Résolution de problèmes sous contrainte de ressources à l'aide de réseaux neuromimétiques : application à l'ordonnancement

Jean-Michel Gallone

FACULTÉ DES SCIENCES

UFR S.T.M. I. A.
Ecole Doctorale IAE + M
Département de Formation Doctorale en Informatique

THÈSE

présentée pour l'obtention du titre de

Docteur de l'Université Henri Poincaré, Nancy I
en Informatique

par **Jean-Michel GALLONE**

Résolution de problèmes sous contrainte de ressources à l'aide de réseaux neuromimétiques : Application à l'ordonnancement

Présentée et soutenue publiquement le 6 janvier 1997 devant la commission d'examen :

Membres du jury :

Présidente	Marie-Claude Portmann	Professeur, Ecole des Mines de Nancy.
Rapporteurs	Claude Kaiser	Professeur, Conservatoire National des Arts et Métiers.
	Shlomo Zilberstein	Professeur, University of Massachusetts Amherst.
Directeurs de thèse	Jean-Paul Haton	Professeur, Université Henri Poincaré Nancy I.
	François Charpillet	Chargé de Recherche, INRIA.
Examinateur	Bernard Amy	Ingénieur de Recherche, Université de Grenoble.
Invité	Michel Barès	Ingénieur, DGA / DRET.

Centre de Recherche en Informatique de Nancy
CRIN - CNRS & INRIA Lorraine
Faculté des Sciences - 54500 VANDŒUVRE-lès-NANCY

A mes deux auteurs préférés...

"Oui"

Claudine, 15 juillet 1995

"Papa"

Baptiste, Septembre 1996

Remerciements

Je souhaite remercier vivement tous les membres du jury :

Claude Kaiser pour avoir accepter de juger mon travail. Ses remarques pertinentes ont permis d'améliorer la clarté de ce manuscrit.

Shlomo Zilberstein pour avoir bien voulu être rapporteur de ma thèse. Son regard expérimenté apporte à mon travail une plus grande crédibilité. Ses travaux ont été un guide à de nombreuses reprises pour mes recherches.

Michel Barès et Bernard Amy pour avoir accepter de juger cette thèse.

Enfin, un remerciement particulier à Marie-Claude Portmann, qui fut à la fois mon rapporteur de thèse et ma tutrice d'enseignement à l'école des Mines. Sa collaboration fut enrichissante tout au long de ma thèse, tant au niveau de la recherche que de la pédagogie. Ces conseils avisés sur l'ordonnancement m'ont été précieux pour la rédaction de ce manuscrit.

Je tiens à exprimer ma reconnaissance à Jean-Paul Haton, qui m'a fait confiance et m'a acceuilli au sein de l'équipe RFIA. Ce fut un plaisir d'effectuer mon travail de recherche sous sa direction.

Un grand merci à François Charpillet, mon directeur de thèse. Sa disponibilité, ses qualités humaines et scientifiques, ainsi que sa faculté à travailler en temps contraint resteront pour moi un exemple. Travailler sous sa direction a été une très grande chance.

Ma gratitude va également à tous les membres de l'équipe RFIA, pour leur amitié et pour les nombreuses discussions et collaborations tout au long de cette période. Je remercie notamment Frédéric Alexandre, pour ses conseils sur les réseaux neuromimétiques; Olivier Aycard, pour sa collaboration sur gaston; Anne Boyer, pour sa constante bonne humeur; Vincent Chevrier, pour la vision multi-agents; Dominique Fohr, pour avoir, entre autre, réalisé l'interface graphique; Norbert Glaser, pour m'avoir motivé à planter mon drapeau; Philippe Haïk, pour sa sympathie; Abdel-Illah Mouaddib, pour les discussions sur les systèmes temps réels; Malek Mouhoub, pour son expérience sur la propagation de contraintes; Martine Kuhlmann, pour sa gentillesse et sa patience; Richard Washington, pour ses traductions du Franglais à l'Anglais.

Si mon séjour au Crin se fit dans d'aussi bonnes conditions, la présence de nombreux amis n'y est pas étrangère, je remercie Coco pour ses rires communicatifs et son éternelle bonne humeur; Ldl pour la relecture attentive du manuscrit, ses mails nycthémères et ses buts légendaires; Djeff pour sa sympathie et ses éternelles informations; ainsi que Remy, Stephane, Yannick, Daniel, XRM, Jean-luc et les footeux...

Je termine en remerciant ceux qui m'ont été indispensable durant ces années, ma famille : Claudine et son éternelle joie de vivre, Baptiste et ses sourires ravageurs, mes parents pour leur soutien constant, Claudine, Fabrice, Lucille, Olivier, Véro (qui ne vera aucun rapport avec sa thèse, j'espère), Pichon, Guillaume.

Enfin je voudrais exprimer une pensée pour Laurent, les cartes d'anniversaires auront désormais toujours trop de place...

Table des matières

Partie 2 : Ordonnancement par contrats

Liste des Figures

Introduction

L'informatique est aujourd'hui omniprésente dans l'industrie. Son introduction est dans la majorité des cas motivée par des aspects essentiellement lucratifs, mais son utilisation est variée et évolue selon la demande. Les objectifs de la plupart des premiers systèmes informatiques étaient essentiellement basés sur des critères de productivité et de rentabilité. L'entreprise utilisait l'ordinateur pour la réalisation de tâches simples et répétitives afin de rendre l'homme plus productif. Ceci permettait un travail plus rapide, plus régulier et plus rentable. Face à l'évolution régulière de la puissance des machines pour des prix en constante diminution, l'informatique s'est largement développée dans l'industrie et les tâches à résoudre sont devenues de plus en plus nombreuses et de plus en plus complexes. D'autre part, dans la conjoncture actuelle, les entreprises sont tenues de miser sur la qualité de leurs produits et de leurs services autant que sur la productivité et la rentabilité. Pour cela, l'informatique peut être un outil formidable.

L'implantation d'un système informatique dans un processus physique a désormais pour but d'augmenter sa rentabilité et sa rapidité mais aussi sa fiabilité, sa sécurité et la qualité des produits fabriqués. Un tel système n'est plus limité à des travaux simples et répétitifs, mais peut se consacrer à des tâches d'un niveau plus élevé telles que le contrôle de processus, le diagnostique de panne, la prise de décision... Cependant, l'augmentation des fonctionnalités du système induit des traitements d'une durée considérablement plus longs. Ainsi le facteur temps devient alors un paramètre important qui influe sur le choix des techniques de résolution à mettre en œuvre.

Considérons, à titre d'exemple, un programme joueur d'échecs. A partir d'une position de l'échiquier, il est possible d'établir la liste exhaustive des différents coups réalisables et de leur affecter une note afin de choisir le meilleur. Cette opération simple nécessite très peu de calcul. Cependant, même une très bonne connaissance des échecs ne suffit pas à garantir un bon niveau de jeu pour le programme. La tentation est donc forte d'étudier les conséquences du coup joué sur le déroulement futur de la partie. Ceci se résout fort simplement en théorie en considérant l'algorithme qui teste toutes les combinaisons possibles et exécute ensuite le coup qui mène à la victoire. Ce programme, qui serait imbattable, n'a pourtant jamais été implémenté, puisqu'une simple étude de complexité montre qu'il faudrait attendre plusieurs siècles avant de jouer le premier coup. Dans ce cas, le meilleur algorithme n'est pas celui qui donne la solution de plus grande qualité, mais celui qui trouve une solution ayant une qualité suffisante pour un temps de calcul raisonnable, très peu de joueurs ayant la patience d'attendre plusieurs siècles...

D'autre part, il est important de bien maîtriser le temps de calcul lorsque le système est plongé dans un monde extérieur qui évolue. En effet, certaines données ont alors une durée de vie au delà de laquelle elles deviennent obsolètes, leur traitement devenant alors inutile. Le système doit tenir compte des durées de validité et, soit donner une réponse avant un certain délai, soit incorporer les modifications extérieures de façon à donner un résultat valable pour les nouvelles données.

Ainsi, le fonctionnement en temps réel est un enjeu important pour un système informatique. Mais sa mise en place nécessite souvent une étude différente du problème traité. D'une part, la qualité d'un algorithme ne se juge plus uniquement sur la qualité du résultat obtenu, mais également sur le temps nécessaire à son obtention. D'autre part, l'aspect dynamique du monde extérieur impose une attention permanente aux changements de situations. Ces deux aspects sont difficilement compatibles. En effet, comme c'est le cas pour un être humain, il est difficile pour un système informatique d'être à la fois focalisé sur un traitement urgent et constamment à l'écoute de l'extérieur. C'est pourquoi, il s'avère nécessaire de trouver un compromis, plus exactement de définir une politique qui permette de décider à tout instant s'il faut se focaliser sur un traitement ou sur la perception du monde extérieur.

Un tel système doit donc dégager des qualités essentielles telles que la rapidité, la fiabilité, la garantie d'un temps de réponse et une certaine capacité de focalisation afin de faire face aux urgences. Devant ces nombreuses exigences, les solutions algorithmiques classiques rencontrent de grandes difficultés pour résoudre les problèmes de complexité importante. Les techniques d'intelligence artificielle constituent un apport intéressant dans ce domaine. Elles permettent une réalisation plus simple par une spécification claire des données et des buts du système. Un second avantage est la possibilité de manipuler des données incertaines. Ceci autorise le travail avec une connaissance approximative du monde et permet donc de donner une solution là où un programme algorithmique classique attendrait plus de précision. Enfin, dans les systèmes d'intelligence artificielle, il est possible de favoriser le respect des échéances en implantant une certaine capacité de focalisation qui restreint l'ensemble des données sur lequel sont effectués les traitements. Néanmoins, l'intégration des techniques d'intelligence artificielle dans un grand système pose encore de nombreuses difficultés telles que la capacité à raisonner sous contrainte de ressources (notamment celles liées au temps réel). En effet, l'indéterminisme des traitements utilisés exclut une gestion fine des ressources que nécessite un système temps réel. Ainsi, les techniques temps réel classiques qui supposent une parfaite connaissance des durées d'exécution ne vont pas pouvoir être utilisées dans ce cadre.

Notons que nous ne nous situons pas directement au niveau des systèmes d'exploitation temps réel, mais plutôt à un niveau supérieur qui utilise de tels systèmes pour proposer des applications temps réel à un opérateur. Ainsi les temps de réponse que nous allons utilisés ici seront plus de l'ordre de la seconde que de la micro-seconde. Nous allons donc pouvoir profiter des services proposés par les systèmes temps réels traditionnels afin de définir un système plus complexe utilisable par un opérateur.

Dans ce contexte, la résolution de problèmes sous contrainte de ressources apparaît comme une solution prometteuse. Il s'agit de considérer l'évolution de l'utilité d'une réponse selon la date à laquelle elle est disponible, d'étudier simultanément l'évolution de la qualité de la réponse selon le temps utilisé pour la résolution. Il est alors possible

d'adopter le comportement optimal, c'est-à-dire de choisir le meilleur compromis entre le coût de l'exécution et la qualité du résultat. Pour la réalisation de tels traitements, les chercheurs en intelligence artificielle ont proposé des modifications intrinsèques des tâches non déterministes afin d'adapter leur comportement en fonction du temps disponible. Pour cela, deux grandes familles d'algorithmes ont été développés : par affinements itératifs ou par méthodes multiples.

Boddy et Dean [Boddy 89] ont défini les algorithmes *anytime* comme une procédure interruptible fondée sur un calcul itératif qui produit toujours un résultat dont la qualité est fonction du temps alloué. Ils supposent pour cela que la qualité du résultat augmente de façon continue avec le temps de calcul. Chaque procédure peut ainsi être caractérisée par ses performances, une estimation de l'évolution de la qualité du résultat selon le temps alloué à la résolution. Cette fonction permet alors la réalisation du meilleur compromis entre le coût de l'obtention et la qualité du résultat. Russel et Zilberstein [Russell 91] ont par la suite étendu cette notion d'algorithme *anytime* en introduisant une différenciation entre deux types d'algorithmes : d'un coté les algorithmes interruptibles qui fournissent une réponse quel que soit l'instant où ils sont interrompus; d'un autre coté les algorithmes par contrats qui se basent sur un temps de calcul fixé *a priori* (le système doit déterminer le temps qui leur sera alloué avant d'entamer leur exécution). Ceci étend l'ensemble des algorithmes *anytime* aux procédures où une variable permet de paramètrer la qualité du résultat et le temps de calcul. Les algorithmes par contrats peuvent ne pas fournir de résultats s'ils sont interrompus avant le temps qui leur avait été initialement alloué (c'est-à-dire si le contrat n'est pas respecté). Cependant, tout algorithme par contrats peut se transformer en un algorithme interruptible modulo une chute de performance.

Le paradigme *anytime* repose sur la capacité du concepteur à produire une solution itérative pour résoudre une problème donné, ce qui n'est pas toujours possible. Il est parfois plus aisé de définir un ensemble de méthodes où chacune utilise son propre algorithme pour la résolution d'une même tâche selon une complexité et une qualité différente. La méthode la plus appropriée au temps de calcul disponible est alors choisie dynamiquement lors de l'exécution. Cette approche, introduite par Lesser dans DVMT, est appelée *calcul approximatif*. Dans [Lesser 88], les auteurs suggèrent trois types d'approximation : approximation de la recherche (diminution de l'espace de recherche exploré en évitant les régions ayant une faible probabilité de contenir des informations intéressantes), approximation des données (en ne considérant que les données les plus importantes), approximation des connaissances (en traitant les données par plusieurs règles au cours d'une même inférence).

Design to time [Garvey 92] est une généralisation du calcul approximatif. Les auteurs ont introduit un modèle de tâche nommé TÆMS et un algorithme cyclique d'ordonnancement. Une tâche est un graphe acyclique de sous-tâches où les arcs ont pour sémantique la relation tâche/sous-tâche, les feuilles étant les méthodes de

résolution des sous-tâches. En utilisant ce modèle de tâche, l'ordonnanceur propose une façon de résoudre un problème dans le temps imparti.

Ainsi, plusieurs méthodes existent pour la construction d'algorithmes progressifs. A l'aide de ces méthodes, il est possible de définir un raisonnement sous contrainte de ressources qui garantisse un temps de réponse. Dans [Zilberstein 96], Zilberstein propose la méthodologie suivante pour la mise en place de tels systèmes :

- **Définir des algorithmes progressifs**. Les connaissances doivent être codées de telle façon qu'il soit possible de réaliser un compromis entre la durée d'exécution et la qualité de la réponse obtenue.

- **Estimer les performances**. Les traitements sont exécutés sur une machine donnée. Selon sa puissance, il est possible de calculer les performances des différents algorithmes, à savoir l'espérance de la qualité de la réponse en fonction du temps de calcul. Cette estimation est nécessaire pour la réalisation d'un module de supervision efficace.

- **Composer**. Dans le cas où les performances de l'algorithme et l'échéance de la réponse sont parfaitement connues, il est possible de calculer *a priori* la séquence d'exécutions la plus efficace. Les réponses sont ainsi délivrées à des dates précises de manière à maximiser l'utilité globale.

- **Superviser**. Dès lors que les performances et/ou les échéances sont grossièrement estimées voire totalement inconnues, le problème d'allocation ne peut plus se faire *a priori*. Il doit être réalisé en cours d'exécution par un module de contrôle qui peut décider à tout instant, selon la qualité de la solution et la proximité de l'échéance, si le traitement doit être stoppé ou non. Dans le cas d'algorithmes progressifs interruptibles, la conduite du raisonnement est simple puisque le traitement s'améliore sans cesse au cours du temps. Il suffit donc de stopper quand la réponse est la plus utile, c'est-à-dire quand la valeur de la qualité diminuée du coût lié au temps d'obtention est maximale. Dans les cas où les algorithmes progressifs utilisés ne sont pas interruptibles, la décision doit se faire dynamiquement en fonction de l'échéance, des performances des algorithmes et de la charge de la machine. Cependant, quand un ou plusieurs de ses paramètres ne sont pas parfaitement connus, il faut définir une politique d'activation qui permette d'espérer un résultat disponible en temps voulu et de qualité satisfaisante.

Cette thèse propose l'utilisation de réseaux neuromimétiques pour la réalisation d'un tel raisonnement. Le but est de définir une méthode efficace qui permette la mise en place simplifiée de résolution de problèmes sous contrainte de ressources. Une méthode de

construction générale est ainsi définie, puis appliquée par la suite à un problème classique d'ordonnancement. Dans un premier temps, le problème est résolu par un système basé sur des réseaux neuromimétiques. Ce système est ensuite modifié afin de mettre en évidence une notion d'approximation pour permettre la réalisation de compromis entre le temps d'exécution et la qualité. Le système se transforme ainsi en un ensemble d'algorithmes sur lequel on définit une politique d'activation permettant une transformation en une seule exécution interruptible. Notre but est ainsi d'ordonnancer des algorithmes qui traitent des problèmes d'ordonnancement.

Organisation du document

La première partie de ce document présente les différents problèmes liés à un fonctionnement temps réel. La prise en compte d'un environnement extérieur nécessite la réalisation de perceptions et d'actions qui sont parfois contraintes par un temps de réponse. Ainsi différents concepts ont été introduits afin de définir un comportement efficace dans un contexte ou la ressource temps est limitée. Nous présenterons la notion de progressivité d'un algorithme qui permet une optimisation du rapport temps de calcul sur qualité du résultat. Nous énumérerons différents types d'algorithmes possédant les qualités désirées ainsi que les modifications à apporter à un algorithme classique pour obtenir les propriétés voulues. Nous présenterons également le système Reakt, un générateur de système à base de connaissances temps réel conçu lors d'un projet Européen Esprit, qui intègre bon nombre de ces concepts. Ensuite nous étudierons les modules de supervision permettant une utilisation optimale des algorithmes progressifs. Enfin nous spécifierons le problème de résolution de problème sous contrainte de ressources de façon plus formelle.

La seconde partie concerne la réalisation d'un problème d'ordonnancement de tâches non préemptives à l'aide d'un système travaillant en temps contraint. Après avoir présenté le problème particulier auquel nous nous sommes intéressés, nous recensons les différentes techniques utilisées pour résoudre au mieux ce problème qui n'a pas de solution optimale calculable en un temps polynomial. Puis nous détaillerons l'utilisation de modèles neuromimétiques (modèles inspirés du fonctionnement de neurones biologiques) pour résoudre des problèmes d'optimisation avec un système permettant de réaliser des compromis entre le temps d'exécution et la qualité de la réponse. Nous présenterons ensuite l'adaptation de ce système pour la résolution du problème d'ordonnancement considéré, ainsi que les facultés du modèle proposé à travailler dans un contexte temps réel. Nous étudierons alors les performances de ce système afin d'établir différentes politiques de supervision optimisant le rapport entre le temps de calcul et le coût de l'obtention selon l'évolution de l'utilité de la réponse au cours du temps.

Nous terminerons par une conclusion sur l'intérêt d'un tel système, ses possibilités d'extension et les différents types d'applications pouvant être résolue de la même façon. Nous étudierons également comment il peut s'intégrer dans un outil tel que Reakt.

Partie 1 : Exécution sous contrainte de ressources

1. Problèmes liés au temps réel

Tout système peut plus ou moins être considéré comme temps réel puisqu'il est impossible de disposer d'un temps infini pour un traitement donné. Cependant le facteur temps n'est pas un paramètre majeur dans tous les cas. Considérons un programme de jeu d'échec, il est possible de jouer contre l'ordinateur sans imposer de limite de temps à chaque tour. Le programme met alors en œuvre toutes les techniques à sa disposition afin de sélectionner le coup qu'il considère comme le meilleur. Mais une variante du jeu consiste à limiter la durée des tours. Ainsi le temps de calcul accordé à l'ordinateur pour le choix d'un coup est borné et sa réponse doit obligatoirement être produite dans le temps imparti. Dans ce cas, le choix des techniques à utiliser pour sélectionner un coup est fortement influencé par le temps de calcul de chacune d'elles. Il s'agit là d'un exemple de système fonctionnant en temps réel.

Ainsi, la vitesse d'exécution est un critère déterminant pour de tels systèmes où la réponse est évaluée non seulement par sa qualité mais aussi par son délai d'obtention, un résultat optimal mais trop tardif étant alors d'une totale inutilité. Ce sont ces systèmes, dont le fonctionnement dépend d'événements externes, qui font l'objet de notre étude.

Il nous faut maintenant définir plus clairement la notion de temps réel. Dans la littérature, les nombreuses définitions proposées sont centrées sur la capacité de prise en compte d'un temps de réponse. Citons par exemple :

> «Il existe une date limite stricte, indépendante des algorithmes employés, pour la production d'une solution.» [O'Reilly 85]

ou :

> «C'est un système capable de garantir une réponse après qu'un certain temps lié au domaine s'est écoulé.» [Laffey 88]

ou encore :

> «Une application temps réel est un système de traitement de l'information ayant pour mission de commander un environnement imposé, en respectant les contraintes de temps et de débit (temps de réponse à un stimulus, taux de perte d'information toléré par entrée) qui sont imposés à ses interfaces avec cet environnement.» [Browaeys 84]

Les problèmes inhérents au temps réel ont été étudiés depuis longtemps en informatique classique et notamment en informatique industrielle. Depuis quelques années, l'utilisation de l'intelligence artificielle a débouché sur une nouvelle vision du problème. Le lecteur peut se référer à [Charpillet 94b] pour un bon survol de l'état de l'art. Ce chapitre présente la principale difficulté rencontrée un système temps réel : le respect des échéances.

1.1. Garantir un temps de réponse

La caractéristique fondamentale d'un système temps réel est la garantie d'un temps de réponse que l'on peut définir de la façon suivante :

> «La garantie d'un temps de réponse est la capacité à réagir devant des événements externes avant une échéance.»

Le système doit pouvoir se synchroniser sur une horloge externe afin d'intégrer la notion de durée d'exécution dans les techniques de résolution qu'il met en œuvre. En effet, dans la conduite d'un processus physique, certains traitements peuvent avoir une date limite au delà de laquelle il est inutile de les exécuter : par exemple le programme d'échec doit jouer avant le délai imposé, de même si l'on considère la navigation d'un robot mobile autonome, il doit être en mesure de contourner un nouvel obstacle qui se présente avant de le heurter.

Le système doit considérer le temps de réponse alloué et le temps de traitement nécessaire face à certains événements afin de pouvoir garantir l'efficacité de ses actions. Dans certains cas, le comportement à adopter va être fortement influencé par les délais à respecter. Le système doit planifier les traitements en fonction de leur urgence et éventuellement de leur importance afin de respecter les temps de réponse imposés. Mais la réalisation d'un planning figé est difficilement envisageable étant donné la nécessité de réactivité. La difficulté vient du fait que le planning établi respectant les échéances peut être remis en cause à tout instant par un changement de situation du monde physique. En effet, un tel changement peut entraîner la création de nouveaux traitements et donc une charge supplémentaire pour le système.

Aux propriétés de réactivité, on désire donc greffer les notions de ponctualité. En effet, un système temps réel est parfois amené à fournir une réponse dans un délai imposé par l'extérieur. Or, si la charge de travail demandée est importante, si le temps passé à réaliser des perceptions ou si les échéances imposées sont courtes, le système ne pourra pas effectuer tous les traitements dans le temps imparti. Par conséquent, il est nécessaire de réaliser un système qui soit le plus efficace possible afin d'éviter les problèmes liés aux limites de temps. Il faut également utiliser des techniques temps réel qui adaptent le

traitement au temps de réponse voulu. Pour cela, plusieurs politiques complémentaires peuvent être adoptées :

- La solution la plus simple et la plus efficace consiste à supprimer les tâches les moins importantes. Mais cette politique peut être catastrophique pour la fiabilité du système, aussi est-elle à éviter dès que possible. C'est la politique classiquement utilisée pour les systèmes temps réel de bas niveau qui doivent garantir l'exécution de tâches sur un processeur. Une nouvelle tâche peut être acceptée ou refusée selon la charge actuelle de la machine et les caractéristiques de la tâche en question. Un bon survol de état de l'art peut être trouvé dans [Alabau 92].

- Une deuxième alternative, beaucoup plus efficace, consiste à améliorer les performances du système. L'idée est de se doter d'une machine suffisamment puissante pour être certain de ne jamais avoir des traitements plus longs que la durée allouée à l'exécution. Pour cela, il faut être en mesure d'évaluer correctement l'offre et la demande, c'est-à-dire la charge maximale du système et la puissance de calcul disponible sur la machine. Si ces estimations sont réalisables, il est alors possible de dimensionner la machine afin de supporter la charge maximale du système. Cela peut se faire de nombreuses façons, en utilisant un microprocesseur plus puissant, en optimisant le programme, en utilisant les techniques de parallélisation... Cette politique est plus efficace que la précédente, mais ne suffit toujours pas pour des problèmes très complexes où aucune machine ne sera suffisamment puissante, ou pour des problèmes où l'estimation de la charge maximale du système n'est pas calculable.

- Ainsi il est souvent nécessaire d'avoir recours à la troisième alternative, qui consiste à réduire la durée de certaines tâches par l'utilisation d'approximation. Cette politique revient à diminuer la qualité de la solution fournie, ce qui raccourcit l'exécution et permet ainsi le respect de l'échéance malgré la charge élevée. Cette approximation sous-entend donc une expertise très poussée devant dégager différents niveaux de résolution correspondant aux différentes qualités de la solution. De plus, pour mettre en œuvre un tel type de raisonnement, il faut modéliser les connaissances de façon à pouvoir les utiliser de manière progressive et développer des algorithmes adaptés à la modélisation choisie.

1.1.1. Améliorer les performances

Pour améliorer les performances, la première idée qui vient à l'esprit est l'augmentation de la puissance de calcul. C'est la façon la plus simple et la plus sûre d'être plus performant, mais c'est également la plus coûteuse. Ainsi, même s'il est nécessaire d'avoir une machine performante pour travailler en temps réel, l'augmentation de cette

puissance de calcul n'est à envisager que si celle qui est disponible actuellement est exploitée au maximum de ses capacités. De plus, bien qu'augmentant rapidement et régulièrement, la puissance des machines reste limitée. Ainsi il semble nécessaire d'avoir recours à d'autres techniques.

L'intelligence artificielle permet une séparation nette entre les connaissances et les algorithmes. L'amélioration des performances peut ainsi se faire à deux niveaux, soit au niveau de l'organisation des données, par une compilation de la base de connaissances, soit au niveau de l'algorithme qui utilise ces données, par des optimisations et éventuellement une parallélisation des traitements, si les moyens techniques utilisés le permettent. Les paragraphes suivants présentent les différentes façons d'améliorer les performances pour chacun de ces niveaux.

1.1.1.1. Compilation

Dans notre cas, la compilation se définit comme un processus de transformation de données brutes en un ensemble de données structurées de façon à être utilisées efficacement par l'algorithme qui les traite. Elle consiste en une réorganisation des connaissances déclaratives initiales afin de rendre l'algorithme qui les utilise plus performant. Il s'agit donc de réaliser sur la base de connaissances des pré-traitements qui ont pour but de faciliter les opérations temps réel ultérieures.

Dans le cas de règles, la compilation correspond, par exemple, à factoriser les parties gauches (les conditions nécessaires à son déclenchement) de façon à ce que le filtrage soit le plus efficace possible. Une autre amélioration possible concernant l'algorithme de filtrage est de conserver les résultats des cycles précédents. Il suffit alors de réaliser les tests liés aux modifications de la base pour obtenir incrémentalement le résultat du nouveau filtrage. Les algorithmes de filtrage à sauvegarde d'état les plus connus sont les algorithmes Rete [Forgy 82] et Treat [Miranker 87].

1.1.1.2. Optimisation

Un gain de temps non négligeable peut également être réalisé par une implémentation efficace des algorithmes qui utilisent la base de connaissances. En effet, un moteur d'inférence, comme tout algorithme peut être implémenté de manière plus ou moins efficace. Dans notre cas, une mauvaise implémentation peut avoir de lourdes conséquences sur le comportement du système. Pour cela, trois étapes d'optimisation peuvent être dégagées :

- L'algorithme sélectionné doit avoir la plus faible complexité possible. En effet, de toutes les optimisations envisageables, celle qui consiste à réduire l'ordre de la complexité de l'algorithme est de loin la meilleure. Mais dans bien des cas, si l'étude a été correctement réalisée, l'algorithme choisi aura une complexité qui ne pourra être réduite. Il est cependant utile de ne pas négliger cette étape. Par

exemple, pour l'algorithme de filtrage évoqué ci-dessus, il est possible de diminuer le nombre de tests grâce à des techniques de factorisation des tests de semi-unification et des jointures.

- L'implémentation de l'algorithme doit être la plus efficace possible. Pour cela certaines règles sont à respecter, telles que la localité de référence (qui vise à limiter les accès mémoire dispersés), l'optimisation de boucle (qui vise à limiter les calculs inutiles dans les itérations). Des outils de mesure de performance et de gestion de la mémoire facilitent une telle optimisation.

- Il ne faut pas négliger les options de compilation du programme qui permettent de réaliser un gain de temps considérable si elles sont adaptées à la machine et à l'implémentation utilisée.

1.1.1.3. Parallélisme

Une troisième possibilité d'amélioration des performances consiste à utiliser la parallélisation des traitements. Cela peut se faire à trois niveaux différents :

- Au plus bas niveau, par l'utilisation d'une machine *pipeline*. Ces machines ont la faculté de pouvoir enchaîner rapidement des opérations grâce au codage des micro-instructions. En effet, ces dernières n'utilisent pas en permanence tous les registres ni les micro-opérateur de la machine. Par exemple durant une multiplication, il est possible de transférer les données qui serviront à la multiplication suivante. De même, pendant le transfert du résultat, la multiplication suivante peut être effectuée de même que le chargement des données de la troisième multiplication. Chaque instruction possède un temps de latence qui peut être soit perdu soit mis à profit pour lancer les instructions suivantes. Ainsi, une bonne exploitation nécessite parfois quelques modifications du programme mais permet un gain de temps considérable.

- A un niveau plus élevé, par l'utilisation d'une machine parallèle, l'exécution sur plusieurs microprocesseurs permet d'améliorer considérablement les performances du système, à condition de les utiliser de manière efficace. Il faut cependant faire attention aux problèmes de maintien de la cohérence des données.

- Au niveau le plus élevé, par la programmation distribuée. Cela consiste à utiliser plusieurs machines, chacune pouvant posséder plusieurs micro-processeurs. Ceci nécessité l'utilisation d'un protocole de communication (tel que MPI [MPI 94] ou PVM [Geist 94]).

Cette politique est efficace, mais coûteuse et ne peut suffire à elle seule à résoudre les problèmes de performance.

1.1.2. Diminuer la complexité

Afin de garantir un temps de réponse, il est important de s'assurer que le système travaille de la façon la plus efficace possible, en exploitant au mieux une machine qui sera de préférence performante. Cependant, une exécution rapide ne peut suffire à garantir un temps de réponse dans tous les cas. En effet, la complexité de l'algorithme à exécuter est un facteur beaucoup plus déterminant que la vitesse de calcul. Pour s'en persuader, il suffit de regarder le tableau de la figure 1 qui exprime les temps d'exécution de diverses instances en fonction de l'ordre de la complexité. Dans ce tableau, on considère une opération quelconque qui s'exécute en une microseconde. Ainsi par exemple pour une complexité de n^2 et un problème de taille 50, il faut réaliser 2500 opérations, ce qui nécessite 2,5 millisecondes.

Complexité	n=10	n=20	n=30	n=40	n=50	n=100
n	0.00001 secondes	0.00002 secondes	0.00003 secondes	0.00004 secondes	0.00005 secondes	0.0001 secondes
n^2	0.0001 secondes	0.0004 secondes	0.0009 secondes	0.0016 secondes	0.0025 secondes	0.01 secondes
n^3	0.001 secondes	0.008 secondes	0.027 secondes	0.064 secondes	0.125 secondes	0.216 secondes
2^n	0.001 secondes	1.05 secondes	17.9 minutes	12.7 jours	35.7 ans	$4x10^{13}$ siècles
3^n	0.059 secondes	58 minutes	6.5 ans	3855 siècles	$2x10^8$ siècles	$1.6x10^{31}$ siècles
n!	3.62 secondes	77 siècles	$8.4x10^{15}$ siècles	$2.6x10^{31}$ siècles	$9.6x10^{47}$ siècles	$3x10^{141}$ siècles

Figure 1 : Temps de calcul en fonction de la complexité

Ce tableau montre très nettement que l'ordre de la complexité est la caractéristique principale pour le calcul du temps de réponse d'un problème, en effet quelles que soient

les techniques d'optimisation utilisées et quelle que soit la puissance de la machine, il est impossible d'exécuter dans son intégralité un programme de complexité n! pour une instance supérieure à 20.

Par ailleurs, il est souvent difficile de connaître avec précision la complexité de l'algorithme. Le problème n'est pas tant le calcul de la complexité en pire cas mais plutôt l'estimation des paramètres dont elle dépend.

Ainsi, il arrive fréquemment que, même avec l'implémentation la plus efficace possible, le problème soit trop complexe pour être résolu de façon optimale dans le temps imparti. Dans ce cas, il faut diminuer la complexité de la technique mise en œuvre. En réalisant une telle diminution, le résultat sera obtenu beaucoup plus rapidement, mais ceci s'accompagne bien évidemment d'une baisse de la qualité de celui-ci. En effet, la réponse produite ne sera plus optimale mais une réponse approchée par des techniques approximatives.

L'utilisation de telles approximations s'avère très intéressante, puisqu'un résultat dont la qualité est dégradée mais qui est obtenu avant la date limite est de loin préférable à un résultat optimal mais tardif, voire impossible à obtenir dans un délai raisonnable. Pour cela, le recours aux techniques d'intelligence artificielle s'avère intéressant, c'est l'objet du chapitre suivant.

1.2. Intelligence artificielle et temps réel

En intelligence artificielle, un système informatique est très souvent décrit comme étant un agent autonome et immergé dans un environnement. Cette définition pose le problème des rapports entre un agent et son environnement physique. En effet, la vision statique supposant que le monde n'évolue pas lorsque l'agent raisonne, est simplificatrice mais n'en est pas moins trop irréaliste car un environnement dynamique engendre d'importantes modifications de fonctionnement. Dès lors, de nombreux problèmes apparaissent :

- Comment réagir face à un événement extérieur imprévu ?
- Comment intégrer des faits nouveaux indisponibles au début du raisonnement ?
- Comment privilégier les actions et les hypothèses prioritaires ?
- Comment être suffisamment rapide pour fonctionner en temps réel ?
- Comment assurer une réponse de qualité en respectant une échéance ?

Ainsi, un agent œuvrant dans un environnement dynamique doit prendre en compte la notion de temps de réponse. Ses actions sont contraintes par le temps et ses performances sont jugées par le rapport qualité sur temps. Un tel agent doit réagir vite tout en pouvant se concentrer pendant un longue période sur une action urgente. Ceci peut s'avérer contradictoire. Aussi, il faut trouver un juste compromis entre la fréquence des mises à jour et le temps de traitement des nouvelles données. En effet, le traitement doit être suffisamment long pour produire une réponse de bonne qualité, mais doit également être suffisamment court pour que les données qui ont déclenché ce traitement restent valides. D'où la nécessité de dégager des qualités essentielles telles que la rapidité, l'adaptabilité, la ponctualité et une capacité de focalisation. Pour faire face à ces nombreuses exigences, l'agent doit utiliser plusieurs techniques, notamment les techniques d'actions réflexes à la manière des agents purement réactifs, mais aussi des techniques d'approximation afin de pouvoir garantir un temps de réponse. Le développement d'agents ayant de telles possibilités au sein d'un système multi-agents permet d'étendre considérablement leurs fonctionnalités.

Cette thèse se focalise plus spécialement sur le respect des échéances, à savoir quel comportement un agent doit adopter afin de pouvoir fournir une réponse satisfaisante tout en étant limité dans son temps d'exécution. Dans ce contexte, résoudre un problème de façon optimale n'est plus alors synonyme de trouver la réponse ayant la meilleure qualité mais plutôt de trouver la meilleure réponse possible en fonction du laps de temps accordé. On parle alors d'optimalité contrainte (*bounded optimality*) [Russell 95].

Afin de décider quelle est la meilleure réponse à fournir selon le temps alloué à l'exécution, il est nécessaire d'étudier les performances des différents algorithmes mis en œuvre. Les courbes de la figure 2 schématisent l'évolution de la qualité du résultat d'un algorithme en fonction du temps qui lui est accordé. Il s'agit d'étudier l'évolution de l'efficacité de l'algorithme au cours du temps pour obtenir ainsi une fonction, appelée dans la littérature *performance profile*, qui possède différentes allures selon le type d'algorithme utilisé :

- **Classique**. Les techniques utilisées classiquement en informatique produisent un résultat optimal après une certaine durée. L'évolution de la qualité de la réponse de ce genre d'algorithme est simple puisque seul le résultat final est exploité, aucun résultat intermédiaire de moindre qualité n'est directement disponible au cours de la résolution. Si l'algorithme n'est pas une méthode approchée, le résultat final possède la qualité optimale. La courbe de performance est représentée sur la figure 2 par la courbe tracée en pointillés.

- **Progressif**. Beaucoup de programmes sont conçus sous forme d'itérations qui améliorent progressivement un résultat intermédiaire. Ainsi, en sauvegardant les résultats successivement obtenus au cours des itérations, l'algorithme dispose à

tout instant d'une réponse dont la qualité augmente avec le temps de calcul. Le traitement peut être interrompu à tout moment, la réponse obtenue par la dernière amélioration est disponible. La qualité du résultat évolue donc selon une fonction en escalier, où chaque marche correspond à une fin d'itération et, par conséquent, une amélioration du résultat. La courbe correspondante est visualisée sur la figure 2 par un trait continu.

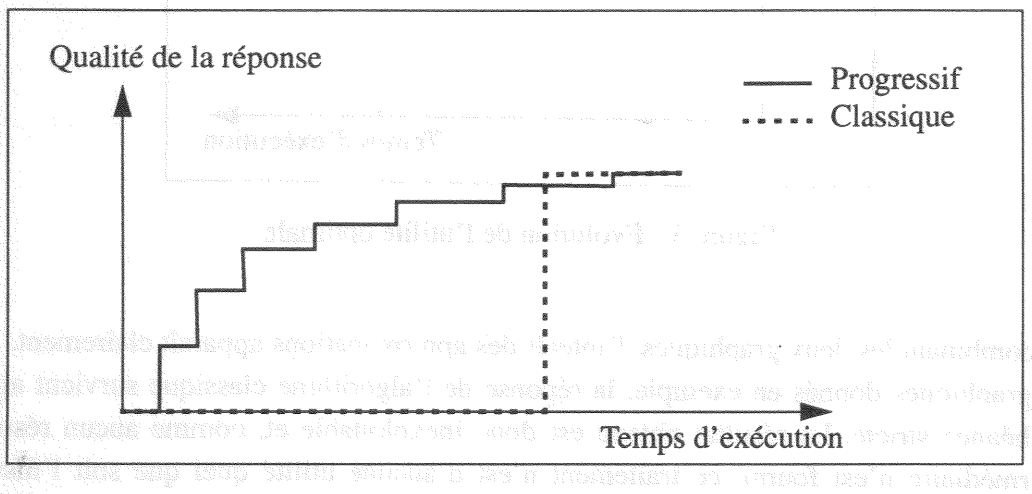

Figure 2 : Evolution de la qualité

Ces deux types d'algorithmes peuvent être considérés comme identiques dans la mesure où le résultat final est de même qualité pour chacun d'eux. Cependant, si l'on intègre la notion de temps de réponse, il en est tout autrement. En effet, la date de disponibilité de la réponse peut s'avérer très importante dans un environnement dynamique. Les causes peuvent être multiples :

- Les données traitées deviennent de moins en moins fiables au cours du temps.
- Le système peut être bloqué en attendant la réponse.
- L'attente de la réponse peut avoir un coût.
- La requête peut posséder une échéance stricte et toute réponse survenant après cette date est totalement inutile.

L'attente est donc pénalisante pour le système. Ainsi, une réponse optimale ne possède pas la même utilité selon la date à laquelle elle est disponible. L'évolution de l'utilité de la réponse optimale au cours du temps peut être grossièrement représenté par la courbe de la figure 3.

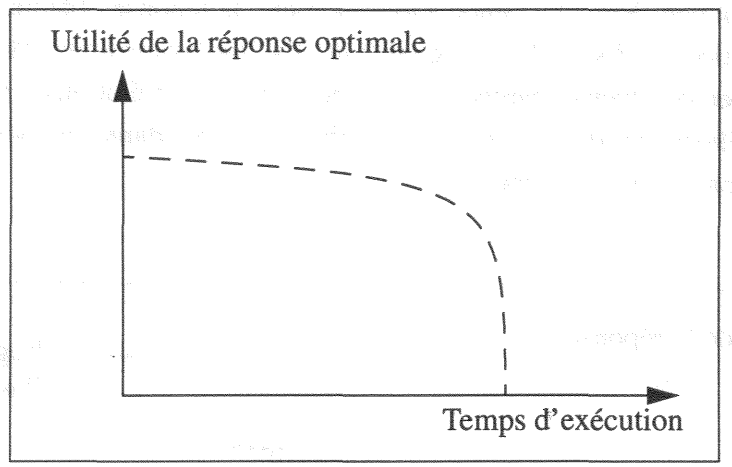

Figure 3 : Evolution de l'utilité optimale

En combinant les deux graphiques, l'intérêt des approximations apparaît clairement. Sur les graphiques donnés en exemple, la réponse de l'algorithme classique survient après l'échéance stricte. Le résultat obtenu est donc inexploitable et, comme aucun résultat intermédiaire n'est fourni, ce traitement n'est d'aucune utilité quel que soit l'instant considéré. Par contre, dans le cas où l'on utilise un algorithme progressif, des résultats intermédiaires sont fournis. Ainsi, même si le résultat optimal qui serait obtenu en exécutant totalement l'algorithme est inutilisable dans notre cas, il est possible de rendre le traitement efficace en le stoppant lorsque la qualité du résultat est suffisante et l'utilité d'une réponse intéressante.

La figure 4 reprend les courbes d'évolution de la qualité de la réponse (voir figure 2) et y intègre le coût de l'attente (voir figure 3) afin de calculer l'utilité de la réponse.

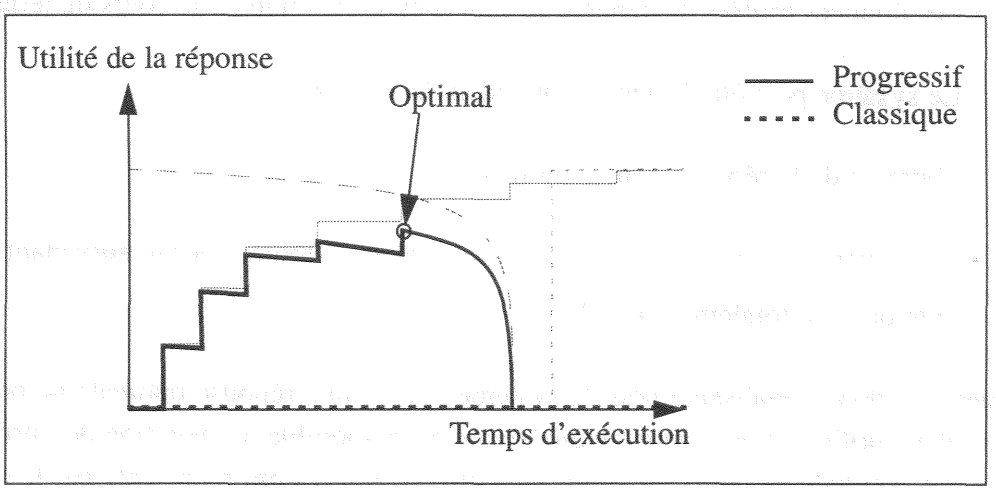

Figure 4 : Evolution de l'utilité

Ainsi, on s'aperçoit que les algorithmes progressifs permettent de maximiser l'utilité de la réponse en réalisant un compromis entre sa durée d'obtention et sa qualité. Le temps d'exécution devient un paramètre à fixer de manière à obtenir la réponse d'utilité maximale. Cette optimisation en fonction du temps d'exécution n'est pas possible dans le cas d'un algorithme classique.

L'objectif de cette thèse est d'étudier les différentes façons de mettre en place des algorithmes progressifs afin de réaliser des compromis entre le coût lié à l'exécution et la qualité du résultat obtenu. Ceci impose :

- L'utilisation d'algorithmes progressifs fournissant des résultats intermédiaires à des intervalles de temps réguliers.

- Une bonne connaissance des performances des algorithmes afin de pouvoir évaluer leurs durées et la qualité des réponses produites.

- Un module de contrôle qui intègre à la fois les notions de coût et de date limite, liées à l'événement et les notions de performance des différents algorithmes. Ceci permet de réaliser l'exécution ou la séquence d'exécutions donnant le meilleur résultat possible.

Ces trois étapes vont permettre la construction d'un système capable d'adapter dynamiquement son comportement selon des contraintes de ressource imposées par l'extérieur. Ce fonctionnement sera particulièrement intéressant dans un cadre temps réel.

2. Algorithmes progressifs

Raisonner efficacement dans un système temps réel n'est pas chose facile. En effet, un même problème ne doit pas être résolu de la même façon selon la charge globale du système. Il faut tantôt être rapide pour respecter une échéance proche et tantôt prendre son temps pour fournir une réponse de qualité. Pour cela, il est nécessaire d'avoir recours à la notion de progressivité.

Le système doit être en mesure de réagir rapidement, mais doit également pouvoir affiner son raisonnement en fonction du temps disponible. Ainsi, il doit appréhender à tout instant la durée des tâches à exécuter ainsi que leur utilité pour l'accomplissement du but à atteindre.

Pour un tel raisonnement, la manipulation de fonctions spéciales pouvant réaliser des compromis entre les durées d'exécution des traitements et la qualité des résultats obtenus est un atout important. Ces fonctions peuvent être regroupées en plusieurs grandes classes : les algorithmes interruptibles, les algorithmes par contrats, les algorithmes à méthodes multiples, le raisonnement progressif. L'étude de ces différentes classes fait l'objet de ce chapitre.

2.1. Algorithmes interruptibles

C'est Dean qui le premier a utilisé le terme *anytime* lors de ses travaux sur les applications dépendant du temps [Dean 88]. Une procédure *anytime* interruptible est un algorithme qui garantit la production d'une réponse quel que soit l'instant de son interruption, avec un résultat qui s'améliore à chaque itération.

2.1.1. Principe

Beaucoup d'algorithmes sont *anytime* par essence. La seule véritable modification consiste à sauvegarder des résultats intermédiaires. D'un point de vue externe, l'algorithme réalise une suite d'affinements sur un premier résultat grossier obtenu rapidement.

Par contre, pour les algorithmes n'étant pas naturellement *anytime*, cette propriété doit être ajoutée en utilisant la notion d'approximation. Il faut en fait chercher un moyen de dégrader le résultat afin que son obtention soit plus rapide. Plusieurs critères peuvent entrer en jeu :

- **Certitude**. C'est le reflet du degré de confiance accordé à l'exactitude de la réponse. L'approximation consiste ici à trouver une solution complète rapidement, mais dont la correction est incertaine. Les affinements successifs confirmeront que cette réponse à des chances d'être juste ou trouveront une autre réponse dont la probabilité d'exactitude est plus forte.

- **Précision**. Dans ce cas, les réponses produites sont approximatives, mais relativement proches de la réponse exacte. Les améliorations successives réduisent progressivement la distance entre ces deux réponses.

- **Granularité**. Un troisième critère d'approximation possible est le niveau de détail de la réponse. Il est en effet possible de considérer les données en différents paquets selon le degré de généralité des connaissances qu'elles véhiculent. Dans ce cas, la solution obtenue rapidement correspond à l'optimale mais sa description est beaucoup plus grossière. Elle sera détaillée au cours des itérations.

2.1.2. Exemples

Etre *anytime* n'est pas plus révolutionnaire que faire de la prose, puisque la plupart des algorithmes itératifs sont par nature *anytime*. Il suffit pour cela que le calcul du résultat se fasse de façon incrémentale. Ainsi, en stockant le résultat après chaque nouvelle itération et en le considérant comme un résultat intermédiaire, l'algorithme produit à chaque étape une solution exploitable, celle-ci s'améliorant au cours du temps. Ceci correspond tout à fait à la définition d'un algorithme *anytime*. Des exemples de techniques se conformant aux caractéristiques désirées existent ainsi dans de nombreux domaines :

- *Approximation numérique*. En analyse numérique, de nombreuses méthodes consistent à itérer sur une série de calculs afin d'approcher de la solution. Par exemple, la méthode des éléments finis pour approcher une fonction continue [Varga 62].

- *Algorithmes probabilistes*. Les Algorithmes *Monte-Carlo* augmentent la confiance de leur résultat en répétant la même itération au hasard dans l'espace de recherche [Harel 87].

- *Programmation dynamique*. Cette méthode repose sur le *principe d'optimalité* défini par Bellman [Bellman 57]. Elle consiste à immerger le problème initial dans un problème plus général, à appliquer le principe d'optimalité sur la formulation du problème puis à le résoudre de façon séquentielle et enfin à oublier les étapes intermédiaires non optimales afin de ne conserver finalement que la solution optimale. L'intérêt dans notre cas est que le problème est décomposé en petits sous-problèmes résolus séquentiellement en construisant

ainsi la solution de façon incrémentale. Il est possible de définir des heuristiques permettant à tout instant d'exploiter les résultats intermédiaires des modules déjà exécutés afin de construire une solution approchée disponible immédiatement. L'utilisation de telles heuristiques permet ainsi de rendre *anytime* les procédures de programmation dynamique [Boddy 91b].

- *Procédures par séparation évaluation* [Roy 70]. Tout comme la programmation dynamique, cette technique consiste en la recherche d'une solution exacte au problème. Comme son nom l'indique, la méthode consiste à effectuer récursivement deux étapes : la séparation (l'espace de recherche est divisé en sous-régions), puis l'évaluation (on calcule une borne supérieure de la solution optimale pour chacune de ses régions). La résolution s'effectue ainsi par une recherche en profondeur de type *meilleur d'abord* (qui peut éventuellement être conduite par des heuristiques) dans l'espace de recherche afin de trouver rapidement un résultat de bonne qualité afin d'éviter l'exploration de sous-régions ayant une borne supérieure de la qualité du résultat inférieure au meilleur résultat actuellement disponible. La méthode consiste ensuite à examiner les sous-régions potentiellement intéressantes pour rechercher des solutions qui possèdent une qualité supérieure au résultat courant. Cette méthode est ainsi intrinsèquement *anytime* puisqu'elle possèdent à tout moment le meilleur résultat compte tenu de l'espace exploré et donc du temps consacré à la résolution.

- *Algorithmes à voisinage simple*. Dans cette technique, une solution est sélectionnée au hasard dans l'espace de recherche, puis certaines heuristiques sont activées afin de sélectionner une solution voisine de meilleure qualité ou qui permette de se rapprocher de la solution optimale. Cependant, dans de nombreux cas, l'algorithme stoppe sur un optimum local. Un exemple connu est la résolution du problème du voyageur de commerce par l'algorithme de 2-opt [Lin 73]. De nombreuses techniques utilisent ce principe : le recuit simulé (voir partie 2 paragraphe 3.5.1 page 79), tabou (voir partie 2 paragraphe 3.5.2 page 79), la descente du gradient...

- *Algorithmes génétiques* (voir partie 2 paragraphe 3.5.3 page 80). Ce sont des algorithmes à voisinages multiples. Ils fonctionnent de la même manière que ceux à voisinage simple, en sélectionnant des solutions aléatoirement puis en recherchant des voisins de meilleur qualité. La différence concerne l'utilisation d'une population de solutions. Des solutions sont sélectionnées au hasard ou au moyen d'heuristiques, puis l'algorithme recombine les solutions afin d'en fournir d'autres. Il sélectionne ensuite les meilleures et réitère l'opération jusqu'à l'obtention d'une solution de qualité suffisante. Il réalise ainsi une suite d'affinements successifs afin de tendre vers une solution optimale [Golberg 89].

2.2. Algorithmes par contrats

Beaucoup de traitements peuvent également être réalisés avec plus ou moins de précision sans pour autant pouvoir être interrompus à tout instant. Par exemple, un chargé de recherche désire écrire un rapport sur ses activités. Il peut le faire de façon très rapide en quelques lignes ou de manière très complète en plusieurs centaines de pages. Entre ces deux extrémités, il est possible d'imaginer de nombreux formats intermédiaires, la taille du document produit étant fonction du temps accordé à la rédaction. Il s'agit d'une tâche progressive, puisqu'elle peut être réalisée selon différents temps d'exécution correspondant à différentes qualités, mais elle n'est pas interruptible pour autant : l'arrêt de la rédaction en cours ne produira pas de rapport satisfaisant. La qualité du résultat est choisie avant le début de l'exécution de la tâche et aucun résultat intermédiaire ne sera fourni avant la fin du traitement.

2.2.1. Principe

Ces algorithmes ont un résultat dont la qualité augmente en fonction du temps qui leur est consacré et peuvent s'exécuter pendant une durée choisie *a priori* : les algorithmes par contrats sont considérés de la même façon que les algorithmes interruptibles par Dean et Boddy, ce qui se justifie par l'existence d'une fonction de performance du même type dans les deux cas. Cependant, Russell [Russell 91] effectue une différence entre ces deux types d'algorithmes. Un algorithme par contrats peut également s'exécuter selon une échelle de temps continue en fonction d'un paramètre, mais le fait de fixer ce dernier avant le début de l'exécution limite fortement les possibilités de contrôle dynamique. En effet, l'arrêt de cet algorithme avant que son exécution n'arrive à terme peut ne pas fournir de réponse, tandis qu'un algorithme interruptible n'a pas *a priori* de date fixée pour la production d'une réponse.

2.2.2. Exemples

En intelligence artificielle, de nombreux problèmes sont traités par des recherches heuristiques. La technique consiste à modéliser les situations par des états et les actions par des transitions entre ces états. Ainsi, l'objectif est de faire passer le système d'un état initial vers un état final en utilisant les transitions possibles. Pour cela, il est possible de générer le graphe complet de tous les états accessibles afin que le problème se résume à une recherche de chemin dans un graphe. Cette recherche est facilitée par l'utilisation d'heuristiques qui évaluent la distance entre un état donné et un état but. Ainsi le parcours peut se faire selon des techniques du type meilleur d'abord afin d'obtenir une solution rapide.

La recherche d'un chemin dans un graphe peut se faire de façon optimale par un algorithme qui examine tous les états de celui-ci, mais la complexité de cet algorithme augmente exponentiellement en temps et en espace lorsque l'espace de recherche s'agrandit. Ainsi, Korf a défini un algorithme IDA* [Korf 85] qui réduit l'espace de recherche et par conséquent la complexité au moyen d'élagage. L'algorithme *minimim lookahead search* [Korf 87] reprend les mêmes idées en limitant la profondeur de recherche. Ainsi, chaque décision est prise en fonction d'un sous ensemble de l'espace de recherche, ce qui permet un gain de temps en échange d'une diminution de la qualité du résultat produit. Choisir la profondeur de la recherche correspond bien à fixer un contrat, c'est-à-dire un compromis entre la durée d'exécution et la qualité de la solution.

L'algorithme RTA* [Korf 88] présente également l'intéressante propriété d'être *anytime* par contrats. Il gère l'avancée en même temps que la planification. En effet, le chemin est calculé de la même manière que le *minimim lookahead search* puis il est validé, c'est-à-dire que l'on répercute directement l'action correspondant à ce choix sur l'environnement. La planification est alors relancée depuis ce nouvel état. Il n'y a pas alors de retours en arrière à proprement parlé, puisque la remise en cause d'une décision concernant un chemin est gérée automatiquement en considérant l'état précédent comme une solution envisageable au même titre que les états suivants possibles. Ainsi, l'algorithme RTA* peut être réalisé avec plus ou moins de rapidité selon la profondeur de recherche choisie et il se classe dans les algorithmes par contrats.

Les travaux de Shekar portent également sur la recherche d'un chemin dans un graphe. Le but est d'optimiser le temps de réponse d'un tel algorithme, c'est-à-dire de faire un effort de planification uniquement si le gain en temps d'exécution est supérieur au temps nécessaire à la planification. La première réalisation, NORA [Shekar 89], a été adaptée à un environnement dynamique pour donner naissance à DYNORA [Hamidzadeh 91] puis à DYNORA II [Hamidzadeh 92]. Ce dernier algorithme consiste à rechercher le meilleur fils pour chaque nœud en regardant à une certaine profondeur, qui n'est pas fixée *a priori*, mais qui est prolongée tant que le rapport temps d'exécution sur temps de planification ne dépasse pas un certain seuil fixé en début de recherche. De la même façon que le *minimim lookahead search* et que RTA*, DYNORA et DYNORA II peuvent être classés parmi les algorithmes par contrats puisqu'ils possèdent un paramètre (le seuil) qui réalise un compromis entre la durée de l'exécution et la qualité du résultat.

2.3. Algorithmes à méthodes multiples

Le but des approximations est de diminuer le temps de calcul, en s'accordant en contrepartie le droit de fournir un résultat de moins bonne qualité. Pour cela, une technique efficace est de trouver des algorithmes approchés qui ont une plus faible complexité que l'algorithme optimal en fournissant un résultat de plus faible qualité

mais tout de même exploitable. D'autres techniques consistent à jouer sur la généralité des données traitées.

V. Lesser [Lesser 88] a répertorié trois types d'approximations permettant une dégradation de la qualité de la solution obtenue au profit d'un gain en temps d'exécution de l'algorithme :

- *Approximation de la recherche ou stratégie.* Cela consiste à utiliser une technique de profondeur d'abord afin d'éliminer les solutions concurrentes et les données redondantes.
- *Approximation des données.* Cela consiste à occulter les données moins importantes et à traiter les données ayant des caractéristiques proches par paquets.
- *Approximation des connaissances.* Cela consiste à agglomérer les règles par paquets afin de les remplacer par un traitement réalisé en une seule inférence.

Ces approximations vont permettre de définir différentes méthodes résolvant le même problème afin d'utiliser la progressivité quand le temps de réponse est faible.

2.3.1. Principe

Les méthodes obtenues par les approximations sont pour la plupart issues d'une expertise réalisée sur plusieurs niveaux selon la proximité de l'échéance de la tâche à prendre en compte. Elles demandent bien sûr plus de travail lors de l'extraction de l'expertise, mais elles apparaissent comme les plus naturelles pour le contrôle d'un système physique temps réel. Elles reposent sur le fait que beaucoup d'événements, qui ne peuvent être traités de manière itérative, ont un nombre limité de techniques de résolutions possibles, à savoir quelques procédures de qualités et de durées différentes.

Les différentes méthodes peuvent être concurrentes ou coopératives. Dans le premier cas, chacune d'elles calculent une nouvelle réponse; tandis que dans le second cas, les nouvelles méthodes réutilisent les résultats précédents. Il est ainsi possible là aussi de retrouver le dilemme entre algorithme interruptible et algorithme par contrats, mais le faible nombre d'étapes justifie l'absence de distinction.

Les affinements successifs permettent donc de traiter un problème de manière progressive. Les paragraphes suivants présentent quelques exemples de méthodes basées sur cette technique.

2.3.2. Calcul imprécis

Pour des problèmes d'ordonnancement en temps réel, Liu [Liu 91] propose le paradigme de l'*imprecise computation*. Il s'agit de décomposer les traitements en plusieurs parties de façon à faire apparaître le résultat progressivement.

La première étape du traitement, appelée partie obligatoire (*mandatory part*), fournit une réponse acceptable de faible qualité. Cette étape n'est pas interruptible, mais possède une durée d'exécution peu importante. Ainsi, cela permet d'obtenir rapidement une solution minimale grossière afin de garantir une réponse dans le temps imparti.

Les autres étapes, appelées parties optionnelles (optional part), constituent une suite d'affinements qui améliorent successivement la qualité du résultat. Chacune de ses étapes peut donc être interrompu à tout instant, le résultat de l'étape précédente est disponible.

Ces algorithmes sont très voisins des algorithmes *anytime*, la différence essentielle est que le nombre d'étapes d'affinement est souvent réduit. De plus ces dernières ont en général une durée plus importante qu'une itération d'un algorithme *anytime*.

2.4. Raisonnement à profondeur variable

Face à un problème complexe n'ayant pas de solution reconnue efficace, l'intelligence artificielle apporte souvent une vue différente en simulant le raisonnement humain. C'est la démarche de Kayser lorsqu'il parle de raisonnement à profondeur variable [Kayser 88]. L'homme a accès à tout instant à une immense quantité de connaissances, celles-ci pouvant être très générales («Nancy est une ville») ou très précises («A Nancy, la place Stanislas est piétonne de 19h30 à 2h durant l'été»). Suivant le raisonnement mis en œuvre, l'homme décide du niveau de précision, ou grain qu'il utilise. L'humain filtre ainsi les données et n'utilise pour son raisonnement que celles dont le niveau de généralité est jugé pertinent.

Le système mis au point par Kayser imite cette démarche. Il raisonne sur un grain élevé de la connaissance puis, si le résultat obtenu est insuffisant, le grain est diminué afin de tenir un raisonnement sur des faits plus précis. Pour cela, des règles d'expansions («Place(x) -> existe (y), ville (y), est_situé (x,y)») sont associées aux concepts. Lorsque le résultat obtenu est trop grossier, il est alors possible d'utiliser ces règles afin de travailler sur une granularité de connaissance plus fine et donc de fournir un résultat plus précis. Les objets ont donc ainsi une représentation progressive qui sera utilisée en fonction du raisonnement tenu.

2.5. Raisonnement Progressif : GREAT

GREAT est un système temps réel destiné à résoudre un but en garantissant un temps de réponse [Mouaddib 92]. Pour cela, il définit deux modes de raisonnement. Un premier mode appelé «grossier» consiste à affaiblir les contraintes du problème en se fondant sur les mécanismes d'approximation décrit par Lesser (voir paragraphe 2.3.). Dans un deuxième temps, le système affine la solution en renforçant les contraintes petit à petit. Cette technique construit ainsi une solution rapide tout en profitant du temps imparti pour l'affiner et donner une solution de meilleure qualité.

Les connaissances sont stockées sous forme de concepts, qui sont des vecteurs d'attributs, chaque attribut étant défini par une valeur, une granularité et une plausibilité. La précision d'un contexte permet de calculer la valeur du grain de chaque attribut. En effet, selon le contexte du raisonnement, une information peut être plus ou moins pertinente et sa granularité sera modifiée. De même, un grain est associé à chaque règle en fonction du grain des concepts contenus dans ses prémisses.

Cette hierarchisation des connaissances autorise le calcul de la qualité d'un concept par la définition de tables d'évaluation. Ces dernières sont composées de trois propriétés : la complétude, la précision et la certitude. La complétude est mesurée par le nombre d'attributs du concept, la précision par la granularité des attributs et la certitude par la plausibilité des valeurs des attributs. GREAT construit ainsi une bibliothèque de tables d'évaluations dans laquelle il choisit, en fonction du niveau et du contexte, la table jugeant la qualité d'une solution.

GREAT est activé par l'arrivée d'une tâche qui doit être effectuée dans un délai spécifié. Il reçoit aussi des variables de contexte qui vont permettre de se focaliser sur des concepts et des attributs précis. Dans un deuxième temps, le distributeur (*dispatcher*) (voir figure 5) classe les attributs par région en calculant leur granularité. Ce classement définit une hiérarchie, chaque niveau contenant des règles et des attributs de grains différents. Les régions sont ensuite filtrées selon des critères de certitude et de temps et selon l'élément physique associé au concept, afin de donner la base de faits la plus réduite possible au module de raisonnement.

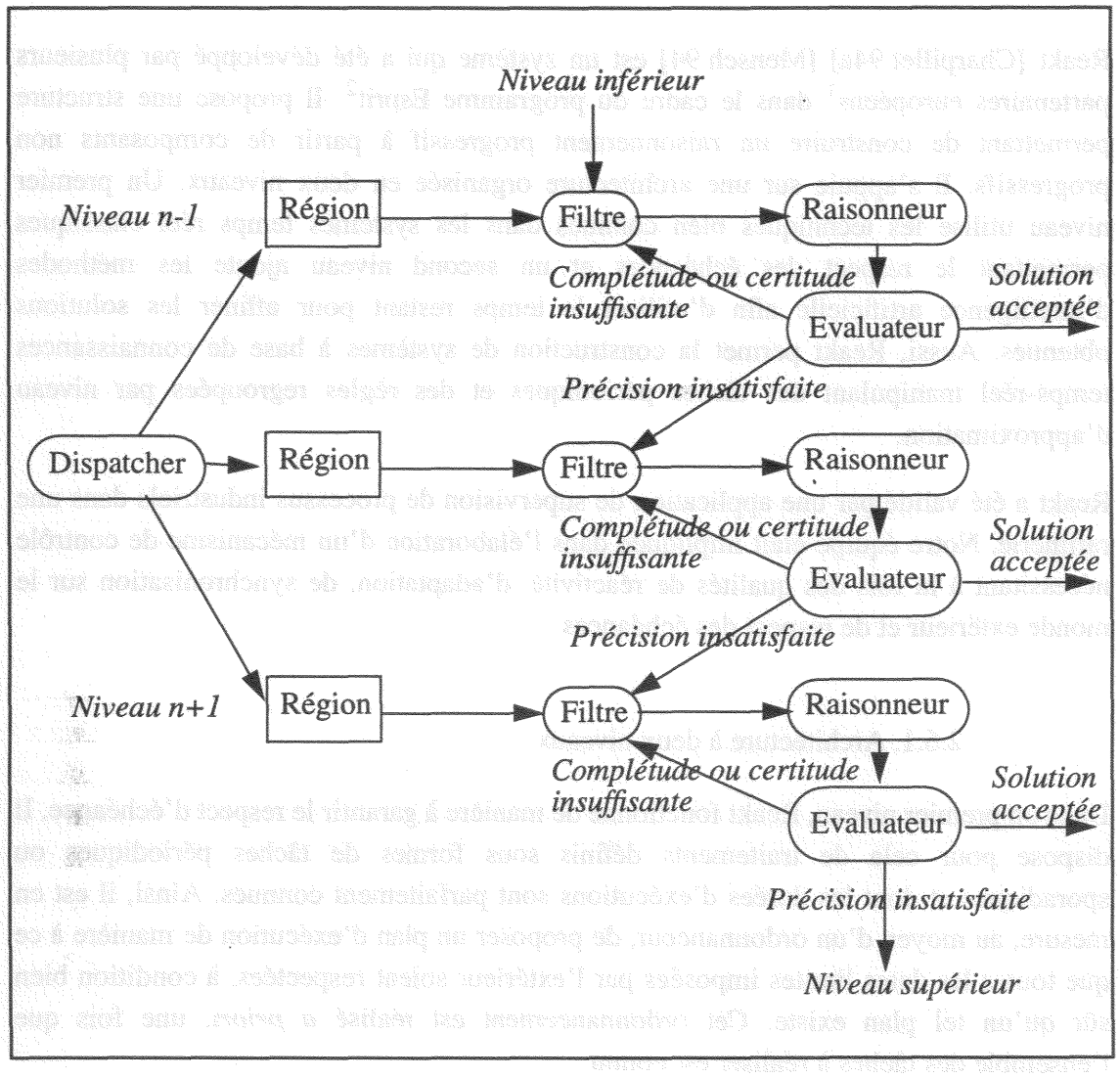

Figure 5 : Fonctionnement de GREAT

Une fois les inférences terminées, un évaluateur calcule la qualité de la solution. Si la solution est incomplète ou incertaine et que le temps imparti n'est pas entièrement écoulé, l'évaluateur active à nouveau le filtre pour qu'il apporte de nouvelles connaissances à la base. Si, par contre, la complétude et la certitude sont satisfaisantes, mais que la précision est insuffisante, c'est le filtre de niveau inférieur qui est activé, à condition bien sûr que le temps imparti ne soit pas écoulé.

Le système GREAT permet donc de tenir un raisonnement progressif qui garantit le respect des échéances, mais il reste limité car il ne prend en compte qu'une seule tâche à la fois. Il a permis de mettre en œuvre des techniques essentielles pour un raisonnement en temps réel, c'est-à-dire la granularité, la focalisation, l'évaluation de la qualité et l'affinement. Ce système a été intégré dans RTSOS, une extension temps réel des sociétés de spécialistes [Mouaddib 92].

2.6. Reakt

Reakt [Charpillet 94a] [Mensch 94] est un système qui a été développé par plusieurs partenaires européens[1] dans le cadre du programme Esprit[2]. Il propose une structure permettant de construire un raisonnement progressif à partir de composants non progressifs. Il s'appuie sur une architecture organisée en deux niveaux. Un premier niveau utilise les techniques bien connues dans les systèmes temps réel classiques permettant le respect des échéances et un second niveau ajoute les méthodes d'intelligence artificielle afin d'utiliser le temps restant pour affiner les solutions obtenues. Aussi, Reakt permet la construction de systèmes à base de connaissances temps-réel manipulant des tâches périodiques et des règles regroupées par niveau d'approximation.

Reakt a été validé par une application de supervision de processus industriels dans une raffinerie. Notre équipe était impliquée dans l'élaboration d'un mécanisme de contrôle nécessitant à la fois des qualités de réactivité, d'adaptation, de synchronisation sur le monde extérieur et de respect des échéances.

2.6.1. Architecture à deux niveaux

Dans un premier niveau, Reakt fonctionne de manière à garantir le respect d'échéance. Il dispose pour cela de traitements définis sous formes de tâches périodiques ou sporadiques et dont les durées d'exécutions sont parfaitement connues. Ainsi, il est en mesure, au moyen d'un ordonnanceur, de proposer un plan d'exécution de manière à ce que toutes les dates limites imposées par l'extérieur soient respectées, à condition bien sûr qu'un tel plan existe. Cet ordonnancement est réalisé *a priori*, une fois que l'ensemble des tâches à réaliser est connu.

Après avoir établi un plan d'exécution, Reakt utilise son deuxième niveau afin d'allouer les plages de temps restant disponibles. L'objectif est de gérer au mieux ce temps pour améliorer les résultats obtenus au premier niveau ou pour réaliser d'autres traitements non contraints par une échéance. Les techniques mises en œuvre sont détaillées dans les paragraphes suivants.

---

1. Les partenaires impliqués dans le projet Reakt sont Thomson-CSF (France), Groupo de Mecánica del Vuelvo (Espagne), Marconi (Angleterre), Etnoteam (Italie), Computas eXpert Systems (Norvège), Syseca (France), Université Politécnica de València (Espagne) et Crin-Inria Lorraine (France).

2. Reakt fait parti du programme Esprit II (Ref. N. 5146 pour la première phase et N. 7805 pour la deuxième phase).

2.6.2. Agent réactif

Reakt peut être utilisé pour définir un agent autonome et indépendant. L'agent obtenu reçoit des buts et génére des plans d'actions afin d'agir de façon autonome jusqu'à leurs résolutions. Il s'agit donc d'un agent cognitif qui peut raisonner seul, mais qui peut également agir en fonction des modifications de son environnement. Pour cela, il est constamment en contact avec celui-ci. Il travaille avec une représentation du monde dans lequel il évolue et réagit à chaque événement qu'il perçoit selon le paradigme stimulus-réponse, à la manière de réflexes associant une action adéquate à la situation rencontrée. Mais, un système purement réactif ne peut être satisfaisant à lui seul car il n'effectue aucune planification des actions futures. En effet, à chaque instant, il évalue la situation extérieure et effectue l'action la plus appropriée à la situation perçue et au but à atteindre. Pour pouvoir effectuer un raisonnement complexe, des actions réflexes sont trop simples. De plus, pour garantir une réponse dans un temps donné, il faut effectuer une planification des actions futures.

La résolution d'un but ne se fait donc pas systématiquement par le déclenchement d'une action simple. En effet, certains cas complexes peuvent parfois nécessiter la réalisation d'une séquence d'actions. Cependant, une suite d'action figée ne permet pas au système de s'adapter aux évolutions extérieures. Ainsi plusieurs possibilités d'enchaînements d'actions vont être proposées pour résoudre un but selon les futures modifications possibles. Le choix final est effectué au cours de l'exécution selon l'évolution de l'environnement. Une réactivité est ainsi défini au niveau des enchaînements. Elle permet de tenir un raisonnement complexe tout en restant attentif à la dynamique du monde.

2.6.3. Modèle de tâche

Les actions simples peuvent être prises en compte par le premier niveau à l'aide de tâches périodiques ou sporadiques. Par contre, pour définir les enchaînements entre les actions, Reakt utilise la notion de plan à la manière de PRS [Ingrand 90] [Ingrand 91]. La résolution d'un but dans un monde dynamique ne peut pas être systématiquement définie par une séquence de résolution fixée *a priori*. La solution adoptée ici consiste à définir un plan modélisant les différentes suites d'actions envisageables, c'est-à-dire un certain nombre de sous-buts à résoudre avec un enchaînement dynamique dépendant de l'état du monde.

Un plan est donc représenté par un arbre de tâches. Chaque tâche correspond à la résolution d'un sous-but et les branches correspondent aux enchaînements entre les sous-buts. Ainsi, il est possible d'affecter des tests aux branches afin d'adapter la séquence. C'est-à-dire que les fils d'un nœud de l'arbre représente l'ensemble des actions qui peuvent suivre l'exécution de ce nœud. A la fin de l'exécution d'une action,

l'action suivante est choisie parmi les successeurs en fonction de la perception du monde extérieur.

Cette structure de données permet l'élaboration d'un plan complexe tout en restant réactif aux évolutions extérieures. Ainsi, avec ce type de plan, Reakt peut tenir un raisonnement complexe, tout en évitant de rester dans une séquence d'actions figées pendant une durée trop longue. Les actions de Reakt sur son environnement suivent donc un plan pré-établi, mais ce plan est défini de telle sorte qu'il s'adapte à la dynamique du monde dans lequel il évolue.

2.6.4. Architecture

Reakt est un outil de développement de systèmes à multi-bases de connaissances fondé sur le modèle de tableau noir à contrôle centralisé (voir figure 6). Il s'agit d'un système où plusieurs agents fonctionnent indépendamment les uns des autres et communiquent à l'aide d'une base de donnée commune, appelée tableau noir, dans laquelle les agents peuvent lire et écrire. La gestion des activations des agents étant soumise à un module de contrôle, le fonctionnement est dicté par des événements associés aux modifications du tableau noir.

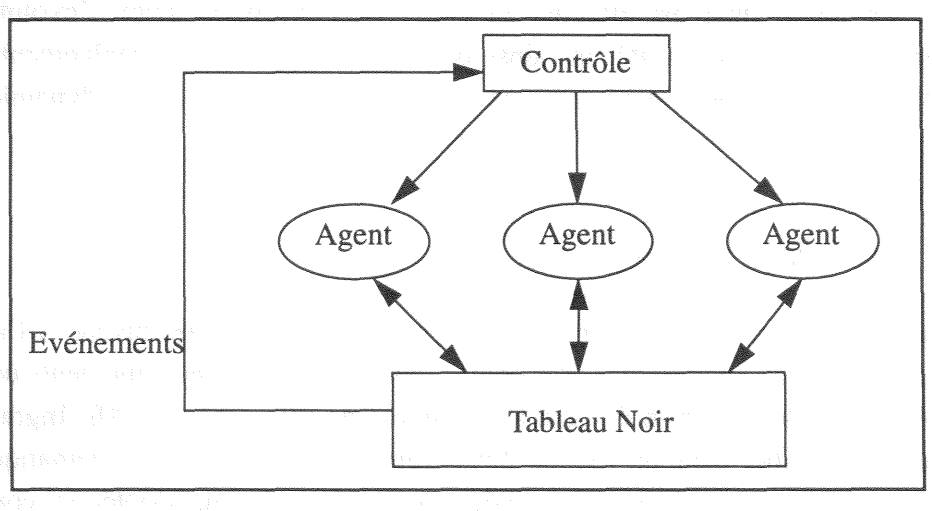

Figure 6 : Architecture à base de tableau noir à contrôle centralisé

Reakt apporte de nombreuses extensions aux systèmes à tableau noir traditionnels afin de permettre un fonctionnement efficace en temps réel. Il intègre à la fois les modules classiques des systèmes temps réel et des modules d'intelligence artificielle qui permettent la gestion d'un raisonnement progressif.

La figure 7 décrit l'architecture de Reakt qui est en fait composée d'un ensemble d'entités communiquant à l'aide de la base de donnée commune suivant le modèle du

tableau noir. On y trouve différents modules:

- des tâches périodiques et des tâches sporadiques qui s'exécutent régulièrement et pour lesquelles la garantie du temps de réponse et la réactivité seront assurées, à condition bien sûr que la charge des tâches ne soit pas supérieure aux capacités d'exécution.

- un module «serveur expert» qui sert de moteur d'inférence et qui crée des actions dont les exécutions doivent se faire de manière progressive afin de pouvoir résoudre au mieux les tâches qui ne sont pas définies de manière périodique.

- un module d'acquisition qui permet de tenir compte des événements extérieurs et, ainsi, de disposer dans le tableau noir d'une représentation du monde qui soit la plus proche possible de la réalité à tout instant.

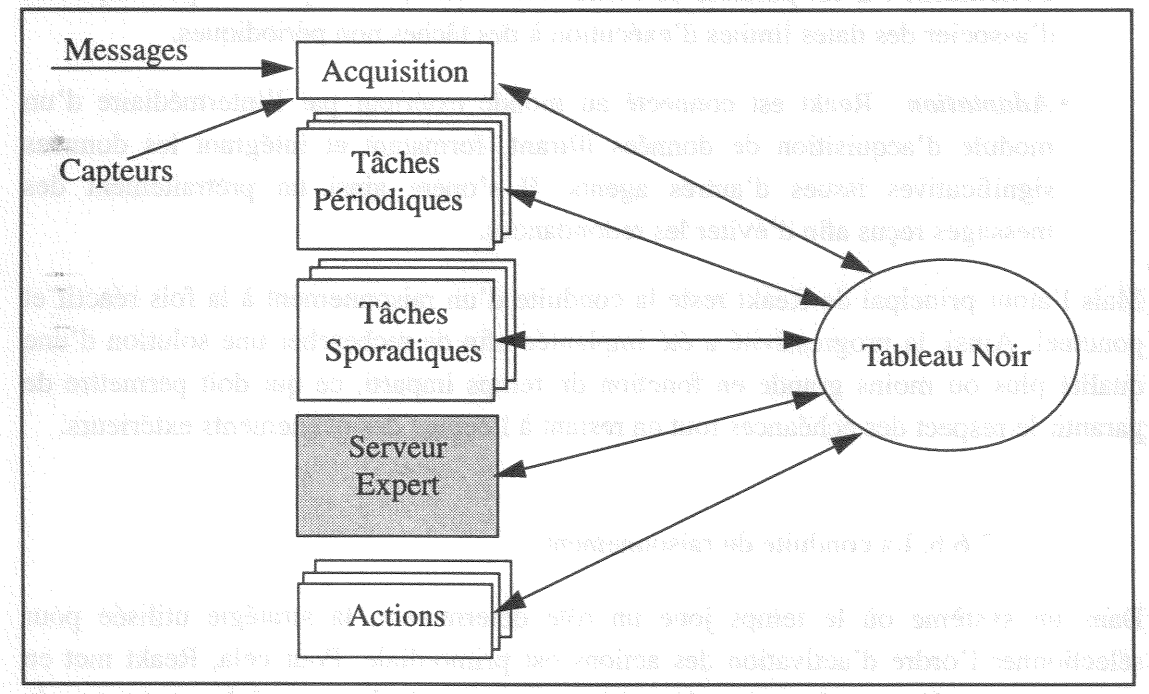

Figure 7 : Architecture de REAKT

2.6.5. Caractéristiques temps réel

De par son architecture et de par la façon dont sont organisées les connaissances, Reakt possède des caractéristiques propices à un fonctionnement temps réel :

- ***Parallélisme*** : l'architecture autorise l'exécution concurrente de plusieurs plans indépendants.

- ***Réactivité*** : les tâches sont interruptibles. Le système peut ainsi suspendre

l'exécution d'un plan moins urgent au profit d'un autre quand un événement intervient.

- ***Performance*** : les connaissances sont exprimées à l'aide de règles d'ordre 0 et d'ordre 1 compilées suivant l'algorithme Treat [Forgy 82], ce qui augmente sensiblement leur efficacité.

- ***Gestion du temps*** : la manipulation de données temporelles permet de tenir un raisonnement prenant en compte le facteur temps et de gérer des relations temporelles entre les données.

- ***Synchronisation*** : Reakt dispose de nombreuses fonctions de manipulation du temps, telles que créer une intention à une heure précise, attendre un certain laps de temps...

- ***Ponctualité*** : il est possible de définir des tâches périodiques ou sporadiques et d'associer des dates limites d'exécution à des tâches non périodiques.

- ***Adaptation*** : Reakt est connecté au monde extérieur par l'intermédiaire d'un module d'acquisition de données filtrant, formatant et intégrant les données significatives issues d'autres agents. Il s'opère ainsi un prétraitement des messages reçus afin d'éviter les redondances.

Mais l'atout principal de Reakt reste la conduite d'un raisonnement à la fois réactif et ponctuel. Ainsi, la progressivité a été implantée afin de rechercher une solution d'une qualité plus ou moins grande en fonction du temps imparti, ce qui doit permettre de garantir le respect des échéances tout en restant à l'écoute des événements extérieurs.

2.6.6. La conduite du raisonnement

Dans un système où le temps joue un rôle déterminant, la stratégie utilisée pour sélectionner l'ordre d'activation des actions est primordiale. Pour cela, Reakt met en œuvre un contrôle complexe dans lequel interviennent plusieurs modules permettant le choix :

- des plans d'actions à effectuer,

- de la séquence d'actions parmi celles proposées par le plan,

- de l'ordre d'exécution des actions lorsque plusieurs plans sont en concurrence

- du temps à consacrer à chacune des actions lorsque celles-ci sont définies de manière progressives.

La conduite du raisonnement (voir figure 8) est définie à l'aide de cinq éléments

principaux:

- Une source de connaissances de contrôle qui évalue la situation courante et sélectionne les traitements à effectuer en fonction des événements qui surviennent. L'objectif est de chercher parmi les plans prédéfinis celui qui est le plus approprié. Pour cela, l'utilisateur fournit un ensemble de règles de contrôle. Ces dernières répertorient tous les événements pouvant parvenir à ce module et leur associent un plan d'action ainsi qu'une partie du tableau noir sur laquelle l'exécution se focalisera.

- Une bibliothèque de plans qui définit les différents traitements possibles correspondant à des réponses à des événements provenant soit du monde extérieur, soit du tableau noir. Ces plans sont en fait des arbres d'enchaînements de sources de connaissances dont le parcours est défini dynamiquement. Cette technique de représentation des connaissances permet d'adapter le traitement aux changements externes, mais pose des difficultés au niveau de l'estimation de la durée du traitement. En effet, si l'arbre est déséquilibré, l'estimation de sa durée est difficile puisque les différents chemins possibles vont présenter un très large éventail de durées d'exécution.

- Un gestionnaire qui définit les actions à exécuter en fonction des plans sélectionnés par la source de connaissance de contrôle. Il examine l'arbre défini par le plan à activer et génére une liste de nœuds qui serviront de modèle d'exécution pour les tâches. Il attend ensuite les résultats d'exécution afin d'envoyer le nœud suivant de l'arbre schématisant le plan.

- Un ordonnanceur qui décide de l'ordre d'exécution des nœuds afin de respecter les contraintes de temps. Il décide également de l'approximation des étapes des plans s'il y a lieu. C'est un module délicat à réaliser car c'est de ses décisions que dépendra la qualité du résultat, la rapidité de la réaction et le respect des échéances.

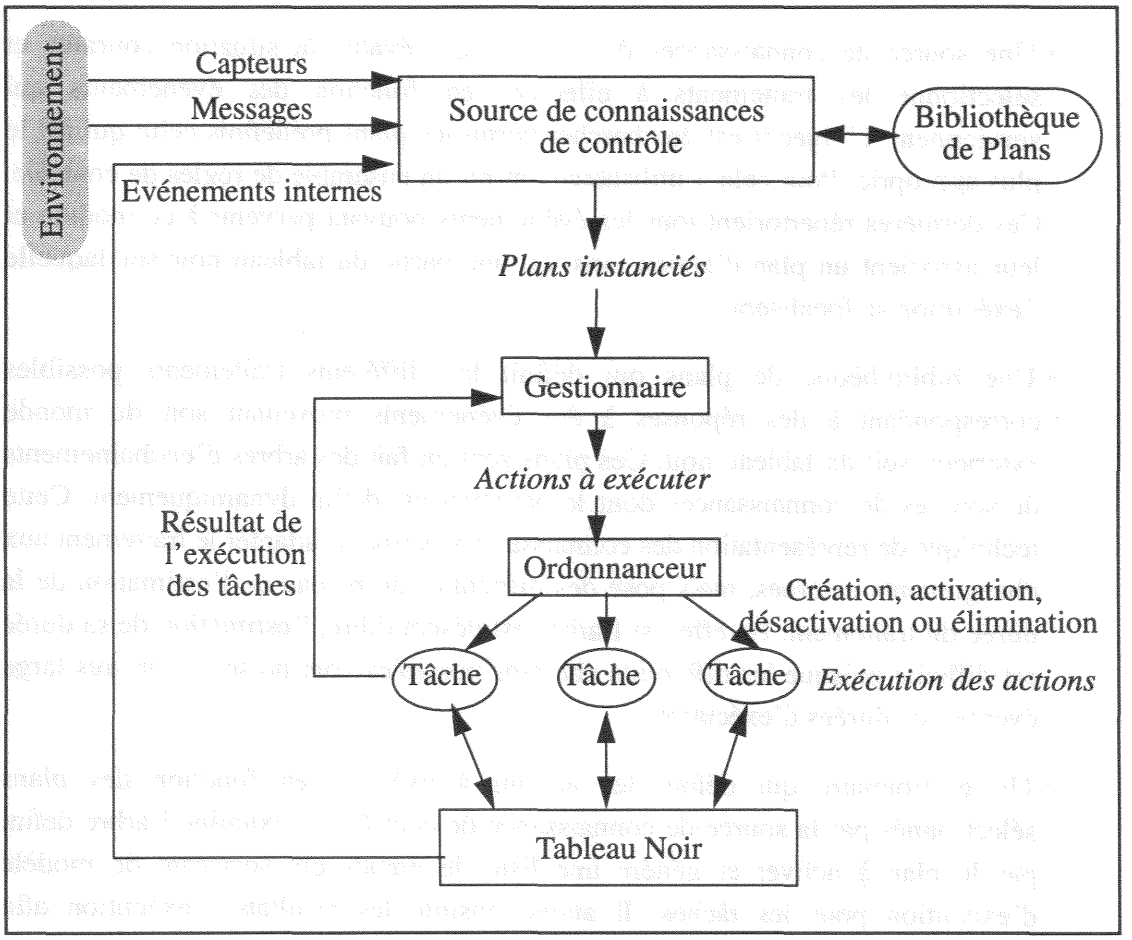

Figure 8 : Contrôle de REAKT

- Un ensemble de tâches qui constituent chacune un système à base de règles. Les règles à déclencher sont extraites de la source de connaissance qu'elles instancient. La base de fait est constituée par le tableau noir ou une partie de celui-ci sur laquelle la tâche se focalise. Lors de leurs interactions avec le tableau noir, les tâches peuvent, par un mécanisme de réflexes, créer des événements qui donneront lieu à d'autres traitements.

2.6.7. Le raisonnement progressif

L'ordonnanceur réalise le raisonnement progressif en sélectionnant le degré d'approximation adéquat : il définit ainsi l'aptitude du système à garantir un temps de réponse.

Afin de rendre le raisonnement progressif possible, il est nécessaire de disposer de données structurées d'une façon adaptée. Ceci exige un effort supplémentaire dès

l'extraction des connaissances expertes. Elles seront organisées par niveaux correspondant à des traitements plus ou moins rapides donnant des résultats de plus ou moins bonne qualité. Les étapes des plans seront ainsi codées par plusieurs sources de connaissances, correspondant aux différents traitements et donc aux différents degrés d'approximation.

Ainsi, chaque source de connaissance est composée d'un paquet de règles minimum à scruter afin de garantir une réponse de qualité minimale. D'autres paquets de règles peuvent être définis afin d'augmenter la précision de la première solution. Il est ainsi possible d'implanter plusieurs niveaux de traitements, chacun fournissant un résultat d'une qualité différente et proportionnelle au temps consacré. Le rôle de l'ordonnanceur est de planifier l'exécution des niveaux afin de garantir le respect de toutes les échéances et d'effectuer le plus d'affinements possibles selon une politique définie qui privilégie les actions les plus importantes et les plus urgentes.

Les choix d'affinements se font à l'aide d'un algorithme d'allocation. La stratégie consiste à affecter les tâches dans leur totalité puis à supprimer les affinements de plus haut niveau jusqu'à ce que toutes les tâches respectent leur date limite d'exécution.

2.6.8. Conclusion

L'architecture Reakt permet la définition d'agents pouvant mener un raisonnement à la fois contraint par le temps et attentif au monde extérieur grâce des mécanismes tels que la focalisation, l'approximation et la réactivité. En effet, l'agent ainsi créé utilise une réactivité à deux niveaux ainsi que des techniques de raisonnement progressif afin de fournir une réponse même si le délai imposé est très court. L'agent fournit donc une solution dont la qualité est fonction du temps consacré à son obtention, ce qui évite dans bien des cas de fournir une réponse de bonne qualité mais trop tardive et donc inutile.

Les fonctionnalités d'un agent modélisé par Reakt sont très intéressantes pour l'intégration d'un tel agent dans un système multi-agents. Reakt est constamment à l'écoute de son environnement et peut également agir sur celui-ci à tout moment. Il est également possible de faire coopérer Reakt avec d'autres agents, qu'ils soient contraints par le temps ou non.

Cependant, cette architecture est une première étape à laquelle de nombreuses extensions peuvent être apportées. En effet, le fonctionnement de l'agent est fortement dépendant de la bibliothèque de plans prédéfinis. On peut donc imaginer un mécanisme d'apprentissage afin d'enrichir dynamiquement cette bibliothèque. D'autre part, la seule technique de coopération avec les agents extérieurs est l'envoi de messages, qui est pris en compte par le contrôle comme tout autre événement. On peut également envisager de faire coopérer deux ou plusieurs agents modélisés par Reakt afin d'obtenir un système multi-agents plus efficace dans un environnement dynamique.

2.7. Conclusion

Plusieurs types d'algorithmes sont ainsi disponibles pour réaliser des traitements progressifs. Ils se différencient principalement par la réutilisation ou non des résultats précédemment acquis. Cette caractéristique est fondamentale pour la définition d'un module de contrôle de ces algorithmes. En effet, en cas d'utilisation des résultats intermédiaires, peu de temps est perdu par l'ajout de la progressivité, à peine le temps de la mise en forme et du stockage afin d'en faire un résultat exploitable. Par contre, dans le cas où les résultats intermédiaires ne sont pas exploités par la suite du traitement (c'est-à-dire qu'ils sont calculés uniquement par sécurité au cas où l'échéance surviendrait brusquement), le temps consacré à leur génération l'est à pure perte si l'échéance est lointaine. Ils se posent alors des problèmes de stratégie, à savoir trouver un compromis entre la fréquence d'obtention des résultats et leur qualité.

Ainsi, les algorithmes interruptibles améliorent sans cesse une réponse grossière acquise rapidement, chaque instant qui leur est consacré est utile. Les algorithmes par contrats quant à eux possèdent différentes exécutions toutes indépendantes, le niveau d'affinement est alors défini *a priori*. Dans ce cas, l'exécution totale d'un contrat rend les exécutions des contrats de plus faible qualité inutiles, et le temps qui leur a été consacré a donc été gaspillé. De même, l'exécution partielle d'un contrat ne produit pas de réponse et s'avère donc sans aucune efficacité pour un temps d'exécution parfois non négligeable. Les méthodes multiples concurrentes, ainsi que la technique de raisonnement progressif posent le même problème de gestion du temps, il faut ainsi des algorithmes d'ordonnancement afin de pouvoir produire un résultat efficace et respectant des échéances fixes.

Par conséquent, chacun de ces algorithmes nécessite un module de supervision différent afin de définir une stratégie d'allocation qui permet de réaliser des compromis entre la fréquence d'obtention de la réponse et sa qualité. Cette stratégie sera définie à partir de connaissances liées à l'application, aux algorithmes et à la puissance de calcul disponible. Les différentes techniques mises en œuvre sont étudiées dans la partie suivante.

3. Supervision

Afin de pouvoir superviser des algorithmes progressifs, il est nécessaire de connaître leurs performances, plus exactement de chiffrer l'augmentation de la qualité de la réponse en fonction du temps consacré à l'exécution. L'objectif est la réalisation d'une fonction associant au résultat un réel indiquant sa complétude et/ou sa précision et/ou sa certitude. Ce chapitre présente les différents types de fonction qu'il est possible d'utiliser ainsi que les principes des mécanismes de conduite à mettre en œuvre pour utiliser au mieux la progressivité une fois les performances connues.

3.1. Mesure de performance

Chaque algorithme est ainsi affecté d'une fonction utilité, appelée *performance profile*, qui exprime l'espérance de la qualité obtenue en fonction du temps consacré à l'exécution. Russell [Russell 91] a défini plusieurs types de fonctions selon le critère mesuré :

- **La distribution de probabilité** (*Performance Distribution Profile*). A chaque instant, on fait correspondre l'ensemble des qualités possibles du résultat ainsi que leur probabilité d'apparition. C'est la façon la plus complète de décrire les performances de l'algorithme, mais elle demande une très bonne connaissance des temps de calcul et la qualité de la réponse. Ce n'est malheureusement pas toujours réalisable dans le cas d'algorithmes non déterministes.

- **L'espérance** (*Expected Performance Profile*). Une façon moins précise mais plus simple de mesurer les performances de l'algorithme consiste à les estimer en moyenne. L'espérance du temps et de la qualité de la réponse servent alors de bases pour caractériser le traitement dans ce cas. Cette estimation comporte bien évidemment moins d'informations qu'une distribution de probabilité, mais sera beaucoup plus aisée à obtenir.

- **Les valeurs extrêmes** (*Perfomance Interval Profile*). Une troisième façon de mesurer les performances consiste à les estimer pour la situation la plus favorable et pour la situation la plus délicate. Les résultats obtenus constituent alors des bornes supérieures et inférieures de l'ensemble des performances possibles. Ces informations peuvent être très intéressantes dans le cas où les variances de la qualité et du temps de réponse sont faibles. Elles peuvent être très imprécises dans le cas contraire.

Grâce à ces fonctions, un module de méta-raisonnement peut réaliser la conduite des exécutions et décider du temps à accorder aux différentes tâches afin de tenter de produire un résultat global ayant une qualité optimale. Comme le montre les paragraphes suivants, la définition de ce module est bien évidemment lié aux types d'algorithmes manipulés.

3.2. Algorithmes interruptibles

Les algorithmes *anytime* possèdent des caractéristiques tellement intéressantes pour un raisonnement temps réel qu'ils n'ont pratiquement pas besoin d'être supervisés. En effet, ils constituent une série continue de niveaux d'approximations, ce qui permet un affinement automatique de la solution en fonction du temps disponible. Si l'objectif est d'obtenir un résultat avant une échéance, il suffit d'exécuter l'algorithme correspondant et de le stopper à la date limite, la solution obtenue possède alors la meilleure qualité possible compte tenu du temps imparti. La planification de ces fonctions est ainsi simplifiée [Boddy 89] et il est possible de garantir une planification optimale si les fonctions de performance sont strictement croissantes [Dean 91].

Cependant, les algorithmes *anytime* se réduisent souvent à des traitements simples et des fonctions bien précises. Lorsqu'un événement nécessite une solution complexe, il faut bien souvent enchaîner plusieurs algorithmes avant l'échéance imposée. Les travaux de Russell et Zilberstein [Russell 91] portent sur la composition de tels algorithmes, le but étant de définir un véritable langage *anytime* à partir de bibliothèques de telles fonctions. Dans [Grass 96] la méthodologie suivante est proposée :

- **Extraire une amélioration simple**. La première étape consiste à créer un algorithme *anytime* à partir d'un traitement classique. Pour cela, il faut localiser la progressivité, c'est-à-dire repérer l'endroit où la plus petite amélioration du résultat à lieu. Il faut alors sauvegarder la solution afin de la rendre disponible en cas d'interruption de l'exécution.

- **Définir une évaluation de la qualité**. Il faut ensuite déterminer un critère d'évaluation de la qualité du résultat produit afin de pouvoir mesurer l'utilité des améliorations successives.

- **Evaluer les performances**. Une fois la mesure de qualité définie, il est possible d'établir le rapport entre la durée d'exécution et la qualité du résultat. Cette courbe peut être réalisée par une évaluation moyenne sur une importante série de tests. Il s'agit d'une fonction qui, à une qualité des entrées et un temps de calcul, associe une estimation de la qualité de la réponse.

- **Compiler les séquences de fonctions simples.** Il est possible de combiner les fonctions simples obtenues par les trois étapes précédentes afin de réaliser des tâches complexes. Le temps à consacrer à chacune des fonctions élémentaires est calculé à partir des fonctions de performance estimées de façon à maximiser l'espérance de la qualité du résultat final.

Ainsi, dans un premier temps, une bibliothèque de fonctions *anytime* élémentaires est définie. Puis ces fonctions sont regroupées par séquences pour réaliser des traitements plus complexes qui ont également la propriété d'être interruptibles. Une fonction de performances est également calculée pour ces nouveaux traitements qui étendent ensuite la bibliothèque initiale.

Par cette technique de composition, l'algorithme à mettre en œuvre sera finalement de type interruptible. Le module de contrôle est alors inutile, il suffit de laisser le traitement se dérouler jusqu'à l'obtention de la réponse optimale ou jusqu'à l'arrivée de l'échéance. En effet, comme les résultats des premiers niveaux sont réutilisés par la suite, la seule décision consiste à sélectionner le niveau après lequel il faut stopper l'exécution en renvoyant la solution actuelle. Les exécutions possibles sont schématisées par la figure 9.

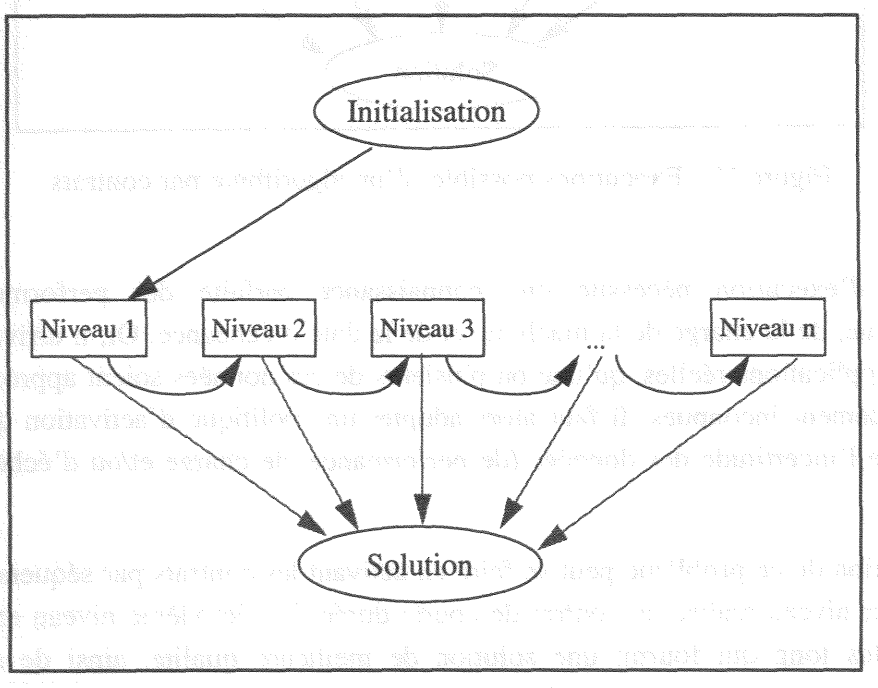

Figure 9 : Exécutions possibles d'un algorithme interruptible

3.3. Les algorithmes par contrats

Ces algorithmes permettent également de consacrer un temps quelconque à l'exécution, la qualité de la solution étant une fonction croissante du temps alloué. La véritable différence est que le temps est alloué *a priori*. Dans le cas idéal, l'algorithme par contrats possède une fonction de performance précise. Il choisit à l'initialisation, en fonction de l'échéance, quel niveau il exécute puis il envoie la solution dès que ce niveau est terminé (voir figure 10).

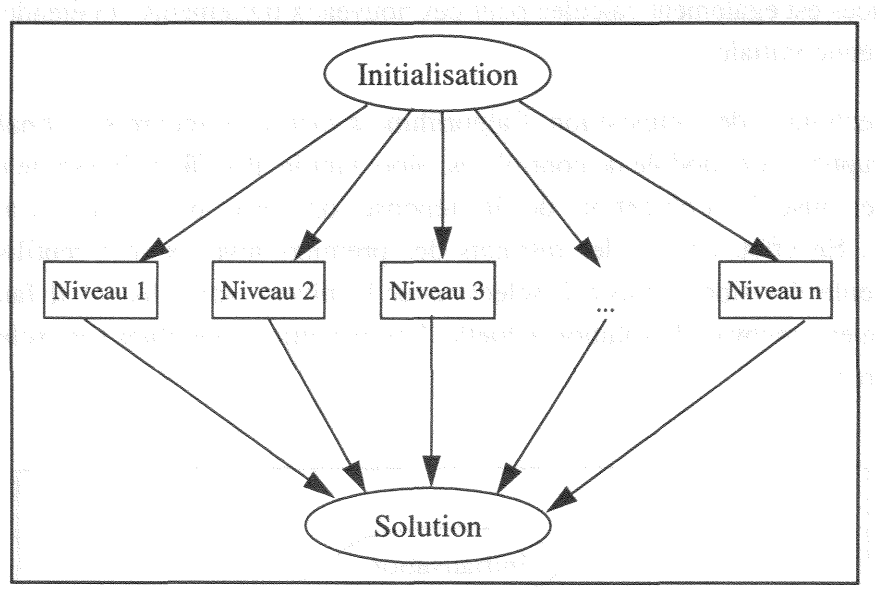

Figure 10 : Exécutions possibles d'un algorithme par contrats

Ce type d'exécution nécessite une connaissance parfaite des performances de l'algorithme, de la charge de la machine, et de la date d'échéance. Or, il arrive souvent dans les applications réelles, qu'une ou plusieurs de ces données soient approximatives voire totalement inconnues. Il faut alors adopter une politique d'activation qui tienne compte de l'incertitude des données (de performance, de charge et/ou d'échéance) du problème.

La résolution de ce problème peut se faire en activant les contrats par séquence. Ainsi, un premier niveau réalise un contrat de courte durée. Un deuxième niveau exécute un contrat plus long qui fournit une solution de meilleure qualité, ainsi de suite. Un algorithme par contrats se transforme ainsi en un algorithme interruptible. Cette transformation est intéressante puisqu'elle permet de traiter les algorithmes par contrats simplement, à la manière d'un algorithme *anytime*. Elle engendre tout de même une forte perte de temps due au fait que tout nouveau contrat achevé rend inutile les exécutions précédentes. De même le temps passé à réaliser un contrat inachevé est totalement inutile.

Dans [Russell 91], les auteurs proposent une telle transformation en sélectionnant chaque nouveau contrat de manière à ce que sa durée soit le double du précédent. Ils démontrent que la perte de temps est limitée à un facteur 4 en respectant une politique d'affectation simple. Autrement dit, un algorithme s'exécutant selon cette méthode serait identique à l'exécution d'un algorithme réellement anytime sur une machine 4 fois moins puissante.

3.4. Les méthodes multiples

Dans le cas où la progressivité est exprimée par des méthodes multiples, la supervision est réalisée par un module d'ordonnancement. Ce module effectue le raisonnement progressif en sélectionnant le degré d'approximation adéquat afin que le système garantisse un temps de réponse.

3.4.1. Calcul imprécis

Dans ce cas, le traitement se décompose en une partie obligatoire suivies d'une ou plusieurs parties optionnelles. La solution proposée dans [Liu 91] consiste à placer en priorité toutes les parties obligatoires des traitements à réaliser. Ces dernières ne sont pas considérées comme interruptibles. La difficulté est ensuite de placer les parties optionnelles de façon à garantir la meilleure qualité globale possible. Ces exécutions sont considérées comme interruptible, des priorités leurs sont affectées. Elles seront effectivement prises en compte dans le temps disponible entre les parties obligatoires dans l'ordre de leurs priorités.

Cette technique d'allocation permet de garantir une réponse minimale à l'ensemble des traitements. De plus, il reste valable dans le cas où les durées des différentes parties optionnelles ne sont pas connues.

3.4.2. Dans un système à base de règles.

Chaque source de connaissance est composée d'un paquet de règles minimum à scruter afin de garantir une réponse de qualité minimale. D'autres paquets de règles peuvent être définis afin d'augmenter la précision de la première solution. Il est ainsi possible d'implanter plusieurs niveaux de traitements, chacun fournissant un résultat d'une qualité différente et proportionnelle au temps consacré.

Le rôle de l'ordonnanceur est de planifier l'exécution des niveaux afin de garantir le respect de toutes les échéances et d'effectuer le plus d'affinements possibles selon une politique définie qui privilégie les actions les plus importantes et les plus urgentes. Pour cela, il dispose d'une estimation correcte des durées d'exécutions selon le paquet de

règles activé. Une autre hypothèse important est que la date limite accordée pour la réponse est parfaitement connue *a priori*.

Les choix d'affinements se font alors à l'aide d'un algorithme d'allocation. Un exemple de stratégie peut consister à affecter les tâches dans leur totalité puis à supprimer les affinements de plus haut niveau jusqu'à ce que toutes les tâches respectent leur date limite d'exécution, un exemple d'algorithme se trouve dans [Gallone 92].

3.5. Conclusion

La supervision d'un algorithme progressif est difficilement généralisable puisqu'elle dépend de nombreux paramètres. En effet, pour se synchroniser sur une horloge extérieure, un traitement doit se caractériser par la donnée d'un algorithme qui s'exécute sur une machine dans un certain environnement. Par conséquent, un contrôle efficace doit prendre en compte à la fois les caractéristiques de l'algorithme, de la machine et de l'environnement.

- **L'algorithme** est modélisé par le type de progressivité qu'il utilise : interruptible, par contrats ou par des méthodes multiples. Il faut d'autre part être en mesure de donner les performances de ces algorithmes. Cependant ces performances ne peuvent être calculées dans l'absolu, car elles sont liées à la machine. Un moyen de résoudre le problème est d'exprimer les performances en terme de complexité qui sera par la suite transformée en temps de calcul selon les caractéristiques de la machine.

- **La machine** est caractérisée par sa puissance de calcul et ses disponibilités. Ces paramètres permettent de construire la fonction utilité des différents algorithmes à partir des connaissances sur leur complexité.

- **L'environnement** impose les échéances aux différents traitements et par conséquent rend les approximations nécessaires et les affinements possibles.

Toutes ces conditions font qu'il existe différents type de contrôle, comme on a pu le voir dans ce chapitre, chacun étant adapté à une situation précise. Le chapitre suivant précise la situation dans laquelle nous nous plaçons afin de réaliser notre application sur l'ordonnancement.

4. Formalisation du problème

Dans un contexte temps réel, le système est en contact avec un environnement extérieur. Son rôle est de transformer des perceptions du monde en des actions effectives permettant de modifier l'environnement en vue d'atteindre un état voulu. Son fonctionnement se résume ainsi en une suite d'actions permettant d'atteindre un objectif. L'environnement est perçu sous forme de contraintes, puis le système choisit parmi ses comportements celui qui effectue les actions maximisant la probabilité de conduire au but (voir figure 11).

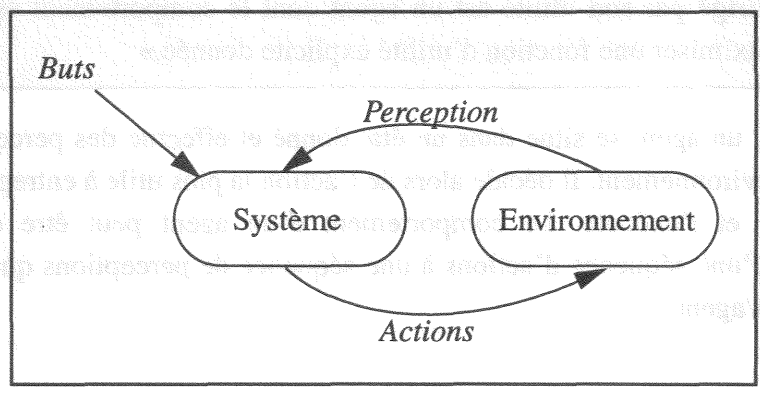

Figure 11 : Agent dirigé par les buts

Cependant, dès lors que l'exécution est contrainte par une ressource telle que le temps, la seule spécification du but est insuffisante pour dicter le comportement d'un agent. Zilberstein propose ainsi l'utilisation d'agents sélectionnant ses actions futures grâce à leur utilité plutôt que par le biais de buts à accomplir [Zilberstein 93].

4.1. Utilité

La notion d'utilité est définie comme une fonction associant un nombre réel à un état de l'environnement. Cette mesure correspond à une évaluation de la contribution d'une action à la réalisation d'un ou plusieurs objectifs dans une unité arbitraire du degré d'achèvement du but. Si l'agent est défini de manière progressive (voir la partie 1, paragraphe 1.1.2 page 28), il dispose ainsi d'un ensemble d'exécutions possibles. Chacune d'elles produit son propre résultat. Elles conduisent à différents états de l'environnement donc différentes évaluations de l'utilité. Les exécutions peuvent alors être sélectionnées de façon à maximiser l'utilité de l'agent.

Zilberstein effectue également une distinction entre l'utilité *explicite* et *implicite*. La première est la plus simple à identifier, elle consiste à considérer l'état courant comme un état final et à mesurer sa distance par rapport au but. L'utilité implicite, quant à elle, est plus précise mais plus complexe puisqu'elle considère l'état courant comme un état intermédiaire dans la résolution. Ainsi, elle prend en compte non seulement l'utilité explicite mais en plus les décisions prises par l'agent qui seront utiles pour ses planifications futures. Par exemple, pour un agent mobile autonome, si l'objectif est de se rendre en un point précis, le fait d'avoir construit le chemin optimal y aboutissant n'a aucun intérêt pour l'utilité explicite mais améliore nettement l'utilité implicite.

Zilberstein propose ainsi la définition d'un agent dirigé par son utilité (*utility driven agent*):

> «Un agent dirigé par son utilité est un agent dont le comportement est défini de manière à maximiser une fonction d'utilité explicite donnée.»

A tout instant, un agent se situe dans un état donné et effectue des perceptions afin de modéliser l'environnement. Il décide alors de l'action la plus utile à entreprendre pour le but recherché et l'exécute. Le comportement d'un agent peut être défini comme l'association d'une séquence d'actions à une séquence de perceptions qui est contrôlée par la fonction agent.

4.2. Objectif

Dans le cas général, l'objectif est de construire un algorithme capable de donner une réponse au bout d'un temps T inconnu *a priori*. Il est néanmoins possible de savoir à chaque instant la probabilité que l'échéance survienne alors que t secondes ont été allouées pour le calcul de la réponse. Cette probabilité est donnée par la fonction de répartition :

$$F(t) = P(t < T)$$

où $P(t < T)$ est la probabilité que le temps imparti t soit inférieur à T.

Si l'on dispose d'un algorithme progressif A, il peut s'exécuter selon différents temps d'exécution, la qualité du résultat étant fonction du temps alloué. Par conséquent, il peut être considéré comme une famille d'algorithmes A_t. Tous les algorithmes A_t traitent le même problème mais fournissent des réponses de différentes qualités puisque chacun a une durée qui lui est propre (égale à t unités de temps).

Le problème revient à fournir la meilleure solution quel que soit le temps imparti à la réponse en sélectionnant la suite optimale d'exécution de l'algorithme $A\ (A_{t1},A_{t2},...)$ afin de maximiser la valeur du résultat obtenu au bout du temps T.

4.3. Contraintes externes

Dans un contexte temps réel, l'agent doit prendre en compte la notion de temps de réponse. Cette notion peut se manifester différemment selon le type de problème à résoudre et selon l'environnement dans lequel l'agent est plongé. Dans de nombreuses applications, l'échéance ne peut malheureusement pas être parfaitement déterminée. Aussi, il est possible de modéliser le caractère incertain de l'arrivée de cette échéance par une loi de probabilité :

- **Evénement certain**. Il s'agit du cas le plus simple où la date limite (*deadline*) accordée à la réponse est parfaitement connue. Toutes les personnes ayant subit une session d'examen connaissent ce type d'échéance. Une fois la limite horaire atteinte, l'examinateur ramasse la copie et il est inutile de continuer à plancher. Dans ce cas, l'utilité de l'algorithme mis en œuvre est nulle si la réponse survient après la date limite. Par contre, si l'exécution est progressive, c'est-à-dire qu'une suite de résultats intermédiaires sont fournis, l'utilité est alors égale au dernier résultat obtenu avant l'échéance. Pour reprendre notre exemple, si une personne qui passe un examen dispose de plusieurs techniques pour obtenir un résultat et si celle sélectionnée ne fournit pas de résultat avant la date où il faut rendre la copie, elle est d'une parfaite inutilité. De même si plusieurs techniques ont pu lui fournir un résultat dans le temps imparti, elle retiendra uniquement le résultat ayant la meilleure qualité. Dans ce cas, l'échéance est parfaitement connue et fixe. La loi de probabilité est donc simple puisque nulle partout sauf à la date limite.

- **Gaussienne**. Si l'échéance est connue approximativement, une courbe gaussienne peut modéliser la loi. Par exemple, on doit recevoir un coup de téléphone vers 17h00, et certaines tâches doivent être impérativement réalisées avant cet événement. La date limite du traitement de ces tâches n'est alors connue que par une grossière approximation. Cependant, on possède tout de même des informations intéressantes sur l'échéance : plus on approche de la date estimée, plus la probabilité qu'elle survienne est importante

- **Exponentielle**. Il se peut que l'échéance soit approximativement connue, mais que plus le temps passe plus elle a de chances de survenir. La modélisation peut alors être effectuée par une loi exponentielle. En effet, la probabilité est très faible avant la date estimée de l'échéance puis augmente exponentiellement à son approche. Après la date estimée, il est pratiquement certain que l'échéance est

survenue. Par exemple, dans le cas d'un robot qui considère l'autonomie de ses batteries, les tâches à accomplir avant que le robot ne soit hors service doivent faire face à une échéance de ce type.

- **Uniforme**. Si l'échéance est totalement inconnue sur un intervalle donné, il s'agit d'une loi uniforme. En effet, il se peut en effet que dans certains contextes, le temps alloué à la réponse ne soit pas calculable *a priori*. Dans ce cas, la probabilité que l'échéance survienne est la même quel que soit l'instant considéré.

Il se peut parfois qu'il n'y ait pas de date limite proprement dit, mais que la notion d'échéance intervienne par le biais d'un coût lié à l'attente de la réponse. En effet, si le temps passé à chercher une solution est coûteux, il est inutile de produire une réponse précise dont l'apport au système sera moins importante que le coût engendré par son calcul.

Un exemple simple consiste à appeler un taxi et le faire attendre. Vous n'avez pas alors de réelle échéance, cependant votre porte-monnaie vous rappellera rapidement que seule les traitements ayant une réelle utilité peuvent être entrepris. Il s'agit ici de trouver un juste compromis entre l'utilité des contrats et le coût de l'attente de leur exécution.

Il faut tout de même remarquer que cette notion de coût n'est pas forcément incompatible avec la notion d'échéance fixe. Il est ainsi possible d'avoir des traitements dont l'exécution engendre un coût fixe et qui, passé un certain délai, ne seront plus d'aucune utilité. Pour reprendre l'exemple précédent, le chauffeur de taxi ne vous voyant pas venir fixera certainement une date limite après laquelle il repartira.

4.4. Performances

Dans un contexte temps réel, le processeur est bien souvent une ressource critique. Ainsi plusieurs agents peuvent vouloir s'exécuter en même temps et provoquer alors une situation de concurrence. Un mécanisme de contrôle est dès lors chargé de choisir l'agent prioritaire. Pour cela, il doit bien évidemment disposer de connaissances lui permettant d'estimer l'intérêt des différentes exécutions. Cette mesure de performances peut se faire par différents moyens. Zilberstein propose l'utilisation d'une fonction utilité qui intègre la notion de coût d'attente et d'échéance [Zilberstein 93]. Russell, lui, juge l'utilité d'après l'historique des états pris par l'agent et intègre la notion d'échéance par une loi de probabilité sur son arrivée [Russell 95].

4.4.1. Utilité dépendante du temps

Une façon simple d'intégrer les contraintes temps réel à un agent est de les prendre en compte dans la fonction utilité. Pour un même agent, fournir une réponse d'une même qualité possédera alors un intérêt différent selon le temps nécessaire à son obtention. Il faut pour cela prendre en compte les différents types d'échéance possible :

- Dans le cas d'une échéance fixe, tant que la durée du traitement garantit son respect, l'utilité se mesure par une évaluation du degré d'achèvement du but à atteindre. Si par contre l'algorithme choisi fournit une réponse après l'échéance, son utilité est nulle au delà de cette date. Si la qualité d'un résultat ***r*** d'un système est définie par la fonction ***Q(r)*** et que l'échéance fixe est à T_0, alors la fonction utilité peut être définie par la formule :

$$U(r,t) = \begin{cases} Q(r) & \text{si } t < T_0 \\ -\infty & \text{sinon} \end{cases}$$

- Si l'environnement impose un coût lié à l'attente de la réponse, l'utilité de cette dernière en sera diminuée d'autant. Si la valeur d'un résultat ***r*** d'un système dans un état de l'environnement est définie par la fonction ***V(r)*** et que le coût d'une unité de temps est fixé à ***c***, la fonction utilité est de la forme :

$$U(r,t) = V(r) - c \cdot t$$

- Dans le cas d'une échéance stochastique, il est impossible de calculer précisément l'utilité compte tenu du fait que l'échéance n'est connue que sous la forme d'une loi de probabilité. On se sert alors de cette loi pour maximiser l'espérance de l'utilité. Par exemple, pour une loi exponentielle, on diminue la valeur de l'utilité exponentiellement à l'approche de la date limite estimée. La fonction est donc de la forme :

$$U(r,t) = V(r) - e^{-ct}$$

4.4.2. Utilité selon l'historique.

Dans [Russell 95], les auteurs proposent un calcul de l'utilité à partir de l'historique des états de l'environnement.

Soit O l'ensemble des observations qu'un agent peut percevoir et A l'ensemble des actions possibles qu'il peut effectuer sur le monde extérieur. Les agents sont étudiés sur

une période modélisée par T, l'ensemble des instants, supposé totalement ordonné, qui peut être représenté par une suite d'entiers positifs sans perte de généralité. L'historique des observations d'un agent, notée o^T, correspond à la séquence des perceptions réalisées par l'agent au cours de la période étudiée. Elle peut ainsi être modélisée par une fonction qui associe une observation O à chaque instant de T. L'ensemble des observations réalisées jusqu'au temps t, notés o^t, est appelé préfixe de l'historique o^T (cet ensemble est la projection de o^Tsur [0..t]). De même, l'historique des actions, notée a^T, est la séquence des actions de l'agent au cours de la période. C'est une fonction qui associe une action de A à tout instant de T. Le préfixe de l'historique des actions jusqu'à l'instant t, noté a^t, est la projection de a^T sur [0..t].

Nous pouvons alors donner une définition plus formelle de la fonction agent. En effet, le comportement d'un agent peut être modélisé par une fonction f qui, à une séquence de perception sur l'intervalle [0..t], associe l'action de l'agent à l'instant t. Ceci se traduit par la relation :

$$a^T(t) = f(o^t)$$

La fonction agent ainsi modélisé possède un but, qu'elle doit accomplir en étant implanté par un certain programme. De plus, l'agent s'exécute dans un environnement E qui évolue au cours du temps. Une description statique n'étant pas satisfaisante, il faut plutôt le concevoir de façon dynamique, c'est-à-dire qu'à chaque instant est associé un état. On définit alors X, l'ensemble des états possibles de l'environnement et x^T une fonction qui à tout instant de T associe un élément de X. Cette fonction est la *trajectoire*, c'est-à-dire la succession d'états occupés par l'environnement.

A l'instant t, l'agent ne perçoit pas complètement l'état de l'environnement x^T(t), mais uniquement l'information filtrée par ses capteurs. Cela s'exprime par l'ajout d'un biais lié à la fonction de filtrage des perceptions fp. Une perception est ainsi définie par la formule :

$$o^T(t) = fp(x^T(t))$$

De même, les actions réalisées par l'agent ne modifient pas directement l'environnement mais agissent sur celui-ci pour obtenir l'état voulu par l'intermédiaire de ses effecteurs. Ceci est modélisé par le biais lié à la fonction de transition de l'environnement, fe. Nous avons ainsi la formule de récurrence :

$$x^T(t+1) = fe(a^T(t), x^T(t))$$

L'utilité est calculée au moyen d'une fonction qui, à un historique d'états de l'environnement, associe un nombre réel correspondant à une note qui juge de son efficacité. La signature de la fonction utilité est alors :

$$U : X^T \rightarrow \Re$$

Cette fonction permet de définir la valeur V d'un agent, qui n'est autre que l'intérêt de l'exécution de l'agent vis-à-vis de l'environnement extérieur, c'est-à-dire l'utilité des effecteurs de l'agent sur l'état de l'environnement :

$$V(f, E) = U(fe(a^T, x^T))$$

Il se peut également que l'agent ne soit pas systématiquement plongé dans le même environnement mais qu'il existe un ensemble d'environnements possibles E, chacun ayant une certaine probabilité de survenir. Si la distribution suit une loi p, la valeur de l'agent est alors estimée par la formule :

$$V(f, \mathrm{E}) = \sum_{E \in \mathrm{E}} p(E) V(f, E)$$

La notion d'utilité étant posée, il est aisé de définir un comportement optimal. En effet, il suffit de chercher la fonction agent f_{opt} qui maximise la valeur de V dans l'ensemble des environnements E :

$$f_{opt} = \arg max_f (V(f, E))$$

Cependant, cette situation idéale est loin d'être fréquente. En effet, pour des raisons de complexités, cette fonction f_{opt} n'est pas toujours implémentable sur les architectures dont nous disposons. C'est-à-dire que cette fonction existe, mais le temps nécessaire à sa réalisation la rend inutile pour la plupart des exemples à traiter. Il suffit de reprendre l'exemple du jeu d'échec pour s'en rendre compte. Ainsi, l'optimalité ne peut pas être recherchée dans l'absolu, mais seulement sur l'ensemble des programmes faisables pour une architecture donnée.

Un programme d'agent est considéré comme une séquence d'éléments appartenant à un ensemble $R = \{ r1, ..., rn \}$ de procédures ou de règles de décision. Chaque procédure de décision recommande (mais n'exécute pas) une action a_i. Elle possède ainsi deux paramètres :

- un temps d'exécution, t_i 0;
- une qualité, $q_i \geq 0$, réelle, qui donne une estimation de la valeur de l'exécution de a_i par la formule $q_i = U([E,a_i])$.

Soit M l'architecture sur laquelle les procédures de décisions sont exécutées. Soit t_M le temps maximum que l'architecture permet d'attribuer à une exécution (dans le cas d'un algorithme *anytime* par contrats il s'agit de la durée du plus long contrat se terminant avant l'échéance). L'architecture M exécute une fonction agent $f = f_1 ... f_m$ en exécutant tour à tour chaque procédure de décision. A l'arrivée d'une échéance, ou lorsque que l'intégralité de la séquence est terminée, l'agent choisit l'action préconisée par la procédure de décision exécutée ayant la qualité la plus élevée.

• **Echéance fixe**

Dans ce cas trivial, il suffit d'exécuter une séquence de contrats pouvant se terminer avant l'échéance et de conserver la réponse ayant la plus forte qualité. Soit $f_1 ... f_j$ le plus long préfixe d'une fonction f tel que $\sum_{i=1}^{j} t_i \leq t_d$, la valeur de la qualité est alors obtenue par la formule :

$$V(f, E) = Q_i \quad \text{avec} \quad Q_i = max\{ q_1, ... , q_j \}.$$

• **Echéance à coût fixe**

L'agent n'ayant pas de réelle échéance, il peut exécuter la séquence complète $f = f_1 ... f_m$ jusqu'à son terme imposée par l'architecture M. Cependant, l'exécution de chaque f_i induit un coût supplémentaire égal au temps d'exécution t_i multiplié par le coût unitaire c. La qualité est ainsi définie par l'équation :

$$V(f, E) = Q_m - c \sum_{i=1}^{m} t_i$$

L'exécution optimale sera celle qui maximise cette fonction, c'est-à-dire celle dont le résultat est le plus utile compte tenu du coût lié à son temps d'obtention.

• **Echéance stochastique**

Dans ce cas, la seule information disponible est une estimation de l'arrivée de la date d'échéance. Ainsi la valeur calculée n'est qu'une simple estimation de la valeur réelle.

Pour chacune des dates possibles, il est possible de calculer la valeur de la fonction d'agent. Il suffit alors d'appliquer la loi de probabilité sur ces valeurs pour obtenir une estimation correcte avec la formule suivante :

$$V(f, E) = \sum_{i=1}^{m} \left(P_d\left(\sum_{j=1}^{i+1} t_j \right) - P_d\left(\sum_{j=1}^{i} t_j \right) \right) Q_i$$

où P est défini par

$$P_d(t) = \int_{-\infty}^{t} P_d(t')dt'$$

et

$$P_d(t) = 1 \text{ pour } t \geq \sum_{i=1}^{m} t_i$$

Lorsque l'échéance est de type stochastique, la séquence d'action optimale ne peut pas être déterminée *a priori*, mais elle est calculée dynamiquement de différentes façons selon la loi d'arrivée de la date d'échéance.

Le but est de calculer pour chaque date t, la meilleure séquence de décisions possible. On utilise pour cela la programmation dynamique. On construit d'une table $S(t, i)$ qui associe à une date t et une décision r_i la séquence de décision fournissant la réponse de meilleure qualité qui se termine par r_i et à laquelle correspond une exécution s'achevant à la date t.

La table se construit par les formules :

$$S(i, 0) = 0 \quad \text{pour toute règle } i$$

$$S(0, t) = 0 \quad \text{pour tout instant } t$$

$$S(i, t) = \max_{k \in 0..i-1} [S(k, t - t_i) + (q_i - q_k)[1 - P_d(t)]]$$

Une fois la table construite, il est possible de choisir très rapidement, pour chaque durée, la séquence de décisions qui fournit le résultat ayant la meilleure qualité.

Partie 2 : Ordonnancement par contrats

1. Objectifs

Comme nous l'avons vu dans les chapitres précédents, l'écriture d'un système temps réel nécessite le respect d'un certain nombre de contraintes. Il faut constamment être à l'écoute des événements extérieurs tout en ayant la faculté de se concentrer sur un travail urgent. Ainsi, le système doit non seulement être en mesure de répondre aux différents stimuli externes, mais aussi être capable de gérer son temps de travail afin d'être ponctuel quand il le faut.

Un système temps réel doit donc être doté d'un module de contrôle permettant une gestion efficace du temps. Ce module doit disposer d'informations diverses pour pouvoir garantir le respect des échéances et sélectionner les traitements les plus utiles de façon à produire un résultat ayant une qualité qui soit la plus élevée possible. Il lui faut connaître la puissance et la charge de la machine, mais aussi les performances des algorithmes qu'il emploie, c'est-à-dire la qualité du résultat produit en fonction du temps consacré à l'exécution dans le cas où l'algorithme est défini de façon progressive. Selon le profil de cette fonction (en escalier, exponentielle, logarithmique, linéaire...), le module de contrôle sera différent. Aussi, nous allons nous concentrer sur un exemple particulier. Nous étudierons alors les performances des algorithmes mis en œuvre et définirons un contrôle adéquat. L'exemple que nous avons choisi est un grand classique en temps réel : un problème d'ordonnancement. C'est en fait un problème que rencontrent tous les systèmes d'exploitation temps réel, ainsi que dans de très nombreuses applications à divers niveaux. Par exemple, pour notre système, le module de contrôle doit réaliser l'ordonnancement des traitements à exécuter.

Cette partie est ainsi destinée à illustrer les concepts introduits dans la partie précédente. Nous avons choisi comme domaine d'application un des problèmes d'ordonnancement connu comme étant NP-complet. Notre but est de chercher une solution correcte (où toutes les tâches son exécutées avant leur date limite) à l'aide d'une méthode approximative. Il est donc impossible de disposer d'une solution réalisable approchée pendant la procédure de recherche. Par conséquent, la mesure de qualité utilisée est la probabilité de se rapprocher de l'obtention d'une solution réalisable qui dépend à la fois de la difficulté à trouver des solutions réalisables et de la méthode utilisée pour essayer de trouver ses solutions. Aussi le système que nous avons implémenté a été testé sur un large échantillon des énoncés possibles afin d'estimer au mieux l'espérance d'obtention d'un résultat correct pour un algorithme donné.

Nous insisterons essentiellement sur l'établissement d'une famille d'algorithmes approximatifs sur laquelle peut s'appuyer le compromis entre qualité et temps de réponse. Nous passerons plus rapidement sur les aspects de supervision. Nous

donnerons néanmoins tous les éléments nécessaires à la réalisation d'un module de supervision de la séquence de traitements mis en œuvre par le système. La théorie décrite dans la partie précédente s'applique en effet sur notre exemple d'ordonnancement et doit permettre la réalisation d'un tel module.

Bien évidemment, le système proposé ici ne se limite pas uniquement aux problèmes d'ordonnancement, il peut être utilisé pour beaucoup d'autres domaines d'application. En effet, le problème étant l'optimisation d'un problème de complexité non polynomial, il s'agit d'une tâche difficile et la résolution d'une telle application pourra être réutilisée dans d'autres cadres pour d'autres problèmes.

2. Ordonnancement de tâches non préemptives

Les techniques d'ordonnancement sont depuis longtemps un sujet de recherche commun à de nombreuses communautés (recherche opérationnelle, intelligence artificielle, Informatique temps réel...). Elles sont utilisées dans un large éventail comme par exemple les applications informatiques ou les processus de production industriels. Dans les systèmes informatiques, l'ordonnanceur est un module fondamental pour le respect des contraintes de temps associées aux tâches qui doivent partager une ou plusieurs ressources. Dans les processus industriels, il faut organiser la production, c'est-à-dire programmer des tâches afin de respecter un ensemble de contraintes (de temps, de précédences ou de disponibilité des ressources non partageables ...) de manière à optimiser un critère donné (coût, rapidité, qualité, ...). Nous renvoyons le lecteur aux nombreux papiers relatant la vaste panoplie d'algorithmes disponibles comme [Blazewicz 93] [Lawler 91] [Portmann 93] [Simons 78]. Une classification, généralement respectée par l'ensemble de la communauté scientifique, consiste à différencier les problèmes selon trois critères notés $\alpha|\beta|\gamma$:

- α spécifie le nombre de ressources disponibles (par exemple, si la ressource à partager est un micro-processeur, dans le cas $\alpha = 1$, on obtient une machine mono-processeur).
- β spécifie les caractéristiques des tâches à ordonner, c'est-à-dire si elles ont des dates d'activation au plus tôt, des dates d'activation au plus tard, des durées fixes, variables ou unitaires, des poids différents, des contraintes de précédences, si elles sont préemptives ou non ...
- γ spécifie la contrainte à optimiser. En effet, de nombreuses solutions sont souvent acceptables, mais, parmi elles, on cherche la meilleure selon le critère voulu qui peut être le coût, la rapidité ou la stabilité en cas d'ajout de nouvelles tâches ...

Dans le cadre d'une application pour le Centre de Programmation de la Marine, nous nous sommes intéressés au cas particulier d'ordonnancement de tâches non préemptives sur une ressource non partageable. L'objectif était la réalisation d'un système de planification pour la répartition automatique des moyens d'autodéfense d'un système de combat. Cela consiste à combiner des contraintes, qu'elles soient temporelles ou liées aux contraintes d'utilisation, pour rechercher une solution guidée par un ensemble d'heuristiques opérationnelles (par exemple, on cherche à détruire toutes les pistes menaçantes en gardant un maximum de marges pour le traitement de chacune d'elles). Il s'agit donc d'un problème de l'ordonnancement pour le système d'armes. Une ressource

non partageable et non consommable doit être allouée à des tâches ayant chacune une date de disponibilité, une date limite et étant non-préemptives (leur exécution doit être globale et ne peut pas être interrompue pour être reprise par la suite). Le critère à optimiser est le nombre de tâches respectant leur date limite. Il faut remarquer que ce critère ne suffit pas pour établir un ordre total entre les solutions. Aussi un second critère tel que minimiser la date de fin de la dernière tâche à exécuter peut être défini bien qu'il ne permette pas lui non plus de classer toutes les solutions. Avec la notation $\alpha|\beta|\gamma$ définie, le problème traité est donc du type $1|r_j,\Delta_j|N_T,C_{max}$.

3. Les approches possibles

Le problème d'ordonnancement décrit dans le chapitre 2 est reconnu comme NP-Complet [Garey 79]. Ainsi, sa résolution pour des problèmes de grande taille ne peut pas se faire de façon optimale, mais nécessite l'utilisation de techniques donnant des solutions approchées. Les paragraphes suivants décrivent les méthodes les plus souvent implémentées.

3.1. Les méthodes exactes

La recherche d'une solution optimale se fait obligatoirement par une méthode qui consiste à étudier toutes les solutions possibles et à sélectionner celle qui est réalisable et qui répond au mieux aux critères fixés comme objectif. Cependant, si le but est d'ordonner n tâches, l'algorithme naïf consiste à étudier *n!* solutions différentes, ce qui est inconcevable lorsque n devient trop grand (voir figure 1, page 28).

Une façon d'implémenter cette recherche exhaustive de manière plus efficace consiste à utiliser une procédure par séparation-évaluation (*Branch and Bound Algorithm*). La technique revient à organiser l'ensemble des solutions sous la forme d'un arbre, appelé arbre de recherche. Les branches de l'arbre représentent une décision (*placer la tâche i avant la tâche j*), les nœuds de l'arbre sont des solutions intermédiaires (*où un certain nombre de tâches sont déjà ordonnées*), les feuilles de l'arbre sont les différents ordonnancements possibles. Le développement complet de l'arbre revient donc à effectuer une recherche exhaustive sur les *n!* solutions. Cependant, l'intérêt d'un tel arbre est qu'il est possible de calculer un majorant de l'objectif au niveau de chaque nœud. En effet, après chaque décision, une borne supérieure de la valeur du critère à maximiser est estimée, cette estimation étant bien sûr la plus faible possible. Il suffit ensuite d'explorer en priorité les branches ayant la plus forte espérance pour la valeur de l'objectif, afin de trouver rapidement une bonne solution en fonction du critère choisi. Une fois cette solution trouvée, toutes les branches ayant une borne supérieure inférieure à la valeur du critère obtenu sont inutiles à explorer. L'arbre est ainsi élagué et la recherche se poursuit plus rapidement sans que la qualité de la réponse soit dégradée.

Cette méthode est efficace à condition bien sûr de définir correctement le calcul du majorant, de façon à élaguer le plus possible l'arbre de recherche. Mais le problème étant NP-Complet, son utilisation reste toutefois réduite à des cas simples où la taille du problème est raisonnable. La taille maximale est fixée en fonction du temps de traitement accordé et de la puissance de calcul mise en œuvre.

3.2. Les méthodes par construction

Face à des problèmes de grande taille, l'espace des solutions est tellement important qu'il est inimaginable de le parcourir entièrement, même en créant un arbre de recherche sur lequel de multiples élagages sont effectués. Par conséquent, des méthodes ont été développées afin de fournir des résultats approchés. Elles cherchent à construire directement le résultat optimum ou plus exactement un résultat qui s'en approche le plus possible. Il s'agit donc de définir un stratégie de construction qui soit relativement simpliste afin de permettre l'obtention rapide d'un résultat, mais qui soit suffisamment fiable pour que le résultat soit de qualité correcte.

La difficulté ici est de définir une façon progressive de construire la solution afin de pouvoir réaliser les choix qui permettent d'obtenir une solution plus globale qui soit de la meilleure qualité possible. Le recours à des techniques *heuristiques* est bien souvent nécessaire. Elles sont basées sur des règles simples qui correspondent à des intuitions humaines. Par exemple, une règle souvent utilisée est celle de Jackson [Jackson 55] qui consiste simplement à classer les tâches à ordonnancer en fonction de leur urgence (*Earliest Deadline First*). Il est à noter que cette règle permet d'obtenir un résultat optimal lorsque toutes les dates de disponibilités sont égales.

De nombreuses autres heuristiques ont été développées. Elles reproduisent en général les diverses règles qu'un être humain utiliserait face à un problème où l'inventaire des possibilités est irréaliste. Elles utilisent diverses notions, telles que l'urgence, la date de fin au plus tôt, le temps de latence (différence entre la date limite et la fin de l'exécution), ...

Par exemple, pour définir un ordre sur les tâches, Chu [Chu 92] associe une priorité à chacune d'elles et sélectionne ainsi la meilleure à un instant donné. La tâche est alors insérée à la fin de l'ordonnancement déjà établi, puis le processus est réitéré sur les tâches restantes. La propriété, qui généralise la règle de Smith [Smith 56], est calculée dynamiquement en fonction de la date de disponibilité de la ressource, de la fenêtre d'exécution de la tâche et de la durée. La formule exacte est fournie dans la figure 12.

$$PRTT(i, \Delta) = max(\Delta, r_i) + max(max(\Delta, r_i) + p_i, d_i)$$

$$avec \begin{cases} \Delta = \text{date de disponibilité} \\ r_i = \text{début de la tâche} \\ d_i = \text{fin de la tâche} \\ p_i = \text{durée de la tâche} \end{cases}$$

Figure 12 : Calcul d'une priorité dynamique

Ces algorithmes effectuent des choix conduisant à un résultat final qui est de plus ou moins bonne qualité. Si la qualité de la solution est jugée insuffisante, il est alors opportun de remettre en cause ces choix en effectuant des retours en arrière (*backtracking*) dans la procédure de décision. Si ces retours ne sont pas limités, l'algorithme aboutit alors à la solution optimale, moyennant bien sûr un coup au moins aussi élevé que les méthodes décrites dans le paragraphe 3.1. Il s'agit donc de trouver un juste compromis entre la durée de calcul et la qualité désirée afin de définir dans quelle mesure de tels retours en arrière sont autorisés.

3.3. Les méthodes par décomposition

Ces méthodes utilisent la démarche inverse de celles par construction, le *leïtmotiv* est plutôt ici «diviser pour régner» (*divide and conquer*). Au lieu de s'attaquer globalement à un problème d'ordonnancement trop complexe pour être réalisé rapidement, la technique est de le diviser en petits sous-problèmes qui peuvent être eux-même décomposés. Au fur et à mesure, les divisions vont réduire la complexité des problèmes à résoudre pour atteindre des tailles raisonnables afin d'être traités rapidement et de manière optimale par une méthode exacte. La décomposition peut se faire à l'aide de différents critères :

- hiérarchiques : le problème est décomposé en différents niveaux correspondant à des données de précisions différentes. Une fois un niveau résolu, la solution retenue fixe certains paramètres des niveaux inférieurs afin de restreindre la complexité des sous-problèmes à résoudre [Erschler 85].

- structurels : une contrainte est relâchée de façon à réduire la complexité du problème. Puis on cherche parmi les solutions trouvées, celle qui satisfait la contrainte relâchée [Roy 70].

- temporels : la taille du problème est réduite par une focalisation temporelle de la résolution. On s'intéresse au problème uniquement dans un intervalle temporel précis puis on déplace cet intervalle sur l'horizon complet de la planification [Portmann 88].

- spatiaux : la décomposition porte sur la façon dont le problème est physiquement implanté, elle consiste par exemple à décomposer les ateliers en sous-ateliers [Portmann 88].

3.4. Les méthodes de propagation de contraintes

La technique consiste à modéliser le problème de façon à représenter une solution comme un ensemble de valeurs attribuées à des paramètres. Il faut, dans un premier

temps, définir les variables qui permettent de décrire complètement une solution. Ensuite, l'affectation des valeurs à ces variables revient à résoudre le problème posé. Cela se fait en imposant des contraintes aux variables à instancier. Dans le cadre d'un ordonnancement, le problème peut s'exprimer par des contraintes temporelles («la tâche doit commencer entre r_i et Δ_i») , des contraintes de précédences («la tâche i doit finir après la tâche j»), des contraintes de ressources («la tâche i a besoin de la machine m pendant p_i»).

La méthode de propagation de contraintes consiste à sélectionner une variable et à lui affecter une valeur parmi celles imposées par les contraintes (il est bien sûr préférable de choisir la valeur qui optimise le critère qui sert d'objectif). En considérant les contraintes de précédences et d'allocation de ressources, ce choix induit des conséquences sur les autres variables. Ainsi, la propagation de cette nouvelle affectation permet éventuellement de restreindre les valeurs possibles pour les variables non instanciées. Puis l'opération est réitérée : sélection d'une nouvelle variable, affectation, propagation. L'arrêt de l'itération peut survenir en cas de réussite, c'est-à-dire quand chaque variable possède une seule valeur possible. Il suffit alors de considérer ces valeurs pour observer la solution obtenue. L'autre arrêt possible est un cas d'échec : une ou plusieurs variables ne peuvent prendre aucune valeur. Mais ce cas il faut remettre en cause un ou plusieurs choix effectués lors des itérations précédentes en effectuant des retours en arrière (*backtracking*) et recommencer les itérations avec de nouvelles valeurs jusqu'à ce que l'arrêt s'effectue sur un cas de succès. Si les retours en arrière ne sont pas limités, l'algorithme itère jusqu'à l'obtention d'une solution. Ceci est utile pour garantir une réponse si elle existe, mais du point de vue du temps d'exécution, cela peut s'avérer catastrophique compte tenu de la complexité du problème. En effet, dans le pire des cas, l'algorithme examine les *n!* possibilités. Il s'agit donc là aussi de trouver une stratégie pour effectuer les retours en arrière qui maximisent la probabilité de trouver une solution sans pour autant les effectuer tous et retomber sur un algorithme exact dont le temps exécution est beaucoup trop important.

3.5. Par voisinage

Plutôt que de chercher à construire une solution par des techniques approchées, une autre possibilité est de se fier à la chance. En effet, puisque l'espace de recherche est connu, il est possible de choisir une solution au hasard. La probabilité de tomber directement sur une solution acceptable est bien entendu très faible. Mais si l'on arrive à ordonner l'espace de recherche, cette solution peut être le point de départ d'une recherche fructueuse. La notion de chance disparaît alors pour laisser place à une probabilité de succès qui peut être très forte selon la technique utilisée. Ainsi, on définit la notion de voisinage entre les solutions comme une certaine similitude, c'est-à-dire

qu'il est possible de passer de l'une à l'autre en appliquant une opération, telle que par exemple inverser l'ordre de 2 tâches.

Les méthodes de voisinage reviennent donc à choisir une solution aléatoirement dans l'espace de recherche puis à regarder parmi les solutions «voisines» s'il en existe une de meilleure qualité. Cette dernière devient alors la solution courante et ses voisins sont ensuite évalués afin d'améliorer à nouveau le résultat. Cette technique peut apparaître comme une recherche aveugle, mais si les voisins sont définis de manière correcte, elle peut se révéler très efficace.

3.5.1. Recuit simulé

L'idée a été proposée par Kirkpatrick [Kirkpatrick 82], elle est inspirée de la thermodynamique. En effet, pour obtenir certaines propriétés, un corps physique doit parfois passer par différents palier de température. Cette dernière est donc un paramètre qui permet de passer par différents états intermédiaires avant d'obtenir l'état final désiré. De la même manière, dans notre espace de recherche, on peut vouloir parfois passer par des solutions qui dégradent la qualité du résultat, mais qui permettent de sortir d'un minimum local, afin par la suite de retrouver une solution qui sera de meilleure qualité. L'idée est donc de définir un paramètre (*la température*) qui détermine dans quelle mesure une solution voisine ayant une moins bonne qualité peut être sélectionnée. En fixant ce paramètre élevé au départ, cela permet de balayer le plus largement possible l'espace de recherche. Puis, en travaillant avec une température faible, la technique devient alors une méthode de descente classique, c'est-à-dire qu'elle consiste à sélectionner le meilleur voisin tant qu'il en existe un.

3.5.2. Tabou

Cet approche [Glover 89] [Glover 90] consiste à parcourir l'espace de recherche en mémorisant à la fois la meilleure solution et les chemins déjà empruntés, sous forme d'une liste tabou. La technique consiste à se diriger constamment vers un voisin représentant une solution de qualité plus élevé (*méthode de descente*), voire celui qui représente la solution de plus forte qualité (*méthode de plus forte pente*). On obtient ainsi rapidement une solution acceptable, mais qui a de fortes chances d'être un optimum local. La solution est retenue comme meilleure solution actuelle, mais la recherche est relancée sur des voisins non encore visités afin de tenter d'améliorer la qualité du résultat final.

3.5.3. Les algorithmes génétiques

Ces algorithmes sont une forme plus complexe des méthodes de voisinage, puisqu'ils travaillent non pas avec une seule solution courante mais avec un ensemble de solutions appelé population. Comme son nom l'indique, la technique s'inspire fortement de la génétique et de la sélection naturelle au cours des évolutions.

Chaque individu de la population est caractérisé par un chromosome qui est en fait un ensemble de valeurs qui permet de codifier une solution du problème. Il est donc possible de lui affecter une qualité qui va mesurer l'adéquation de l'individu au critère à optimiser. Le principe de l'algorithme est de générer de nouvelles populations avec des individus plus forts. Pour cela, une procédure de croisement est définie. Elle consiste à sélectionner au hasard deux individus (les chances de sélection étant proportionnelles à la qualité de l'individu) qui vont servir de parents pour créer un fils, qui subira éventuellement une mutation. L'ensemble des fils générés devient ainsi une nouvelle population qui contiendra des individus de meilleure qualité. En itérant de la sorte sur les populations, les solutions manipulées seront progressivement affinées pour obtenir un résultat final de qualité acceptable [Golberg 89].

Pour un panorama des utilisations de ces algorithmes pour les problèmes d'ordonnancement, le lecteur peut se référer à [Caux 95].

3.5.4. Les réseaux de neurones

L'utilisation des réseaux de neurones est fondée sur le calcul parallèle d'entités simples appelées neurones formels. Beaucoup de travaux sur les modèles connectionnistes sont dédiés à la reconnaissance des formes et à la classification. Dans ces modèles, chaque neurone effectue un traitement sur un fragment de la perception. D'autres travaux connectionnistes, plus rares, portent sur les problèmes d'optimisation. Dans ce cas, les propriétés qui émergent d'un réseau sont exploitées pour résoudre un ensemble de contraintes. C'est typiquement le cas du modèle de Hopfield décrit dans le chapitre suivant.

Cardeira, dans sa thèse [Cardeira 94a], effectue une application intéressante de ce modèle pour l'ordonnancement de messages dans les systèmes temps-réels. Herault [Hérault 91], [Hérault 95] propose également un modèle de réseau, *les réseaux de neurones pulsés*, dérivé de celui de Hopfield qui permet également de résoudre un ensemble de contraintes avec la possibilité de jouer sur les importances relatives de ces contraintes au cours du processus d'optimisation.

4. Une approche neuromimétique

4.1. Le modèle de Hopfield

4.1.1. Le réseau

Hopfield et Tank [Hopfield 85] se sont inspirés de la *théorie des verres de spins* [Hamman 87] pour proposer un réseau de neurones entièrement connectés (voir figure 13) qui permet de résoudre des problèmes d'optimisation. Ils ont démontré l'efficacité de leur modèle sur le problème du voyageur de commerce. De plus, ce réseau correspondant à un circuit électrique, cela permet de réaliser une solution câblée très rapide une fois le réseau correctement défini.

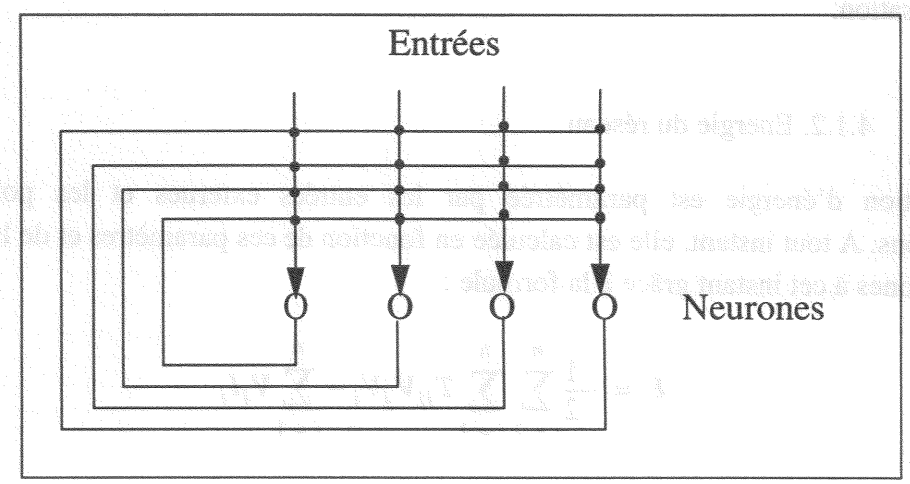

Figure 13 : Modèle de Hopfield

Dans ce réseau, chaque neurone possède une entrée externe et réalise à chaque instant une somme pondérée de la valeur des autres neurones avec cette entrée, puis effectue un seuillage de la valeur obtenue afin de conserver une valeur comprise entre 0 et 1 (voir figure 14). Cette valeur est alors utilisée en entrée des autres neurones à l'instant suivant.

Le réseau est ainsi récursif puisque la sortie de chaque neurone est connectée aux entrées des autres. Il n'est donc pas systématiquement dans un état stable. Il évolue au contraire au cours du temps, l'état du réseau à un instant donné étant entièrement défini par la valeur du vecteur des neurones.

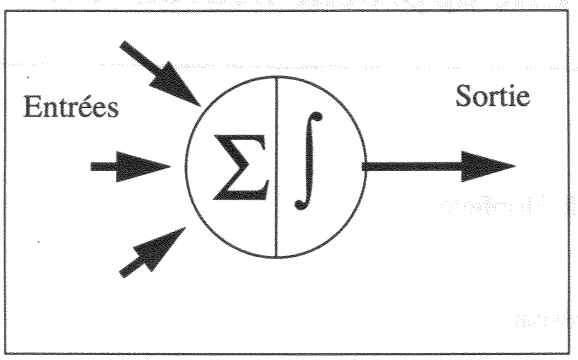

Figure 14 : Neurone formel

Le comportement de ce réseau est modélisé par une valeur globale qui est en fait un vecteur composé des valeurs de chacun des neurones à un instant donné. La valeur de ce vecteur permet d'associer une énergie globale à l'état actuel du réseau. C'est l'évolution de cette énergie au cours du temps qui est utilisée pour résoudre les problèmes d'optimisation.

4.1.2. Energie du réseau

La fonction d'énergie est paramètrée par les entrées externes et les poids des connexions. A tout instant, elle est calculée en fonction de ces paramètres et de la valeur des neurones à cet instant grâce à la formule :

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij}V_jV_i - \sum_{i=1}^{n} V_iI_i$$

avec

- T_{ij} poids entre le neurone *i* et le neurone *j*.
- I_i valeur de l'entrée externe du neurone *i*.
- V_i valeur du neurone *i*.

Cette fonction d'énergie décroît au cours du temps et le réseau converge alors vers un état stable qui correspond à la valeur minimale pour l'énergie. Ainsi, en faisant coïncider cette fonction d'énergie à un objectif à minimiser, cette convergence correspond à une optimisation. Lorsque l'état stable est atteint, la valeur du réseau indique l'état du système pour la valeur optimale.

L'état initial du réseau dépend des valeurs fournies en entrée. Il évolue ensuite vers un état stable. Cette convergence est liée à une perte d'énergie et correspond à une modification de la valeur des neurones au cours du temps. Afin de garantir une convergence vers un état stable d'une modélisation d'un tel réseau, certaines propriétés exprimées par Cohen et Grossberg sont nécessaires [Cohen 83] :

- La matrice de poids doit être symétrique (pour tout i, j : $T_{i,j} = T_{j,i}$).
- Il ne doit pas y avoir d'auto-connexions (pour tout i : $T_{i,i} = 0$).
- Il ne doit pas y avoir de connection positive (pour tout i, j : $T_{ij} \leq 0$).
- La fonction de seuillage doit être strictement croissante.

Le réseau se stabilise donc pour une valeur minimale de la fonction d'énergie dont le profil est connu et qui peut être adaptée en modifiant les entrées externes et les poids des connexions. Ainsi, en faisant correspondre la fonction d'énergie à une fonction de contraintes, la stabilisation aura lieu quand les contraintes seront optimisées.

Hopfield a utilisé cette propriété pour modéliser des problèmes sous forme d'une fonction de contraintes à minimiser. La fonction de contrainte doit être mise sous une forme particulière afin de la faire correspondre à une fonction d'énergie d'un réseau. Il ne reste plus alors qu'à construire ce réseau et à le faire se stabiliser pour extraire les valeurs pour lesquelles la fonction de contrainte est minimale. Ce travail a été réalisé pour résoudre le problème du voyageur de commerce [Hopfield 85].

4.1.3. Représentation du problème

Nous sommes donc confrontés à un problème d'optimisation sous contraintes. La première étape consiste à représenter le problème topologiquement. En effet, il faut modéliser le problème par un réseau de Hopfield, où chaque neurone a une signification particulière (par exemple, il peut indiquer l'état d'une ressource à un instant donné : occupée ou libre). Il faut ensuite définir les contraintes à appliquer sur ce réseau de manière à coder le problème. Ces contraintes sont alors mises sous la forme d'une fonction d'énergie d'un réseau de Hopfield afin de calculer les paramètres du réseau (poids des connections et valeurs des entrées). Une fois ce codage réalisé, les itérations sont effectuées jusqu'à ce que le système converge vers un état stable. Il ne reste plus alors qu'à décoder le résultat en fonction des valeurs des neurones pour le minimum d'énergie obtenu en fin d'itération.

4.1.4. Modélisation des contraintes

Taggliarini et Christ [Tagliarini 91] ont utilisé le modèle de Hopfield pour résoudre un problème très simple mais qui peut servir de base au codage de problèmes plus complexes. L'idée est de positionner dans un ensemble total de n variables booléennes, un certain nombre k de variables à la valeur 1 et d'imposer la valeur 0 aux autres. Nous verrons par la suite comment de telles contraintes peuvent se composer et permettre de coder des problèmes plus complexes.

- **La règle *k de n***

Dans cet exemple, la topologie est simple, il suffit de faire correspondre un neurone à chaque variable et d'appliquer les contraintes sur le vecteur ainsi défini. Dans notre cas, si l'on considère V_i comme la valeur du $i^{ème}$ neurone, la contrainte est la suivante :

$$\sum_{i=1}^{n} V_i = k \qquad \text{avec} \qquad V_i = 0 \quad \text{ou} \quad V_i = 1$$

Le fait d'imposer la valeur des neurones à 0 ou à 1 revient à dire qu'il faut que le produit de leur valeur V par (1-V) doit être nul. Par conséquent, la contrainte $V_i = 0$ *ou* $V_i = 1$ *pour tout* i est équivalente à dire que :

$$\sum_{i=1}^{n} V_i(1 - V_i) = 0$$

De même imposer que la somme des V_i soit égal à k revient à dire que :

$$k - \sum_{i=1}^{n} V_i = 0$$

et donc qu'il faut minimiser la valeur de :

$$\left(k - \sum_{i=1}^{n} V_i \right)^2$$

Si l'on reprend la contrainte initiale qui était d'avoir une somme égale à k avec des valeurs de V_i égales à 0 ou 1, résoudre cette contrainte revient à minimiser la fonction :

$$E = \left(k - \sum_{i=1}^{n} V_i \right)^2 + \sum_{i=1}^{n} V_i(1 - V_i)$$

qui, après transformations algébriques [Tagliarini 91], se réécrit sous la forme ;

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} (-2) V_i V_j - \sum_{i=1}^{n} V_i(2k - 1)$$

Cette fonction peut s'interpréter comme une fonction d'énergie d'un réseau de Hopfield composé d'une matrice de n neurones ayant les poids des connexions définis par :

$$T_{ij} = \begin{cases} -2 & \text{si i} \neq \text{j} \\ 0 & \text{sinon} \end{cases}$$

et des entrées externes égales à :

$$I = 2k - 1 \qquad \text{Pour tout i}$$

Ce type de contrainte s'exprime donc facilement par un modèle de Hopfield. De plus, en ajoutant des neurones liés aux contraintes, il est possible de définir des contraintes de type «au moins» ou «au plus».

- **La règle *au moins k de n***

Soit, par exemple, un ensemble de trois variables sur lequel au moins deux variables doivent être fixées à 1 : le problème se code par trois neurones représentant les trois variables auxquels est ajouté un quatrième neurone, appelé neurone de relâchement (*slack neuron*). Ce dernier neurone sert à contrebalancer le flou dû à la contrainte «au moins», c'est-à-dire qu'il vaut 1 dans le cas où deux variables sont fixées à 1 et 0 dans le cas où trois variables sont égales à 1. Dans la figure 15, les quatre neurones et la contrainte *3 de 4* expriment le fait qu'au moins 2 neurones doivent être actifs parmi les 3 premiers.

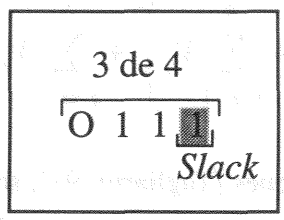

Figure 15 : Contrainte «*au moins*»

• **La règle *au plus k de n***

De la même façon, soit un ensemble de trois variables sur lequel au plus deux variables doivent être fixées à 1 : le problème se code par trois neurones représentant les trois variables auxquels sont ajoutés deux neurones qui composent la zone de relâchement (*slack*). Ces deux derniers neurones servent à contrebalancer le flou dû à la contrainte «au plus», c'est-à-dire qu'ils valent 1 dans le cas où les trois variables sont fixées à 0, ou un des deux vaut 1 si une seule variable est fixée à 1, ou alors ils valent 0 dans le cas où les trois variables sont égales à 1. Dans la figure 16, les cinq neurones et la contrainte *2 de 5* expriment bien le fait qu'au plus 2 neurones doivent être actifs parmi les 3 premiers.

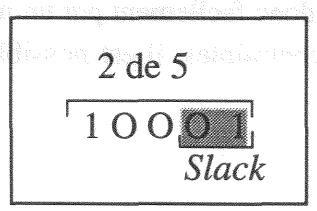

Figure 16 : Contrainte «*au plus*»

• **La règle *succ k de n***

Dans [Cardeira 94b], Cardeira utilise ce type de réseau pour l'expression et la résolution de problèmes simples de planification de tâches préemptives. Il propose également une extension à la règle de *k de n* : la règle *succ k de n* qui, dans un vecteur de *n* neurones, active k éléments successifs. L'auteur utilise cette règle pour s'attaquer au problème d'ordonnancement de tâches non-préemptives. Pour cela, il utilise des neurones inhibiteurs qu'il définit de la manière suivante :

> «Un neurone inhibiteur est un neurone qui peut inhiber une connexion en fonction de son état. Le neurone inhibiteur peut être positif ou négatif selon qu'il bloque la connexion quand il est à 1 ou quand il est à 0.»

La règle *succ k de n* s'exprime alors en fixant les valeurs des connexions entre les éléments du vecteur d'entrée auquel la règle est appliquée de la façon suivante :

$$T_{ij} = \begin{cases} 0 & \text{si li - jl} \leq 1 \text{ pour tout i,j} \\ -(4k+6) & \text{si li - jl} > 1 \text{ pour tout i,j} \\ -2inh(x_{i+1}, x_{j+1}) & \text{si li - jl} = 1 \text{ pour tout i,j} \neq n \\ -2inh(x_{i-1}, x_{j-1}) & \text{si li - jl} = 1 \text{ pour tout i,j} \neq 0 \end{cases}$$

où *inh*(a,b) indique que les neurones a et b sont inhibiteurs.

Les entrées externes étant égales à :

$$I_i = (2k-3)(2k-1) \qquad \text{Pour tout i}$$

Cette règle est intéressante puisqu'elle permet de passer simplement de l'ordonnancement de tâches non préemptives à l'ordonnancement de tâches préemptives. Cependant, l'ajout de neurones inhibiteurs augmente considérablement la complexité puisque chaque mise à jour de neurone nécessite, en plus du calcul de la somme pondérée, des multiplications supplémentaires pour prendre en compte l'effet des neurones inhibiteurs. Aussi, une représentation différente du problème permettant la prise en compte de la non-préemptivité des tâches sans l'utilisation de cette règle sera plus efficace.

Il est ainsi possible de modéliser un problème donné par un ensemble de variables booléennes sur lesquelles les contraintes sont appliquées à l'aide de règles du type :

- exactement *k* variables doivent être vraies parmi un ensemble donné de taille *n*,
- au moins *k* variables doivent être vraies parmi un ensemble donné de taille *n*,
- au plus *k* variables doivent être vraies parmi un ensemble donné de taille *n*,
- exactement *k* variables successives doivent être vraies parmi un ensemble donné de taille *n*

Plusieurs contraintes peuvent être définies sur un même ensemble de variables, ce qui permet d'appliquer des contraintes plus complexes en les exprimant sous forme de plusieurs règles. Nous verrons dans le chapitre suivant un exemple permettant de traiter des problèmes d'ordonnancement par l'utilisation d'une telle modélisation.

4.2. Paramètrage

La modélisation du problème par une fonction d'énergie à optimiser permet l'obtention d'un réseau de Hopfield ainsi que ses paramètres, c'est-à-dire le poids de ses connexions et la valeur de ses entrées externes. Cependant, il reste des paramètres à définir afin de déterminer la manière dont le parallélisme est utilisé et/ou simulé et la façon dont les contraintes sont combinées.

4.2.1. Simulation du parallélisme

A l'origine, le réseau est sensé réaliser toutes les mises à jour des neurones de façon parallèle mais cela nécessite l'utilisation d'un processeur par neurone, ce qui dans bien des cas n'est pas envisageable. Par conséquent, le parallélisme doit être simulé, c'est-à-dire qu'il faut réaliser des choix d'implantation pour pallier au manque de processeurs. Différents choix sont envisageables :

- Mode synchrone. Il s'agit de réaliser une simulation qui se rapproche le plus possible du fonctionnement parallèle. Cela consiste donc à réaliser une copie du vecteur de neurones à l'instant t et à calculer la nouvelle valeur à l'instant $t+1$ à partir de la copie.

- Mode asynchrone. Dans ce cas, les valeurs les plus récentes sont utilisées à chaque itération. La mise à jour du vecteur de n neurones est effectuée au fur et à mesure des calculs. Quand le vecteur est parcouru séquentiellement pour i allant de 1 à n, la valeur à l'instant t du $i^{ème}$ neurone dépend donc de la valeur à l'instant t des $i-1$ premiers neurones et de la valeur à l'instant $t-1$ des $n-i$ derniers neurones.

- Mode bloc-séquentiel. Il s'agit de la technique intermédiaire, qui consiste à fractionner le vecteur de neurones en blocs. A l'intérieur d'un bloc, les calculs se font en parallèle. Les blocs, quant à eux, sont activés de façon séquentielle en effectuant les mises à jour régulièrement dans le vecteur. Cependant, avec des blocs de taille 1, ce mode est équivalent au mode asynchrone. Par contre, si le vecteur est décomposé en un seul bloc, le fonctionnement est alors identique au mode synchrone.

4.2.2. Stratégie d'activation

Le fonctionnement en mode asynchrone consiste donc à calculer les nouvelles valeurs des neurones en fonction des valeurs disponibles les plus récentes. Par conséquent, l'ordre d'activation peut influencer le résultat final. Si le but est de simuler au mieux le parallélisme, les calculs parallèles devraient être effectués dans un ordre aléatoire.

Cependant, l'ordre d'activation peut être utilisé pour diminuer le nombre d'itérations nécessaire à la convergence. Il peut également permettre d'ajouter un critère de sélection lorsque plusieurs solutions existent et correspondent à différents états stables du réseau. Par exemple, dans le cas d'un ordonnancement, en activant les neurones dans l'ordre chronologique, les solutions «au plus tôt» sont privilégiées. Si le sens contraire est choisi, l'optimisation se fait selon un critère «au plus tard», à condition bien sûr que plusieurs solutions résolvent toutes les contraintes imposées à l'ordonnancement.

4.2.3. Pondération des contraintes

Un autre paramètre important concerne la façon dont est construite la fonction d'énergie à partir des différentes contraintes. En effet, chaque contrainte correspond à une fonction à minimiser, l'objectif étant de minimiser cette somme globale. Cependant, une somme simple peut ne pas être satisfaisante, dans la mesure où certaines contraintes peuvent prendre des valeurs plus élevées et donc avoir une influence plus grande (le résultat est l'obtention d'une fonction d'énergie qui possède beaucoup d'optima locaux). Ainsi, il est possible de pondérer les contraintes afin de leur affecter une importance égale et d'obtenir une fonction d'énergie moins accidentée.

4.3. Des résultats progressifs

Rappelons que le problème auquel nous sommes confrontés est de complexité exponentielle. A chaque ensemble n de tâches correspond $n!$ possibilités de les ordonner. Nous savons qu'il n'existe pas d'algorithme polynomial capable de résoudre le problème [Garey 79]. Par conséquent, notre but est d'obtenir une solution approchée dans un temps raisonnable. Le système défini dans ce chapitre permet d'obtenir un résultat après avoir effectué un certain nombre de cycles, correspondant à la mise à jour de l'ensemble du réseau. La qualité de la solution ainsi obtenue se mesure de façon binaire: un ordre est correct si toutes les tâches respectent leur échéance et incorrect sinon. Cependant, une mesure de la qualité peut être réalisée si l'on considère la probabilité que l'ordre fournit soit correct. C'est pourquoi, pour comparer deux systèmes qui fournissent une solution approchée au problème considéré, des tests sont effectués afin de savoir quelle est la probabilité pour chacun d'eux de fournir une séquence correcte.

Chercher une solution par notre système revient donc à passer un contrat au sens défini dans la partie 1 au chapitre 3.3, page 56. Un certain laps de temps est alloué au système en échange de l'obtention d'un résultat d'une certaine qualité (plus exactement ici en échange d'une certaine probabilité d'obtenir une solution correcte). Par conséquent, en définissant plusieurs contrats, c'est-à-dire un ensemble de couples (durée d'exécution - qualité du résultat), la résolution revient à l'exécution d'un algorithme *anytime*. Celui-ci

possède ainsi des caractéristiques fondamentales pour s'intégrer dans un système fonctionnant en temps-réel.

Les paragraphes suivants montrent de quelle façon il est possible de définir des affinements permettant de paramètrer la définition de différents contrats.

4.3.1. Variation de la vitesse de convergence

L'étude du comportement du réseau au cours des itérations montre que les variations sont importantes à l'initialisation du système et s'amenuisent au fur et à mesure des itérations, les dernières ne servant qu'à stabiliser les valeurs des neurones déjà sélectionnés. Ainsi, les décisions se faisant en début de convergence, il est possible d'optimiser le fonctionnement en forçant les neurones choisis à prendre rapidement une valeur stable (0 ou 1).

Pour cela, on influe sur la fonction de seuillage utilisée par les neurones (voir paragaphe 4.1.1. , page 81). A tout instant, pour calculer sa valeur, chaque neurone effectue une somme pondérée de ses entrées puis effectue un seuillage afin de conserver une valeur dans l'intervalle [0, 1]. Pour cette dernière fonction, on utilise une tangente hyperbolique, puis on normalise le résultat pour parvenir dans l'intervalle voulu. Il est toutefois possible d'appliquer un coefficient à la somme avant de calculer la tangente. On a ainsi la fonction de seuillage suivante :

$$f(x) = \frac{1}{2}(1 + \tanh(u \cdot x))$$

Notons que cette fonction est strictement croissante et respecte ainsi la condition nécessaire à la convergence exprimée dans le paragraphe 4.1.2, page 82.

Dans cette équation, la variable u détermine la pente de la sigmoïde, c'est-à-dire qu'une valeur faible conduit à des valeurs continues sur l'intervalle [0,1], tandis qu'une valeur plus forte force le neurone à prendre une valeur qui se rapproche de 0 ou de 1. Ainsi, dans notre cas, il est intéressant de débuter les itérations avec une valeur relativement faible de u, puis d'augmenter progressivement sa valeur pour passer plus rapidement sur les itérations ne servant qu'à confirmer une décision déjà prise.

La valeur de l'incrémentation de la variable u au cours des itérations du réseau détermine le nombre d'itérations nécessaires à la stabilisation du système et donc son temps de calcul. Parallèlement, il détermine la probabilité de succès de la résolution.

Le fait de varier le nombre d'itérations est une technique analogue à *l'approximation en champ moyen* inspiré de la thermodynamique [Hérault 91]. Le principe de ce dernier est

de simuler l'évolution d'un système physique jusqu'à l'équilibre thermodynamique pour réaliser l'optimisation d'une fonction. Le paramètre utilisé est la température du système. Ainsi en faisant varier ce paramètre il est possible de simuler différentes techniques physiques telles que la trempe (refroidissement brutal) ou le recuit (refroidissement lent) et d'étudier les différents états ainsi obtenus, chacun d'eux correspondant à une solution de l'algorithme.

La façon dont la pente de la sigmoïde augmente au cours du temps permet la définition de différents contrats au sens défini dans la partie 3 au chapitre 3.3, page 56. Ainsi la dérivée de cette pente est un paramètre qui rend possible l'utilisation du système selon différents modes, chacun correspondant à une durée d'exécution et un espoir de succès qui lui sont propres. Il s'agit donc bien d'un système *anytime* dans le sens où le temps consacré au traitement est variable, a condition d'être fixé *a priori*. De plus la qualité du résultat est bien dépendante de la durée choisie.

4.3.2. Variation de la granularité

Dans le paragraphe précédent, nous venons de montrer comment, en modifiant le nombre d'itérations, il était possible d'obtenir des résultats plus ou moins rapides selon la qualité de résultat désirée ou plus exactement la marge d'erreur autorisée. Une autre possibilité de réaliser des approximations (c'est-à-dire un gain de temps au détriment d'une baisse de la qualité de la solution) consiste à modifier la durée de chaque itération. Pour cela rappelons qu'une itération correspond à la mise à jour de l'ensemble des neurones du réseau. Aussi, son temps de calcul est directement lié à la taille du réseau, plus exactement au nombre de neurones mis en œuvre. Ainsi, pour réaliser un gain de temps sur une itération, il faut nécessairement diminuer le nombre de neurones, donc le nombre de variables définissant le problème. Un tel choix conduit obligatoirement à la manipulation de variables plus générales, d'où un traitement plus «grossier» et une réponse moins précise. Nous allons pour cela utiliser la notion de granularité introduite dans le chapitre 2, page 35.

En conséquence, selon la granularité des informations des variables codées par le réseau, il est possible de définir différents temps d'exécution et différentes qualités de résultat associées. Cependant, ces affinements étant fortement dépendants du type d'information codé dans le système, il est impossible de les définir de façon générique. Nous verrons sur l'exemple d'ordonnancement de tâches de quelle façon la progressivité est mise en œuvre en jouant sur cette granularité.

4.4. Utilisation de la progressivité

Nous avons ainsi définis différentes possibilités pour réaliser des approximations afin de pouvoir disposer d'une représentation adaptée à des problèmes où le temps joue un rôle important. Il nous faut maintenant préciser la façon dont cette représentation peut être utilisée afin de produire un résultat de bonne qualité dans le temps imparti.

4.4.1. Recuit en champ moyen

Une possibilité est de s'inspirer de la thermodynamique pour simuler les techniques de recuits [Hérault 91]. On considère que notre système est maintenu à une température bien définie. L'état du système évolue en fonction de la température : il sera stable pour des températures basses et instable pour des températures élevées. Ainsi si l'on peut paramètrer la température, il est possible de modifier l'état du système.

Le technique consiste à chauffer le système, afin de le placer dans un état instable, puis à le laisser refroidir pour le placer dans un état stable et observer si ce dernier correspond à un état désiré. La façon dont le refroidissement est effectué peut entraîner de fortes modifications de l'état final obtenu. Il est en effet possible de le refroidir brutalement (ce qui correspond à une technique de trempe) ou de faire baisser la température très lentement (ce qui correspond à une technique de recuit).

Le recuit en champ moyen consiste à faire baisser la température selon un certain pas de décrémentation pendant un certain nombre d'étapes, correspondant à des paliers de température. Puis le système est placé dans un état stable par un refroidissement brutal et l'état final obtenu est observé. Tant que cet état n'est pas satisfaisant, le système est réchauffé et le refroidissement répété avec cette fois un nombre de paliers de température plus important.

Dans sa thèse [Hérault 91], Hérault propose la construction d'un réseau neuromimétique dont le fonctionnement est inspiré de cette méthode. Il utilise un modèle de Hopfield avec l'algorithme présenté dans la figure 17.

1. *Fixer la température T_0*
   Fixer le facteur de décroissance de la température entre deux mises à jour consécutives d'une même variable : decT = 0.995
   Fixer le nombre de mises à jour de l'ensemble des variables d'optimisation : Nbscan
   Tirer aléatoirement les valeurs initiales des variables
2. *Répéter pour n allant de 1 à Nbscan*
   Pour chaque neurone i du réseau, pris dans un ordre aléatoire
   Mettre à jour la valeur du neurone i
   Fin pour
   Faire décroître la température: T = decT .T
   Fin répéter
3. *Forcer les valeurs des Neurones à 0 ou 1*
   Si le système a convergé
   Alors déduire la solution de la valeur finale des neurones
   Sinon Nbscan est trop petit, retourner à l'étape 1
   Fin si

Figure 17 : Algorithme du recuit en champ moyen

Cette technique permet l'utilisation de diverses méthodes qui ont des durées d'exécutions de plus en plus longues, chacune ayant une probabilité plus forte d'éviter les minima locaux et donc d'obtenir un résultat correct. Elle agit ainsi par approximations successives, ce qui est une propriété fort intéressante dans un cadre temps réel.

Dans notre cas, le test qui permet de savoir si le système a convergé peut se faire rapidement au cours des itérations. Il suffit pour cela de mesurer la fonction d'énergie globale du réseau, sa variation diminue exponentiellement au cours des itérations. Ainsi lorsque l'énergie diminue très faiblement, le réseau a atteint un état stable (les dernières itérations ne servent qu'à faire passer les neurones d'une valeur proche de 1 à 1 et ceux ayant une valeur proche de 0 à 0). Par conséquent, le nombre de paliers (*Nbscan* dans l'algorithme du recuit) n'est pas fixé *a priori*, mais dépend du temps nécessaire à la convergence, plus exactement de la valeur du facteur de décroissance (*decT* dans l'algorithme du recuit). Chaque essai de convergence peut ainsi se faire avec un facteur de décroissance différent.

Rappelons que dans notre cas, la température correspond à l'inverse de la pente de la sigmoïde. L'algorithme que nous allons utilisé est donc celui décrit dans la figure 18.

1. *Fixer à une valeur forte le facteur de croissance de la pente entre deux mises à jour consécutives d'un même neurone incP*
   Tirer aléatoirement les valeurs initiales des neurones
2. *Fixer la pente initiale de la sigmoïde P_0*
   Tant que le système n'a pas convergé
   Pour chaque neurone i du réseau, pris dans un ordre aléatoire
   Mettre à jour la valeur du neurone i
   Fin pour
   Augmenter la pente de la sigmoïde P= incP .P
   Fin tant que
3. *Si la valeur de l'énergie finale est minimale*
   Alors déduire la solution de la valeur finale des neurones
   Sinon Diminuer la valeur de incP et retourner en 2.
   Fin si

Figure 18 : Notre Algorithme

Cette technique est intéressante puisqu'elle exécute une série de convergences qui ont de plus en plus de probabilités d'éviter les minima locaux. Cependant, elle s'avère relativement longue compte tenu des multiples essais répétés pour régler la pente de la sigmoïde. Notre but est donc de définir une stratégie dans la séquence d'essais effectués afin de maximiser la probabilité de succès tout en minimisant le nombre d'essais et donc la durée d'exécution. Pour cela, il faut dans un premier temps isoler les différents essais, que nous appellerons des contrats, au sens défini dans le paragraphe 2.2, page 38 de la partie 1. C'est l'objet du paragraphe suivant.

4.4.2. Contrats

A l'opposé des méthodes de recuit, une autre technique consiste à définir un ensemble de contrats, chacun correspondant à une certaine valeur du pas de décrémentation de la courbe de la sigmoïde. Par conséquent un contrat est en fait une convergence du réseau d'un état initial instable vers un état stable, soit par analogie avec la thermodynamique, le refroidissement du système.

Chaque contrat possède une valeur de décrémentation de la sigmoïde qui lui est propre et qui définit ses performances en terme de pourcentage de succès et en terme de temps de calcul. Il est possible d'estimer les performances des différents contrats définis afin

de proposer une politique d'enchaînement de ceux-ci qui ait la plus forte probabilité de produire un résultat correct tout en respectant un temps d'exécution donné. Cette politique est fortement dépendante du domaine puisque les résultats des contrats peuvent largement différer d'une application à l'autre, aussi sera-t-elle étudiée dans le chapitre suivant sur un problème spécifique : celui de l'ordonnancement de tâches non préemptives.

5. Ordonnancement par réseaux neuromimétiques

Le problème auquel nous nous intéressons est l'allocation de tâches non préemptives sur une ou plusieurs ressources non partageables et non consommables. Chaque tâche est caractérisée par une date de début, une date de fin et une durée (voir figure 19).

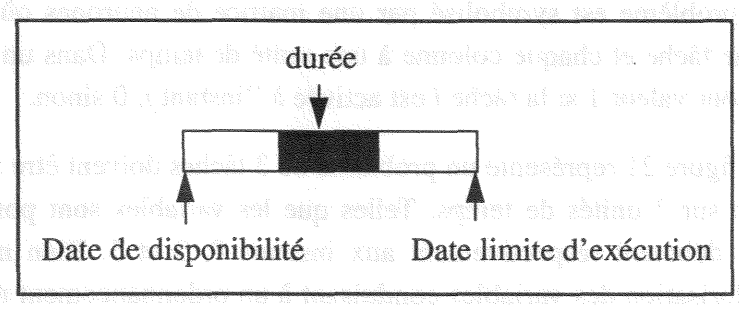

Figure 19 : Tâche à ordonnancer

Ainsi, une solution d'un problème où 3 tâches doivent partager une ressource peut être représentée par un diagramme de Gantt tel que celui présenté dans la figure 20. Sur cet exemple, la tâche 3 est exécutée en premier, suivi de la tâche 2 et enfin de la tâche 1.

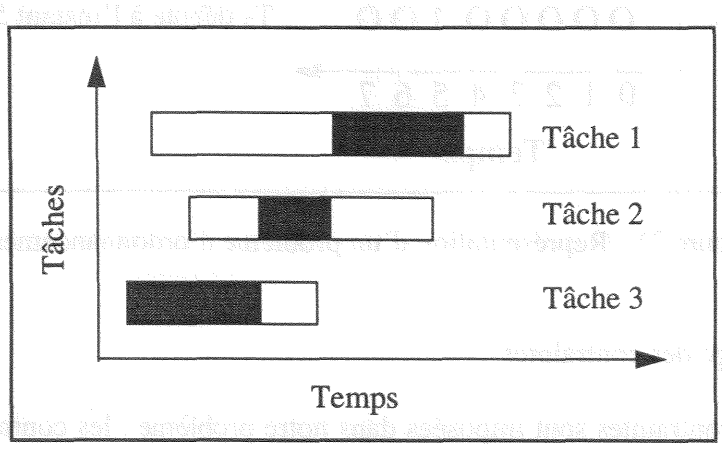

Figure 20 : Ordonnancement de 3 tâches sur une machine

Il est intéressant d'étudier dans quelle mesure le modèle de Hopfield, présenté dans le chapitre précédent, peut permettre de résoudre notre problème d'allocation de tâches.

5.1. Représentation

La première étape consiste à définir le problème topologiquement, c'est-à-dire à trouver un réseau de neurones qui puisse représenter le problème à un instant donné et sur lequel il sera possible d'imposer des contraintes afin de spécifier l'objectif à atteindre.

Le but du problème d'ordonnancement est de définir à quel moment chaque tâche doit être activée. Ainsi, le problème peut être décrit par un ensemble de variables booléennes où chacune représente le fait que la tâche i débute à l'instant t. En discrétisant le temps en n unités, le problème est symbolisé par une matrice de neurones où chaque ligne correspond à une tâche et chaque colonne à une unité de temps. Dans un état stable, le neurone (i,j) a pour valeur 1 si la tâche i est activée à l'instant j, 0 sinon.

Par exemple, la figure 21 représente un problème où 3 tâches doivent être allouées à une même ressource sur 7 unités de temps. Telles que les variables sont positionnées, les tâches 1 2 et 3 débutent respectivement aux instants 0, 3 et 5. Rien n'interdit pour l'instant une valorisation des variables conduisant à un ordonnancement non réalisable, cette vérification est réalisée à l'aide des contraintes imposées sur la matrice.

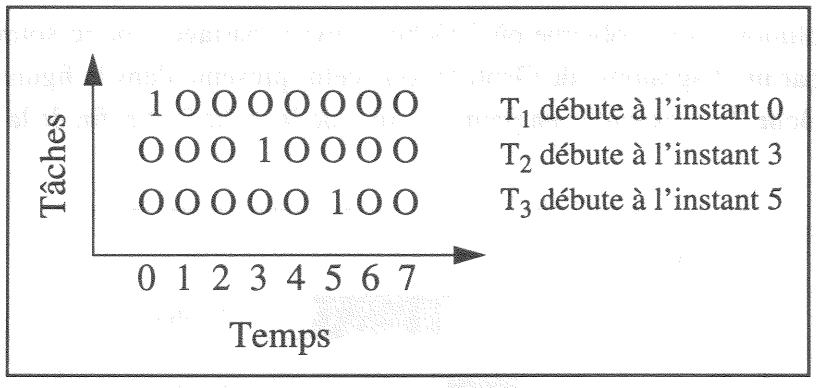

Figure 21 : Représentation d'un problème d'ordonnancement

5.2. Codage des contraintes

Deux types de contraintes sont imposées dans notre problème : les contraintes liées aux tâches elles-même et les contraintes liées à la ressource à partager.

Une contrainte propre à une tâche consiste à dire que l'exécution ne peut débuter avant la date r_i à laquelle la tâche i est disponible et doit se terminer avant la date limite Δ_i. Ceci s'exprime simplement à l'aide d'une règle de type *1 de n* entre la date de début au plus tôt et la date de début au plus tard (soit la date limite diminuée de la durée de la tâche), comme le montre la figure 22. Il y a ainsi une contrainte de ce type pour définir chacune des fenêtres temporelles, c'est-à-dire une par tâche à ordonner.

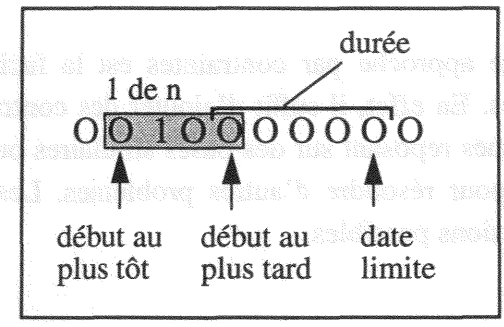

Figure 22 : Contrainte liée à une fenêtre temporelle

Une fois ces contraintes exprimées, la spécification complète du problème doit prendre en compte le partage d'une ressource commune par toutes les tâches. Cela consiste à préciser qu'à tout instant, il y a au plus une tâche active. Il faut pour cela ajouter des neurones de relâchement (*slack neurons*) qui donnent une indication sur l'état d'activation de la ressource.

La ressource est représentée par un vecteur de neurones dont la taille correspond au nombre d'unités de temps dans l'horizon du planning (le temps total sur lequel l'ordonnancement est effectué). A chaque instant, une contrainte de type *1 de n* exprime le fait que soit une tâche i a été activée dans les p_i unités de temps précédentes (p_i correspond à la durée de la tâche i), soit la ressource est inactive à cet instant (voir figure 23). Ainsi, en exprimant cette contrainte à chaque instant où la ressource est disponible (soit *a priori* à chaque unité de temps), le problème codé correspond bien à celui qui a été initialement énoncé.

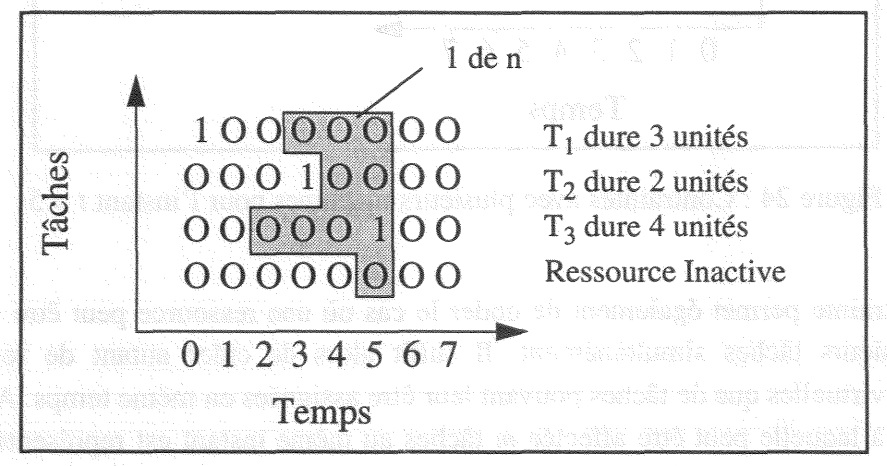

Figure 23 : Contrainte liée à la ressource pour l'instant $t = 5$

5.3. Extensions

L'un des avantages d'une approche par contraintes est la facilité d'extension à des problèmes plus complexes. En effet, il suffit d'ajouter des contraintes supplémentaires pour résoudre des problèmes reposant sur des bases similaires ou, dans certains cas, de modifier les contraintes pour résoudre d'autres problèmes. Les paragraphes suivants présentent quelques extensions possibles.

5.3.1. Ressources parallèles

Dans le cas où l'on dispose non plus d'une seule ressource, mais de k ressources identiques en parallèle, il suffit d'ajouter un vecteur de relâchement par ressource, et de réaliser un *k de n* sur les contraintes de ressource. La contrainte d'allocation unique exprimée à chaque unité de temps indique désormais que k ressources sont disponibles et effectue les affectations correspondantes : à tout instant t, l tâches sont actives ($l < k$) et k - l ressources demeurent disponibles. La figure 24 montre le type de contrainte à exprimer avec 2 machines pour 3 tâches.

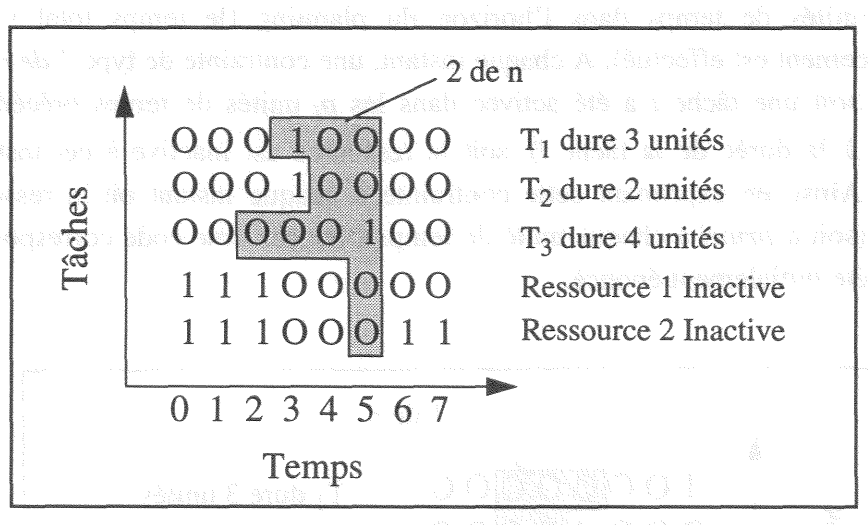

Figure 24 : Contraintes avec plusieurs machines pour l'instant $t = 5$

Cette contrainte permet également de coder le cas où une ressource peut être partagée entre plusieurs tâches simultanément. Il suffit alors de créer autant de ressources parallèles virtuelles que de tâches pouvant leur être assignées en même temps. Ainsi une ressource à laquelle peut être affectée m tâches au même instant est représentée par m lignes de relâchement de la même façon que si m ressources non partageables étaient disponibles.

5.3.2. Ressources multiples

Supposons maintenant qu'une tâche nécessite plusieurs ressources simultanément pour s'exécuter, chacune pouvant être utilisée pendant une durée différente. L'activation d'une tâche ne pourra se faire que si toutes les ressources dont elle a besoin sont disponibles au même instant. Les ressources seront libérées au fur et à mesure de l'exécution selon leur durée d'allocation à la tâche.

Dans ce cas, il suffit d'exprimer une contrainte par ressource nécessaire. La conjonction de ces contraintes signifie que si une des tâches est activée alors les ressources qui lui sont nécessaires sont indisponibles pendant leurs durées d'allocation respectives.

Sur l'exemple de la figure 25, la tâche T_1 a besoin pour s'exécuter de la ressource R_1 pendant 3 unités de temps et de R_2 pendant 2 unités de temps, T_2 a besoin pendant 2 unités de temps de R_1 et 4 de R_2, T_3 nécessite R_1 sur 4 unités de temps et R_2 sur 3. A tout instant, ces contraintes sont exprimées à l'aide de deux règles de type *1 de n* agissant sur le même ensemble de neurones (afin de faire apparaître plus lisiblement les contraintes, cet ensemble est dupliqué sur la figure).

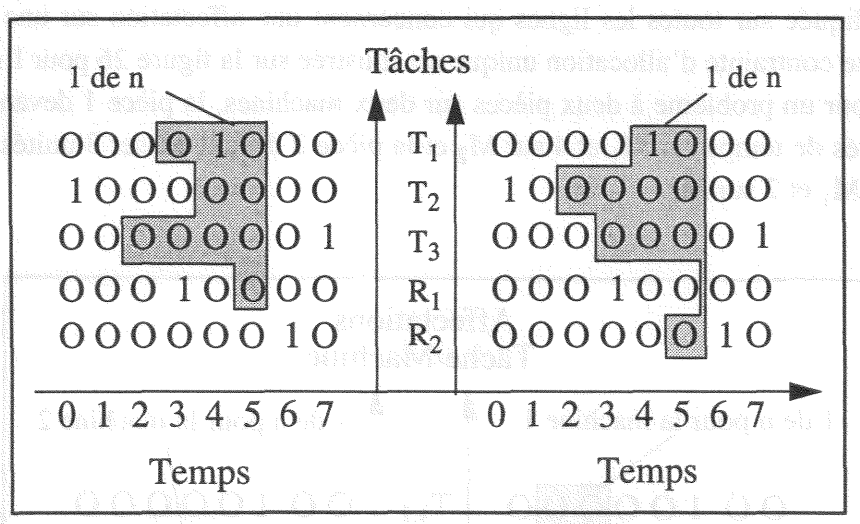

Figure 25 : Contraintes sur deux ressources pour l'instant $t = 5$

5.3.3. Ordonnancement d'atelier

Il est également possible de coder un problème de type ordonnancement d'atelier (*Open Shop Scheduling*) où l'on dispose d'un certain nombre de machines et d'un certains nombre de produits à réaliser. Chaque produit est déterminé par une gamme, c'est-à-dire un ensemble de tâches de durées variables à réaliser sur les différentes machines. Le but est alors d'ordonner les différentes tâches (les différents passages des pièces sur les

machines). Une machine ne peut traiter plusieurs pièces à la fois et une pièce ne peut se trouver sur plusieurs machines simultanément.

Ce cas est assez éloigné du premier problème traité. Ainsi, il demande plusieurs adaptations. Dans un premier temps, la représentation doit être modifiée. En effet, les variables ne sont plus une simple affectation <tâche / temps> pour une machine donnée, mais une affectation triple <tâche / temps / machine>. Par conséquent, la matrice de neurones est constituée de m (le nombre de machines) lignes par tâche et d'un nombre de colonnes égal au nombre d'unités de temps. Trois types contraintes sont définies :

- Le premier type de contrainte s'exprime pour chacune des lignes. Cela permet la définition de la fenêtre temporelle (voir figure 22, page 99). En effet, chacune des pièces peut avoir une date d'arrivée avant laquelle elle est indisponible et une date limite avant laquelle les traitements doivent être effectués. Cette contrainte exprime cet état de fait et doit par conséquent être appliquée à toute les lignes d'un produit, c'est-à-dire à toutes les affectations pièce / machine possibles.

- Le deuxième type de contrainte est relatif au fait qu'une machine ne peut usiner qu'une seule pièce à la fois. Pour exprimer cela, une règle de type *1 de n* est appliquée sur toutes les lignes qui concernent une affectation sur une machine. Cette contrainte d'allocation unique est illustrée sur la figure 26 pour l'instant t = 5 pour un problème à deux pièces sur deux machines, la pièce 1 devant passer 2 unités de temps sur M_1 et 4 sur M_2 et la pièce 2 devant passer 3 unités de temps sur M_1 et 2 sur M_2.

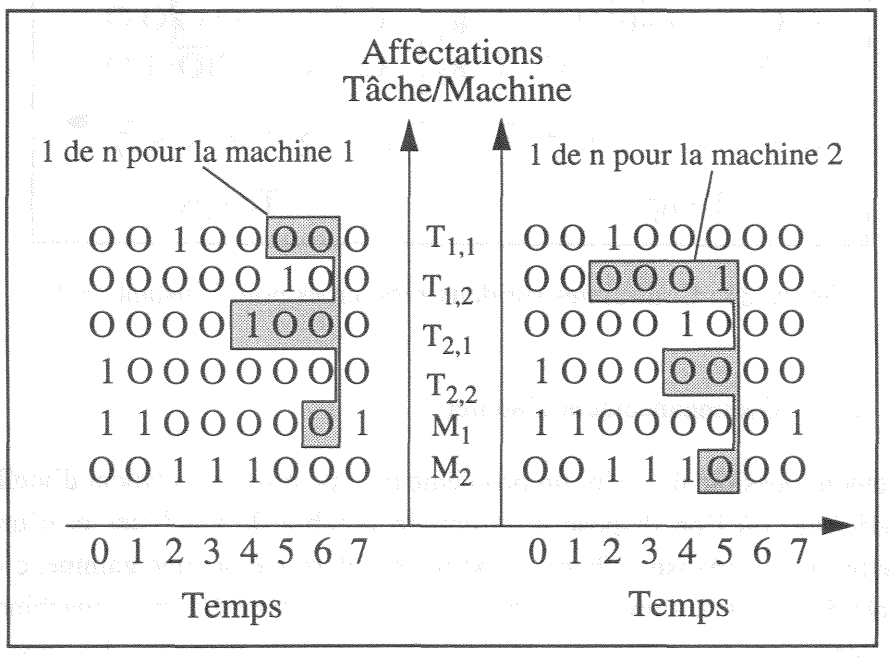

Figure 26 : Contraintes d'allocation de ressources pour l'instant t = 5

• Les deux types de contraintes précédentes ne permettent pas de garantir l'allocation unique d'une pièce pour un instant donné. C'est pourquoi, pour chaque tâche, une contrainte supplémentaire est nécessaire afin de s'assurer qu'une même pièce n'est pas usinée par deux machines différentes simultanément . Cette contrainte est du type *au plus 1 de n*. Elle nécessite un neurone de relâchement qui a comme sémantique : «la pièce n'est sur aucune machine à l'instant *t*». Ainsi, pour représenter cette contrainte à chaque instant, il faut un vecteur de neurones d'une taille égale au nombre d'unités de temps. La figure 27 présente l'exemple d'une tâche qui à tout instant peut être soit affectée à la machine M_1 pour 2 unités de temps, soit affectée à la machine M_2 pour 4 unités de temps, soit non affectée. Cette contrainte s'exprime par une règle de type *1 de n* sur chacune des unités de temps de l'horizon.

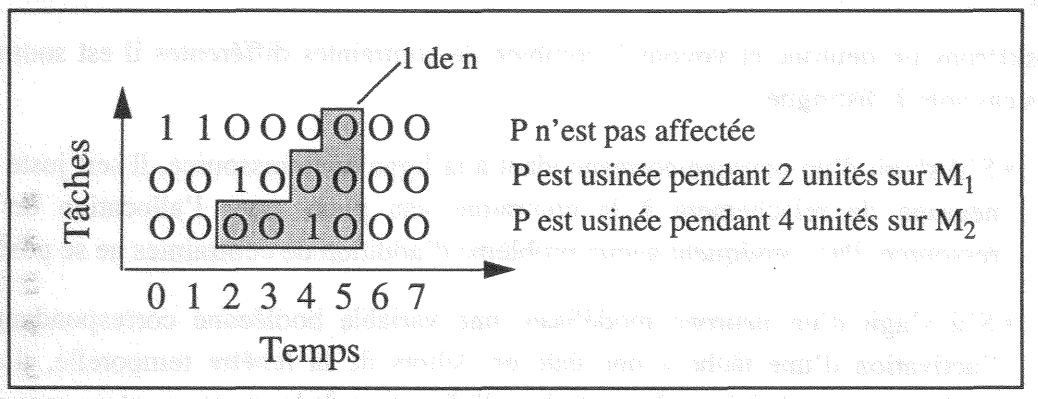

Figure 27 : Contrainte d'allocation de la pièce P à l'instant $t = 5$

Finalement, le réseau possède m+1 lignes pour chaque pièce à traiter et une colonne par unité de temps. Chaque ligne possède une contrainte qui définit sa fenêtre temporelle et à chaque instant est associé une première contrainte qui assure l'unicité de l'allocation des machines et une seconde pour l'unicité de l'allocation des pièces.

5.4. Paramètrage

Une fois les variables et les contraintes spécifiées, il est possible de définir précisément le réseau de neurones nécessaire ainsi que les poids affectés aux connections. Cependant, il reste tout de même quelques choix à réaliser pour effectuer l'implémentation. En effet, comme explicité dans le paragraphe 5.2, page 98, deux types de contraintes sont à appliquer : les unes sont liées à la définition de la fenêtre temporelle de la tâche, les autres à l'allocation de la ressource. Il faut donc choisir la façon dont ces contraintes sont combinées afin de les satisfaire simultanément sans favoriser l'une ou l'autre.

De plus, dans de nombreux cas, la mise à jour des neurones ne peut se faire de façon parallèle. Il faut donc soit définir une séquence de mise à jour, soit déterminer une technique de simulation du parallélisme.

Les paragraphes suivants présentent les choix que nous avons effectués concernant notre application ainsi que les raisons de ces choix.

5.4.1. Pondération des contraintes

Chaque tâche est soumise à plusieurs contraintes qu'elle doit satisfaire afin d'obtenir un résultat final exploitable. Pour cela, il faut définir la façon dont est réalisée la combinaison de contraintes, à savoir si une simple addition suffit ou s'il faut procéder à une pondération auparavant.

Considérons un neurone et voyons à combien de contraintes différentes il est soumis. Trois cas sont à distinguer :

- S'il s'agit d'un neurone correspondant à la ligne de la ressource, il sert juste de neurone de relâchement à la contrainte «au plus» pour l'allocation de la ressource. Par conséquent aucun problème d'addition de contraintes ne se pose.

- S'il s'agit d'un neurone modélisant une variable booléenne correspondant à l'activation d'une tâche à une date en dehors de la fenêtre temporelle, il est également soumis à la seule contrainte d'allocation de la ressource et ne pose pas de problème.

- Le troisième cas est plus délicat. En effet, si le neurone considéré modélise une variable booléenne correspondant à l'activation d'une tâche dans sa plage horaire, le problème de cumul des contraintes se pose. Le neurone est soumis à une contrainte qui lui impose de se déclencher entre sa date de début au plus tôt et sa date de début au plus tard, mais il est également soumis à plusieurs contraintes d'allocation indiquant qu'il peut s'exécuter seulement si la ressource est disponible. En conséquence, pour une tâche qui dure p unités de temps, un neurone est soumis à p contraintes d'allocation et une seule contrainte de temps. Ainsi, la recherche rapide d'une solution risque fort de se terminer sur un optimum local simple correspondant au cas où tous les neurones du vecteur de relâchement sont actifs. Ceci revient à satisfaire p contraintes et à en violer une seule pour chacune des tâches. Des tests expérimentaux ont montré que cet état est souvent atteint si la convergence est rapide et que la durée des tâches est importante. Pour tenter de pallier à ce problème, il est possible de pondérer la contrainte liée à la fenêtre temporelle par un coefficient égal à la durée de la tâche à laquelle elle correspond, l'optimum local pourra ainsi être évité dans de nombreux cas.

En conclusion, un seul cas pose des problèmes qui peuvent être fréquemment évités par une simple pondération des contraintes avant d'en effectuer la somme. Nous allons donc adopter cette solution pour notre application. Ainsi chaque contrainte portant sur le respect de la date d'arrivée et d'échéance aura un poids égal à la durée de la tâche et chaque contrainte portant sur l'allocation unique de la ressource aura un poids de 1.

5.4.2. Simulation du parallélisme

Un autre choix important pour l'implémentation du réseau concerne la simulation du parallélisme. Les différents choix possibles ont été présentés dans le paragraphe 4.2.1, page 88.

Au sein de notre laboratoire, par l'intermédiaire du Centre Charles Hermite[1], nous disposons pour nos tests de quatre véritables machines parallèles composées chacune de 8 micro-processeurs, ce qui permet d'effectuer des calculs rapides dans le cas où le mode synchrone est choisi. Cependant, il s'avère que dans ce mode les neurones en concurrences oscillent longuement avant de se stabiliser et provoquent ainsi une augmentation du nombre d'itérations nécessaire. A l'inverse, en mode asynchrone, un neurone capte rapidement l'énergie et inhibe les autres aux étapes suivantes. En utilisant une machine mono-processeurs, le mode asynchrone reste en moyenne aussi rapide que le mode synchrone sur une machine a 8 micro-processeurs, pour un coût, en terme de puissance de calcul, moindre. Le mode asynchrone sera donc privilégié pour nos tests, mais il est important de ne pas oublier que le système peut être plus performant à condition de disposer d'un grand nombre de micro-processeurs en parallèle.

5.5. Simulation

En l'absence de véritable corpus, mais afin de tester la validité des différentes hypothèses, nous avons créé un générateur de test qui permet de simuler des ensembles de tâches à activer. Ce générateur est, dans un premier temps, couplé à un algorithme naïf afin de vérifier que les tests générés possèdent réellement une solution, c'est-à-dire qu'il existe un ordre d'activation des tâches pour lequel toutes les dates limites d'exécution sont respectées.

Le générateur est paramètré par le nombre de tâches à ordonner et la longueur de l'horizon de planification, la longueur moyenne des tâches étant constante pour l'ensemble des tests. Cependant, il existe une certaine corrélation entre ces deux paramètres. Ainsi, afin de conserver des tests ayant une difficulté similaire quel que soit

1. Le Centre Charles Hermite est un projet fédérateur de recherche en modélisation et Calcul à Hautes Performances de la région lorraine.

le nombre de tâches, l'horizon de planification est calculé de manière à ce que le rapport entre le nombre de tests faisables et le nombre de tests générés soit constant.

La figure 28 présente un exemple de test où 10 tâches ont été correctement ordonnées sur 37 unités de temps. Chaque ligne correspond à une tâche exceptée la première qui n'est active que si la ressource est inutilisée à un instant donné. Chaque colonne représente une unité de temps. Ainsi, chaque case informe sur l'état d'une tâche à un instant donnée : si c'est le début de l'activation, elle est noire, sinon elle est blanche. Les traits horizontaux situés dans les parties supérieures des cases signalent les fenêtres d'exécution de chacune des tâches. Enfin, les traits horizontaux situés au milieu des cases indiquent la durée d'une tâche. Ils sont placés immédiatement après l'activation afin de contrôler facilement si l'exécution arrive à son terme avant la date limite.

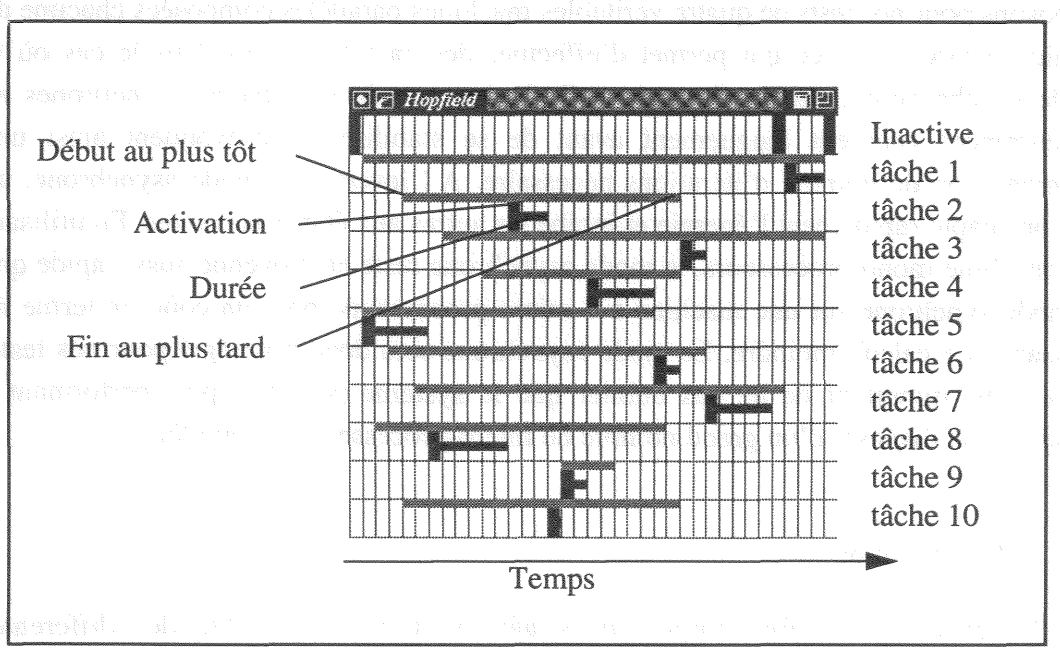

Figure 28 : Exemple d'exécution

Sur cet exemple, l'ordre d'activation des tâches est 5-8-2-10-9-4-6-3-7-1, les dates d'activations correspondantes étant respectivement 1, 6, 13, 16, 17, 19, 24, 26, 28, 34. La ressource est inutilisée aux instants 0, 33 et 37.

5.6. Performances

Notre méthode ne peut avoir d'intérêt que dans la mesure où les résultats produits sont de suffisamment bonne qualité et que les durées d'exécutions ne sont pas trop importantes. Une première série de tests est donc nécessaire afin de procéder à une telle évaluation. Ces tests sont juste donnés à titre indicatif, afin d'établir un ordre de

grandeur de la rapidité d'exécution et une idée des performances en terme de pourcentage de solutions correctes produites. L'implantation de l'algorithme n'est pas totalement conçu pour fonctionner de façon totalement optimale (par exemple aucune parallélisation n'est effectuée pour ces tests).

5.6.1. Comparaison avec des techniques exactes

La figure 29 présente un ensemble de tests réalisés sur une station de travail Sun 10-40 fonctionnant à 60 Specs[1]. Chaque ligne correspond à la moyenne d'une série de 10000 tests d'ordonnancement. Les deux premières colonnes spécifient le type de tests effectués, c'est-à-dire le nombre de tâches à ordonnancer et l'horizon de planification, de chacun des problèmes que l'on cherche à résoudre. Les trois colonnes suivantes présentent la durée moyenne de résolution des tests, exprimée en millisecondes, pour trois techniques de résolution :

- la première technique consiste à une simple énumération de toutes les solutions possibles jusqu'à ce que l'on en obtienne une qui soit réalisable. Cette technique sera évidemment très efficace pour des tout petits problèmes, puisque pour 3 tâches, il faut réaliser au maximum 6 tests. Par contre, le nombre de solutions étant égal à l'exponentielle du nombre de tâches, dès que celui-ci augmente, il est rapidement impossible d'utiliser une telle technique en un temps raisonnable.

- la seconde technique est un algorithme de propagations de contraintes dérivé de AC-7 [Mackworth 77] et de AC-3 [Bessière 95] proposé dans [Mouhoub 96]. Cela consiste à modéliser le problème sous forme d'un ensemble de contraintes, puis à propager ces contraintes afin de définir les restrictions à appliquer sur les dates d'activations de chacune des tâches. Puis on sélectionne une valeur pour chacune des tâches ayant plusieurs possibilités d'affectations et on examine les conséquences sur les dates d'activations des autres tâches jusqu'à l'obtention d'une solution complète ou jusqu'à ce qu'une tâche n'ait aucune solution. Dans ce cas on modifie une des affectations choisies et on réitère les propagations.

- la troisième technique consiste à utiliser notre système. Cette technique étant une technique approchée, il faut également mesurer le taux de réussite, c'est-à-dire le pourcentage de problèmes résolus. Cette mesure apparaît dans la dernière colonne.

Sur ces exemples, le rapport entre la durée de la tâche et sa fenêtre d'exécution (ce qui est appelé marge dans la littérature) était au maximum de 4.

1.Un Spec correspond à la puissance de calcul d'une machine VAX.

Tâches	Horizon	Itérative	Propagation	Hopfield	Réussite d'Hopfield
3	12	0.11	2.7	1.6	99 %
4	16	0.15	4.6	5.7	98 %
5	20	0.27	7.0	12.3	97 %
10	37	330	23.0	108	88 %
11	45	9987	37.6	270	89 %
13	60	420796	92.7	268	93 %
15	70	-	540	426	83 %
20	90	-	11064	1040	72 %

Figure 29 : Comparaison de durées d'exécutions de quelques méthodes

Comme on pouvait le supposer, pour des problèmes de petites tailles, il est plus efficace de chercher à construire une solution en cherchant à énumérer tous les cas possibles, ceux-ci étant peu nombreux. Cependant, dès que le problème prend des proportions, plus importantes, la technique de Hopfield est plus efficace. Notons que la technique est d'autant plus rapide que le taux de réussite est grand, puisque les cas d'échecs nécessitent un temps de traitement plus important (car dans ces cas de nombreux contrats sont exécutés) et font ainsi augmenter la durée moyenne. Ceci explique pourquoi, par exemple, le système est plus rapide pour 13 tâches sur 60 secondes que pour 11 tâches sur 45 unités de temps.

5.6.2. Comparaison avec une technique approchée

Les résultats des premiers tests indiquent que notre système semble intéressant dès lors que la taille du problème est importante. En effet, les temps de calcul restent raisonnables et les taux de réussite sont satisfaisants.

Afin de s'assurer de la compétitivité de notre méthode, une deuxième série de tests compare l'approche neuromimétique avec une méthode approchée, en l'occurrence la méthode heuristique de [Chu 92] présentée dans le paragraphe 3.2, page 76. Les résultats sont présentés dans la figure 31. Les deux premières colonnes spécifient le

problème traité, la troisième présente les résultats obtenus par un réseau de Hopfield, la quatrième ceux obtenus par la méthode heuristique.

Tâches	Horizon	Hopfield	Heuristique
3	18	99.8 %	96.1 %
4	24	99.7 %	95.3 %
5	30	99.5 %	94.7 %
6	36	99.5 %	94.3 %
7	42	99.5 %	93.8 %
8	48	99.6 %	93.6 %
9	54	99.6 %	92.9 %
10	60	99.1 %	92.5 %
15	90	95.0 %	87.6 %

Figure 30 : Comparaison des taux de réussite sur des exemples simples

Sur des cas relativement simples[1], les deux techniques donnent des résultats similaires, ceux du modèle de Hopfield étant tout de même supérieurs et se dégradent beaucoup moins avec l'augmentation du nombre de tâches.

Voyons maintenant les résultats sur un problème plus complexe. Sur les tests de la figure 31, l'horizon a été choisi en fonction du nombre de tâches de manière à ce que environ 50 % des exemples générés aléatoirement possèdent au moins une possibilité d'ordonnancement. Ainsi, sur le corpus crée par le générateur, seulement la moitié des énoncés ont été retenus et proposés en entrée des deux systèmes d'ordonnancement. Pour ces problèmes, la marge individuelle de chacune des tâche était inférieure à la durée (autrement dit, la fenêtre temporelle d'une tâche était au maximum égale à deux fois la durée d'exécution de celle-ci) et, sur les résultats finaux, la ressource à allouer était inactive en moyenne entre 1 à 3 unités de temps.

1.Ces exemples sont considérés simples car l'horizon est important compte tenu du nombre de tâches à planifier. Ainsi, 95 % à 99% des exemples générés possédaient une solution.

Tâches	Horizon	Hopfield	Heuristique
3	12	99.4 %	91.1 %
4	16	98.2 %	86.3 %
5	20	97.0 %	80.4 %
6	23	94.8 %	71.6 %
7	27	93.2 %	64.5 %
8	30	90.5 %	56.0 %
9	34	88.8 %	49.3 %
10	37	88.0 %	41.8 %

Figure 31 : Comparaison des taux de réussite sur des exemples complexes

Dans une situation fortement contrainte, les résultats de la technique heuristique chutent fortement, alors que ceux du modèle de Hopfield restent acceptables. Notre approche est donc particulièrement intéressante dans ces cas où la technique heuristique fait défaut.

Notons qu'il est inutile de réaliser des tests sur les temps d'exécutions si aucune parallélisation n'est effectuée sur le réseau neuromimétique. En effet, la méthode heuristique est forcément beaucoup plus rapide dans tous les cas, puisqu'elle est de complexité inférieure : pour un problème de n tâches à planifier sur p unités de temps, la complexité de la méthode heuristique sera en $O(n^2)$, alors que la complexité de notre système sera en $O(n^2.p^2.iter)$; *iter* étant le nombre d'itérations nécessaires à la convergence. Remarquons que la complexité liée à la construction du réseau est négligeable car en $O(n^2.p^2)$, cela consiste donc à réaliser une itération suppémentaire.

5.7. Progressivité

En utilisant les techniques détaillées dans le paragraphe 4.3, il est possible de définir ce système sous forme d'un ensemble d'algorithmes représentant chacun un contrat, c'est-à-dire une certaine probabilité d'obtenir une réponse correcte en échange d'un temps de traitement donné. Cela permet de fournir un système qui s'exécute de manière progressive et par conséquent interruptible à tout instant pour obtenir une solution. Cette dernière possède une qualité proportionnelle à sa durée d'obtention.

5.7.1. Vitesse de convergence

Une première façon de réaliser différentes approximations est directement liée au processus d'optimisation. Cela consiste à modifier la rapidité de la convergence du réseau de Hopfield. La convergence s'achève dès que le système est dans un état stable où chaque neurone possède une valeur extrême (0 ou 1). Comme on l'a vu dans le paragraphe 4.3.2, page 91, cette valeur dépend de la pente de la sigmoïde associée aux neurones. Or cette pente augmente au cours du processus de convergence, les neurones prendront donc plus ou moins rapidement leur valeur finale selon la façon dont cette pente augmente. C'est la valeur de la croissance de cette pente que l'on désigne par le terme *vitesse de convergence*.

La vitesse de convergence consiste ainsi à jouer sur la pente de la fonction de seuillage utilisée par les neurones. Ce paramètre est fort utile dans un cadre temps réel, puisqu'il permet de borner le nombre d'itérations de chaque contrat et donc de garantir un temps de réponse. Chacune des valeurs de ce paramètre peut ainsi correspondre à un niveau d'approximation donné, chaque niveau étant ainsi la façon dont la pente de la sigmoïde est augmentée au cours du temps.

La figure 32 présente des résultats d'exécutions (le ***performance profile*** au sens défini dans le paragraphe 3.1. , page 53 de la partie 1 de cette thèse) pour différents contrats. L'axe horizontal représente le nombre d'itérations, l'axe vertical représente le nombre de contraintes violées pour une itération donné, l'épaisseur de la courbe correspond à la valeur de l'écart type du nombre moyen de contraintes non satisfaites pour l'itération donnée. Ainsi chaque segment représente l'intervalle dans lequel la qualité du résultat a de fortes chance de se trouver.

Ces résultats ont été calculés sur un corpus de 100000 problèmes d'allocations de 6 tâches sur 23 unités de temps. Chaque tâche avait une durée comprise de 1 à 6 unités de temps. Chacun des problèmes proposés avait une solution satisfaisant toutes les contraintes et donc pour laquelle la courbe est égale à 0 à la dernière itération.

Les contrats sont numérotés de 1 à 8 dans l'ordre décroissant de la vitesse de convergence. Chaque contrat s'exécute en un nombre d'itérations moyen qui est presque égal au double du contrat précédent.

Sur ces différents graphiques, la notion de progressivité apparaît clairement, puisque pour chacun des contrats, le nombre de contraintes violées diminue au cours des itérations. Ainsi, pour chacune des valeurs de la vitesse de convergence, la qualité du résultat fourni est une fonction croissante du temps d'exécution.

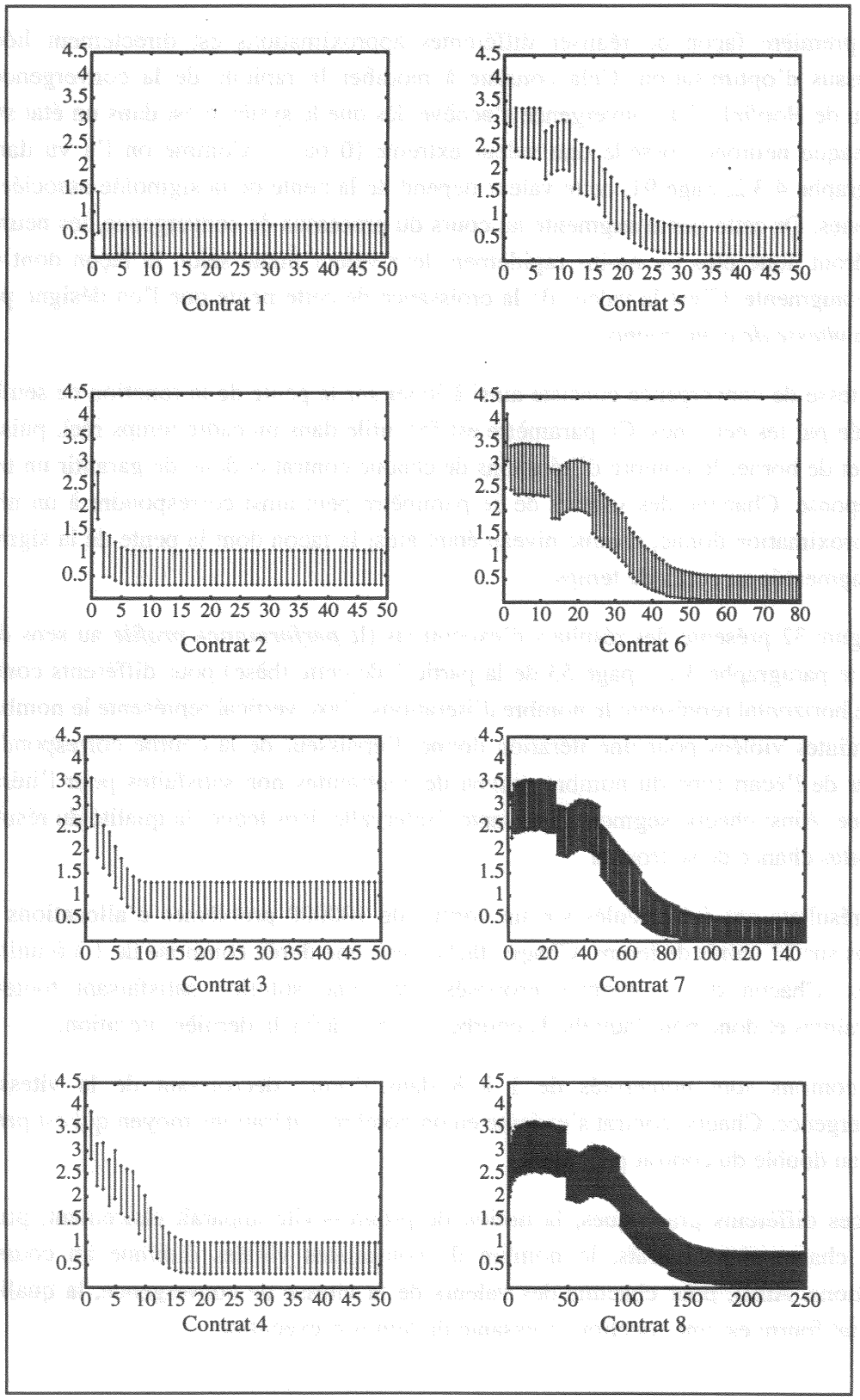

Figure 32 : Erreur moyenne pour différents contrats

Une autre possibilité de faire apparaître les résultats progressivement consiste à enchaîner les différents contrats. Ainsi, en exécutant les différentes approximations les unes à la suite des autres jusqu'à l'obtention d'un résultat correct, il est possible de calculer le pourcentage de problèmes résolus pour un nombre d'itérations donné et ainsi d'en déduire le pourcentage de chance de produire une solution correcte en fonction du nombre d'itération.

La figure 33 présente les taux de réussite sur un ensemble de tests de taille suffisamment important pour garantir un intervalle de confiance raisonnable (inférieur à 0.1%). Cette série de mesure permet de mesurer l'utilité de la séquence de contrats (le ***performance profile***) selon le temps de traitement qui lui est consacré.

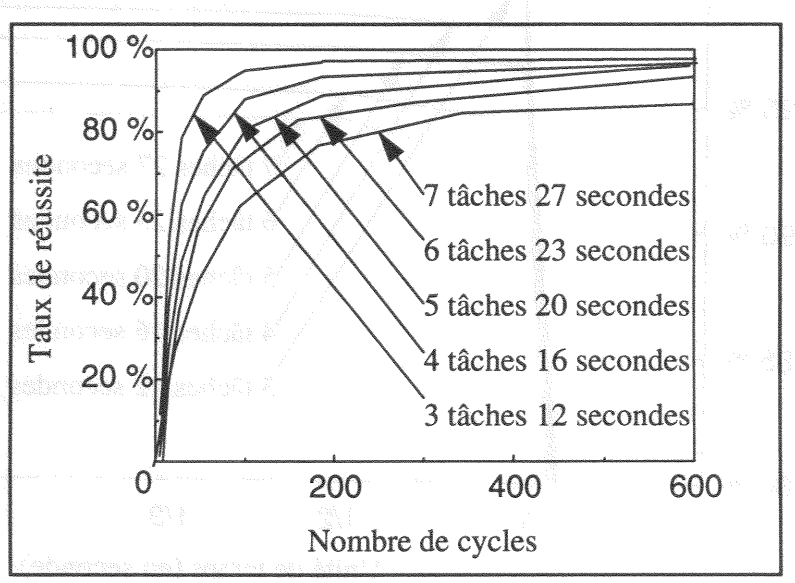

Figure 33 : Fonction utilité des algorithmes

Cette courbe est utile pour gérer l'ordonnancement de séquence de contrats dans le cas où plusieurs algorithmes sont en concurrence. En effet, elle permet d'estimer avant l'exécution de la séquence l'espérance d'obtenir une solution à notre problème en fonction du temps, et donc du nombre de cycles affectés à la séquence.

5.7.2. Granularité

Le second paramètre permettant la réalisation d'affinement est la granularité des données représentées dans le réseau (voir paragraphe 4.3.2, page 91).

Dans la représentation choisie pour notre réseau, le temps est discrétisé et chaque colonne représente une unité de temps. Ainsi, le réseau est constitué de $(n+m).t$ neurones avec n le nombres de tâches, m le nombre de machines et t le nombre d'unités

de temps. La durée d'une itération est donc directement proportionnelle à la façon dont l'horizon est découpé en unité de temps et la qualité de la réponse sera d'autant plus précise que les unités de temps sont petites.

D'autre part, en divisant l'unité de temps minimale (qui correspond au plus grand commun diviseur des durées à allouer), cela permet d'agrandir l'espace de recherche et ainsi de faciliter la convergence de la fonction d'énergie. La figure 34 montre l'augmentation du taux de réussite en fonction de l'unité de temps minimale.

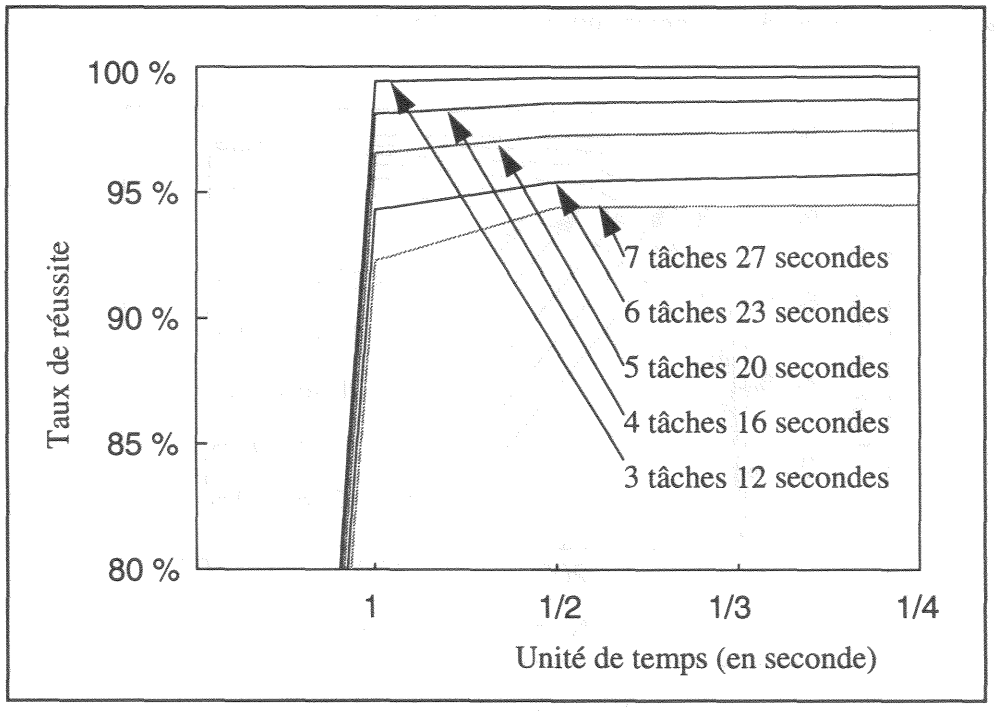

Figure 34 : Qualité en fonction du grain

5.7.3. Enchaînements

Les deux paragraphes précédents présentent deux façons différentes de réaliser des approximations. Le système peut être considéré comme un ensemble d'algorithmes déterminés chacun par deux paramètres : la vitesse de convergence du réseau et la longueur d'une unité de temps.

Le système peut ainsi être représenté par une matrice à deux dimensions, où chacune des cases correspond à une certaine valeur des deux variables d'approximation (voir figure 35). La résolution d'un problème revient alors à sélectionner une case dans cette matrice et à exécuter l'algorithme correspondant. En cas d'échec, il est nécessaire de choisir une nouvelle case et relancer l'exécution correspondante. On réitère l'opération

jusqu'à l'obtention d'un résultat correct ou jusqu'à ce que la date limite accordée pour la réponse soit atteinte.

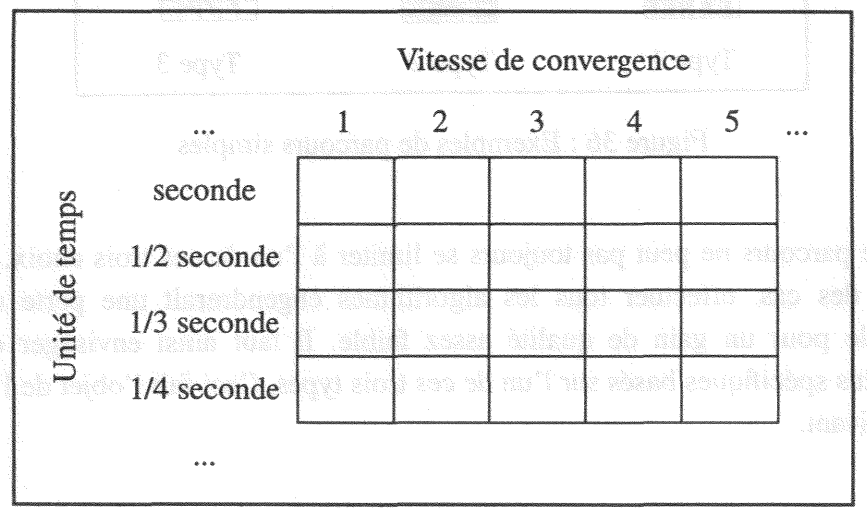

Figure 35 : Matrice des niveaux d'affinement

Notre problème est de maximiser la probabilité de fournir une réponse au problème pour une date limite fixée ou une loi de probabilité d'arrivée de cette échéance. Cela peut se résumer à la sélection du parcours de la matrice qui donne la plus forte espérance de fournir une solution correcte, tout en s'attachant à mettre régulièrement le résultat à jour afin de disposer d'un résultat intermédiaire de qualité acceptable quelle que soit la date à laquelle survient l'échéance.

Notre but étant de garantir une réponse dans le temps imparti, la solution la plus naturelle consiste à débuter par l'exécution du contrat le plus rapide afin de disposer dans les plus brefs délais d'un premier résultat quitte à ce qu'il soit de faible qualité. Ce résultat est ensuite affiné par le contrat ayant une durée légèrement supérieure afin de favoriser une amélioration rapide. Cette technique consiste donc à exécuter tous les contrats dans l'ordre croissant de leur durée. Ce parcours de la matrice se fait de trois façons différentes, selon l'influence des différents affinements sur le temps d'exécution. Il est possible de calculer exactement la complexité de chacun des contrats et ainsi d'en déduire leur durée. Il reste ensuite à estimer la qualité du résultat de chacun d'eux et d'en déduire quel affinement sera le plus utile à développer en premier. Si l'un des deux paramètres est nettement moins coûteux en temps ou nettement plus avantageux en qualité, on choisira un parcours de type 1 ou 2 (voir figure 36). Par contre, si les deux directions d'affinement ont des utilités semblablement équivalentes, le parcours sera plutôt du type 3.

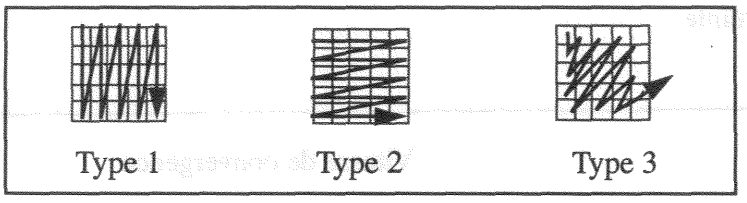

Figure 36 : Exemples de parcours simples

Cela dit, le parcours ne peut pas toujours se limiter à l'un de ces trois choix. En effet, dans bien des cas, effectuer tous les algorithmes engendrerait une perte de temps considérable pour un gain de qualité assez faible. Il faut ainsi envisager différents parcours plus spécifiques basés sur l'un de ces trois types. Ceci fait l'objet de l'étude du chapitre suivant.

6. Composition d'algorithmes

Grâce au modèle de Hopfield, le système dispose d'une matrice de contrats pour résoudre le problème posé. Chaque ordonnancement a plus ou moins de chances d'être résolu avant sa date limite selon la séquence de contrats choisie. La séquence idéale se termine avant la date limite et donne la plus forte probabilité de succès. Sa construction nécessite ainsi une parfaite connaissance des performances (c'est-à-dire du rapport qualité du résultat sur temps d'exécution) des différents contrats afin de maîtriser les durées d'exécutions tout en maximisant le taux de réussite. Les tests réalisés dans le premier paragraphe de cette partie ont justement pour but d'établir la fonction *utilité* des différents contrats de façon à pouvoir par la suite construire un algorithme de conduite efficace des exécutions pour le système final. Puis, afin de rechercher la politique d'activation qui va nous permettre de construire une politique optimale, il nous faut répondre à plusieurs questions :

- Par quel algorithme est-il préférable de commencer la séquence d'exécution ?
- Quelle variable d'approximation est-il préférable d'affiner en priorité ?
- Comment les variables d'approximation doivent-elle être augmentées ?
  - par une progression arithmétique ? de quel pas ?
  - par une progression géométrique ? de quelle raison ?
  - par une autre progression ?

Afin d'analyser le comportement de notre système selon les différentes politiques proposées, nous avons mis au point un générateur de tests aléatoires. Nous avons généré un corpus de tests comprenant des exemples d'ordonnancements faisables, c'est-à-dire pour lesquels il existe un ordre des tâches qui garantit le respect de toutes les dates limites. La taille de ce corpus a été déterminée de manière à ce que l'intervalle de confiance des taux de réussite reste raisonnable (inférieur à 0.01%) et les tirages ont été fait de telle sorte qu'ils couvrent le plus largement possible l'espace des ordonnancements possibles.

Pour les tests présentés, le corpus utilisé contient des problèmes d'ordonnancement de 10 tâches, d'une durée variant de 1 à 6 secondes, devant s'exécuter sur une machine dans intervalle de temps de 37 secondes (appelé horizon). Il s'agit d'un problème relativement complexe puisque seuls cinquante pour cent des problèmes générés possédaient effectivement une solution (les autres n'ont donc pas été retenus dans le corpus), de plus les solutions utilisent très fortement la ressource (elle est inactive pendant 1 à 3 secondes seulement pour la plupart des tests).

6.1. Performance des contrats

Dans un premier temps, nous allons nous intéresser aux performances des différents contrats, c'est-à-dire aux pourcentages d'ordonnancements correctement résolus compte tenu des temps de calcul associés. Il s'agit de regarder le comportement du réseau de neurones selon les valeurs affectées aux variables d'approximations.

La matrice qui contient les différents contrats d'exécution est définie par les valeurs prises par les variables d'affinement (voir paragraphe 4.3, page 89). Dans nos tests, la variable représentant l'unité de temps est limitée à quatre valeurs qui sont respectivement la seconde, la demi-seconde, le tiers de seconde et le quart de seconde. Cette limitation est essentiellement due aux temps de calculs importants pour des valeurs plus précises alors que l'amélioration des résultats est très faible. La variable représentant la vitesse de convergence (voir paragraphe 5.7.1, page 111) pourra quant à elle prendre 32 valeurs, qui sont incrémentées par pas de 4 de 0 à 128, la valeur 0 correspondant à la vitesse la plus forte et donc le niveau d'approximation le plus grossier. Notons que le nombre d'itération n'est pas une fonction linéaire de la vitesse de convergence, elle est plutôt de type exponentielle. La matrice possède par conséquent 128 contrats différents qui correspondent aux 128 couples d'affectations possibles des variables d'approximations.

La figure 37 présente des tests de durées correspondant à chacun des contrats sur le corpus d'exemples généré de 10 tâches à ordonnancer sur un horizon de 37 secondes. Ces durées sont exprimées en secondes. Les tests ont été effectués sur une station de travail SUN 10-40 fonctionnant à 60 Specs. Il s'agit de durées en temps réel et non en temps de calcul sur le micro-processeur. Elles ont bien entendu toutes été mesurées sur la même machine dans les mêmes conditions : la machine était totalement allouée à notre système. L'axe horizontal représente la valeur de la vitesse de convergence, l'axe de profondeur représente l'unité de temps (le plan horizontal correspond donc la matrice de contrats définie dans le paragraphe 5.7.3, page 114), la durée d'un contrat est représentée par la hauteur de la courbe au point correspondant aux valeurs des variables d'affinement du contrat.

Cette première série de mesures permet de constater que la progressivité selon l'unité de temps est beaucoup plus sensible que celle selon la vitesse de convergence. Autrement dit, une variation de la vitesse de convergence entraîne une augmentation du temps de calcul moins conséquente que celle provoquée par un affinement de la précision de l'unité de temps. Ainsi, à taux de réussite égal, il sera préférable d'affiner en premier lieu la vitesse de la convergence puis seulement le longueur de l'unité de temps minimale.

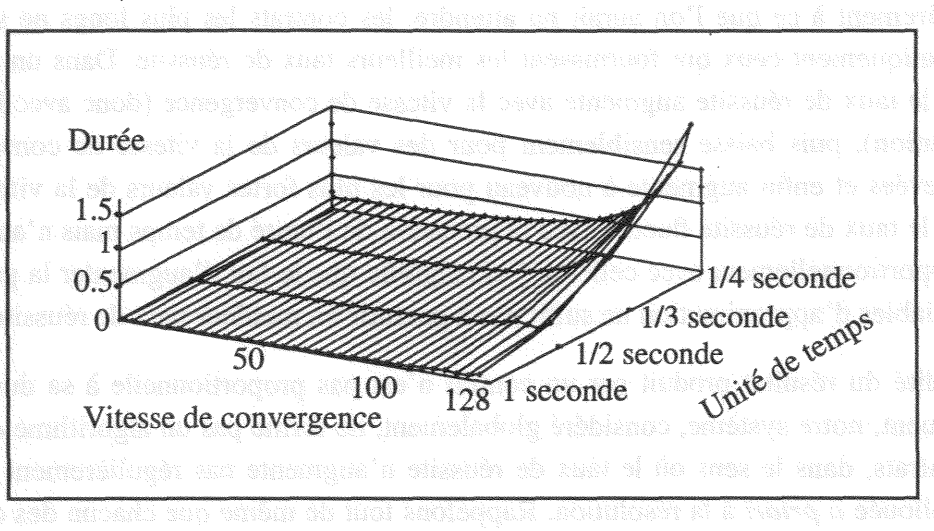

Figure 37 : Durées des contrats

Cependant, il faut bien évidemment comparer les résultats obtenus par les différents contrats avant de pouvoir conclure sur leur utilité relative. C'est l'objet de la série de mesure présentée dans le graphique suivant.

La figure 38 présente les taux de réussite de chacun des contrats, c'est-à-dire le pourcentage d'ordonnancements où chaque tâche se termine avant sa date limite. Le corpus d'exemple a été présenté en entrée du réseau et nous avons mesuré le pourcentage de problèmes correctement résolus pour chacune des différentes valeurs des variables d'approximation. Tout comme dans le graphique précédent, la matrice de contrats est représentée par le plan horizontal, le résultat d'un contrat correspond à la hauteur de la courbe au niveau du point lui correspondant dans la matrice.

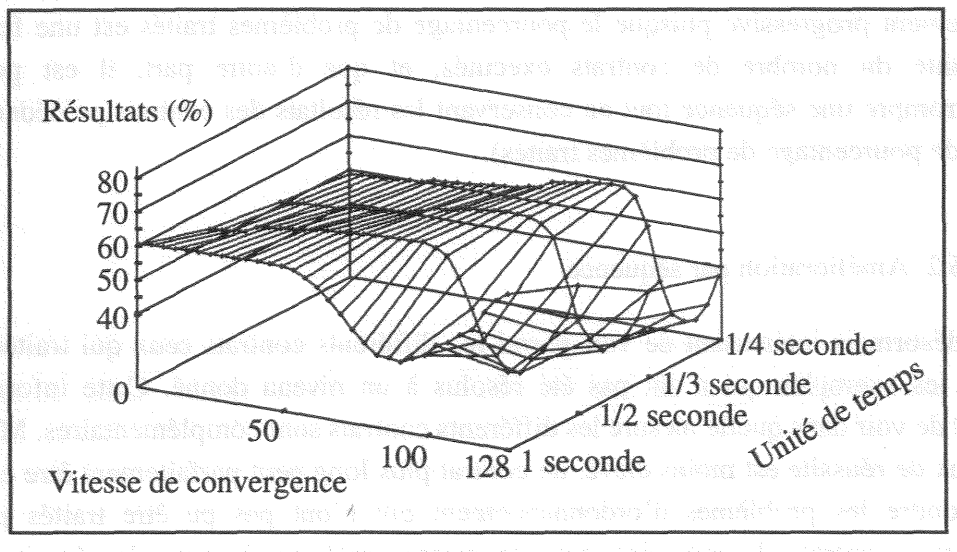

Figure 38 : Taux de réussite des contrats

Contrairement à ce que l'on aurait pu attendre, les contrats les plus longs ne sont pas systématiquement ceux qui fournissent les meilleurs taux de réussite. Dans un premier temps, le taux de réussite augmente avec la vitesse de convergence (donc avec la durée d'exécution), puis baisse sensiblement pour des valeurs de la vitesse de convergence plus élevées et enfin augmente à nouveau pour les plus fortes valeurs de la vitesse. De même, le taux de réussite fluctue avec la précision de l'unité de temps mais n'augmente pas proportionnellement avec celle-ci. Cela signifie que le fait d'augmenter la précision des variables d'approximation ne suffit pas à garantir un meilleur taux de réussite.

La qualité du résultat produit par un contrat n'est pas proportionnelle à sa durée. Par conséquent, notre système, considéré globalement, ne forme pas un algorithme *anytime* par contrats, dans le sens où le taux de réussite n'augmente pas régulièrement avec la durée allouée *a priori* à la résolution. Rappelons tout de même que chacun des contrats est *anytime* puisque, comme on l'a vérifié avec les tests de la figure 32, page 112, le nombre d'erreurs diminue régulièrement avec la durée d'exécution. La sélection *a priori* de la meilleure exécution ne peut donc pas se limiter à une recherche du plus long contrat se terminant avant l'échéance. Il faut en effet sélectionner le contrat respectant la durée maximale allouée à l'exécution tout en donnant la meilleure espérance possible de succès.

D'autre part, en cas d'échec d'un contrat, si l'échéance est suffisamment éloignée, il est parfois possible d'exécuter un ou plusieurs autres contrats afin d'améliorer la solution au problème. L'exécution d'une telle séquence de contrats permet au système d'être globalement *anytime*. En effet, considérons la performance d'une séquence de contrats en terme de pourcentage de problèmes traités : le premier contrat de la séquence résout un certain nombre de problèmes, il est ensuite possible de lancer l'exécution du deuxième contrat sur les problèmes non résolus afin d'améliorer le taux de réussite de la séquence et ainsi de suite pour les autres contrats. L'exécution d'une telle séquence sera globalement progressive puisque le pourcentage de problèmes traités est une fonction croissante du nombre de contrats exécutés, et que d'autre part, il est possible d'interrompre une séquence tout en conservant les résultats des contrats précédents (en terme de pourcentage de problèmes traités).

6.2. Amélioration par séquence

Il est désormais intéressant de voir parmi les différents contrats ceux qui traiteront le mieux les exemples qui n'ont pas été résolus à un niveau donné. Cette information permet de voir dans quelle mesure les différents contrats sont complémentaires. Même si son taux de réussite est moins élevé, un contrat plus long peut parfaitement être capable de résoudre les problèmes d'ordonnancement qui n'ont pas pu être traités par les précédents contrats. Il peut ainsi tout de même améliorer le taux de réussite de la séquence d'exécutions. D'après les résultats précédents, le contrat le plus rapide (ayant

la vitesse de convergence fixée à 0 et l'unité de temps égale à la seconde) semble donner un taux de réussite important. Nous avons donc décidé de tester les différentes séquences à partir de ce contrat. Les séries de tests présentées dans les paragraphes suivants ont ainsi pour objectif l'étude de l'utilité des différentes séquences de contrats débutant par le plus rapide de la matrice.

6.2.1. Séquence de deux exécutions

La courbe de la figure 39 concerne des tests exécutés sur la partie du corpus non traitée par le premier contrat $C_{0,0}$ (le plus rapide). Tous les ordonnancements sont dans un premier temps traités avec des valeurs des variables d'approximation les plus basses possibles, afin de tenter d'obtenir une réponse rapide. En cas d'échec, on tente de résoudre le problème par chacun des autres contrats $C_{i,j}$ (i et j désignent les indices de C dans la matrice) afin de voir quels sont ceux qui seront les plus performants en cas d'enchaînement de contrats débutant par $C_{0,0}$. Tous les taux de réussites sont ainsi au moins égaux à celui du premier contrat, la différence entre le taux de réussite d'un contrat $C_{i,j}$ et celui de $C_{0,0}$ indique le pourcentage de problèmes traités par $C_{i,j}$.

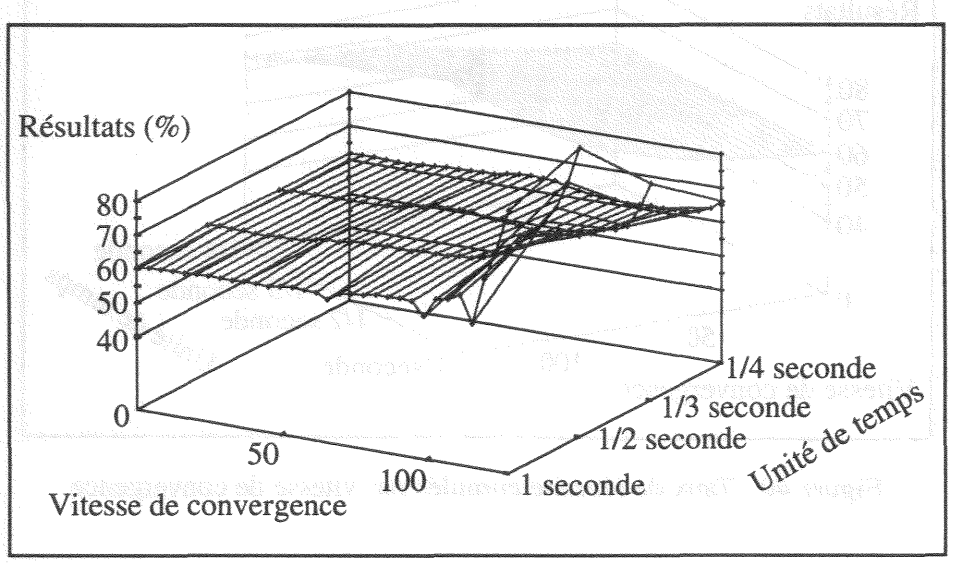

Figure 39 : Taux de réussite des séquences de deux contrats

Sur le corpus, tous les contrats $C_{i,j}$ améliorent le taux de réussite obtenu par l'exécution de $C_{0,0}$. Cela signifie qu'ils sont tous aptes à traiter au moins quelques problèmes irrésolus par le contrat $C_{0,0}$. Cependant, l'augmentation du taux de réussite reste dans l'ensemble assez faible pour les contrats ayant des valeurs des variables d'approximation peu élevées, c'est-à-dire des contrats proches de $C_{0,0}$ dans la matrice. Les résultats tendent à montrer que ce sont les contrats les plus éloignés de $C_{0,0}$ (donc

ceux ayant une vitesse de convergence plus forte) qui sont les plus adaptés pour les problèmes non résolus par le contrat $C_{0,0}$.

6.2.2. Exécution par affinement de la vitesse de convergence

D'après les tests précédents, il semble s'avérer utile d'effectuer une série de contrats. Afin de vérifier cette constatation, la seconde série de tests consiste en l'exécution séquentielle de tous les contrats pour une unité de temps donnée : pour chacune des valeurs de j, une séquence est définie comme étant la composée des $C_{i,j}$ pour i variant de 0 à 128. Ainsi, les séquences d'exécution sont constituées des lignes de la matrice de contrats. Pour chacune des valeurs j de l'unité de temps, le corpus de problèmes est traité par le contrat le plus rapide, $C_{0,j}$. Puis, si sa résolution échoue, la vitesse de convergence est affinée et le traitement est relancé, ainsi de suite... Les résultats de cette série de tests sont présentés dans la figure 40.

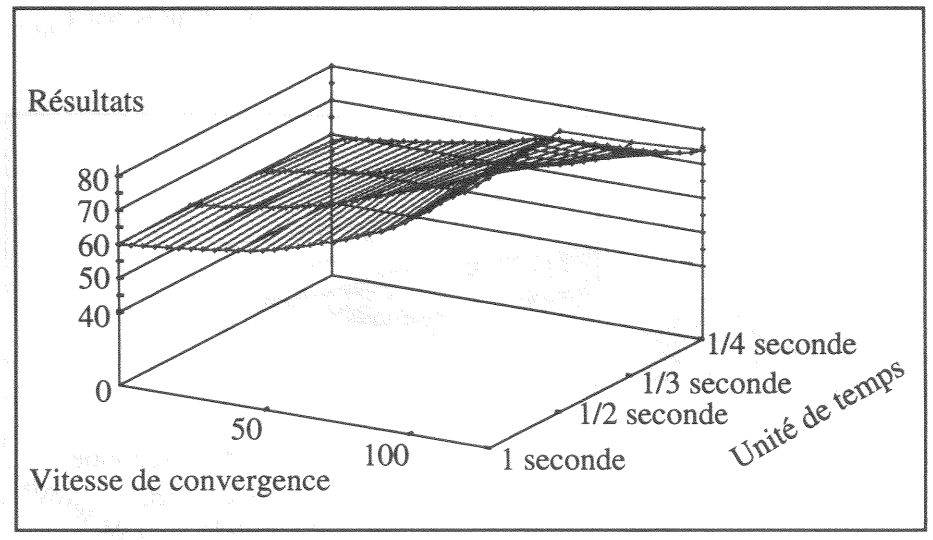

Figure 40 : Taux de réussite cumulés par vitesse de convergence

Dans toutes les séquences (il y a une séquence par valeur possible de l'unité de temps), chaque contrat réussit à traiter des problèmes non résolus par les précédentes exécutions. Ainsi le taux de réussite augmente constamment lors de la séquence d'exécutions. En augmentant la vitesse de convergence, les exemples refoulés par les premiers niveaux ont de plus en plus de chances d'être traités de façon correcte.

Le taux de réussite de la séquence d'exécution augmente après chaque exécution d'un contrat, c'est-à-dire que plus le temps alloué à la résolution est important, plus la probabilité est forte que le résultat soit correct. De plus, la séquence peut être stoppée à tout instant, le résultat retourné est alors celui de l'exécution du dernier contrat exécuté

entièrement. Nous sommes bien ici en présence d'un algorithme *anytime* interruptible dont la fonction utilité s'obtient en traçant la courbe ayant en ordonnée les résultats de la figure 40 et en abscisse les durées cumulées de la figure 37. Ainsi, en exécutant des séquences de contrats qui affinent successivement la valeur de la vitesse de convergence, nous obtenons une exécution qui possède l'intéressante propriété de progressivité.

Nous allons dans un deuxième temps nous pencher sur l'autre dimension d'approximation, à savoir la précision de l'unité de temps.

6.2.3. Exécution par affinement de l'unité de temps

La série de tests de la figure 41 effectue l'exécution en séquence de tous les contrats représentant des valeurs différentes de l'unité de temps pour une vitesse de convergence constante. Les séquences d'exécution sont donc constituées par les colonnes de la matrice de contrats : pour chacune des valeurs de i, une séquence est définie comme étant la composée des $C_{i,j}$ pour j variant de 0 à 3 (en terme d'indice de la matrice, ou alors variant de 1 seconde à 1/4 de seconde en terme de valeur de la variable d'affinement). Pour chacune des valeurs de la vitesse de convergence i, le corpus de problèmes est traité par le contrat le plus rapide, $C_{i,0}$. Puis, si sa résolution échoue, l'unité de temps est affinée et le traitement est relancé, ainsi de suite... L'enchaînement des contrats continue jusqu'à ce que le problème soit résolu ou jusqu'à ce que l'amélioration due à l'affinement de l'unité de temps soit négligeable (dans notre cas, à partir du quart de seconde).

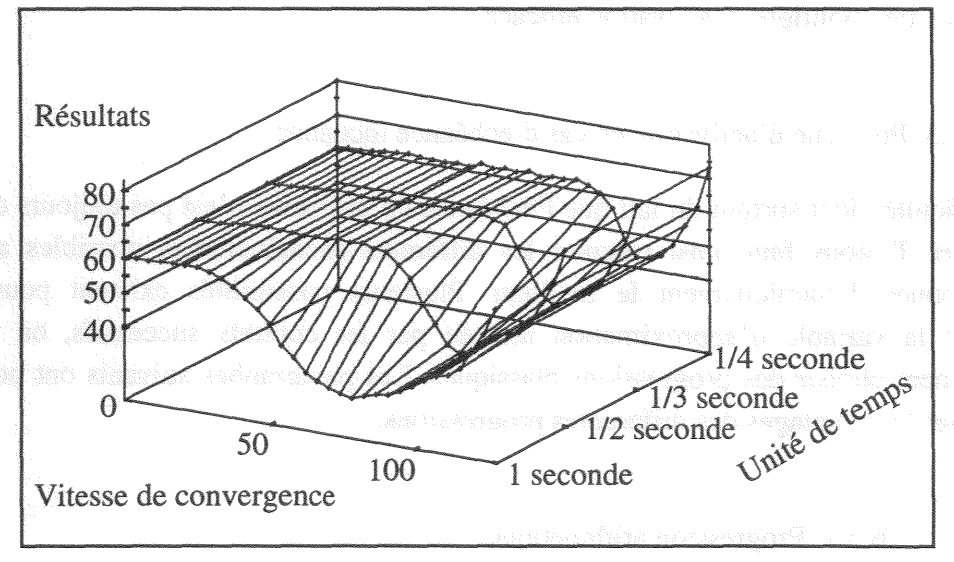

Figure 41 : Taux de réussite cumulés par unité de temps

Dans ces séquences d'améliorations, chaque contrat réussit à traiter des problèmes non résolus jusqu'alors, d'où la constante progression des résultats. Comme dans la série de tests précédente, les séquences d'exécution des contrats correspondent à des algorithmes *anytime*, avec une fonction utilité qui croît avec le temps : le taux de réussite de la séquence d'exécution augmente après chaque exécution d'un contrat, c'est-à-dire que plus le temps alloué à la résolution est important, plus la probabilité que le résultat soit correct est forte, et l'exécution peut être stoppée à tout instant, le résultat retourné est alors celui produit par le dernier contrat exécuté entièrement.

6.2.4. Conclusions

Les performances d'une séquence d'affinement selon l'unité de temps sont nettement inférieures à celles d'une séquence d'exécution cumulée selon la vitesse de convergence pour une unité de temps fixée. De plus les durées d'exécutions sont moins élevées dans ce deuxième cas. Par conséquent l'affinement selon la vitesse de convergence sera prioritaire sur celui selon l'unité de temps.

D'autre part, il n'est pas nécessairement optimal d'exécuter tous les contrats les uns à la suite des autres. En effet, deux contrats proches dans la matrice ont une forte probabilité de traiter des problèmes similaires. Il est peut être alors plus intéressant d'enchaîner la séquence sur l'exécution d'un contrat plus éloigné dans la matrice. Toutefois, en sélectionnant un contrat qui utilise des variables d'approximations très fines, il est probable d'obtenir une durée importante et ainsi ne pas fournir de réponse si l'échéance est proche. Il faut donc trouver un juste compromis et, pour cela, il est nécessaire d'avoir recours à une politique d'activation efficace.

6.3. Politique d'activation en cas d'échéance inconnue

La difficulté vient surtout du fait que l'échéance du problème n'est pas toujours connue *a priori*. Il nous faut ainsi calculer les différents enchaînements possibles afin de sélectionner dynamiquement le meilleur. Plusieurs possibilités existent pour faire évoluer la variable d'approximation utilisée par les contrats successifs, on pourra notamment choisir des progressions classiques. Les paragraphes suivants ont pour but d'étudier les avantages des différentes progressions.

6.3.1. Progression arithmétique

Si l'on considère une progression arithmétique de pas 1 pour la vitesse de convergence, cela revient à exécuter une ligne de contrat sur la matrice, c'est-à-dire, le test de la figure 40, page 122. Si l'on choisit une progression de pas 2, cela revient à exécuter une ligne de matrice en sélectionnant un contrat sur deux, et ainsi de suite. Afin de

déterminer quel est le meilleur pas possible, il faut étudier les performances des différentes suites. C'est l'objet de la série de tests de la figure 42. Pour ces tests, nous avons construit plusieurs systèmes qui utilisent différentes suites arithmétiques (de pas respectifs 1, 2, 4, 8, 16, 32 puis 64) et, à titre de comparaison, nous avons également mesuré les performances d'une suite géométrique de raison 2. Chaque test consiste à mesurer le pourcentage de problèmes d'ordonnancement résolus selon le nombre d'itérations effectuées.

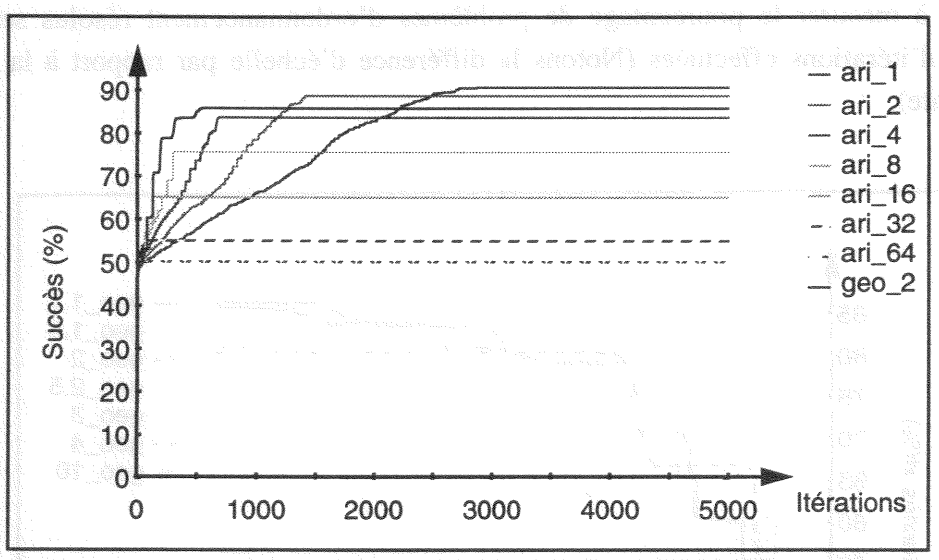

Figure 42 : Performances pour plusieurs progressions arithmétiques

Ces tests montrent que plus le pas de progression est rapide, plus le taux de réussite augmente rapidement. Cependant, pour des pas trop élevés, peu de contrats sont réalisés, et ainsi le taux de réussite final reste relativement plus faible. Par conséquent, le choix du pas optimal doit se faire selon la date limite accordée à la réponse. Un pas de 1 est plus efficace pour une échéance survenant après 2500 itérations, un pas de 2 pour une échéance entre 1250 et 2500, un pas de 4 pour une échéance entre 600 et 1250, ainsi de suite...

Cependant, si l'on compare ces résultats avec ceux d'une suite géométrique, on constate que pour des échéances inférieures à 1300, quelle que soit la date d'échéance, la suite géométrique est plus intéressante. Seuls les suites arithmétiques de raison 1 et 2 sont légèrement meilleures pour des échéances supérieures. Par conséquent, dès lors que l'arrivée de l'échéance ou que la durée d'exécution est mal maîtrisée (par exemple si la charge de la machine est inconnue), une suite géométrique sera préférable comme politique d'activation.

Il nous reste maintenant à déterminer quelle sera la meilleure raison d'une progression géométrique. C'est l'objet du chapitre suivant.

6.3.2. Progression géométrique

Une progression géométrique de raison *n* consiste à sélectionner une séquence de contrats telle que le $i^{ème}$ ait une durée égale à n fois celle du $i\text{-}1^{ème}$. Comme pour la progression arithmétique, il faut déterminer la meilleure raison possible. Pour cela, il faut étudier les performances des différentes suites. C'est l'objet de la série de tests de la figure 43. Pour ces tests, nous avons construit plusieurs systèmes qui utilisent différentes suites géométriques (de raisons respectives 1.1; 1.5; 2; 2.5; 3; 4 puis 10). Chaque test consiste à mesurer le pourcentage de problèmes d'ordonnancement résolus selon le nombre d'itérations effectuées (Notons la différence d'échelle par rapport à la figure précédente).

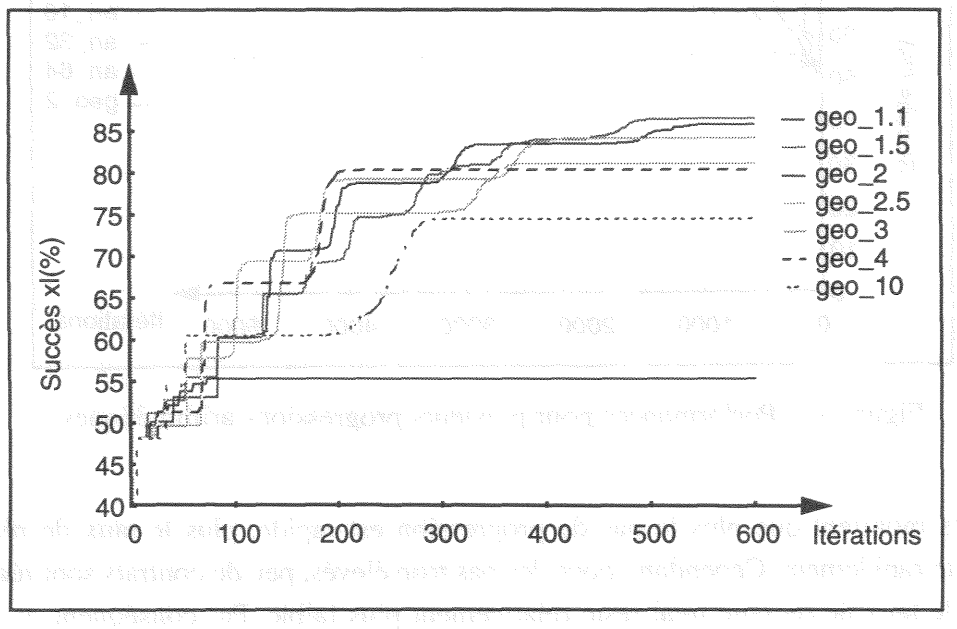

Figure 43 : Performances pour plusieurs progressions géométriques

Au vue de ce graphique, il est difficile de conclure sur le fait qu'une raison est meilleure qu'une autre. Il faut évidemment éviter une raison trop faible, telle que 1.1 qui revient presque à utiliser une suite arithmétique. Une raison trop grande donne également des résultats finaux faibles liés au petit nombre de contrats réalisés. Par contre pour les autres raisons (1.5; 2; 2.5; 3 et 4), le choix optimal ne peut se faire que si l'on connaît parfaitement l'échéance et plus exactement le nombre d'itérations réalisables avant que celle-ci ne survienne. En cas d'échéance inconnue, il faut trouver un juste compromis. Le paragraphe suivant se propose d'étudier le cas d'une progression de raison 2.

6.3.3. Progression géométrique de raison 2

Les tests précédents semblent montrer qu'une suite géométrique de raison 2, semble être une politique d'activation performante lorsque le temps accordé au calcul du résultat est inconnu *a priori*. Cette séquence d'activation permet ainsi de trouver un bon compromis entre choisir un contrat suffisamment plus éloigné dans la matrice pour fournir des résultats corrects et être suffisamment proche pour fournir un résultat rapide. La solution consiste donc à exécuter une suite de contrats dont la durée est doublée à chaque fois [Zilberstein 96]. Si le premier contrat possède une durée de t secondes, la séquence se poursuit par un contrat d'une durée de $2t$, le $i^{ème}$ contrat exécuté aura une durée de $2^i.t$ secondes. Une façon d'estimer si une telle politique est efficace est de calculer le pourcentage de temps utile, c'est à dire le rapport du temps consacré à des contrats qui ont fourni un résultat exploité sur le temps d'exécution total.

Soit R le ratio indiquant le temps utile. Le calcul de R est différent selon que le système fournit un résultat ou non. Il est également possible de donner une estimation moyenne de R étant donné les probabilités de succès $P(i)$ des différents contrats. Il nous faut ainsi distinguer plusieurs cas :

- Si un ordonnancement correct est produit après le premier contrat, la durée d'exécution est entièrement utile. D'où un R donné par la formule :

$$R = t/t = 1$$

- D'une manière générale, si le $n^{ième}$ contrat produit un résultat valide, alors le temps consacré aux $(n-1)$ premiers contrats était inutile. R s'obtient alors par la formule :

$$R = \frac{2^n \cdot t}{\sum_{i=0}^{n} 2^i \cdot t} = \frac{2^n}{(2^{n+1} - 1)} = \frac{1}{2 - \frac{1}{2^n}}$$

R tend ainsi vers 50% quand n tend vers l'infini, c'est à dire quand la date limite est éloignée par rapport à la durée des contrats. Quand un résultat satisfaisant est produit, on peut donc en conclure que plus de la moitié du temps consacré à l'exécution l'a été à bon escient (il s'agit en fait de la durée du dernier contrat).

- Dans le pire des cas, la date limite survient juste après la fin du $n^{ième}$ contrat. Dans ce cas, aucun résultat n'est produit et une solution approximative est extraite du résultat du dernier contrat entièrement exécuté. Ainsi, le temps perdu à cause de cette interruption est égal à la somme des durées d'exécution des contrats de 1 à $(n-2)$ et de la durée d'exécution du contrat interrompu (soit au

maximum un peu moins de $2^n.t$ secondes). R est donc calculé de la manière suivante :

$$R = \frac{2^{n-1} \cdot t}{\sum_{i=0}^{n} 2^i \cdot t} = \frac{2^{n-1}}{(2^{n+1} - 1)} = \frac{1}{4 - \frac{1}{2^{n-1}}}$$

R tend vers 25% quand n tend vers l'infini c'est à dire quand la date limite est éloignée par rapport à la durée des contrats. Quand aucun résultat satisfaisant n'est produit, on peut donc en conclure que plus du quart du temps consacré à l'exécution l'a été à bon escient (il s'agit en fait de la durée du dernier contrat exécuté entièrement).

- Dans l'on désire estimer R dans le cas moyen, il nous faut la probabilité de succès des différents contrats $P(i)$. Puis on calcule la formule :

$$R = \left(\sum_{i=0}^{n} \frac{P(i)}{2 - \frac{1}{2^i}} \right) + \frac{1 - \sum_{i=0}^{n} P(i)}{4 - \frac{1}{2^{n-1}}}$$

Ce taux moyen est calculé à la fin de ce paragraphe pour notre exemple d'ordonnancement.

Cette politique présente donc l'intérêt de garantir que dans le pire des cas (pas de solution correcte produite) au minimum le quart du temps a été utile.

Dans la série de tests suivante, cette politique a été appliquée pour la sélection de la suite des valeurs prises pour la vitesse de convergence. Pour chacune des valeurs de l'unité de temps, les contrats exécutés successivement ont une durée double du précédent. La figure 44 présente les résultats de ces tests. Ils confirment l'amélioration croissante des taux de réussite avec la durée des contrats.

Ce test a été effectué pour chacune des valeurs de l'unité de temps. Notons que ce test est équivalent à celui qui a produit les fonctions *utilité* de la figure 33, page 113. La différence de profil entre les fonctions vient de l'échelle de la vitesse de convergence qui est logarithmique ici, alors que celle de la figure 33 est linéaire.

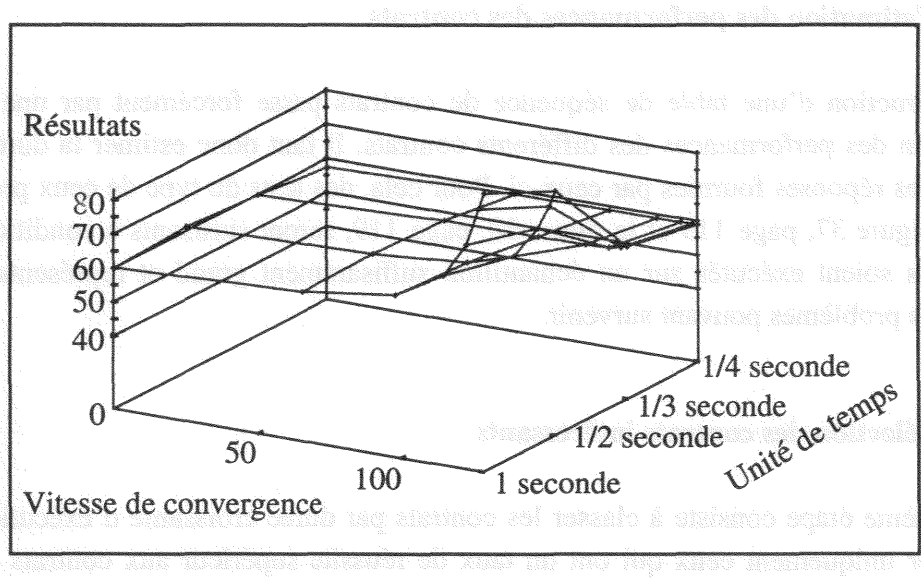

Figure 44 : Suite d'exécutions de durée double

Si l'on considère le taux de réussite pour une unité de temps égale à 1 seconde, on obtient les probabilités :

P(0) = 0.581; P(1) = 0.007; P(2) = 0.020; P(3) = 0.059;
P(4) = 0.078; P(5) = 0.085; P(6) = 0.031; P(7) = 0.021;
et donc **T = 0.7678**

Cela signifie que sur notre exemple, en moyenne seulement **23,2 %** du temps d'exécution a été gaspillé.

6.4. Politique d'activation en cas d'échéance connue

Dans le cas où l'échéance est connue, que ce soit parfaitement ou au travers d'une distribution d'une loi de probabilité, le problème est différent. L'objectif est désormais de maximiser l'utilité de la séquence de contrats compte tenu de la probabilité d'arrivée de l'échéance. Ainsi on cherche à construire une fonction de contrôle optimal qui à chaque durée d'exécution associe la séquence d'exécutions qui fournit le taux de réussite le plus élevé. Une fois cette table construite, on sélectionne la meilleure séquence selon la probabilité d'arrivée de l'échéance.

Cette construction est longue compte tenu du nombre imposant de tests à mettre en œuvre, mais ne pose pas véritablement de gros problèmes de programmation. Elle se fait selon plusieurs étapes :

• **Estimation des performances des contrats**

La construction d'une table de séquence de contrats passe forcément par une bonne estimation des performances des différents contrats. Il faut donc estimer la durée et la qualité des réponses fournies par ceux-ci. Pour cela, des tests du type de ceux présentés dans la figure 37, page 119 et la figure 38, page 119, seront suffisants à condition bien sûr qu'ils soient exécutés sur un échantillon suffisamment grand et représentatif des différents problèmes pouvant survenir.

• **Sélection des contrats intéressants**

La deuxième étape consiste à classer les contrats par durée croissante d'exécution et à conserver uniquement ceux qui ont un taux de réussite supérieur aux contrats qui les précédent. Etant donnés deux contrats C_1 et C_2, si C_1 nécessite une durée d'exécution plus grande que C_2 et que par ailleurs C_2 possède un taux de réussite plus élevé que C_1, la séquence de contrats composée uniquement de C_1 ne présente aucun intérêt. Ainsi, pour la recherche des séquences d'exécutions optimales, on ne conserve que le contrat C_2 auquel on associe sa durée et son taux de réussite. Dans notre exemple, si l'on applique cette technique sur le corpus généré de problèmes de 10 tâches à ordonnancer sur 37 secondes, seulement 25 contrats sont retenus sur les 128 initiaux.

Ces contrats vont ainsi servir de séquences initiales, on placera à la suite les séquences d'exécutions susceptibles d'être intéressantes.

• **Estimation des séquences intéressantes**

Pour chacun des contrats retenus, on désire tester l'enchaînement sur un contrat de durée plus importante. La durée d'exécution sera obtenue facilement par l'addition des durées individuelles, par contre le taux de réussite nécessite la réalisation de tests supplémentaires. Il est cependant possible de borner ce taux. En effet, les résultats des deux contrats étant connus, il est possible de calculer le taux maximum (cas où les ensembles des problèmes traités sont disjoints) et le taux minimum (cas où il y a recouvrement entre les ensembles des problèmes traités) de la séquence.

Si la borne supérieure du taux de réussite est plus petite qu'une séquence de durée moins importante, la séquence est automatiquement rejetée. Dans le cas contraire, la séquence est testée et conservée uniquement si elle présente un rapport du taux de réussite sur le temps d'exécution intéressant.

• **Passage aux séquences suivantes**

L'étape précédente fournit ainsi l'ensemble des séquences de deux contrats. Il s'agit maintenant de passer aux séquences constituées de trois contrats. Pour cela, on reprend les séquences optimales déterminées à l'étape précédente et on tente d'y ajouter un troisième contrat, d'une durée supérieure aux contrats précédents. Dans ce cas également, la durée peut être calculée auparavant et il est possible de borner *a priori* le taux de réussite afin de déterminer s'il est utile d'effectuer les tests correspondants. Cette étape est donc identique à la précédente, la suite du processus est en fait l'itération sur cette étape afin de constituer les séquences optimales pour un nombre de plus en plus important de contrats.

Conclusion et Perspectives

Conclusion

Dans un cadre temps réel, un système informatique œuvre dans un environnement dynamique et doit prendre en compte la notion de temps de réponse. Ses actions sont contraintes par le temps et ses performances sont jugées par le rapport qualité sur temps. Un tel agent doit réagir vite tout en se concentrant parfois pendant un longue période sur une action urgente. Ceci peut s'avérer contradictoire. Aussi, il faut trouver un juste compromis entre la fréquence des mises à jour et le temps de traitement des nouvelles données. En effet, le traitement doit être suffisamment long pour produire une réponse de bonne qualité, mais doit également être suffisamment court pour que les données qui ont déclenché ce traitement restent valides. D'où la nécessité de dégager des qualités essentielles telles que la rapidité, l'adaptabilité, la ponctualité et une capacité de focalisation. Pour faire face à ces nombreuses exigences, de nombreuses techniques temps réel ont été développés, mais ces techniques sont difficilement adaptable en intelligence artifielle. En effet, les algorithmes utilisés sont essentiellement non déterministes et par conséquent leur performances imprévisibles, l'estimation en pire cas étant rarement satisfaisante. L'agent doit ainsi utiliser le calcul flexible, c'est-à-dire des techniques d'approximation afin de pouvoir garantir à la fois la qualité et la ponctualité de la réponse. Le développement d'agents ayant de telles possibilités au sein d'un système multi-agents permet d'étendre considérablement leurs fonctionnalités.

Cette thèse s'est focalisée plus spécialement sur le respect des échéances à travers la mise en place d'un système de résolution en temps contraint. Le problème est de savoir quel comportement un agent doit adopter afin de pouvoir fournir une réponse satisfaisante tout en étant limité dans son temps d'exécution. Dans ce contexte, résoudre un problème de façon optimale n'est plus alors synonyme de trouver la réponse ayant la meilleure qualité mais plutôt de trouver la meilleure réponse possible en fonction du temps nécessaire à son obtention. Pour réaliser ce compromis, le système manipule des programmes pouvant s'exécuter selon différents temps, la qualité du résultat varie en fonction de celui-ci.

Nous avons étudié les différentes façons de mettre en place des algorithmes progressifs afin de réaliser des compromis entre le coût lié à l'exécution et la qualité du résultat obtenu. La progressivité pouvant être régulière dans le cas d'algorithmes *anytime,* par paliers dans le cas de méthodes multiples ou définie *a priori* dans le cas d'algorithmes par contrats. La mise en place d'un raisonnement efficace dans un environnement temps réel passe ainsi par cette première étape de modification des traitements afin de les rendre progressifs. Cette transformation nécessite parfois uniquement la mise en forme de résultats intermédiaires, dans d'autres cas, elle peut nécessiter une analyse plus poussée du problème.

Dans cette thèse nous proposons l'utilisation de réseaux neuromimétiques permettant la réalisation de tels algorithmes. Ces modèles ont l'avantage de pouvoir se paralléliser

facilement pour obtenir un calcul efficace. Le modèle de réseau que nous utilisons est un modèle récurrent qui permet l'optimisation de problèmes modélisés par des contraintes. Nous proposons une technique de modification du fonctionnement interne des neurones pour paramètrer la rapidité de convergence du réseau, la qualité du résultat étant fonction de la durée d'exécution. Cette technique permet ainsi la définition aisée d'un ensemble d'algorithmes, chacun étant un contrat différent, c'est-à dire un compromis entre la durée d'exécution et la qualité du résultat. De plus chacun de ces contrats est lui-même progressif, puisqu'il peut être interrompu à tout instant, la qualité de la réponse fournie augmentant progressivement avec le temps d'exécution.

Une fois les algorithmes progressifs par contrats obtenus, une bonne connaissance de leurs performances est nécessaire afin de pouvoir évaluer leurs durées et la qualité des réponses produites. Pour le modèle que nous proposons, le calcul de la complexité d'une itération se fait aisément, la difficulté réside en l'estimation du nombre d'itérations. Cependant, il est possible de borner ce nombre et ainsi de fournir une durée d'exécution en pire cas, ce qui permet de garantir une réponse pour une durée d'exécution donnée. L'estimation de la qualité des résultats est plus délicate, puisqu'elle nécessite la réalisation de statistiques sur un grand nombre d'exemples générés aléatoirement.

Enfin, dans un troisième temps, il faut définir un module de contrôle intégrant à la fois les notions de coût et de date limite, liées à l'événement et les notions de performance des différents algorithmes pour réaliser l'exécution produisant un résultat ayant l'utilité la plus forte possible. Pour cela, nous avons étudiés plusieurs politiques d'enchaînement des contrats. Dans le cas où les connaissances externes (échéance de la réponse et charge de la machine) sont maîtrisées, il est possible d'utiliser une précompilation des différentes séquences afin de sélectionner la meilleure. Par contre, dans le cas où les connaissances liées au domaine sont imprécises voire inconnues, nous proposons l'utilisation d'une politique suivant une suite géométrique de raison 2. Cette politique permet l'obtention de résultats fréquents qui ont été obtenus, dans le pire des cas, par l'utilisation du quart du temps d'exécution.

Nous avons ensuite illustré le fonctionnement de notre système sur une application d'allocation à une ressource de tâches non préemptives, possédant une date de disponibilité et une échéance. Ce problème NP-complet doit très souvent être réalisé en temps réel. Nous avons montré comment ces différentes étapes étaient mises en œuvre pour traiter un problème d'optimisation. Ainsi dans un premier temps l'objectif était de trouver une méthode approximative qui traite correctement ce problème non polynomial. Puis, des variables d'affinement ont été dégagées afin de définir des contrats d'exécutions, chacun ayant sa propre durée et sa propre espérance d'obtention d'un résultat correct. Enfin plusieurs séries de tests ont permis l'évaluation des performances des différents contrats. Puis nous avons proposé une politique de supervision des activations qui permet la réalisation de la séquence de contrats obtenant l'algorithme *anytime* le plus efficace possible.

Cette technique permet ainsi la mise en place d'un système progressif qui sera facile à étendre étant donné sa modélisation sous forme de contraintes. Par exemple, sur notre problème d'ordonnancement, nous avons montré de quelle façon ajouter des contraintes pour résoudre des problèmes d'allocation de plusieurs ressources, ou même pour résoudre un problème d'ordonnancement d'atelier (*Open Shop Scheduling*).

Notre système résout ainsi des problèmes d'optimisation de complexité non polynomiale par des techniques approchées dont la durée d'exécution peut être paramètrée. Cette caractéristique fondamentale va permettre son utilisation dans un cadre temps réel où la durée d'exécution est contrainte par le temps et l'utilité d'une réponse est jugée par le rapport qualité sur temps. De plus, par la définition d'une politique d'activation, notre système définit d'une bibliothèque d'algorithmes *anytime*, qui seront utilisables au même titre que d'autres algorithmes itératifs interruptibles ou encore d'autres paquets de règles progressifs.

Perspectives

Nous réfléchissons actuellement à la mise en œuvre d'une politique d'activation adaptée au cas où la date limite pour fournir une réponse est partiellement connue. Il se peut en effet que cette date soit inconnue *a priori*, mais que la possibilité que l'échéance survienne à un instant donné puisse être modélisée par une loi de probabilité. Dans ce cas, il est possible de choisir une politique d'activation qui s'adapte à cette loi de manière à garantir des résultats de bonne qualité aux instants où la probabilité d'occurrence de l'échéance est la plus forte.

Pour la construction de la politique d'activation, il nous a fallu évaluer notre système, c'est-à-dire estimer les performances des contrats, nous avons ainsi mesuré les taux de réussite moyens de plusieurs séquences d'activation et sélectionner celle qui donnait les meilleurs résultats en moyenne. Cette mesure repose sur l'hypothèse que les énoncés sont indépendants et qu'ils sont ainsi traités par un contrat donné avec un succès très proche. La mesure de la variance semble indiquer que cette hypothèse est valide. Cependant, il semblerait intéressant de chercher à définir une classification des différents énoncés afin de vérifier si certains contrats ne sont pas plus performants selon la catégorie de problèmes traités. Cette classification pourrait être faite en amont du système et dicter ainsi la politique d'activation à adopter.

Notre système est en réalité une séquence progressive d'algorithmes progressifs. Cette propriété est utile en cas d'arrêt brutal d'un contrat, mais n'est pas utilisée lors de la supervision des enchaînements. Il est peut-être possible d'utiliser cette propriété par une évaluation de la qualité du résultat en cours de contrat. La politique optimale serait alors parfois d'interrompre le contrat en cours pour en commencer un nouveau. Cependant, il ne faut pas que l'évaluation de la qualité ne soit trop coûteuse, la principale qualité d'une supervision temps réel étant la rapidité.

D'autre part, comme notre système permet la réalisation de tâches simples de façon progressives, il peut être utilisé dans un système de prise de décision temps réel plus complexe. Notre système réaliserait alors des fonctions de bas niveau mises en concurrence ou utilisées par un autre système. Par exemple, il est possible d'imaginer son utilisation pour le modèle des sources de connaissances pour une boite à outils temps réel telle que Reakt. A un événement donné, Reakt associe un plan d'actions sous la forme d'un arbre. Afin de mettre en œuvre un raisonnement progressif, chacune de ces actions doit être définie de façon progressive. Il faut pour cela disposer d'une bibliothèque de fonctions progressives dont les durées sont connues afin d'être gérées par un ordonnanceur qui garantit ainsi la meilleure approximation globale en ajustant correctement le degré d'affinement des actions progressives. La difficulté de réalisation d'une telle application est notamment la définition de la bibliothèque actions progressives, c'est à cette demande que répond notre système.

Glossaire

Agent : Système en contact avec un environnement extérieur. Son rôle est de transformer des perceptions du monde en des actions effectives permettant de modifier l'environnement en vue d'atteindre un état voulu.

Agent réactif (*reactive agent*) : Agent effectuant des actions de bas niveau selon une stratégie prédéfinie. Son fonctionnement est basé sur le paradigme stimulus / action.

Agent cognitif (*deliberative agent*) : Agent exécutant des procédures de décisions autres que de la simple recherche d'information.

Agent planificateur (*planning agent*) : Agent cognitif utilisant le résultat de ses actions passées pour en déduire des contraintes sur les futurs états possibles et ainsi sélectionner l'action à venir la plus adéquate.

Agent dirigé par les buts (*utility driven agent*) : Agent dont le comportement est défini de manière à maximiser une fonction d'utilité explicite donnée.

Algorithme anytime (*anytime algorithm*) : Algorithme itératif garantissant de produire une réponse quel que soit l'instant de son interruption. La qualité du résultat s'améliore en fonction du temps consacré à l'exécution.

Algorithme par contrats (*contract anytime algorithm*) : Algorithme pouvant être exécuté selon une échelle de temps continue en fonction d'un paramètre fixé avant le début de l'exécution.

Fonction performance (*performance profile*) : Evolution de la qualité de la réponse d'un algorithme au cours du temps.

Optimalité (*optimality*) : Comportement d'agent visant à maximiser la qualité du résultat selon le critère donné en objectif.

Optimalité contrainte (*bounded optimality*) : Comportement d'agent visant à maximiser la qualité du résultat selon le critère donné en objectif et selon le coût lié à la durée d'obtention de ce résultat.

Réactivité (*reativity*) : capacité d'un système à rester en contact permanent avec le monde extérieur pour répertorier les changements importants et les traiter convenablement. C'est le temps de prise en compte d'un événement externe par le système.

Système temps réel (*Real-Time system*) : C'est un système capable de garantir une réponse après qu'un certain temps lié au domaine s'est écoulé.

Temps de réponse (*response time*) : Temps mis par le système pour reconnaître et répondre à un événement externe.

Utilité (*utility*) : Fonction associant un nombre réel à un état de l'environnement. Ce dernier correspond à une évaluation dans une unité arbitraire du degré d'achèvement du but à atteindre.

Utilité explicite (*explicit utility*) : Utilité mesurée en considérant l'état courant comme état final.

Utilité implicite (*implicit utility*) : Utilité mesurée en considérant l'état courant comme état intermédiaire d'une résolution.

Bibliographie

[Agre 87]
P. Agre et D. Chapman. *Pengi : An implementation of a theory of activity.* Actes de AAAI-87, pp 268-272.

[Alabau 92]
M. Alabau et T. Dechaize. Ordonnancement temps réel par échéance. Technique et science informatiques, vol 11, n.3, pp 59-123, 1992.

[Alexandre 95]
F. Alexandre, C. Cardeira, F. Charpillet, Z. Mammeri et M.-C. Portmann *Compu-Search Methodologies II: Scheduling Using Genetic Algorithms and Artificial Neural Networks*, Production and Scheduling of Manufactoring System, Ed. Artiba A. and Elmaghraby S.E, 1995.

[Bellman 57]
R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bessière 95]
C. Bessière, E. Freuder et J.C. Regin. Using inference to reduce arc consistency computation. Actes de IJCAI-95. pp 592-598.

[Blazewicz 93]
J. Blazewicz, K. Ecker, G. Schmidt, J. Weglarz. *Scheduling in Computer and Manufactoring Systems*, Springer-Verlag, 1993.

[Boddy 89]
M. Boddy et T. Dean. *Solving time-dependant planning problems*. Actes de IJCAI-89. pp 979-984.

[Boddy 91a]
M. Boddy. *Solving time-dependent problems: A decision theoric approach to planning in dynamic environnements*.Technical report CS-91-06, Brown University.

[Boddy 91b]
M. Boddy. *Anytime problem solving using dynamic programming.* Actes de AAAI-91. pp 738-743.

[Brooks 91]
R. A. Brooks. *Intelligent whitout Representation*. Artificial Intelligence, vol 47. pp 139-160, 1991.

[Browaeys 84]
F. Browayes et al. *SPECTRE : Proposition de noyau normalisé pour les exécutifs temps réel.* Technique et Science Informatiques, vol 3, n.1, 1984.

[Cardeira 94a]
C. Cardeira. *Ordonnancement temps réel par réseaux de neurones.* Thèse de Doctorat à l'Institut National Polytechnique de Lorraine, 1994.

[Cardeira 94b]
C. Cardeira, Z. Mammeri. *Neural Networks for Multiprocessor Task Scheduling.* Proc. of 8th Euromicro Workshop on Real-Time Systems, IEEE CS, June, Vaesteraas, Sweden, 1994.

[Caux 95]
C. Caux, H. Pierreval et M-C. Portmann. *Les algorithmes génétiques et leur application aux problèmes d'ordonnancement.* Automatique Productique Informatique Industrielle, vol 29, n.4, pp 409-443, 1995.

[Charpillet 92]
F. Charpillet, J-M. Gallone et A-I. Mouaddib. *A test case validating progressive reasoning.* Délivrable projet Esprit Reakt CRIN-D5.2.3, Centre de Recherche en Informatique de Nancy, Vandœuvre-lès-Nancy, 1992.

[Charpillet 94a]
F. Charpillet et A. Boyer. *Incorporating AI techniques into predictable real-time systems : Reakt outcome.* Quatorzièmes Journées Internationales d'Avignon, pp 121-135.

[Charpillet 94b]
F. Charpillet et P. Théret. *Dossier : IA et temps réel.* Bulletin de l'Association Française pour l'Intelligence Artificielle, vol 17, pp 19-42, avril 1994.

[Chu 92]
C. Chu, M-C. Portmann. *Some New Efficient Methods to Solve the* $n/1/r_i/\sum T_i$ *Scheduling Problem*, Europeen Journal of Operational Research, vol 58, pp 404-413, 1992.

[Cohen 83]
M.A. Cohen et S.Grossberg. *Absolute stability of global pattern information and parallel memory storage by competitive neural networks.* IEEE Transaction on System Man and Cybernetics, vol 13, pp 815-826, Sept / Oct 1983.

[Dean 88]
T. Dean et M.Boddy. *An analysis of time dependent planning.* Actes de AAAI-88. pp 49-54.

[Dean 91]
T. Dean. *Planning and control.* Editions Morgan Kaufmann, 1991.

[Dodhiawala 89]
R. Dodhiawala, N.S. Sridharan et C. Pickering. Real-Time AI System : Definition and Architecture. IJCAI'89, pp 256-261.

[Erschler 85]
J. Erschler, G. Fontan et C. Merce. *Consistency of the Dissaggregation Process in Herarchical Planning*. Operations Research, vol 34, pp 464-469, 1985.

[Forgy 82]
C. Forgy. *Rete : A fast match algorithm for many pattern : many object pattern match problem*. Artificial Intelligence, vol 19, pp 17-37, 1982.

[G2 90]
An Introducion to G2. Gensym Corporation, 1990.

[Gallone 92]
J-M. Gallone. *Un algorithme d'ordonnancement approximatif*. Rapport technique du Centre de Recherche en Informatique de Nancy. TR-92-276.

[Gallone 94]
J.M. Gallone, F. Charpillet et V. Chevrier. *Un modèle d'agent pour un raisonnement contraint par le temps*. Journée systèmes multi-agents du PRC-IA, Paris 1994.

[Gallone 95]
J.M. Gallone, F. Charpillet et F. Alexandre. *Task Scheduling Using Neural Network*. IEEE International Conference on Emerging Technologies and Factory Automation, pp 509-520, 1995.

[Gallone 96a]
J-M. Gallone et F. Charpillet. *Hopfield Neural Network for Scheduling Non Preemptive Tasks*. Proc of 12th European Conference on Artificial Intelligence, pp 223-227, 1996.

[Gallone 96b]
J.M. Gallone et F. Charpillet. *Composing Approximated Algorithms Based on Hopfield Neural Network for Building a Resource-Bounded Scheduler*. IEEE International Conference on Tools with Artificial Intelligence, pp 445-446, 1996.

[Garey 79]
M.R. Garey et D.S. Johnson. *Computers and intractability: a Guide to the theory of NP-Completness*, Freeman, San Francisco, 1979.

[Garvey 92]
A. Garvey et V. Lesser. *Design to time real-time scheduling*. Université de Massachussetts, Mars 1992.

[Geist 94]
A. Geist, A Beguelin, J.J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM : a users' guide and tutorial for networked parallel computing.* Mit Press, Cambridge, 1994.

[Glover 89]
F. Glover. *Tabu Search, Part I.* ORSA Journal on computing 1, pp 190-206, 1989.

[Glover 90]
F. Glover. *Tabu Search, Part II.* ORSA Journal on computing 2, pp 4-32, 1990.

[Golberg 89]
D.E. Golberg. *Genetic algorithm in search, optimization and machine learning.* Addison-Wesley, 1989.

[Grass 96]
J. Grass et S. Zilberstein. *Anytime Algorithm Development Tools.* M. Pittarelli (Ed.), SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling, 7(2), 1996.

[Hamidzadeh 91]
B. Hamidzadeh et S. Shekar. *A real-time planning algorithm to meet reponse-time constraints in dynamic environments.* IEEE 1991, Actes de TAI. pp 228-235.

[Hamidzadeh 92]
B. Hamidzadeh et S. Shekar. *Can real-time search algorithm meet deadlines ?* Actes de AAAI-92. pp 486-491

[Hamman 87]
J. Hamman, M. Occio. *Les verres de spin et l'étude des milieux désordonnés.* Pour la science, pp 40-50, Février 1987.

[Hansen 96]
E.A. Hansen et S. Zilberstein. *Monitoring the progress of anytime problem solving.* National Conference on Artificial Intelligence, Portland, Oregon, 1996.

[Harel 87]
D. Harel. *Algorithmics: the spirit of computing.* Addison-Welsey, 1987.

[Haton 89]
J.P. Haton. *Panorama des systèmes multi-agents.* Onzièmes journées francophones sur l'informatique. Architectures avancées pour l'IA.

[Haton 91]
J-P. Haton, N. Bouzid, F. Charpillet, M-C. Haton, B. Lâastri, H. Lâastri, P. Marquis, T. Mondot, A. Napoli. *Le raisonnement en Intelligence Artificielle.* Inter-edition 1991

[Hérault 91]
L. Hérault. *Réseaux de neurones récursifs pour l'optimisation combinatoire.* Thèse de Doctorat à l'Institut National Polytechnique de Grenoble, 1991.

[Hérault 95]
L. Hérault. *Réseaux de neurones pulsés pour l'optimisation - Application à l'allocation de ressources.* Automatique Productique Informatique Industrielle, vol 29, n.4, pp 471-506, 1995.

[Hopfield 85]
J.J. Hopfield et D.W. Tank. (1985) *Neural computation of decisions in optimization problems.* Biological Cybernetics, 52, pp 141-152

[Horvitz 87]
E.J. Horvitz. *Reasoning about beliefs and Actions under Computation Ressources constraints.* Proc. of Workshop on Uncertainty in artificial Intelligence, Seattle, Washington, 1987.

[Ingrand 90]
F.F. Ingrand, M. Georgeff. *Managing deliberation and reasoning in real-time AI systems.* Actes de DARPA-90, Workshop on Innovative Approaches to Planning.

[Ingrand 91]
F.F. Ingrand, M. Georgeff et A.S. Rao. *An architecture for real-time reasonnning and system control.* IEEE, 1991.

[Jackson 55]
J.R. Jackson, (1955) *Scheduling a Production Line to Minimize Maximum Tardiness.* Research Report 43, Management Science Research Project, Univertity of California, Los Angeles, 1955.

[Kirkpatrick 82]
S. Kirkpatrick, C.D. Gelat, M.P. Vecchi. *Optimization by simulated annenealing.* IBM Research Report RC 9355, 1982.

[Korf 85]
R.E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search.* Artificial Intelligence, 27(1). pp 97-109, 1985

[Korf 87]
R.E. Korf. *Real-time Heuristic search: first results.* Actes de AAAI-87. pp 133-138.

[Korf 88]
R.E. Korf. *Real-time Heuristic search: new results.* Actes de AAAI-88. pp 139-144.

[Kayser 88]
D. Kayser, E. Bonte, J. Cataing, P. Grandemange, S. Grumbach et F. Levy. *Description d'un raisonneur à profondeur variable.* Actes d'Avignon-88. pp 118-132.

[Laffey 88]
T.J. Laffey, P.A. Cox, J.L. Schmidt, S.M. Kao et J.Y. Read. Real-Time Knowledge-based systems". AI Magazine. Spring 88. pp 27-45.

[Lalanda 92]
P. Lalanda. *Conduite du raisonnement dans un système à base de tableau noir temps réel.* Thèse de doctorat de l'université Henri Poincaré Nancy I, 1992.

[Lawler 91]
E.L. Lawler, J.K. Lensta, A.H.G. Rinnooy Kan, D.B. Shmoys. *Sequencing and Scheduling: algorithms and Complexity.* Handbooks in Operations Research and management Science, Volume 4: Logistics of Production and Inventory, North-Holland, Amsterdam.

[Lementec 92]
J-C. Lementec. *Atome-TR: A real-time control for blackboard scheduling.* Actes d'Avignon 1992.

[Lesser 88]
V.R. Lesser, J. Pavlin et E.H. Durfee. *Approximate processing in real-time problem solving.* Artificial Intelligence, 9(1). pp 49-61, 1988.

[Lin 73]
S. Lin et B.W. Kernigham. *An effective heuristic for the travelling salesman problem.* Operation Research 21. pp 498-516, 1973

[Liu 91]
J. Liu, K. Lin, W. Shih, A.Yu, J. Chung et W. Zao. Algorithm for scheduling imprecise computation. IEEE Transactins on computer, 24(5). pp 58-68, 1991.

[Mackworth 77]
A.K. Mackworth. Consistency in networks of relations. Artificial Intelligence. Vol 8, pp 99-118, 1977.

[Mensch 94]
A. Mensch, D. Kersual, A. Crespo, F. Charpillet et E. Pessi. *Reakt: real-time architecture for time-critical knowledge-based systems.* Intelligent Systems Engineering, pp 153-167, Automn 1994.

[Metha 93]
S. Metha et L. Fulop (1993) *An analog neural network to solve the hamiltonian cycle problem.* Neural Networks, vol 6, pp 869-881

[Miranker 87]
D.P. Miranker. *TREAT: A New Efficient Match Algorithm for AI Productions Systems.* PhD Thesis, Columbia University, 1987.

[Moore 84]
R.L. Moore, L.B. Hawkinson, C.G. Knickerbocker et L.M. Churchman. *A real-time expert system for process control.* Actes de CAIA-84, pages 569-576.

[Mouaddib 92]
A-I. Mouaddib, F. Charpillet, Y. Gong et J-P. Haton. *Real-time specialist society.* Actes de SICICI-92, IEEE singapore section. pp 751-754.

[Mouaddib 93]
A-I. Mouaddib. *Contribution au raisonnement progressif temps réel dans un univers multi-agents.* Doctorat de l'université Henri Poincaré Nancy I, octobre 1993.

[Mouhoub 96]
M. Mouhoub. *Contribution à l'étude des techniques de propagation de contraintes symboliques et numériques pour le raisonnement temporel.* Thèse de doctorat de l'université Henri Poincaré Nancy I, 1996.

[MPI 94]
Message Passing Interface Forum. *MPI : A message passing interface standard.* International Journal of Supercomputer Application, vol 8, n. 3/4, 1994.

[O'Reilly 85]
C.A. O'Reilly et S.A. Cromarty. *Fast is not Real Time in Designing Effective Real Time Expert AI Systems.* Applications of Artificial Intelligence II548. pp 249-257.

[Portmann 88]
M-C. Portmann. *Méthodes de décomposition spatiales et temporelles en ordonnancment de la production.* RAIRO-APII, 1988, 22, pp 439-451.

[Portmann 93]
M-C. Portmann. *Manufactoring Scheduling.* International Conference on Industrial Engineering and Production Management, Mons (Belgium), June 1993.

[Roy 70]
B. Roy. *Algèbre moderne et théorie des graphes.* Tome 2, Dunod, Paris, 1970.

[Russell 91]
S.J. Russell et S. Zilberstein. *Composing real-time systems.* Actes de AAAI-91. pp 212-217.

[Russell 95]
S.J. Russel et D. Subramanian, Provably bounded-optimal agents. Journal of Artificial Intelligence Research 1. 03/1995. pp 1-36

[Schoppers 87]
M.J. Schoppers. *Universal plans for reactive robots in unpredictable environments.* Actes d'IJCAI-87. pp 1039-1046.

[Shekar 89]
S. Shekhar et S. Dutta. *Minimising Response Times In Real Time Planning And Search*. Actes d'IJCAI-89. pp 238-242.

[Simons 78]
B. Simons.(1978) *A fast algorithm for single processor scheduling.* 19th Annual Symp. Foundations of Computer Science, 246-252.

[Smith 56]
W.E. Smith. *Various Optimizers for Single-Stage Production.* Nav. Res. Log. Quart., 1956, 3, p 59-66.

[Tagliarini 91]
G.A. Tagliarini, J.F. Christ et E.W. Page. *Optimization using Neural Networks.* IEEE transactions on computers, vol 40(12), pp 1347-1358, 1991.

[Varga 62]
R.S. Varga. *Matrix Iterative Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1962.

[Wilson 88]
G.V. Wilson et G.S. Pawley. *On the stability of the TSP algorithm of Hopfield and Tank.* Biological Cybernetics, 58 pp 63-70, 1988.

[Zilberstein 92]
S. Zilberstein et S. J. Russell. *Efficient Resource-Bounded Reasoning*. AT-RALPH, Proc. first AIPS, pp260, 266, 1992.

[Zilberstein 93]
S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms.* Ph.D. dissertation, (Techinal Report N. CSD-93-743), Computer Science Division, University of California, Berkeley, 1993.

[Zilberstein 96]
S. Zilberstein et S. J. Russell. *Optimal Composition of Real-Time Systems.* Artificial Intelligence, vol 82(1-2), pp 181-213, 1996.

Université HENRI POINCARÉ NANCY 1

Nom : **GALLONE**

Prénom : **Jean-Michel**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 13 JAN 1997 UHP 002/97

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY 1 LE PRESIDENT J.P. FINANCE

3C-F9

UNIVERSITÉ HENRI POINCARÉ - NANCY 1
24-30, rue Lionnois B.P.3069 54013 Nancy cedex
Tél : 83 32 81 81 Fax : 83 32 95 90
Adresse électronique : _____ @presn1.u-nancy.fr

Résumé

Dans un environnement dynamique, un système informatique voit ses actions contraintes par le temps et ses performances jugées par le rapport qualité sur temps. Dans cette approche l'objectif n'est plus alors de trouver la réponse de meilleure qualité à un problème donné, mais de chercher le meilleur compromis entre le temps alloué à la réponse et la qualité de la réponse fournie. Le modèle développé dans cette thèse repose sur une résolution progressive d'un problème de façon à disposer d'une première réponse rapidement, quitte à ce que sa qualité soit grossière. L'originalité de l'approche vient de l'utilisation de réseaux neuromimétiques pour la définition d'algorithmes progressifs. Les réseaux auxquels nous nous sommes intéressés réalisent en fait des optimisations de fonctions sous contraintes, ce qui permet de couvrir un large éventail de problèmes. De plus, nous avons réussi à dégager des paramètres d'affinement sur ces réseaux de façon à fournir un ensemble d'algorithmes possédant chacun une qualité et un temps de traitement différents. Ainsi un même problème peut être traité différemment selon l'échéance imposée pour l'obtention de la solution. La seconde étape de nos travaux consiste en la sélection de la suite d'algorithmes à activer en fonction d'une date limite ou d'une loi de probabilité de celle-ci de façon à garantir un résultat qui ait la meilleure qualité possible tout en étant régulièrement amélioré. Ce système a été validé sur une application fondamentale pour le fonctionnement temps-réel qui est l'ordonnancement des tâches.

Mots-clefs : raisonnement sous contrainte de ressources, tâches progressives, algorithmes anytime, ordonnancement, réseaux neuromimétiques, optimisation sous contrainte.

Abstract

In a dynamic environment, systems have time constrained actions and performance is measured by the evolution of quality over time. In such a context, the optimal answer is the one that realizes the best trade-off in computation time versus result quality. In this thesis, we propose a model based on progressive computation in order to quickly get a general result that will be refined afterwards. The originality of our approach is the use of neural networks to define progressive algorithms. We focused our attention on networks that realized constraint optimization, which allows us to tackle a wide range of applications. Furthermore, we succeed in defining refinement parameters on the networks so as to get a set of algorithms each having its own quality and its own computation time. Thus, a given problem can be treated in a different way depending on the deadline for the result. The second step of our work consists of the choice of the activation sequence of the algorithms depending on the deadline or the probability distribution of the deadline in order to guarantee a result that has the optimal quality while being frequently improved. This system has been validated on a fundamental application for real-time computation : the scheduling of non-preemptive tasks.

Keywords : resource bounded reasoning, progressive tasks, anytime algorithms, scheduling, neural networks, bounded optimality.