est de trouver un chemin, de profondeur connue et constante (le nombre de villes à visiter), de manière à obtenir un ensemble (séquence) d'arêtes de coût total minimum. Un état est donc une séquence de villes, une DES villes. Le nombre de ville étant constant et le but étant de n'y passe qu'une fois, toutes les séquences envisagées ont un même nombre d'éléments (villes).

Par contre, dans la planification, deux nœuds représentent deux situations (ensemble de propositions) voisines. Une arrête correspond à l'action qui permet de passer d'une situation à une autre. La valuation de l'arrête est donc le coût de l'action, tous égaux. Dans l'anomalie de *Sussman* par exemple, toutes les actions ont la même « importance » : *cout(*`A:B→C`*)* = *cout(*`B:A→Table`*)* = ...

Dans la planification, c'est donc la profondeur qu'il faut minimiser, les valuations étant égales, il faut **le faire en un minimum d'itérations**. Ces techniques aurait donc un intérêt réel si les actions possibles avaient des coûts distincts. Par exemple, il est préférable (de coût moindre) de soulever et déplacer une caisse vide plutôt qu'une caisse pleine.

INTERETS POUR LA PLANIFICATION

Lorsqu'elles sont utilisées dans l'optimisation du problème du voyageur de commerce par exemple, chaque élément `u` que ces méthodes visitent est une configuration possible, à savoir un parcours passant une fois et une seule par chacune de toutes les villes. Chaque élément est donc bien une solution potentielle, l'objectif étant d'en trouver une optimale : la fonction objectif retourne la longueur d'un parcours, qu'il s'agit bien ici de minimiser.

Par contre, dans la problématique de la planification, **on ne possède pas au départ de solution à optimiser**, à savoir une séquence valide d'actions, amenant de l'état initial au(x) but(s) à atteindre, et de longueur plus ou moins importante qu'il s'agirait de réduire. Nous ne pouvons donc faire autrement que de fusionner les étapes d'extraction et d'optimisation de solution, en admettant que cela soit possible.

Par conséquent, **une approche pourrait consister à** prendre comme élément (solution) initial l'état initial du système. Chaque solution future correspondrait à un ajout d'une ou plusieurs actions (au milieu ou à la suite ?…). La fonction objectif, prendrait alors en compte à la fois :

- le nombre de conditions (propositions) respectées par la séquence,
- le nombre de celles violées (qui n'est pas complémentaire) et
- le nombre d'actions dans la séquence considérées.

Puisque toute séquence solution respectera le même nombre de conditions (toutes) et en violera autant (aucune), c'est sur le nombre d'actions dans la séquence que portera

l'optimisation. Il est alors important de **normaliser ces 3 critères**, en leur conférant dans la fonction objectif une importance relative au rôle qu'ils jouent. Comme contre exemple, une somme des valeurs brutes ne convient pas : une séquence de 500 actions (+500) qui vérifie 3 conditions (-3) et en viole 2 (+2) soit une qualité de (+499), n'est pas meilleure qu'une autre séquence de 700 actions (+700) qui vérifie toutes les conditions (-10) et n'en viole aucune (+0) de qualité (+690), bien au contraire…

Comme on peut le voir ici, l'efficacité de la technique utilisée sera directement déterminée par la **pertinence de la fonction objectif**.

Après avoir étudié ces différentes méthaheuristiques, il serait interessant de les implémenter non pas de manière exclusive, mais au contraire de **façon complémentaire**, comme des « moteurs » distincts de résolution. On pourra alors choisir lequel utiliser en fonction peut être du types d'actions considérées par exemple, ou bien encore faire des tests comparatifs entre eux sur un problème donné afin de comparer leurs performances.

L'intérêt de cette étude préalable, est de nous avoir permis **d'entrevoir les points** sensibles à considérer, afin de pouvoir par la suite envisager des solutions adaptées et donc plus efficaces, en prenant pourquoi pas ces techniques comme point de départ.
Il serait en effet avisé de **retenir pour la suite** les notions suivantes :
fonction d'évaluation et d'estimation, monotone, ou température variable,
diversification et intensification,
mémoire flexible, et liste tabou : mémorisation adaptable des mauvais éléments,
degré d'aspiration ou l'aptitude à « gracier les gentils bandits »,
Ainsi, nous savons désormais que la roue existe, mais que la roue ovale n'est pas la plus efficace…

Il est à noter enfin, que les méthodes d'exploration locale présentées ici, comme d'autres, utilisent le choix aléatoire pour limiter l'impact et s'affranchir des vastes domaines sur lesquels portent l'exploration. Je pense notamment aux **arbres de décision** et aux algorithmes associés, comme ceux utilisés notamment en **datamining** pour l'extraction ou la reconnaissance de formes dans des bases de très grandes tailles. Serait ce là une autre ouverture ?

Une fois ces méta-heuristiques implémentées pour le choix des actions, il reste cependant à étudier comment envisager la structure de représentation des données, à savoir les actions et les états du système. Pour cela, nous allons étudier un exemple de planificateur nommé GRAPHPLAN.

Description de GRAPHPLAN

QUELQUES HYPOTHESES INITIALES DE SIMPLIFICATION

Avant de se lancer dans la description de l'algorithme, il est utile de préciser quelques hypothèses de départ à respecter dans l'implémentation afin de s'assurer de la pertinence du planificateur.

- **Temps atomique** : l'exécution des actions est indivisible et non interruptible.
- **Effets déterministes** : l'effet de quelqu'action que ce soit, et l'état du monde une fois cette action exécutée, sont déterminés (connus).
- **Omniscience** : le Planificateur a une connaissance totale et a priori du monde initial et de ses propres actions.
- **Causes de changement de bas niveau** : les seules possibilités de changements du monde, considéré comme statique par défaut, sont les actions possibles du Planificateur.

DEUX PHASES SUCCESSIVES:

On peut maintenant détailler le fonctionnement de *Graphplan*, qui fonctionne en deux étapes successives.

- **expansion du graphe**,

  Création d'un graphe de planning en avant dans le temps, jusqu'à réalisation d'une condition nécessaire mais NON suffisante pour l'existence.

- **extraction de la solution**.

  Recherche en chaînage arrière d'un plan qui résoudrait le problème,
  SINON, on étend le Graphe d'un niveau supplémentaire (appel de la phase d'expansion).

AVANTAGE : augmente le nombre d'actions explorées donc la flexibilité,
INCONVENIENT : augmente le temps de résolution

NOTION DE MUTEX (MUTUAL EXCLUSION RELATIONSHIP) :

Une des notions fondamentales sur lesquelles repose *Graphplan*, est la relation d'exclusion mutuelle. Cela traduit le fait que deux actions ou deux propositions peuvent se produire simultanément.

Les différents types de *Mutex*, sont :

- **Entre deux actions** instanciées,
  - **Effets inconsistants** :
    lorsque des deux actions, la première a au moins un effet qui est la négation d'un des effets de la seconde action.
  - **Interférence** :
    si l'une des deux actions supprime une pré-condition ou un *add_effect* de l'autre.
  - **Besoins compétitifs** :
    lorsque des deux actions, la première a au moins une pré-condition qui est Mutex avec une pré-condition la seconde action.
- **Entre deux propositions**
  - Si l'une est négation de l'autre,
  - Au niveau du support (actions du niveau précédent)

L'absence du Mutex est appelée **consistance**.

Concrètement, gérer la notion de *Mutex* permet de traduire le fait par exemple qu'un objet ne peut être en deux endroits différents en un même instant !

Déterminer l'ensemble de tous les Mutex qui s'appliquent à une action peut s'avérer aussi difficile (et long) que de trouver une solution valide. Le but est donc de traduire, et propager, les Mutex évidents sous forme de règles ou contraintes, afin d'éviter au maximum des axes de recherche infructueux, et de vérifier au moment de l'extraction de la solution s'il n'existe alors d'autres Mutex. Empiriquement, c'est la détermination des *Mutex* qui prend le plus de temps dans la création du graphe.

Enfin, il faut distinguer les **Mutex persistants**, qui traduisent une exclusion permanente comme c'est le cas entre « A » et « non A », et les **Mutex non-persistants**, qui traduisent une exclusion temporaire. Ce deuxième type d'exclusion peut disparaître par exemple avec l'apparition à un niveau k de l'action « no-op » dans le domaine d'une proposition, si cette proposition est apparue pour la première fois au niveau précédant (k-1). L'apparition de cette nouvelle action dans le domaine fourni en effet un autre moyen de soutenir cette proposition, et donc d'éviter le Mutex. Elle peut donc être maintenue par rapport au niveau précedant, sa génération étant alors repoussée à un prochain niveau.

OPTIMISATIONS POSSIBLES :

- Extraction de solution :
  - *forward checking*
  - *memoisation,*
  - apprentissage basé sur l'explication.
- Expansion du graphe :
  - considération de la croissance des mondes fermés,
  - compilation des schéma d'action pour supprimer les termes statiques par l'analyse de types,
  - *regression focussing,*
  - expansion de graphe *in-place*.

Il faut garder à l'esprit, que de manière générale, les meilleures optimisations que l'on puisse faire sur un algorithme de planning, reste encore de l'adapter de manière spécifique au problème considéré.

Améliorations apportées

QUE REPRESENTER : UN ETAT DU SYSTEME OU LA SEQUENCE D'ACTIONS ?

Un espace de Mondes

Un Monde contient un état, une configuration, possible du système.

Tout au long de ce rapport, nous nous efforcerons d'illustrer la théorie à l'aide d'un exemple concret appelé l'anomalie de *Sussman* dans le monde des cubes :
trois cubes A, B et C sont dans la configuration initiale où B est sur la Table et A est sur C qui est lui même sur la Table.
Le but est d'arriver à avoir les 3 cubes empilés dans l'ordre alphabétique, à savoir A sur B, B sur C et C sur la Table (A / B / C / Table).
En clin d'œil à M. Adjout, pour ceux qui ont compris la problématique un tri par ordre alphabétique ne saurait être une solution: le planificateur doit également fonctionner dans le cas où le but serait d'avoir A sur C, C sur B et B sur la Table (A / **C** / **B** / Table) ! ! !

Dans cet exemple, un Monde est la disposition des cubes les uns par rapport aux autres. Par exemple, le Monde final est :

```
(on A B) (on B C) (on C Table) (clear A)
```

La recherche de solution dans un espace de mondes peut se faire selon différents algorithmes ou heuristiques déjà bien éprouvés : en largeur d'abord, en profondeur d'abord, A* (théorie des jeux)…
De plus, la recherche peut se faire de manière prospective (procédure `ProgWS` pour *Progression in a World Space*) ou rétrospective (procédure `RegWS` pour *Regression in a World Space*). Cette deuxième possibilité est probablement plus intuitive.

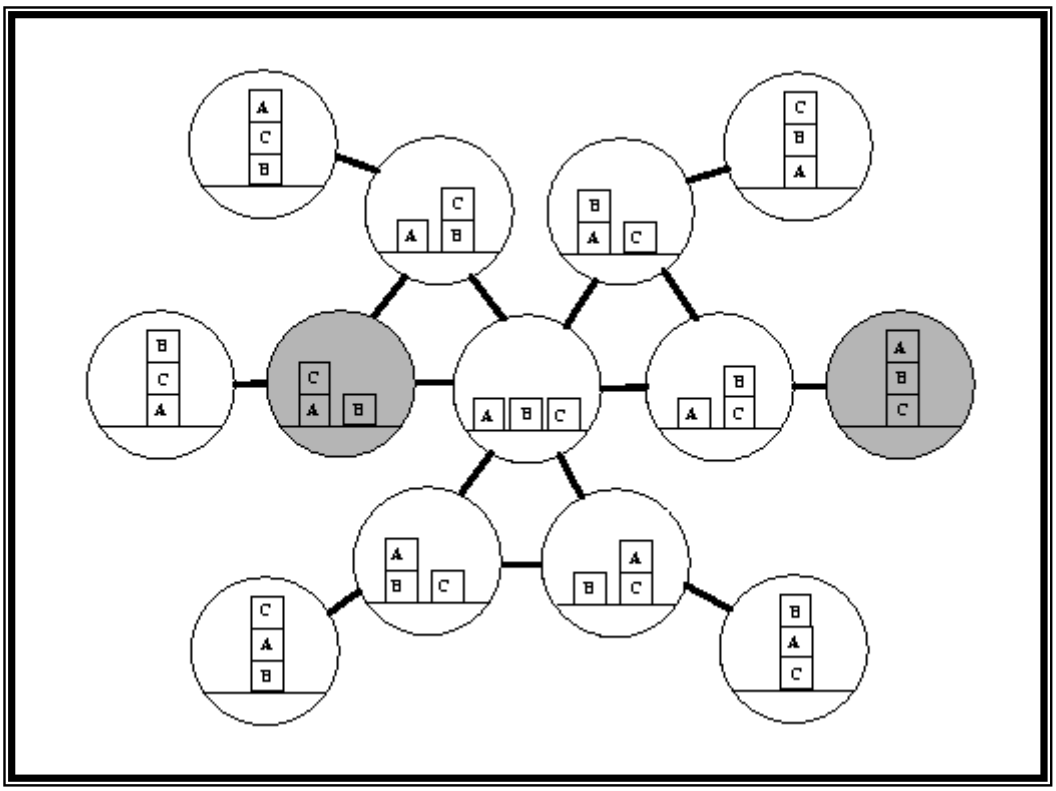


Figure 7: *Représentation du Monde Des Cubes en Graphe de Mondes*

Un espace de Plans

Par opposition au Monde, un Plan contient un ensemble d'actions permettant d'accéder à un monde donné en partant du monde initial.

Dans l'exemple des cubes, **un** plan final serait :

```
Move_C_from_A_to_Table ,
Move_B_from_Table_to_C ,
Move_A_from_Table_to_B .
```

Comme le souligne l'article indéfini devant « plan final », l'espace de plan permet de considérer les notions de **non unicité** et d'**ordre partiel** : pour atteindre nos buts finaux en partant de l'état initial, il existe plusieurs chemins, plusieurs plans ! Cela confère des avantages certains à la conception d'espace de Plans, justifiant ce choix par rapport au premier, mais nous y reviendrons par la suite.

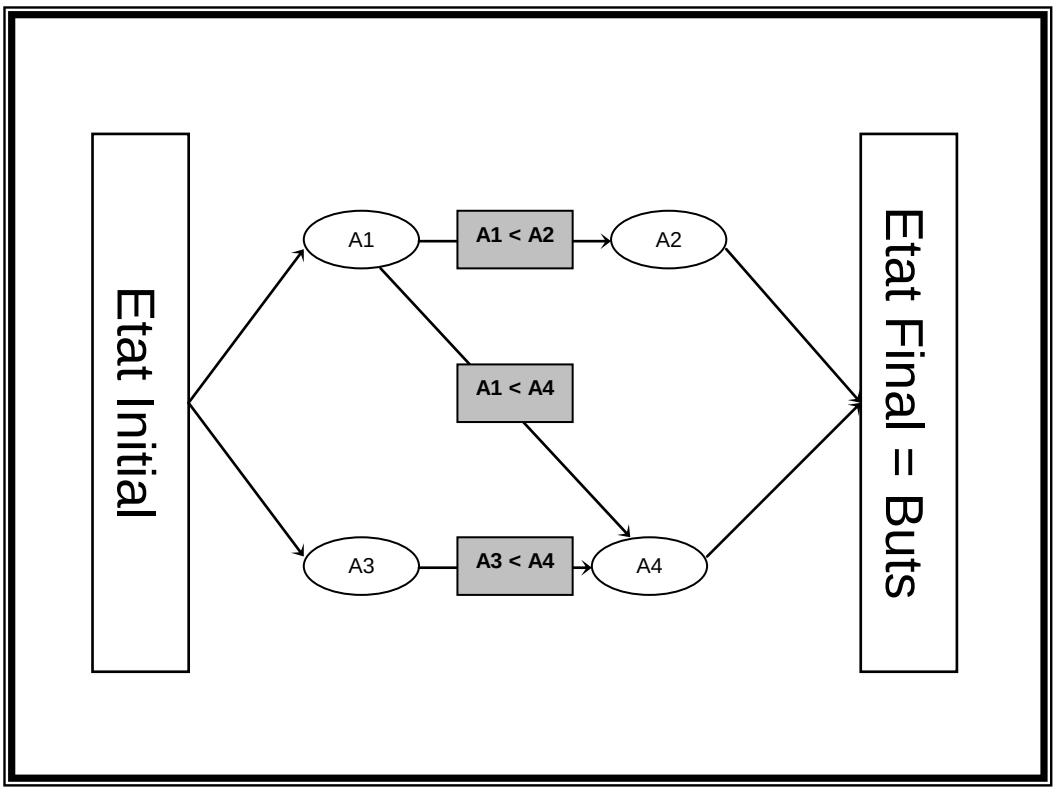


Figure 8: *Exemple de Plan*

Ce plan linéaire peut être linéarisé par exemple de la façon suivant: A1, A2, A3, A4. En effet, cette sequence fait bien apparaître les quatres actions tout en respectant les trois contraintes (dans les rectangles grisés).

TYPES DE RESOLUTION ET SOLUTION OBTENUE

Ordre Total : rigide et peu efficace

L'ordre total consiste à ordonner entre elles **toutes** les actions à réaliser pour atteindre le but, de manière à obtenir une séquence totalement linéaire, sans aucun degré de liberté dans l'absolu. Mais en réalité, un ordre total avec des variables non instanciées impliquerait des degrés de liberté.

L'algorithme permettant d'obtenir un ordre total n'est pas des plus efficace, mais il est simple à implémenter, du moins plus que ceux permettant l'ordonnancement partiel.

Ordre Partiel : diffère la décision MAIS maintient la consistance.

L'ordonnancement partiel du planning, permet de différer la décision le plus tard possible : seules les décisions essentielles sont prises et effectives. Cela confère une plus grande

souplesse au Planificateur, qui peut éventuellement donner comme réponse un ensemble de solutions potentielles, au lieu d'une solution unique. Ainsi, on pourra a posteriori obtenir un ordre total par ajout de contraintes ultérieures et/ou externes.

Comme illustration, imaginons que nous devions planifier le transport de deux caisses C1 et C2 par avion du Bourget à Roissy. Il apparaît évident que C1 doit être chargé avant le décollage, et de même pour C2. Par contre, établir maintenant que C1 doit être chargée avant C2 (ou l'inverse) dans l'avion, nécessiterait un temps de traitement supplémentaire, alors que ça n'a pour l'instant pas d'importance. Par contre, laisser la possibilité de choisir au moment du chargement l'ordre de réalisation permettra de prendre alors en compte la fragilité des caisses, la disponibilité de la main d'œuvre et ses aptitudes, la simultanéité éventuelle des actions (C1 et C2 pourront peut être être (dé)chargées en même temps)…

L'ordonnancement partiel d'actions est donc un ordonnancement total d'ensemble d'actions :

- on doit D'ABORD {« amener l'avion au Bourget »},
- PUIS {« charger C1 », « charger C2 »} dans l'ordre que l'on veut,
- PUIS {« amener l'avion à Roissy »},
- PUIS {« décharger C1 », « décharger C2 »} dans l'ordre que l'on veut,

Mais l'ordonnancement étant partiel, il nécessite de constamment tester et maintenir la consistance, à savoir la cohérence des « sous » séquences d'actions les unes par rapport aux autres. Pour cela on étend la structure du Plan en ajoutant la notion de **Lien Causal**.

RECHERCHE D'UN PLAN

Comme nous l'avons vu, chaque action possède des pré conditions qui doivent être remplie pour que cette action puisse être accomplie, et produire alors ses effets.

Algorithme simplifié permettant de trouver la séquence d'actions solution :

0. l'état courant (constitué d'un ensemble de propositions) est l'état initial, et la solution est vide.
1. Si l'état courant est vide, retourner la solution
2. on choisit une proposition, un but à atteindre, parmi celles de l'état courant,
3. puis une action qui possède comme pré condition cette proposition.
   a) soit cette action existe, dans ce cas l'état courant est :
      i. diminué de la proposition sélectionnée en 2.,
      ii. augmenté des pré conditions de l'action choisie en 3.,

iii. Aller au 4.

b) soit il n'y a aucune action valide, et dans ce cas,

i. soit on peut retourner en arrière, et prendre au niveau précédant une autre action,

ii. soit il n'y a pas de solution, et on sort de l'algorithme

4. On ajoute l'action à la solution,
5. On retourne à l'étape 1.

Cela permet d'établir le Graphe de Planning.

Un Graphe de Planning comporte deux types de niveaux et trois types d'arêtes.

Les niveaux alternent entre des niveaux de propositions qui contiennent des nœuds propositionnels (chacun libellé d'une proposition) et des niveaux d'actions qui contiennent des nœuds d'actions (chacun libellé d'une action). Le premier niveau est propositionnel et chacun de ses nœuds est libellé d'une des propositions des conditions initiales. Il faut noter également l'existence d'actions sans effet (actions « no-op »), qui permettent de propager des propositions au niveau propositionnel suivant.

Les arrêtes représentent les différentes relations possibles entre les niveaux d'actions et de propositions. Un nœud action d'un niveau *i*, est relié :

- aux propositions du niveau *i-1* par des arrêtes « pré conditions »,
- aux propositions qu'il produit (effets) du niveau *i+1* par des arrêtes « *add_edges* »,
- aux propositions du niveau *i+1* qu'il réfute par des arrêtes « *delete_edge* ».

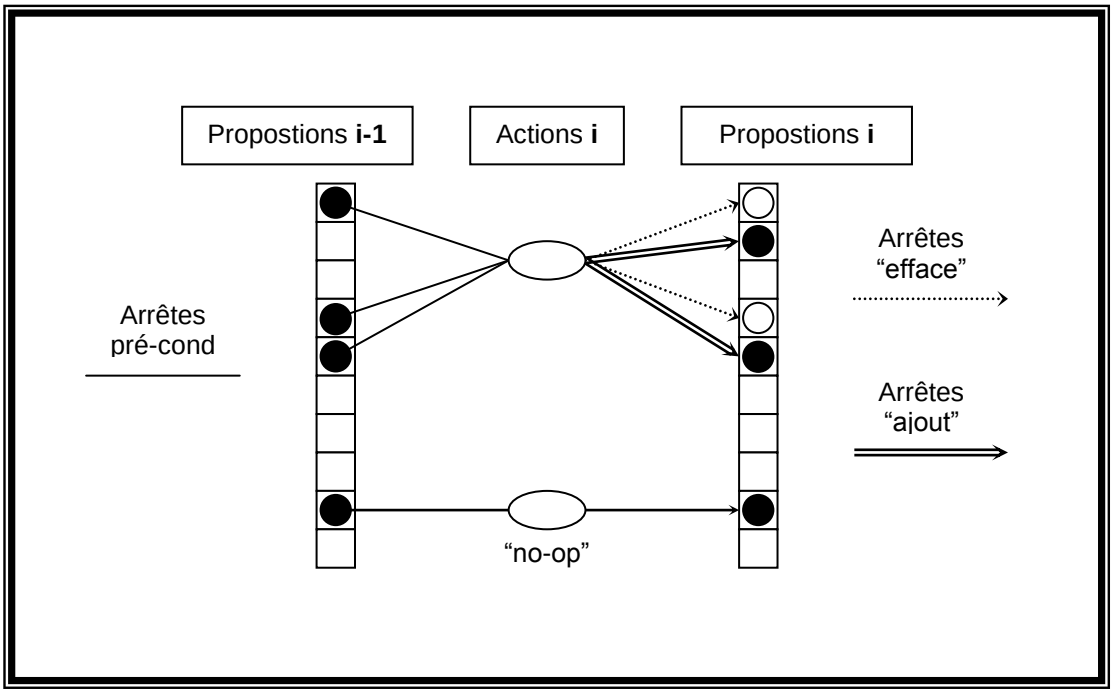


Figure 9: *Transition entre deux niveaux propositionnels consécutifs*

A la vue de cet algorithme, se posent plusieurs questions.

1. Tout d'abord **dans quel sens opérer la recherche**, à savoir de manière **déductive**, en chaînage avant (on part de l'état initial, et on se dirige vers le but à atteindre, ce qui paraît d'une évidence…), ou de manière **inductive**, en chaînage arrière (on part du but à atteindre et on remonte vers les conditions initiales… en fait plus efficace car le facteur de branchage est souvent plus faible).
2. Si l'ordre de sélection (étape 2.) des propositions (buts) à un impact, comment choisir :
   - On cherche à résoudre d'abord tous les buts d'un niveau avant de passer à un autre niveau, ce qui correspond à une **recherche en largeur d'abord.**
   - Ou on prend comme sous but une proposition de l'état initial, trouve une action qui corresponde, puis on prend comme nouveau sous but les pré conditions de cette action, et ainsi de suite jusqu'à arriver au but final, puis on continu avec comme nouveau sous but une autre proposition de l'état initial… sorte de **recherche en profondeur d'abord**.

Un début de réponse à cette question est donné dans le chapitre sur la satisfaction de contraintes, et plus particulièrement la *Mémoisation*…

La stratégie de recherche en largeur d'abord rend le Planificateur relativement insensible à l'ordre de sélection des buts. Approfondissons cet aspect.

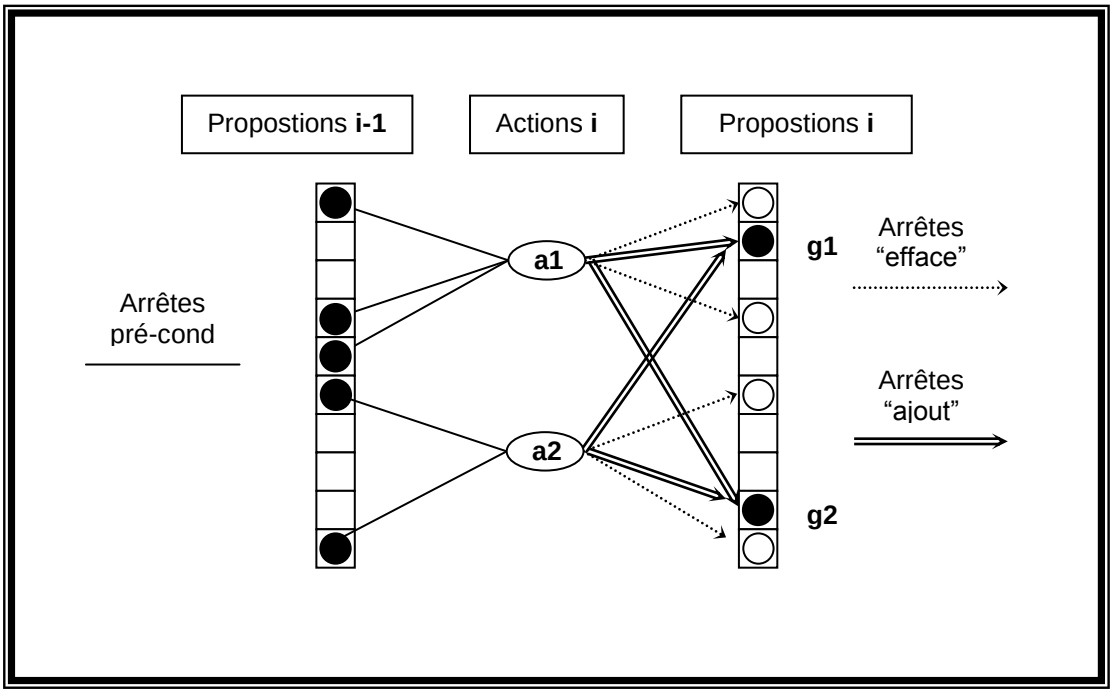


Figure 10: *Contre exemple d'ensemble **minimal d'action accomplissant** les buts*

On dit qu'un ensemble non exclusif d'actions A à un temps t-1 est un ensemble minimal d'actions accomplissant G si :

1. chaque goal de G est un *add_effect* d'une action A, et
2. aucune action ne peut être supprimée de A de sorte que les add_effect des actions restantes contiennent encore G

{a1, a2} n'est pas un ensemble minimal accomplissant {g1, g2}, car on peut ici supprimer a1 sans faire disparaître ni g1 ni g2.

PLANS, LIEN CAUSAL ET MENACE

Le fait de considérer un ordre partiel, bien que très avantageux, peut donc générer des conflits entre les sous séquences.

Considérons une action A_P (productrice) qui produit un effet E qui est la pré condition d'une autre action A_C (consommatrice) : le Planificateur conclu que A_P devra être exécutée avant A_C. On établi donc un **lien causal** de A_P vers A_C avec l'attribut E.
Supposons maintenant qu'une autre action A_M (menaçante) produise l'effet E' contraire de E. Cette action apparaît donc comme une **menace** potentielle pour l'action A_C donc pour le lien causal précédemment établi. Le Planificateur va donc l'ordonner par rapport à ce lien, en essayant tout d'abord et si possible de la placer avant (***demotion***, $A_M < A_P$), sinon après (***promotion***, $A_C < A_M$), ou enfin il retournera une erreur dans le troisième et dernier cas.

Pour illustrer cette notion, reprenons notre exemple.

```
AP : Move_A_from_C_to_Table
 ↓   E : (clear C)
AC : Move_B_from_Table_to_C


AM : Move_A_from_Table_to_C
 ↓   E' : not(clear C)
….
```

Comme l'action A_M produit la proposition `E' = non(E)`, elle est menaçante pour l'action A_P qui elle produit la proposition `E`.

L'HYPOTHESE DES MONDES CLOS

Cela consiste à poser comme faux (négatif) toute proposition dont on ne sait si elle est vraie (positive) ou fausse.

Dans l'anomalie de *Sussman*, les propositions (vraies ou positives) initiales sont :

```
(on B Table)
(clear B)
(on C Table)
(on A C)
(clear A)
```

Ainsi, en considérant l'hypothèse des Mondes Clos, on obtient à posteriori des propositions qui sont par définitions fausses ou négatives et, notamment, dans notre exemple :

```
not (clear A)
```

L'intérêt de cette hypothèse apparaît si le planificateur veut comme toute première étape de la résolution instancier l'opérateur `move` de manière à prendre dés le départ le cube B pour le poser directement sur le cube C, sans avoir au préalable enlevé le cube A .

SCHEMA D'ACTION, ANALYSE DE TYPE ET SIMPLIFICATION

Comme nous l'avons vu en introduction, les auteurs de STRIPS ont choisi un formalisme de conjonctions pour exprimer les pré conditions et les effets d'une actions.
Cette représentation étant très limitée, nous allons voir certaines extensions permettant de rendre le langage d'action plus expressif.

Schéma d'action

Cela consiste a paramétrer les actions, ce qui permet en fait de **factoriser** toute action ayant le même comportement pour des valeurs différentes de leurs paramètres, et limite donc la duplication du code, avec les avantages qui en découlent.
A l'inverse, par la suite lors de son fonctionnement, l'algorithme instanciera les variables, pour établir l'ensemble des actions effectives obtenues à partir d'un schéma.
La logique propositionnelle manipulant des variables discrètes, cela permet de se rapprocher d'une réalité continue.
Lorsqu'elle est paramétrée, une action est appelée **opérateur**.

Contraintes de dépendance

Ce paramétrage implique de traduire explicitement des contraintes qui étaient implicites avant, et qui sont fondamentales pour le bon fonctionnement du Planificateur.
Concrètement, lorsque l'on considère l'action de « prendre `premier_objet` et le placer sur `second_objet` », il est nécessaire pour assurer la stabilité du Planificateur, de préciser que `premier_objet` et `second_objet` doivent être différents. Bien que cela puisse nous

paraître absurde, ce problème ne se pose pas lorsque l'on considère l'action « prendre `Cube_B` et le placer sur un `Table` ».

En fait il s'agit d'ajouter des **contraintes** dites **de dépendance** qui spécifient qu'une variable donnée peut être égale (**codesignation**) ou non (**noncodesignation**) à une valeur ou une (des) variable(s).

Dans notre exemple, ces contraintes se traduiraient comme indiqué ci-dessous en gras :

```
(define (operator move))
 :parametre (?bloc ?from ?to)
 :precondition     (and    (on ?bloc ?from)
                           (clear ?bloc)
                           (clear ?to)
                           (≠ ?bloc ?from)
                           (≠ ?bloc ?to)
                           (≠ ?from ?to) )
        : effect   (and    (on ?bloc ?to)
                           (not (clear ?to))
                           (not (on ?bloc ?from))
                           (clear ?from) )
```

Quelques points sensibles

Ce paramétrage nécessite de faire particulièrement attention à certains points qui peuvent devenir critiques.

- Toutes les variables doivent être instanciées dans l'état initial, notamment pour assurer la validité de la quantification universelle…
- Il faut faire attention de ne pas utiliser le même nom (« ?x » par exemple) pour deux variables distinctes.
- Ces contraintes de dépendance peuvent être présentes soit en temps que pré condition d'un opérateur, soit dans l'ensemble des contraintes de dépendance noté ***B***.

Analyse de type :

Mais dans ce cas, les combinaisons possibles deviennent considérables, alors que celles réellement possibles, qui traduisent effectivement la réalité, peuvent demeurer peu nombreuses.

C'est **l'analyse de type** qui permet de réduire ces domaines aux valeurs pertinentes. Tout d'abord elle détermine le type (nom de la variable), qui demeure constant, puis elle repère les valeurs effectivement prises par la variable, qu'elle affecte au domaine.

La forme la plus simple d'analyse de type consiste à parcourir l'ensemble des prédicats pour déterminer ceux présents dans les pré conditions mais absents dans tous les effets : ils sont alors dis **statiques**. Dans l'anomalie de *Sussman*, le prédicat `cube` est statique puisque les

termes de la forme `(cube ?c)` sous entendu `(is ?c Cube)` ne sont présents que dans des pré conditions et jamais dans des effets, même sous forme négative (`not(is Cube ?c)`). Cela vient ici du fait que l'on a posé comme hypothèse de départ que l'on s'interdisait de détruire ou créer des objets.

Une fois les prédicats statiques déterminés, le planificateur remplacera les instanciations d'actions qui ne font intervenir que des prédicats statiques et des constantes, par des instances de bas niveau où les pré conditions sont supprimées.

Concrètement, dans l'exemple suivant,

```
(define (operator drive)
    :parameters    (?v ?from ?to)
    :precondition  (and    (vehicle ?v)
                           (location ?from)
                           (location ?to)
                           (road-connected ?from ?to)
                           (at ?v ?from)
    :effect        (and    (not (at ?v ?from))
                           (at ?v ?to))
```

si les valeurs des 3 paramètres ont été précédemment établis comme appartenant chacune uniquement au domaine d'un prédicat statique (ici `vehicle`, `location` et `road-connected`), alors cet opérateur est remplacé par une instance où les 4 termes en gras sont supprimés:

```
drive-Iveco-Paris-IssyLesMx
    :precondition  (at Iveco Paris)
    :effect        (and    (not (at Iveco Paris))
                           (at Iveco IssyLesMx))
```

Reste à savoir si ce gain en terme de rapidité au niveau de l'instanciation ne va pas se faire au détriment de la stabilité et de la robustesse du planificateur. En effet, il ne faut pas que la suppression de contraintes emmène des incohérence au sein du système.

En fait, l'intérêt de l'analyse de type apparaît véritablement lorsque l'on considère la résolution du planificateur en terme de Problème de Satisfaction de Contraintes…

EFFETS CONDITIONNELS

Structure

On les utilise lorsque l'on a des actions paramétrées et que l'on veut ajouter des effets supplémentaires **uniquement** pour certaines valeurs de paramètres.

Dans notre exemple, lorsque l'on prend un cube sur la table pour le poser sur un autre cube, l'origine (la table) ne devient pas `clear` après l'action, contrairement aux cas où l'origine est un cube.

Il serait donc intéressant d'étendre la structure d'un effet pour prendre cela en compte. Pour y parvenir, on lui ajoute une clause `WHEN` qui possède deux attributs `antecedent` et `consequent`. Le premier est l'ensemble des pré conditions que doit vérifier le système pour être affecté des effets exprimés par le second attribut.

Un effet conditionnel est de la forme [`(*)` = forme conjonctive ou simple littéral] :

EFFET
- logique
- **WHEN**
  - **antecedent** (*)
  - **consequent** (*)

Mise en œuvre

Dans le cas où une action possède un effet conditionnel, ce dernier a une structure similaire à celle de l'action elle-même : on peut assimiler l'`antecedent` aux pré conditions et le `consequent` à l'effet lui-même. On a donc une en quelque sorte une structure récursive, qui pourrait nous orienter vers une structure à base de frames pour la représentation:

ACTION
- **precondition**
- **effect**
  - logique
  - **when**
    - **antecedent**
    - **consequent**

L'effet va donc se comporter comme une « sous » action, et son traitement algorithmique sera semblable à celui de l'action.

Tout d'abord, comme pour une action (nouvellement ajoutée au Plan) on ajoute ses pré conditions aux buts à atteindre, il faut faire de même avec l'`antecedent`.

De plus, tout comme l'action est sujette aux Mutex, il en est de même pour les effets conditionnels, à une différence près. En effet, il faut aussi tester la consistance pour un effet conditionnel. Mais, en présence d'un `consequent` menaçant, en plus des deux traitements *demotion* et *promotion* disponibles pour résoudre un conflit dans le cas d'un effet non

conditionnel, on a ici à notre disposition une troisième possibilité: la **confrontation**. Ainsi, si un effet conditionnel est menaçant, on pose le contraire de son `antecedent` comme nouveau sous but à atteindre, et si on parvient à l'atteindre le `consequent` menaçant est évité.

STRUCTURE DISJONCTIVE

On peut convertir les pré conditions d'une action et les `antecedant` d'un effet en union de propositions, voire même en Formes Normales Disjonctives (DNF) à l'aide d'algorithme éxistants. Mais bien que cela permette de traduire plus aisément la réalité, la recherche dans un graphe prenant en compte la disjonction est **rapidement explosive** : c'est donc à « consommer avec modération » !

En ce qui concerne l'effet non conditionnel ou le `consequent` d'un effet conditionnel, la disjonction est plus difficile à rapprocher de la réalité dans la mesure où elle est synonyme d'**effet aléatoires**, puisque pour des pré requis identiques, on peut avoir ceci **OU** bien cela comme conséquence, sans pouvoir déterminer laquelle des deux se produit effectivement. Il faut donc alors se rapprocher de la **Théorie de l'incertitude**.

QUANTIFICATION UNIVERSELLE

Toujours dans un but de simplification, de réduction du code et d'augmentation de la fonctionnalité du langage, il serait agréable de pouvoir définir en une seule fois une action, c'est à dire un comportement, dont l'exécution se fasse sur l'ensemble de **toutes** les valeurs que pourrait prendre un de ses paramètres.
C'est à cet effet que la **quantification universelle** a été mise en place, introduisant les opérateurs `foreall` et `exist`.

L'opérateur universel :

L'exemple des cubes n'étant pas adapté pour présenter la structure de cet opérateur, nous allons nous appuyer sur un autre exemple.
Imaginons une serviette sur laquelle seraient posés des objets. Le comportement qui consiste à dire que « si l'on déplace la serviette, tous les objets placés sur cette serviette sont également déplacés », pourrait se traduire en quantification universelle de la manière suivante :

```
(define (operator move)
            :parameters           (?that ?from ?to)
    :precondition  (and    (serviette ?that)
```

```
                                    (at ?that ?from)
                                    (≠ ?to ?from))
:effect         (and    (at ?that ?to)
                                    (not (at ?that ?from))
                        (foreall ((object ?x))
                                    (when   (in ?x ?that)
                                            (and    (at ?x ?to)
                                              (not (at ?x ?from)))))))
```

Cet opérateur peut être utiliser :

- soit dans les pré conditions (la commande UNIX `rmdir` est un bon exemple ici, puisque pour qu'elle soit exécutée, il faut comme pré condition que tous (`foreall`) les fichiers qu'elle contient soient effacés),
- soit dans les effets (ici, c'est la commande `chmod *` qui fait figure de bon exemple, puisque lors de son exécution, tous (`foreall`) les fichiers du répertoire courant sont affectés).

Etant donné le nombre, parfois important, et la diversité, propre à chaque répertoire considéré, de fichiers, il est impensable de coder ces commandes autrement qu'avec un opérateur de type `foreall`.

L'intérêt flagrant de cet opérateur, est de ne pas avoir à spécifier le domaine et le type d'une variable.

Quelques hypothèses initiales

Afin de faciliter la mise en place de la quantification universelle, il est utile de faire quelques hypothèses initiales, à savoir :

- Le Monde du système est représenté comme un **univers fini et statique** d'objets,
- ce qui implique concrètement que la destruction et/ou la création d'objet durant l'exécution est impossible : comme contre exemple prenons l'action qui produirait l'effet
  `(not (insect Coccinelle))`, impliquant la destruction de l'objet Coccinelle.
- De plus, l'objet doit avoir un type, et ce **type doit être spécifié** dans l'état initial, par une assertion du type : `(type_name object_name)` ou `(vehicle Coccinelle)`

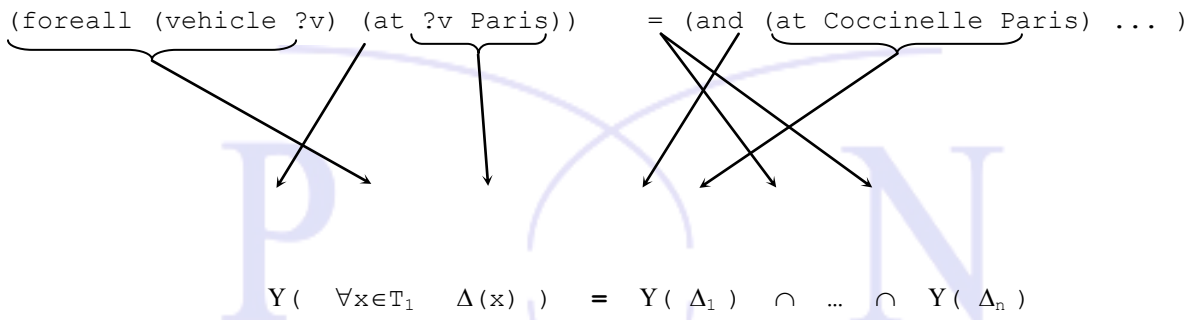
La base universelle

De manière à assurer l'obtention systématique des buts et sous-buts représentés avec la quantification universelle, on a recours à la *base universelle* ou *base de Herbrand*, qui se formalise de la manière suivante :

$\Upsilon(\Delta)$ **=** Δ si Δ ne contient pas de quantifieur,

$Y(\forall x \in T_1\ \Delta(x)) = Y(\Delta_1) \cap \ldots \cap Y(\Delta_n)$

De manière plus concrète, on peut rapprocher cette formule à l'exemple qui consisterait à dire que « toutes les voitures du Monde sont à Paris » :



Les Δ_i correspondent aux interprétations possibles de $\Delta(x)$ selon les différentes valeurs de x dans T_1.

L'opérateur existentiel : `exist`

De même que l'on vient de définir un opérateur universel, on pourrait définir un opérateur existentiel. Mais, cela étant équivalent à des effets disjonctifs, on serait obligé de prendre en considération la notion d'incertitude, comme nous l'avons souligné précédemment.
On peut cependant étendre la base universelle pour prendre en compte la quantification existentielle :

$$Y(\exists y \in T_1\ \Delta(y)) = T_1(y) \cap Y(\Delta(y)) \equiv \exists x / \forall y\ f(x,y)$$
$$Y(\forall x \in T_1\ \exists y \in T_2\ \Delta(x,y)) = T_2(y_1) \cap Y(\Delta_1) \cap \ldots \cap T_2(y_n) \cap Y(\Delta_n) \equiv \forall x / \exists y\ f(x,y)$$

Une fois encore, les Δ_i correspondent aux différentes interprétations possibles de $\Delta(x,y)$ pour y fixé à une valeur donnée de T_2 et pour x prenant la $i^{ème}$ valeur de T_1. De même, $T_2(y_i)$ est remplacé par la $i^{ème}$ valeur de y dans T_2 et est appelé *constante de Skolem.*
Le résultat étant une intersection finie, il apparaît évident que T_2 est de dimension fini.

Ce qui donne pour la première formule, avec notre exemple des situations géographiques des véhicules :

```
(exist ((location ?l))          location = {Paris, IssyLesMx}
      (foreall ((vehicle ?v)) vehicle = {Coccinelle, K2000, GL}
            (at ?v ?l)))
```

correspond à :

```
(and  (location ?l)
      (at Coccinelle ?l)
      (at K2000 ?l)
      (at GeneralLee ?l))
```

Ou encore pour la seconde formule :

```
(foreall ((location ?l))      location = {Paris, IssyLesMx}
     (exist ((vehicle ?v))    vehicle = {Coccinelle, K2000}
          (at ?v ?l)))
```

équivaut à :

```
(and  (vehicle ?v1)
      (at ?v1 Paris)
      (vehicle ?v2)
      (at ?v2 IssyLesMx))
```

REGRESSION FOCUSSING

Le graphe de planning contient l'ensemble de toutes les actions que peut effectuer l'agent implicite qui est représenté par le planificateur. Cependant, une grande partie de ces actions sont sans intérêt pour le (sous) but que le Planificateur est en train de considérer. En fait, la phase d'expansion de graphe ne tient actuellement pas compte du but que le Planificateur cherche à résoudre, ce qui provoque des pertes de temps inutiles, notamment dues à la considération de Mutex, qui sont pour le compte sans rapport avec le but à atteindre.

L'idée serait donc de travailler sur un graphe d'avantage orienté vers les buts.
Un **graphe de génération de faits** est un graphe AND-OR créé à partir des buts et domaines des actions. Les nœuds sont des conjonctions ou des disjonctions, de buts ou d'actions, selon la profondeur considérée.

- La racine, qui représente le but à atteindre, est la CONJONCTION de sous-buts. En effet, pour que le but final soit vérifié, TOUS les sous-buts doivent l'être également.
- Par contre, un sous-but est la DISJONCTION des actions qui peuvent le supporter (ayant ce but comme effet), car l'exécution d'UNE SEULE action amenant à ce sous-but est suffisante.

SCHEMA

Au risque de rendre le graphe incomplet, on peut accroître la rapidité du Planificateur en ne lui soumettant que les actions qui apparaissent dans le graphe de génération de faits. Cette approche est similaire à celle du filtre de McDermott.

L'EXPANSION DE GRAPHE IN-PLACE

L'idée est de supprimer la duplication du travail, en mémorisant une nouvelle fois une information sur les traitements précédemment effectués.

Notons tout d'abord :

- que les propositions et les actions sont croissantes (si présente dans un niveau i, alors présente dans tout niveau supérieur),
- et que les *Mutex* et *Nogood* sont décroissants (si présent dans un niveau i, alors présent dans tout niveau inférieur).

Il apparaît alors intéressant de les **affecter d'un entier** :

- pour les propositions et les actions, l'entier représente le niveau de la première apparition,
- pour les *Mutex* et *Nogood*, il indique au contraire le niveau de la dernière apparition.

Le gain en rapidité et en espace de recherche, se fait ici au détriment d'une compatibilité plus délicate.

SERIALIZATION ORDERING

Cette amélioration pourrait se résumer par la maxime « diviser pour mieux régner ».
En effet, le but est d'arriver à décomposer le problème à résoudre en une série (**serialisation**) de sous problèmes qui devront être résolus successivement dans le temps et dont les buts de l'un seraient les pré conditions de l'autre. Ainsi, une fois obtenue cette séquence de sous problèmes dont on connaît les pré conditions et les buts, la résolution de chacun d'entre eux peut être faite par le Planificateur de manière indépendante, voire parallèle, bien que l'exécution, elle, devra respecter l'ordre.

En fait cette problématique se décompose en deux étapes :

- il faut tout d'abord reconnaître si le problème est sérialisable (beaucoup ne le sont pas !), en détectant par exemple une structure acyclique,
- alors seulement on peut le sérialiser, mais avec prudence, car il est crucial de déterminer un ordre satisfaisant pour ne pas perdre des temps considérables de calcul.

L'utilisation de graphe causal, peut ici se révéler intéressante, dans la mesure où :

- l'acyclicité du graphe prouve que le planning est sérialisable,
- et que dans ce cas, le tri topologique du graphe donne l'ordre de sérialisation.

Le gain de rapidité apporté par la sérialisation tient du fait qu'elle supprime le *backtracking* entre les sous-buts.

INCERTITUDE & REACTIVITE FACE A L'APPARITION DE SITUATIONS IMPREVUES

Comme nous l'avons précisé au départ, jusqu'à présent nous considérions que le Planificateur avait une parfaite connaissance de son environnement, et que les seules modifications qui pouvaient être apportées au système étaient les effets résultants des actions du Planificateur.
Mais comment gérer maintenant le fait que des actions externes au Planificateur puissent modifier le Monde ? Pour reprendre l'exemple des cubes, que se passerait-il si après que le Planificateur ait posé le cube A sur la Table, Mafalda l'emmenait dans sa chambre ? Comment réagirait le Planificateur au moment où il voudrait reprendre le cube A pour le poser sur le cube B ?

A l'automne 1998, la NASA a lancé une sonde qui intégrait notamment un Planificateur. En effet, la problématique des longs voyages dans l'espace est d'économiser au maximum l'énergie pour parcourir des distances toujours plus grandes. Pour cela, il faut laisser éteint un maximum de composants, pour ne les allumer qu'au moment où ils doivent être réellement utilisés.
La partie qui nous intéresse tient au fait que les radiations dans l'espace sont importantes, et par conséquent endommagent fréquemment les différents composants. Le dysfonctionnement d'un composant est bien une situation qui n'a pas été provoquée par le système (la sonde), et qui est donc bien externe au Planificateur. Pour pallier à cela, il y a une forte redondance au niveau du nombre de composants mais aussi de leur diversité : plusieurs moyens sont donc disponibles pour atteindre un but.

A ce niveau, les deux phases suivantes sont à distinguer :

- tout d'abord il faut percevoir que le composant est endommagé,
- ensuite, il faut prévoir que l'effet qui va résulter de l'utilisation du composant puisse être anormal.

Pour ce qui est de la perception des disfonctionnements, des capteurs sont disposés aux endroits sensibles. La modification du système (Monde) est alors représentée sous forme de propositions pour être ajoutée à la base. Par exemple, pour un composant endommagé, il

faudra mettre son état à `off` ou mieux à `break`. Au même titre que dans l'exemple des cubes on mettait `(clear C)` après avoir déplacé le cube A du cube C vers la Table, ici on ajouterait : `(break Component_JC75)`.

Par contre, pour rendre le Planificateur efficace, il faudrait qu'il puisse prévoir qu'un composant soit endommagé, que son utilisation puisse produire un effet différent de d'habitude. Pour cela, on ajoute des **formules de transition**, qui permettent de prendre en compte le fait qu'un composant soit défaillant. Ainsi, pour une même action, on aura plusieurs effets, et donc plusieurs Mondes possibles en sortie, que l'on affecte chacun d'une probabilité. On aura donc comme réponse non pas un état unique, mais une liste d'état possibles ordonnés selon leur probabilité d'apparaître.

L'idéal, serait évidemment que cette probabilité d'existence pour un état donné soit dynamiquement modifiable :

- que ce soit en fonction de ce qui c'est déjà produit par le passé, sorte de *Case Based Reasoning*, ou plus modestement, on pourrait lier par exemple la « probabilité de défaillance » à la fréquence des disfonctionnements effectifs d'un composant,
- ou par l'intermédiaire d'autres règles du genre « si on se rapproche d'une étoile, la probabilité que le composant soit défaillant est plus grande », bien que dans ce cas il soit difficile de donner concrètement une valeur précise à cette probabilité.

En conclusion, la tolérance à l'incertitude, permet de spécifier des informations incomplètes ou des effets incertains, de percevoir et de prévoir le Monde (extérieur).

L'extraction de la solution comme CSP

PRESENTATION

Une fois que la représentation des actions est devenue paramétrée, la résolution du planning peut devenir un **Problème de Satisfaction de Contraintes** (*Constraint Solving Problem* ou CSP).

Les problèmes de satisfaction de contraintes, et les techniques qu'ils mettent en jeu, étant un domaine vaste, c'est volontairement que nous ne les présenterons que de manière succincte, le but ici, étant de souligner les intérêts qu'ils présentent pour la résolution de planning.

PROBLEME DE SATISFACTION DE CONTRAINTES (CSP)

La première et plus sensible étape dans un CSP, est la définition des domaines du problème, à savoir les variables, leurs domaines et les contraintes.

- L'ensemble des n **variables** , $X=\{ x_1, x_2, ..., x_n\}$:
  Les variables sont d'abord les buts puis les sous-buts successifs que doit atteindre le Planificateur.
- L'ensemble des n **domaines** $D_{i/1\leq i\leq n}$:, où D_i est le domaine de x_i :
  Le domaine d'une variable est donc l'ensemble des actions supportées par la variable, c'est à dire celles qui peuvent amener au but en question.
- Les **contraintes** :
  Elles représentent les Mutex qui peuvent exister entre les différentes variables (buts) et/ou les actions.

Le but est d'affecter à toutes les variables une valeur de son domaine de manière à respecter toutes les contraintes (sans en violier aucune). Si on y parvient, on obtient une solution, voire plusieurs. On parle alors d'**instanciation totale consistante,** sachant qu'une instanciation peut être :

- totale (toutes les variables sont affectées d'une valeur) ou partielle,
- consistante (toutes les contraintes sont respectées) ou non.

Parfois, le problème peut être doté en plus d'une **fonction objectif**, dont le but est d'ordonner les différentes réponses que l'on est susceptible d'obtenir, afin d'obtenir la meilleure réponse. C'est une fonction à valeurs réelles de l'ensemble des variables. Pour plus de commodité, nous allons considérer que l'optimisation consiste en fait à minimiser plus qu'à maximiser. L'avantage dans ce cas, est que la fonction sera alors bornée (borne inferieure), par ZERO, alors que si l'on cherche à la maximiser, la borne (supérieur) sera « +∞ » ce qui n'est pas forcément pratique, bien qu'en informatique « +∞ = 2^32 ».

Lorsque les contraintes du problèmes sont peut nombreuses ou peu « contraignantes » il arrive que l'on obtient plusieurs solutions possibles : on dit alors que le problème est **sous-contraint** (dans le cas contraire, lorsqu'il n'y a aucune solution possible, il est dit **sur-contraint**).
On voit ici l'utilité des contraintes de dépendance et de l'affectation initiale de chaque variable (considérée dans le précédant chapitre) comme assurance que le problème n'est pas (trop) sous-contraint (nombre suffisant de contraintes), garantissant ainsi l'unicité des valeurs des variables donc celle de la solution.

Une fois cela établi, on peut schématiser les différentes instanciations successives sous la forme d'un un arbre que l'on appellera l'**arbre de construction des solutions**. Les nœuds sont constitués par des instanciations partielles et les branches représentent chacune une variable instanciée : la branche située entre deux nœud V_i et V_{i+1} correspond à la variable qu'il faut instancier pour passer de V_i (première instanciation) à V_{i+1} (deuxième instanciation « un peu moins » partielle).

En fait, il s'agit ici plus exactement d'un **problème de satisfaction dynamique de contraintes**, car la résolution du CSP correspond à l'affectation de valeurs (actions) aux variables (buts). Or ce traitement doit être fait pour chaque niveau du Planificateur : une fois le traitement terminé pour un niveau donné, on ré-initialise les trois ensembles variables / domaines / contraintes par rapport au niveau suivant.
Ainsi, dans un CSP dynamique, les variables (ou propositions, ici) ont une **durée de vie** au cours de la résolution plus courte que celle de la résolution globale. En effet, en plus des contraintes qui traduisent le problème à résoudre, comme dans un CSP classique, dans le cas dynamique, il y a des **contraintes dites d'activation** du genre : « SI (X > 7) ALORS (Activer Z) », où la propriété « activé » indique à l'algorithme de propagation qu'il doit prendre en compte et propager les contraintes où elle intervient.

ALGORITHME

Lorsque l'algorithme rencontre une instantiation inconsistante, ou qu'il n'y a plus aucune valeur possible pour la variable à instancier, il défait ou dépile l'affectation des précédantes variables jusqu'à retrouver une situation favorable. Ce « retour en arrière » à une étape favorable, qui correspond à une remontée dans l'arbre de recherche jusqu'au nœud précédant, s'appelle le ***backtracking***.

Les **choix de la variable à instancier puis de la valeur à lui affecter** sont déterminant dans l'efficacité de l'algorithme.
En effet, si l'on choisi dés le départ les bonnes valeurs pour chacune des variables à instancier, aucun backtrack ne sera nécessaire, et l'on obtiendra directement la solution.
A l'inverse, si l'on choisi directement une variable dont plus aucune valeur n'est consistante, dont le domaine des valeurs possibles est réduit au vide, on détermine de suite l'impasse et on peut opérer le backtrack sans perdre de temps dans des affectations intermédiaires et inutiles, causes d'une perte de temps donc d'efficacité.
Ces choix sont des points à approfondir pour améliorer l'efficacité de l'algorithme, ce que nous verrons dans les prochains paragraphes.

Une fois la représentation du problème de planning sous forme de CSP établie, on peut alors appliquer les différentes optimisations propres à la satisfaction de contraintes…

LE FORWARD-CHECKING

Lorsque l'on se trouve à un nœud $\mathbf{N_i}$, après avoir choisi une variable $\mathbf{V_i}$ parmi celles non encore affectées, et avant de la tester, on réduit son domaine des valeurs inconsistantes avec les variables dernièrement instanciées : c'est ce que l'on appelle les **techniques à test avant** ou ***Look-Ahead***. Cela permet d'élaguer de manière significative l'arbre de recherche en supprimant tout sous arbre ayant pour racine l'instanciation partielle actuelle augmentée de la variable $\mathbf{V_i}$ à laquelle on a affecté une des différentes valeurs inconsistantes : on voit donc « à l'avance » les échecs potentiels.

C'est à ce niveau qu'intervient l'**analyse de type**. En effet, pour réduire le domaine d'une variable, encore faut il qu'il soit explicitement déterminé. Or, dans le cas par exemple de *l'anomalie de Sussman*, lorsque l'utilisateur va définir le problème en énonçant notamment les opérateurs possibles, il ne va pas énumérer toutes les valeurs accessibles par la variable `?bloc` de l'opérateur `move`. Ce travail, réduit ici, peut en effet devenir conséquent avec de nombreuses variables ayant des domaines étendus. L'analyse de type va donc scruter toutes les propositions, au départ, avant le début de la résolution, et ainsi déterminer pour chaque prédicat, l'ensemble des valeurs qu'il peut prendre, c'est à dire son domaine. Après l'analyse de type sur l'état initial du système, « on » constate que l'opérateur `on` prend toujours l'une des formes `(on A xxx)` ou `(on B xxx)` ou `(on C xxx)`. Ainsi,

lorsque l'on retrouve la variable `?bloc` dans la définition, puis l'instanciation, de l'opérateur `move` sous la forme `(on ?bloc ?from)`, on sait que son domaine est {`A`, `B`, `C`}. Par la suite si l'on a obtient successivement `(≠ ?bloc B)` puis `(≠ ?bloc C)`, on pourra conclure, (et simplifier) que `?bloc` vaut `A`.

Les méthodes à test avant permettent donc de limiter considérablement la quantité de *backtracking* effectués sans toutefois les supprimer totalement.
En effet, lorsque l'algorithme tombe dans une impasse où le domaine (des valeurs encore possibles) de la variable choisie est réduit au vide, on dit alors que le domaine **collapse**, l'algorithme revient à la variable précédemment instanciée. Pour continuer l'exemple du paragraphe précédant, une troisième condition du type `(≠ ?bloc A)`, impliquerait que la variable `?bloc` doit également être différente de `A`, ce qui entraînerait l'absence de solutions possibles ici, provoquant un retour en arrière.

Il est important de noter qu'une valeur choisie puisse être indirectement inconsistante avec l'instanciation partielle, à savoir qu'elle se révélera inconsistante seulement dans plusieurs étapes, après avoir affecter une ou plusieurs autres variables. Cela revient en fait au problème initialement énoncé qui est de choisir dans un ordre judicieux les variables que l'on va affecter.

Une **optimisation** possible est d'arriver à déterminer quelle est la variable qui provoque le *backtrack*, pour y revenir directement, sans tester les variables intermédiaires. Pour cela il faudra mémoriser des information sur les variables précédemment instanciées : c'est l'idée que reprend la ***Memoisation***.

MEMOISATION

Le principe consiste à mémoriser ce qui a déjà nuit ou qui peut nuire à la résolution du problème.
Nous avons vu précédemment, que l'ordre partiel était un ordre total sur des ensembles d'actions, chaque ensemble regroupant toutes les actions qui peuvent être accomplies en une même étape. Dans ces conditions, l'application la plus simple de la *mémoisation*, consiste pour chaque étape (ensemble d'action) lorsque l'on trouve une action valide, à marquer comme infaisable toute action exclusive à la première. Par exemple, si nous avions 2 avions et 2 cargaisons à transporter, après avoir sélectionner l'action « charger C1 dans Avion1 », on marquerait comme infaisable l'action « charger C2 dans Avion1 » (Avion1 étant déjà plein). Cependant, dans le cas de schéma d'action, c'est à dire d'action paramétrées, il faut se demander si la détermination des actions exclusives à l'action en considérée, n'est pas plus long que d'explorer ces actions exclusives, voire même impossible, puis de faire un backtrack…

A un niveau i du parcours de graphe du planning, si un ensemble de (sous) buts {P, Q, R, S} est déterminé comme non résolu, c'est à dire s'il n'a été trouvé aucune action établissant la consistance des sous buts, ayant ces sous buts comme effet, l'ensemble {P, Q, R, S}, appelé **nogood**, est mémorisé, stocké dans une table de hachage et, si jamais il réapparaît dans un niveau k>i, l'algorithme pourra directement établir l'inconsistance du nouvel ensemble de sous buts (qui contient le nogood), sans avoir à parcourir les actions disponibles.

Une amélioration possible, serait de pouvoir déterminer les quels des sous buts provoquent l'inconsistance, pour ne mémoriser qu'eux, cela présentant un double avantage :

- tout d'abord une utilisation moindre de la mémoire,
- mais surtout, cela augmenterait les chance de détecter la présence du véritable nogood, et l'efficacité serait de même accrue. Pour revenir à l'exemple précédant, si seuls R et S provoquent l'inconsistance, mais que l'on stocke {P, Q, R, S}, et si l'on a l'ensemble {R, S, T} qui apparaît dans l'un des niveaux suivants comme ensemble de sous buts à atteindre, l'algorithme cherchera à atteindre chacun des sous buts un par un, provoquant une perte de temps inutile, puisque de toute façon vouée à l'échec.

Une idée pour arriver à ces fins, serait de rapprocher entre eux les différents nogood, pour faire ressortir les plus grandes parties communes.

Toujours dans l'exemple précédant, si à un niveau i on établi le nogood {P, Q, **R, S**}, puis à un niveau j>i on établi le nogood {**R, S**, T}, alors si à un niveau k>j on doit résoudre l'ensemble {G, **R, S**}, il faudrait peut être commencer par R puis par S, car apparemment, il y a de forte chance pour que ces 2 buts soient inconsistants.

Attention, {R, S, T} peut très bien être inconsistant, sans que {R, S} le soit forcément!

Par contre, à l'inverse, si jamais {R, S} est inconsistants, il faut l'ajouter à la liste des nogoods, et en supprimer tous les nogoods qui le contiennent, car si {**R, S**} est un nogood, alors {P, Q, **R, S**}, {**R, S**, T} et tout autre ensemble contenant {R, S} l'est forcément.

Il apparaîtrait donc ici, que l'ordre de choix des buts à atteindre ait un impact sur l'efficacité du planificateur et la rapidité de la résolution.

L'ORDONNANCEMENT DYNAMIQUE DE VARIABLES

L'idée ici, est la suivante. A un moment donné de la résolution, lorsqu'une partie des variables est déjà instanciée, l'ordre d'instanciation des variables restantes sera déterminé en fonction des variables déjà instanciées et de leurs valeurs.

En effet, lorsque l'on est en cours de résolution (d'instanciation), il faut tenir compte des précédents choix d'instanciation et des valeurs choisies pour trouver la variable pertinente qui doit être nouvellement instanciée.

Ces optimisations, comme c'est souvent le cas, doivent faire un compromis entre rapidité d'exécution et espace mémoire utilisé.

ILOG Solver, « My Getting Started »

INTRODUCTION

Trois phases pour coder un problème en *Solver*:

- déclarer les **variables** sous contrainte et leur domaines,
- poster les **contraintes**,
- **chercher** la (les) solutions(s).

POURQUOI cette présentation de *Ilog Solver* version 3.41 ?… Pour deux raisons.

- Tout d'abord, parce que la prise en main de l'aspect fonctionnel de *Solver*, a été une étape à part entière de mon stage, puisque le choix a été fait au départ, après consultation de la littérature su r le sujet, d'utiliser de la programmation par contrainte pour le moteur du planning, et en l'occurrence S*olver* puisque *Pacte Novation* est partenaire avec *Ilog.*
- Mais aussi parce que cette présentation structurelle permet à la fois d'illustrer certains concepts de la programmation par contraintes, et aussi d'avoir une approche pragmatique de la manière de réaliser un moteur de programmation par contraintes, ce qui pourrait être un des prolongement de ce stage, étant acquis qu'en optimisant le moteur pour la problématique de la planification, en s'appuyant sur les travaux réalisés dans ce domaine, les résultats pourraient être améliorés de manière significative.

C'est toujours dans cette optique que j'ai joint en annexe la description des principales classes citées dans cette partie. Descriptions extraites de « *ILOG Solver 4.3, Reference Manual* », mais en ombre restreint, le but étant d'apporter des informations pertinentes, non pas d'accroître de manière significative l'épaisseur de ce rapport !

LE MANAGER

Un manager est un objet C++ regroupant et contenant toutes les données (variables et contraintes) d'un CSP donné.

C'est une instance de `IlcManager`.

En général, besoin d'une seule instance, mais possibilité d'en instancier à volonté : autant que de CSP à traiter.

Pour le créer : `IlcManager m(<mode>);`

Deux `<mode>` possibles pour créer le manager :

- `IlcEdit` (voir chapitre suivant): en mode *edit*, on peut ajouter et retirer contraintes et buts jusqu'à l'appel de la fonction membre `IlcManager::nextSolution` (Cf. paragraphe « La Recherche »). On peut permuter alternativement entre les modes *edit* et recherche.
- `IlcNoEdit` : le manager est créé en mode recherche, et il ne peut plus en sortir. Conseiller quand l'application a été débutée sous une version antérieure à la 3.2 de *Solver*.

L'association des variables et contraintes à un manager se fait en spécifiant le manager au moment de leurs déclarations respectives (Cf. paragraphes « Les Variables » et « Les Contraintes »).

Chaque variable et chaque contrainte ne peuvent appartenir qu'à un seul manager : **les managers ne partagent ni les variables, ni les contraintes**.

En fin d'application l'appel de la fonction membre `IlcManager::end` nettoie toutes les allocations faites pour les variables et les contraintes d'un manager, ce qui rend son appel nécessaire, avant de sortir de l'application, pour chacun de tous les manager instanciés.

```
int main()
{
    IlcManager m(IlcEdit);

    // declaration des variables du probleme
    IlcIntVar aVar(m);
    // declaration des contraintes du probleme
    // debut de recherche de(s) la solution(s)

    m.end();
return 0 ;
}
```

Les principales fonctions membres :

- IlcManager(),
- add(IlcGoal ou IlcConstraint),
- end(),
- fail(<label>),
- openLogFile() et closeLogFile(),
- getNumberOfVariables() et getNumberOfConstraints(),

- getTime(),
- nextSolution(),
- remove(IlcGoal ou IlcConstraint),
- restart(),
- setObjMin(),
- setTimeLimit(tempsEnSecondes),
- solve(),
- storeSolution(),

MODE EDIT

Quand on créé une contrainte, elle n'est pas propagée tant qu'elle n'a pas été ajoutée (associée) à un manager.

Cette propagation peut être immédiate après l'ajout de la contrainte, ou différée : tant que le manager est en mode *edit*, aucune contrainte n'est propagée, ce qui permet alors d'en ajouter et supprimer sans que cela n'ait aucun effet. Le manager ne fait que mettre à jour sa liste de contraintes à satisfaire, sans en propager, sans exécuter de buts, ni modifier les domaines des variables.

Cela ne se fait qu'avec l'appel de la fonction `IlcManager::nextSolution()`, qui indique à *Solver* que l'on met un terme au mode *edit* afin d'obtenir des solutions.

L'avantage, est qu'une fois que l'on a trouvé une solution, on peut très bien souhaiter modifier la modélisation du problème en fonction de cette solution, en ajoutant ou supprimant des contraintes : cela n'est possible qu'en mode *edit*.

Une fois une solution trouvée, on peut choisir de :

- terminer l'application,
- ou de ne terminer seulement CETTE session d'édition, et d'en relancer une autre avec les mêmes paramètres initiaux, pour voir par exemple l'impact de l'ajout/suppression d'autres contraintes. Pour cela, il suffit de faire appel à la fonction membre `IlcManager::restart` (`m.restart()`), afin de remettre *Solver* dans le même état initial.

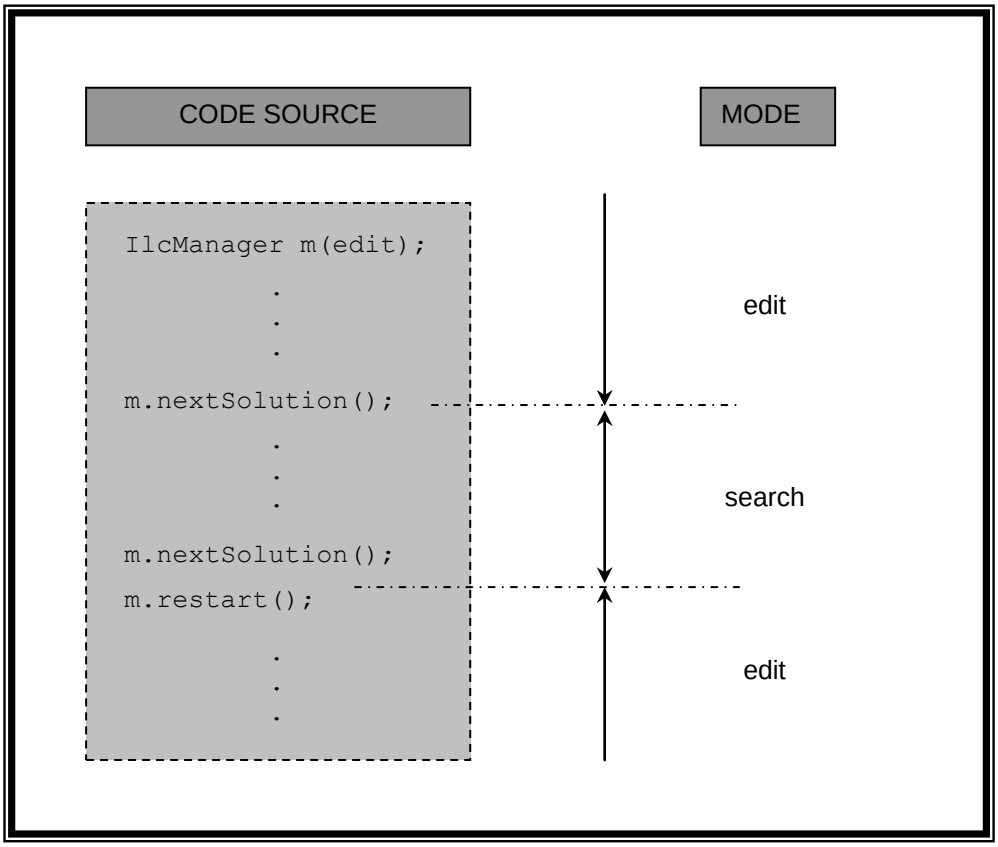


Figure 11: Mode dans lequel se trouve le Manger en différentes étapes de l'algorithme.

LES VARIABLES

Pour *Solver*, les variables sont des objets C++ (appartenance à des classes et fonctions membre héritées).

En tant que tel, selon les **trois types de valeurs** qu'elle peut prendre, à savoir :

- `IlcInt` (valeur entière),
- `IlcFloat` (valeur virgule flottante)
- et `IlcIntSet` (la valeur est alors un ensemble fini d'entiers),

la variable *Solver* est une instance d'une des trois classes `IlcIntVar`, `IlcFloatVar` ou `IlcIntSetVar` , les deux premieres étant respectivement héritées de `IlcIntExp` et `IlcFloatExp`.

Une extension interessante de `IlcIntSetVar` est `IlcAnySetVar`, où les valeurs sont des ensemble finis de variables de type `IlcAnySet`, à savoir des domaines de pointeurs sur n'importe quel type C++.

Les **attributs intéressants** et importants de IlcIntExp, donc d'une variable *Solver*, sont :

- **`name`**, accéssible par les méthodes `setName()` et `getName()` qui permettent de donner un nom à toute variable,
- **`domain`**, et **`domain-delta`** (uniquement pour les types `IlcIntVar` et `IlcFloatVar`), détaillés dans le paragraphe suivant sur les domaines,
- **`object`**, de type `IlcAny`, et accessibles via `setObject()` et `getObject()`. En lui affectant un objet de type `IlcInt` par exemple, on peut représenter l'index d'une variable dans un tableau de variables par exemple.

Les **principales fonctions membres** sont :

- les constructeurs,
- `getManager()`, qui permet de récupérer un pointeur sur le manager de la variable,
- `getValue()` et `isBound()`, détaillées ci-dessous,
- `getSize()` et `isInDomain(IlcAny value)`, détaillés dans le chapitres suivant sur les domaines,
- `removeDomain(IlcAnyArray array)`, `setDomain(IlcAnyArray array)` et `removeValue(IlcAny value)` qui sont également détaillées dans le chapitre sur les domaines,
- `whenDomain(IlcGoal c)` et `whenValue(IlcGoal c)` qui permettent d'affecter une contrainte à ces événements , et ainsi de l'exécuter à chaque fois que le domaine est modifié ou lorsque la variable est instantiée (détaillées dans le chapitre sur les contraintes),
- `setValue(IlcAny value)`.

Des objets sont déjà prédéfinis par *Solver* pour représenter :

- des bouléans (`IlcBool`),
- des entiers (`IlcInt`),
- des nombres à virgule flottante (`IlcFloat`),
- des tableaux de ces deux types (`IlcIntArray` et `IlcFloatArray`),
- des ensembles (`IlcSet`),
- et bien d'autres types.

Ces variables peuvent être combinées avec des fonctions (mathématiques) pour former des expressions puis être intégrées dans des structures de type tableau, ces derniers pouvant à leur tour être combinés toujours en temps que variable…

On peut enfin déclarer des objets dont les attributs sont des variables contraintes.

Pour déclarer une variable, on doit définir son **domaine**, et le **manager** auquel elle est associée.

Dans le cas de deux variables `var1` et `var2`, entières, comprises entre 0 et 9 inclus, et associées au manager `m`, la déclaration se fait de la manière suivante :

```
IlcIntVar var1(m, 0, 9), var2(m, 0, 9);
```

Pour déclarer une variable comme combinaison d'autres variables déjà définies :

```
IlcIntVar var3 = 10 * var2 + var1;
```

Pour déclarer une variable `var4` associée au même manager `m`, ayant le domaine de cardinalité 2 suivant {7, 77}, on écrit :

```
IlcIntVar var4(m, 2, 7, 77);
```

Pour déclarer une variable de type tableau , il existe plusieurs façons, selon les arguments :

- le nom du manager qui contient cette variable est chaque fois nécessaire.
- Le nombre d'éléments de ce tableau.
- la liste de ces éléments,

  ```
  IlcIntVarArray myVarArray(m, 3, var1, var2, var3) ;
  ```
- les valeurs minimale et maximale de chacun de leurs domaines respectifs, tous identiques,
- ou encore les valeurs minimales, maximales et de l'intervalle séparant deux valeurs consécutives du domaine sont par contre les paramètres qui determinent les differentes façons de définir un tableau de variables.

On peut ainsi déclarer dynamiquement des variables, ce qui peut s'averer très utile lorsque le nombre de variables à définir dépend des données en entrée.

```
IlcInt      myIntArray[myDomainSize];
MyInitialize(&myIntArray);
IlcIntArray    myIlcIntArray(myIntArray);
IlcIntVar   myVar(m, myDomainSize, myIlcIntArray);       (I)
IlcIntVarArray myVarArray(m, myArraySize);   (II)
```

`myVar (I)` correspond à une variable dont le domaine est `myIlcIntVarArray`, et `myVarArray (II)` correspond à un tableau de `myArraySize` variables, qu'il faudra définir par la suite.

Pour **obtenir la valeur d'une variable**, on utilise la fonction membre **`getValue`**`()`. Par exemple pour obtenir APRES RECHERCHE la valeur de la variable `myVarArray` on écrit :

```
myVarArray.getValue();
```

Pour être sûr de ne pas générer d'erreurs, il est préferable de s'assurer au préalable que cette variable est bien instanciée :

```
if ( myVar.isBound() )
    return myVar.getvalue() ;
else
    return -1;
```

On peut également **forcer l'instanciation d'une variable** avec une valeur, en utilisant la fonction **`setValue()`**. Mais la façon dont *Solver* gère cette fonction, nous conduit à l'étudier dans le chapitre suivant sur les domaines.

```
IlcIntVar x(m, 0, 1);
x.setValue(1);
```

Cette instruction réduit le domaine de x au singleton {1} : 1 doit appartenir au départ au domaine de x

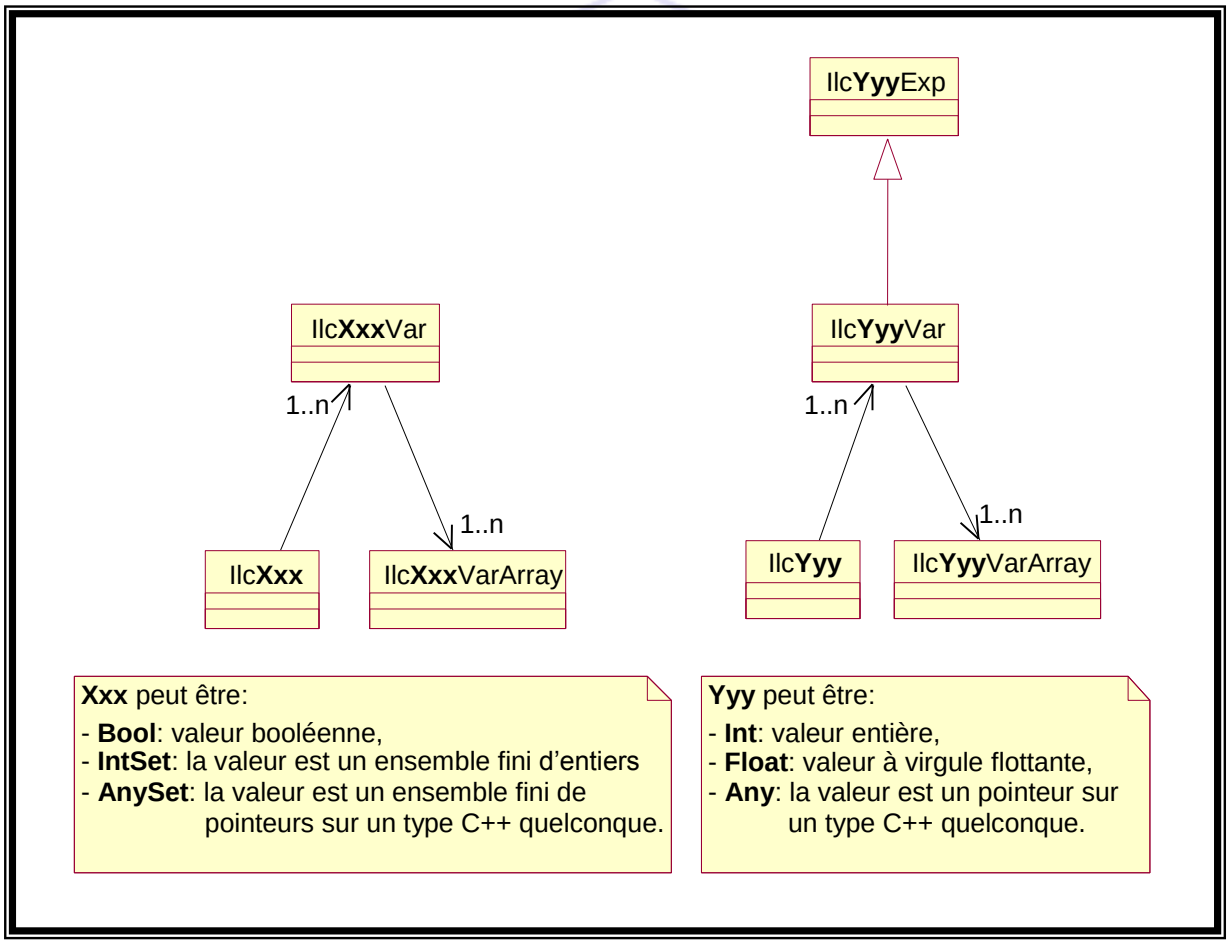


Figure 12: Hiérarchie des types de base dans Ilog Solver.

LES DOMAINES

Le domaine d'une variable est l'ensemble des valeurs ou des domaines possibles pour cette variable.

{0, 1, 2, 3} == [0 ... 3] Notation *Solver* pour un domaine continu
{7, 3} == [3 7] Notation *Solver* pour un domaine discontinu
{0, 1, 2, 3, 7} == [0 ... 3 7] Notation *Solver* pour un domaine discontinu

Solver travaille sur les domaines des variables en supprimant systématiquement toutes les valeurs qui ne satisfont pas les contraintes. Cela s'appelle la **réduction de domaine**.
Cette réduction de domaine peut être forcée de différentes manières par l'utilisateur à l'aide des fonctions membre suivantes:

- **`setMin`**`(IlcInt min)`, qui modifie la valeur minimale du domaine avec la nouvelle valeur `min` qui doit être incluse dans l'ancien domaine,
- **`setMax`**`(IlcInt max)`, de même avec la borne supérieure,
- **`setRange`**`(IlcInt min, IlcInt max)`, qui supprime tous les éléments qui sont soit inférieurs à `min`, soit supérieurs à `max`,
- **`removeRange`**`(IlcInt min, IlcInt max)`, qui supprime tous les éléments qui sont à la fois supérieurs ou égal à `min`, et inférieurs ou égal à `max`,
- **`removeValue`**`(IlcInt value)` qui retire la valeur `value` du domaine,
- **`setValue`**`(IlcInt value)` qui retire du domaine toutes les valeurs différentes de `value`, et si `value` n'appartient pas au domaine `IlcManager::fail()` est retourné,
- **`removeDomain`**`(myIlcSet ou myIlcArray)`, qui retire du domaine l'ensemble `myIlcSet` ou le tableau `myIlcArray`,
- **`setDomain`**`(myIlcSet ou myIlcArray)`, qui retire du domaine les éléments qui ne sont pas dans `myIlcSet` ou dans `myIlcArray`.

Comme le montrent ces fonctions membre, **les domaines ne sont que réduits**, et jamais étendus : cette stratégie est donc monotone. On obtient ainsi comme propriété que **les algorithmes que *Solver* utilise se terminent forcement**, ils ne tournent pas indéfiniment : la réduction continue abouti soit à une solution, soit à une impasse.

C'est véritablement les domaines qui sont le siège de l'algorithme de propagation. A tel point que *Solver* a ajouté la notion de domaine temporaire, représentée par l'attribut **domain-delta** dans les classes IlcIntVar et IlcFloatVar.
En effet, quand la propagation d'un événement est déclenché pour une variable donné, cette variable est mise dans la file d'attente, si elle n'y était pas déjà. De plus, les modifications du domaine d'une variable contrainte sont enregistrées dans ce domaine spécial appelé domain-delta. Ce domaine peut être consulté pendant la propagation des contraintes de la

variable en question. Une fois que toutes les contraintes postée pour cette variable ont été traitées, le domain-delta est alors effacé. Si la variable est de nouveau modifiée, alors ce processus est entièrement répété.

LES CONTRAINTES

Dans la théorie, une contrainte en Programmation Par Contrainte est caractérisée comme une relation mathématique portant sur (et liant) les variables du Problème de Satisfaction de Contraintes.
Pour *Solver*, les contraintes sont aussi des objets au sens C++, comme les variables, que l'on doit créer et auxquels on peut donner un nom. Certaines, les plus courantes, sont déjà définies dans *Solver*. Mais l'utilisateur a la possibilité d'en créer pour satisfaire ses besoins tels qu'ils soient.
Elles peuvent être exploitées par *Solver* pendant la recherche avant instantiation de toutes les variables.

Il existe des **contraintes de classe** qui s'appliquent à des objets d'une classe donnée qui héritent alors des contraintes de cette classe.
Il existe des **méta-contraintes**, qui sont des contraintes portant sur d'autres contraintes.

Telles qu'elles soient, les contraintes possèdent les **fonctions membre** suivantes :

- les constructeurs,
- **`getManager`**`()`, qui permet de récupérer un pointeur sur le manager de la variable,
- `getImpl()`, renvoie l'objet contrainte, son implémentation,
- `setName()` et `getName()`, permet de manipuler le nom, comme pour les variables,
- `setObject()` et `getObject()`, Cf. chapitre sur les variables,
- **`isTrue`**`()` et **`isFalse`**`()`, qui renvoient `IlcTrue` quand la fonction est respectivement vérifiée ou non, quoi qu'il arrive.

On déclare les contraintes sur des variables en créant un objet, puis on l'ajoute à un manager.
Une fois la **contrainte postée** (i.e. on demande à *Solver* de la prendre en compte), *Solver* la respectera pendant la recherche.

Pour poster une contrainte qui traduit que `var1` doit être différente de zéro, dans le manager m, on utilise la fonction membre `IlcManager::add(IlcConstraint)`:

```
m.add( var1 != 0 );
```

On peut également poster des contraintes portant sur plusieurs variables :

```
m.add( var1 + var2 == var3 );
```

Il est également possible de **combiner ces expressions** de manière simple mais puissante avec les **opérateurs logiques classiques**:

```
m.add( (nbOuvriers == nbOutils)
       || (nbPelles > 5) && (nbCasques > 5) );
```

Cette possibilité repose sur le fait qu'une contrainte retourne une valeur (`IlcTrue` ou `IlcFalse`) selon qu'elle est vérifiée ou non. Les opérateurs supportés sont: *non* (**!**), *et* (**&&**), *ou* (**||**), *ou exclusif* (**!=**), *équivalence* (**==**) et *implication* (**<=**).

L'opérateur *implication* (**<=**) peut à première vue paraître étrange, mais il est justifié pour deux raisons:

- tout d'abord il n'a pas d'équivalent typographique en C++,
- mais aussi parce que si l'on considère que `IlcFalse` vaut 0 et `IlcTrue` vaut 1, alors l'expression *Solver* "c1 <= c2" (où c1 et c2 sont des contraintes et "<=" l'opérateur *Solver implication*), est toujours valable si c1 et c2 sont des entiers et "<=" l'opérateur d'inégalité "inférieur ou égal" associé, comme le montre le tableau récapitulatif suivant.

c1	C2	! c1	c1 || c2	c1 && c2	c1 != c2	c1 == c2	c1 <= c2
0	0	1	0	0	0	1	1
0	1	1	1	0	1	0	1
1	0	0	1	0	1	0	0
1	1	0	1	1	0	1	1

Cette façon de poster les contraintes ne permet pas de contrôler le moment où elles sont prises en compte. Pour remédier à cela, il existe deux manières différentes de faire, selon ce que l'on cherche à faire.

Tout d'abord, on peut poster une **contrainte conditionnelle** c2, qui ne sera postée que si le but c1 est vérifié. Par exemple, la contrainte

- SI la charge est supérieure à une tonne (c1),
- ALORS doubler l'épaisseur initiale (c2),

peut se traduire en *Solver*, de la manière suivante:

```
m.add( IlcIfThen( (epaisseur > 1000),
                  (epaisseur == 2 * EPAISSEUR_ZERO) );
```

Cet appel peut néanmoins être évité. En appliquant la formule "**(A ⇒ B) ⇔ (□ || B)**" (où □ signifie "non A"), la précédante instruction devient en effet:

```
m.add( (epaisseur ≤ 1000) || (epaisseur == 2 *
EPAISSEUR_ZERO) );
```

Une autre façon de contrôler le moment où est prise en compte la contrainte, est de l'attacher à l'un des trois événement d'une variable, à savoir:

- `myVar.whenDomain(myContrainte)`,
- `myVar.whenRange(myContrainte)`,
- `myVar.whenValue(myContrainte)`,

qui postent la contrainte `myContrainte` respectivement quand, le domaine, l'intervalle entre les deux bornes du domaine, ou la valeur, sont modifiés.

Une fois ajoutées, on peut également supprimer des contraintes à l'aide de la fonction `IlcManager::remove(IlcConstraint)`. Il faut cependant être prudent.

```
m.add( X != 0 );
m.remove( X != 0 );
// Ne fonctionne pas:
// la contrainte (x != 0) est toujours postée.

IlcConstraint myConstraint( X != 0);
m.add( myConstraint );
m.remove(myConstraint );
// Fonctionne:
// la contrainte (x != 0) n'est plus postée.
```

Afin de ne pas réinventer la roue, et de se simplifier la tache par la même occasion, il faut avoir connaissance de l'**existence de contraintes prédéfinies**.

Par exemple celle qui prend comme paramètre un tableau de variables, et qui impose que toutes ces variables soient différentes : **`IlcAllDiff`**. Si l'on reprend l'exemple `myVarArray` du paragraphe sur les variables, pour spécifier que `var1`, `var2` et `var3` doivent être différentes dans le manager `m`, on écrit :

```
IlcIntVar var3 = 10 * var2 + var1;
IlcIntVarArray myVarArray(m, 3, var1, var2, var3) ;
m.add( IlcAllDiff(myVarArray) );
```

Il y a aussi la fonction **`IlcNullIntersect`**`(myIlcSet1, myIlcSet2)` (où les deux ensembles paramètres sont soit de type `IlcAnySet`, soit de type `IlcAnySetVar`) qui crée et retourne une contrainte qui, une fois postée, impose que ses deux arguments aient une intersection vide dans toute solution générée.

Voilà pour ce qui est des contraintes prédifinies. Mais si l'on a besoin de mettre en place une ou plusieurs contraintes particulières, il est nécessaire d'approfondir la structure de la classe *Solver* `IlcConstraint`.

Une contrainte dans *Solver*, est donc une instance:

- soit de la classe `IlcConstraintI` dans le cas d'une contrainte prédéfinie par *Solver*,
- soit de notre propre classe qui doit alors hériter de `IlcConstraintI`.

Cette dernière possède trois fonctions virtuelles qu'il faut obligatoirement surcharger:

- `propagate`,
- `post`,
- `isViolated`.

La fonction **`propagate`**`()` défini comment les domaines des variables contraintes (paramètres de la contrainte) doivent être modifiées par la contrainte. C'est cette définition qui sera exécutée par l'algorithme de propagation de contraintes. Comme nous l'avons vu dans le chapitre sur les domaines, ces modifications consistent uniquement en la suppression des valeurs (réduction) qui ne satisfont pas la contrainte. Cette fonction est appelée à chaque fois que la condition de `post()` est vérifiée pour effectuer alors le travaille imposé par la contrainte.

Pour pouvoir être utilisée par ultérieurement par l'algorithme de propagation, une contrainte doit être stockée: c'est le rôle de la fonction **`post`**`()`. Elle permet de lier la contrainte aux variables correspondantes. Pour cela, elle doit associer la contrainte aux événements de propagation déclenchés par les expressions contraintes, à l'aide des fonctions membre `IlcIntExp.whenRange()`, `IlcAnyExp.whenDomain()` et `IlcAnyExp.whenValue()`.

Cette fonction n'est appelée qu'une seule fois, au début, et indique à l'algorithme de propagation quand la contrainte doit être exécutée.

Enfin, la dernière mais pas la moindre, est la fonction qui permet de définir sous quelle(s) condition(s) la contrainte est obligatoirement violée: elle est nommée **`isViolated`**`()`. En fait, cette fonction, et la notion associée de contrainte violée, est plus délicate qu'il n'y paraît à première vue. En effet, si cette fonction renvoie `IlcTrue`, alors on est sûre que la contrainte ne peut pas être vérifiée, mais ce n'est pas réversible: il se peut très bien que la contrainte ne soit pas vérifiée et que la fonction retourne `IlcFalse`. Par contre, la fonction ne retournera jamais `IlcTrue` s'il existe une possibilité que la contrainte soit vérifiée. Cette apparente ambiguïté est faite pour les cas où il serait coûteux en temps de calcul de déterminer si oui ou non la contrainte est violée: dans de telles circonstances, la fonction `isViolated()` retourne `IlcFalse`, ce qui est donc le comportement par défaut. Cela induit que l'on est certain du résultat que lorsque la valeur de retour est `IlcTrue`. Bien que cette fonction virtuelle implémente une partie de la sémantique de la contrainte concernée, sa redéfinition n'est pas obligatoire.

Il est à noter pour conclure que l'objet C++ représentant la contrainte possède comme attribut(s) une copie des variables ou valeurs impliquées dans la contrainte (ou arguments de cette contrainte).

Au final, en illustration de ce qui précède, la (re)définition par exemple d'une contrainte imposant que les deux variables (arguments) qu'elle lie soient différentes, pourrait donner cela avec *Solver*.

```
class CMyIlcDiffVariable : public IlcConstraintI
{
public:
    CMyIlcDiffVariable(IlcManager m, IlcIntVar x, IlcIntVar
y);
    virtual ~CMyIlcDiffVariable();

    virtual IlcBool isViolated() const;
    virtual void post();
    virtual void propagate();

private:
    IlcIntVar         myX;
```

```
    IlcIntVar         myY;
};
CMyIlcDiffConstraint::CMyIlcDiffConstraint( IlcManager m,
                                            IlcIntVar x,
                                            IlcIntVar y )
: IlcConstraintI(m), myX(x), myY(y)
{
}
CMyIlcDiffConstraint::~CMyIlcDiffConstraint()
{
}
void CMyIlcDiffConstraint::propagate()
{
    if (myX.isBound())
          myY.removeValue(myX.getyValue());
    if (myY.isBound())
          myX.removeValue(myY.getyValue());
}
void CMyIlcDiffConstraint::post()
{
    myX.whenValue(this);
    myY.whenValue(this);
}
IlcBool CMyIlcDiffConstraint::isViolated() const
{
    return ( myX.isBound()
            && myY.isBound()
            && (myX.getValue() == myY.getValue()) );
}
IlcConstraint IlcDiff(IlcIntVar x, IlcIntVar y)
{
    IlcManager m = x.getManager();
    return new (m.getHeap()) CMyIlcDiffConstraint(m, x, y);
}
```

LA RECHERCHE

De manière standard, *Solver*,

- propage les contraintes une à une,
- et à chaque fois réduit si possible les domaines des variables contraintes,
- puis essaye de propager cette réduction aux autres domaines,
- puis passe à la contrainte suivante.

Il propage la **réduction de domaine**., en fonctionnant selon le principe du *Branch And Bound* (*User's Manual*, p 131).

Solver fourni également la possibilité de définir des algorithmes de recherche **non déterministes** (i. e. la séquence des futures actions de résolution n'est pas connu à l'avance), en déclarant des buts, comme instance de la classe `IlcGoal`. En fait, plus que des buts, cela correspond plus exactement à des STRATEGIES de recherche.

La stratégie de recherche par réduction de domaine, vue ci dessus, **est déjà prédéfinie** dans *Solver* et implémenter dans le but `IlcGenerate` et `IlcBestGenerate`.
Cette fonction prend comme argument un tableau de variables dont *Solver* doit trouver des valeurs qui satisfassent aux contraintes du problème.
Pour reprendre notre exemple des trois variables, si l'objectif est de trouver des valeurs pour ces trois variables, on écrira :

```
m.add( IlcGenerate(myVarArray) );
```

En effet, pour que cette stratégie soit prise en compte par *Solver*, il faut l'ajouter au manager.

Tout comme les variables et les contraintes, il est possible de **combiner les buts**, et d'en ajouter ou supprimer au cours de la recherche, grâce notament aux fonctions `IlcOr` et `IlcAnd`.

Une fois exprimée la stratégie de recherche il ne reste plus qu'à indiquer à *Solver* que l'on souhaite obtenir une solution, ce qui se fait au moyen de la fonction membre `IlcManager::nextSolution`.

Lorsque le problème est sous-contraint (cad que les contraintes sont peu nombreuses ou faibles), le problème peut avoir plusieurs solutions valides. Il est alors intéressant d'ajouter une contrainte qui correspond en quelque sorte à un critère de qualité des différentes solutions, et que l'on appelle **fonction de objectif**. Par défaut, le comportement de *Solver* concernant cette fonction objectif est de minimiser son paramètre: pour le maximiser, il suffit de le faire précéder d'un signe "-".
Par exemple, si nos trois variables précédentes sont les durées respectives des trois phases d'un processus de production, il peut être intéressant de pouvoir spécifier qu'on aimerait que

ces durées soient minimales, ce qui se traduit par le fait que la somme des trois variable est minimum.

Concrètement, il faut indiquer à *Solver*, dans l'ordre :

- les variables intéressantes dont on veut récupérer les valeurs, ici nos trois variables,
- indiquer le critère à minimiser, la somme des variables ici,
- appeler la fonction jusqu'à ce qu'il n'y plus de solution. La dernière générée sera forcement la meilleure, puisque la fonction objectif indique le critère qui doit être amélioré à chaque fois.

Voilà donc ce que cela pourrait donner en *Solver* :

```
IlcIntVarArray allVar(m, 3, var1, var2, var3);
IlcIntVar sumVar = var1 + var2 + var3;
m.add(IlcGenerate(allVars));
for (IlcInt i=0 ; i < allVars.getSize() ; i++)
{
    allVars[i].setStorable();
}
m.setObjMin(sumVar);
while (m.nextSolution())
{
    m.out() << costVar << endl;
    m.storeSolution();
}
m.restart();
m.nextSolution();
```

Voilà pour ce qui est de l'utilisation standard de *Solver*. Mais pour adapter la résolution à sa propre problématique en vue de l'optimiser, comme cela a été le cas pour mon stage, il est nécessaire de bien distinguer les trois phases successives de tout moteur de PPC (et de voir comment *Solver* les gère), à savoir dans l'ordre :

- le choix d'une variable à instancier,
- le choix d'une valeur à affecter à cette variable,
- et la **propagation des contraintes** faisant intervenir cette variable.

Pour cela, *Solver* permet à l'utilisateur :

- de choisir une stratégie pour chacun des deux choix précédant, parmi celles déjà implémentées,
- et même de définir entièrement ses propres stratégies en fonction de ses besoins.

Pour ce qui est de la première possibilité, *Solver* a en effet déjà défini les stratégies les plus répandues. Pour le choix de la variable à instancier, il suffit de signaler celle que l'on désir lors de l'appel de la fonction `IlcGenerate` :

```
m.add(IlcGenerate(myVarArray, IlcChooseXXX));
```

où `IlcChooseXXX` peut prendre l'une des valeurs suivantes :

- `IlcChooseMinSizeInt` : (*) la plus petite cardinalité (taille),
- `IlcChooseMaxSizeInt` : (*) la plus grande cardinalité (taille),
- `IlcChooseMinMinSizeInt` : (*) la plus petite borne inférieure,
- `IlcChooseMaxMinSizeInt` : (*) la plus grande borne inférieure,
- `IlcChooseMinMaxSizeInt` : (*) la plus petite borne supérieur,
- `IlcChooseMaxMaxSizeInt` : (*) la plus grande borne supérieur.

(*) = choix en premier de la variable dont le domaine possède

Pour ce qui est de chois de la valeur, ou de la définition personnelle de stratégies, il faut définir son propre goal *Solver* (`ILCGOAL`). On pourra alors utiliser les fonctions membre des variables *Solver* pour contrôler le choix de valeur.
Lorsque la variable n'est pas instantiée, on prend une valeur du domaine et on tente de la lui affecter. Il faut alors une structure capable de prendre en compte l'éventualité où cette instantiation soit invalide : c'est ce que *Solver* appelle **un point de choix**, et se traduit par la fonction `IlcOr`. Son fonctionnement est le suivant :

- (L1) enregistrement de l'état courant,
- (L2) i = 1
- (L3) exécute le i-ème sous-goal
- (L4) si succès, alors on sort,
- (L5) sinon on rétabli l'état, i++ et retourne à (L3) : on ***backtrack***.

On appelle sous-goal, les paramètres de la fonction IlcOr. Le nombre maximum de paramètres pour cette fonction étant de cinq, il suffit d'imbriquer cette fonction pour passer outre cette restriction.
On peut ajouter un argument facultatif, qui correspond à l'indice du point de choix que l'on considère. Ainsi, dans l'implémentation d'un goal, si l'on souhaite générer un *backtrack*, on peut choisir le niveau de celui-ci en indiquant son indice dans l'appelle à la fonction `IlcManager::fail(IlcInt indice)`. Par défaut on remonte au dernier.

```
ILCGOAL1(MyIlcInstantiate, IlcIntVar, theVar)
{
   if (var.isBound())
      return 0;
   IlcInt val = var.getMin();
   return IlcOr(var == val, IlcAnd(var != val, this));
}
```

```
ILCGOAL1(MyIlcGenerate, IlcIntVarArray, theVarArray)
{
   IlcInt index = chooseIndex(theVarArray);
   If (index == -1)
      return 0;
   return IlcAnd(MyIlcInstantiate(getManager(),
                                  theVarArray [index]) ,
                 this);
}

m.add(MyIlcGenerate(myVarArray));
m.nextSolution();
```

où `ILCGOAL` est suivi directement du nombre de paramètres du goal qu'on est en train de définir, de son nom (qui n'entre pas dans le nombre de paramètres), puis de couples de la forme « type du paramètre i, nom du paramètre i ».

Voilà pour ce qui est de la théorie, mais en pratique *Solver* ajoute une subtilité : dés que la propagation des contraintes aboutie à ce que le domaine d'une variable ne contienne plus qu'une seule valeur, alors *Solver* passe outre les stratégies de choix éventuellement défini par l'utilisateur, pour prendre d'abord cette variable et lui affecter l'unique valeur du domaine. Ce n'est qu'après qu'il respecte les stratégies indiquées.
Cela donne l'algorithme suivant :

- tant qu'il y a une variable non instantiée,
  - s'il existe une variable non instantiée telle que le domaine ne possède plus qu'une seule valeur,
    - alors choix de cette variable,
    - sinon choix d'une variable selon la stratégie de l'utilisateur,
  - choix d'une valeur pour la variable choisie,
  - propagation des contraintes faisant intervenir la variable choisie,

A tout moment, si un échec surgit, la fonction `IlcManager::fail` est appelée.

`IlcManager::restart`
`IlcManager::IlcBestGenerate`.

L'AFFICHAGE

Utiliser la fonction membre `IlcManager::out`

La fonction membre `IlcManager::printInformation` est à utiliser avec les intructions

`m.openLogFile("sendmory.log");`

en début de programme, et

`m.closeLogFile();`

en fin de programme

UN EXEMPLE POUR COMPRENDRE

```
// Problem Description
// -------------------
// Solve the cryptarithm:
//               S E N D              9 5 6 7
//             + M O R E    =>      +   1 0 8 5
//             ---------            -----------
//             M O N E Y            = 1 0 6 5 2

#include <ilsolver/ilcint.h>

ILCDEMON1(PrintVar, IlcIntVar, var)
{
      IlcManager m = var.getManager();
      m.out() << var << endl;
}

int main()
{
      IlcManager m(IlcEdit);

      #if defined(ILCLOGFILE)
            m.openLogFile("sendmory.log");
      #endif

      //$doc:DECL1
      IlcIntVar S(m, 0, 9), E(m, 0, 9), N(m, 0, 9), D(m, 0, 9),
                M(m, 0, 9), O(m, 0, 9), R(m, 0, 9), Y(m, 0, 9);
      //end:DECL1

      //$doc:CON1
      m.add( S != 0 );
```

```
        m.add( M != 0 );
        //end:CON1

        //$doc:DECL2
        IlcIntVar send =            1000*S + 100*E + 10*N + D;
        IlcIntVar more =            1000*M + 100*O + 10*R + E;
        IlcIntVar money = 10000*M + 1000*O + 100*N + 10*E + Y;
        //end:DECL2

        //$doc:CON2
        m.add( send + more == money );
        //end:CON2

        //$doc:ARR
        IlcIntVarArray letters(m, 8, S, E, N, D, M, O, R, Y);
        m.add(IlcAllDiff(letters));
        //end:ARR

        //$doc:SEARCH
        m.add(IlcGenerate(letters));
        m.nextSolution();
        //end:SEARCH

        //$doc:PRINT
        m.out() << "    " << send.getValue() << endl
                << " +  " << more.getValue() << endl
                << " = " << money.getValue() << endl;
        //end:PRINT

        m.printInformation();

        #if defined(ILCLOGFILE)
                m.closeLogFile();
        #endif

        m.end();
        return 0;
}
```

Implémentation présentée dans la littérature

STRUCTURE DE DONNEES

ALGORITHME

Algorithme: My_Algo(**<Å Ơ ℒ ℬ>**, **agenda**, **Δ**)

1. Terminaison:

Si (**agenda** est vide) alors

return(**<Å Ơ ℒ ℬ>**);

2. Selection du but:

Retirer un but (**Q , A_C**) de **agenda**

(a) Si (**Q** est sentance quantifiée) alors

insérer la base universelle Y(**Q, A_C**) dans **agenda**

aller_a **3.**

(b) Si (**Q** est conjonction de **Q_i**) alors

insérer chaque (**Q_i , A_C**) dans **agenda**

aller_a **3.**

(c) Si (**Q** est disjonction de **Q_i**) alors

choisir un **Q_k** parmi les **Q_i**

insérer (**Q_k , A_C**) dans **agenda**

aller_a **3.**

(d) Si (**Q** est litteral) et ((non(**Q**) $\rightarrow$ **A_C**) existe dans **ℒ**) alors

return(echec)

3. Selection de l'action:

Soit **A_P** = choix (non deterministe) d'une action

(appartenant :

- soit a **Å**, deja instanciée,
- soit a **Δ**, nouvellement instanciée)

ayant un effet (ou consequent) R, qui verifient :

(a) **(A_P < A_C)** consistant avec **Ơ**

(b) **R** et **Q** unifient avec **ℬ**

Si (**A_P** est vide) alors

return(echec)

sinon

(1) Soit **ℒ'** = **ℒ** $\cup$ **($A_P \rightarrow Q \rightarrow A_C$)**

(2) Soit $\mathcal{B}' = \mathcal{B} \cup ((u, v) \in \mathbf{MGU(Q, R, \mathcal{B})})$

(3) Soit $\mathcal{O}' = \mathcal{O} \cup \{(A_P < A_C)\}$

4. Considerer de nouveaux actions et effets:

Si ($A_P \in \mathring{A}$) alors

$\mathring{A}' = \mathring{A}$

$\mathcal{B}' = \mathcal{B}$

$\mathcal{O}' = \mathcal{O}$

agenda' = **agenda**

sinon

$\mathring{A}' = \mathring{A} \cup \{ A_P \}$

$\mathcal{O}' = \mathcal{O} \cup \{ (A_0 < A_P < A_\infty) \}$

$\mathcal{B}' = \mathcal{B} \cup (A_P$.NonCodesignation **)**

Si (A_P.Effect est Conditionnel) alors

agenda' = **agenda** $\oplus$ (A_P.Effect.Antecedant; A_P)

sinon

agenda' = **agenda** $\oplus$ (A_P.Precondition/**MGU(Q, R, $\mathcal{B}$)**; A_P))

5. Protection des liens causals:

Pour tout lien causal $\ell = (A_i \text{–}Q\rightarrow A_j)$ de $\mathcal{L}$ faire

Pour toute action A_M menacant ℓ faire

Choisir une contrainte d'ordonnancement consistante parmi:

1. **Demotion:** ajouter ($A_M < A_i$) a $\mathcal{O}'$ ou,
2. **Promotion:** ajouter ($A_j < A_M$) a $\mathcal{O}'$,
3. **Confrontation:**

Si (A_M.Effect est Conditionnel) alors

agenda' = **agenda'** $\oplus\{$**(**non(.Antecedant)\MGU(**P,** non(.Consequent))**);** $A_M)\}$

4. Sinon return(echec)

6. Appel recursif:

Si ($\mathcal{B}'$ est inconsistant) alors

return(echec)

sinon

My_Algo (**<Å′ Ơ′ ℒ′ ℬ′>, agenda′, Δ)**

COMMENTAIRES

0.

<Å Ơ ℒ ℬ>$_0$ = **PLAN_NUL** = < { A_i, A_∞ } { (A_i < A_∞) } {} {} >

agenda$_0$ = { (A_∞.PreCondition(i), A_∞)$_i$ }, où A_∞.PreCondition(i) représentent toutes les pré conditions de l'action A_∞.

Générer les actions **START** et **END** comme suit :
START n'a aucune pré conditions, et ses effets sont les propositions vraies dans l'état initial (importance ici de l'hypothèse des *Mondes Clos*)
END , à l'inverse, n'a aucun effet, et ses pré conditions sont les buts à atteindre par le planificateur.

1.

Si **agenda** est vide, tous les buts sont réalisés,
On retourne donc le plan courant.

2.

On extrait un (sous) but **Q** parmi ceux encore à atteindre (= **agenda**) : **Q** → **A**$_C$.
Pour la suite, on assimile **Q**.Precondition et **Q**.Effect.Antecedent, qu'on note « →**Q** »

-(a)-

Si →**Q** est définie à l'aide d'un opérateur, on « développe » alors →**Q**, et on ajoute le résultat à **agenda**

-(b)-

Si →**Q** est une intersection de propositions, alors on ajoute chacune des propositions dans **agenda**

-(c)-

Si →**Q** est une union de propositions, alors on en choisi une que l'on ajoute dans **agenda**

-(d)-

Si non(→**Q**) est une précondition déjà nécessaire (impliquée dans un lien causal), alors échec.

3.

Une fois que l'on a déterminé le (sous) but **Q** que l'on veut accomplir, il faut choisir une action **A**$_P$ dont l'effet (ou le consequent dans le cas d'un effet conditionnel) **R** est similaire :
A$_P$ tel que (**A**$_P$ → **R** et **R ≡ G**).
La difficulté vient des schéma d'action : **Q** et/ou **R** peuvent en effet être paramétrés. D'où l'utilisation de la fonction **MGU** (*Most General Unifier*) qui considérant en plus les contraintes de dépendance, retourne un ensemble de codesignation permettant de réduire au maximum les domaines des variables de **R** de manière à l'unifier le plus possible avec **Q**.

On ajoute alors :

(1) Le nouveau lien causal ($\mathbf{A_P}$, $\mathbf{A_C}$, **Q**) ainsi obtenu à $\mathcal{L}$,

(2) Chacune des *codesignation* à $\mathcal{B}$,

(3) La nouvelle contrainte d'ordonnancement ($\mathbf{A_P} < \mathbf{A_C}$) à $\mathcal{O}$.

Le (1) permet de conserver la nouvelle action à l'état de schéma (flexibilité maximale), tout en gardant ses valeurs possibles pour ne pas avoir par la suite à considérer que celles la, et pour garantir l'unicité de la solution finale.

4.

On ajoute les `NonCodesignation` d'une action ou les `antecedant` d'un effet conditionnel à $\mathcal{B}$, dans le cas où il ne sont pas déjà impliqués dans un lien causal.

Si $\mathbf{A_P}$`.PreCondition`

- est Logique -$\xrightarrow{\text{AddTo}}$ **agenda**
- est NonCodesignation -$\xrightarrow{\text{AddTo}}$ $\mathcal{B}$

Dans le cas où l'effet est conditionnel, on ajoute non pas la `precondition` de l'action, mais l'`antecedent` de cet effet.

5.

Dans le cas où il y a possibilité de Menace, se contenter d'ajouter si possible une des deux contraintes d'ordonnancement possibles (*demotion* ou *promotion*), mais sans unifier (c'est à dire sans appliquer les `codesignation`): attendre le plus tard possible, que cela devienne inévitable.

Dans le cas d'un effet conditionnel, on peut enfin essayer la *promotion* :

`(3.) =`

Si un `Effect.Consequent` est menaçant on ajoute `non(Effect.Antecedant)` aux buts à atteindre, en espérant ainsi faire disparaître l'`Effect.Consequent`

`(4.) =`

Aucune contrainte n'est consistante

6.

Avant de poursuivre, on teste si l'ensemble des contraintes de dépendance sont bien consistantes entre elles.

Algorithme effectivement implémenté

STRUCTURE DE DONNEES

Les données sont implémentées comme suit.

J'ai tout d'abord représenté **les propositions** sous la forme d'un tableau d'entiers dont chaque case contient une valeur, à savoir l'indice de l'action qui soutient cette proposition, sachant qu'en plus des actions possibles, il existe deux autres valeurs qui correspondent aux notions de **no-op** (persistance d'une proposition) et de **non-activation** (la proposition concernée n'est pas active à cet instant). De plus, la notion d'**action inverse** (`move_A_from_B_to_C` et `move_A_from_C_to_B`) est prise en charge, ce qui est synonyme de gain de temps pendant la résolution, où l'on peut accéder de façon simple et rapide à l'inverse d'une action, pour l'interdire au niveau suivant et ainsi éviter de revenir en arrière, voire de boucler.

Autant les actions sont donc indexées de –1 à « `nombre_total_d_actions` – 1 », autant les propositions sont en fait indexées de 1 à « **2** `* nombre_total_de_propositions` », la deuxième moitié correspondant aux négations de propositions, permettant ainsi de prendre en compte dans un même niveau la coexistence d'une proposition A et de son opposé non(A).

J'ai de plus crée une classe `CProposition` et une `CAction`.

La première, contient notamment comme attribut :

- `name`,
- `isActivate`,
- `isEnable`,
- `mutexArray`,
- et `support`.

La seconde elle, possède les attributs suivants :

- `name`,
- `preCondList`,
- et `postCondList`,

chacun d'entre eux étant un tableau de taille « **2** `* nombre_total_de_propositions` ». En effet, comme nous l'avons vu dans la partie bibliographique, une action possède une liste des propositions qu'elle fait apparaître (`add_list`) et une liste des propositions qui étaient

présentent dans ses pré-conditions, mais ont disparus des effets, du niveau suivant (`del_list`). Or, comme les actions sont tout d'abord considérées dans le sens avant pour l'extension de graphe, puis dans le sens arrière pour l'extraction de solution, il aurait été nécessaire de dupliquer cette structure. C'est pour remédier à cela que nous avons donc ajouté aux propositions du problème leurs négations.

FONCTIONNEMENT DE L'ALGORITHME

De plus, point intéressant, ces deux listes sont automatiquement initialisées des bonnes valeurs (propositions) au début de l'algorithme par une **analyse de type simplifiée**. En effet, d'après le nom de l'action, l'algorithme déduit les pré-conditions et post-conditions ce cette action puis l'ajoute dans les domaines de ses pré-conditions. Il est vrai que cela repose actuellement simplement sur le parsage d'une chaîne de caractère, mais néanmoins cette fonction existe, et pourra servir de support à une version plus évoluée qui pourra par exemple reposé sur la notion de schéma d'actions.

Enfin, lorsque des Mutex sont calculés dans la phase d'extension de graphe, il est important que ce temps de calcul soit exploité, en le propageant dans l'algorithme de résolution de *Solver*, afin qu'il ne réitère pas inutilement des actions équivalentes. Pour cela, il faut convertir ces Mutex en une réduction de domaine préalable à la résolution par *Solver*. Cela peut se faire par :

- l'ajout de contraintes interdisant pour les variables concernées les valeurs correspondantes,
- ou directement créer les variables Solver avec des domaines déjà réduits des valeurs (actions) qui provoquent des Mutex.

Les autres fonctionnalités de l'algorithme sont détaillées et expliquées dans le paragraphe « *Ce qui a été amélioré par rapport à l'existent* », dans le chapitre « *Discussion* ».

RESULTATS

Les résultats sont ceux obtenus sur l'exemple du « Monde des Cubes ».

Discussion

CE QUI ETAIT PREVU MAIS QUI N'EST PAS FAIT…

Après l'étude bibliographique que j'ai menée, et les conclusions portant notamment sur les travaux de *Kambhampati* [Kamb00], il a été décidé que j'utiliserai la programmation par contraintes pour implémenter l'algorithme. Cela a impliqué de connaître le fonctionnement d'un moteur de propagation de contraintes, et donc mon auto-formation concernant *ILOG Solver*. Je pensais au départ qu'il serait suffisant d'appréhender les fonctionnalités de base, mais afin d'optimiser l'algorithme et de mettre en pratique les travaux précédemment étudiés, il a été nécessaire d'approfondir ma connaissance de *Solver*.
Cela c'est fait au détriment de certaines fonctionnalités spécifiées au départ du stage. Ces **fonctionnalités non implémentées** sont les suivantes…

Tout d'abord, ayant pris au départ le « Monde des Cubes » comme exemple, pour l'implémentation et les tests successifs de l'algorithme, je suis resté sur une structure simplifiée mais adaptée à la représentation des actions. Ainsi, l'algorithme ne prend pas en compte les **schémas d'actions** ou actions génériques.

Cette fonctionnalité étant nécessaire pour pouvoir implémenter les **effets conditionnels** et la **quantification universelle**, ces deux derniers sont également non implémentés.

CE QUI A ETE AMELIORE PAR RAPPORT A L'EXISTANT

En contrepartie, le temps passé sur *ILOG Solver*, tant au niveau technique (fonctionnalités intrinsèques) que sur le plan théorique (rappels de Programmation Par Contraintes notamment), m'a permis d'**apporter deux améliorations** notables.

La première est d'ordre structurel. Dans tous les rapports que j'ai lus, les propositions étaient stockées dans des tables de hachage. Ainsi, lorsque l'on voulait y accéder, on devait le faire à partir de la fonction de hachage associée, avec les coûts que cela implique.
En ce qui me concerne, je suis revenu au simple **tableau d'entiers**. Chacune des propositions possibles étant toutes associées à une valeur entière, toutes distinctes: la i-ème

case du tableau contient donc la valeur entière de la proposition d'indice i. Cette valeur entière correspond de même à l'indice d'une action, chacune des actions possibles ayant également un indice, tous différents.
Outre les gains de temps obtenus par l'accès direct aux variables et valeurs associées, il est aisé de transférer cette structure dans le monde *Solver*, qui contient par exemple les types `IlcInt`, `IlcIntArray`, `IlcIntVar` et `IlcIntVarArray`, dont les noms parlent d'eux-mêmes.

La seconde amélioration est d'ordre fonctionnelle, et repose sur la **notion de Mutex**.
Rappelons tout d'abord qu'il existe des Mutex dits permanents (par exemple les propositions « A » et « non(A) » seront toujours Mutex) et d'autres temporaires. De plus, les Mutex temporaires sur les propositions d'un niveau k du graphe de plan, sont obtenus à partir de ceux portant sur les actions qui génèrent ce niveau, eux-mêmes obtenus de trois façons différentes, mais toujours à partir des Mutex existants entre des propositions du niveau k-1 et/ou k.
Or, deux PROPOSITIONS sont Mutex si TOUTES **[i]** les actions du domaine de la première sont Mutex avec TOUTES les actions du domaine de la seconde. Au contraire, deux ACTIONS sont Mutex (de type « effets inconsistants ») si AU MOINS **[ii]** une de ses pré-conditions (ou post-conditions, même remarque…) de la première est Mutex avec AU MOINS une de celles du domaine de la seconde.
A partir de là, il peut être efficient de remarquer **que le nombre de Mutex temporaires est d'autant plus faible**:

- que le nombre d'actions (valeurs) dans les domaines des propositions (variables) est élevé (Cf. **[i]**),
- et que pour chacune de ces actions le nombre de pré-conditions ou post-conditions…) est lui restreint (Cf. **[ii]**).

Or *Graphplan* procède en deux étapes successives à savoir :

- l'extension de graphe,
- puis l'extraction de la solution
    - si le dernier niveau généré lors de la première étape contient au moins tous les buts
    - et que ces derniers ne sont pas Mutex entre eux.

Pour savoir si cette deuxième condition est vérifiée, l'algorithme vérifie tout d'abord s'il n'y a pas de Mutex permanent (ce qui peut être vérifié dès le départ pour savoir si les buts à atteindre sont compatibles entre eux), et si c'est le cas, il calcule alors tous les éventuels Mutex temporaires. Cette perte de temps, croissante de manière exponentielle avec le nombre de variables et la taille de leurs domaines, peut parfois être évité en se basant sur la précédente remarque.
L'intérêt de cette remarque est d'autant plus grand que la taille des domaines des variables est élevé et que le nombre de ces variables est également grand : la probabilité

d'existence de Mutex temporaire est plus faible, et donc la possibilité de supprimer des temps de calcul croissants est élevée.

Dans le « Monde des Cubes » par exemple, il s'avère, en raison de la taille relativement importante des domaines des propositions et le faible nombre de pré-conditions pour chaque action, qu'il n'apparaît jamais de Mutex de type « effets inconsistants ».

De plus, **la construction des Mutex dans *Graphplan* a un comportement monotone**. Cela signifie que deux propositions qui ne sont pas (plus) Mutex à un niveau donné ne pourront jamais le devenir dans un niveau ultérieur. A partir de la, à chaque niveau du graphe de plan, j'ai mémorisé pour chaque proposition si elle présentait un Mutex non persistent avec chacune des autres. Et d'après la monotonie, pour toute proposition, il n'est nécessaire de recalculer l'existence éventuelle d'un Mutex non persistent avec une seconde proposition, que si ces deux propositions étaient déjà Mutex au niveau précédant, dans le cas contraire (pas de Mutex), on est certain qu'elles demeurent non-Mutex.

CE QU'IL RESTE A CREUSER

Une des fonctionnalités que l'on pourrait ajouter serait un interpréteur du type *Lex & Yacc*, qui permettrait une plus grande souplesse dans la préhension des entrés.

Une autre voie envisageable, serait l'ajout d'un compilateur *SAT* qui transformerait les entrées en forme normale conjonctive, comme le souligne *Weld* dans [Weld98].

Mais si le but final est d'obtenir un planificateur performant, et non pas d'améliorer les performances de l'existant, notamment *Graphplan*, la solution serait peut être de « *think different ».*

J'implore par avance l'indulgence de M. Adjaoute, pour la vulgarisation qui va suivre et pour la définition que j'y donne des agents… Je me contente juste de reprendre la nomenclature et les grandes lignes de l'article pour présenter ce qui me semble être un axe de recherche intéressant et prometteur. N'ayant pas eu le temps durant mon stage d'étudier cette possibilité, mais l'étude faite en cours m'ayant laissé entrevoir la complexité mais aussi les possibilités des systèmes multi-agents, j'aimerais simplement reprendre les grandes lignes d'un article paru dans le numéro de Mai 2000 de *Pour La Science* [BoTh00], intitulé « Intelligence Collective ».

A en juger par les nombreux travaux portant dessus, et notamment ceux de M. Pollack et de N. Nilson (qui a obtenu des résultats très intéressants avec un groupe de robots dotés de fonctions simples), les agents autonomes nous promettent de belles surprises…

La problématique est la suivante. Un groupe d'individus autonomes, à savoir non supervisés par une même structure globale doit faire de nombreux allés/retours entre une origine et une destination (la supervisation entraînant, par un phénomène d'étranglement, une perte d'efficacité croissante avec le nombre d'individus dans le groupe et leur diversité…). Pour

cela, les individus ont le choix entre deux chemins distincts, l'un étant deux fois plus long que le premier. Comment faire pour que la majorité prenne rapidement le plus court chemin ?

Les entomologistes se sont aperçus que certains insectes qui vivent en colonie (pour ne pas citer les fourmis ! ! !), résolvaient le problème par le dépôt d'un marqueur (phéromones) sur le parcours emprunté. Les individus ayant emprunté le plus court chemin à l'aller, et au retour, seront les premiers revenus, ce qui augmentera la densité de marqueur sur ce parcours, favorable.

Cette remarque est remise en question lorsque le chemin le plus long est ouvert en premier. La solution est alors de conférer un pouvoir « volatile » au marqueur. L'équilibre entre le dépôt et l'évaporation est important, puisque c'est lui qui va déterminer le caractère intéressant d'un chemin. C'est donc de lui que va dépendre le comportement de tout le groupe.

Une variante de cette problématique est le cas où il y a n destination à visiter, d'intérêt décroissant avec le temps de manière hétérogène.

De la même façon que précédemment, les destinations les plus proches seront d'abord visitées, puis lorsque leur intérêt sera devenu trop faible, voire nul, ce sera le tour des plus éloignées.

Le problème du voyageur de commerce peut alors se résoudre ainsi :

- on lâche des individus qui vont au hasard de ville en ville,
- dès qu'il y en a un qui a visité toutes les villes, il fait demi-tour,
- en déposant un marqueur le long de leurs parcours respectifs,
- en quantité inversement proportionnelle à la distance parcourue,
- et qui s'évapore.
- Une fois que tous les individus sont de retour, on les relâche à nouveau…

En un endroit donné du parcours, la densité de marqueur est donc :

- la somme qu'a déposée chaque individu qui est passé en cet endroit,
- moins ce qui s'est évaporé depuis le début.

On peut ainsi déterminer avec ce type de raisonnement,

- à la fois le chemin le plus court dans des problèmes statiques où les destinations et leurs intérêts respectifs sont fixes,
- et aussi détecter un chemin de remplacement dans le cas de problèmes dynamiques, où l'intérêt des sources varie.

Quel est le lien avec la planification me demanderez-vous ?

Et bien si l'on remplaçait la carte routière du voyageur de commerce par un graphe de plan, ou une ville par un état du système à un instant donné ou par une proposition ?… Dans ce dernier cas, on pourrait alors passer outre les hypothèses initiales de *Graphplan* sur l'omniscience du planificateur et les causes de changement de bas niveau, stipulant qu'il connaît tous les états possibles du système et qu'il est le seul à pouvoir les modifier. Le planificateur peut donc prendre en compte les événements extérieurs. Un tel événement se

traduit par l'ajout (suppression) de propositions, ce qui correspond à un intérêt soudainement accru (nul).

De plus, le caractère autonome des individus, quel que soit son degré, permet d'appréhender des plans d'actions non parallèles, par opposition aux algorithmes supervisés ou déterministes.

Par exemple, les robots de Nilson son capables :

- de changer de trajectoire quand ils rencontrent un autre robot, qui constitue un obstacle imprévu,
- de suivre plusieurs buts simultanément, décalant d'un but à l'autre si leurs importances changent.

Ce rapprochement entre les agents et la planification est certes simplifié, mais la description qui le précède l'est tout autant. En effet, cette dernière considère que les « agents » sont passifs (déposent un marqueur et empruntent le chemin le plus marqué), et ne possèdent pas de comportement propre. Il serait en effet intéressant de leur ajouter les notions :

- de priorité/hiérarchie,
- d'humeur changeante,
- de durée de vie,
- de fiabilité/honnêteté
- ou encore d'affinités respectives …

Les systèmes multi-agents apparaissent donc comme un moyen de concevoir des systèmes capables de s'adapter rapidement à des conditions changeantes. Cependant cette flexibilité, qui fait défaut aux algorithmes classiques, sera directement fonction de la pertinence des connexions entre les agents. Cette simplification de l'individu, associés en nombre, pour résoudre des taches toujours plus complexes n'est qu'une métaphore de la citation de *Confucius* « Le tout est plus grand que la somme des parties » qui parlait déjà d'émergence…

VI.

Conclusion Générale

Après avoir étudié certains rapports sur la planification et les dernières avancées dans ce domaine, j'ai établi les bases théoriques d'un algorithme de planification et une structure de donnée adaptée.

J'ai ensuite exploré la voie des algorithmes de recherche locale (Recuit Simulé et Recherche Tabou) pour finalement conclure ,en accord avec mon maître de stage, que je prendrais la direction de la Programmation Par Contraintes. C'est donc en m'appuyant sur les travaux de *Kambhampati* [Kamb00] notamment, que j'ai implémenté un algorithme en Programmation Par Contraintes sous *Microsoft Visual C++ (*avec une approche des *MFC)* et *ILOG Solver*.

L'algorithme opère comme suit.

- Il prend l'ensemble des propositions qui définissent l'état courant (au départ il s'agit de l'état initial), lui applique toutes les actions qui peuvent l'être, afin d'obtenir un nouvel état, qui peut éventuellement contenir des propositions contradictoires : c'est l'extension de graphe.
- Si ce nouvel état ne contient pas au moins tous les buts finaux, l'algorithme recommence ce processus. Sinon, il créé une variable *Solver* pour chaque proposition de chaque état généré, et les contraintes *Solver* à partir des contraintes du problème, puis il lance la génération de solution de *Solver*, qui cherche une action pour soutenir chaque proposition : c'est l'extraction de la solution.
- Si sa résolution aboutie, il a alors une action pour chaque proposition de chaque niveau, ce qui constitue un plan d'actions solution, sinon il recommence le processus.

Il s'avère au final, que mon algorithme sera terminé pour la soutenance.

Les améliorations à apporter sont de plusieurs ordres :

- celles prévues mais pas encore implémentés,
- celles non prévues au départ, mais intéressantes,
    - sur le plan technique, comme l'ajout d'un interpréteur comme *Lex & Yacc*, pour plus de souplesse au niveau de la syntaxe des entrées,
    - ou sur le plan tactique. En effet, en partant de l'article [BoTh00], et en m'appuyant sur les cours de cette année, il paraît fort intéressant d'explorer la voie des systèmes multi-agents pour envisager une solution au problème de la génération de plans d'actions.

Sur le plan personnel, le point qu'il me faudrait améliorer est la prise en compte d'une méthodologie dans mon travail, afin de structurer ce dernier pour m'éviter de me disperser ou de négliger certains aspects.

Cependant, ce stage m'a beaucoup apporté, que ce soit :

- sur le plan **théorique**, avec l'apprentissage continu lié à la partie bibliographique, que ce soit dans la recherche sur Internet ou l'étude elle-même,
- sur le plan **technique**, avec l'apprentissage de la librairie *Microsofts Foundation Class (MFC)* et des logiciels *Rational Rose 98*, *Microsoft Visual C++ 6.0* et *ILOG Solver 4.31* (… ainsi que Word et Powerpoint !),
- sur le plan **relationnel**, puisque lors de la phase initiale de recherche bibliographique j'ai été en contact avec chercheurs et universitaires, américains notamment, et lors de l'implémentation, j'ai également pris contact avec des ingénieurs, de Ilog France par exemple,
- et enfin sur le plan **professionnel**, avec la nécessité de me prendre en charge et de faire preuve d'autonomie, prendre des initiatives, et notamment de demander quand cela est nécessaire.

De plus, ce stage m'a offert d'intéressantes perspectives, à savoir :

- l'élaboration d'un rapport en vue de le présenter à la conférence *« The 19th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group* », qui se déroulera les 14 et 15 décembre 2000 à *The Open University, Milton Keynes, Buckinghamshire (UK)*. Ce qui implique des résultats de qualité, et un rapport soumis avant le 17 septembre 2000,
- et enfin une proposition d'embauche au sein de Pacte Novation.

VII.

Acteurs & Bibliographie

Principaux acteurs

En plus de celles détaillées ci-après, vous pourrez trouvez des références à de nombreux acteurs de la planification, surtout, aux deux adresses suivants :

- http://www.cs.yale.edu/people/faculty.html
- http://www.cs.ucl.ac.uk/intelligent_systems/past_projects/agents/

BLUM Avrim L.

Situation : *School of Computer Science Carnegie Mellon University, Pittsburgh*
Mail : avrim@cs.cmu.edu
Publications : [BlFu97]
Travaux :

Il a travaillé notamment sur Graphplan avec FURST pour le compte de *Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF,* et de l'*Advanced Research Projects Agency (ARPA)*. Il était soutenu par *NFS National Young Investigator* (CCR-9357793) et une *Sloan Foundation Research Fellowship.*

FURST Merrick L.

Situation : *School of Computer Science Carnegie Mellon University, Pittsburgh*
Mail : mfx@cs.cmu.edu
Publications : [BlFu97]
Travaux :

Il a travaillé notamment sur Graphplan avec BLUM pour le compte de *Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF,* et de l'*Advanced Research Projects Agency (ARPA)*. Il était soutenu par *NFS National Young Investigator* (CCR-9119319).

FOX Maria

Situation : *Department Computer Science, University of Durham, Durham*
Mail : Maria.Fox@durham.ac.uk
Web : http://www.dur.ac.uk/~dcs0www/personnel/dcs0mf.html
Publications : []
Travaux :

Les travaux de Maria Fox portent sur la conception et la mise en place d'un planificateur basé sur *Graphplan*, appelé STAN. Ce dernier a donné de bons résultats lors de la compétition sur la planification AIPS-98 qui s'est tenue à l'université de *Carnegie Mellon* du le 7 au 10 juin

1998. Le travail sur STAN se poursuit en commun avec *Derek Long*. Maria dirige également le groupe de planification de Durham et est membre du groupe de Raisonnement Assisté par Ordinateur.

KAMBAHAMPATI Subbarao,

Situation : *Department of Computer Science and Engineering, Arizona State university,*
Mail : rao@asu.edu
Web : http://www.eas.asu.edu/~csedept/people/faculty/rao.html
Publications : [Kamb00]
Travaux :

Il a travaillé sur l'utilité et la mise en place des techniques de Programmation par Contraintes dans *Graphplan*.

Ses intérêts de recherche sont les suivants :

- Planificateurs automatisés,
- Planification et problèmes associés en Intelligence Artificielle,
- Résolution des problèmes de coopération,
- Méthodes d'apprentissage pour les planificateurs,
- Technologies d'Intelligence Artificielle pour la création,
- Fabrication et Robotique.

NILSON Nils J.

Situation : *Robotics Laboratory, Department of Computer Science, Stanford University,*
Mail : nilsson@cs.stanford.edu
Web : http://robotics.stanford.edu/users/nilsson/
Publications :
Travaux :

Nilson a développé dans les années 70 un planificateur remarqué appelé *STRIPS* (*STanford ResearchInstitute Problem Solver*) ainsi que le formalisme associé. Son article a été publié en 1971 dans le journal « *Artificial Intelligence* ».

Actuellement ses recherches portent sur le travail de robots ayant des qualifications et une flexibilité beaucoup plus grande que ceux développés jusqu'ici. En plus de ce qui a été programmé par leurs créateurs, ces robots doivent pouvoir apprendre par l'expérience et développer leurs propres plans pour réaliser les tâches qu'on leur donne. Pour cela, Nilson a encore développé un formalisme, appelé algorithmes « teleo-réactifs (TR) » pour programmer ces robots. Ses recherches ont prouvé qu'il est facile pour des humains d'écrire ces algorithmes qui peuvent de plus être générés et modifiés par des algorithmes d'apprentissage et de planification. L'embarcation de ces algorithmes TR dans une architecture multi-agent permet à des robots de poursuivre plusieurs buts simultanément, décalant d'un but à l'autre si l'importance des buts change.

Beaucoup de ces travaux ont été réalisés avec son étudiant de PhD, Scott Benson.

POLLACK Martha E.

Situation : *Computer Science and Intelligent System Program at Pittsburgh University*
Mail : pollack@cs.pitt.edu
Web : http://www.cs.pitt.edu/~pollack/
Publications : http://www.cs.pitt.edu/~pollack/distrib/chrono-pubs.html
Travaux :

Elle a une approche de la planification et de la génération de plans résolument tournée vers les systèmes multi-agents. D'après elle, les planificateur actuels sont trop influencés par les buts à atteindre, et doivent être entièrement repensés. Elle travaille actuellement sur les représentations, les algorithmes et les architectures qui permettent de trier les taches de manières différentes. Voici donc les projets sur lesquels elle travaille présentement :

- les agents dans la génération dynamique de plans d'actions,
- la génération de plans dans des environnements dynamiques,
- plate-forme robotique pour l'éducation des étudiants en licence d'informatique,
- l'application des techniques de planification en Intelligence Artificielle aux problèmes d'ingénierie.

WELD Daniel S.,

Situation : *Department of Computer Science & Engineering, Washington University,*
Mail : weld@cs.washington.edu
Web : http://www.cs.washington.edu/homes/weld/weld.html
Publications : [Weld94], [Weld98] et http://www.cs.washington.edu/homes/weld/pubs.html
Travaux :

En dehors d'un curriculum impressionnant, les recherches actuelles de Weld portent surtout sur les agents intelligents et la planification, comme le montre la liste non exhaustive de ses dernières publications:

- Planification temporelle avec un raisonnement sur les exclusions mutuelles,
- Le moteur LPSAT et ses applications dans la planification de ressources,
- Avancées récentes en Intelligence Artificielle pour la planification,
- Compilation SAT automatique pour les problèmes de planification.

Webographie

ACTEURS DE LA PLANIFICATION

URL : http://www.cs.yale.edu/people/faculty.html

URL : http://www.cs.ucl.ac.uk/intelligent_systems/past_projects/agents/

Contenu : Listes d'acteurs de l'Intelligence Artificielle, et de la planification notamment,

URL : http://www.dur.ac.uk/~dcs0www/personnel/dcs0mf.html

Contenu : Le page Web de **Maria Fox**,

URL : http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/www/home.html

Contenu : Le page Web de **Avrim Blum**,

URL : http://www.cs.pitt.edu/~pollack/

Contenu : Le page Web de **Marta Pollack**,

URL : http://robotics.stanford.edu/users/nilsson/

Contenu : Le page Web de **Nils J. Nilsson** (STRIPS),

URL : http://www.eas.asu.edu/~csedept/people/faculty/rao.html

Contenu : Le page Web de **Subbarao Kambhampati**,

URL : http://www.cs.washington.edu/homes/weld/weld.html

Contenu : Le page Web de **Daniel S. Weld**,

PUBLICATIONS CONCERNANT LA PLANIFICATION

URL : http://www.dur.ac.uk/~dcs0www/research/stanstuff/html/publications.html

Contenu : Le page Web des publications de **Maria Fox**,

URL : http://www.cs.cmu.edu/afs/cs.cmu.edu/user/avrim/www/Papers/pubs.html

Contenu : Le page Web des publications de **Avrim Blum**,

URL : http://www.cs.pitt.edu/~pollack/distrib/chrono-pubs.html

Contenu : Le page Web des publications de **Marta Pollack**,

URL : http://rakaposhi.eas.asu.edu/papers.html

Contenu : Le page Web des publications de **Subbarao Kambhampati**,

URL : http://www.cs.washington.edu/homes/weld/pubs.html

Contenu : Le page Web des publications de **Daniel S. Weld**,

URL : http://www.cis.udel.edu/~decker/courses/889b.html,

URL : http://logos.uwaterloo.ca/~fbacchus/on-line.html,

URL : http://www.isi.edu/soar/astt/index.html,

URL : http://www.salford.ac.uk/planning/papers.htm,

URL : http://www.cs.washington.edu/research/jair/,

Contenu : D'autres pages de publications à télécharger.

PLANIFICATION

URL : http://www-poleia.lip6.fr/~jacopin/

Contenu : .

URL : http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/www/graphplan.html

Contenu : Le site web sur Graphplan.

URL : http://www.dur.ac.uk/~dcs0www/research/stanstuff/planpage.html

Contenu : .

URL : http://mcs.open.ac.uk/plansig2000/

Contenu : .

URL : http://www.hud.ac.uk/scom/research/Artform/planning.html

Contenu : .

URL : http://www.salford.ac.uk/planning/

Contenu : .

URL : http://www.informatik.uni-ulm.de/ki/ecp-99.html

Contenu : .

URL : http://liawww.epfl.ch/~frei/Cours-IA/Serie16/serie16.html

Contenu : .

URL : http://www.cirl.uoregon.edu/research/warp.html

Contenu : .

URL : http://www.salford.ac.uk/planning/news.htm

Contenu : .

A, ALGORITHME*

★★★ **URL :** http://www3.vtt.fi/tte/staff/bon/thesis/chap2/chap2_part1.html
FR **Contenu :** Le seul site intéressant que j'ai trouvé en français : peut un peu théorique ?

★★ **URL :** http://www-poleia.lip6.fr/~mynard/
FR **Contenu :** Une thèse de l'Université Pierre et Marie Curie notamment sur A*.

★★ **URL :** http://www.gameai.com/
US **Contenu :** Site très intéressant et complet sur l'Intelligence Artificielle dans les jeux.

★★ **URL :** http://www.kanding.dk/Astar.html
Contenu : Une application graphique plutôt réussie de A* : qui ne connaît pas *Warcraft* ?

★★ **URL :** http://www.geocities.com/SiliconValley/Lakes/4929/astar.html
US **Contenu :** Le code source et des exemples concrets de A* pour les développeurs

★ **URL :** http://www.cirl.uoregon.edu/research/warp.html
US **Contenu :** Une page pour voir à quoi peu servir A*…

★ **URL :** http://theory.stanford.edu/~amitp/GameProgramming/
US **Contenu :** A* et la recherche de chemin dans la programmation de jeu

URL : http://www.student.nada.kth.se/~f93-maj/pathfinder/4.html
US **Contenu :** Quelques algorithmes de recherche de chemin

RECUIT SIMULE, ALGORITHME

★★★ **URL :** http://www3.vtt.fi/tte/staff/bon/thesis/chap3/chap3.html
FR **Contenu :** Une fois encore sur ce site, une présentation détaillée.

★★ **URL :** http://web.inrets.fr/ur/dart/cohen/algsa.html
FR **Contenu :** DAISI, une application de détection d'accidents à télécharger en démonstration.

★★ **URL :** http://www-sop.inria.fr/mefisto/java/tutorial1/node12.html
FR **Contenu :** APPLET JAVA du recuit simule (Voyageur de commerce) et d'autres algorithmes: pas mal!

★ **URL :** http://www.ibcp.fr/~deleage/Cours/MaitBioch/98_99/sld180.htm
FR **Contenu :** Une application en biochimie.

URL : http://perso-sc.enst-bretagne.fr/~chonavel/pap/sig_aleatoire/sig_aleatoire/node180.htm
FR **Contenu :** Principe général en général !

TABOU, METHODE OU RECHERCHE

★★ **URL :** http://www.lip6.fr/reports/lip6.1997.030.html
FR **Contenu :** Une thèse de doctorat sur l'Intégration du Raisonnement à Partir de Cas dans la Méthode Tabou.

★★ **URL :** http://dmawww.epfl.ch/~delay/projet1/projet1.html

FR **Contenu :** Présentation en ligne d'une application de la méthode Tabou de l'Ecole Polytechnique de Lausanne.

Bibliographie

B

[BlFu97] **Titre :** *Fast Planning Throught Planning Graph Analysis,*
Auteurs : Avrim L. BLUM et Merrick L. FURST,
Info : Version finale dans *Artificial Intelligence, 90* :281-300, 1997.

[BoTh00] **Titre :** *L'intelligence en essaim,*
Auteurs : Eric BONABEAU et Guy THERAULAZ,
Info : Dans *Pour la Science,* 271, 66-73, *Dossier mars 2000*.

D

[DaVo90] **Titre :** *Dynamic tabu list management using the reverse elimination method,*
Auteurs : F. DAMMEYER et S. VOSS,
Info : *Anals of Operation Research, 41,* 31-46, 1990.

[Dows93] **Titre :** *Simulated annealing,*
Auteurs : R. AGLESE,
Info : Dans *Modern heuristic techniques for combinatorial problems,* (pp. 20-69) C. R. REEVES (Ed.), *Blackwell Scientific Publications,* 1993.

E

[Egle90] **Titre :** *Simulated annealing : a tool for combinatorial research,*
Auteurs : R. AGLESE,
Info. : *European Journal of Operation Research, 46,* 271-281,1990.

G

[GlGr89] **Titre :** *New approaches for heuristic search : a bilateral linkage with AI*,
Auteurs : F. GLOVER et H. J. GREENBERG,
Info. : *European Journal of Operation Research, 39,* 1989.

[GlLa93] **Titre :** *Tabu Search,*
Auteurs : R. AGLESE,

Info : In *Modern heuristic techniques for combinatorial problems,* (pp. 70-150) C. R. REEVES (Ed.), *Blackwell Scientific Publications,* 1993.

[Glov89] **Titre :** *Tabu search – part. I*,
Auteurs : F. GLOVER,
Info. : *ORSA Journal on Computing, 1*(3)*,* 190-206.
Télécharger : ?

[Glov90] **Titre :** *Tabu search – part. II*,
Auteurs : F. GLOVER,
Info. : *ORSA Journal on Computing, 2*(1)*,* 4-32.

[Glov96] **Titre :** *Tabu search and adaptative memory programming – advanced, applications and cahllenges*,
Auteurs : F. GLOVER,
Info. : Dans R. S. Barr, R. V. Helgason, et J. L. Kennigton (Eds.), *Interfaces in computer science and operations research* (pp. 1-75). Kluwer Academic Publisher.

[GTD93] **Titre :** *A user's guide to tabu search*,
Auteurs : F. GLOVER, E. TAILLARD et D. DE WERRA,
Info. : *Anals of Operation Research, 41,* 3-28, 1993.

H

[HNR68] **Titre :** *A formal basis for the heuristic determination of minimum cost paths*,
Auteurs : P. E. HART, N. J. NILSON et B. RAPHAEL,
Info. : *IEEE Transactions on SSC*, 4, 1968.

J

[JAMS89] **Titre :** *Optimization by simulated annealing : an experimental evaluation ; part I, graph partitioning*,
Auteurs : D. S. JOHNSON, C. R. ARAGON, L. A. McGEOCH et C. SCHEVON,
Info. : *Operation Research, 37*(6), 865-892, 1989.

K

[Kamb00] **Titre:** *Planning Graph as a (Dynamic) CSP : Exploiting EBL, DDB and other CSP Search Techniques in Graphplan*,

Auteurs : Subbarao KAMBHAMPATI,
Info. : *Journal of Artificial Intelligence Research,* 12,(2000), 1-34.
Télécharger : http://rakaposhi.eas.asu.edu/pub/rao/gp-ebl-tr.ps.

[KGA93] **Titre:** *Large-scale cotroled rounding using tabu search with strategic oscillation*,
Auteurs : J. P. KELLY, B. L. GOLDEN et A. A. ASSAD,
Info. : *Anals of Operation Research, 41,* 69-84, 1993.

[KGV83] **Titre:** *Optimisation by simulated annealing*,
Auteurs : S. KIRKPATRICK, C. GELLAT et M. VECCHI,
Info. : *Science, 220,* 671-680, 1983.

[Korf85] **Titre :** *Depth-first iterative deepening : an optimal admissible tree search*,
Auteurs : R. KORF,
Info. : dans *Artificial Intelligence, 27*, 97-109.

M

[MRRT53] **Titre :** *Equation of State Calculations by fast Computing Machines*,
Auteurs : N. METROPOLIS, A. et M. ROSENBLUTH, A. et E. TELLER,
Info. : *Journal of Chemical Physics* 21, 1953.

[Myna97] **Titre :** *Exploration locale oscillante heuristiquement ordonnée*,
Auteurs : Laurent MYNARD,
Info. : thèse de l'université Pierre et Marie Curie (Paris 6), janvier 1998.
Télécharger : http://www-poleia.lip6.fr/~mynard/ps/mathese.ps.gz

N

[Nils82] **Titre :** *Principles of Artificial Intelligence, Symbolic Computation*,
Auteurs : N.J. Nilsson,
Info. : Ed. Springer-Verlag, Berlin Heidelberg, New York, 1982.

P

[PeKi82] **Titre :** *Studies in Semi-Admissible Heuristics*,
Auteurs : J. PEARL et J.H. KIM
Info. : IEEE Transactions PAMI-4, juillet 1982, pp. 392-400

W

[Weld94] **Titre :** *An Introduction to Least Commitment Planning*,
Auteurs : Daniel S. Weld,
Info. : dans *AI Magazine*, Summer / Fall 1994.
Télécharger : ftp://ftp.cs.washington.edu/pub/ai/pi.ps

[Weld98] **Titre :** *Recent Advances in AI Planning*,
Auteurs : Daniel S. Weld,
Info. : dans *AI Magazine*, 1999.
Télécharger : ftp://ftp.cs.washington.edu/pub/ai/pi2.ps

Index

VIII.

Annexes