Numero d'ordre : 2832

Contribution à l'algorithmique anytime : contrôle et conception

THÈSE

présentée et soutenue publiquement le 23 novembre 2000

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Arnaud DELHAY
Ingénieur ISEN

Composition du jury

Président :	Michel PETITOT, Professeur	LIFL, USTL
Rapporteurs :	François CHARPILLET, Chargé de Recherche	INRIA-Lorraine, Nancy
	Shlomo ZILBERSTEIN, Professeur,	University of Massachussets
Examinateurs :	Abdel-Illah MOUADDIB, Maître de Conférence	CRIL, Université d'Artois, Lens
	Philippe SARAZIN, Ingénieur	DGA/DSP/STTC/DT/SC, Paris
	Patrick TAILLIBERT, Ingénieur	Thomson-CSF Detexis, Elancourt
Directeurs :	Max DAUCHET, Professeur	LIFL, USTL
	Philippe VANHEEGHE, Professeur	LAIL, Ecole Centrale de Lille

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U F R d'I E E A - Bât M3 - 59655 VILLENEUVE D'ASCQ CEDEX
Tél +33 (0)3 20 43 47 24 - Télécopie +33 (0)3 20 43 65 66 - e-mail direction@lifl.fr

Mis en page et composé avec LaTeX et la classe thlifl.

Remerciements

Le travail présenté dans ce mémoire a été réalisé au sein du Laboratoire d'Informatique Fondamentale de Lille (LIFL UPRESA 8022) et au sein du Laboratoire Signaux et Systèmes de l'ISEN. Il a été financé par une bourse DGA/CNRS.

Mes plus vifs remerciements vont à Monsieur Max Dauchet, mon directeur de thèse, qui m'a encadré durant ces trois années et qui a su entretenir ma motivation par son expérience, son enthousiasme et sa passion de tous les instants. Que Monsieur Philippe Vanheeghe, Professeur à l'Ecole Centrale de Lille, soit associé à ces remerciements pour avoir participé à cet encadrement, pour ses conseils avisés, ses nombreuses questions aussi "génantes" qu'intéressantes, et son humour.

Je témoigne toute ma gratitude à Monsieur Michel Petitot, Professeur à l'Université des Sciences et Technologies de Lille, pour avoir bien voulu présider le jury de cette thèse.

Je suis très reconnaissant envers Monsieur François Charpillet, Chargé de Recherche au LORIA de Nancy, d'avoir été rapporteur de cette thèse et d'avoir pris le temps d'examiner son travail.

Je remercie très vivement Monsieur Shlomo Zilberstein, Professeur à l'Université du Massachussets à Amherst, d'avoir accepté d'être rapporteur de cette thèse et pour les discussions passionnantes que nous avons eues ainsi que le travail que nous avons pu effectuer ensemble.

Je remercie également Monsieur Abdel-Illah Mouaddib, Maître de Conférence à l'Université d'Artois d'avoir accepté de participer à ce jury et pour les travaux que nous avons effectués ensemble.

Que Monsieur Philippe Sarazin de la Délégation Générale pour l'Armement soit également remercié pour sa participation au jury et pour avoir cru à ce sujet de recherche et aidé à le promouvoir à la DGA.

Je remercie vivement Monsieur Patrick Taillibert de la société Thomson-CSF Detexis d'avoir été à l'initiative de ce sujet de thèse et d'avoir collaboré à ce travail par ses longues discussions passionnées et ses critiques constructives, ainsi que par sa participation active à la mise en pratique de cette étude.

Je remercie Monsieur Jean-Marc Geib et Monsieur Bernard Toursel, directeurs du LIFL pour leur accueil au sein de leur laboratoire, et Monsieur Jean-Noël Decarpigny, trop tôt disparu, et Monsieur Léon Carrez, directeurs de l'ISEN, pour leur accueil au sein de leur établissement.

Que toutes les personnes ayant participé à ce travail, soit directement par leur contribution, leurs commentaires, leurs relectures ou leur aide pratique, soit indirectement en me soutenant et me supportant durant ces trois années soient associées à ces remerciements. J'ai une pensée particulière pour mes plus proches collègues de l'ISEN : Laurent, Manu et Manu (ils se reconnaitront), Philippe, Annemarie, Christophe, Marie, Dominique L., Jean-Michel, Dominique D., Pascal, Christelle, Roberte, Marie-Christine; et du LIFL : Jean-Stéphane, Sébastien, Bruno et Annie (qui m'a guidé vers le plus court chemin dans le labyrinthe administratif). Merci à mes collègues de l'IUT de Lannion pour leur accueil et leur aide dans la dernière ligne droite.

Merci à tous de m'avoir aidé à arriver jusque là. Recevez ma reconnaissance et mon amitié.

à mes Parents,
à Nathalie,
à ma Famille.

Table des matières

Introduction

La perfection d'une pendule n'est pas d'aller vite mais d'être bien réglée
[*Reflexions et Maximes*, Vauvenargues, 1746]

1 Complexité algorithmique et temps-réel

La prise en compte du temps-réel est un aspect essentiel dans un nombre d'applications de plus en plus important, qu'elles concernent des domaines comme l'aide à la navigation, la régulation de systèmes, l'aide à la décision, la simulation, la planification ou encore le diagnostic. Le soucis est alors d'obtenir une solution exploitable dans les délais imposés par les contraintes du domaine.

Il convient de préciser la notion de temps-réel qui est parfois utilisée abusivement. Le temps-réel ne signifie pas tant qu'un processus doit être rapide, mais un peu plus que cela: tous les programmes doivent être assez rapides pour répondre aux exigences de leur utilisateur, et dire qu'un programme est rapide ne suffit pas à le rendre temps-réel. Le temps-réel signifie de travailler dans un environnement où il existe des *stimuli* imprévisibles, des erreurs dues à des pannes de composants du système, de l'incertitude sur l'état de l'environnement, sur les effets de tous ces *aleas* sur les performances du système [BB92]. La rapidité est seulement une partie du temps-réel et doit être complétée par une garantie de réponse à une date donnée. En résumé, il faut à un processus temps-réel qu'il soit *assez* rapide pour répondre à des sollicitations extérieures. Par exemple, il doit être capable de fournir une solution *exécutable* dans un délai fixé à l'avance.

Un des aspects les plus comprométants pour nombre d'applications et leur intégration dans un système temps réel est que beaucoup de ces problèmes sont de complexité élevée (problèmes dits NP-durs) et sont donc réputés impraticables tant que P≠NP [GJ79]. C'est en particulier le cas pour les applications dites d'Intelligence Artificielle (IA) et leur adaptation à un environnement temps-réel est un des importants sujets de recherche actuels [MHA+95]. La plupart de ces applications conduisent d'ailleurs à des problèmes de très forte complexité combinatoire et le temps de calcul, même pour des instances de taille modeste, peut devenir catastrophiqement élevé.

Les algorithmes en découlant ne peuvent donc pas être "rapides" avec une instance de taille réaliste pour l'application envisagée. Il ne peut donc même pas être question de temps-réel, d'autant que la résolution des cas moyens (qui donnent le temps de calcul

moyen) peut être très éloignée du pire cas. En résumé, il existe deux types de problèmes : la longueur rédhibitoire du traitement total et le manque de prédictibilité quant aux temps de calculs.

Plusieurs solutions se présentent alors pour respecter les contraintes de temps. Parmi celles-ci, il peut s'agir de développer des techniques *ad-hoc* afin de garantir et de contrôler les temps d'exécution. Cette solution, souvent utilisée dans l'approche temps-réel classique, ne permet pas le développement d'applications très "intelligentes". Un autre moyen est d'essayer de se contenter d'une solution dégradée, plutôt que d'attendre la fin du calcul et le résultat complet. Les notions de qualité et d'utilité évaluent alors la réponse intermédiaire donnée par l'algorithme en fonction du temps de calcul écoulé. C'est ce que permettent de faire les algorithmes anytime.

2 Les algorithmes anytime

Les algorithmes anytime [DB88] sont un moyen de faire un compromis entre les exigences algorithmiques d'applications complexes et du temps-réel. Ils s'inscrivent dans le cadre de ce qu'on appelle le raisonnement en temps contraint et ont de nombreuses alternatives comme les algorithmes approximatifs [Hor87], le raisonnement progressif [Mou93] qui dérivent tous de la même idée de compromis entre les performances d'un algorithme et son temps de calcul.

L'idée des algorithmes anytime, plus particulièrement pour les problèmes de complexité élévée, est d'arrêter le calcul avant la construction de la solution optimale (c'est à dire la solution fournie par un algorithme complet), très coûteuse en temps, et de se contenter d'un résultat intermédiaire. Ce résultat est quantifié à l'aide d'un critère de qualité qui permet, par exemple, de le situer par rapport à la solution optimale. Le profil de performance, graphe de la qualité en fonction du temps, retranscrit ce compromis.

De nombreux travaux sur la gestion de ce compromis fourni par les algorithmes anytime existent. Ils consistent principalement à optimiser la qualité du résultat final fourni par un algorithme ou un ensemble d'algorithmes pour un temps de calcul donné. Des études prennent également en compte l'incertitude qui peut exister sur le temps de calcul disponible.

Dans cette thèse, nous proposons d'étudier une situation dans laquelle un évènement hostile vient interrompre le calcul et où il faut fournir une réponse exploitable au moment de cette interruption. La date d'occurence de l'évènement n'est pas connue exactement, mais est seulement définie par un intervalle sur lequel l'évènement a une probabilité uniforme d'arriver. Ce problème, facilement soluble à l'aide d'algorithmes anytime interruptibles, n'est pas aussi évident quand on ne dispose que d'algorithmes anytime à contrat qui, eux, ne sont pas interruptibles.

ZILBERSTEIN et RUSSEL [ZR96] proposent une solution qui permet de transformer un algorithme à contrat en un algorithme interruptible en minimisant le coefficient multiplicatif sur le temps nécessaire pour avoir la même qualité que pour l'algorithme original.

Nous proposons une autre approche, la maximisation de la qualité moyenne, qui permet de donner le plus souvent la meilleure qualité, sur l'ensemble des occurences possible de l'évènement hostile. La distribution de la probabilité d'occurence de cet évènement est uniforme sur l'intervalle, car nous supposons n'avoir aucune information sur sa date d'arrivée.

Il existe plusieurs méthodes d'ordonnancement des algorithmes anytime [Zil93] [HZ96], en supposant qu'il soit facile de construire les algorithmes anytime. S'il existe des travaux sur une standardisation de la conception des algorithmes anytime [GZ96], ils supposent que le critère de qualité et la fonction qui permet de générer les entrées de l'algorithme pour la construction du profil de performance existent et ont été construites par le concepteur de l'algorithme. Les problèmes de choix du critère de qualité et de la fonction de génération des entrées représentatives du fonctionnement de l'algorithme dans son contexte d'utilisation, même si elles semblent fortement dépendantes de l'application, ne sont pas pour autant aisés à résoudre. C'est pourquoi nous proposons, par l'intermédiaire d'expérimentations à la fois simples et représentatives, de soulever les problèmes qu'il est possible de rencontrer dans ce cadre. Ainsi, nous verrons, grâce à un apport théorique de la théorie algorithmique de l'information, encore appelée complexité de Kolmogorov [LV90] [LV97] [Dub98], que l'approche aléatoire pour le choix des entrées n'est pas forcément la plus appropriée pour obtenir des profils de performance représentatifs des cas réels. Cette approche nous montrera qu'il n'est pas aisé de construire des profils de performance.

La thèse que nous présentons dans ce document apporte donc deux contributions principales indépendantes, l'une sur le plan de l'ordonnancement des algorithmes à contrat, l'autre sur un plan plus pratique de la conception des algorithmes anytime, et notamment sur le choix d'un critère de qualité et sur la production de profils de performances à partir d'entrées choisies.

3 Organisation du document

Dans un premier chapitre, nous présentons l'approche des algorithmes anytime. Nous définissons dans une première partie les notions de qualité, d'utilité et de profil de performance. Nous distinguons deux types d'algorithmes anytime et décrivons comment les utiliser. Dans une seconde partie, nous présentons les principales techniques de gestion de tels algorithmes, et notamment la problématique de composition des algorithmes anytime. Quelques alternatives à l'approche algorithmique anytime sont présentées à cette occasion.

L'approche consistant à maximiser la qualité moyenne sur un intervalle de temps sera l'objet du deuxième chapitre. Nous présenterons tout d'abord le contexte de ce problème pour ensuite proposer quelques résultats analytiques préliminaires. Nous démontrons que si le profil de performance est de forme convexe (centre de courbure vers le haut) alors une stratégie à un seul contrat donne la qualité moyenne maximale. Au contraire, nous prouvons que dans le cas de profils de performances concaves (centre de courbure vers le bas, encore appelé profil de performance avec *diminishing returns*), deux contrats choisis

pour maximiser la qualité moyenne donne un meilleur résultat qu'un seul contrat. Dans le cas d'une stratégie à un seul contrat nous prouvons que le contrat est supérieur (respectivement inférieur) à la moitié de l'intervalle si le profil de preformance est convexe (respectivement concave). Dans un deuxième temps, nous considérons ce problème du point de vue discret à l'aide d'une approximation du profil de performance par une fonction constante par morceaux qui garantit la même erreur sur la qualité moyenne que sur le profil de performance. Nous démontrons qu'avec cette aproximation, il suffit de choisir les contrats parmi les seuils de la fonction constante par morceaux et que le problème d'optimisation combinatoire résultant est NP-difficile en général. Dans le cas où l'intervalle est assez grand pour contenir tous les seuils, le problème devient polynomial et nous donnons un algorithme. Enfin, quelques expérimentations nous permettent de discuter de la mise en oeuvre de cette approche et montrent que la maximisation de la qualité moyenne peut être raisonnable en pratique si on accepte de faire une erreur (minime) en fixant le nombre de contrats.

Le troisième et dernier chapitre aborde l'aspect pratique de la conception des algorithmes anytime et est indépendant du deuxième chapitre. Nous nous plaçons dans la situation d'un programmeur voulant concevoir un algorithme anytime pour la première fois. Nous nous intéressons plus particulièrement aux questions du choix d'un critère de qualité pertinent et du choix des entrées qui vont servir à construire le profil de performance de l'algorithme. Dans un premier temps, nous montrons par des exemples simples, des algorithmes de tri, que les problèmes de choix du critère de qualité ne sont pas simples. Nous montrons par exemple qu'il existe des critères qui n'ont de sens que pour un algorithme, même si le problème général reste le tri. Ensuite, nous abordons la question du choix des entrées représentatives du point de vue expérimental, toujours à l'aide des algorithmes de tri. Nous exhibons des comportements gênants pour la construction de profils de performance en faisant varier des paramètres comme la taille des entrées, les façons de générer les entrées. Nous mettons alors en cause le choix de la génération aléatoire pour la génération d'entrées représentatives de la réalité. Nous montrons alors comment la théorie de la complexité de Kolmogorov peut donner un éclairage sur ce problème. Quelques exigences techniques caractéristiques de l'implémentation des algorithmes anytime sont également évoquées en fin de ce chapitre.

Enfin, nous concluons cette thèse et envisageons les perspectives à ce travail.

Chapitre 1

Un compromis temps de calcul/performance

Attendons un peu pour finir plus vite [*Essais*, Francis Bacon, 17ème siècle]

Implémenter des problèmes réputés complexes dans des environnements temps réel est souvent impossible pour cause de contraintes de temps trop fortes ou alors d'algorithmes trop coûteux en temps de calcul. Il faut alors se contenter soit d'algorithmes *ad hoc* afin de tenir les contraintes du temps réel au prix d'une certaine perte "d'intelligence", soit accepter d'utiliser une solution dégradée, incomplète mais donnée dans les délais impartis. Faire un compromis entre le temps de calcul et la qualité du résultat produit par l'algorithme semble donc être une bonne approche pour garder toute la richesse du problème initial et résoudre ce dilemme.

Ce chapitre a pour objectif de passer en revue différentes techniques existantes permettant de faire un tel compromis. Nous exposons donc tout d'abord les bases de l'algorithmique anytime. La composition de modules anytime proposée par ZILBERSTEIN [Zil93] et RUSSEL [ZR96] sera également exposée. Finalement, nous présenterons différentes approches de gestion des algorithmes anytime, permettant de raisonner sur le temps alloué à de tels algorithmes afin d'optimiser leurs performances. Ce sera également l'occasion d'évoquer quelques alternatives aux algorithmes anytime.

1.1 Algorithmique anytime

Le terme d'algorithme *anytime* a été employé pour la première fois par DEAN et BODDY [DB88] à la fin des années 80 dans un travail concernant la planification temporelle (*time dependant planning*). Dans le même temps, en 1987, la notion d'algorithmique flexible (*flexible computation*) était introduite par HORVITZ [Hor87] dans un contexte d'aide à la décision en temps contraint. Depuis, les travaux dans le domaine de l'algorithmique anytime étendent de plus en plus leur champ d'application, passant par des problèmes incluant de la gestion de capteurs [ZR93], par d'autres concernant l'optimisation

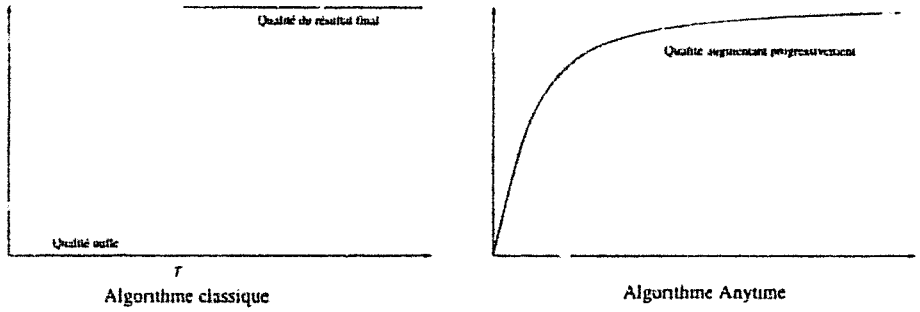


FIG. 1.1 – *Différence entre un algorithme classique et un algorithme anytime*

de requêtes dans les bases de données [SD89], ou encore par des problèmes de planification [MC96], cette liste n'étant pas exhaustive. Des travaux notables sur la composition de modules anytime ont été effectués par ZILBERSTEIN [Zil93] et RUSSEL [ZR96]. Nous présenterons ces travaux dans ce chapitre. A notre connaissance, la plupart des recherches actuelles sur l'algorithmique anytime portent sur la gestion de ces algorithmes. Il s'agit de trouver la meilleure stratégie pour obtenir le meilleur ordonnancement de plusieurs algorithmes et les meilleures allocations de temps. Le but de ces stratégies d'ordonnancement est d'obtenir la performance optimale de l'algorithme utilisé et ainsi de garantir le meilleur résultat au moment exigé. C'est ce même objectif que vise MOUADDIB dans une alternative aux algorithmes anytime, le *raisonnement progressif* [Mou93] [MZ95]. Nous reviendrons sur ces travaux dans le paragraphe 1.3.4.

Les algorithmes anytime offrent donc la possibilité d'un compromis temps de calcul/performance défini par un profil de performance. C'est-à-dire, comme illustré sur la figure 1.1, qu'au lieu d'exploiter un résultat final, il est possible d'obtenir des résultats intermédiaires, donc non optimaux au sens du résultat final, mais dont on peut quantifier la valeur en fonction du temps de calcul écoulé. Comme dans le cas de problèmes NP-difficiles, les temps de calculs peuvent devenir très vite rédhibitoires, l'avantage d'un algorithme anytime est de pouvoir stopper les calculs au moment où l'on estime que le résultat est satisfaisant en terme de qualité. De plus, il est fréquent que les calculs restant à effectuer pour atteindre le résultat final soient des raffinements très coûteux en temps mais peu avantageux en terme d'amélioration du résultat. Réciproquement, le graphe représentant l'évolution de la qualité en fonction du temps permet de donner la qualité d'un résultat pour un temps de calcul fixé à l'avance.

1.1.1 Définitions

Nous reprenons tout d'abord la définition donnée par BODDY [Bod91].

Définition 1.1 *Algorithme anytime*

Un algorithme anytime est un algorithme itératif qui garantit de produire une réponse à toute étape du calcul, où la réponse est supposée s'améliorer à chaque itération.

Nous complétons cette définition avec les caractéristiques originelles données par DEAN et BODDY [DB88] :

- préemptibilité : les algorithmes anytime peuvent être suspendus et relancés au même point avec un temps de réponse négligeable afin de se prêter au techniques d'ordonnancement
- interruptibilité : ils peuvent être interrompus à tout moment

Les *algorithmes anytime* ou *flexibles* [Hor87] [Hor88] sont des algorithmes qui travaillent de façon itérative sur un problème afin de produire des résultats dont la qualité augmente avec le temps. Toutefois, il n'est pas nécessaire que l'algorithme se trouve sous forme itérative, mais seulement que la qualité du résultat de cet algorithme varie de façon itérative. Par exemple, un algorithme de tri rapide(*quicksort*) sous forme récursive améliore le résultat à chaque récursion en tendant vers la liste triée.

Afin de quantifier les performances de l'algorithme anytime et d'exploiter le compromis performance/temps de calcul, il est nécessaire de disposer des notions suivantes :

la mesure de performance : à la place de la notion binaire de performance (l'algorithme n'est pas terminé ou est terminé et fournit le résultat optimal), un algorithme anytime retourne un résultat intermédiaire d'une certaine *qualité* 1.1.

la capacité de prédiction : les algorithmes anytime "contiennent" des informations sur leur propre comportement au cours du calcul. Ces informations sont synthétisées dans les profils de performance et permettent de gérer les ressources de temps que l'on va allouer à l'algorithme.

Dans les paragraphes suivants, nous allons donc définir ces deux notions. Nous verrons ensuite quels sont les différents types d'algorithmes anytime.

1.1.1.1 Utilité et Qualité

Afin d'évaluer de façon quantitative les performances d'un algorithme, il est nécessaire de définir des critères de performance. La *qualité* est une évaluation du résultat brut fourni par un algorithme. Pour prendre en compte cette performance dans un environnement réel, on introduit alors le critère d'*utilité*. Nous allons immédiatement illustrer ces notions avec deux exemples.

Exemple 1 : Système de diagnostic médical.

Imaginons un système destiné à détecter une maladie ou un quelconque problème de santé d'un patient. Ce système prend donc des diverses mesures comme la tension artérielle, le pouls, etc. Avec ces mesures, il essaie de déterminer un diagnostic sachant que plus il aura de temps pour le faire, plus ce diagnostic sera fin. Malheureusement, pendant que le système calcule, le patient concerné voit sa santé se dégrader dangeureusement (situation aux urgences par exemple). Et plus la santé du patient se dégrade, moins le diagnostic, aussi fin soit-il, sera exploitable. On dit que le diagnostic perd de son utilité au cours du temps.

Exemple 2 : Le voyageur de commerce

Le problème du voyageur de commerce est de trouver un trajet de coût minimum passant une et une seule fois par chaque ville, partant d'une ville quelconque et y revenant en fin de parcours. Au fur et à mesure de l'optimisation du trajet du voyageur de commerce, le coût de ce dernier diminue. La qualité du résultat de cet algorithme augmente donc. Pendant que cet algorithme optimise le trajet, le temps passe (et il peut passer très vite avec un tel problème NP-difficile à résoudre). Les clients potentiels que le voyageur doit visiter peuvent alors s'impatienter, changer de fournisseurs, etc. Là encore, l'intérêt de continuer l'optimisation chute avec le temps et le résultat fourni perd son utilité.

Nous pouvons en tirer les définitions qui suivent :

Définition 1.2 *La qualité*

Pour un algorithme donné, la qualité du résultat $Q(t,d)$ de l'algorithme est une application de l'ensemble des couples (temps de calcul t, donnée en entrée d) dans $\mathbb{R}$ *ou* $[0,1]$ *(quand il est possible de normaliser).*

La qualité est la mesure de la performance brute de l'algorithme en fonction du temps de calcul. C'est-à-dire que l'on considère l'algorithme dans l'absolu, et non pas intégré dans un système placé dans un environnement donné. C'est le bénéfice absolu d'obtenir un résultat. Elle peut être, ou ne pas être, reliée à la fonction d'utilité du système qui inclut l'algorithme. Elle doit mesurer certaines caractéristiques de l'algorithme qui s'améliorent avec le temps. La qualité peut être discrète ou continue et définie empiriquement ou selon un modèle.

Définition 1.3 *L'utilité*

Pour un algorithme et un environnement donnés, l'utilité $U(q,t)$ d'un résultat est une application de l'ensemble des couples (temps de calcul, qualité) dans $\mathbb{R}$ *ou* $[0,1]$.

Souvent employée dans un contexte réactif, l'utilité définit la valeur du résultat de l'algorithme en fonction des changements de l'environnement dans le temps. C'est une fonction arbitraire et subjective de mesure de performance de l'algorithme, car elle dépend du contexte (environnement et système) dans lequel elle est employée. Des modèles simples intègrent fréquemment le coût du "raisonnement" (du calcul par rapport à l'urgence de la situation) et le coût de retarder une action.

Pour distinguer ces deux notions, HORVITZ [Hor87] parle de *Object-value utility* et de *Inference-related utility*, respectivement. Nous illustrons ces deux notions de qualité et d'utilité sur la figure 1.2, en faisant intervenir le coût du temps, qui, ajouté à la qualité, donne l'utilité résultante.

Il reste toutefois à choisir le bon critère de qualité (ou d'utilité) afin d'évaluer les performances d'un algorithme. Voici quelques exemples de critères d'évaluation [Zil93]:

- Précision : reflète le degré de précision dans la valeur retournée par l'algorithme. C'est typiquement le critère utilisé pour les algorithmes de calcul numérique. La

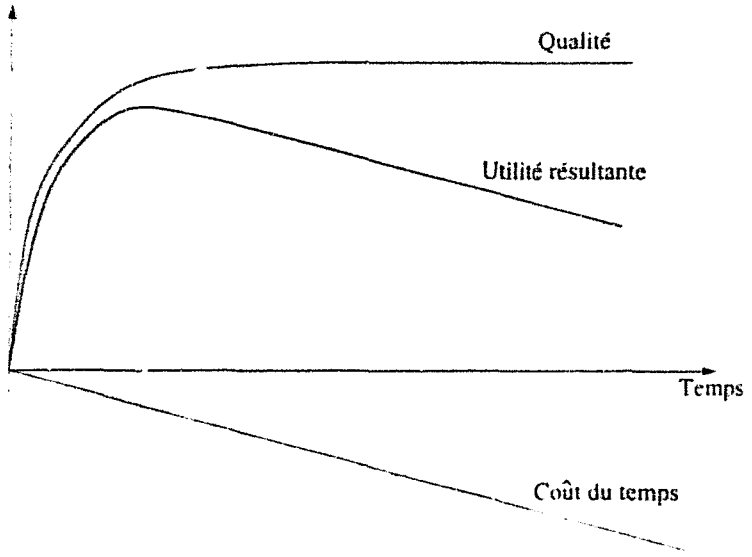


FIG. 1.2 - *Qualité, Coût du temps et Utilité*

qualité peut par exemple être définie par la formule suivante : $1 - 2^n$ où n est le nombre de chiffres significatifs. C'est une estimation de l'erreur faite en limitant le nombre de chiffres significatifs.

- Certitude : reflète la probabilité de correction du résultat. Considérons par exemple, un processus de diagnostic anytime qui prend en compte de plus en plus de paramètres mesurés avec le temps. La certitude que le diagnostic est correct est alors bien une fonction croissante du temps.
- Spécificité : reflète le niveau de détail du résultat. Dans ce cas, l'algorithme anytime produit toujours des résulats corrects, mais le niveau de détail est augmenté avec le temps. Prenons par exemple un algorithme de planification hiérarchique qui retourne tout d'abord un plan abstrait de haut niveau. Chaque étape dans le plan peut être rafinée en continuant la planification. Quand le temps d'exécution augmente, le niveau de détail augmente jusqu'à ce que le plan soit composé d'opérateurs élémentaires qui peuvent être exécutés facilement. Un plan détaillé peut être exécuté plus rapidement qu'un plan de haut niveau et a une plus grande qualité.

Ce critère est donc fortement dépendant de l'application. Nous allons maintenant nous attacher à la représentation de la qualité, qui est appelée *profil de performance* et que nous exposons dans la suite.

1.1.1.2 Profils de performance

Généralités

L'intérêt du profil de performance est principalement de synthétiser tout la capacité de prédiction que nous avons sur le comportement de l'algorithme. Il a en plus l'avantage

d'offrir une représentation visuelle de la qualité en fonction du temps. De plus, l'expression de la qualité en fonction du temps par une formule mathématique n'est pas toujours possible, et c'est le profil de performance, représentation en extension, qui est alors le plus approprié.

Le profil de performance peut avoir différentes présentations, dont celles qui suivent.

Forme compacte : La forme compacte est appelée ainsi car elle permet de définir le profil de performance de manière synthétique, le plus souvent à l'aide d'une fonction mathématique.

Forme discrète : La forme discrète revient à diviser le temps et la qualité en un nombre fini de parties, ce qui revient à représenter le profil de performance dans un tableau spécifiant la qualité pour un éventail d'allocations de temps.

Quand on construit le profil de performance, on s'intéresse à la représentation graphique de la qualité $Q(t,d)$ avec t le temps alloué, et d la donnée en entrée qui varie. Selon les cas, le profil de performance choisi sera d'un type différent :

- *profil de performance idéal* : si la qualité ne dépend que du temps, on a un profil de performance idéal, sur une courbe unique.
- *profil de performance par intervalle (Interval Performance Profile ou IPP)* : si la qualité varie en fonction de la donnée en entrée, et que l'on peut définir deux fonctions $Q_1(t)$ et $Q_2(t)$ telles que $\forall d, Q_1(t) \leq Q(t,d) \leq Q_2(t)$.
- *profil de performance moyen (Expected Performance Profile ou EPP)* : si la qualité varie en fonction de la donnée en entrée, on définit alors un profil moyen sur l'ensemble des courbes de qualité obtenues pour des données en entrée fixées.

$$Q(t) = \frac{\sum Q(t,d)}{nbre\ de\ d}$$

- *profil de performance conditionnel (Conditional Performance Profile ou CPP)* : si la qualité varie en fonction d'une propriété de la donnée en entrée, qui peut être sa qualité selon le même critère qu'en sortie ou tout autre moyen de mieux spécifier le comportement de l'algorithme en fonction de l'entrée.
- *profil de performance probabiliste (Distributed Performance Profile ou DPP)* : si la qualité varie en fonction de la donnée en entrée, et que l'on affecte pour un temps de calcul t donné, une probabilité d'apparition de la qualité q calculée empiriquement à partir de plusieurs exécutions de l'algorithme sur l'ensemble des données. $P(q,t)$ ou $P(q|t)$ est la probabilité d'obtenir une qualité q pour un temps de calcul t.

Cette liste n'est bien sûr pas exhaustive et il est possible de combiner plusieurs de ces types de profil de performance, toujours dans le but de mieux prédire le comportement de l'algorithme anytime utilisé.

Nous avons vu l'essentiel des formes que pouvaient revêtir les profils de performance Voyons maintenant les différentes façons de les construire.

Construction de profils de performance

Nous pouvons distinguer essentiellement trois méthodes de construction de profils de performance: l'analyse de l'algorithme et représentation par une fonction, la méthode statistique, et enfin, la méthode de la trajectoire.

analyse de l'algorithme: Cette méthode consiste à faire une analyse de l'algorithme afin de déterminer sa qualité en fonction du temps. Le profil de performance est déduit directement du tracé de la fonction. L'avantage est de manipuler une fonction plutôt que le profil de performance. Par contre, l'analyse est délicate, voire impossible dans le cas général. Par exemple, le comportement d'un planificateur est très difficile à évaluer autrement que dans le pire des cas.

méthode statistique: Très souvent utilisée, elle consiste à observer le comportement de la qualité du résultat en se basant sur un échantillon d'entrées, que l'on espère représentatif, afin d'obtenir une *carte de la qualité (quality map)*, c'est-à-dire la description brute des couples (temps, qualité) mesurés comme sur la figure 1.3. On peut ensuite faire une approximation du profil en construisant un *profil de performance moyen (EPP)* [Zil93] [Zil96], ou en définissant une distribution de probabilité sur la qualité de la sortie pour obtenir un *profil de performance probabiliste*. Cette méthode de construction empirique des profils de performance peut présenter des difficultés, car il s'agit de choisir judicieusement les entrées qui vont servir à "l'apprentissage" du comportement de l'algorithme. Nous revenons sur ces difficultés dans le chapitre 3.

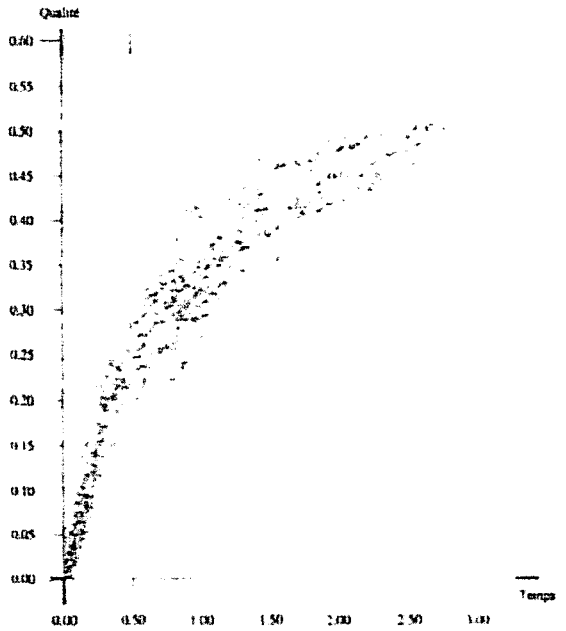


FIG. 1.3 - *Carte de la qualité de l'algorithme du voyageur de commerce construit avec la méthode statistique [Gra96]. Le critère de qualité est le rapport du coût du chemin optimal sur le coût du chemin courant*

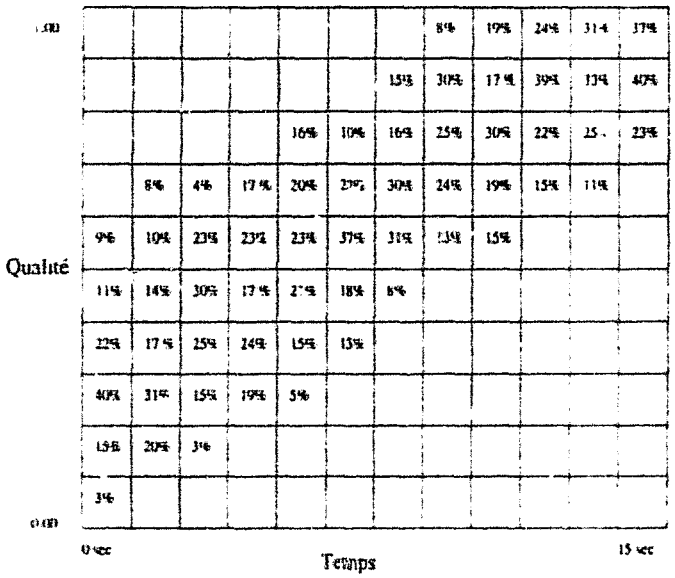


FIG. 1.4 - *Profil de performance probabiliste de l'algorithme du voyageur de commerce sous forme discrète tabulaire [Gra96]*

méthode de la tangente : A partir d'une qualité à l'instant présent et de sa tendance d'évolution définie par sa tangente, cette méthode permet de prédire le comportement futur du profil de performance (figure 1.5). Malheureusement, cette prédiction n'est valable qu'à court terme et n'est donc pas généralisable. En effet, rien n'empêche que le profil de performance connaisse plusieurs inflexions durant le calcul. La tangente connaît alors de fortes variations et il devient difficile de faire une prédiction efficace.

Nous pouvons constater que la manière de construire le profil de performance d'un algorithme influencera fortement la représentation de celui-ci. Pour une méthode de construction statistique, on utilisera de préférence une forme discrète tabulaire (comme sur la figure 1.4), alors que pour une analyse algorithmique, quand elle est réalisable, on emploiera plus volontiers une forme compacte, la formule analytique correspondant au profil trouvé.

Dans la plate-forme proposée par GRASS et ZILBERSTEIN [GZ96], les auteurs décrivent la génération du profil de performance avec les étapes qui suivent. Il s'agit de la construction d'un profil de performance probabiliste conditionnel, c'est-à-dire que le profil de performance est toujours sous la forme d'une distribution sur la qualité de sortie pour chaque temps d'exécution fixé, mais en plus, il dépend de la qualité des instances que l'on met en entrée de l'algorithme. Dans cet exemple particulier, il s'agit du profil de performance du tri à bulle et la qualité des instances est évaluée comme la qualité de la sortie de l'algorithme. Il est toutefois possible de ne pas choisir le même critère en entrée et en sortie.

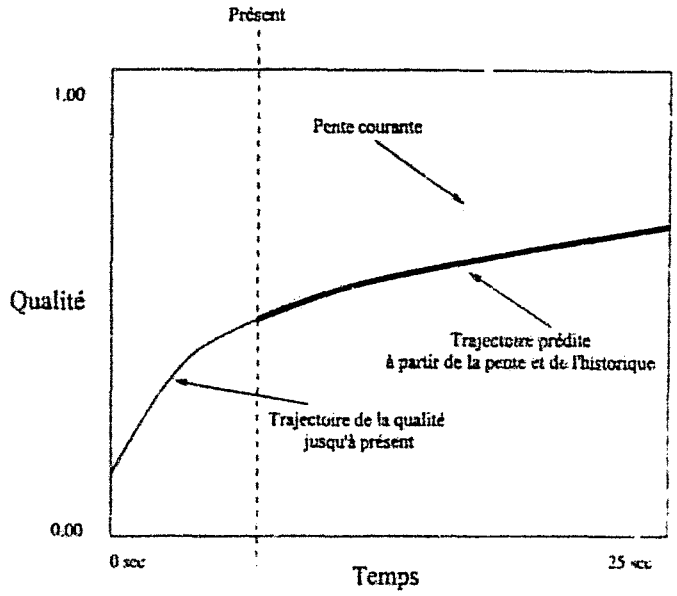


FIG. 1.5 – *Profil de performance par la méthode de la tangente [Gra96]*

Voici donc les étapes de construction décrites :

- estimation du temps de calcul total en utilisant l'algorithme sur quelques instances. On définit alors la taille et la résolution du profil de performance (granularité du temps mesuré).
- génération automatique d'un grand nombre d'instances.
- enregistrement de l'évolution de la qualité pour chaque instance. On construit la carte de qualité.
- à partir de la carte de qualité, on construit la distribution de qualité pour chaque temps donné. On obtient alors un profil de performance probabiliste.
- réitération du processus pour chaque qualité d'entrée choisie pour obtenir un profil de performance conditionnel.

Utilisation des profils de performance

A l'aide des profils de performance, il est possible de prédire que, pour un type d'entrée donné, un temps d'allocation donné, nous obtiendrons une qualité déterminée par le profil. Mais il est aussi possible de prédire, pour un type donné d'entrée, le temps nécessaire pour obtenir une qualité donnée, dans la fenêtre d'utilisation du profil, bien entendu. Ces possibilités de prédiction sont un avantage indéniable par rapport aux algorithmes classiques. Ainsi, est-il possible de gérer les temps de calcul d'algorithmes selon le temps disponible dans une situation réelle et l'étude des stratégies de gestion de ce temps de calcul en fonction de la qualité fait l'objet de la plupart des études sur l'algorithmique anytime. Nous en présenterons quelques unes dans les sections qui suivent. Il est donc possible avec cette approche de savoir s'il est valable de continuer des calculs dans le cas

de problèmes complexes ou si le résultat calculé est suffisant et ainsi arrêter des calculs très longs et augmentant peu la qualité du résultat.

En complément des notions de qualité et de profil de performance, nous résumons ici un certain nombre de propriétés désirables proposées par ZILBERSTEIN pour construire un algorithme anytime. Ces propriétés sont préconisées dans l'objectif du contrôle des algorithmes anytime pour la maximisation de la qualité du résultat qu'ils fournissent [Zil96].

une qualité mesurable : Quand la qualité d'un résultat intermédiaire peut être calculée précisément. Par exemple, dans le cas d'un calcul numérique convergent vers une solution optimale, la qualité est mesurable si on a une estimation de l'erreur.

une qualité reconnaissable : quand la qualité d'un résultat peut être calculée facilement pendant l'exécution (c'est-à-dire en temps constant). Par exemple, dans un problème de planification de trajectoire (*path planning*), la qualité du résultat dépend de la distance à l'optimum. Dans ce cas la qualité est mesurable, mais pas reconnaissable.

la consistence du profil de performance : quand la qualité du résultat est corrélée au temps de calcul et à la qualité d'entrée. En général, les algorithmes anytime ne garantissent pas une qualité de réponse déterministe étant donné un temps de calcul, mais il est important d'avoir une "variance étroite" pour que la prévision de qualité soit possible.

le *diminishing returns* : l'amélioration de la qualité est plus grande au début du calcul et diminue avec le temps. Cette propriété est très importante pour l'optimalité de certaines techniques de contrôle que nous verrons plus loin. Toutefois, il peut exister des profils de performance qui n'ont pas cette propriété.

1.1.1.3 Types d'algorithmes anytime

Généralités

Si la définition originelle de DEAN et BODDY ne laisse aucun doute sur l'interruptibilité des algorithmes anytime, il existe un autre type, les algorithmes anytime à contrat, proposés par ZILBERSTEIN [Zil93]. C'est à la façon d'allouer le temps que la distinction entre algorithme anytime interruptible et algorithme anytime à contrat est faite.

Définition 1.4 *Algorithmes anytime interruptibles*

Les algorithmes interruptibles peuvent être interrompus à n'importe quel moment pour donner un résultat partiel, puis reprendre à l'endroit du calcul où ils étaient arrêtés.

L'exemple d'un tel algorithme est donné dans la section 1.1.2.

Définition 1.5 *Algorithmes anytime à contrat*

Un contrat, temps de calcul fixé avant l'exécution leur est alloué. S'ils sont interrompus avant la fin de leur contrat, il se peut qu'ils ne fournissent aucun résultat utile.

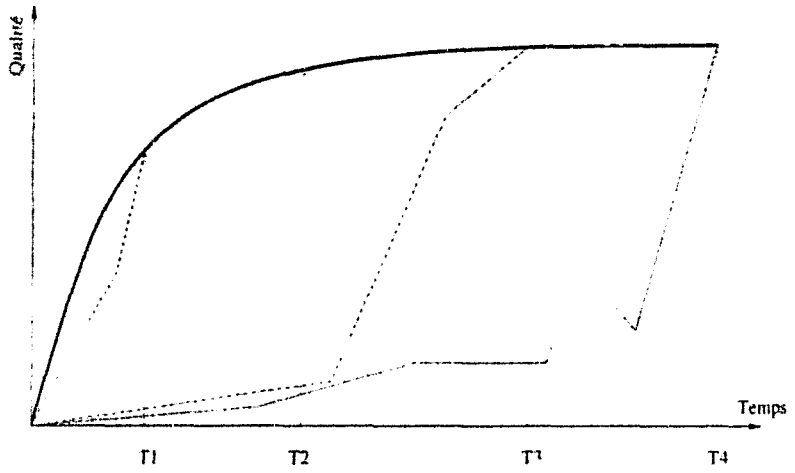


FIG. 1.6 – *Profil de performance d'un algorithme à contrat*

Un algorithme à contrat tient compte du temps de calcul qui lui est alloué pour choisir une stratégie. Sur la figure 1.6, nous avons représenté les différents parcours que la qualité du résultat peut emprunter selon le contrat alloué. Un algorithme à contrat "choisira" sa stratégie de résolution. Les résultats intermédiaires seront quelquefois de moins bonne qualité, mais finalement meilleurs à l'expiration d'un contrat plus long. Cela signifie que seule compte la fin du contrat et que, contrairement à l'algorithme anytime interruptible, l'algorithme anytime à contrat n'est pas obligé de toujours augmenter la qualité entre l'instant de sa mise en route et la fin du contrat.

Illustrons la différence entre le comportement d'un algorithme à contrat et un algorithme interruptible sur une situation concrète. Imaginons que nous avons un problème de d'évacuation d'une île à résoudre à cause de la menace d'erruption d'un volcan. Si on ne sait pas de combien de temps on dispose avant que le volcan devienne dangereux, il faut hiérarchiser l'évacuation en donnant des ordres de priorité. Par exemple, évacuer les femmes et les enfants en urgence, puis les hommes, enfin les forces de l'ordre. La première vague d'évacuation commence alors dès que possible, et les autres suivent les unes après les autres. On est dans le cas d'un algorithme interruptible. Par contre, si on sait exactement quand le volcan sera dangereux, il est possible d'agir autrement: par exemple de construire des radeaux et des embarcations permettant d'évacuer plus de personnes et de biens au moment final. Il est certain que si cette préparation devait être interrompue, elle se solderait par une catastrophe puisque personne n'aurait été évacué (une qualité nulle). Mais il est possible que cette préparation donne un meilleur résultat final que la préparation interruptible pour un même temps de préparation. Nous sommes alors dans le cas à contrat.

Un exemple d'algorithme à contrat est l'algorithme RTA* de KORF [Kor90], inspiré de l'algorithme A* [HNR68]. Son principe est celui d'un algorithme de recherche du "meilleur d'abord" ou *best first search*. Son principal intérêt est que le temps de calcul est borné par une constante et qu'en modifiant l'horizon de recherche, on modifie cette constante. Dans l'article [Kor90] où il présente RTA*, KORF n'évoque pas la notion d'algorithme à contrat.

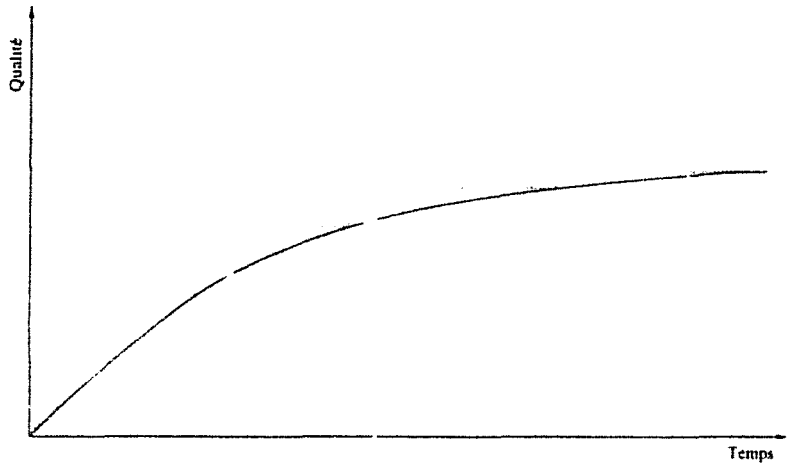


FIG. 1.7 - *Transformer un algorithme à contrat en un algorithme interruptible*

Par contre, il parle de qualité de la solution et garantit que cette qualité augmente avec l'horizon de recherche. Ainsi, en fixant les horizons de recherche, on se fixe un contrat et la qualité augmente avec la longueur des contrats.

Transformation d'un type d'algorithme anytime en un autre

ZILBERSTEIN et RUSSEL [ZR96] ont proposé une transformation d'un algorithme anytime à contrat en un algorithme anytime interruptible.

Théorème 1.1 *De l'algorithme interruptible à l'algorithme à contrat*

Pour tout algorithme interruptible A, et pour une donnée fixée, il existe B un algorithme à contrat tel que $Q_B(t) \leq Q_A(t)$. Il suffit d'insérer dans A un arrêt après une durée t, le contrat.

En effet, au lieu de l'interrompre à tout moment, on l'interrompt au bout du contrat de temps alloué. Par contre, ce ne sera pas forcément le meilleur algorithme à contrat que l'on aurait pu construire pour répondre au problème posé. Le fait de connaître le temps dont on dispose peut en effet permettre, pour certains problèmes, de construire des algorithmes à contrat plus performants que les algorithmes interruptibles (voir plus haut notre exemple d'évacuation).

La transformation d'un algorithme à contrat en un algorithme interruptible n'est pas évidente. Le théorème qui suit indique comment procéder.

Théorème 1.2 *De l'algorithme à contrat à l'algorithme interruptible [ZR96]*

Pour tout algorithme à contrat A, on peut construire un algorithme interruptible B tel que, pour une entrée donnée, $Q_A(t) \leq Q_B(4t)$.

Schéma de la preuve :

La transformation consiste à lancer l'algorithme à contrat de façon répétée, en doublant les contrats à chaque itération (figure 1.7). Si une interruption survient, le résultat de la dernière itération est la réponse. On peut alors prouver que, pour obtenir une qualité équivalente avec l'algorithme interruptible ainsi construit, il ne faut pas plus de quatre fois le temps nécessaire à l'algorithme à contrat initial. Ce coefficient 4 a été prouvé optimal dans [ZCC99].

Notons que le critère d'optimisation dans la transformation proposée ici est la minimisation du facteur multiplicatif entre le temps nécessaire à un algorithme à contrat et celui nécessaire à l'algorithme interruptible correspondant pour obtenir une qualité fixée. On pourrait envisager d'autres critères d'optimisation.

1.1.2 Construction et utilisation d'un algorithme anytime

Après avoir exposé les caractéristiques essentielles des algorithmes anytime, nous allons illustrer ces notions par un exemple de construction d'un algorithme anytime interruptible à partir d'un algorithme de résolution du problème du voyageur de commerce.

1.1.2.1 Construction de l'algorithme

D'après la description que nous venons de faire, pour obtenir un algorithme anytime, il faut extraire ou construire :

- *un pas élémentaire d'amélioration itérative* : C'est une caractéristique que nous pouvons retrouver dans beaucoup d'algorithmes qui ne demandent alors aucune modification.
- *un évaluateur du résultat ou critère de qualité* : C'est une des difficultés principales de la construction des algorithmes anytime que de trouver un critère de qualité du résultat intermédiaire calculé. Chaque nouveau problème, et chaque nouvel algorithme demande un critère différent.
- *un profil de performance* : Nous utiliserons une des trois méthodes décrites dans le paragraphe 1.1.1.2 pour construire le profil de performance, et plus particulièrement la méthode statistique. Une fois le critère de qualité et les entrées choisies, il ne reste plus qu'à construire la carte de la qualité et en déduire un profil de performance.

GRASS et ZILBERSTEIN [GZ96] considèrent que les fonctions dites de support, c'est-à-dire la fonction d'évaluation de la qualité et la fonction de génération des instances (des entrées de l'algorithme choisi) sont du ressort du concepteur de l'algorithme. Cela semble naturel tant ces deux fonctions dépendent de l'application. Pour l'implémentation de l'algorithme anytime, ils préconisent également de fournir un pointeur sur la structure

de donnée sur laquelle travaille l'algorithme afin de pouvoir récupérer le résultat à tout moment, ainsi qu'une information sur l'avancée de l'algorithme.

Appliquons ces principes à un algorithme de résolution du problème du voyageur de commerce.

1.1.2.2 Le voyageur de commerce (TSP)

Le problème du voyageur de commerce est de construire un cycle Hamiltonien (une tournée) de coût minimum à partir des sommets donnés. Nous travaillons sur la base d'un algorithme présenté dans [Zil93]. Le coût associé à une arête du graphe est ici la distance euclidienne et le graphe est supposé totalement connecté. Le cycle Hamiltonien de coût minimum est donc le chemin (revenant au point de départ) le plus court au sens de la distance usuelle sur un plan. Nous appellerons ce chemin une tournée. Le principe de l'algorithme est de choisir deux arêtes au hasard dans la tournée calculée jusqu'au point considéré et d'échanger ces deux arêtes si la tournée formée avec les deux arêtes échangées est plus courte que l'ancienne tournée. C'est un algorithme stochastique qui ne garantit pas de trouver la tournée optimale (la plus petite).

Etape 1 : La fonction itérative existe déjà dans cet algorithme, qui se décompose comme suit:

- Initialisation : choisir une tournée au hasard (une permutation des sommets).
- Corps : échanger deux arêtes du graphe au hasard, de telle sorte que le coût du nouveau trajet soit inférieur à celui de l'ancien :

$$Coût(Tournee(n+1)) \leq Coût(Tournee(n))$$

Etape 2 : Le critère de qualité initialement choisi par ZILBERSTEIN est le rapport de la longueur de la tournée optimale sur la longueur de la tournée courante. Remarquons que le problème du voyageur de commerce est un des nombreux cas où il n'existe pas d'algorithme polynomial permettant d'avoir une approximation de la valeur du coût de la solution optimale d'aussi près que l'on veut. Ce qui rend un critère de qualité utilisant le coût de l'optimal impossible à calculer. Ainsi, empiriquement on pourra prendre le critère suivant :

$$q(t) = \frac{longueur(Tournee_{init})}{longueur(Tournee_{courrante})} \tag{1.1}$$

Ce critère peut encore poser des problèmes car la qualité est fortement sensible à la tournée initiale. Nous reviendrons sur cette question dans le chapitre 3.

Etape 3 : Comme la stratégie de choix des nouvelles arêtes à inverser le hasard, il est impossible d'analyser l'algorithme afin d'en déduire le profil de performance. Il faut donc utiliser la méthode statistique. Les paramètres G et $iter$ dans l'algorithme 1.1 représentent respectivement le graphe de départ et le nombre d'itérations que nous voulons faire faire à l'algorithme.

ALG 1.1 Un exemple d'algorithme anytime pour le TSP : *Random improvement Tour*

programme *TSP-anytime*
Entrée: G un graphe
iter entier
Sortie: *Tournée* une permutation d'entiers
début
Tournée ← *Tournée_initiale(G)*
coût ← *Coût(Tournée)*
Enregistre_résultat(Tournée)
pour i **de** 1 **à** *iter* **faire**
a_1 ← *Choix_arete_hasard(Tournée)*
a_2 ← *Choix_arete_hasard(Tournée)*
d ← *Coût(Tour)* - *Coût(Echange(Tournée,* a_1, a_2*))*
si $d > 0$ **alors**
Tournée ← *Echange(Tournée,* a_1, a_2*)*
coût ← *coût* - d
Enregistre_résultat(Tournée)
fin si
fin pour
Signal(Terminaison)
fin

avec :
- *Tournée_initiale* : fonction qui établit une tournée initiale au hasard à partir d'un graphe G, complètement connecté.
- *Coût*: fonction de calcul du coût d'une tournée (voir équation 1.1).
- *Enregistre_résultat*: fonction d'enregistrement du résultat courant dans une zone de mémoire accessible à un programme extérieur.
- *Choix_arête_hasard* : fonction de choix au hasard d'une arête de la tournée courante.
- *Echange* : fonction d'échange de deux arêtes.

Cet algorithme calcule donc *iter* nouvelles tournées. Le limiter par le nombre d'itérations *iter* est nécessaire puisqu'aucun critère d'arrêt basé sur l'évaluation du coût de la tournée courante par rapport à la tournée optimale n'est définit. Dans le cas contraire, il pourrait "chercher" indéfiniment de nouvelles améliorations. L'algorithme est interruptible puisqu'il sauvegarde à tout moment un résultat intermédiaire récupérable. Il peut être géré par un autre programme, dit de contrôle, qui lui envoie un signal d'interruption ou le fait redémarrer après une interruption. Les carte de qualité et profil de performance probabiliste résultant de cet algorithme sont exposés sur les figures 1.3 et 1.4, respectivement.

1.1.3 Conclusion

Nous venons de présenter les bases de l'algorithmique anytime. Nous avons vu qu'un algorithme anytime offrait la possibilité de gérer un compromis entre le temps de calcul et la qualité du résultat. Ceci est rendu possible par l'introduction de la notion de résultat intermédiaire et de qualité.

Pour gérer ce compromis, il reste à définir des stratégies d'allocation de temps à ces algorithmes et leur ordonnancement. C'est ce que nous présenterons dans le reste du chapitre, en commençant par une approche particulière permettant de composer plusieurs algorithmes anytime pour en obtenir un plus complexe. Puis nous verrons quelques approches de contrôle des algorithmes anytime et de quelques unes de leurs alternatives.

1.2 Composition optimale d'algorithmes anytime

Nous abordons dans cette section la problématique de la composition des algorithmes anytime. La composition de modules anytime permet de décomposer une application complexe en plusieurs algorithmes anytime élementaires. Le but est de simplifier et accélérer ainsi la conception d'applications anytime. Nous présentons donc ici les résultats essentiels de ZILBERSTEIN [Zil93] et RUSSEL [ZR96], auxquels nous devons cette approche permettant d'obtenir un algorithme à contrat en composant des algorithmes anytime.

1.2.1 Définition du problème de compilation

Le terme de compilation est utilisé par ZILBERSTEIN et RUSSEL pour faire référence à une transformation de l'algorithme global en un algorithme anytime composé, associé à une information sur l'allocation des différents composants utilisables par un *moniteur*. Ce dernier est soit du type passif, si l'optimisation et l'ordonnancement sont faits avant toute exécution, soit du type actif si ces ordonnancement et optimisation peuvent être remis en cause lors de l'exécution, pour des raisons de réactivité à l'état de l'environnement. Le principe de la compilation est résumé sur la figure 1.8.

ZILBERSTEIN et RUSSEL traitent de la compilation des algorithmes caractérisés par un profil de performance conditionnel (CPP) monotone par rapport à la qualité de l'entrée et un contrôle (moniteur) passif.

Définition 1.6 *Monotonicité par rapport à la qualité de l'entrée*

Un CPP pour un algorithme A vérifie la propriété de monotonicité par rapport à la qualité de l'entrée si, pour p et q des qualités d'entrée de A, on a :

$$\forall p, \forall q, p < q \Rightarrow CPP_{A,p} \preceq CPP_{A,q}$$

On dit dans ce cas que le profil de performance conditionnel pour A et q domine le profil de performance conditionnel pour A et p. C'est-à-dire que la qualité en sortie croît avec la qualité en entrée.

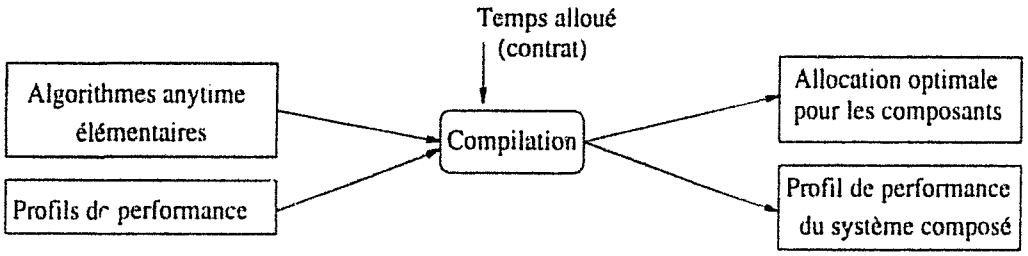


FIG. 1.8 – *Principe de la composition d'algorithmes anytime*

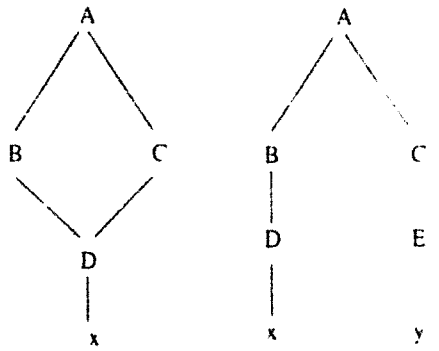
1.2.1.1 Complexité

Faisons une analyse plus formelle d'une classe générale de problèmes de compilation générée par la composition fonctionnelle d'algorithmes anytime. Les expressions fonctionnelles sont créées par une fonction dont les paramètres sont soit des variables d'entree, soit d'autres expressions fonctionnelles.

Définition 1.7 *Expression fonctionnelle*

Une expression fonctionnelle sur F, un ensemble de fonctions anytime, d'entrée I, est :

- *une variable d'entrée $i_j \in I$, ou*
- *une expression $f(g_1,g_2,...,g_n)$ où $f \in F$ et g_i est une expression fonctionnelle.*



(a) DAG (b) arbre

FIG. 1.9 – *Représentation d'une expression fonctionnelle*

Dans le cas des algorithmes à contrat, la compilation consiste à trouver pour chaque allocation totale t, la meilleure façon d'ordonner les composants pour optimiser la qualité du résultat de l'expression fonctionnelle. Notons qu'une façon générale de représenter les expressions fonctionnelles est le DAG (*Directed Acyclic Graph* ou Graphe Acyclique Orienté) comme sur la figure 1.9(a) pour l'expression $A(B(D(x)),C(D(x)))$ où $D(x)$ est

une sous-expression répétée. Mais quand une expression fonctionnelle n'a pas de sous-expression répétée alors sa représentation est sous la forme d'un arbre (figure 1.9(b) pour l'expression fonctionnelle $A(B(D(x)),C(E(y)))$.

Nous allons maintenant pouvoir donner les résultats de complexité de la compilation d'expressions fonctionnelles. Ce problème est défini comme un problème d'optimisation, mais il est plus commode de résoudre la propriété correspondante (problème de décision). Il y a équivalence du point de vue de la complexité algorithmique entre une propriété et son problème [GJ79] [Del96].

Définition 1.8 *Le problème GCFE (Global Compilation of Functionnal Expression)*

Etant donnés :

- *une expression fonctionnelle e,*
- *le profil de performance des modules anytime élémentaires entrant dans la composition,*
- *et l'allocation totale de temps B,*

Pour une constante K donnée, existe-t-il un ordonnancement des composants qui résulte en une qualité supérieure ou égale à K?

Théorème 1.3 *Le problème GCFE est NP-complet au sens fort.*

ZILBERSTEIN en fait la preuve en réduisant le problème de sac à dos partiellement ordonné [GJ79] à *GCFE*. Un problème NP-complet au sens fort [GJ79], s'il n'existe pas d'algorithme pseudo-polynomial pour le résoudre.

Dans le cas particulier d'une version réduite du problème *GCFE*, c'est-à-dire le cas où l'expression fonctionnelle ne comporte pas de sous-expression répétée, le problème conserve une combinatoire complexe.

Théorème 1.4 *Le problème GCFE structuré en arbre est NP-complet.*

La preuve de la NP-complétude est faite en réduisant le problème du sac à dos, un problème NP-complet. Le problème est le même que précédemment mais il existe une relation d'ordre total dans l'ensemble des paquets [GJ79].

Pour distinguer la différence de complexité entre les deux cas, il faut imaginer le nombre de comparaisons supplémentaires nécessaires avec un ordre partiel. De même, une représentation avec un DAG implique que pour une même sous-expression, il n'y a pas forcément la même allocation de temps, ce qui complique le problème.

Toutefois, ces complexités ne sont pas satisfaisantes. C'est pourquoi nous présentons dans la suite une méthode qui rend la compilation praticable, sous certaines conditions.

1.2.1.2 La compilation locale

L'idée de la compilation locale est de remplacer un seul problème d'optimisation complexe par un ensemble de problèmes d'optimisation simples dont le nombre croît linéairement avec la taille du programme à compiler. Si les problèmes d'optimisation locaux peuvent être résolus de façon polynomiale, alors le calcul total devient lui aussi polynomial. Il faudra toutefois s'assurer que cette méthode est optimale, c'est-à-dire qu'elle donne le même résultat que la compilation globale.

Définition 1.9 *Compilation locale*

La compilation locale est l'optimisation de la qualité de sortie d'une construction algorithmique en ne considérant que les profils de performance des ses sous-composants immédiats.

Théorème 1.5 *Optimalité de la compilation locale*

Soit e une expression fonctionnelle de profondeur arbitraire n, dont les CPP satisfont à l'hypothèse de monotonicité de la qualité d'entrée, alors pour toute entrée et pour une allocation totale de temps t: $Q_e^L(t) = Q_e^G(t)$ où $Q_e^L(t)$ est le résultat de la compilation locale (L), et $Q_e^G(t)$ est celui de la compilation locale.

Nous savons, grâce au théorème 1.5, que l'optimisation locale donne un résultat optimal sur un problème global. Il faut alors poser les conditions sous lesquelles nous pouvons réduire la complexité du problème de compilation. Cette réduction est définie comme suit :

Théorème 1.6 *Réduction de la complexité de la compilation locale*

Posons n la taille de l'expression fonctionnelle et utilisons des profils de performance discrets avec une allocation maximale de temps t. Le problème GCFE structuré en arbre est polynomial en nt sous les conditions de monotonicité de la qualité des entrées et de facteur de branchement borné.

La monotonicité garantit l'optimalité de la qualité du résultat. Il suffit alors de répéter $O(n)$ fois la compilation locale, le nombre de noeuds dans l'arbre de composition. Le facteur de branchement borné par k entier permet de dire que la compilation globale est de complexité $O(nt^k)$.

Il n'y a pas de contradiction avec le fait que le problème du sac à dos est NP-complet, puisque ce théorème prouve que le problème GCFE est polynomial en nt. Ce problème n'est pas polynomial en fonction de la taille de ses entrées, mais en fonction de leur valeur.

En termes de besoins en espace, la compilation locale a la même complexité que la compilation globale, car elle nécessite seulement $O(n)$ profils de performance séparés supplémentaires. L'espace occupé est donc seulement multiplié par un facteur constant, pour une instance du problème donnée.

ZILBERSTEIN [Zil93] propose d'étendre ces résultats à des familles de composition plus larges. D'après l'optimalité de la compilation locale, il faut que l'opérateur de composition respecte deux règles :

- qu'il produise des résultats dont la qualité dépend de la qualité de ses entrées et du temps d'allocation qu'il lui est donné.
- que les CPP de l'opérateur vérifient la monotonicité. Beaucoup d'opérateurs utiles, comme les opérateurs conditionnels ou booléens, ou encore les boucles, vérifient ces deux conditions. Ces cas sont abordés dans [Zil93].

1.2.1.3 Heuristiques

La compilation locale ne donne pas de très bons résultats quand elle est appliquée à des expressions fonctionnelles avec des sous-expressions répétées. Le problème est qu'elle alloue du temps à tous les noeuds de l'arbre, et les sous-expressions répétées sont alors traitées plusieurs fois différemment. Afin de remédier à ce problème, ZILBERSTEIN a proposé dans sa thèse [Zil93] trois méthodes d'allocation qui traitent les expressions fonctionnelles en général. Ces trois méthodes ont été développées en utilisant la représentation discrète en tableau des profils de performance. La complexité est le cout en temps de calculer l'allocation optimale des composants pour une qualité d'entrée et un temps total quelconques. Pour obtenir une allocation de complexité raisonnable, il a donc fallu faire des concessions sur l'un ou l'autre des paramètres de la compilation, comme le facteur de branchement, le nombre de sous-expressions répétées, mais aussi sur l'optimalité parfois.

1.2.2 Conclusion sur la composition de modules anytime

Les résultats de ZILBERSTEIN et RUSSEL présentés ici nous montrent que la composition modulaire d'algorithmes anytime peut être implémentée. En particulier, nous savons que :

- le profil de performance d'un système composé, sous la forme d'un algorithme à contrat, est calculable automatiquement,
- le système résultant est transformable en algorithme interruptible au prix d'une pénalité multiplicative constante.

Le problème de la compilation est ainsi formalisé et résolu dans le cas d'expressions fonctionnelles, hors ligne la plupart du temps grâce à la formalisation des CPP. Les travaux actuels sont dirigés vers l'évaluation de cette approche sur des essais "grandeur nature", l'étude d'autres structures de composition, l'extension des algorithmes anytime à l'action et la mesure anytime et enfin le développement d'un environnement de programmation anytime amorcé dans [GZ96].

1.3 Gestion des algorithmes anytime

L'algorithmique anytime vise un compromis entre le temps d'exécution et la qualité du résultat de tels algorithmes. A cette fin, des études ont défini des techniques de gestion (le terme de *monitoring* est parfois utilisé) des algorithmes anytime. Et c'est l'objet de cette section de présenter quelques unes de ces principales approches de gestion.

1.3.1 Généralités

Par gestion nous entendons le choix des algorithmes à exécuter ainsi que la durée d'exécution et les instants de déclenchement. Il y a donc en fait deux parties dans ce que nous devrions appeler la gestion des algorithmes anytime : une partie de planification (le choix des algorithmes) et une autre d'ordonnancement (gestion du temps). Le but est finalement d'optimiser le résultat en terme de qualité tout en respectant les dates limites à l'aide d'un ou de plusieurs algorithmes anytime. Nous verrons que les approches présentées diffèrent surtout par la structure des tâches utilisées, les relations entre ces tâches, et la façon de gérer les dates limites.

1.3.2 Approche de Boddy et Dean : Deliberation Scheduling

BODDY et DEAN proposent une méthode, le *Deliberation Scheduling* (DS) [BD94], qui est une première approche pour la combinaison de plusieurs algorithmes anytime. Ils utilisent pour cela des profils de performances approchés, du type continu linéaire par morceaux avec une pente décroissante. Cette dernière propriété est appelé le "*diminishing returns*", ce qui se comprend bien intuitivement puisque le gain attendu d'un morceau à l'autre est de moins en moins grand avec le temps. Le diminishing returns est une hypothèse essentielle pour l'algorithme d'ordonnancement proposé et pour son optimalité. Le modèle d'algorithme utilisé ici, à part son type de profil de performance simplifié, est celui des algorithmes anytime interruptibles originels [DB88].

Quand on parle de combinaison d'algorithmes dans le cas du *Deliberation Scheduling*, il ne s'agit pas de composition d'algorithme comme dans l'approche de ZILBERSTEIN et RUSSEL présentée dans la section 1.2. DEAN et BODDY considèrent en fait un système devant répondre à un ensemble de conditions ou problèmes. A chacune de ces conditions correspond un algorithme et un profil de performance associé. La réponse à chacune des conditions est supposée totalement indépendante des réponses aux autres conditions. Il s'agit donc du partage d'une ressource de temps de calcul entre plusieurs algorithmes indépendants. L'un n'exploite pas les résultats de l'autre. Ainsi, il n'y a pas de relation de précédence entre ces algorithmes autre que celle imposée par les dates limites distinctes qui font que l'un doit se terminer impérativement avant l'autre.

Hors cette contrainte de date limite, le partage du temps est entièrement libre. Etant donné un profil de performance pour chacun de ces algorithmes, la question du Deliberation Scheduling est de trouver les allocations explicites de temps que l'on doit faire pour

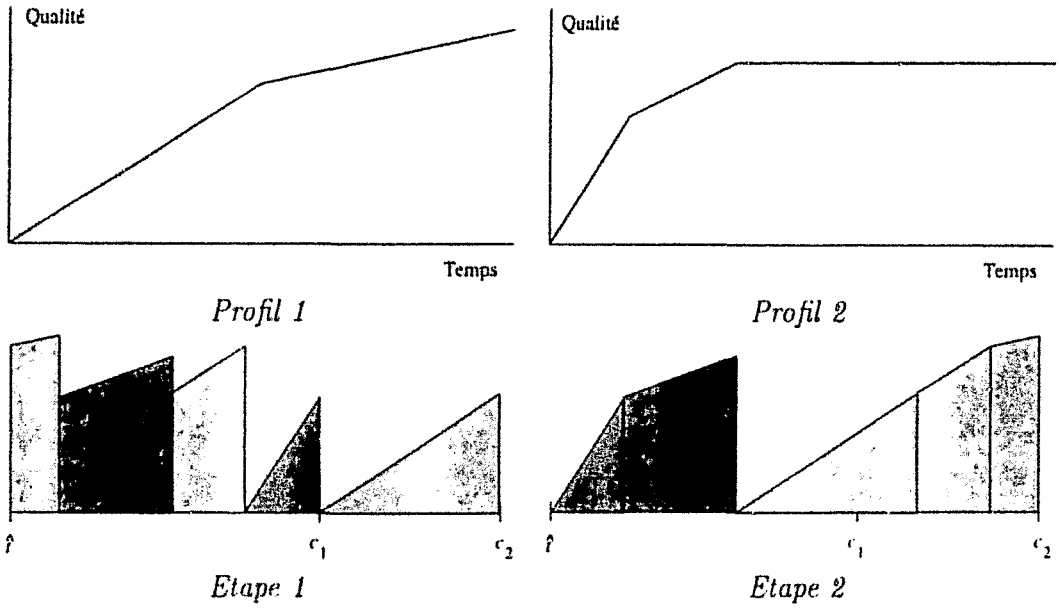


FIG. 1.10 - *Exemple de fonctionnement de l'algorithme DS*

chacun des algorithmes afin qu'ils produisent les résultats de la meilleure qualité possible avant leurs dates limites propres. Pour cette optimisation de la qualité, l'algorithme DS se base sur l'effet attendu de ces allocations sur les performances du système définies par les profils de performance. Le critère d'optimisation choisi par DEVY et BODDY ici est de maximiser la somme des qualités des résultats fourni par chacun des algorithmes. Ceci suppose que les qualités ont des valeurs adéquates reflétant leur importance vis à vis des autres qualités. Si on suppose par exemple qu'aucune qualité n'a plus d'importance qu'une autre, la qualité pour chacun des algorithmes sera normalisée. Dans un premier temps, les dates limites sont supposées être connues précisément.

Le principe de l'algorithme de Deliberation Scheduling (DS) est de partir de la dernière date limite (la plus tardive) et de travailler à rebours. A chaque itération de la boucle principale, le programme alloue un certain intervalle de temps de calcul à un algorithme, dont la date limite arrive plus tard que l'instant auquel on se trouve. Le programme DS consiste en fait en trois boucles itératives : la première initialise les variables d'allocations, la deuxième détermine combien de temps allouer à chacun des algorithmes, et la troisième détermine à quels instants les algorithmes doivent être lancés. Nous décrivons la stratégie de cet algorithme DS sur la figure 1.10.

Dans cet exemple (figure 1.10), on considère deux algorithmes dont les profils de performance sont Profil 1 et Profil 2, linéaires par morceaux par hypothèse. Les deux dates limites respectives sont c_1 et c_2, avec $c_1 < c_2$. Comme nous le disions plus haut, hormis la phase d'initialisation, il y a donc deux étapes dans cet algorithme DS : l'allocation de temps à chaque algorithme et l'ordonnacement des exécutions.

La première étape commence donc par affecter à une date de référence nommée t (temps courant) la date limite la plus tardive (ici $t = last(+\infty) = c_2$). L'algorithme procède ensuite à rebours en partant de cette date t jusqu'à ce que t soit égal à $\hat{t}$, la date

de départ du calcul. A chaque itération, DS calcule un intervalle de temps Δ à allouer, égal au minimum de trois paramètres: la différence $t - \widehat{t}$ (le temps total restant à allouer), la différence $t - last(t)$ (intervalle entre la date t et la date limite la plus tardive avant t, ici égal à $t - c_1$) et enfin le temps minimum à allouer pour qu'un profil change de pente. Ensuite DS cherche l'algorithme avec le plus fort gain pour lui allouer le temps Δ. Cet algorithme est choisi parmi ceux dont la date limite est supérieure ou égale à t (ici il n'y a que c_2 à la première itération). Le plus grand est entendu pour un algorithme à l'endroit du profil correspondant au temps déjà alloué pendant les itérations précédentes. Ensuite le temps t est bien sûr décrémenté du temps alloué, soit $t - \Delta$.

Un résultat important de BODDY et DEAN sur cet algorithme DS est qu'il est optimal pour ce types de profils de performance et de dates limites. Il est important pour obtenir cette optimalité que les profils de performance respectent le *diminishing returns*. Si le gain n'est pas garanti de diminuer avec le temps, c'est l'étape de choix du profil avec le plus fort gain qui est mise en défaut. En effet, si le *diminishing returns* est respecté, DS ne restreint en aucun cas ses choix futurs étant donné que les intervalles suivants sont garantis d'avoir un gain égal ou inférieur. Quand un profil est choisi pour son gain maximal sur un intevalle donné, il est certain que c'est le meilleur choix, car à l'avenir, avec le *diminishing returns*, un intervalle ne peut pas avoir de gain plus grand et remettre en cause le choix effectué. Un exemple de situation défavorable où les profils de perfomances ne respectent pas le *diminishing returns* est donné dans l'article [BD94].

BODDY et DEAN proposent d'ailleurs une extension où les dates limites sont incertaines, mais l'algorithme, inspiré du premier, n'est plus optimal. Nous ne présenterons pas cet algorithme. L'application principale de cette approche est la planification temporelle (*Time-dependant Planning* [BD94]).

1.3.3 Approche de Hansen et Zilberstein

Cette approche utilise les algorithmes anytime tels que nous les avons décrits dans la première partie de ce chapitre. Le problème qui se pose ici est un problème de contrôle des algorithmes anytime:

> "faire un compromis optimal entre temps et qualité requiert de déterminer combien de temps doit calculer l'algorithme, et quand l'arrêter et agir sur la solution courante disponible"[HZ96]

Le principe est d'obtenir la qualité la plus élevée dans un temps minimal. Pour cela, HANSEN et ZILBERSTEIN [HZ96] [Zil93] introduisent la notion de profil de performance probabiliste (*probabilistic performance profile* ou *PPP*) qui décrit une distribution de probabilité de la qualité et du temps. A chaque point de la courbe $PPP(q|t)$ obtenue, ou à chaque case du tableau dans la représentation discrète tabulaire, on associe une probabilité d'obtenir une certaine qualité q après un temps de calcul t. En complément de ce modèle de profil de performance, il est nécessaire d'avoir un modèle d'utilité dépendante du temps d'une solution.

A partir de ces hypothèses, le temps de calcul optimal d'un algorithme anytime peut être calculé avant sont exécution, sans contrôle (*monitoring*) lors de son exécution, en résolvant la formule suivante :

$$arg \max_t \sum_q PPP(q|t)U(q,t)$$

Dans certains cas, l'utilité dépend aussi de l'environnement et notamment de son état quand la solution de l'algorithme a influencé ce dernier. HANSEN et ZILBERSTEIN essaient alors de modéliser cette incertitude sur l'état de l'environnement en introduisant $P(s|t)$ une probabilité de se trouver dans un état s après un temps t. On a alors $U(q,s,t)$ l'utilité d'arrêter l'algorithme anytime après un temps t avec une solution de qualité q obtenue dans un état s. L'équation définissant le temps de calcul optimal s'écrit alors :

$$arg \max_t \sum_q \sum_s PPP(q|t)P(s|t)U(q,s,t)$$

La méthode que nous venons de décrire est un moyen d'obtenir le temps de calcul optimal au sens de la meilleure qualité/utilité possible quand on veut faire le calcul avant de lancer l'algorithme. HANSEN et ZILBERSTEIN proposent également de calculer ce temps optimal mais cette fois-ci pendant l'exécution de l'algorithme. Pour cela, ils expliquent comment étendre le cadre qui vient d'être décrit en précisant quelle forme doivent prendre les informations de performance de l'algorithme manipulé et en proposant un modèle de "moniteur" d'exécution capable d'améliorer un temps d'arrêt de l'algorithme avec l'information collectée en temps réel (pris au sens *run-time*, c'est-à-dire pendant l'exécution). Nous nous contenterons d'expliquer les grands principes de ce modèle décrits dans [HZ96] et [Zil93].

Pour tenir compte des informations collectées pendant l'exécution, les auteurs introduisent un *profil de performance dynamique* qui permet de prédire les améliorations que l'on peut espérer sur une solution obtenue dans un état s et après un temps de calcul t si on laisse l'algorithme poursuivre pendant un temps supplémentaire Δt. Soit $P(q|f,t,\Delta t)$ la probabilité d'obtenir une solution de qualité q en relançant un algorithme anytime (interruptible, sinon, avec un algorithme à contrat, il faudrait le relancer de zéro) pour un temps de calcul supplémentaire Δt quand la solution courante obtenu après un temps t possède une propriété f. Cette propriété peut être la qualité, ou toute autre caractéristique de la solution courante.

HANSEN et ZILBERSTEIN insistent d'ailleurs sur les difficultés qui peuvent exister à essayer d'obtenir un tel profil de performance, sachant déjà que pour certains algorithmes il peut être impossible d'évaluer la qualité en cours d'exécution. Cependant ils supposent que beaucoup d'algorithmes possèdent cette propriété de prédiction. D'autres améliorations concernant la capacité de prédiction peuvent être imaginées, comme l'estimation du comportement futur de l'algorithme à partir de la trajectoire déjà suivie.

Enfin, il est nécessaire de tenir compte des changements dans l'environnement. Les mêmes extensions que pour les profils de performance sont possibles pour le modèle d'occurence des états de l'environnement. Par exemple, on peut noter $P(s'|s,\Delta t)$ la probabilité d'arriver dans un état s' après un temps Δt sachant que l'on est dans un état s.

Tous ces modèles sont construits pour déterminer l'instant optimal de l'arrêt de l'algorithme anytime interruptible. C'est donc à partir de ces définitions que HANSEN et ZILBERSTEIN proposent tout d'abord une stratégie de contrôle myope : tant que la qualité du résultat est croissante (représentée chez eux par l'EVC, *Expected Value of Computation*, c'est-à-dire la différence de valeur entre le résultat espéré et le résultat courant) après un temps de calcul supplémentaire Δt, alors on continue le calcul. La stratégie est dite myope car Δt représente un pas élémentaire de temps. Ils prouvent l'optimalité de cette stratégie quand la valeur espérée du calcul (EVC) est négative à $(t + \Delta t, q + \Delta q)$ si elle déjà négative en (t,q). C'est-à-dire quand l'EVC possède un maximum. Un complément d'étude, que nous n'exposons pas ici, propose de tenir compte du temps de réponse du contrôle exercé sur l'algorithme anytime. Dans la solution proposée ce temps de contrôle représente un coup fixe, ce qui semble raisonnable.

En résumé, cette approche apporte une description d'un modèle de contrôle dynamique, c'est-à-dire pendant l'exécution, sur les algorithmes anytime interruptibles et une stratégie pour "régler" la fréquence du contrôle afin de réduire les temps de réponse pendant l'exécution de l'algorithme. Quelques pistes de discussion sur les complications émanentes à la réalisation de profil de performance et à l'évaluation de la qualité d'un résultat final ou intermédiaire sont données.

1.3.4 Le raisonnement progressif

Le raisonnement progressif, utilisé par MOUADDIB [Mou93], est une approche qui tire son principal intérêt de la difficulté pour certaines applications de construire des profils de performances. C'est le cas, par exemple, du raisonnement à base de connaissance [MZ95] (*Knowledge-based reasoning*), où les techniques actuelles ont du mal à s'adapter à l'algorithmique anytime, en raison de la grande variabilité de la performance en fonction du temps selon les instances mises en entrée de l'algorithme.

Afin d'éviter de construire un profil de performance peu prédictif, le raisonnement progressif utilise plusieurs niveaux de traitement afin de transformer graduellement une solution imprécise en solution précise. Le fait de structurer les entrées, en accordant un poids à chacune d'elles en fonction de son importance, et la connaissance est un facteur important de l'espace de recherche. En effet, le nombre de données et de connaissances à chaque niveau est restreint à un sous-ensemble sélectionné par un critère dit de *granularité*. En limitant l'espace de recherche, on a beaucoup plus de chance de pouvoir prédire le comportement de l'algorithme en terme de performance. Notons que la hiérarchisation de la connaissance convient bien aux applications telles que la planification hiérarchique.

C'est donc par la structure des tâches que le raisonnement progressif se distingue. Comme nous le disions plus haut, il n'est pas nécessaire d'avoir un profil de performance, mais juste un ensemble de qualités associées aux niveaux d'exécution d'une tâche. Chaque résultat fourni à un niveau donné est exploitable et représente une vue intermédiaire (approximée) de la solution optimale. La transition entre les niveaux est faite en ajoutant des données et des traitements plus précis qu'auparavant. Il est fait autant de transitions que nécessaire pour atteindre la qualité optimale ou le temps limite fixé *a priori*. Il peut

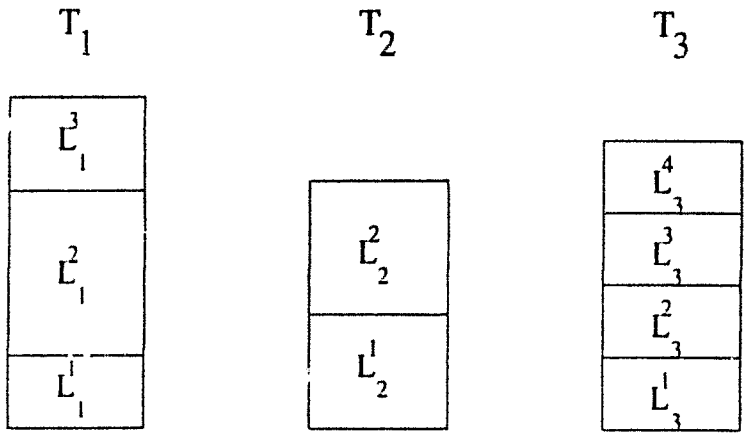


FIG. 1.11 - *Structure de tâche pour le raisonnement progressif*

exister de surcroît une incertitude sur le temps nécessaire à l'exécution de chaque niveau ainsi que sur les dates limites qui peuvent varier dynamiquement au cours de l'exécution.

Nous n'avons considéré pour le moment qu'une seule tâche pour le raisonnement progressif. MOUADDIB et ZILBERSTEIN [MZ98] proposent de gérer un problème décomposé en plusieurs unités indépendantes comme illustré sur la figure 1.11. De plus, ils proposent que la liste des unités soit mise à jour dynamiquement.

La structure de tâche est la suivante :

- un ensemble de tâches $S = \{T_1, T_2, ..., T_n\}$ où $T_i = \{H_i, D_i\}$
  - H_i, D_i sont respectivement la hiérarchie et la date limite de la tâche i.
  - $H_i = \{L_i^1, L_i^2, ..., L_i^P\}$ avec L_i^j le j ème niveau de la tache i considérée.
  - chaque niveau est associé à une distribution de probabilité discrète du temps d'exécution et à une qualité.

Etant donné un ensemble S, le problème de l'ordonnancement des tâches est de construire un ordre d'exécution de ces tâches et de chacun des niveaux qui maximise l'utilité du système et de savoir comment réviser cet ordonnancement quand de nouvelles tâches arrivent dans S. Afin de résoudre ce problème, MOUADDIB et ZILBERSTEIN proposent de modéliser le problème d'ordonnancement et de contrôle du raisonnement progressif par un problème de contrôle de processus de décision markovien (MDP). Les états du processus représentent alors l'état courant des calculs en terme de tâche et de niveau ainsi que de temps écoulé. A chaque état est associée une quantité, appelée récompense, dépendante de la qualité du résultat. La récompense d'un état est la quantification du bénéfice de l'exécution de chaque niveau ou d'une tâche. Le modèle de transition est défini par l'incertitude sur la durée d'exécution associée au niveau sélectionné pour l'exécution. En fait, les deux types de transitions possibles sont soit "exécuter un niveau supplémentaire", soit "passer à la tâche suivante en exécutant son premier niveau". Sachant cela, il est supposé que les tâches sont ordonnées en fonction de leur date limite et qu'une fois

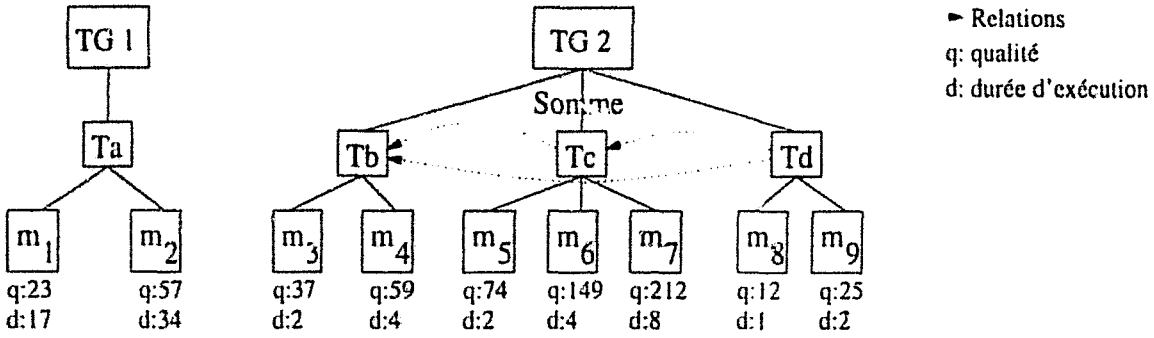


FIG. 1.12 – *Exemple de structure de tâche à ordonnancer par le "design-to-time".*

faite la transition vers une tâche suivante, il n'est pas question de revenir sur la tâche "abandonnée".

Grâce à cette approche et avec ses hypothèses, MOUADDIB et ZILBERSTEIN établissent qu'une politique optimale pour le MDP (un ensemble de transitions qui donne la récompense finale maximale) est un ordonnancement optimal pour le problème de raisonnement progressif. L'arrivée de nouvelles tâches en cours d'exécution est prise en compte en reclassant la nouvelle tâche en fonction de sa date limite. Des extensions du modèle que nous venons de décrire sont en cours ou achevées. Nous pensons en particulier à la prise en compte d'une incertitude dans la qualité, la dépendance entre les qualités (résultat d'un niveau précedent), et à la gestion des différentes tâches dans un enchaînement non-linéaire, et la prise en compte de plusieurs modules par niveau et d'un environnement dynamique [ZM99].

Pour situer cette approche de raisonnement progressif par rapport à l'algorithmique anytime, nous pouvons dire que l'existence de niveaux d'exécution fait que le raisonnement progressif s'inscrit également dans l'approche de compromis entre temps et qualité.

1.3.5 Design-to-time scheduling

Le "Design-to-Time", proposée par GARVEY et LESSER [GL93], vise également à élaborer un plan d'exécution en tenant compte de tout le temps disponible et en maximisant la qualité du résultat fourni. Dans cette approche, le problème à résoudre est modélisé en un ensemble de tâches interdépendantes, avec des alternatives pour accomplir chacune des tâches et non-pas une solution unique. Il existe alors pour chaque tâche un éventail de réponses possibles de différentes qualités et de différents temps d'exécution. La qualité de la solution au problème global est une fonction de la qualité des tâches individuelles Le principal sujet de recherche sur le "design-to-time" porte la prise en compte de l'interaction entre les sous-problèmes quand on construit un plan d'exécution. GARVEY et LESSER distinguent des interactions "dures" qui doivent être respectées pour trouver des solutions correctes (par exemple, des contraintes de précédence) et des interactions "souples" qui peuvent améliorer ou influencer les performances (par exemple, les contraintes "facilitantes", c'est-à-dire la tâche A facilite l'exécution de la tâche B).

Nous décrivons un exemple de la représentation d'un problème qui peut être résolu par le "design-to-time" en figure 1.12 basé sur le modèle TAEMS (Task Analysis, Environment Modeling, and Simulation [DL93]). Dans la struture de tâche TAEMS, les feuilles du graphes représentent les parties exécutables, appelées *méthodes* (ici m_1, m_2 etc.), et les autres noeuds, c'est-à-dire les noeuds supérieurs (ici T_a, T_b, etc.) des tâches composées dont la qualité est fonction de la qualité des sous-tâches. Chaque graphe isolé est appelé *groupe de tâches* (TG_i). Les connexions horizontales entre les tâches représentent les interactions, comme :

- *déclenche (enables)* : la tâche A doit atteindre un certain seuil de qualité avant que la tâche B puisse commencer correctement son exécution.

- *facilite (facilitates)* : si la tâche A atteint un certain seuil de qualité, l'exécution de la tâche B est facilitée, c'est-à-dire qu'elle produit un résultat de qualité supérieure dans un même délai ou produit un résultat de même qualité pour un délai plus court.

Etant donné une structure de cette forme, le principe de l'ordonnancement "design-to-time" est de construire dynamiquement des plan d'exécution en privilégiant les plans qui, par ordre de priorité, aboutissent à des qualités non-nulles pour tous les groupes de tâches, maximisent la somme des qualités de tous les groupes de tâches, et minimise le temps total d'exécution des méthodes. Le résultat de cet algorithme d'ordonnancement est un plan d'exécution qui spécifie quelles méthodes exécuter, quand les exécuter et quelles valeurs sont attendues après leur exécution.

GARVEY et LESSER travaillent donc à l'amélioration de cet algorithme d'ordonnancement et à l'extension de leur modèle de tâche afin, par exemple, d'introduire de l'incertitude dans la durée d'exécution et la qualité des méthodes et de nouvelles relations entre les tâches. Il proposent même d'étendre le "design-to-time" à l'ordonnancement des algorithmes anytime par un moyen simple [GL96] : choisir quelques temps d'exécution et leur qualité correspondante dans le profil de performance afin de les considérer comme des méthodes différentes pour une tâche. Le problème alors est de déterminer combien de points il faut prélever dans le profil de performance pour obtenir des résultats satisfaisants, c'est-à-dire un ordonnancement dont le temps d'exécution est raisonable. Le nombre de points est assez faible (de l'ordre de 4 ou 5 méthodes) car ensuite, même sans considérer la complexité de l'ordonnancement, sa qualité n'augmente plus sensiblement.

Un des derniers axes d'étude de GARVEY et LESSER est de considérer l'algorithme d'ordonnancement "design-to-time" comme un algorithme anytime[GL96]. En effet, cet algorithme donne des résultats toujours meilleurs si un temps supplémentaire lui est alloué, c'est-à-dire si un plus grand nombre d'alternatives sont considérées. Par contre, les auteurs sont peu optimistes sur la possibilité de construire un profil de performance *a priori* pour l'ordonnnancement d'une structure de tâche particulière.

1.4 Conclusion

Nous venons de présenter une analyse bibliographique dans le domaine de l'algorithmique anytime. Nous avons pu voir que pour obtenir le compromis entre la performance de l'algorithme et son temps de calcul, il était nécessaire de définir un "évaluateur" appelé qualité, qui placé dans un environnement applicatif faisait place à la notion d'utilité, et de construire un profil de performance, reflet de l'évolution de cette qualité en fonction du temps de calcul. Nous avons également vu que deux types d'algorithmes anytime existaient (interruptibles et à contrat) et qu'il était possible de passer de l'un à l'autre avec certaines concessions. Nous avons également exposé les indications préconisées par ZILBERSTEIN pour la construction de tels algorithmes. Enfin, un travail sur la composition d'algorithmes anytime a été présenté.

Nous avons dans un deuxième temps abordé les problèmes d'utilisation des algorithmes anytime et notamment l'exploitation de compromis temps/performance. Cette partie a donc concerné les techniques d'ordonnancement et de contrôle des algorithmes anytime. Nous avons vu que des alternatives à ces algorithmes existaient, notamment le raisonnement progressif permettant, entre autres, de se passer de la notion de profils de performance.

Remarquons que ces approches différentes au premier abord, se rejoignent dans le principe de compromis temps/performance. C'est pourquoi, nous les classons toutes dans l'approche algorithmique anytime. Notons que l'ordonnancement des algorithmes à contrat, de par leur nature non-interruptible, se fait avant de lancer l'exécution, contrairement aux algorithmes interruptibles. De plus, les techniques d'ordonnancement proposées supposent la plupart du temps qu'un évènement interrupteur va survenir à une date fixée et précise, ou au pire, situé avec une certaine incertitude sur un intervalle de temps limité.

Il est alors légitime de se demander s'il n'existe pas des situations pour lesquelles non-seulement il n'y a pas d'information sur l'arrivée de l'évènement interrupteur, à part un intervalle de temps d'une certaine taille qui peut être conséquente, mais aussi pour lesquelles nous ne disposons pas d'algorithme interruptible, mais seulement à contrat. Dans le chapitre suivant nous proposons une approche dans laquelle il est possible de donner un ordonnancement d'un algorithme à contrat qui garantit le meilleur résultat en moyenne sur l'intervalle de temps. Cette approche peut être vue comme une alternative à la transformation d'un algorithme à contrat en algorithme interruptible proposée par ZILBERSTEIN et RUSSEL, présentée dans ce premier chapitre.

Chapitre 2

Maximiser la qualité moyenne

> En toutes choses les extrêmes sont rares, les choses moyennes très communes
> [Platon]

A notre connaissance, la plupart des études menées à ce jour sur l'optimisation de la qualité des algorithmes anytime n'ont considéré que la qualité du résultat final. Le problème auquel nous nous intéressons ici est celui de maximiser la qualité moyenne d'un algorithme anytime à contrat sur un intervalle de temps. Tout d'abord, nous illustrons et motivons informellement ce problème avec quelques situations concrètes. Ensuite, nous prouvons que le problème est NP-dur, mais quadratique si l'intervalle de temps est assez grand. Finalement, nous donnons des résultats empiriques qui nous permettront d'envisager les cas pratiques pour l'utilisation de notre approche.

2.1 Exposé du problème

2.1.1 Situation

Les problèmes NP-durs comme la planification ou la prise de décision ne peuvent être traités de manière raisonnable par des méthodes exhaustives. C'est la raison pour laquelle DEAN et BODDY [DB88] ont introduit la notion d'algorithme anytime. Ces algorithmes offrent un compromis entre temps et performance. Ils sont donc caractérisés par leurs profils de performance qui permettent une prédiction sur la qualité du résultat qui dépend de la longueur du temps d'exécution. Cette façon de faire a été utilisée dans des domaines variés comme le contrôle de robots (ZILBERSTEIN et RUSSEL [ZR93]), le raisonnement à base de connaissance (MOUADDIB et ZILBERSTEIN [MZ98]) et les agents réactifs (ADELANTADO et DE GIVRY [Ad95]), entre autres.

La qualité n'est pas la caractéristique essentielle du résultat d'un calcul : ce qui importe vraiment est son utilité. L'idée intuitive est que, dans beaucoup de situations, l'utilité d'un résultat décroît avec le temps, et qu'un résultat de qualité moyenne obtenu rapidement est plus intéressant que le meilleur résultat obtenu après un temps de calcul excessif

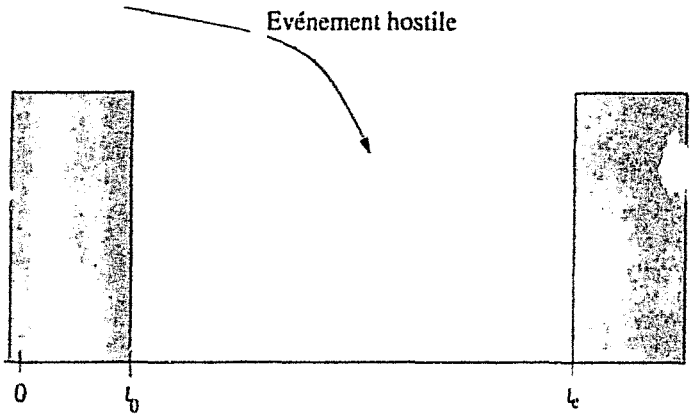


FIG. 2.1 - *Un événement hostile peut arriver à tout moment sur* $[t_0,t_e]$

(ZILBERSTEIN et RUSSEL [ZR96]). Mais quand l'algorithme est utilisé dans un certain environnement, l'utilité peut être d'une autre nature. C'est le cas dans les exemples suivants, tous associés à une "situation de crise" :

- Une personne (P) doit donner à son chef de service (C) un rapport dans la matinée. P sait seulement que C réclamera le rapport à n'importe quel moment entre 8 heures du matin et midi. Le problème pour P est qu'essayer de faire un rapport avec la meilleure qualité possible ne serait la bonne stratégie que si le rapport était réclamé à midi. Si ce n'était pas le cas, aucun rapport ne serait disponible! Ainsi, il semble plus prudent d'assurer un brouillon de qualité minimale pour 8 heures et, une fois que ce brouillon est prêt, de demarrer la rédaction d'un meilleur rapport, en espérant que le chef de service C ne passera que tard dans la matinée.
- Quand une tornade est annoncée, il reste très peu de temps pour se préparer (et préparer sa maison). Aussi, il est important de fabriquer des protections de la meilleure "utilité" possible, par exemple de s'assurer que des actions minimales seront effectuées d'abord (protéger les enfants) avant d'améliorer la qualité des protections (clouer les volets).

La question que nous nous posons est donc la suivante: dans la situation d'un événement hostile qui pourrait intervenir à n'importe quel moment sur un intervalle de temps donné $[t_0,t_e]$, la probabilité de l'événement hostile étant considérée uniforme sur l'intervalle, quelle est la réponse à cet événement hostile à partir de $t = 0$ qui donne la meilleure chance de survie sur toute l'étendue de l'intervalle de temps? (voir figure 2.1)

2.1.2 Les méthodes possibles

Dans la suite, nous supposons qu'un critère de qualité est défini pour les résultats donnés par l'algorithme. La définition de la qualité est fortement dépendante du problème (précision du résultat, bénéfice obtenu, pourcentage d'un possible succès). Nous supposons toujours que la qualité augmente (ou au pire reste constante un moment) avec le temps.

Cette définition de qualité est conforme aux définitions données dans l'état de l'art, donc aux définitions usuelles du domaine de l'algorithmique anytime.

2.1.2.1 Les algorithmes interruptibles

Dans la situation décrite précédemment, un algorithme anytime interruptible serait la meilleure solution. Il suffit alors de lancer l'algorithme de "contre-attaque" au début de l'intervalle, quand on vient juste d'être prévenu de la possibilité de l'événement hostile. Lorsque ce dernier survient, l'algorithme est interrompu et le résultat du calcul est celui obtenu au moment de l'interruption. C'est donc la meilleure réponse qu'il était possible de donner au moment de l'arrivée de l'événement hostile.

Malheureusement, il n'est pas toujours possible d'obtenir ce genre d'algorithmes, au moins pour les raisons suivantes :

- Dans le cadre de la conception modulaires d'algorithmes anytime. En effet, si on veut faire une composition d'algorithmes anytime, interruptibles ou à contrat, pour obtenir un module composé anytime, on n'obtient que des modules composés à contrats [Zil93].
- Les algorithmes à contrats peuvent être transformés en algorithmes interruptibles [Zil93], mais nonobstant le fait que le temps d'exécution est quatre fois plus long pour obtenir la même qualité, cette façon de faire ne peut s'appliquer que si la longueur des contrats peut être choisie librement (dans ce cas précis, la longueur des contrats est une suite exponentielle), ce qui n'est pas forcément le cas général.

Finalement, même avec un authentique algorithme interruptible, il existe des situations pour lesquelles le cas à contrat réapparait : si on applique les résultats de l'algorithme sur le monde extérieur, il n'est plus possible de laisser l'algorithme continuer le calcul pour améliorer la solution à partir de la précédente obtenue: il doit être redémarré de zéro. Par exemple, cela peut arriver dans les applications de contre-mesure, si le résultat de l'algorithme interruptible est une action de brouillage qui provoque elle-même une décision de contre-brouillage de la part de l'ennemi, etc.

2.1.2.2 Les algorithmes à contrat

Il est immédiat que les algorithmes à contrat posent un problème car ils ne sont pas, par définition, interruptibles. Imaginons la même situation "d'attaque" et un algorithme à contrat pour préparer une réponse. Si jamais l'attaque intervenait avant la fin du calcul (la fin du contrat), nous n'aurions aucune garantie sur la qualité du résultat, par définition d'un algorithme à contrats. Il est même fort probable que nous n'ayons en conséquence aucune réponse du tout à opposer à l'événement hostile.

Nous illustrons sur les figures 2.2, 2.3 et 2.4 différentes tactiques envisageables.

Le premier objectif est donc de palier au problème de non-interruptibilité de l'algorithme à contrat, c'est à dire qu'il faut assurer un minimum de qualité très tôt. L'algo-

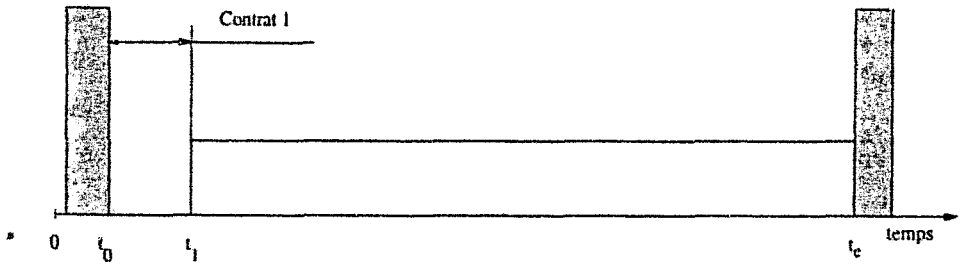


FIG. 2.2 - *Préparation rapide, mais peu efficace*

rithme est donc lancé sur un contrat court. Le problème est que la réponse calculée est de qualité très basse, d'autant plus que le contrat est court. Même si on dispose d'une riposte qui couvre une bonne partie de l'intervalle, elle ne sera pas forcément très efficace.

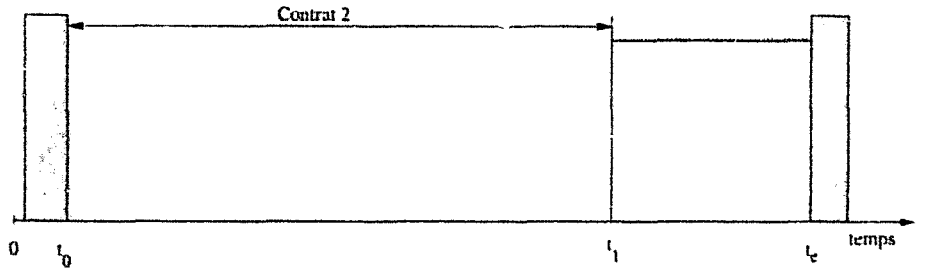


FIG. 2.3 - *Préparation efficace mais lente*

Pour obtenir une meilleure qualité, il faut lancer l'algorithme avec un contrat plus grand (figure 2.3). Toutefois un problème subsiste : la qualité de la réponse est grande, mais elle n'est utilisable qu'après la fin du contrat. Sur la durée du contrat, on est sans protection.

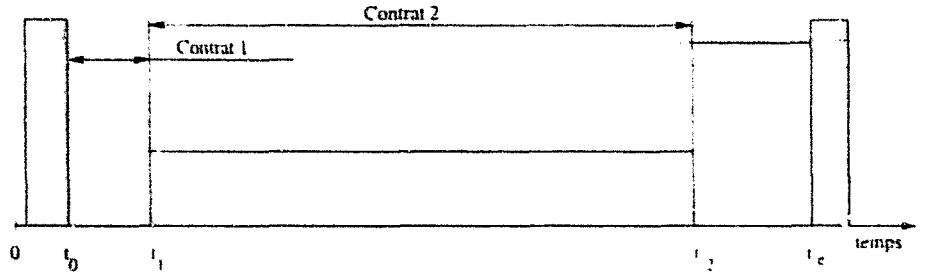


FIG. 2.4 - *Préparation combinée*

La solution que nous proposons (figure 2.4) part d'un constat simple : si on lance l'algorithme avec un contrat assez court, il reste toujours du temps pour lancer une deuxième fois l'algorithme et obtenir un meilleur résultat. Pour évaluer la qualité de la réponse, nous utilisons une nouvelle mesure, la qualité moyenne, qui consiste simplement à calculer la qualité pondérée par la portion de l'intervalle sur lequel elle s'applique. Cette mesure nous donne une idée du compromis entre la qualité d'un résultat et le temps pendant lequel il

est disponible. Il faut toutefois réunir trois conditions:

- la somme des contrats doit être strictement inférieure à la longueur de l'intervalle
- chaque nouveau contrat doit être plus long que le précédent pour améliorer le résultat. La monotonicité du profil de performance implique qu'une stratégie contraire serait inutile.
- En utilisant la définition de la qualité moyenne, nous pouvons envisager qu'il y ait de nombreux cas où notre tactique sera plus efficace que les deux premières. Autrement dit, sur la figure 2.4, nous obtenons une meilleure qualité moyenne, c'est à dire une plus grande chance de survie sur tout l'intervalle.

L'approche que nous décrivons peut être illustrée sur un exemple concret d'application : la prédiction d'un signal. La prédiction a pour but, pour un signal dont on connaît un certain nombre d'échantillons, c'est à dire de valeurs mesurées à des instants déterminés, de prédire la valeur de ce signal dans le futur. Cette prédiction peut servir, par exemple, à des programmes d'identification de cibles et constitue une application typique dans le domaine du traitement du signal.

Pour résoudre ce problème, il existe plusieurs algorithmes dont l'un consiste à trouver les coefficients d'un prédicteur d'un ordre fixé *a priori*. Plus l'ordre de ce prédicteur est élevé, plus l'erreur faite sur la prédiction est faible. Par contre, les coefficients déjà calculés ne sont pas réutilisables pour calculer les coefficients d'un prédicteur d'ordre supérieur si, pour calculer des coefficients du prédicteur, on résoud les équations normales. Il est donc nécessaire de recommencer les calculs des coefficients depuis le début.

Cet algorithme peut donc être facilement adapté à l'algorithmique anytime dans le cas à contrats car il garantit que l'erreur diminue avec le temps de calcul (lié à l'ordre de la prédiction), et il est nécessaire de relancer l'algorithme pour exécuter un contrat plus long (pour un ordre plus élevé). Pour une description complète des principes de cet algorithme, nous renvoyons le lecteur à [Hay96], [PM96] et [DV00].

2.2 Formalisation

Le problème est de maximiser une mesure que nous avons appelée qualité moyenne sur une fenêtre de temps où l'arrivée d'un événement hostile est possible. Notre stratégie est la suivante : l'algorithme est lancé une première fois pour un contrat déterminé (de longueur inférieure à la longueur totale de la fenêtre, évidemment). Nous obtenons alors un résultat d'une certaine qualité, dépendant du contrat, et rien avant la fin de ce premier calcul. Nous avons donc une qualité nulle sur $[t_0,t_1]$, et une qualité $f(t_1 - t_0) = f(\theta_1)$ à partir de t_1. Ensuite, si l'algorithme est relancé pour un nouveau contrat, cette qualité est mise à jour à la fin de l'intervalle. Et ainsi de suite, jusqu'à la fin de la fenêtre temporelle (voir figure 2.5).

Le problème de maximisation de la qualité moyenne sur un intervalle de temps consiste à trouver le nombre de contrats et leurs valeurs pour obtenir une qualité moyenne maxi-

male. Nous allons tout d'abord définir la qualité moyenne d'après la description que nous venons de faire.

Nous considérerons un ensemble de n contrats $< \Theta_n >= \{\theta_1,\theta_2,...,\theta_n\}$. A cause de la forme monotone croissante des profils de performance, il faut choisir un contrat plus long pour obtenir une meilleure qualité. Aussi les contrats seront supposés croissants de θ_1 à θ_n. Nous donnons ci-dessous une description de la qualité moyenne qui est la moyenne (au sens usuel) des qualités disponibles, c'est-à-dire des qualités obtenues au dernier contrat achevé lors d'une interruption au hasard sur l'intervalle de temps.

qualité moyenne - qualité intégrale

La qualité moyenne de f sur une fenêtre de temps $[t_0,t_e]$, relative à un choix $< \Theta_n >$ de n contrats est décrite par la formule :

$$\overline{Q}(f, < \Theta_n >) = \frac{1}{t_e - t_0}(f(\theta_1)\min(0,\theta_1 - t_0) + \sum_{i=1}^{n-1} f(\theta_1)\theta_{i+1} + f(\theta_n)(t_e - \sum_{i=1}^{n} \theta_i))$$

La formule ci-dessus n'est valable que dans le cas où la contrainte de somme est vérifiée :

$$\sum_{i=1}^{n} \theta_i < t_e$$

La qualité intégrale est égale à la qualité moyenne multipliée par la longueur de l'intervalle de temps $t_e - t_0$. La qualité intégrale sera donc notée $Q(f, < \Theta_n >)$ et la qualité moyenne ou espérance de qualité $\overline{Q}(f, < \Theta_n >)$.

$$Q(f, < \Theta_n >) = f(\theta_1)\min(0,\theta_1 - t_0) + \sum_{i=1}^{n-1} f(\theta_1)\theta_{i+1} + f(\theta_n)(t_e - \sum_{i=1}^{n} \theta_i)$$

On notera $Q(f)$ le maximum de qualité intégrale, de même $\overline{Q}(f)$ le maximum de qualité moyenne.

Sur la figure 2.5, nous avons représenté d'une part le profil de performance avec les contrats choisis, figurés comme des temps de calcul, d'autre part l'espérance de qualité avec les contrats qui se succèdent. θ_1, θ_2 et θ_3 sont les contrats choisis pour maximiser la qualité moyenne de f sur $[t_0,t_e]$. t_0 est le début de l'intervalle où il est possible de réagir (d'exploiter le résultat de l'algorithme), t_e sa fin. L'origine des temps est fixée à $t = 0$, c'est à dire qu'à partir de cet instant, il est possible de commencer les calculs (lancer le premier algorithme). L'instant 0 est donc celui à partir duquel on est prevenu de l'arrivée de l'événement hors [illegible] somme de A_1, A_2 et A_3 est la qualité intégrale.

Nous distinguons deux cas à cause de la place de t_0 par rapport au premier contrat. Nous pourrions envisager de considérer plusieurs contrats avant t_0. Il faut donc justifier le fait qu'il n'y en a qu'un seul. Imaginons deux contrats avant t_0. Le résultat est représenté sur la figure 2.6.

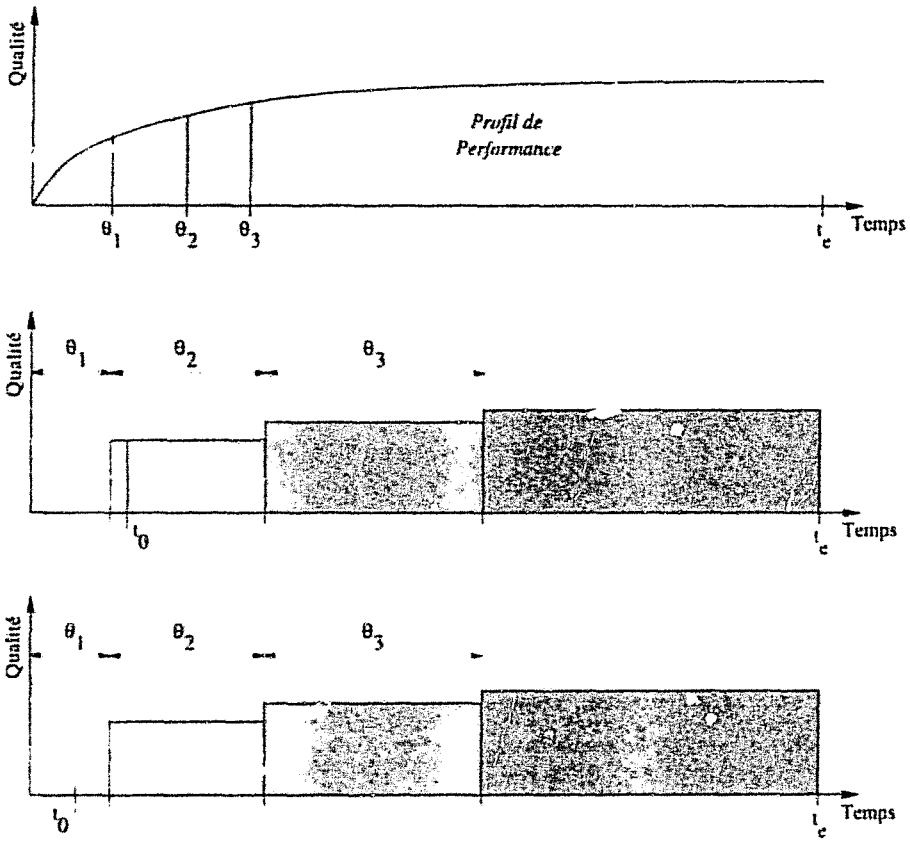


FIG. 2.5 – *Profil de performance et qualité moyenne*

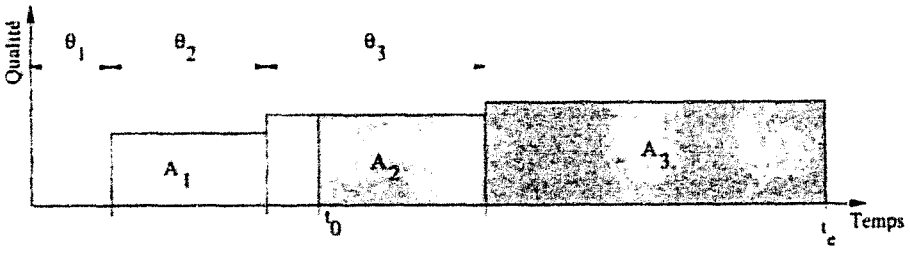


FIG. 2.6 – *Deux contrats avant* t_0

Nous pouvons constater que, dans ce cas, seule la qualité après le deuxième contrat est prise en compte. Le premier contrat n'a donc servi à rien. De plus, le temps consommé par ce premier contrat aurait pu servir à améliorer la qualité obtenue à l'instant t_0. Dans le contexte de la recherche de l'espérance de qualité maximale, nous pouvons donc affirmer que s'il existe un contrat avant t_0 dans la stratégie de qualité moyenne optimale, alors il est unique. Ceci justifie donc bien le fait que nous ne distinguions que deux cas pour la définition de la qualité intégrale.

Remarques sur la longueur du premier contrat :

- La somme des deux premiers contrats est supérieure à t_0. En effet, dans le cas contraire, le premier contrat ne servirait à rien puisqu'on ne tire bénéfice du résultat calculé qu'à l'instant t_0. Il y a donc au plus un contrat avant t_0.
- De cette première remarque, il vient que le deuxième contrat est nécessairement plus grand que $t_0/2$ car la somme des deux premiers contrats et les contrats sont croissants.
- Il se peut qu'il soit intéressant de faire un premier contrat moins long que t_0. En effet, cela peut permettre de lancer l'algorithme pour un contrat supplémentaire et augmenter ainsi la qualité moyenne, si la qualité augmente "fortement" pour ce contrat.

Définition 2.1 *Maximiser la qualité moyenne*

Le problème de la maximisation de la qualité moyenne $\overline{Q}(f, < \Theta_n >)$ *est de trouver, pour une fenêtre temporelle donnée* $[t_0, t_e]$, *le nombre .. de contrats ainsi que la longueur de ces contrats qui maximisent l'expression de* $\overline{Q}(f, < \Theta_n >)$.

2.3 Etudes selon les types de fonctions de qualité

Nous avons essayé dans un premier temps d'exhiber des lois analytiques générales sur le comportement de la qualité moyenne, et, si possible, de mettre en évidence des classes de fonctions caractéristiques de ces comportements. C'est pourquoi nous avons considéré tout d'abord un cas "idéal" de profils de performance continus et dérivables afin d'en tirer une première classification. Les résultats que nous avons obtenus sont exprimés dans ce paragraphe et ont été résumés dans [DDTV98].

Il nous est tout de suite paru naturel que le choix du nombre d'interruptions nécessaires et de leur position pour optimiser l'espérance de qualité dépendrait fortement de l'allure du profil de qualité de l'algorithme. Aussi, nous avons commencé l'étude des fonctions de qualité linéaires qui séparent deux types de fonctions: les fonctions de qualité convexes et concaves (le cas le plus souvent recherché habituellement). Nos résultats analytiques préliminaires ont confirmé cette classification. Notons toutefois que ces deux types de fonctions ne couvrent pas entièrememt la classe des fonctions continues, dérivables et croissantes.

2.3.1 Résultats pour les profils de performance convexes

Nous traitons ici le cas des profils de performance convexes, c'est à dire dont la dérivée seconde est positive. Ce premier cas n'est d'évidence pas le plus intéressant d'un point de vue pratique, car la plupart des profils de performance recherchés sont concaves (à dérivée seconde négative). En effet, c'est naturellement l'assurance d'un résultat "d'assez bonne qualité assez vite" qui est recherchée.

Toutefois, nous obtenons des résultats définitifs sur le comportement de la qualité moyenne en fonction du nombre et de la longueur des contrats. Ces résultats sont exprimés dans le théorème 2.1.

Théorème 2.1 *Maximisation de la qualité moyenne pour les profils de performance convexes*

- *Pour les profils de performance convexes, le meilleur résultat pour la qualité moyenne est obtenu avec un seul contrat.*
- *Soit f, un contrat unique qui donne la meilleure qualité moyenne sur l'intervalle $[t_0,t_e]$. Alors $f > \frac{t_e}{2}$.*
- *L'unique contrat est défini par l'équation suivante :*

$$\theta = t_e - \frac{f(\theta)}{f'(\theta)}$$

si f' est la dérivée première de la fonction de qualité f (valable pour f concave ou convexe).

Remarque :

Ces résultats n'ont pas de sens si l'on permet que $\theta_1 < t_0$. Dans ce cas, une stratégie à un seul contrat donne un contrat optimal $\theta_1 = t_0$ à la limite. Donc cette stratégie avec $\theta_1 < t_0$ est moins bonne de toute façon sauf si $t_0 > \frac{t_e}{2}$. Dans ce dernier cas, tout dépend de la valeur de t_0.

Pour un contrat unique, l'idée intuitive du théorème 2.1 est assez naturelle. Comme le profil de performance est convexe, c'est à dire qui croît lentement au début, on a intérêt à stopper le calcul tard.

La deuxième partie du théorème 2.1 signifie qu'il n'est pas utile de relancer plusieurs fois l'algorithme avec des contrats croissants pour obtenir la meilleure qualité moyenne. Nous pouvons affirmer dans ce cas des fonctions de qualité convexes, que le problème de maximisation de la qualité moyenne sur un intervalle peut être résolu en temps constant. La formule donnée dans la troisième partie est valide en général (cas convexe ou concave). La preuve complète du théorème 2.1 est reportée dans le paragraphe 2.3.3 afin de ne pas troubler l'enchainement des résultats.

2.3.2 Résultats préliminaires pour les fonctions concaves

Le cas des fonctions concaves que nous allons aborder maintenant nous a posé plus de problèmes. En général, la qualité moyenne maximale nécessite plusieurs contrats, mais la question reste de savoir combien et si ce nombre est fini. Toutefois, nous avons exhibé des résultats préliminaires que nous exposons dans les théoremes suivants, et expliquons comment nous traitons le problème.

Théorème 2.2 *Maximiser la qualité moyenne pour les profils de performance concaves - résultats préliminaires*

- *Pour $f(t)$, un profil de performance concave, et θ, une interruption unique qui donne la meilleure qualité moyenne Q, alors $\theta < \frac{t_s}{2}$.*
- *Pour $f(t)$, un profil de performance concave, faire deux contrats donne une meilleure qualité moyenne que faire un seul contrat.*

Pour les mêmes raisons que ci-dessus (en 2.3.1), les profils de performance concaves donnent un assez bon résultat assez vite, donc on peut arrêter le calcul assez vite également.

Pour la seconde partie du théorème, l'idée est que dans le cas d'un contrat unique celui qui donne la qualité moyenne maximale est toujours inférieur à $t_s/2$, donc il reste assez de place pour un second contrat qui améliorera la qualité moyenne pour un seul contrat.

A ce jour, nous n'avons de preuve analytique que cette propriété s'étend aux ordres supérieurs (c'est à dire que trois contrats valent mieux que deux, etc.). Toutefois, de façon plus pragmatique, nous pouvons imaginer que faire beaucoup de contrats n'aura pas un grand intérêt pour deux raisons :

- le peu de gain à chaque nouveau contrat. En effet, la qualité augmente, mais le temps sur lequel elle s'applique diminue (voir cette impression illustrée sur la figure 2.7). Cela laisse penser que le nombre de contrats sera limité pour obtenir une qualité moyenne maximale, sinon une très bonne approximation pour ce maximum.
- les problèmes de contrôle des algorithmes (monitoring), qui si le nombre de contrats est grand, augmente les temps de réponse d'autant.

C'est pourquoi nous avons traité le problème de façon discrète dans la section 2.4. Nous reviendrons sur l'utilité de déterminer la qualité moyenne maximale avec beaucoup de contrats dans la suite.

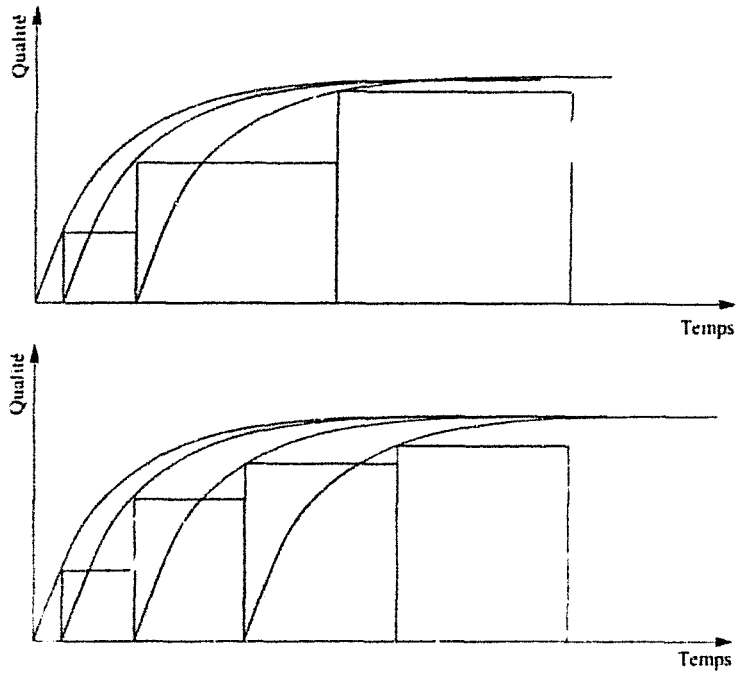


FIG. 2.7 - *Peu de gain pour un contrat supplémentaire*

2.3.3 Détail des preuves

Rappelons la fonction de qualité intégrale : (la qualité moyenne diffère seulement d'une constante $T = t_e - t_0$)

$$Q(f, <\Theta_n>) = f(\theta_1)min(0, \theta_1 - t_0) + \sum_{i=1}^{n-1} f(\theta_i)\theta_{i+1} + f(\theta_n)(t_e - \sum_{i=1}^{n} \theta_i)$$

On notera $Q_1, Q_2, ...Q_n$ les qualités moyennes pour une stratégie à 1, 2 ou n contrats.

Les deux cas (selon le signe de $t_0 - \theta_1$) que nous avons distingués n'influencent pas fondamentalement les résultats.

Théorème 2.1 :

Après la remarque en fin de théorème, nous considèrons que nous ne traiterons que le cas où il est possible de choisir "librement" θ_1, c'est à dire quand $t_0 < \theta_1$.

Première partie :

- Un maximum pour la fonction de qualité $Q_1 = (t_e - \theta_1)f'(\theta_1)$ ailleurs qu'en 0 ou en t_e doit vérifier l'équation :

$$(t_e - \theta_1)f'(\theta_1) - f(\theta_1) = 0$$

- Le fait que la fonction f soit convexe implique que

$$\frac{f(\theta_1)}{\theta_1} \leq f'(\theta_1)$$

- Donc, pour le contrat qui rend Q_1 maximum, il vient

$$0 \leq (t_e - 2\theta_1)f'(\theta_1) \leq (t_e - \theta_1)f'(\theta_1) - f(\theta_1)$$

qui implique que $\theta_1 > \frac{t_e}{2}$(car le second membre est nul).

Deuxième partie :

La première étape consiste à prouver que, pour $n > 2$, le maximum de Q_n est obtenu pour $\theta_k = 0$ et que ce maximum est plus petit que celui de Q_2, c'est à dire que toutes les stratégies à plus de deux contrats sont moins bonnes que le meilleur résultat pour une stratégie à deux contrats.

- . existe un optimum local alors toutes les dérivées partielles de Q_n par rapport aux différents contrats sont nulles. Nous affirmons que c'est impossible car elles ne peuvent même pas être égales: en effet,

$$\forall n > 2, \frac{\delta Q_n}{\delta \theta_1} = \frac{\delta Q_n}{\delta \theta_2}$$

. a pour conséquence

$$\theta_2 f'(\theta_1) = f(\theta_1) + \theta_3 f'(\theta_2)$$

Cette dernière équation ne peut pas être vraie car $\theta_1 < \theta_2 < \theta_3$ et $f(t)$ est positive et $f'(t)$ croît. Donc une des dérivées partielles de Q_n par rapport à θ_1 ou θ_2 est non-nulle et Q_n n'a pas d'optimum local sauf quand θ_1 ou θ_2 atteignent leurs bornes, 0 ou t_e.

- Parce que

$$\sum_{k=0}^{n} \theta_k \leq T$$

si un des contrats $\theta_k = t_e$, alors tous les autres sont nuls et la qualité moyenne est nulle également. Donc le maximum ne peut exister que pour un $\theta_k = 0$

- Si nous supposons qu'un des $\theta_k = 0$ alors $Q_n = Q_{n-1}$. Donc le maximum de Q_n est plus petit que celui de Q_{n-1}. Par récurrence, on obtient finalement que $\forall n > 2, \max(Q_n) \leq \max(Q_2)$

La seconde étape consiste à prouver qu'un seul contrat donne une meilleure qualité moyenne que deux contrats pour les profils de performance convexes:

- Soit $M_{1,f}$ et $M_{2,f}$, les maxima de $Q_1(f)$ et de $Q_2(f)$ respectivement. Le problème est de prouver que $M_{1,f} > M_{2,f}$. Dans ce but, deux autres fonctions de qualité g et h, définies ci-dessous, sont utilisées:
  - h est une fonction linéaire, $h(0) = 0$ et $h(\frac{t_e}{2}) = f(\frac{t_e}{2})$

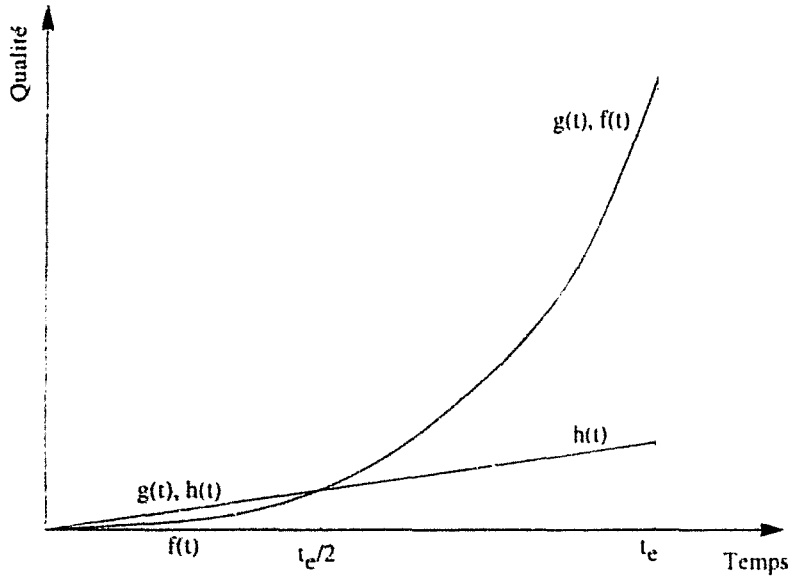


FIG. 2.8 - *Profils de performance f, g et h*

- $g = h$ sur $[0, \frac{t_e}{2}]$ et $g = f$ sur $[\frac{t_e}{2}, t_e]$.

Lemme 2.1 *Pour Q_2, les deux instants de relance de l'algorithme t_1 et t_2 sont tels que $t_1 < \frac{t_e}{2} < t_2$.*

Parce qu'il y a deux contrats et la qualité augmente à chaque étape, $t_1 < \frac{t_e}{2}$. Dans le cas opposé, il serait impossible de placer un second contrat plus grand. De plus, si on suppose que $t_2 < \frac{t_e}{2}$, cela nous mène à $M_{2,g} < M_{2,g}$, qui est absurde. Plus simplement, remarquons enfin que cela nous donnerait une possibilité d'un troisième contrat qui serait meilleur que la stratégie à deux contrats. Ce choix n'est donc pas optimal.

Preuve de la deuxième partie :

- Comme $g = h$ sur $[0, \frac{t_e}{2}]$, $M_{2,g} = M_{2,h} = M_{2,h}(t_1, t_2)$. On peut prouver que dans le cas de f linéaire, le maximum de qualité moyenne est obtenu pour $\forall t_1, t_2 = \frac{t_e}{2}$. Donc $M_{2,h}(t_1, t_2) < M_{2,h}(t_1, \frac{t_e}{2})$. Comme g domine h, $M_{2,h}(t_1, \frac{t_e}{2}) < M_{2,g}(t_1, \frac{t_e}{2}) < M_{2,g}$.
- Le calcul de $M_{2,g}$ et de $M_{1,g}$ prouve que $M_{2,g} = M_{1,g}$.
- Comme g domine f, alors $M_{1,g} < M_{2,f}$.
- Nous prouvons que $M_{1,g} = M_{1,f}$ car $M_{1,g}$ est obtenu pour $t_1 \geq \frac{t_e}{2}$. Alors $M_{1,f} \geq M_{2,f}$. Finalement, on obtient $M_{1,f} \geq M_{2,f}$ en utilisant le fait que la dérivée de $M_{1,f}$ est nulle.

Troisième partie :

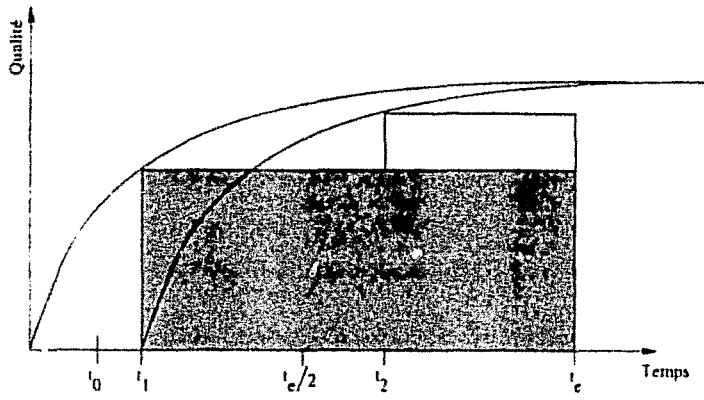


FIG. 2.9 – *Profils de performance concaves - Deux contrats valent mieux qu'un seul*

La preuve vient directement de l'expression de la dérivée de la qualité moyenne par rapport à l'unique contrat.

$$\frac{dQ}{d\theta_1} = (t_e - \theta_1)f'(\theta_1) - f(\theta_1) = 0 \Rightarrow \theta_1 = t_e - \frac{f(\theta_1)}{f'(\theta_1)}$$

Théorème 2.2 :
Première partie :

La preuve est du même type que pour le théorème 2.1 et le fait que f soit concave plutôt que convexe inverse l'inégalité. C'est à dire

$$\frac{f(\theta_1)}{\theta_1} \geq f'(\theta_1)$$

Pour le cas où $\theta_1 < t_0$, de la même façon que dans le théorème 2.1, il vient que le meilleur contrat est égal à t_0 à la limite. Cette stratégie est toute naturelle: calculer le plus possible avant le début des "hostilités".

Seconde partie :

La preuve de cette partie découle directement de la première proposition. En effet, si pour un contrat unique, la longueur du contrat est toujours inférieure à $\frac{t_e}{2}$, il reste alors de la place pour un contrat plus grand que le premier contrat (qui améliore donc la qualité moyenne). La figure 2.9 illustre cette situation.

2.4 Le cas discret : approximation du profil de performance

Toutes les propriétés, définitions et notations que nous donnons dans la section 2.3 sont valables pour un quelconque profil de performance f, répondant aux propriétés de

l'algorithmique anytime. Nous avons choisi de faire des approximations de la fonction f pour traiter le problème de façon algorithmique. Cette approximation est faite par des fonctions en escaliers. Nous la définissons dans la suite et citons les propriétés intéressantes (pour "améliorer" l'algorithme) qui en découlent.

Tout d'abord, grâce au lemme 2.2, nous pouvons remarquer que l'approximation d'un profil de qualité peut se faire de telle façon que l'erreur sur le maximum de qualité moyenne pour un profil de performance approximé ne soit pas plus grande que l'erreur faite sur l'approximation du profil en lui même. Il suffit pour cela d'approximer le profil de performance à ε près, ce que nous définissons de suite, ε étant une constante réelle fixée *a priori*.

Définition 2.2 *approximation par en-dessous d'une fonction f à ε près*

On dit qu'on approxime par en-dessous une fonction f à ε près, si on construit une fonction g telle que $g < f$ et $\forall t \in [t_0,t_e], f(t) - g(t) \leq \varepsilon$, où ε est un réel positif fixé.

Lemme 2.2 *lemme d'approximation*

Si g approxime f par en-dessous à ε près, alors $\overline{Q}(g) \geq \overline{Q}(f) - \varepsilon$.

Preuve du lemme 2.2 :

- Démontrons tout d'abord que

$$\forall \varepsilon, \forall < \Theta >, \overline{Q}(f, < \Theta >) - \overline{Q}(g, < \Theta >) \leq \varepsilon$$

Pour cela, il faut écrire en détail la différence

$$Q(f, < \Theta >) - Q(g, < \Theta >) = min(0,\theta_1 - t_0)(f(\theta_1) - g(\theta_1)) +$$

$$\sum_{i=1}^{n-1} \theta_{i+1}(f(\theta_i) - g(\theta_i)) \quad (t_e - \sum_{i=1}^{n} \theta_i)(f(\theta_n) - g(\theta_n))$$

et utiliser la majoration de $(f - g)$ par ε, en simplifiant :

$$Q(f, < \Theta >) - Q(g, < \Theta >) \leq \varepsilon\{(t_e - t_0) - \theta_1\}$$

Donc

$$\overline{Q}(f, < \Theta >) - \overline{Q}(g, < \Theta >) \leq \varepsilon \frac{\{(t_e - t_0) - \theta_1\}}{t_e - t_0}$$

Ce majorant est plus petit que ε.

- Démontrons ensuite que

$$\forall \varepsilon', \exists < \Theta > | \overline{Q}(f, < \Theta >) \geq \overline{Q}(f) - \varepsilon'$$

C'est la propriété du maximum de qualité moyenne. Pour cela, il faut considérer l'allocation optimale $< \Theta >_{max}$ pour $\overline{Q}(f, < \Theta >)$, et diminuant par exemple le dernier contrat de façon à baisser de ε' la qualité moyenne.

- Les deux premières relations impliquent que

$$\forall\epsilon,\forall\epsilon',\exists < \Theta > |\overline{Q}(f, < \Theta >) - \overline{Q}(g, < \Theta >) \geq \epsilon + \epsilon'$$

Autrement dit, à la limite de ϵ',$\forall\epsilon$,$\overline{Q}(f) - \overline{Q}(g) \leq \epsilon$.

L'erreur faite sur la qualité moyenne en approximant f par g ne sera pas plus grande que l'erreur d'approximation sur f.

L'approximation que nous avons choisie est l'approximation par une fonction en escalier (définition 2.3) qui nous permettra d'obtenir un problème discret.

Définition 2.3 *fonction en escalier*

C'est une fonction constante par morceaux, qui approxime un profil de performance soit par en-dessous à ϵ près, soit à $\pm\epsilon$ près. Nous appelerons seuil l'abscisse d'une discontinuité dans une fonction en escalier (voir algorithme 2.1).

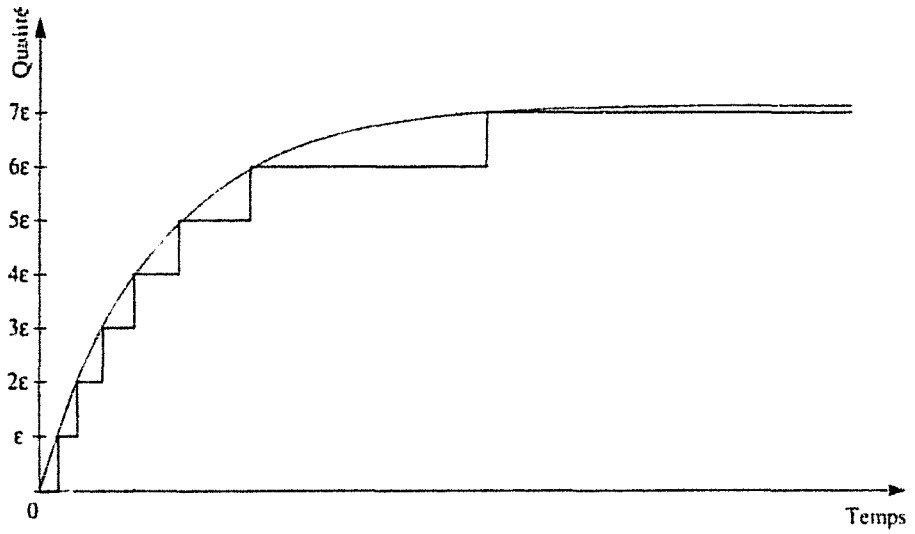


FIG. 2.10 *Approximation d'une fonction f par en-dessous à ϵ près*

Sur la figure 2.10. $\epsilon = 0.05$ et correspond à 5% d'erreur, c'est à dire puisque la valeur du maximum est de 1 (ou tend fortement vers 1). Si le ratio de l'erreur sur le total est fixé à ϵ, alors le nombre de marches sera exactement de $1/\epsilon$.

Remarques :

- Le nombre de seuils est fini. En effet, si l'intervalle d'étude est fini, la valeur de l'erreur détermine le nombre de seuils.

- On pourra prendre comme valeur maximale de g, la valeur rationnelle la plus proche par en-dessous de la valeur du maximum de f. Ainsi, toutes les valeurs de qualité seront rationnelles.
- Considérons la fonction $f_\varepsilon = f + \frac{\varepsilon}{2}$. Il est équivalent d'approximer f par en-dessous à ε près et d'approximer f_ε à $\pm\frac{\varepsilon}{2}$ près, au sens où ces deux approximations donnent les mêmes seuils (mais pas les mêmes qualités).

ALG 2.1 Algorithme d'approximation d'une fonction par une fonction constante par morceaux

---

procedure approxime $(\varepsilon, f, x_1, x_2)$
Début
  $x \leftarrow$ max_diff(x_1, x_2, f)
  si $x > \varepsilon$ **alors**
    max_diff(x_1, x, f)
    max_diff(x, x_2, f)
    insère$(x,$ *liste_seuils*$)$
  fin si
  retourne (*liste_seuils*)
Fin

max_diff est une fonction qui retourne l'abscisse x du maximum en valeur absolue de l'écart entre $f(x)$ et la droite passant par x_1 et x_2.

insère est une fonction qui insère l'élément x dans une liste de seuils, constituée au départ de $\{x_1, x_2\}$.

Voyons maintenant l'intérêt de l'approximation d'un profil de performance par une fontion constante par morceaux. Le lemme 2.3 dit qu'il suffit d'explorer le sous-ensemble de points d'abscisse constitué des seuils pour trouver les allocations optimales pour obtenir la qualité moyenne maximale avec le profil de performance g, fonction en escalier.

Lemme 2.3 *transformation en problème combinatoire*

Soit g une fonction en escaliers

Si une suite finie ordonnée $< \Theta_n > = < \theta_1, \theta_2, \ldots, \theta_n >$ de contrats maximise la qualité moyenne, alors tous les contrats se terminent sur des seuils.

Preuve du lemme 2.3 :

- L'idée est qu'il faut profiter le plus tôt possible de la qualité du résultat calculé
- Prenons une allocation de temps pour des contrats $< \Theta >$, dont θ un de ces contrats finissant entre deux seuils.
- La qualité entre les deux seuils, le deuxième seuil étant exclus dans la marche d'escalier, est constante.

- Prenons alors une autre allocation $< \Theta' >$, identique à $< \Theta >$, à l'exception de θ. Pour remplacer ce contrat, un autre $\theta' < \theta$ est pris, mais sur la même marche d'escalier.
- En faisant la différence entre les deux qualités moyennes obtenues, et en posant θ_i le contrat précédent θ dans $< \Theta >$, et θ_{i+1} le suivant de θ, il vient finalement:

$$Q(g, < \Theta >) - Q(g, < \Theta' >) = (\theta - \theta')(g(\theta_i) - g(\theta_n))$$

- Or, le premier facteur du produit est positif et le second est négatif. La différence entre les deux qualités moyennes est négative et prendre le contrat le plus tôt possible sur la marche d'escalier est bénéfique.
- Par généralisation, à la limite, on conclut que les contrats se finissent toujours sur un seuil.

Nous obtenons donc un problème de maximisation de la qualité moyenne pour un profil de performance en escalier comme les seuls contrats possibles pour le maximum de qualité moyenne. Nous l'appellerons *MAXQSF* pour *MAXimization of the average Quality of a Stepwise Constant Function*. La définition formelle de *MAXQSF* et l'étude de méthodes pour la résolution de ce problème font l'objet des deux sections suivantes.

2.5 Qualité moyenne dans une fenêtre quelconque

Il faut noter que les profils de performance sont souvent acquis de manière expérimentale et leur stockage nécessite une forme discrète. De plus, le lemme d'approximation établit que la qualité moyenne est approximée avec la même erreur que celle faite sur le profil de performance en l'approximant par une fonction en escalier, ce qui nous garantit une erreur maximale choisie (ε) en fonction de la finesse de découpage du temps. C'est pourquoi nous avons choisi de nous attacher au problème de maximisation de la qualité moyenne pour un profil de performance sous la forme d'une fonction continue par morceaux [DDTV99] [DD99]. Plus qu'une façon d'éviter les difficultés dues au cas du profil de performance continu, cette approche prend son sens dans la représentation usuelle d'un profil de performance sous la forme discrète.

Définition 2.4 : *MAXQSF*

Nous appellons $MAXQSF(f,t_0,t_e)$, le problème de la maximisation dans une fenêtre de temps $[t_0,t_e]$ de la qualité moyenne d'un algorithme anytime muni d'un profil de performance sous forme d'une fonction en escalier croissante f.

Dans un premier temps, nous avons voulu étudier la complexité de *MAXQSF*, avant de nous lancer dans la recherche d'algorithmes efficaces. Notre conclusion a été que *MAXQSF* est un problème NP-dur, c'est à dire qu'il n'y a pas d'algorithme "praticable" pour le résoudre de façon exhaustive tant que les classes de complexité P et NP sont distinctes

(pour les définitions sur la complexité algorithmique et les classes de complexité, voir [GJ79] et [Del96]). Nous avons ainsi évité des optimisations algorithmiques aussi longues qu'inutiles.

Rappelons que pour prouver que notre problème est NP-dur, il faut que tout problème NP-dur se réduise polynomialement à ce problème. En utilisant la transitivité pour la réduction polynomiale, il est suffisant de montrer qu'un problème NP-complet se réduit polynomialement à notre problème, puisque que tout problème NP-dur se réduit à un problème NP-complet.

La principale difficulté est de trouver le probleme NP-complet candidat à la réduction polynomiale et de construire cette réduction. $MAXQSF(f,t_0,t_e)$ peut faire penser au problème du sac à dos, que nous noterons $Knapsack(< A >,b)$. Rappelons que $Knapsack(< A >,b)$ consiste à déterminer une partie de $< A >=< a_1,a_2,...,a_n >$ les a_i étant des entiers naturels, dont la somme des éléments soit maximale et inférieure à b, entier naturel. Ceci est en fait une version simplifiée du sac à dos présentée dans [GJ79], mais n'enlève rien à la complexité du problème de la classe des problèmes NP-complets. La ressemblance de $Knapsack(< A >,b)$ avec $MAXQSF(f,t_0,t_e)$ réside dans l'idée du placement maximum (espérance de qualité maximale) de nombres (les contrats) dans espace donné (l'intervalle $[t_0,t_e]$).

Nous avons donc construit une réduction polynomiale de $Knapsack(< A >,b)$ vers $MAXQSF(f,t_0,t_e)$. Notons que l'on suppose que les éléments de $< A >$ sont différents deux à deux. De même que ci-dessus, le caractère NP-dur est conservé par la restriction. La preuve de la NP-complétude suivra l'idée de cette preuve.

2.5.1 Schéma de la preuve de la réduction

On va réduire polynomialement $Knapsack(< A >,b)$ à $MAXQSF(f,t_0,t_e)$. Pour cela, pour toute instance $< A >$ de $Knapsack(< A >,b)$, on va construire en temps polynomial une fonction $f_{<A>}$ croissante en escalier dont la maximisation de la qualité moyenne revient à résoudre $Knapsack(< A >,b)$.

L'idée est de faire correspondre à des seuils les éléments à mettre en sac. La difficulté est que choisir ou ne pas choisir un seuil comme interruption influe sur la qualité dans une proportion qui dépend des autres choix d'une manière difficile à quantifier. Pour s'en convaincre, voir l'expression de Δ, qui exprime le gain de qualité pour l'ajout d'un contrat (*cf.* paragraphe 2.5.2).

Pour palier cette difficulté, $f_{<A>}$ va être construite de façon à alterner des seuils (numéros impairs) correspondants aux éléments de $< A >$, et des seuils (numéros pairs) construits tels qu'ils sont nécessairement tous pris comme contrats dans une qualité moyenne maximale. Ainsi, les seuils pairs encadrent les seuils impairs, et "séparent" l'effet des choix sur les seuils impairs. Cette construction est faite de sorte que "retenir le seuil correspondant à un élément a de $< A >$" augmente de a la qualité (par rapport à ne pas le retenir).

La contrainte de taille du sac devient ici la contrainte de somme sur les seuils de nu-

méro impair; la réduction de *Knapsack* à $MAXQSF$ apparaît alors clairement. D'après la méthode d'alternance des seuils que nous venons de décrire, $f_{<A>}$ a donc 2n+1 seuils: $\theta_0,\theta_1 = a_1,\theta_2,\theta_3 = a_2,\ldots,\theta_{2i-1} = a_i,\theta_{2i},\theta_{2i+1} = a_{i+1},\ldots,\theta_{2n}$.

Pour des raisons techniques que la preuve éclaire, nous sommes amenés à prendre θ_{2i} très proche de a_i. Il y a donc une alternance de marches très courtes et de marches presque entières (voir figure 2.11). Notons par ailleurs que la hauteur des marches décroît exponentiellement, ce qui est proche des cas recherchés dans la pratique (profils de performance concaves: propriété "diminishing return").

Remarque : La réduction repose étroitement sur la correspondance entre *Knapsack* et la contrainte de somme. Nous verrons que quand la contrainte de somme disparaît, le problème devient polynomial (section 2.6).

L'allure des courbes est représentée sur la figure 2.11.

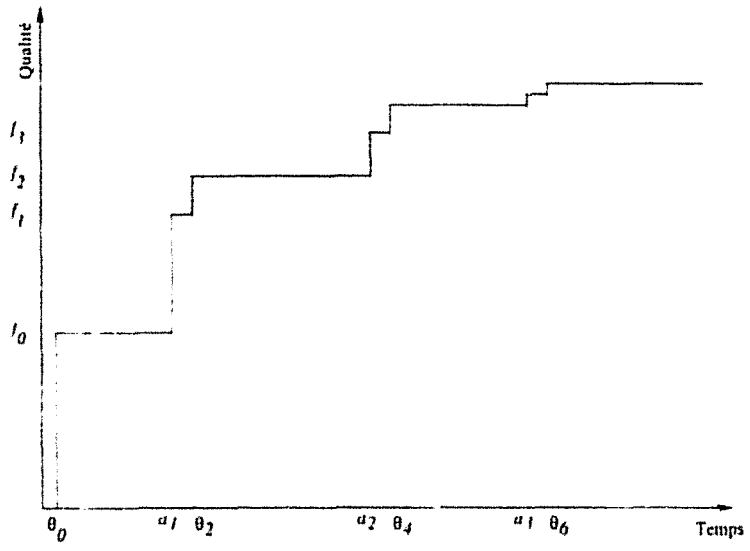


FIG. 2.11 - *Construction de la fonction* $f_{<A>}$

Commentaire :

Une question pourrait être de se demander si le problème demeure NP-dur quand il est restreint au cas des fonctions en escalier "concaves". f est dite concave si son lissage par des segments joignant les arêtes des marches l'est. C'est le cas fréquent où la qualité s'améliore de plus en plus lentement. Plusieurs indices montrent que le problème demeure NP-dur. Il faut remarquer d'abord que les fonctions construites dans notre preuve, sans être convexes (au sens topologique), le sont " à peu près". Surtout, notre construction se restreint au cas "particulièrement simple" où les choix libres de seuils sont séparés par des choix forcés; il et donc probable, même sur des fonctions concaves, la complexité du cas général demeure NP-dure.

Notons que la complexité d'une instance (f,t_0,t_e) du problème $MAXQSF$ est comme il se doit la taille de son écriture.

2.5.2 Preuve de la réduction

Il reste donc à réduire un problème NP-complet à $MAXQSF$, et nous avons choisi $Knapsack(< A >,b)$ pour les raisons indiquées plus avant.

2.5.2.1 Conventions

- $Knapsack(< A >,b)$ désigne le problème d'optimisation du remplissage d'un sac à dos, avec la donnée de $< A >$, ensemble de n entiers positifs $< A >=< a_1,a_2,...,a_n >$. b est un entier désignant la contenace du sac. Nous pouvons supposer les a_i différents 2 à 2 et classés dans l'ordre croissant sans réduire la complexité.
- S désigne la somme des a_i.
- Nous allons considérer des fonctions ayant $2n+1$ seuils $\theta_1,\theta_2,...,\theta_n$ et une fenêtre de temps $[0,t_e]$. Le problème qui nous intéresse devient $MAXQSF(f,t_e)$. Cette simplification ne doit pas "dénaturer" notre problème orignal, et elle ne le fait pas car nous sommes dans le cas particulier où $t_0 = 0$. En fait, t_0 n'intervient que dans la définition analytique de la qualité moyenne et définit l'instant à partir duquel l'évènement hostile peut intervenir. Autrement dit, sa valeur ne change que la valeur de la qualité moyenne mais pas la nature complexe du problème $MAXQSF(f,t_0,t_e)$. Enfin, si une restriction d'un problème est NP-dur alors le problème lui-même est au moins NP-dur.
- Notons $\Delta(\theta,\theta',\theta'',\theta_\varphi)$ "le gain de qualité obtenu en ajoutant le contrat θ' entre θ et θ'', θ_φ étant le dernier contrat" (sachant qu'il n'y en a pas d'autre entre θ et θ'). A partir de la formule de $Q(f, < \Theta >)$ de la section 2.2, nous vérifions immédiatement que ():
  - dans le cas courant :
  $$\Delta(\theta,\theta',\theta'',\theta_\varphi) = \theta''(f(\theta') - f(\theta)) - \theta'(f(\theta_\varphi) - f(\theta))$$
  - si θ' est le premier contrat :
  $$\Delta(-,\theta',\theta'',\theta_\varphi) = \theta'' f(\theta') - \theta' f(\theta_\varphi)$$
  - si θ' est le dernier contrat :
  $$\Delta(\theta,\theta',-,-) = (f(\theta') - f(\theta))(t_e - \theta_1 - \theta_2 - ... - \theta - \theta')$$
- Dans le contexte d'une fonction f en escalier ayant pour seuils $s_0,...,s_i,...,s_p$, nous posons $\Delta_i = \Delta(s_{i-1},s_i,s_{i+1},s_p)$ et $f(s_i) = f_i$. Il vient alors :
$$\begin{aligned}\Delta_0 &= s_1 f_0 - s_0 f_p \\ \Delta_i &= s_{i+1}(f_i - f_{i-1}) - s_i(f_p - f_{i-1}) \\ \Delta_p &= (f_p - f_{p-1})(t_e - s_0 - ... - s_p)\end{aligned}$$

- D'après les définitions données plus haut :
  - Δ_0 est l'accroissement de qualité en ajoutant θ_0 comme contrat, θ_1 étant aussi un contrat et le dernier contrat étant θ_{2n}.
  - Δ_i est l'accroissement de qualité en ajoutant le contrat θ_i entre les contrats θ_{i-1} et θ_{i+1}, le dernier contrat étant θ_{2n}.
  - $\Delta_{2n}(< s >)$ est l'accroissement de qualité en ajoutant pour dernier contrat θ_{2n} au lieu de θ_{2n-1},$< s >$ étant l'ensemble des seuils choisis comme contrats (y compris θ_{2n}).

2.5.2.2 Lemmes préliminaires

Lemme 2.4 : *Lemme d'indépendance*

Si $\theta' \neq \theta_\varphi$, $\Delta(\theta,\theta',\theta'',\theta_\varphi)$ *ne dépend pas des choix hors de l'intervalle* $[\theta,\theta'']$.

Pour la preuve, il faut réécrire l'expression de $\Delta(\theta,\theta',\theta'',\theta_\varphi)$ et constater le lemme sur l'expression trouvée.

Lemme 2.5 : *Lemme de monotonie*

$\Delta(\theta,\theta',\theta'',\theta_\varphi)$ *croit avec* θ'' *et décroît quand* θ *ou* θ_φ *croît.*

De même, la preuve est dans l'expression de $\Delta(\theta,\theta',\theta'',\theta_\varphi)$.

2.5.2.3 Lemmes fondamentaux

Nous allons faire ici deux propositions. La première décrit la réduction polynomiale de *Knapsack* vers $MAXQSF$, donc la construction de la fonction en escalier. La seconde proposition prouve l'équivalence entre résoudre $MAXQSF$ et résoudre *Knapsack*.

Lemme 2.6 : *Construction polynomiale du profil*

Pour toute donnée $< A >=< a_1,a_2,...,a_n >$ *et* b, *on peut construire en temps polynomial une fonction croissante en escalier* $f_{<A>}$ *de fenêtre de temps* $[0,T = 0.5 + S + b]$ *ayant* $2n+1$ *seuils* $\theta_1,\theta_2,....,\theta_{2n}$ *et qui vérifie* $\forall i, 1 < i < n$.

- *(P1)* $\theta_{2i-1} = a_i.\theta_0 < \frac{1}{3n}$ *et* $\theta_{2i} < a_i + \frac{1}{3n}$,
- *(P2)* $\Delta_{2i-1} = a_i$ *et* $\Delta_{2i} > S$ *(en particulier,* $\Delta_{2n}(< s >) > S$ *pour tout choix* $< s >$*),*
- *(P3) Remplacer un contrat* θ_{2i-1} *par le contrat* θ_{2i} *ou par le contrat* θ_{2i-2} *améliore la qualité.*

Indication sur la preuve :

Prendre $f_i - f_{i-1} = K^{4n-i}$ avec K suffisamment grand (mais de taille liée polynomialement à $< A >$), et construire $\theta_{2i} < a_i + \frac{1}{3nK^{2n}}$. Une vérification longue conduit à la preuve (*cf.* section 2.5.3).

Lemme 2.7 *: lemme du choix des seuils*

Tout choix optimal de seuils pour $f_{<A>}$ est constitué des $n+1$ seuils de numéro pair et d'une sous suite de seuils de numéros impairs qui résoud Knapsack($< A >$,b).

Indications sur la preuve :

Montrer que les opérations suivantes améliorent successivement la qualité (raisonner sur les parties décimales pour la préservation de la contrainte de somme):

- D'abord remplacer un contrat θ_{2i-1} par le contrat θ_{2i} ou θ_{2i-2} si l'un de ces deux n'était pas un contrat. Ensuite, si θ_{2i} n'est pas un contrat, supprimer tous les contrats qui sont des seuils impairs et ajouter θ_{2i}.
- La qualité optimale requiert donc de prendre tous les seuils pairs comme contrats. Nous achèvons la preuve en remarquant que l'optimisation de la qualité équivaut alors à un choix de seuils impairs dont la somme est maximale mais inférieure à b.

2.5.3 Détail des preuves

Nous donnons ici les preuves détaillés des deux lemmes que nous venons de présenter : le lemme de construction polynomiale du profil et celui du choix des seuils.

Preuve lemme 2.6 :

Nous allons considérer deux constantes K et K' que nous prendrons suffisamment grandes, et nous construirons f par la récurrence $f_i - f_{i-1} = K'K^{2n-i}$. Nous aurons besoin pour la proposition (P3) d'une condition (P'1), donnée dans l'équation 2.1, plus forte que (P1) :

$$(P'_1)\ \theta_{2i-1} = a_i, \theta_0 < \frac{1}{3n} \text{ et } \theta_{2i} < a_i + \frac{1}{3nK^{2n}} \tag{2.1}$$

1/ Montrons par récurrence sur i que pour tout K assez grand, (P1') et (P2) sont vérifiés.

- Initialement, pour tout K et K' assez grands, il vient pour θ_0 assez petit, $\Delta_0 > S$. En effet, $\Delta_0 = \theta_1 f_0 - \theta_0 f_{2n} > f_0 - \theta_0 f_{2n}$ car $\theta_1 = a_1 > 1$ donc

$$\Delta_0 > f_0 - \theta_0(f_0 + K'K^{2n+1})$$

Par exemple, $f_0 > 2S$ et $\theta_0(f_0 + K'K^{2n+1}) < S$ conviennent.

- Supposons $\theta_0,...,\theta_{2(i-1)}$ construits et (P'1) et (P2) vérifiés jusque i-1.
Pour obtenir $\Delta_{2i-1} = a_i$, il est suffisant de prendre

$$\theta_{2i-1} - a_i = a_i \frac{1 + f_{2n} - f_{2i-1}}{f_{2i-1} - f_{2i-2}}$$

En effet, dans ce cas,

$$\Delta_{2i-1} = a_i(f_{2i-1} - f_{2i-2}) + a_i(1 + f_{2n} - f_{2i-1}) - \theta_{2i-1}(f_{2n} - f_{2i-2}) = a_i$$

en remarquant que $\theta_{2i-1} = a_i$.
De plus,

$$\theta_{2i} - a_i = a_i \frac{1 + \sum_{j=2i}^{2n} K^{2n-j}}{K'K^{2n-2i+1}} \approx \frac{1}{K'K^{2n-2i+1}}$$

Pour K assez grand ($K'K^{2n-2i+1} > 3n$), nous obtenons bien $\theta_{2i} - a_i < \frac{1}{3n}$.

- Pour obtenir $\Delta_{2i} > S$, nous remarquons que, compte tenu de $a_{i+1} \geq a_i + 1$(les a_i sont entiers et distincts), de $\theta_{2i+1} = a_{i+1}$ et de $\theta_{2i+1} > a_i + \frac{1}{3nK^{2n}}$, il vient:

$$\Delta_{2i} = \theta_{2i+1}(f_{2i} - f_{2i-1}) - \theta_{2i}(f_{2n} - f_{2i-1})$$

d'où

$$\Delta_{2i} > (1 + a_i)(f_{2i} - f_{2i-1}) - (a_i + \frac{1}{3nK^{2n}})(f_{2n} - f_{2i-1})$$

Et

$$\Delta_{2i} > (1 + a_i)K'K^{2n-2i} - (f_{2n} - f_{2n-1} + ... + f_{2i} - f_{2i-1})(a_i + \frac{1}{3nK^{2n}})$$

soit encore

$$\Delta_{2i} > (1 + a_i)K'K^{2n-2i} - (a_i + \frac{1}{3nK^{2n}})K'(\frac{K^{2n-2i+1} - 1}{K - 1})$$

et finalement

$$\Delta_{2i} > (1 - \frac{1}{3nK^{2n}})K'K^{2n-2i}$$

Or cette expression tend vers l'infini avec K' et K, donc pour K et K' assez grand, $\Delta_{2i} > S$.

- Finalement. pour montrer $\Delta_{2n}(< \Theta >) = (f_{2n} - f_{2n-1})(t_e - \sum(< \Theta >)) > S$, on remarque que:

  - $f_{2n} - f_{2n-1} = K'K^0$ qui tend vers l'infini avec$K$,
  - $t_e - \sum(< \Theta >) > 0.1$ car $< \Theta >$ contient soit des entiers soit des réels très proches (de $\frac{1}{3n}$), donc au pire, la partie décimale de $\sum(< \Theta >) < n\frac{1}{3n} < 0.4$. Or, la partie décimale de t_e vaut 0.5. Nous obtenons donc bien $\Delta_{2n}(< \Theta >) > S$.

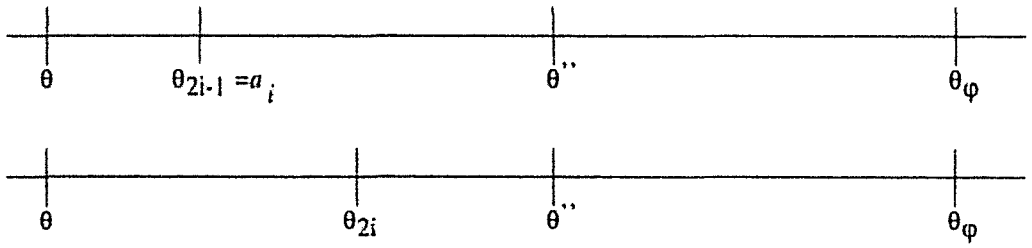


FIG. 2.12 - *Situation de θ,θ'',θ_{2i-1},θ_{2i} et θ_φ*

- Nous venons de prouver que (P1) et (P2) sont vrais au rang i.
- La réduction est polynomiale (car elle revient à un parcours des a_i) si nous remarquons que la taille des données construites est aussi polynomiale ($|K'^{2n}| = 2nlog(K)$).

2/ (P3)a: Preuve que Remplacer un contrat Δ_{2i-1} par le contrat Δ_{2i} améliore la qualité.

- Le raisonnement fait ci-dessus sur les décimales montre que la contrainte de somme est conservée.
- Soit un choix de contrats où θ est le dernier avant θ_{2i-1}, θ'' est le premier après θ_{2i}, et θ_φ est le dernier.

Hormis les cas limites que nous verrons ensuite, il s'agit de prouver que

$$Shift = \Delta_i - \Delta_{i-1} > 0$$

c'est à dire que

$$\begin{aligned} Shift &= \theta''(f_{2i} - f_\theta) - \theta_{2i}(f(\theta_\varphi) - f_\theta) - \theta''(f_{2i-1} - f_\theta) + \theta_{2i-1}(f(\theta_\varphi) - f_\theta) \\ &= \theta''(f_{2i} - f_{2i-1}) - (\theta_{2i} - a_i)(f(\theta_\varphi) - f_\theta) > 0 \end{aligned}$$

- Remarquons que la quantité croît avec θ'' et θ, et croit quand θ_φ décroît. Alors

$$\begin{aligned} Shift > \; & a_{i+1}(f_{2i} - f_{2i-1}) - (\theta_{2i} - a_i)(f_{2n} - f_0) \\ = \; & a_{i+1}(f_{2i} - f_{2i-1}) - \frac{1}{3nK'^{2n}}(f_{2n} - f_0) \\ = \; & a_{i+1}K'K'^{2n-2i} - \frac{1}{3nK'^{2n}}\frac{(K'^{2n}-1)}{K'-1} \end{aligned}$$

  qui est positif si K assez grand.
- Le cas initial revient au cas précédent avec $f_0 = 0$. En effet, nous sommes dans le cas où il n'y a pas de contrat avant θ_{2i-1} (voir paragraphe 2.5.2.1).
- Le cas final revient à prouver (cas où il n'y a plus de contrat après θ_{2i}, hormis θ_φ):

$$Shift = (f_{2i} - f_\theta)(t_e - \theta_1 - \theta_2 - ... - \theta_{2i}) - (f_{2i-1} - f_\theta)(t_e - \theta_1 - \theta_2 - ... - \theta_{2i-1})$$

$$= (f_{2i} - f_{2i-1})(t_e - \sum(< s >)) - \theta_{2i}.(f_{2i} - f_\theta) + \theta_{2i-1}(f_{2i-1} - f_\theta)$$

Or $(t_e - \sum(< s >)) > 0.1$ et $\theta_{2l-1} = a_i$
donc

$$Shift > 0.1(f_{2i} - f_{2i-1}) - \frac{f_{2i}}{3nK^{2n}} + a_i(f_{2i-1} - f_{2i-2})$$

et

$$Shift > 0.1(K'K^{2n-2i}) - \frac{f_{2i}}{3nK^{2n}} + a_iK'K^{2n-2i-2} > 0$$

pour K assez grand.

3/ (P3)b : Preuve que remplacer un contrat θ_{2i-1} par le contrat θ_{2i-2} améliore la qualité

- Il n'y a pas de problème de contrainte de somme, car $\theta_{2i-2} < \theta_{2i-1}$
- Nous voulons $Shift = \Delta_i - \Delta_{i-1} > 0$.

$$Shift = \theta''(f_{2i-2} - f_\theta) - \theta_{2i-2}f(\theta_\varphi) - \theta''(f_{2i-1} - f_\theta) + \theta_{2i-1}f(\theta_\varphi)$$

$$= (f(\theta_\varphi) - f_\theta)(a_i - \theta_{2i-2}) - \theta''(f_{2i-1} - f_{2i-2})$$

Or $a_i - \theta_{2i-2} > 0.5$ (en fait à 2/3) car

$$a_{i+1} \geq a_{i-1} + 1 \text{ et } \theta_{2i-2} < a_{i-1} + \frac{1}{3nK^{2n}}$$

donc

$$Shift > (f(\theta_\varphi) - f_\theta)0.5 - \theta''(f_{2i-1} - f_{2i-2}) > t_e(f_{2i-1} - f_{2i-2})$$

Il faut donc prouver que

$$\frac{f_{2i-1} - f_{2i-2}}{f(\theta_\varphi) - f_\theta} = K'(1 - \frac{1}{K}) < \frac{0.5}{t_e}$$

ce qui est vrai pour K assez grand.

- Dans le cas particulier du premier contrat, nous concluons comme plus haut.

Preuve lemme 2.7 :

1. Tout choix optimal de contrats contient tous les seuils de numéro pair?
   (a) supposons que θ_{2i-1} est un contrat et pas θ_{2i}. Par des considérations sur les parties décimales, nous vérifions que remplacer le contrat θ_{2i-1} par le contrat θ_{2i} conserve la contrainte de somme (cf démonstration (P3)a). D'autre part, (P3)a assure que cette opération augmente la qualité.
   (b) supposons que θ_{2i-1} est un contrat et pas θ_{2i-2}. Si nous remplaçons le contrat θ_{2i-1} par le contrat θ_{2i-2}, la contrainte de somme est conservée et (P3)b assure que la qualité augmente.

(c) Nous déduisons de (a) et (b) que dans un choix optimal, si θ_{2i-1} est un contrat alors θ_{2i-2} et θ_{2i} en sont aussi. Donc, d'après (P2), si on retire θ_{2i-1} dans un choix optimal, on diminue la qualité de la quantité a_i.

(d) supposons qu'un θ_{2i} ne soit pas un contrat dans un choix optimal. D'apres (c) et (P2), le retraitde tous les seuils impairs diminue la qualité d'au plus S. D'après la définition de t_e, il est possible d'ajouter le contrat θ_{2i} en préservant la contrainte de somme, et d'après (P2), cet ajout augmente la qualité d'au moins S, ce qui achève la preuve du point 1. En effet, il est facile de vérifier que plus les contrats voisins sont éloignés, plus un ajout donné augmente la qualité (voir définition des Δ_i et lemme de monotonie). Or (P2) minore par S l'augmentation de qualité dans le cas où les contrats voisins sont les plus proches possibles.

2. Soit Q la qualité correspondant au choix de tous les seuils pairs et aucun impair comme contrats.
Identifions un choix B d'un sous-ensemble de $< a_1,a_2,...,a_n >$ au choix de contrats constitué des seuils pairs et des seuils θ_{2i-1} tels que a_i appartienne à B. Appelons $\sum B$ la somme des éléments de$B$.
D'après (P2), la qualité associée àB est $Q+\sum B$ et la somme des seuils est $t_e+\sum B$. D'après 1/, un choix optimal de contrats est donc un choix B du type considéré qui maximise $\sum B$ sous la contrainte de somme $\sum B < b$, ce qui correspond à résoudre $Knapsack(< A >,b)$.

2.6 Qualité moyenne dans une longue fenêtre

2.6.1 Préambule

Nous supposerons que la fonction en escalier comporte p seuils, notés $s_1,s_2,...,s_p$.

On va chercher à définir une récurrence sur le nombre de contrats pris dans la séquence des seuils et calculer ainsi la qualité optimale.

Posons tout d'abord quelques notations:

- $< \Theta_k >=< \theta_1,\theta_2,...,\theta_k >$ est une séquence ordonnée de k contrats.
- $< \Theta_k|\{\theta_i\} >=< \Theta_k > \cup\{\theta_i\}$ est une séquence ordonnée de k+1 contrats se terminant par θ_i.
- $[S_k] = \{s_1,s_2,...,s_k\}$ est une séquence ordonnée de k seuils de la fonction constante par morceaux f.
- et une certaine quantité $\tilde{Q}$ définie par:

$$\tilde{Q}(f, < \Theta_k|\{\theta_i\} >) = \min(0,\theta_1 - t_0)f(\theta_1)$$

$$+\sum_{l=2}^{k-1} f(\theta_l)\theta_{l+1} + f(\theta_k)\theta_i + f(\theta_i)(t_e - \sum_{l=1}^{k} \theta_l - \theta_i)$$

Le cas qui nous intéresse comporte au moins deux contrats. Les formules données ci-dessus sont valables uniquement avec au moins trois contrats. Le cas avec deux contrats revient à éliminer la somme (deuxième terme de l'équation) et poser k = 2.

De même

$$\tilde{Q}(f, < \Theta_k | \{\theta_j, \theta_i\} >) = \min(0, \theta_1 - t_0) f(\theta_1)$$

$$+\sum_{l=2}^{k-1} f(\theta_l)\theta_{l+1} + f(\theta_k)\theta_j + f(\theta_j)\theta_i + f(\theta_i)(t_e - \sum_{l=1}^{k} \theta_l - \theta_j - \theta_i)$$

De la même façon que ci-dessus, le cas qui nous intéresse comporte au moins trois contrats. Par définition, la formule ci-dessus n'est valable qu'avec quatre contrats. Le cas avec trois contrats revient à éliminer la somme (deuxième terme de l'équation) et poser $k = 2$.

Hypothèse :

On considère que t_e est assez grand pour contenir la somme de tous les seuils possibles s_i, d'où

$$\sum_{i=1}^{p} s_i < t_e$$

Une "longue" fenêtre signifie donc que t_e est assez grand pour que la contrainte de somme disparaisse.

Remarque :

Si l'hypothèse ci-dessus est faite, il est certain que le dernier seuil est dans le choix optimal.

- En effet, si nous choisissons une sous-liste des seuils $< \Theta_k >$ ne contenant pas s_p comme séquence de contrats, le dernier seuil, alors il suffit de rajouter (car c'est toujours possible avec l'hypothèse principale) le dernier contrat s_p à $< \Theta_k >$ pour améliorer la qualité intégrale.
- Donc, un choix de contrats $< \Theta_k >$ ne contenant pas s_p, le dernier seuil, comme dernier contrat n'est pas optimal pour l'espérance de qualité.
- Dorénavant, le dernier contrat θ_φ est fixé et est égal à s_p.

2.6.2 Lemme fondamental

Le lemme suivant définit une récurrence sur les seuils de la fonction en escalier f. Il dit qu'en parcourant la liste des seuils, il est possible de construire le choix optimal des contrats qui permettront de calculer la qualité intégrale maximale.

Lemme 2.8 :

Posons s_i et s_j respectivement le dernier seuil et l'avant dernier choisis comme contrats. Ce ne sont pas des seuils contigus dans le cas général.

Pour une séquence de contrats $< \Theta_k | \{s_j, s_i\} >$, et θ_φ le dernier contrat fixé, égal au dernier seuil, il vient :

$$\forall s_m < s_j < s_i, \max(\tilde{Q}(f, < \Theta_k | \{s_j, s_i\} >)) =$$
$$\max\{\max(\tilde{Q}(f, < \Theta_k | \{s_m, s_j\} >)) + s_i(f(s_j) - f(\theta_\varphi))\}$$

et en particulier :

$$\forall s_i < \theta_\varphi, Q(f) = \max(\tilde{Q}(f, < \Theta_k | \{s_i, \theta_\varphi\} >))$$

Preuve du lemme 2.8 :

- Il faut d'abord écrire

$$\tilde{Q}(f, < \Theta_k | \{s_j, s_i\} >) = \tilde{Q}(f, < \Theta_k | \{s_j\} >) + s_i(f(s_j) - f(\theta_\varphi))$$

- Nous en déduisons la première proposition du lemme en passant au maximum de l'expression que nous venons de trouver et en remarquant que la différence est indépendante des autres contrats que s_i, s_j, et θ_φ :

$$\begin{aligned} \forall s_m < s_j < s_i, \quad & \max(\tilde{Q}(f, < \Theta_k | \{s_j, s_i\} >)) = \\ & \max\{\tilde{Q}(f, < \Theta_k | \{s_j\} >) + s_i(f(s_j) - f(\theta_\varphi))\} = \\ & \max\{\max(\tilde{Q}(f, < \Theta_k | \{s_m, s_j\} >) + s_i(f(s_j) - f(\theta_\varphi)))\} = \\ & \max\{\max(\tilde{Q}(f, < \Theta_k | \{s_m, s_j\} >)) + s_i(f(s_j) - f(\theta_\varphi))\} \end{aligned}$$

- Pour la seconde proposition du lemme, il faut remarquer que $\theta_\varphi \in < \Theta >_{max}$, si $< \Theta >_{max}$ est la séquence de contrats optimale, donc par définition,

$$\forall s_i < \theta_\varphi, Q(f) = \max(\widetilde{Q}(f, < \Theta_k | \{s_i, \theta_\varphi\} >))$$

donne la qualité maximale, si nous poursuivons la récurrence jusqu'au dernier seuil, θ_φ.

2.6.3 Algorithme QUALITE

Nous avons vu que le problème $MAXQSF$ de maximisation de la qualité moyenne sur une intervalle de temps était d'une complexité réputée impraticable en général. Ici, nous traitons du cas particulier où l'intervalle de temps est grand. C'est-à-direque l'intervalle est de largeur supérieure à la somme de tous les seuils de la fonction constante par morceaux qui approxime le profil de performance. Dans cette configuration, la contrainte de somme disparait puisqu'il est possible de choisir tous les seuils comme contrat. Mais il reste à choisir bons contrats parmi l'ensemble de tous les seuils possibles car un choix comprenant tous les seuils ne donne pas forcément la qualité moyenne maximale. Le but de cette partie est donc de décrire un algorithme polynomial permettant de faire ce choix optimal de contrats.

Le principe de l'algorithme est de parcourir la liste des seuils de la fonction f. Pour chacun des seuils s_i et grâce à la récurrence que nous venons d'établir dans le paragraphe précédent, il est possible de calculer $\max(\widetilde{Q}(f, < \Theta_k | \{\theta_\varphi\} >)))$ en parcourant tous les rangs précédents, et garder la séquence de contrats optimale. Finalement au rang p ($\theta_\varphi = s_p$), la qualité intégrale maximale est obtenue et la séquence de contrats correspondante. Le principe est celui de la programmation dynamique (voir algorithme 2.2).

Complexité :

La complexité de calcul d'un rang est donc en $O(n)$. La complexité de l'algorithme est donc quadratique.

2.7 Approximation de la qualité moyenne optimale

Nous revenons ici sur l'intuition que nous étions faire sur une heuristique possible pour le problème $MAXQSF$. En effet, nous avions vu sur la figure 2.7, que calculer la qualité moyenne pour un nombre plus grands de contrats n'apportait pas beaucoup de précision sur la qualité moyenne maximale. Nous avons donc fait une série d'essais pour confirmer cette tendance.

Les résultats obtenus pour le calcul du maximum de la qualité moyenne ne tiennent pas compte jusqu'ici de la durée du calcul de cette qualité moyenne maximale. Cela peut devenir un sérieux inconvénient dans une application pratique de notre approche car

ALG 2.2 Algorithme de calcul de la qualité moyenne maximale dans le cas d'une fenêtre longue

Début
  Init:
  $\theta_1 \leftarrow s_1$ /co/ un seul seuil considéré /oc/
  $\tilde{Q}(f, < \Theta_k | \{s_1\} >) \leftarrow (t_e - \max(\theta_1, t_e)) f(\theta_\varphi)$
  pour i **de** 2 **à** p **faire**
    $Q_{temp} \leftarrow 0$
    pour j **de** 1 **à** i-1 **faire**
      si $(\max\{\tilde{Q}(f, < \Theta_k | \{s_j\} >) + s_i(f(s_j) - f(\theta_\varphi))\} > Q_{temp})$ **alors**
        $Q_{temp} \leftarrow \max\{\tilde{Q}(f, < \Theta_k | \{s_j\} >) + s_i(f(s_j) - f(\theta_\varphi))\}$
      fin si
      $\tilde{Q}(f, < \Theta_k | \{s_i\} >) \leftarrow Q_{temp}$
    fin pour
    $Q(f) \leftarrow \tilde{Q}(f, < \Theta_k | \{s_i\} >)$ /co/ $i = p$ en fin de boucle /oc/
  fin pour
  retourne $(Q(f))$
fin

nous avons prouvé que notre problème général était NP-dur! Heureusement, atteindre la solution optimale n'est pas une condition nécessaire pour appliquer notre méthode: un ensemble de contrats approchant la valeur optimale de la qualité moyenne sur l'intervalle de temps est suffisant. C'est la raison pour laquelle nous avons mené une série d'expériences destinées à estimer la "complexité pratique" de notre problème. Et le résultat est bien meilleur que nous ne l'espérions: pour tous les cas que nous avons étudiés , la qualité moyenne atteinte par le meilleur choix de 2 contrats était toujours à une distance inférieure à 2.75% de la qualité moyenne optimale.

Ces résultats concernent une famille de fonctions monotones croissantes qui approximent les profils de performance le plus souvent rencontrés dans les cas pratiques. C'est pourquoi nous pensons que ces résultats ont un grand intérêt pour la généralité du problème.

L'expérience est divisée en deux parties. Tout d'abord, nous avons étudié l'influence de la pente à l'origine du profil de performance sur la valeur de la qualité moyenne. Pour cela nous avons considéré une famille de fonctions, définie par l'équation qui suit, et où le paramètre "a" permet de contrôler la coubure de la fonction comme montré dans le graphique associé sur la figure 2.13.

$$f(t) = \frac{at}{(a-1)t + 1}$$

Chaque fontion ("a" variant de 1 à 205 par incréments de 5) est approximée par une fonction constante par morceauxde 50 seuils pour lesquels nous avons calculé la qualité moyenne maximale; nous avons également calculé l'erreur par rapport à cette valeur maximale quand des ensembles de contrats plus petits sont considérés. Nous nous repo-

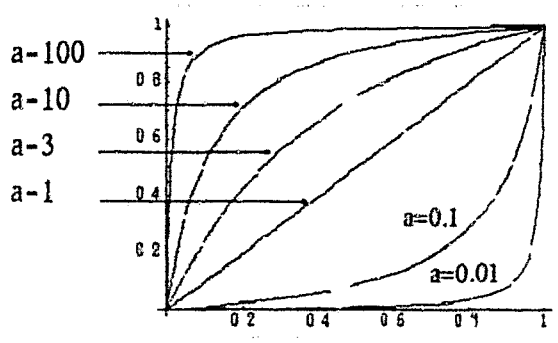


FIG. 2.13 – *Profils de performance pour les expériences*

nb contrats	1	2	3	4	5	6
0.1	*					
1	*					
5	10.83 %	0.23 %	*			
25	12.33 %	2.47 %	0.25 %	*		
45	11.59 %	2.71 %	0.59 %	*		
65	10.75 %	2.63 %	0.60 %	0.01 %		
85	9.97 %	2.44 %	0.53 %	*		
105	9.52 %	2.36 %	0.46 %	0.01%	*	
125	8.98 %	2.29 %	0.45 %	0.02 %	*	
145	8.55 %	2.22 %	0.39 %	0.02 %	*	
165	8.29 %	2.02 %	0.35 %	0.01 %	*	
185	7.96 %	1.83 %	0.28 %	*		
205	7.54 %	1.68 %	0.31 %	*		

TAB. 2.1 – *Erreur pour un nombre limité de contrats*

sons sur de la programmation sur de nombres entiers afin d'éviter le problème classique des nombres à virgule flottante et nous avons tiré avantage de la contrainte de somme pour limiter le temps de calcul. Le résultat est résumé dans le tableau 2.1 où a est la courbure représentée dans la première colonne, n est le nombre de contrats représenté dans la première ligne et * représente la solution optimale.

Pour $a \leq 1$, la qualité maximale est obtenue avec un seul contrat; ce qui est le résultat auquel nous nous attendions à cause du théorème 2.1. Dans les autres cas , il est clair que la meilleure qualité moyenne obtenue avec 2 contrats est très proche du meilleur résultat qu'on obtiendrait sans limiter le nombre de contrats.

Nous avons aussi étudié l'influence de la localisation temporelle de l'intervalle de temps (t_0), qui a été fixé à 0 pour la première expérience, alors que $t_e - t_0$ restait égal à 1. Cela peut être d'une grande importance pour beaucoup d'applications quand du temps est disponible avant l'arrivée de l'évènement hostile. Ici encore, comme nous le montrons sur la figure 2.14, restreindre le temps de délibération au calcul de 2 contrats seulement

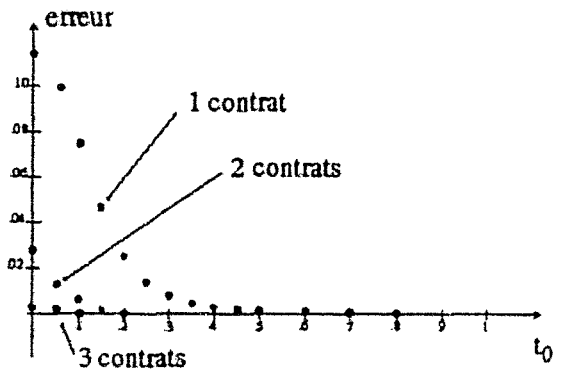


FIG. 2.14 - *Influence de t_0 sur le nombre de contrats*

donne des résultats très précis (même si un seul contrat donne rapidement de très bons résultats).

Dans cette expérience, la courbure a été mise à 51 et, alors que t_e était augmenté, "l'heuristique de la contrainte de somme" devenait moins efficace; nous avons alors manipulé une fonction constante par morceaux de seulement 30 seuils pour garder un temps calcul des optima raisonnables.

2.8 Conclusion

Nous venons de présenter une méthode d'optimisation hors ligne de l'allocation de temps à un algorithme à contrat de façon à obtenir la meilleure réponse moyenne face à un événement hostile survenant uniformément sur un intervalle de temps. Nous avons appelé ce problème MAXQSF. Jusqu'alors, aucune méthode analytique générale n'a été trouvée pour résoudre le problème de maximisation de la qualité moyenne sur un intervalle de temps. C'est la raison pour laquelle nous avons proposé des solutions pour des profils de performance discrets. Cette restriction n'est pas un inconvénient car :

- une représentation discrète tabulaire des profils de performance n'est pas rare
- la qualité moyenne résultante d'un profil de performance discret peut être aussi proche que l'on veut d'un profil de performance continu donné.

Nous avons montré que le problème MAXQSF était NP-difficile en général et quadratique dans le cas où l'intervalle est assez grand. Ceci, malheureusement, n'est pas fréquemment rencontré dans les applications réelles. Malgré cela, les expérimentations que nous avons menées nous ont conduit à la constatation que la "complexité en pratique" de ce problème était assez faible, ce qui rend l'utilisation de cette approche possible dans le cadre d'applications temps-réel. Une courte étude théorique plus complète afin de trouver une explication à ce comportement est envisageable.

Notons que cette approche peut être vue comme une alternative à la transformation d'un algorithme anytime à contrats en un algorithme anytime interruptible proposée par ZILBESTEIN et RUSSEL [ZR96]. En effet, la qualité moyenne pour un algorithme à contrats

A peut-être définie par :

$$\frac{1}{t_e - t_0} \int_{t_0}^{t_e} f^*(t)\, dt$$

où f^* est la qualité d'un algorithme interruptible A^* associé à A par la relation $A^*(t) = A(\theta_i)$ où θ_i est le dernier contrat exécuté à l'instant t.

ZILBESTEIN et RUSSEL [ZR96] donnent une construction simple pour transformer un algorithme à contrats A de qualité f en un algorithme interruptible A^* de qualité f^* tel que $f^*(4t) \geq f(t)$. La première différence avec notre étude est que nous considérons le cas d'un intervalle de temps donné sur lequel il faut optimiser la qualité moyenne. Dans notre cas, cela mène à la construction de A^* qui optimise la qualité moyenne. La seconde différence est que notre approche permet d'utiliser des algorithmes à contrats dont la longueur des contrats est déterminée par des méthodes, de durée fixe chacune, alors que ZILBESTEIN et RUSSEL supposent que la longueur de chaque contrat peut être choisie arbitrairement.

Notons que la probabilité d'occurrence de l'événement hostile a été supposée uniforme, car nous supposions n'avoir aucune information sur son arrivée. Il existe des situations où la probabilité n'est pas constante, où plus d'informations sont disponibles *a priori* . Une étude pourrait se concentrer sur ce point.

Il faut noter à ce sujet un travail de ZILBERSTEIN, CHARPILLET et CHASSAING [ZCC99] qui traite de l'optimisation de la qualité d'une séquence d'algorithmes à contrat. La dernière partie concerne le cas d'une date limite stochastique connue. Les auteurs proposent une solution générale utilisant la programmation dynamique pour des profils de performance discrets. Il est intéressant de comparer ce travail au nôtre dans le cas particulier où la date limite stochastique est une distribution uniforme de probabilité sur un intervalle $[t_0, t_e]$. Notre avis est que nous poursuivons le même objectif : trouver une séquence optimale de contrats pour maximiser la qualité à l'instant où l'interruption apparaît. Les principales différences sont :

- les auteurs de [ZCC99] considèrent une incertitude sur la qualité. De ce point de vue, le cadre est plus général.
- ils utilisent une fonction d'utilité dépendante du temps là où nous utilisons la qualité de la sortie de l'algorithme à contrat.
- la situation présentée dans ce chapitre est un cas particulier de celle décrite dans leur travail, dans laquelle il n'y a pas d'information sur la date d'occurrence de l'interruption. La méthode que nous proposons pour résoudre le problème est assez différente : quand on n'a aucune information sur la date limite/interruption, il est nécessaire de définir une politique de d'ordonnancement qui convienne à toutes les dates limites possibles sur l'intervalle. Par conséquent, la maximisation de la qualité moyenne est la meilleure stratégie possible, par définition, c'est à dire la meilleure réponse sur la moyenne des dates limites possibles sur tout l'intervalle.

Une étude intéressante consisterait à appliquer une distribution uniforme avec l'approche de ZILBERSTEIN, CHARPILLET et CHASSAING et de voir si celle-ci tend vers le même

résultat, c'est à dire la qualité moyenne maximale. Il n'y a pas de preuve à ce jour que ces deux méthodes se rejoignent.

Une autre extension pourrait concerner notre critère d'évaluation, qui est la qualité moyenne. Même si ce critère assure la meilleure chance de survie d'un point de vue statistique, il peut en exister d'autres, qui pourraient par exemple prendre en compte le nombre de contrats (pour limiter l'effet d'*overhead* en prenant peu de contrats).

Jusqu'alors, nous avons modélisé et remplacé l'algorithme anytime que nous considérions par son profil de perfomance et avons simulé son comportement par cet intermédiaire. Afin de valider notre étude, ainsi que d'autres approches de raisonnement en temps contraint (raisonnement progressif, *flexible computation*, *design to time*, *imprecise computation*...), il est nécessaire d'appliquer ces techniques sur des algorithmes anytime réels.

Dans cet objectif, un atelier d'expérimentation pourrait être implémenté. Il aurait une vocation de démonstration, qui permettrait à un développeur de se rendre compte des techniques disponibles en raisonnement en temps contraint, et une vocation de d'évaluation afin de mesurer les conséquences de l'utilistation de telles techniques pour une application particulière. Une première étape pourrait être par exemple d'implémenter l'allocation de contrat pour la qualité moyenne sur un intervalle de temps.

Mais pour réaliser cette plate-forme, il faut tout d'abord que nous nous intéressions à la conception des algorithmes anytime. D'après nos lectures, cette question semble avoir été quelque peu négligée au bénéfice des recherches sur les techniques d'ordonnancement des algorithmes anytime pour l'optimisation de leur qualité (finale ou moyenne). "Comment fabrique-t'on les algorithmes anytime?" C'est la question que nous nous sommes posée avec le regard d'un programmeur qui découvre l'algorithmique anytime. Notre objectif serait d'aboutir idéalement à une méthode de construction des algorithmes anytime. La conception des algorithmes anytime est donc l'objet de mise en question que nous proposons et le sujet du chapitre qui suit.

Chapitre 3

Contribution à la conception d'algorithmes anytime

> Dans le jeu du pile ou face, si on tire le côté face cent fois à la suite, cela nous apparaît extraordinaire, parce que le nombre presque infini de combinaisons qui peuvent apparaître à partir de cent lancers sont divisés en séquences régulières, ou dans lesquelles on peut facilement estraire une loi, et en séquences irrégulières, qui sont incomparablement plus nombreuses. [*Un essai sur les probabilités*, Pierre-Simon, Marquis de Laplace, 1819]

Dans l'étude sur la qualité moyenne présentée dans le précédent chapitre précédent, nous avons expérimenté notre approche uniquement sur des profils de performance. Ces profils de performance étaient des fonctions générées à partir d'une équation et non à partir d'un algorithme à contrat réel. Il peut sembler naturel de se demander si les profils de perfromance sont réalistes, c'est-à-dire représentatifs des cas courrants rencontrés dans les applications réelles. C'est la raison pour laquelle nous nous sommes intéressés à la construction de profils de performance à partir d'algorithmes réels. Plus généralement, nous avons décidé de nous focaliser sur la conception d'algorithmes anytime et sur les problèmes que cela peut poser. C'est aussi une façon de nous assurer du réalisme des profils de performance convexes (avec "increasing returns").

Il existe des conseils pour la construction des algorithmes anytime et nous les avons présentés dans le premier chapitre. Mais, à notre connaissance, beaucoup de solutions qui utilisent les algorithmes anytime sont des techniques *ad-hoc*, très spécifiques à l'application (voir par exemple [Sal98], ou [dV97]...). En fait, les efforts dans le domaine de l'algorithmique anytime semblent porter sur la gestion de ces algorithmes.

Toutefois, nous pouvons noter une contribution de GRASS et ZILBERSTEIN concernant une approche générale de la conception des algorithmes anytime [GZ96]. Ces travaux consistent en une plate-forme qui permet une "standardisation" de la conception et de la manipulation des algorithmes anytime. Cette application permet la génération systématique de profils de performance et leur gestion dans le cadre de l'optimisation de la qualité de sortie. Différentes strategies de gestion sont ou peuvent être implantées. Le

principal objectif est de fournir un outil standard à la communauté des chercheurs travaillant sur l'algorithmique anytime sous la forme de bibliothèques de *fonctions anytime*, (c'est-à-dire algorithme + profil de performance) et d'une application, ceci permettant de se concentrer sur la gestion des algorithmes.

Une hypothèse faite dans le travail que nous venons de citer est que le critère de qualité et la fonction de génération des entrées sont fournis par l'utilisateur de la plate-forme. Il paraît évident que le concepteur d'un algorithme est la personne la mieux placée pour connaître le comportement de l'algorithme et l'utilisation qu'il va en faire. Mais une petite série d'expériences à propos de ces deux paramètres nous ont montré que ces deux questions n'étaient pas si évidentes et nous imaginons que cela peut être le même problème pour le commun des programmeurs. C'est la raison pour laquelle notre étude présentée dans ce chapitre a tout d'abord consisté à montrer les difficultés dans le choix d'un critère de qualité et le choix des entrées représentatives pour la construction d'une carte de qualité qui reflète le réel comportement futur de l'algorithme. Dans chaque partie de cette étude, nous ne donnons pas de méthode générale, mais nous essayons de mettre en évidence certains "pièges" à éviter et de donner quelques conseils pour la construction des profils de performance de mainère statistique. Cette étude peut être considérée comme un complément à l'approche de GRASS et ZILBERSTEIN et se situe en amont de l'utilisation de leur plate-forme.

Nous ne faisons ici que résumer les principaux résultats obtenus dans cette étude. La plupart des expériences et résultats détaillés sont exposés en Annexe A et des compléments en Annexe C.

3.1 Principaux résultats

Nous nous attachons ici à la construction de profils de performance pour des algorithmes anytime. Dans un premier temps, il est nécessaire de définir une critère de qualité pour l'évaluation de la performance de l'algorithme en fonction du temps. il existe alors plusieurs cas de figures en ce qui concerne l'allure du profil en fonction de la dépendance aux données en entrée de l'algorithme :

- Si la qualité ne dépend pas du tout de la donnée en entrée, alors le profil de performance correspond au profil de performance analytique, c'est-à-dire celui défini par une formule analytique. Il ne se pose aucun problème de prédiction, mais ce cas de figure est malheureusement très rare.
- Si la qualité varie beaucoup avec la donnée en entrée, il devient plus difficile voire impossible de construire un profil de performance car la carte de qualité est trop dispersée et la variance de la qualité pour un temps de calcul donné trop grande.

C'est principalement ce dernier cas qui nous a intéressé. L'utilisation des profils de performances par intervalles (IPP) [Zil93] dans le cas où la donnée en entrée influence fortement la qualité permet de tirer des données au hasard. Le profil de performance consiste alors à encadrer par le haut et le bas la carte de qualité obtenue. L'épaisseur

de la bande encadrée détermine la capacité de prédiction et nous avons tout intérêt à ce qu'elle soir la plus étroite possible.

Il pourrait être également envisagé de faire dépendre la qualité de la sortie d'une qualité d'entrée ou de tout autre caractéristique de la donnée en entrée de l'algorithme permettant de discriminer différents groupes de donnée. Il est alors possible de construire des profils de performances conditionnels qui permettent alors une prédiction plus fine.

Le problème est que pour beaucoup de problèmes NP-durs en général (ceux très nombreux qui sont polynomiaux en moyenne), la qualité pour un temps de calcul donné dépend fortement de la donnée et peut provoquer des comportements extrêmes en terme de qualité et très éloignés du cas moyen. Nos expérimentations montrent même que faire dépendre la qualité de sortie de la qualité de l'entrée ou d'une méthode de génération des entrées particulière ne suffit pas à obtenir des profils de performance conditionnels très prédictifs (voir Annexe A). Il y a donc un piège à tirer des données au hasard pour construire un profil de performance dans cette situation.

C'est la théorie de la complexité de Kolmogorov [LV97] qui nous apporte un éclairage sur la question. Cette théorie, dont il n'est pas nécessaire de comprendre le détail pour la comprendre et en appliquer les principes, permet de formaliser la notion d'aléatoire et fait un lien entre la complexité des données (grossièrement le taux de hasard contenu dans la donnée) et la complexité des algorithmes.

En particulier, la théorie de la complexité de Kolmogorov nous permet de montrer que, même si la plupart des données (sous-entendu sur l'ensemble des données de même longueur) sont aléatoires et donnent des comportements polynomiaux en temps de calcul, ce sont les données fortement structurées qui ont la plus forte probabilité d'apparition dans la réalité (grâce à la mesure de Levin). Ainsi, si on tire au hasard, c'est à dire si on suit une probabilité d'apparition uniforme, on a de forte chances de tomber sur un cas moyen du point de vue algorithmique. Mais en réalité, l'apparition des données ne suit pas une loi uniforme. Comme les données structurées ont une forte probabilité de générer les cas pires (théorèmes de Li et Vitànyi et de Vlasie), il y a de fortes chances, en tirant des données au hasard, "d'oublier" des cas extrêmes sur le profil de performance, très éloignés du cas moyen.

La proposition que nous faisons est qu'il faudrait pouvoir spécifier les données afin d'obtenir des cartes de qualité représentatives de la réalité. Le travail que nous avons effectué sur les exemples d'algorithme de tris, qui a permis de mettre en évidence des familles de données aux comportements différents, doit être poursuivi sur chacun des problèmes posés. Il existe d'ailleurs des études dont les résultats permettent de définir une méthode de génération automatique des entrées "difficiles" [Vla95]. Cela reste un travail difficile de pouvoir spécifier les données de cette façon.

3.2 Conclusion

Dans ce chapitre, nous nous sommes concentrés sur les problèmes pratiques rencontrés lors de la construction des algorithmes anytime. Des considérations sur des exemples simples, comme le problème du voyageur de commerce nous ont fait douter de la facilité de construction des profils de performance. C'est la raison pour laquelle nous avons adopté le point de vue de l'utilisateur de ces algorithmes, une approche naïve mais critique afin d'essayer de résoudre ces problèmes de conception.

Dans cette étude, nous avons mis en évidence deux questions qui ne sont pas résolues de façon générale dans la majorité des travaux concernant la conception des algorithmes anytime :

- choisir un critère de qualité pertinent
- collecter les données réalistes pour construire les cartes de qualité et les profils de performance

Ces problèmes sont considérés comme étant du ressort de l'utilisateur des algorithmes anytime, ce qui peut sembler naturel à première vue. En effet, ces deux questions sont très dépendantes du problème posé.

Mais nous avons choisi de nous concentrer sur ces questions, car nous estimons qu'il existe des "pièges" qu'il est possible d'éviter. Grâce à cette approche expérimentale de la conception des algorithmes anytime, nous avons mis quelques uns de ces pièges en évidence à l'aide d'exemples suffisamment simples et représentatifs des problèmes usuels (tri et voyageur de commerce) et avons tenté de donner quelques conseils.

Quant au choix d'un critère de qualité, nous avons vu qu'il doit être spécifique à l'algorithme et pas seulement au problème résolu. En effet, ce critère peut avoir un sens ou le perdre selon la méthode de résolution du problème employée. Mais il dépend évidemment aussi du but visé, particulièrement du type de résultat intermédiaire que nous voulons obtenir. En effet, le résultat d'un algorithme est utilisé dans un certain contexte et peut être aussi réutilisé par un autre algorithme en tant qu'entrée. Le choix d'un critère de qualité est donc un compromis entre ces deux points de vue.

Le problème de la construction d'un profil de performance n'est pas triviale non-plus. Quelques considérations sur des cas réalistes nous montrent que le choix aléatoire des entrées est rarement la bonne méthode pour obtenir des profils de performance qui prédisent efficacement un comportement réaliste de l'algorithme. Nous proposons d'utiliser la complexité de Kolmogorov pour confirmer ce fait. Cette théorie montre que les instances structurées, c'est-à-dire non-aléatoires, sont les plus fréquentes en réalité. Cette étude a montré qu'il est nécessaire de faire une analyse fine de l'algorithme et de la fonction de génération des entrées avant de générer la carte de qualité.

Même si nous ne produisons pas de méthode complète pour construire des profils de performance, notre point de vue est que la théorie de la complexité de Kolmogorov peut être une bonne approche pour l'analyse des données et la génération des profils de performance. Beaucoup d'études sur la génération d'instances "difficiles" au sens combinatoire

et la spécification des instances en fonction de leur complexité de Kolmogorv sont en cours (voir par exemple celle de VLASIE [Vla97] sur le problème 3-COL). L'analyse du comportement des algorithmes et de la complexité de Kolmogorov des instances est très dépendante du problème posé. Mais l'avantage de la complexité de Kolmogorov est qu'elle permet l'analyse individuelle des instances de nos problèmes, alors que nous ne disposions auparavant que de résultats statistiques.

Nous pensons donc que la construction d'algorithmes anytime et en particulier de leurs profils de performance n'est pas aisée en général, mais qu'il est possible pour un "débutant" d'éviter certains pièges à condition d'utiliser la théorie de la complexité de Kolmogorov.

Conclusion

Conclusion générale

La première contribution de cette thèse concerne un problème d'ordonnancement d'un algorithme anytime à contrat. La situation que nous présentons est celle de l'arrivée d'un événement sur un intervalle de temps. Le problème était de savoir quelle pouvait être la meilleure réponse à donner si cette événement peut venir interrompre les calculs avec une probabilité uniforme sur tout l'intervalle de temps. Nous avons donc proposé une stratégie qui consiste à relancer l'algorithme anytime à contrat afin d'assurer les meilleures chances de "survie" sur l'intervalle. Par définition, c'est la maximisation de la moyenne des qualités qui donne ce résultat. L'objectif est alors de calculer quelles valeurs auront les contrats successifs et combien seront nécessaires pour atteindre le maximum de qualité moyenne.

Nous apportons une solution analytique pour des profils de performance convexes consistant à maximiser la qualité moyenne pour un seul contrat. Une équation définit la valeur de ce contrat qui est supérieur à la moitié de l'intervalle. Pour les profils de performances concaves, il est nécessaire de lancer au moins deux contrats pour obtenir la qualité moyenne maximale. Nous n'avons pas, jusqu'alors, de solution analytique dans le cas général.

Nous avons proposé d'approximer le profil de performance continu par une fonction constante par morceaux. Ce choix nous permet de choisir les contrats dans l'ensemble des seuils de la fonction et d'obtenir ainsi un problème de combinatoire. Ce problème, que nous avons baptisé $MAXQSF$ se révèle être NP-dur en général. Dans le cas particulier où l'intervalle de temps est assez grand (pour contenir la somme des seuils), nous proposons un algorithme quadratique. Ce cas où la fonction est très concave par rapport à la longueur de l'intervalle est assez rare à notre avis et nous a poussé à mener quelques expérimentations afin d'étudier la mise en pratique de notre approche. Des essais effectués sur des profils de performance de formes variées nous ont permis de constater que l'erreur faite en bornant (par 3) le nombre de contrats était très faible (de l'ordre du pourcent) par rapport à la qualité moyenne maximale.

Nous avons ensuite tenté d'implanter des algorithmes anytime et notamment de construire des profils de performance pour ces algorithmes. Des tentatives sur des exemples d'algorithmes simples ont mis en évidence un certain nombre de difficultés.

Notre seconde contribution a donc porté sur les problèmes pratiques de choix d'un

critère de qualité et des entrées nécessaires à la construction d'un profil de performance. Même si ces problèmes peuvent sembler très liés à l'application visée, ils semblent également négligés dans la plupart des travaux concernant l'algorithmique anytime. Des solutions *ad-hoc* sont souvent proposées.

Dans un premier temps, nous avons montré par des exemples d'algorithmes de tri que ces deux questions pourtant fondamentales, ne sont pas faciles. Il est tout d'abord nécessaire de choisir judicieusement un critère de qualité. Il est possible de produire des comportements en qualité très différents selon l'algorithme et le critère utilisé. Certains critères de qualité ne conviennent d'ailleurs pas à certains algorithmes même si le problème traité reste le même.

Ensuite, la variation de la taille des entrées, de la manière de générer les données (de façon structurée ou aléatoire) ont notamment provoqué l'élargissement des cartes de qualité obtenues, mettant en évidence les problèmes liés à la génération aléatoire des entrées. La théorie de la complexité de Kolmogorov nous a alors apporté un éclairage sur le choix des entrées pour la construction du profil de performance. En particulier, les données "rencontrées dans la réalité" ont une forte probabilité d'être structurées et non-aléatoires. Nous estimons donc que les difficultés rencontrées imposent de prendre un grand nombre de précautions lors de la conception des algorithmes anytime.

Perspectives

Nous avons montré que le problème $MAXQSF$ d'optimisation combinatoire était NP-dur en général, et quadratique quand l'intervalle était assez grand. Cette dernière situation est malheureusement très peu souvent rencontrée dans les cas réels. Toutefois, les expériences que nous avons menées laissent à penser que la complexité "en pratique" de ce problème reste très faible, ce qui rend cette approche envisageable pour des applications pratiques. Une rapide étude exploitant la forme concave des profils de performance permettrait de donner une explication théorique à ce comportement.

Une autre extension pourait se concentrer sur notre critère d'évaluation, c'est à dire la qualité moyenne. Même si ce critère donne les meilleurs résultats du point de vue statistique, il peut en exister d'autres, comme par exemple de considérer le nombre de contrats nécessaires. Le limiter permettrait de contrôler les temps morts dûs à la relance de l'algorithme, que nous avons négligés ici. Prendre en compte ces temps morts dans le modèle est, à notre avis, assez facilement envisageable.

En ce qui concerne l'hypothèse sur la distribution de probabilité d'occurence de l'événement interrupteur, supposant que nous n'avions aucune information, nous avons choisi prendre cette distribution uniforme sur l'intervalle. Il existe certainement des situations pour lesquelles on dispose de plus de connaissance *a priori* et l'étude du problème avec une distribution non-uniforme serait des plus intéressantes. Le calcul de la qualité moyenne selon cette nouvelle probabilité devient alors plus complexe.

Enfin, mais cela ne clot pas la liste des perspectives, cette approche de qualité moyenne

peut être envisagée avec plusieurs algorithmes à contrats différents qui se partageraient l'intervalle de temps. Ces algorithmes peuvent avoir un but unique auquel chacun contribue ou de multiples objectifs contribuant à améliorer la qualité globale ou un vecteur de qualité ce que propose MOUADDIB dans [Mou00]. Une première extension prenant en compte plusieurs algorithmes anytime à contrats indépendants est proposée dans [DMZ99].

Nous avons vu que la théorie de la complexité de Kolmogorov apportait un point de vue éclairant sur le choix des entrées représentatives des problèmes réels. Elle apporte également des informations sur le comportement des algorithmes (principalement ceux NP-complets). Des études permettant de définir des générateurs automatiques d'entrées complexes (au sens où elles entrainent les pires cas) existent déjà dans des cas bien particuliers (3-COL par VLASIE [Vla95]). Envisager de telles études pour chacun des problèmes apporteraient beaucoup à la construction des profils de performances réalistes. Il faut noter également un travail intéressant de THIÉBAUX, SLANEY et KILBY [TSK00] proposant une méthode permettant d'estimer la difficulté d'un problème d'optimisation (en général) en fonction de la forme des entrées. Cette étude est susceptible de fournir une piste intéressante pour la génération de données d'entrées ou l'analyse de données d'entrée déjà existantes.

Enfin, nous espérons que des efforts porteront sur le problème de l'implantation des algorithmes anytime. Comme notre étude du dernier chapitre et les nombreuses questions posées par les "nouveaux venus" dans le domaine de l'algorithmique anytime, il existe encore de nombreux remparts à la mise en pratique systématique de cette approche. Les travaux de GRASS et ZILBERSTEIN [GZ96] sont une bonne initiative en ce sens, et nous espérons que notre étude éclairera également cette voie. L'algorithmique anytime a besoin d'être popularisée et le dévoloppement d'une plate-forme applicative complète assistant le programmeur du début à la fin serait une très bonne manière de le faire. Nous imaginons une plate-forme qui se scinderait en trois parties :

- la première consisterait à aider le choix d'un critère de qualité et des entrées représentatives de l'application visée par le test de différents paramètres infuençant l'allure des cartes de qualité (taille des entrées, méthode de génération, etc.). Un tel outil serait l'aboutissement de notre application, utilisée dans le dernier chapitre.
- la deuxième partie se chargerait de la génération automatique des cartes de qualité à partir du critère de qualité et des entrées choisies à l'étape précédente. Il resterait alors à automatiser la construction du profil de performance d'un type choisi par l'utilisateur.
- Enfin, le troisième et dernier module concernerait le test des différentes techniques d'ordonnancement des algorithmes anytime, qu'ils soient interruptibles ou à contrats, mais aussi de la forme adoptée dans le raisonnement progressif et d'autres approches alternatives.

Nous espérons que ce projet ambitieux pourra être développé dans les années qui viennent.

Bibliographie

[Ad95] Martin Adelantado and Simon de Givry. Reactive/anytime Agents, Towards Inteligent Agents with Real-Time Performance. In *Workshop on Anytime Algorithms and Deliberation Scheduling*, Montréal, Canada, August 1995. IJCAI'95.

[BB92] R. J. A. Buhr and D. L. Bailey. *An Introduction to Real-Time Systems: From Design to Multitasking with C/C++*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1992.

[BD94] Mark Boddy and Thomas Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, June 1994.

[Bod91] Mark Boddy. *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments*. Ph. d dissertation, Department of Computer Science, Brown University, 1991.

[Bre79] D. Brelaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[CDM98] Rémy Card, Eric Dumas, and Franck Mével. *Programmation linux 2.0, API système et fonctionnement du noyau*. Edition Eyrolles, 2ème tirage edition, 1998.

[DB88] Thomas Dean and Mark Boddy. An analysis of time-dependant planning. In *AAAI'88*, Saint Paul, Minnesota, August 1988.

[DD99] Arnaud Delhay and Max Dauchet. Complexity of the maximization of the average quality of anytime contract algorithms. Rapport interne lifl-09-99, Laboratoire d'Informatique Fondamentale de Lille, USTL, September 1999. ftp://ftp.lifl.fr/pub/reports/internal/1999-09.ps.gz.

[DDTV98] Arnaud Delhay, Max Dauchet, Patrick Taillibert, and Philippe Vanheeghe. Optimization of the average quality of anytime algorithms. In *Workshop on Monitoring and Control of Real-Time Intelligent Systems*, pages 1-4, Brighton, UK, August 1998. ECAI'98.

[DDTV99] Arnaud Delhay, Max Dauchet, Patrick Taillibert, and Philippe Vanheeghe. Maximization of the Average Quality of Anytime Contract Algorithms over a Time Interval. In *Proc. of IJCAI'99*, pages 212-217, Stockholm, Sweden, August 1999.

[Del96] Arnaud Delhay. Complexité algorithmique et planification. Master's thesis, DEA d'informatique du LIFL, Lille, France, June 1996.

[Del98a] Jean-Paul Delahaye. *Jeux mathématique et mathématique des jeux.* Bibliothèque Pour la science, Paris, 1998.

[Del98b] Jean-Paul Delahaye. *Logique, informatique et paradoxes.* Pour la Science, diffusion Belin, Paris, 1998.

[DL93] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *In proceedings of 11th National Conference On Artificial Intelligence*, pages 221–224, Washington, 1993.

[DMZ99] Arnaud Delhay, Abdel-Illah Mouaddib, and Shlomo Zilberstein. Sequencing Multiple Contract Algorithms. In *Workshop on Scheduling and Planning meet Real-Time Monitoring in a Dynamic and Uncertain World*, pages 79–83, Stockholm, Sweden, August 1999. IJCAI'99.

[Dra87] Draft American Standard. *IEEE Trial-Use standard, Portable Operating System for Computer Environments.* Institute of Electrical and Electronics Engineers Inc., 2nd printing edition, November 1987.

[Dub98] J.-C. Dubacq. Introduction à la théorie algorithmique de l'information. Research Report 98-05, Ecole Normale Supérieure de Lyon, 15 January 1998.

[dV97] Simon de Givry and Gérard Verfaillie. Optimum Anytime Bounding for Constraint Optimization Problems. In *Workshop on Buliding Ressource-Bounded Reasonning Systems*, AAAI-97, 1997.

[DV00] Emmanuel Duflos and Philippe Vanheeghe. *Estimation et Prédiction, Eléments de cours et exercices résolus.* Collection Sciences et Technologies. Edition Technip, 2000. à paraître au second semestre 2000.

[Edi87] Editions Micro Applications, editor. *La Bible PC.* Livre Data Becker, 1987. ISBN 2-86899-137-8.

[FGS93] Christine Froidevaux, Marie-Claude Gaudel, and Michel Soria. *Types de données et algorithmes.* Ediscience International, 1993. ISBN: 2-7042-1217-1.

[Gil99] Franck van Gilluwe. *Programmation système, Ressources d'experts.* Campus Press, 1999. ISBN 2-7440-0559-2.

[GJ79] M.R. Garrey and D.S. Johnson. *Computers and Intractability: A Guide to the Theroy of NP-Completeness.* Freeman, San Francisco, CA, 1979.

[GL93] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 6(23):1491–1502, 1993.

[GL96] Alan Garvey and Victor R. Lesser. Design-to-Time Scheduling and Anytime Algorithms. *SIGART Bulletin*, 7(3), 1996.

[Gra96] Joshua Grass. Reasonning about computational resource allocation, An introduction to anytime algorithms. *Crossroads, the ACM student magazine, University of Massachussets*, September 1996. http://anytime.cs.umass.edu/ jgrass/school/papers racra.html.

[GZ96] Joshua Grass and Shlomo Zilberstein. Anytime Algorithm Development Tools. *SIGART Bulletin, Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2):20–27, April 1996.

[Hay96] Monson H. Hayes. *Statistical digital signal processing and modelling.* John Wiley and Sons, Inc., 1996.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syt. Sci. Cybern.*, 4:100–107, 1968.

[Hor87] Eric Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 3rd AAAI Workshop on Uncertainty in artificial Intelligence*, pages 429–444, Seatle, WA., August 1987. Association for Uncertainty in Artificial Intelligence.

[Hor88] Eric Horvitz. Reasoning under varying and uncertain ressource constraints. In CA. Morgan Kaufmann, San Mateo, editor, *Proceedings of the 7th National Conference on Artificial Intelligence*, Mineapolis, MN., September 1988.

[Hor90] Eric Horvitz. *Computation and Action Under Bounded Resources.* PhD thesis, Stanford University, 1990.

[HZ96] Eric A. Hansen and Shlomo Zilberstein. Monitoring Anytime Algorithms. *SIGART Bulletin, Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2), April 1996.

[JM95] David S. Johnson and Lyle A. McGeoch. *Local Search in Combinatorial Optimization*, chapter The Traveling Salesman Problem: A Case Study in Local Optimization, pages 215–310. E.H.L. Aarts and J.K. Lenstra, John Wiley and Sons, London, 20 November 1995. Preliminary version.

[Joh96] David S. Johnson. A Theorician's Guide to the Experimental Analysis of Algorithms. http://www.search.att.com/ dsj, 1996. Draft.

[KLV97] Walter Kirchherr, Ming Li, and Paul Vitányi. The Miraculous Universal Distribution. *Mathematical Intelligencer*, 19(4):7-15, 1997.

[Kor90] Richard E. Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42:189–211, 1990.

[Lew93] Donald Lewine. *POSIX Programmer's Guide, Writing Portable UNIX Programs.* O'Reilly & Associates Inc., March 1993.

[LV90] Ming Li and Paul M. Vitányi. *Kolmogorov Complexity and Its Applications.* Elsevier - MIT Press, Leeuven, 1990. pages 187-254.

[LV92] Ming Li and Paul M. Vitányi. Average case complexity under the universal distribution equals worst-case complexity. *Inf. Proc. Letters*, 3:145-149, 1992.

[LV97] Ming Li and Paul M. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 2nd edition edition, 1997.

[MC96] Philippe Morignot and François Charpillet. An Anytime Look at Task Planning. In Gerry Kellcher, editor, *Proceedings of 15th UK Planning and Scheduling SIG*, John Moores University, 1996.

[MHA+95] David J. Musliner, James A. Hendler, Ashok K. Agrawal, Edmund H. Durfee, Jay K. Strosnider, and C. J. Paul. THe Challenges of Real-Time AI. *IEEE Computer*, 28(1):58–66, January 1995.

[Mou93] Abdel-Illah Mouaddib. *Contribution au raisonnement progressif et temps-réel dans un univers multi-agents.* Thèse de doctorat, Université de Nancy I, 1993.

[Mou00] Abdel-Illah Mouaddib. Multi-criteria decision quality optimization as a Scheduling problem. In *ICTAI 2000*, 2000.

[MZ95] Abdel-Illah Mouaddib and Shlomo Zilberstein. Knowledge-Based Anytime Computation. In *Proceedings of 14th IJCAI*, pages 775–781, Montréal, Canada, 1995.

[MZ98] Abdel-Illah Mouaddib and Shlomo Zilberstein. Optimal Scheduling of Dynamic Progressive Processing. In *Proceedings of 11th ECAI*, pages 499–503, Brighton, UK, August 1998.

[PM96] John G. Proakis and Dimitris G. Manolakis. *Digital signal processing. Principles, Algorithms, and Applications.* Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Sal98] Thierry Salvant. *CAAM, Un Modèle d'Agent Anytime Coopératif pour l'Aide à la Décision en Temps Critique.* Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, 19 March 1998.

[SD89] S. Shekhar and S. Dutta. Minimizing response times in real-time planning and search. In *Proceedings of 11th IJCAI*, pages 238–242. IJCAI'89, 1989.

[SF96] Robert Sedgewick and Philippe Flajolet. *An Introduction to Analysis of Algorithms.* Addison Wesley Publishing Company, 1996.

[Ste92] Richard W. Stevens. *Advanced Programming in the UNIX Environment.* Professional Computing Series. Addison Wesley, 1992. ISBN 0-201-56317-7.

[TSK00] Sylvie Thiébaux, John Slaney, and Phil Kilby. Estimating the Hardness of Optimisation. In *Proceedings of European Conference on Artificial Intelligence*, Berlin, Germany, August 2000. ECAI'00.

[Vla95] Dan R. Vlasie. Systematic Generation of Very Hard Cases for Graph 3-Colorability. In *Proceedings of 7th IEEE ICTAI*, pages 114–119, 1995.

[Vla97] Dan R. Vlasie. The Very Particular Structure of the Very Hard Instances. In *Proceedings of 13th AAAI*, pages 266–270, 1997.

[ZCC99] Shlomo Zilberstein, François Charpillet, and Philippe Chassain. Real-Time Problem Solving with Contracts Algorithms. In *Proceeding of 16th IJCAI*, pages 1008–1013, Stockholm, Sweden, August 1999.

[Zil93] Shlomo Zilberstein. *Operational Rationality through compilation of anytime algorithms.* Ph. d dissertation, Computer Science Division, University of California, Berkeley, CA, 1993.

[Zil96] Shlomo Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, pages 73–83, 1996.

[ZM99] Shlomo Zilberstein and Abdell-Illah Mouaddib. Reactive Control of Dynamic Progressive Processing. In *Proceedings of 16th IJCAI*, pages 1268–1273, Stockholm, Sweden, August 1999.

[ZR93] Shlomo Zilberstein and Stuart J. Russel. Anytime Sensing, Planning and Action : A Practical Model for Robot Control. In *Proceeding of the 13th IJCAI*, Chambery, France, 1993.

[ZR96] Shlomo Zilberstein and Stuart J. Russel. Optimal composition of real-time systems. *Artificial Intelligence*, 82:181–213, 1996.

Table des figures

Liste des tableaux

Annexe A

Contribution to the design of anytime algorithms

> In the game of heads and tails, if head comes up a hundred times in a row, then this appears to us extraordinary, because the almost infinite number of combinations that can arise in a hundred throws are divided into regular sequences, or those in which we observe a rule that is easy to grasp, and into irregular sequences, that are incomparably more numerous |*A philosophical essay on probabilities*, translation from French, Pierre-Simon, Marquis de Laplace, 1819|

In the study of the average quality presented in the previous chapter, we experimented our approach on "simulated" anytime contract algorithms, that is we considered that algorithms were only represented by their performance profiles. These performance profiles were functions that we generated. It could seem natural to ask ourselves if these performance profiles are real, *i.e.* representative of the current cases encountered in real applications. That is the reason why we are interested in constructing our performance profiles for algorithms that we chose, and why we are not satisfied anymore with "artificial" ones. More generally, we decided to focus on the design of anytime algorithms and on the problems that it could cause. It is a way to ensure realism and to verify if the convex performance profiles (with "increasing returns") we talked about can be met in reality.

There exist advices to build anytime algorithms and their performance profiles that we presented in the state of the art. But, as far as we know, many solutions using anytime algorithms are *ad hoc* techniques, very application specific (see for example |Sal98|, or |dV97|...). In fact the efforts of the community are essentially focused on monitoring anytime algorithms.

Nevertheless, we have to note the contribution of GRASS and ZILBERSTEIN concerning a general approach of design of anytime algorithms |GZ96|. This approach consists in a platform that enables standardization of design and manipulation of anytime algorithms. This application enables the systematic generation of performance profiles and the monitoring of the algorithms in the framework of the optimization of the output quality.

Differents strategies of monitoring are or can be implemented. The main goal is to supply a standard tool to the research community working on anytime algorithms under the form of libraries of *anytime functions* (algorithm + performance profile) and of an application. This allows to concentrate on monitoring.

An assumption made in the previous work is that the quality criteria and the generation function of instances are supplied by the platform user. It is obvious that the designer of an algorithm is the best person to know its behavior and the use that will be made of it. But a few simple experiments showed us that these two questions are not obvious and we imagine that it would be the same problem for a common programmer. That is why our study in this chapter will first consist in showing the difficulties in choosing a quality criterion and in choosing the representative instances to build a quality map that reflects the real future behavior of the algorithm. In each part of this study, we do not supply general methods but try to exhibit some pitfalls to avoid and pieces of advice to give to the "beginner" in designing anytime algorithms. This study can viewed as a complement to the approach of GRASS and ZILBERSTEIN and takes effect just before the use of their platform.

First we will exhibit simple but important questions about the two points we quoted above. Next we address the question of choosing the quality criterion, then the problem of generating realistic instances. These two sections will be illustrated with simple but representative algorithms. Appendix B also addresses the problem in a technical and practical way by presenting how we implemeted the tools for measuring computation time and acquiring the quality maps.

A.1 Is it so easy to design anytime algorithms?

The main advantage of anytime algorithms is to enable prediction of their performance over time. To value this performance, we have to define a quality criterion and to construct a performance profile by selecting inputs to trace it. In this section, we will discuss choices made in the litterature [Zil93][HZ96] to do this work with the particular example of the Traveling Salesman Problem (TSP).

The TSP presented in [Zil93][HZ96] is the problem of finding an optimal tour (the shortest hamiltonian cycle) in a totally connected graph. The cost of a path is defined with the euclidian distance. The algorithm proposed is the "randomized improvement tour", inpired of the 2-OPT algorithm [JM95], that starts from a complete tour (a random permutation of the cities) and randomly choose two vertice. These vertice are exchanged if the quality increases after this operation.

Quality criterion

Let us first consider the choice of a quality criterion. A direct approach lead us to choose the following quality criterion, that values the distance to go to the optimum [HZ96]:

$$q(t) = \frac{Cost(optimal\_tour)}{Cost(tour(t))}$$

Several problems are connected to this approach. The first one is that this quality is not recognizable because it can not be reasonably computed at run-time. As a matter of fact, you have to compute the cost of the optimal or to know it in advance. The problem with the TSP is that it is a NP-complete problem for which there exists no polynomial method to approximate the cost of optimal (without computing the optimal itself) with an error as small as we want [JM95]. The only solution is to know the optimal in advance and then comes the second problem. For the same reason, even if there are polynomial (but not so tractable in reality because they demand heavy equipment to solve one instance in several days!) local optimizations that gives good estimates (with no garantee), the optimal is still costly to find. Moreover, what is the interest in looking for an intermediate solution when we already have the optimal solution? One can argue that it is just to build quality maps and performance profiles, but how can we use them if we can not locate the quality of the beginning tour?

An alternative solution is given by the use of the following quality criterion that considers the improvement from the beginning of the algorithm.

$$q(t) = \frac{Cost(init\_path)}{Cost(tour(t))}$$

This criterion is usually adopted in papers that take the TSP as an example [Gra96] [Zil96]. The problem with this criterion is that there is no constant value to refer to and from an instance to another, the initial tour can be far from the optimal or close to the optimal. That could generate respectively a highly evolutive performance profile or a "flat" one (figure A.1). By using this type of non consistent performance profile (with wide variance), we loose prediction capacity.

A solution to this problem could be the use of the trajectory prediction method by considering at run-time the slope of the quality over time so far and predict the future behavior. For this method, it is necessary to assume that the quality have a "smooth" progression and no discontinuity. To conclude this part, it is clear that the choice of a good criterion is not so easy.

Performance profiles

It is clear that the proposed TSP algorithm can not be analysed to deduce an anlytic performance profile because the improvement is randomized. The remaining solution is the statistical study of the behavior by generating instances and building the corresponding quality map in function of time and, if necessary, the input quality.

In the case of the TSP, the method proposed to generate inputs is to randomly choose lots of them in the set of all possible instances. The numerous instances seems to be a garantee of the realism of the choice. As a matter of fact, users of this method hope to

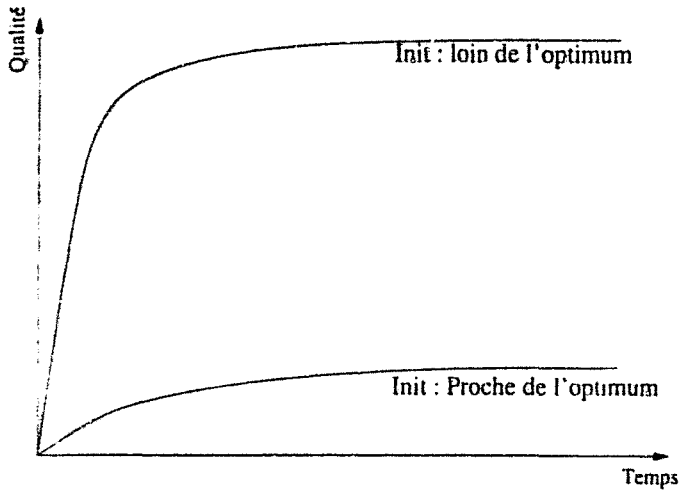


FIG. A.1 - *Two extreme behaviors for the same algorithm for TSP with a quality criterion that measures the gain regarding initial situation*

capture a large sample of the behavior of the algorithm. And the result is often statisfactory with a narrow variance around an average. But the number of cities compared to the size of the map can have a real influence on the behavior of the randomize improvement tour algorithm. The number of cities in [Zil93] and [Gra96] for example, is large and the dimensions (the number of points) of the map is of the same order. We experimented the algorithm on 10 cities randomly located on a map of 10000 x 10000 points. As shown on figure A.2, the small number of cities seems to lead to extreme behavior.

Intuitively, if the number of cities is large and if they are randomly chosen, then we have high probability to be in the average cases (because they are in large majority). If the number of cities is small, we have a higher chance to get "extreme" instances. We admit that anytime algorithms can be of poor interest in the case of 10 cities, but this shows that we need to take precautions in choosing our inputs in the set of all possible instances to build a predictive and realistic performance profile.

All these remarks lead us to the claim that design of anytime algorithms, especially the choice of the quality criterion and of the inputs, is not an easy job for reasons that could be independant of the goals of the designer and the use of the algorithm in its applicative context. That is why we propose a discussion about these two points to exhibit possible pitfalls and perhaps some piece of advice. This work is a preliminary approach to the use of GRASS and ZILBERSTEIN[GZ96] platform. The last part of this chapter will concern some practical problems of the implementation of anytime algorithms, especially software environment.

A.2 The choice of a quality criterion

We just saw with the TSP that the choice of the quality criterion that quantifies the performance of the result of the anytime algorithm can be a pitfall question. We chose sort

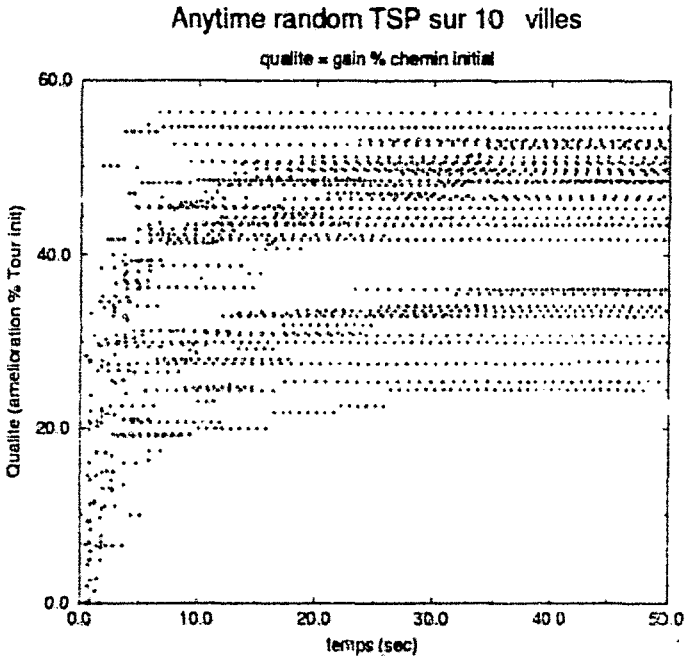


FIG. A.2 – *Behavior of "randomize improvement tour" TSP algorithm on instances containing 10 randomly chosen cities on a map of 10000 points by 10000 points*

algorithms to illustrate this question. Using such simple algorithms (polynomial in term of complexity) is not senseless. As a matter of fact, even the computation time of polynomial algorithms can be unacceptable for practical applications if the input instance is too large. Just consider sorting files in a huge database. The advantages of the sort algorithms are that they are simple enough to be analyzed and sufficient to isolate and exhibit problems in building anytime algorithms.

In this section, we consider the problem of choosing a quality criterion and especially regarding two aspects. The first aspect is the influence of the algorithm used to solve the same problem and consequences on the quality criterion. The second one is the problem of the input quality and how to choose a quality that caracterizes the input of the algorithm. We will precise the definition of the input quality.

As we do not consider the performance profile problem for the moment, we only try to build a *quality map*[GZ96], that is a brute trace of the behaviour of the output quality with time. Performance profiles are produced only after considerations and computation on a quality map.

A.2.1 Experimental results

In this section we explore several sort algorithms and observe the influence of the choice of a quality criterion.

A.2.1.1 Ordinary Selection Sort

First, we observe the ordinary selection sort[FGS93]. The principle of this is the most elementary way of sorting a list (Algorithm A.1):

- look for the smallest element of the list by sequentially exploring the list from the beginning to the end of the list and placing it at the first position.
- do the same work considering the remaining list, *i.e.* the original one minus the first (and smallest) element.
- do this until the remaining list is limited to one element.

ALG A.1 Ordinary selection sort [FGS93]

```
procedure ord-selection-sort
begin
  i ← 1
  while i < TABSIZE do
    j ← i
    for k from i + 1 to TABSIZE do
      if t[k] < t[j] then j ← k
    endfor
    t[j] ↔ t[i]
    i ← i + 1
  endwhile
end ord-selection-sort
```

In the worst case, the number of comparisons in this algorithm is clearly $\theta(n^2)$. The complexity in the average case is identical. As for the number of exchanges, it is $\theta(n)$ in the worst and the average case (see [FGS93]). To test this algorithm, we used a list of 14000 elements. This list is randomly generated.

At this time we observe the behavior of sort algorithms starting from a random list. The mixing algorithm (algorithm A.2)

- starts from a sorted list and randomly choose (with *rand* function in C) an element of the list to place in first position,
- continues with the remaining list, that is the original one minus the first element, and stops when the remaining list is a singleton.

For the ordinary selection sort, we chose a criterion that values the percentage of elements that are already sorted for sure. This criterion is very easy to compute and clearly takes constant time and is noted q_s. It consists in fact in counting the steps of the algorithm. So it really counts the sorted elements in the case of a sort like selection sort. In the case of an other sort that does not put at first the smallest elements in the first positions (like quicksort for example), it just reflects the progress of the algorithm to termination.

We can observe on figure A.3 that the quality of the input is 0% (no steps for the algorithm). We took 100 measure points and adjusted the number of iterations between

ALG A.2 Mix algorithm by swapping couples of elements

```
procedure mix_random
begin
  for idx from 1 to TABSIZE do
    taille ← TABSIZE − idx
    pos ← idx + random() modulo taille
    t[pos] ↔ t[idx]
  endfor
end mix_random
```

two measure points so as the algorithm can reach 100% of quality and terminate. The graph clearly confirms that the ordinary selection sort algorithm with the quality criterion chosen improves the quality of the output at each iteration. A trial with a measure point at each iteration would prove it completely.

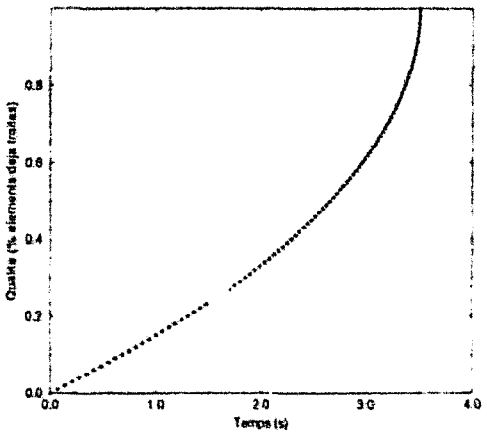


FIG A.3 - *Behavior of an ordinary selection sort algorithm on a list of 14000 randomly mixed integers - quality measures the percentage of already treated elements - 100 regular measure points*

Ordinary selection sort with this quality criterion is a good candidate to be an algorithm with a convex performance profile which can analytically treated in the framework of the average quality. It is an interruptible algorithm but can be used as a contract one. The behavior of this algorithm can be easily explained. At each step, the quality is incremented by 1 elementary unit ($1/TABSIZE$)). The more the algorithm goes to the end of the list, the shorter is the main loop because it only consists in searching the smallest element in the remaining list that is shorter and shorter. As a consequence, it takes fewer and fewer operations (and time) to increase quality.

The evolution of quality from 0 to 1 (or 0% to 100%) is natural as well. This criterion directly reflects the progress of the algorithm which sequentially sorts the list. At each loop, the algorithm has sorted the beginning of the list (before its current position in the

list) and it increases quality. This choice seems natural, but why choose this one rather than others? Let us explore other possible quality criteria.

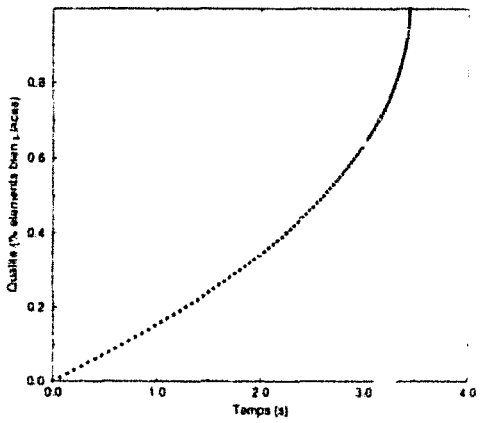


FIG. A.4 - *Behavior of an ordinary selection sort algorithm on a list of 14000 randomly mixed integers - quality measures the percentage of well positioned elements - 100 regular measure points*

Rather than evaluating the progress of the algorithm, we could count the number of elements that are in their final position. We called this criterion q_{bp}. As a matter of fact, the ordinary selection sort has a particularity that some of other sort algorithms do not have: it never moves an element that is in its final position. The explanation is the following:

- if the considered position contains an element in its final position, the search will be useless because this element is the smallest of the remaining list.
- if it is not the case, the unique possibility is that the element in the current position will be exchanged with the smallest one. The result of this operation is to move the smallest element to the right position and the other one from a wrong position to another position, that could sometimes be the final one.

The trial we did on a random list is on figure A.4. Except the quality criterion, the trial is done in the same conditions as the previous one. The choice of this criterion on a random list does not change the look of the graph, excepting the fact that the quality of the input list is rarely 0. That means that there are few elements in their final position in the list before the beginning of the sort, which is not surprising.

Then, we tested the criterion q_b that measures the proportion of direct neighbors that are in the right order. The main difference here is the quality of a random input that is close to 0.5 as shown on figure A.5. This time, this quality criterion does not seem as natural, but the results show an increasing graph again

Finally, a criterion q_d that measures the average distance between the current position of an element and its final position is used, multiplied by (-1) to get an increasing quality

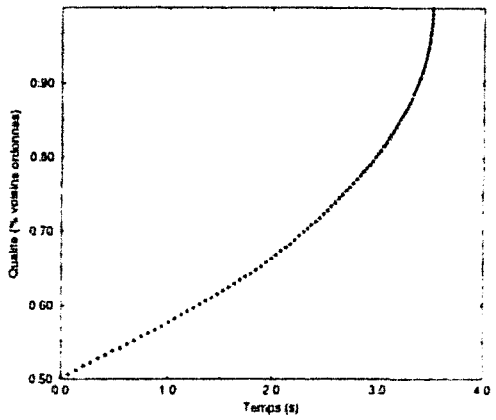


FIG. A.5 – *Behavior of an ordinary selection sort algorithm on a list of 14000 randomly mixed integers - quality measures the percentage of well positioned direct neighbors - 100 regular measure points*

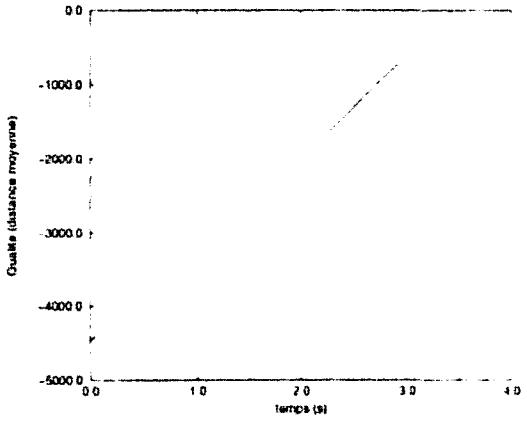


FIG. A.6 *Behavior of an ordinary selection sort algorithm on a list of 14000 randomly mixed integers - quality measures the distance between current position and final position of elements - 100 regular measure points*

criterion. The following formula defines q_d:

$$q_d(t) = -\frac{\sum_{i=0}^{TABSIZE} |tab[i] - i|}{TABSIZE}$$

Results on figure A.6 do not show the same curvature as for the other criteria. In fact, it is difficult to compare this criteria with the other beacuse q_d is not normalized. The difference with other criteria is that there is no absolute and constant value to compare with (with the size of the list). To normalize it, we could measure the decrease of the average distance compared to the initial average distance. But this initial value of average distance

changes with the input list. A way to ensure a quality function with values between 0 and 1 can be to divide q_d by the size of the list $TABSIZE$ once more. The multiplicative coefficient 2 is used because average distance is never greater than $\frac{1}{2}TABSIZE$. Finally the new q_d will be:

$$q_d(t) = 1 - 2\frac{\sum_{i=0}^{TABSIZE} |tab[i] - i|}{TABSIZE^2}$$

The results are similar to what we got on figure A.6. The difference is the value of the resulting quality (see figure A.7).

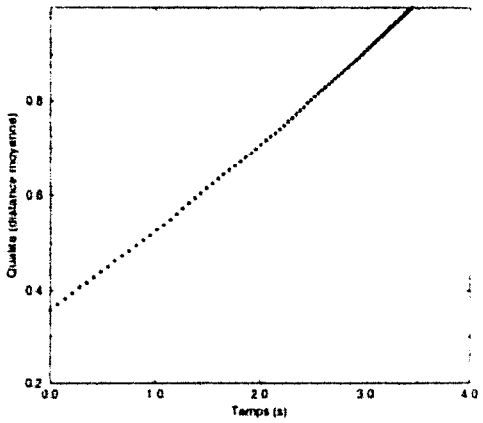


FIG. A.7 - *Behavior of an ordinary selection sort algorithm on a list of 14000 randomly mixed integers - quality measures the average normalized distance between current position and final position of elements - 100 regular measure points*

Note that two of these criteria are not very easy to compute: q_{bp} and q_d. To use these criteria, we have to assume that the list is composed of the naturals from 1 to 14000. Then the criteria are easy to compute. In the case of a list of reals, names, or everything else that is not comparable to a position in a list leads to difficulties in computation of quality. But we sometimes continue to use them to show the effects of these choices. To avoid penalizing this criteria compared to the others, the computation time of the quality is not included in the total time.

For all these criteria, the ordinary selection sort always increases quality. Should it always be the case for every sort algorithms? To answer this question, we did trials on other algorithms, the bubble sort and the quicksort with the same quality criteria.

A.2.1.2 Bubble sort

Now we compare these previous results with those of the bubble sort algorithm [FGS93]. This algorithm (A.3) is divided as follows:

- starting from the end of the list towards the first position, it exchanges two neighbors

if they are not in order (if the left one is greater than the right one)
- it goes to the next (right) position, and does the same working with the remaining list until the remaining list is a singleton.

For example, if the smallest element is at the end of the list, you have the impression that a bubble goes up to the first position like a bubble to the surface of water. The complexity of this algorithm in number of [illegible] is the same as the complexity of the ordinary selection sort, that is $\theta(n^2)$. The [illegible]ty in number of exchanges (and it seems natural regarding the principle of this algorithm) is worse than for the ordinary selection sort, that is $\theta(n^2)$ (see [FGS93] for details).

ALG A.3 Bubble sort[FGS93]

```
procedure bubble sort
begin
  i ← 1
  while i < n do
    for j from n to i+1 step -1 do
      if t[j] < t[j − 1] then t[j] ↔ t[j − 1]
    endfor
    i ← i + 1
  endwhile
end bubble sort
```

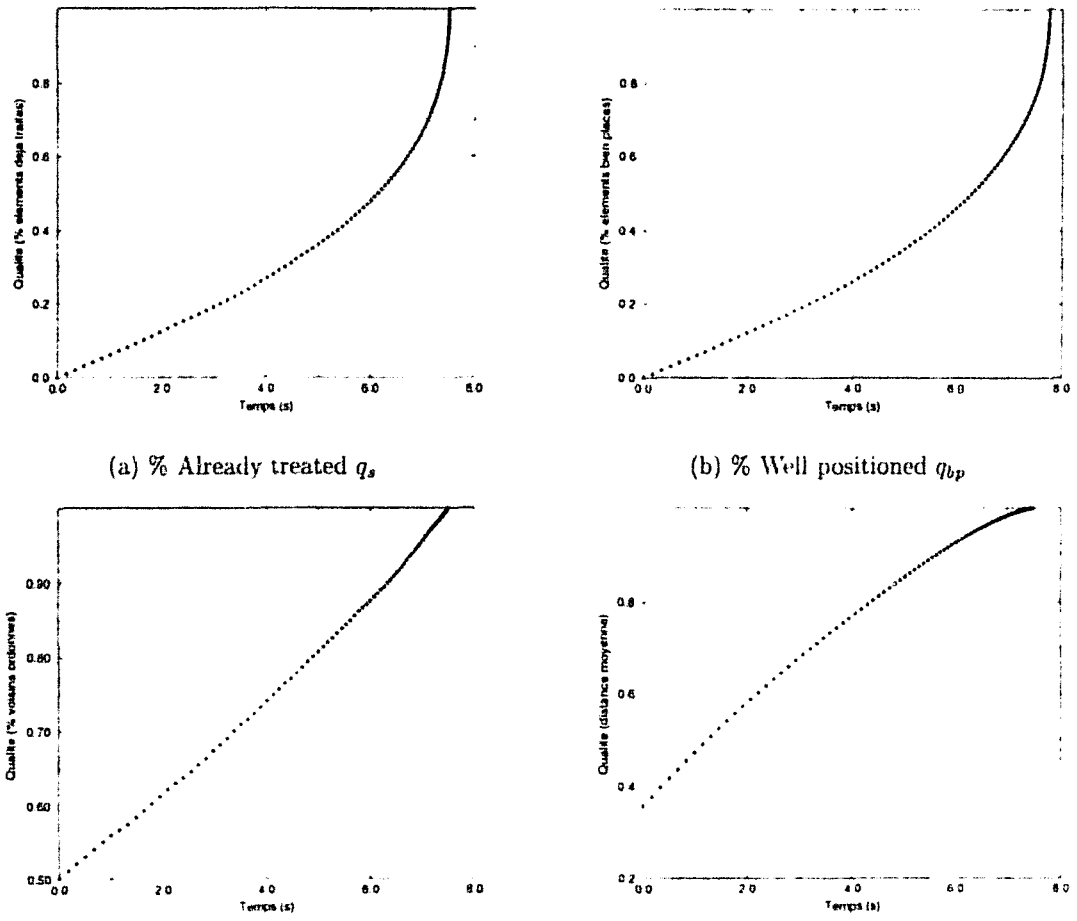
We did trials in the same conditions as described above for the ordinary selection sort. We tested the four criteria presented and the results on a random list are presented on figure A.8.

First of all, note that execution time is longer than for the ordinary selection sort. It is due to the number of exchanges on average that are more numerous for the bubble sort ($\theta(n^2)$) and the number of comparisons that is of the same order.

At first glance, no differences seem to appear between the behavior of the algorithms regarding the first two quality criteria. As for the percentage of ordered neighbors (q_v) and the average distance to the final position (q_d), it is different. For q_v, the graph is linear (in fact a little bit convex) for the bubble sort, when it was clearly convex for the ordinary selection sort. For q_d, the graph is concave when it seemed *quasi* linear before. It is not surprising to see differences between two algorithms that have the same ultimate goal but not the same method. More than differences between algorithm behavior, it is important to note that criteria may change the look of the final quality map for the same algorithm. It is clearly presented on figures A.8(a) and A.8(d).

A.2.1.3 Quicksort

Ordinary selection sort and bubble sort are algorithms that have the same total average complexity, that is $\theta(n^2)$. Quicksort [SF96] is different as its complexity on average is $\theta(n \log n)$ and its worst case complexity is $\theta(n^2)$ [FGS93]. The algorithm that sorts the



(a) % Already treated q_s

(b) % Well positioned q_{bp}

(c) % Well positioned neighbors q_v

(d) Average distance q_d

FIG. A.8 - *Behavior of a bubble sort on a random list of 14000 integers taken between 1 and 14000 - 100 measure points*

numbers in an array $a[g : d]$ is divided into two parts, where g is for the extreme left position and d for the extreme right one:

- partitions the array into two independent and smaller parts. The partitioning process puts the element that was in the last position in the array into its correct position, with all smaller elements before it and all larger elements after it. This *partitioning element* can be differently chosen. For example, it can be the element that was in the first position or can be randomly chosen.
- then sorts the two parts defined by this partition

A recursive version, where the partioning element is the one that was in the last position, is presented on Algorithm A.4.

ALG A.4 Quicksort [SF96]

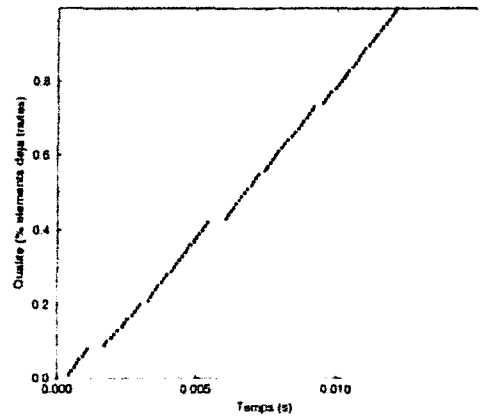
```
procedure quicksort(g, d: integer)
begin
  if d ≥ g then
    v ← a[g]; i ← d − 1; j ← g
    repeat
      repeat i ← i + 1 until a[i] ≥ v
      repeat j ← j − 1 until a[j] ≤ v
      t ← a[i]; a[i] ← a[j]; a[j] ← t
    until j ≤ i
    a[j] ← a[i]; a[i] ← a[d]; a[d] ← t
    quicksort(g, i − 1)
    quicksort(i + 1, d)
end quicksort
```

The results of trials with the four quality criteria are presented on figure A.9. Note that computation time is far shorter than for the previous algorithms. The average complexity of the quicksort is less than selection and bubble sorts. As usual, the first two criteria q_s and q_{bp} that are quite similar do not lead to different behaviors. The third one q_v gives similar results except for the beginning quality that is close to 50%. The last one, q_d, that mesures the average distance between the current and the final position of an element is very different and gives a graph with diminishing returns.

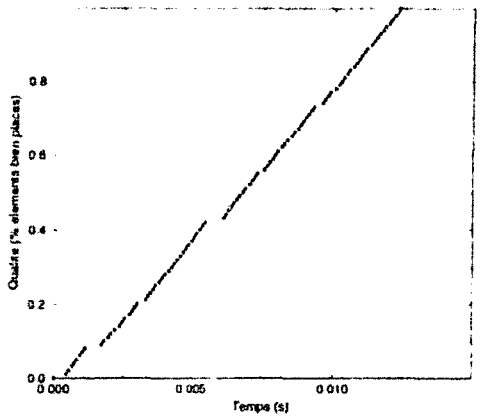
We can observe the characteristic behavior of this type of algorithm on all the graphs and especially on figure A.9(d). There are several slow increasing slope steps on this graph and some intervals before these steps. This is due to the natural recursive form of the algorithm. As a matter of fact, before increasing quality, that is before beginning the sort of a part of the list, the algorithm has to partition the list into two parts. This results in intervals with no (apparent) progress because we take measure of computation time after each recursive call of *quicksort*. In fact, during these intervals there is progress, even if it is not measured, as the algorithm places the smallest elements to the left of the partition element and the largest ones to its right. As a consequence, the elements of the list are closer to their final position. But when the algorithm is treating a branch, it partitions (and sorts) smaller and smaller lists and it results in the steps we observe.

These properties of the quicksort can be observed on the other figures (A.9(a), A.9(b) and A.9(c)). As a matter of fact, these graphs seems to be stepwise linear. The steps are the same ones as those quoted before, but on the contrary of q_d, there is no improvement of quality during partition of the list (or very little). This is due to the fact that with q_s and q_{bp} we measure the percentage of well positioned elements and there is no improvement until the elements are in their final position. With q_v, the position of one is compared to its neighbor and there is little hope to change relative ordering even if the "largest" elements are seperated from the "smallest" ones.

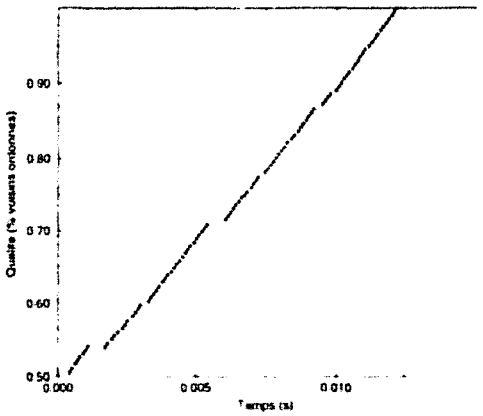
In the particular case of quicksort, the graph obtained here (figure A.9(d)) can help us to analyze what type of list we have to sort. As shown on this figure and considering



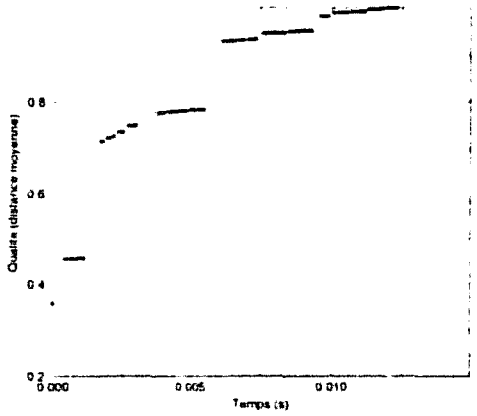
(a) % Already treated q_s



(b) % Well positioned q_{bp}



(c) % Well positioned neighbors q_v



(d) Average distance q_t

FIG. A.9 - *Behavior of a quicksort on a random list of 14000 elements chosen between 1 and 14000 - 100 measure points*

what we just said about the behavior of quicksort, we can observe that there are steps of different length, corresponding to the different length of the parts made during the sort. Finally, we can note that a sorted list or a reversed list would have given a graph with as many time gaps as elements. This is the case where the recursion tree is of maximal depth (and so computation time).

A.2.1.4 Other inputs?

Here we anticipate a little the paragraph about choosing the inputs, by using a different mixing strategy. But the aim is only to show what other influence the quality criterion can have on the aspect of a quality map.

So we can imagine that lists are not always random and we consider that they can have more regularities. For example, they can be lists with sorted parts. We implemented an algorithm that mixes a list by randomly choosing two elements several times and exchanging their positions. It is especially interesting to apply this algorithm (Algorithm A.5) on a sorted list.

ALG A.5 Mix of sorted list by swapping couples of elements

```
procedure mix
begin
  for cpt from 1 to nb_permutations do
    repeat i ← random() modulo (n + 1) until i = t[i]
    repeat j ← random() modulo (n + 1) until (j = t[j]) and j ≠ i
    t[i] ↔ t[j]
  endfor
end mix
```

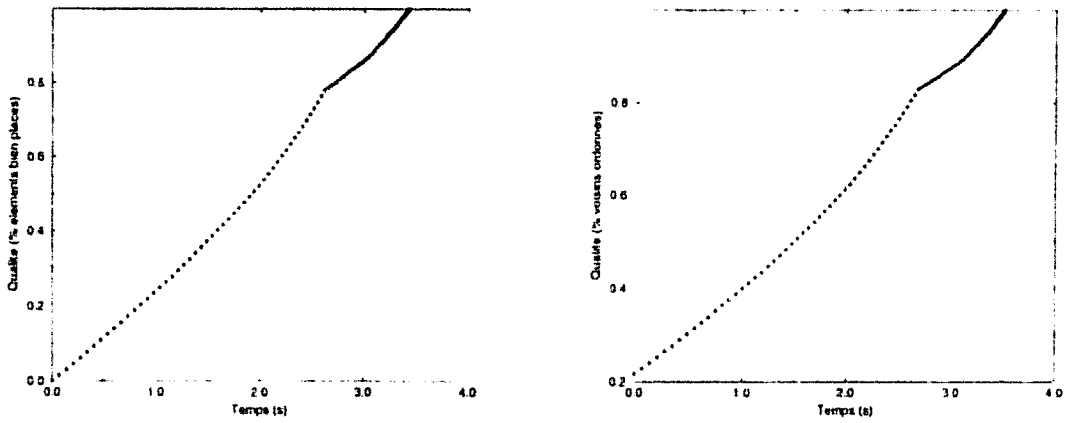
With algorithm A.5, we can control the rate of mixing by giving a value to *nb_permutations*. We also test if the element has been exchanged by comparing it with its position. For example, if the number of permutations is 3500 on a list of 14000 elements, there will be 7000 elements that will change their place by couple. For the second quality criterion q_{bp}, it leads to a quality of 50% for such an input list, as we are sure that an element is exchanged only once. We could apply a similar algorithm on a reverse list, but we have to use another test for verifying that an element has not been exchanged. This time we have to compare it with $(TABSIZE - (j - 1))$ where j is its position.

If we use a mixing rate of 100%, that is a number of permutations equal to the half of the size of the list, it should lead us to the same type of lists that we get with random mixing strategy presented above. In fact, it is a little different as with algorithm A.5, we are sure that there is no element in its final position. But the drawback of this approach is that it can be much longer than algorithm A.2 that continuously "updates" a list of non-treated elements.

Note that mixing strategy enables to control the beginning quality for q_{bp}, not for q_v. This is due to the method used to choose the couples to be exchanged. If we have chosen to exchange two neighbors, the q_v quality could have been controlled. As a conclusion, there is one mixing strategy, or more generally one preparation method, of inputs for one quality criterion.

Here we will show that all the criteria that we presented above are not suitable for all algorithms, even if the results given so far seem to prove the contrary. We also present a particular and unexpected behavior of the bubble sort and the ordinary selection sort.

Ordinary selection sort Here we present and explain unexpected results we got with an reversed list mixed with a rate of 25%, that is we are sure that 25% of the elements are not in their final position (regarding algorithm A.5). We selected the two particular cases of q_{bp} and q_v quality criteria. These results are presented on figure A.10.



(a) % Well positioned elements q_{bp} (b) % Well positioned neighbors q_v

FIG. A.10 - *Behavior of an ordinary selection sort on a sorted list of 14000 integers chosen between 1 and 14000, randomly mixed with a rate of 25% - 100 measure points*

We can observe on both graphs that there are two phases in the progress of the algorithm in term of quality. To explain this fact, it is important to note that the ordinary selection sort exchanges two elements at each step. On a totally reversed list it results in two more well positioned elements at each step, as the first one was in the last position, the last one was in the first position and they have been exchanged. In the case where we partially mixed the reversed list, we verify that the first phase stops when the algorithm reaches half of the total steps necessary to finish the sort. Until this point, there is chance to improve quality twice instead of only once. It is only a chance (perhaps near 75% of probability) as there are elements that are not in their "reverse" position. After this point, there is no chance to place two elements at the same time and this leads to a lower quality improvement speed. The same explanation suits q_v on figure A.10(b) because a reverse list is of bad quality (0 in fact!) and at each step there is a chance to order two more couples of neighbors.

Bubble sort and q_v The same type of unexpected behavior is obtained when the bubble sort is applied to a sorted list mixed with a rate of 25% and the quality criterion q_v. These results are presented on figure A.11.

But the interpretation of this behavior is different. As a matter of fact, we observe that the inflexion point happens sooner when the mixing rate is smaller. The reason is that with bubble sort, the smallest elements that have been exchanged with the largest ones quickly go back to their position as the algorithm begins a step from the end of the

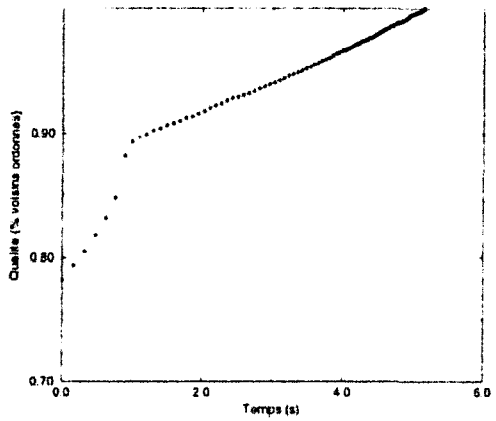


FIG. A.11 - *Behavior of bubble sort on a sorted list of 14000 integers chosen between 1 and 14000, randomly mixed with a rate of 25% - quality measures the percentage of well positioned neighbors - 100 measure points*

list. But the largest elements move towards the end of the list and their final position much more slowly: one position per step in fact, when a small element goes back to its final position in a few steps (often one, for the smallest ones). And there is no gain of quality until an element moving to the right reaches its final position with this criterion.

As a conclusion, with bubble sort, we can say that the smallest elements quickly go back to their positions and consequently quickly inproves the number of ordered neighbors, and that the largest ones slowly move to the right toward their final positions. There is a number of steps in this algorithm when all the smallest elements are well positioned and after that only the largest elements increase quality. This is the reason for the inflexion point. The fact that it happens sooner when the mixing rate is small is linked to the small number of smallest elements in this case.

Difficulties with Bubble sort and Quicksort On figures A.12 and A.13, we show the difficulties that we can meet in choosing a quality criterion.

For bubble sort (figure A.12), the quality value decreases from nearly 77% towards 0% after the first step measurement. The reason is due to the criterion applied in this case. We measure the number of well positioned elements and the bubble sort consists in placing the first element in its final position by translating to the right every element that was before the previous position of this smallest element. As a consequence, every element that was in its final position in this subset is moved to the right and quality falls. The technique consisting in keeping the last best quality (it is possible because quality here is recognizable) could be applied but it is clear that it is not very pertinent in this case.

As for Quicksort (figure A.13), we consider a reversed list with a mixing of 25%. Here we observe short periods of decreasing quality on the large steps of slow improvement. In

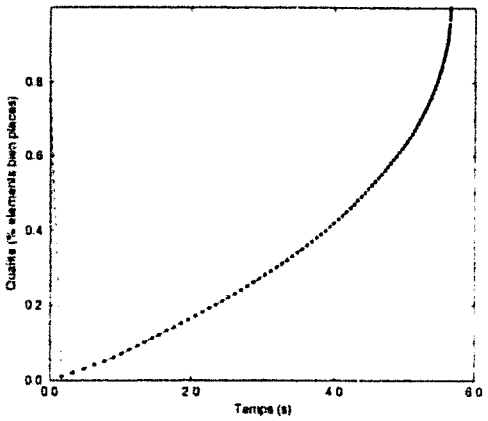


FIG. A.12 - *Behavior of a bubble sort on a sorted list of 14000 integers chosen between 1 and 14000, randomly mixed with a rate of 25% - quality measures the percentage of well positioned elements - 100 measure points*

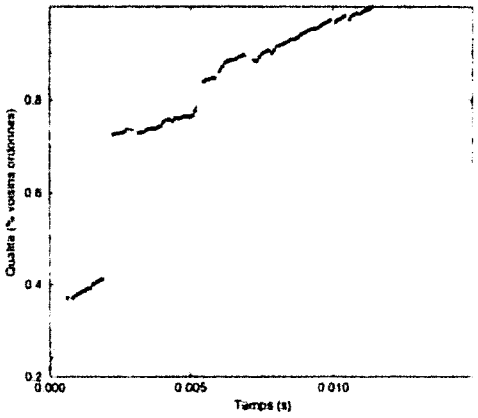


FIG. A.13 - *Behavior of a bubble sort on a reverse list of 14000 integers chosen between 1 and 14000, randomly mixed with a rate of 25% - quality measures the percentage of well positioned neighbors - 100 measure points*

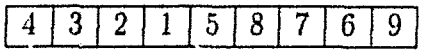
fact, quicksort does not provide any guarranty on getting better and better quality with this criterion. Let consider the following simple example and the first partition of the list in the first step:

Original list:

6	7	8	9	1	2	3	4	5

that becomes, after the first partition when the partition element is the last one (here

5):



Here the partition procedure exchanges 4 and 6, 3 and 7, 2 and 8, 1 and 9, and finally places 5 by exchanging it with 9 that is in its position. Quality of the original list is 7/8 and quality of the second list is 3/8 according to the criterion that counts the proportion of ordered neighbors. It confirms that this quality can decrease when using quicksort.

Contrary to the bubble sort case presented above, it seems valuable to keep the last best quality to get a monotonous graph as there is no long time to wait for an improvement. But if we condider our last example, it is not so obvious.

These two cases show that there are some disadvantages in choosing particular quality criteria for some algorithms. Some of these difficulties can sometimes be "repaired" by keeping the last best quality, some others not.

A.2.2 Discussion

A.2.2.1 Interpretation of experimental results

In this section, we observe the behavior of three sort algorithms with four possible quality criterion. Even if the problem of choosing a quality criteria seems to be simple *a priori*, we exhibit several difficulties in solving it.

As for ordinary selection sort and bubble sort, they give the "same" partial result, that is they ensure that the first part of the list is sorted. That is why they are called *progressive sort algorithms* [FGS93]. As a consequence, we could think that q_s, measuring the progress of the algorithm and q_{bp} would be good candidates. But we saw that the second one is not suitable for bubble sort in case of partially sorted input lists. We will discuss about the probability of getting such lists in the next section.

Insertion sort algorithms give a sorted first part of the list but it is not definitive as new elements can be inserted. We could imagine other quality criteria for this, like measuring the number of sorted elements at the beginning of the list. Quicksort, that we studied, accept all the criteria that we proposed under certain conditions.

There seems to exist a criteria for a specific way of sorting. More generally, it is important to study the way an algorithm works before trying to define a quality criteria. It might seem obvious, but all cases have to be considered, as we did for sort algorithms and particular input lists.

After considering the choice of a criteria for an algorithm, we have to consider the point of view of the programmer, that is the context in which the algorithm will be used, and the possibility of this use as an anytime algorithm.

A user of sort algorithms must have an idea of what type of intermediate result he wants. For example, is it a list with a sorted beginning that he is interested in? If so,

he will probably use q_s or q_{bp}. Is he interested in lists where the elements are not far from their final position? Then q_d will be suitable for him; and so on with other criteria. Information that could be important is whether the result of this algorithm will be used by another anytime algorithm; then the quality criterion for outputs must make sense for the input of the following algorithm.

But there is one more point to insist on: the practical pertinence of the criteria and its use in the framework of anytime algorithms. For instance, remember a property that is necessary for interrutible algorithms: a recognizable quality criterion, that is a criterion that can be computed at run-time [Zil96]. We can consider that a criterion is recognizable when it can be computed in constant time. Constant time is imposed to control this quality computation in the monitoring application, and taken into account with certainty about its cost. All the criteria that we proposed are recognisable if the length of the list is constant. The question is: is it reasonable to compute a quality in a time of the same order value of one step of the algorithm (not rare in case of polynomial algorithms)? The answer is positive if the monitoring strategy is not completely senseless and interrupts the anytime algorithm after each step to get a correct value of quality. It is clear that such a strategy is useless. As for the practical interest of criterion, take the measure of the percentage of well positioned elements q_{bp} and the measure of the average distance to the final position of an element q_d. As we said above, there is a necessary assumption before using these criteria: the lists have to be composed of naturals from 1 to $TABSIZE$, the size of the list. Without this assumption, the computation time for quality is of the same order as the computation time for sort itself. So, these criteria have a limited application field. It could be interesting to ask the same question for other algorithms, and especially for the applicability of criteria used in the case of the Traveling Salesman Problem that we quoted in the introduction discussion of this chapter.

When it is hard to get an analytical definition of the criterion, it could be interesting to use functions that bind the real quality of the results, that is a graph for the upper limit and another the lower limit. This approach is used by DE GIVRY and VERFAILLIE in a particular problem of VCSP (Valued Constraint Satisfaction Problem) [dV97]. In this paper, the authors produced an anytime bounding of the quality for the CSP. That means that quality evaluation is itself an anytime process in the sense that the evaluation of this quality is more and more precise when time passes, and bounding is more and more narrow. The problem here is that it is an anytime bounding for a VSCP and not a bounding for an anytime VCSP. As a result, quality evaluation is not precise at the beginning of the process and in the framework of anytime algorithms, it is especially interesting to have a precise quality at the beginning of a process to avoid long and "useless" computation.

A.2.2.2 Input quality vs. beginning quality

Here we try to pinpoint a definition that could be ambiguous when beginning with anytime algorithms. We saw in the previous paragraph that inputs of the sort algorithms have a certain non-zero quality regarding the ouput quality criteria. That means that the

algorithm perhaps does not do the same work if this quality varies.

But a problem appears if the input is not of the same nature as the output. This assumption is very natural in general. For example, the input of a TSP algorithm is not a random hamiltonian cycle, but graph with a certain connection rate, a certain geographic distribution of cities, etc.

That is the reason why we use the term *beginning quality* when we refer to the quality of the input before the start of the algorithm regarding the output quality criteria. It is sometimes abusively called the input quality. Even so it can be appropriate in some cases like in the sort problem, in other cases input quality is something very different. So when we want to evaluate real input quality, we also use the term of *quality of the input*, to insist on the difference with beginning quality. The quality of the input evaluates the input to discriminate output quality maps. This can be used to get conditional performance profiles as in [Zil93] and [GZ96].

Sometimes beginning quality and input quality will be equal, but there are no rules against using several criteria at the same time to qualify an input. For example, in the case of sort, it can be usefull to use q_{bp} before applying the quicksort. If q_{bp} is high, quicksort then becomes inefficient. And the output quality can be measured by q_d depending on what application these outputs will be used.

All these considerations show that all the applications are not easy to use as anytime algorithms and that a quality criterion for simple or more complex algorithms is not easy to define. Assuming that a quality criterion is defined, let us see what sort of difficulties are met in building a performance profile.

A.3 Construction of performance profiles

Building performance profiles, in the case where no anilytical method enables doing it directly, is mainly divided into two parts:

- the collection of data: it consists in choosing instances that will be the inputs of the algorithm, so as to collect couples of data composed of computation time and corresponding output quality, and build the quality map. The quality map is the superposition of all these couples (time, quality) on a bidimentional graph.
- tranformation into a performance profile: as a quality map is often a simple spray of points, a process that transforms it into a performance profile is needed.

There is a subsequent work on exploiting data of a quality map to produce a performance profile [Zil93][HZ96][GZ96]. For example, by considering the influence of the inputs, we can get a *conditionnal performance profile*. Sometimes the performance profile is a mean of the quality for each possible interruption instant and that leads to an expected performance profile. To take the frequency of appearance of couples (time, quality) into account, and in fact the probability to get a particular value of quality at a fixed time, we can build *distribution performance profiles*. Performance profiles that consider

the state in which the process is just before being interrupted to resume the algorithm are also possible under the name of *dynamic performance profiles*[HZ96]. That is the reason why we will focus on the collection of data in this section.

A.3.1 Collecting Data

The problem with collecting data is to choose the representative inputs for the anytime algorithm that would give the quality map that predicts at best the future behavior of the algorithm. We assume in this case that no algorithm analysis can directly supply a compact performance profile. These inputs are used for learning how the algorithm behaves, and we hope that inputs given in a real si. ation will be very similar so as we can use the performance profile.

When the user of anytime techniques knows the context very well in which he will use his anytime algorithm, the problem becomes easier. But even in this situation, it is sometimes hard to consider all the possible cases of inputs that will influence the behavior of the algorithm. How can we get the right instances, especially when we have poor information about the context and about the real inputs?

The general and usual approach to "learn" a quality map is often to generate random inputs and consider that they are representative because they are the most current ones in the finite set of all instances (we will see why in this section) [Zil93]. A study done by GRASS and ZILBERSTEIN [GZ96] considered specific properties of inputs to produce conditional performance profiles for bubble sort. But they consider that quality, defined by the ratio of couples of neighbors that are in order, can not go under 50%. It is not the case and the simple example of a reversed list gives 0% of quality. So they neglected the possible inputs under 50% of quality.

Thus in this section we will focus on the following questions:

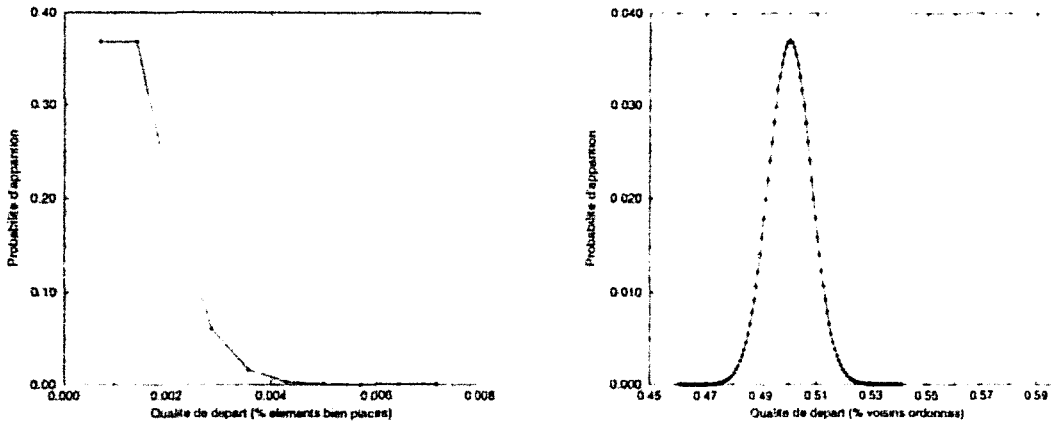
- Are the random inputs realistic because they are numerous?
- What risks do we take by neglecting some "rare" inputs?

We first observe several experimental results to try to exhibit some characteristic behaviors.

A.3.1.1 Random generation of inputs

As, we said, the random generation method for inputs are often used for the systematic choice in the set of instances. Assuming that it is a good strategy for the sort algorithms, we tried to analyse the behavior of these algorithms applied on random inputs. First we had a look at the distribution of the quality of the inputs, that is, we studied what we called the beginning quality of the inputs. Then we focused on quality maps produced by the sort algorithms applied on many random inputs. Finally we considered problems induced by the use of random generation, that is avoiding extreme cases and the influence of the size of the instances.

Distribution of instances Before considering the behavior of a sort algorithm applied to random inputs, we examine how the quality of the inputs is distributed when they are chosen with a random method. As a matter of fact, if the variance of quality is "wide", then there is very little hope to get a narrow quality map. Here we assume that output quality criteria that we defined in the previous section are valuable criteria to qualify the beginning quality of the inputs. The only exception is the first quality criterion, q_s that reflects the progress of the algorithm and does not evaluate output quality. With this criterion, beginning quality is always 0. The results of the quality distributions according to the criteria q_{bp} and q_v are presented on figure A.14.



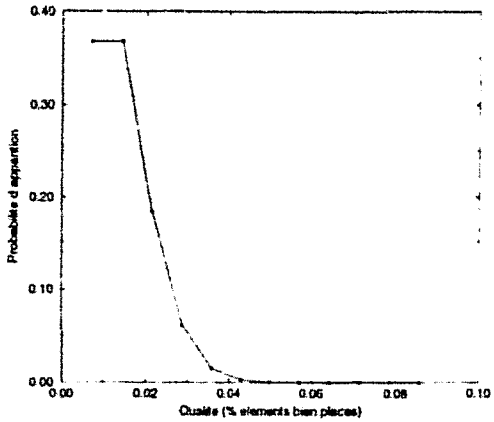
(a) % well positioned elements q_{bp} (b) % well positioned neighbors q_v

FIG. A.14 - *Distribution of beginning quality of random lists of 1400 elements chosen between 1 and 1400 - 10 000 000 samples*

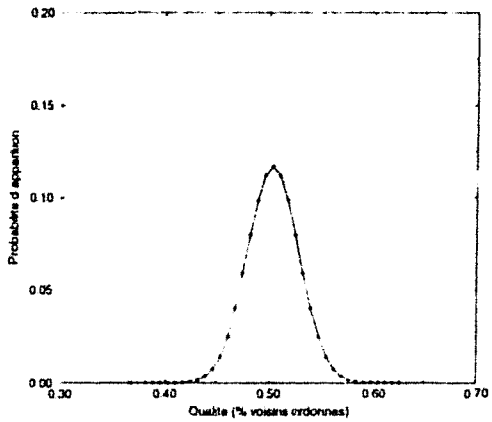
As we can observe on figure A.14 for q_{bp} and q_v, the fact that random instances under a uniform distribution of probability are in large majority is confirmed. So we have a very high chance to get a random input. The resulting variance is quite narrow for each criterion (respectively 0.00205 and 0.00233).

Note that the instances that we chose have a relatively large size (1 400) Then the next point to observe is the influence of the size of the inputs on the variance. The results presented on figure A.15 show that if the size of the list decreases, the probability to get extreme cases (in the quality sense) is higher and higher. The reason is that the random lists are less and less numerous. For these results, the variance on quality is higher and higher (from 0.0246 to 0.226 for q_{bp} and from 0.0248 to 0.243 for q_v). We only show results for q_{bp} and q_v, but q_d has the same property. In fact, it seems that this phenomenon is independant of the quality and depends only on the inputs and their size.

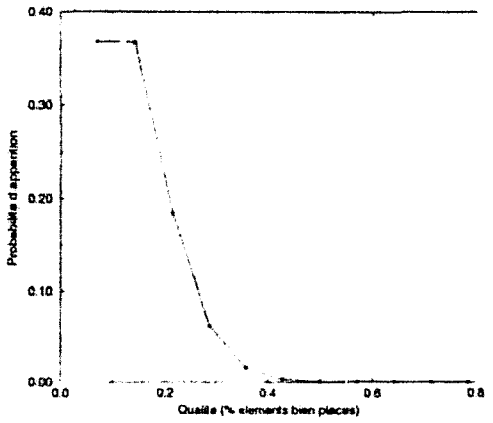
As shown here, the size of input data is a very serious point to take into account when we randomly choose our inputs in the set of possible instances. We will see what impact this has on the behavior of the algorithm in the next paragraph.



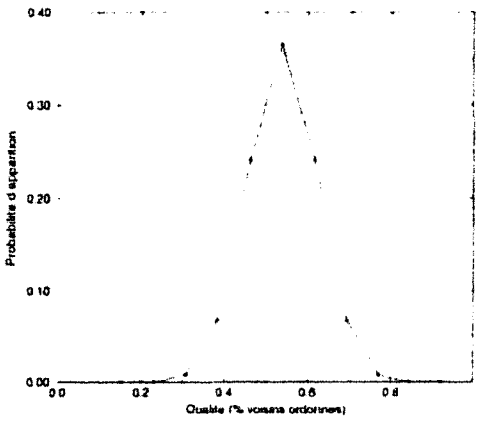
(a) % well positioned elements q_{bp} on 140



(b) % well positioned neighbors q_v on 140



(c) % well positioned elements q_{bp} on 14



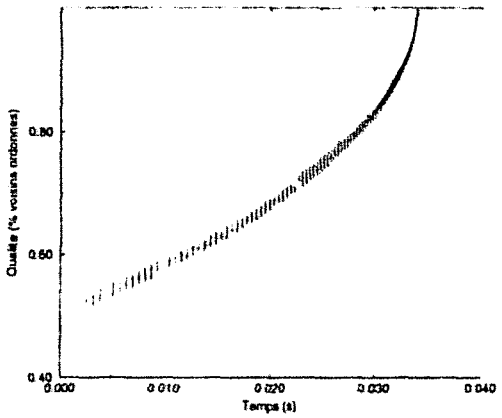
(d) % well positioned neighbors q_v on 14

FIG. A.15 - *Distribution of beginning quality of random lists of n elements chosen between 1 and n - 10 000 000 samples*

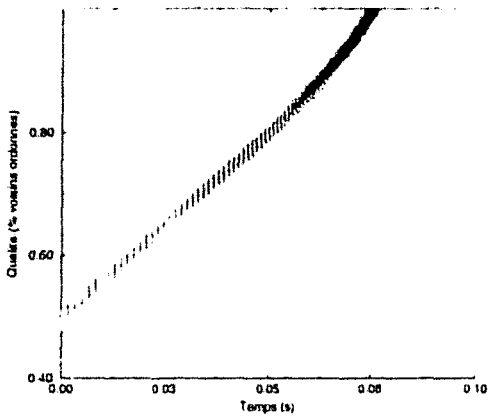
Behavior of algorithms with random inputs To study the behavior of the three sort algorithms - ordinary selection sort, bubble sort and quicksort - we selected the criterion q_v as it can be applied to every type of list, containing naturals or not. In fact, what is interesting here is the variation of quality in function of input data, but also of computation time. Our goal is to see if the quality map can have a narrow variance with such random inputs that we describe above.

The results shown on figure A.16 confirms that sort algorithms applied to random lists with this quality criterion q_v give a "narrow" quality map. Moreover, the more time passes, the more the quality map is narrow. This property is clearly independant of the quality criterion. Trials with other quality criteria give the same type of behavior (see Appendix C.1).

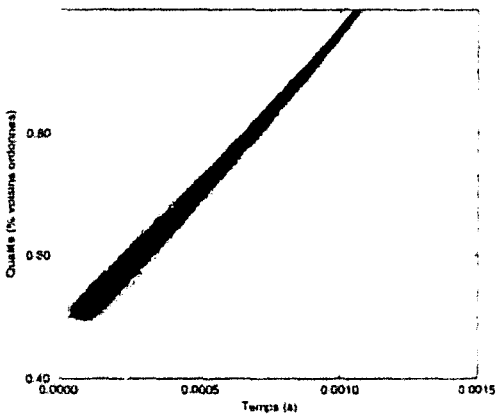
Before concluding that the random method for choosing inputs is the right one, we



(a) Ordinary selection sort



(b) Bubble sort

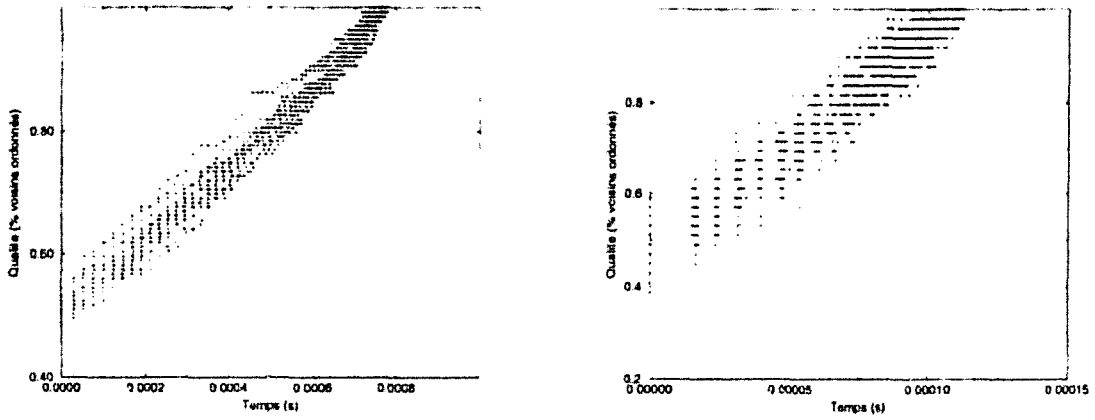


(c) Quicksort

FIG. A.16 - *Behavior of sort on random lists of 1400 elements - Quality q_v, percentage of well positioned neighbors - 100 measure points - 100 samples*

study the influence of the size of inputs on the behavior of the algorithm and see if the quality map is still narrow or if it is enlarged due to the wide distribution of inputs. For example, a trial with quicksort valued by q_d (Appendix C.1.2) shows a very wide quality map.

As expected, the wider distribution of inputs applied to a bubble sort leads to a wider quality map as shown on figure A.17 with lists of 140 and 50 elements. Complementary experimental results are given for other sort algorithms in Apendix C.2. They lead to the same conclusion. We exhibited a first disadvantage of the random method for generating inputs for an algorithm: the narrow character of the quality map is dependant on the size of the inputs. Thus we are not sure of getting a consistent performance profile.



(a) 140 elements (70 measure points)

(b) 50 elements (25 measure points)

FIG. A.17 - *Behavior of bubble sort on random lists of n elements chosen between 1 and n - Quality q_v, percentage of well positioned neighbors - 100 samples*

About the worst case? Exhibiting extreme cases when diminishing the size of the inputs lead us to the question of the worst case. But what type of worst case, that is among what criterion, can be defined for an anytime algorithm. There is first the classical combinatorial worst case, that is the instance that forces the algorithm to the longest computation time on a fixed machine. But worst case in terms of quality is not the same in general. For example, a worst combinatorial case for the quicksort is a completely sorted list. In terms of quality, it is clear that it is the best case, because whatever the criterion that evaluates the progress toward the final result, the quality will be 100%, from start to termination of the algorithm.

For example, what is a worst combinatorial case for the ordinary selection sort? This sort algorithm is composed of two parts at each step: a search in the list for the smallest element and a swap of two elements if the first one is not the smallest one. This could be modeled as following:

$$t_{computation} = Lt_c + kt_e$$

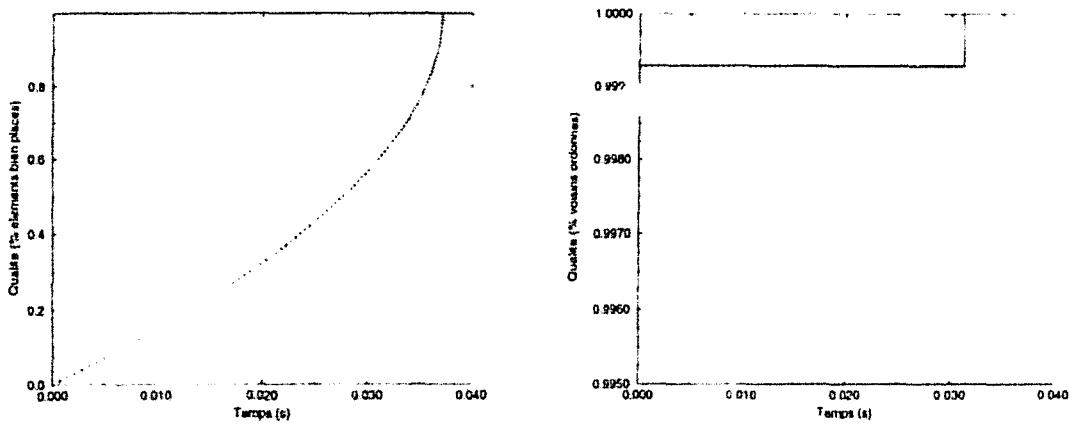
where:

- L is a value of the order of square of list length and t_c is computation time for one comparison
- k is the number of echanges and t_e is computation time for one exchange

In every list of constant length, the number of comparisons is constant because the search is systematic and goes all through the remaining list at each step. What can vary is the number of exchanges and a worst combinatorial case will be one instance that forces the algorithm to a maximal number of exchanges, that is one per step. For example, the list beginning with 2 to n, and 1 in the last position. At each step, the smallest element is at the end of the list, so we are sure to make one exchange.

The question is now to define the worst case regarding quality. The way could be to get

a lower bound of the quality map. It is obvious that it depends on the quality criterion. Again for the ordinary selection sort, the case we described above (list [2,3,...,n,1]) gives one of the worst quality if we consider criterion q_{bp}, proportion of well positioned elements. It leads to a worst case regarding quality from start to termination of the algorithm as it gives the slowest increase of quality at each step (only one more element is positioned) and the worst computation time (figure A.18(a)).



(a) criterion q_{bp}

(b) criterion q_v

FIG. A.18 - *Search of worst case in term of quality for ordinary selection sort - List [2,3,...,n,1]*

As for criterion q_v, the quality of the list is not the worst anymore, and on the contrary it is near the best one we could get. This quality will be constant until the last step where it reach 100% (figure A.18(b)). For this criterion, the worst beginning quality is got with the reversed list. But this particular instance does not force the algorithm to the maximal number of exchanges, but only the half of the maximum. The reason is that two more elements are well positioned at each step and after reaching the middle of the list, there is no more exchange as the list is sorted. As a consequence, the quality increases quicker than average cases and the resulting graph crosses the other graphs. This is not a lower bound of the quality map. It seems that there is no possibility for q_v to define one as we always have to be under other graphs and consequently to begin with the lowest quality which corresponds to the reversed list.

These considerations on the worst-case notion show that defining a lower bound of a quality map corresponding to one particular instance is not always possible. Then, in general, a "worst-case" instance in terms of quality and intermediaite results has no sense.

A.3.1.2 Other methods for generating instances

In the previous section about the choice of a quality criterion, we used an algorithm for mixing data and fixing the beginning quality. This method can be used to observe the behavior of sort algorithms on partially sorted lists and verify the influence of the

beginning quality. We will see how the quality maps generated from these type of inputs are related to the quality maps generated from random instances.

Random mixing of a non-random list As we said above, the mixing strategy presented in algorithm A.5 is designed to control beginning quality for the criterion q_{bp}. Then we study the behavior of the ordinary selection sort with this criterion. The input lists are generated from a sorted list with a mixing rate of 1/100, 1/10, 1/5 and 1/2. We compare these graphs with the graph got with a random input list. The results are shown on figure A.19.

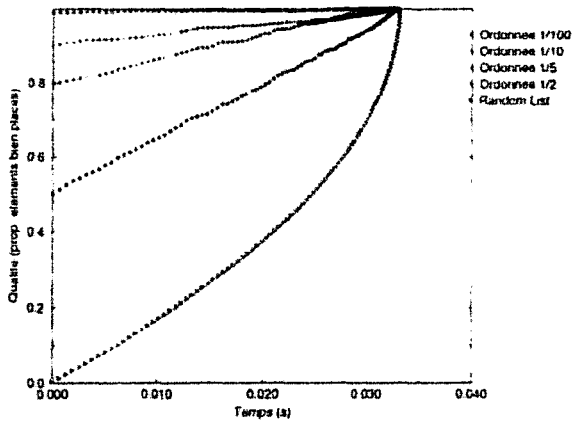
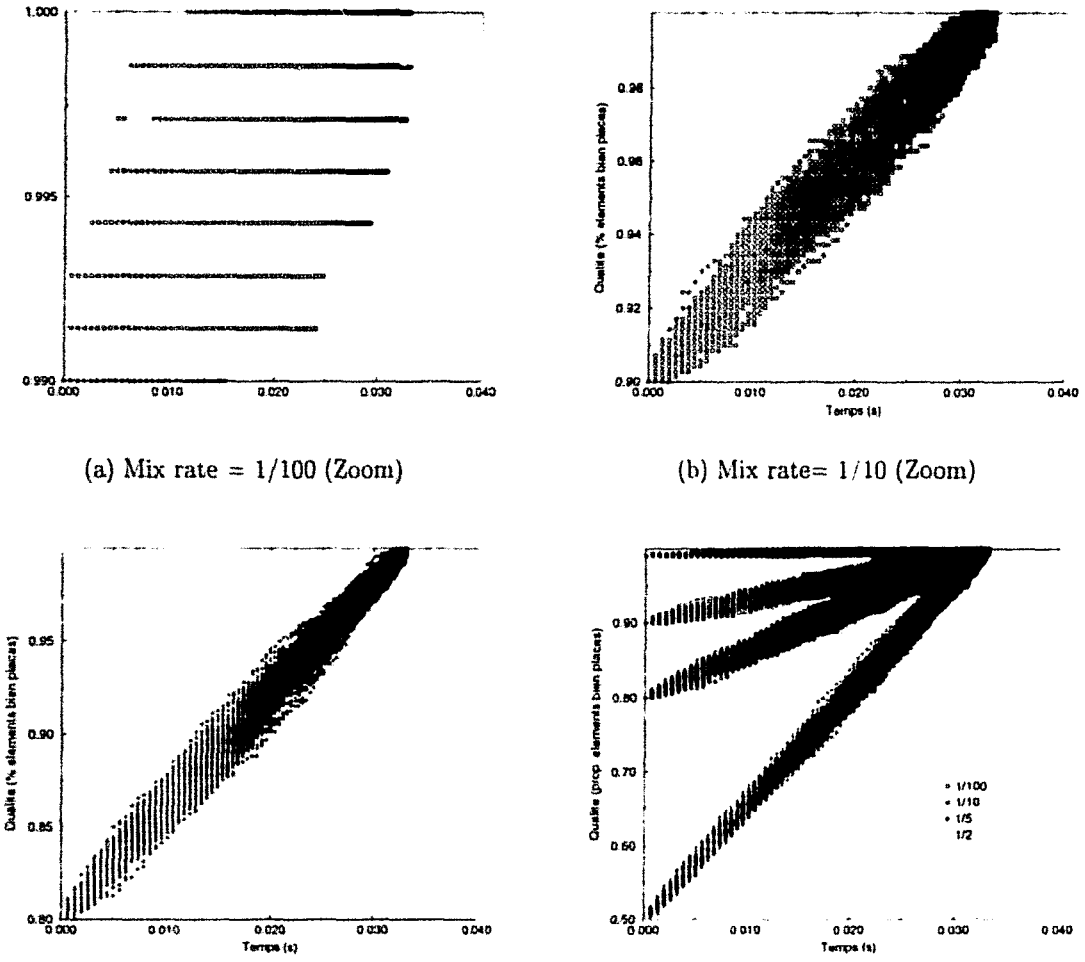


FIG. A.19 - *Behavior of an ordinary selection sort on sorted lists of 1400 integers chosen between 1 and 1400, then randomly mixed with various rates - quality q_{bp}, percentage of well positioned elements - 100 measure points per list*

First, we can observe on figure A.19 that we effectively control the beginning quality for the sorted list, as for rates 1/100, 1/10, 1/5 and 1/2 we get respectively a quality of 0.99, 0.9, 0.8 and 0.5. To verify if starting with sorted list and preparing the inputs lists like we do would be a good approach to build conditional quality map, we did trials by fixing a beginning quality. The results are shown on figure A.20.

The first result on figure A.20(a) with a very high beginning quality shows that there is very little hope to predict the behavior of the ordinary sort algorithm between quality of 0.99 and 1. There are few (8) possible levels of quality from start to termination - and it is natural - but there are many possible intermediate computation times. The uncertainty comes from computation time. A quality q_{bp} of 0.99 for a list of 1400 elements means that there are 14 elements that are not in their final position. Considering the way of doing of this sort algorithm, if there are lots of these 14 elements whose final position is at the beginning of the list, then the list will be very quickly sorted. On the contrary, if the final position of these elements are at the end, then it will take the maximum computation time to place them well. The more the proportion of unsorted elements grows (when beginning quality decreases), the more these extreme cases are close from each other as we can see on figures A.20(b), A.20(c) and A.20(d).



(a) Mix rate = 1/100 (Zoom)

(b) Mix rate= 1/10 (Zoom)

(c) Mix rate = 1/5 (Zoom)

(d) Every mix rate

FIG. A.20 - *Behavior of an ordinary selection sort on sorted lists of 1400 integers chosen between 1 and 1400, then randomly mixed with a various rates - quality q_{bp}, percentage of well positioned elements - 100 measure points per list - 200 samples*

The resulting quality map looks thiner and thiner. This impression is mainly due the relativity of the point of view. As a matter of fact, we used a different scale defined by extreme values of quality and time for each graph. As show on the last figure (A.20(d)) where we draw all the four quality maps, the width of the possible quality values for a given time is in fact increasing. But we can verify that the first graph with beginning quality at 0.99 is now satisfactory from this point of view. As a conclusion on these trials, we can say that evolution of quality is not perfectly predictible even by fixing beginning quality, but it is rather satisfactory if we choose a large scale of quality, from 0 to 1 for example.

The next question is to know if a (quasi) constant beginning quality is sufficient to condition the behavior of this sort algorithm with it. Let us consider trial done on reversed lists. We did the same mix on these lists to get results shown on figure A.21.

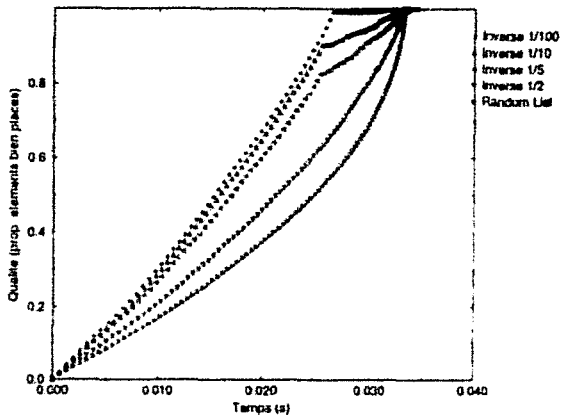


FIG. A.21 - *Behavior of an ordinary selection sort on reverse lists of 1400 integers chosen between 1 and 1400 - Quality q_{bp}, percentage of well positioned elements - 100 measure points per list*

For all the so prepared reversed list, quality is very close or equal to 0. As a matter of fact, with our mixing method, by starting from a reversed list, we have very poor chance to place one or several elements in the right position, even this chance increases with mixing rate. Then we get lists prepared with a different method but with (quasi) constant beginning quality. We can immediatly verify that the answer to our last question is negative: it is not sufficient to fix the beginning quality, as different methods to fix it to the same value can produce different behaviors.

Conclusion As we saw in our previous trials, there appear two problems when we randomly choose inputs in the set of all instances, following the uniform distribution:

- the variance of beginning quality can be large, logically leading to a wide quality and loss of predictibility (consistency of the performance profile)
- even when we fix beginning quality by using a unique or several random methods, there are possibilities to generate extreme behaviors.

The first point could be solved if we consider that resulting quality maps are satisfactory in the case of inputs with large size. In this case, quality maps are relatively narrow. It could seem natural to use anytime algorithms on large inputs, but there exist problems that have high complexity and that need huge computation time even for small inputs.

The second point is more serious as it proves that there exist inputs that lead to extreme behavior, even with a constant beginning quality. The question is to know if these cases are representative of real cases. Note that extreme cases that we exhibited are rather "structured" inputs in the sense that we use precise deterministic or semi-deterministic methods to produce them. More over, they are partially ordered. If we consider "totally" random inputs, we know that they are far more numerous than partially sorted inputs. This can be proved by simple considerations on the number of possible permutations in a

list or in a subset of this list (see paragraph A.3.1.1). Then we could consider that these extremes cases rarely happen. But is it the right approach to consider that every list has the same chance to be generated in reality?

In fact, the notion of randomness is not always clear: all the lists that we produced contains randomness, less for the second generation method with partially sorted inputs. Thus we need a more precise definition of random and quantification of the risks we take when neglecting extreme cases seen above. Another question is to verify if uniform probability is a good model of appearance of lists, and inputs in general. In the next paragraph, by considering the Kolmogorov complexity theory, we get some clues about the answer to these questions and what to do (or not) for the generation of representative instances.

A.3.2 Analysis with Kolmogorov complexity

As we just saw with the examples of sort and TSP algorithms, we question the random choice of instances of a problem to get the most representative inputs in reality. So we need a tool that could enable us to characterize the instances that we choose, to confirm that these instances are realistic or not, and what would be the consequences on the "practical" complexity, *i.e.* the computation time, of the algorithm. A theory, namely the Kolmogorov complexity theory, authorizes the analysis of the structure of data, and reversely its randomness [LV97]. In this section, we present the basic principles of the Kolmogorov complexity, then we explain how this theory could help us in the building of a quality map and a performance profile.

A.3.2.1 Principles of Kolmogorov complexity theory

The majority of the results, definitions and theorems exposed here can be found in the book of Li and Vitányi [LV97].

The Kolmogorov complexity theory is an approach that enables the evaluation of absolute amount of information contained in individual objects. This quantity only depends on the language used for the description and hopefully, it can be proved that every reasonable choice of programming languages leads to an invariant evaluation to an additive constant. This quantity of information is called Kolmogorov complexity, or Chaitin-Kolmogorov complexity.

Intuitively, the amount of information in finite string is the size (in number of binary digits, or bits) of the shortest program that generate this string. Just take the following string as an example:

$$s =' 01100101011010010110100100 1101'$$

We can easily believe that s is the result of the 30 tosses of a coin. We can hardly believe it for s':

$$s' =' 11111111111111111111111111111111'$$

However with 30 tosses, s and s' have the same probability to appear, even though our intuition tells us that s' is less "random" than s. The difference is in the fact that we prefer to explain the string s' with the program 'print "1" 30 times', rather than with pure chance. As s', the transcendental number π contains few information because it can be generated by a short program, even if its decimals seem random. So, we said that an object that contains regularities like s' is *compressible* since it has a shorter description than itself.

Définition A.1 *Kolmogorov complexity*

Let assume an effective coding of each Turing machine on an alphabet $\{0,1\}$: $M_1, M_2, \ldots$

The Kolmogorov complexity of each string $x \in \{0,1\}^*$, *where* ε *is the empty string, is:*

$$K(x) = min\{|M_j| : M_j(\varepsilon) = x\}$$

Where ε *is the empty string.*

$K(x)$ *is the length of the shortest program starting from scratch that generates* x.

If a supplementary information y on x is supplied, the conditional Kolmogorov complexity is defined by :

$$\forall y \in \{0,1\}^*, K(x|y) = min\{|M_j| : M_j(y) = x\}$$

As a conclusion on our example, we can say that string s' has a smaller Kolmogorov complexity than string s. It is difficult to say more, and especially to compute the precise value of $K(x)$ because the Kolmogorov complexity is not computable. But in practice, $K(x)$ can be approximated with an upper bound by enumeration of every description program. Then we say that $K(x)$ is semi-computable.

Définition A.2 *Incompressibility*

Let c *be an integer constant. The string* $x \in \{0,1\}^n$ *is* c*-incompressible if* $K(x) = n - c$

That means that there is no description shortest than $n-c$. The notion of incompressibility (say c-incompressible with a small constant c) offers an elegant way to express randomness in a finite string. We can say that x is random if and only if x is incompressible [LV90]. As a matter of fact, to describe a random string, we need as many bits as in the string itself. Intuitively, an incompressible string x is a string without any structure, thus that can not be described by a program shortest than "print x".

As Li and Vitànyi said [LV97], even if the Kolmogorov complexity theory is full of deep and sophisticated mathematics, it is sufficient to know only basis to apply with success this notion. So we will only present the results that interest us for our argumentation, knowing that this theory is not limited to this restriction.

A.3.2.2 Application to the choice of the instances of a problem

Remember hat our problem is to build graphs, that is performance profiles, that represent the b avior of an anytime algorithm. This graph will be then used to predict quality of the results in function of computation time in a real framework. As a consequence, we have to use representative instances of the real cases to produce a realistic performance profile. But what is a representative instance? And how to prove it?

A theorem on compressibility can inform us on the proportion of structured strings, or more generally, structured data.

Théorème A.1 *(Theorem 3.3.1 in [LV97])*

- $\forall n,\ max\{K(x) : l(x) = n\} = n + K(n) + \theta(1)$
- *for all fixed constant* r, *the number of strings* x *of length* $l(x) = n$ *with* $K(x) \leq n + K(n) - r$ *does not exceed* $2^{n-r+\theta(n)}$.

We are especially interested in the second part of the theorem. It means that the proportion of data of length n that are compressible of more than r bits is less than 2^{-r}.

This result means that, by doing a random choice in the set of data of constant length, there is a high probability to get random data as they are more numerous. By random choice, we mean with a uniform probability, that is equal chance for each data to be chosen. These results could justify the approach adopted by the majority of anytime algorithms designers [Zil93][GZ96]. They use the random choice of instances to build the quality map. As random instances represent the vast majority of the set of all instances of the problem (they are supposed to be finite and of constant length). We will now see that this random choice is not always the right method with the help of simple considerations.

The notion of compressibility used here is the same one when we talk about compression of data in the current computer science language. As a matter of fact, the compression algorithm work on the principle to reduce repetitions and eliminates regularities. So we can reason on this notion by considering usual compression problems of "every day life".

Let us consider the content of a hard disk in a computer. We can claim without lots of risks that information on a hard disk are representative of data that are computed by usual and realistic algorithms. The use of compression program on this hard disk is often used to gain storage space. And the results of this compression often gives good results (assuming that we are not trying to compress already compressed data). This simple consideration means that the majority of data on a hard disk are not random as they are compressible. As a consequence, we can deduce that the usual and realistic data are certainly not random. Thus the uniform distribution is not a good description of the appearance of data in reality. That is the reason why we use the notion of *universal distribution* also called *Levin measure* [LV97]

Définition A.3 *Universal distribution [LV97]*

The distribution m *that, for each string* x, *gives a probability* $m(x) = 2^{-K(x)}$ *is called the universal distribution.*

With this distribution, we can easily establish that very structured data (with low $K(x)$) have a high probability an that "complex" or random data (with high $K(x)$) have a low probability. Levin has proved that **m** dominates multiplicatively each enumerable distribution. The Levin measure is then a good way to measure the probability of appearance of a data in reality.

Let us try to turn into a formal argument to justify this universal distribution by quoting [KLV97].

> Suppose we observe a binary string s of length n and want to know whether we must attribute the occurrence of s to pure chance or to a cause. "Chance" means that the literal s is produced by fair coin tosses. "Cause" means that there is a causal explanation for s having happened - a causal explanation that takes m bits to describe. The pure chance of generating s *itself* literally is about 2^{-n}. But the probability of generating a *cause* for s is at least 2^{-m}. In other words, if there is some simple cause for s (s is regular), then $m \ll n$, and it is about 2^{n-m} times more likely that s arose as the result of some cause than literally by a random process.

To make the link with reality, we have to make the assumption that what happens in real world is like a computation, that the objects that we meet are the product of the computation of the physical world. By a first order approximation, the physical world can be considered as a Universal Turing Machine". Even if it is not provable, this seems very reasonable.

These ideas seem to be rather natural. As a matter of fact, objects in reality are rather structured. Otherwise the reality would look like a sort of fog. As for the sort case, for example, a lot of lists we have to sort are structured, like two sorted list to merge, or partially sorted lists. These lists are rarely totally random. One point that could confirm this fact is that people using the quicksort algorithm [SF96] often mix the lists before applying this algorithm. The reason is that quicksort has a complexity of $\theta(n \log n)$ in the average case that is random lists, and a complexity of $\theta(n^2)$ in the worst case, that is on sorted or reversed lists where the partition tree has a maximal depth [SF96].

Concerning our choice of instances to build a quality map, we do the following proposition with the help of previous considerations.

Proposition A.1 *Random is not current*

When we have to choose instances that are representative of the real case, considerations with Kolmogorov complexity proves that the random approach is not always the right method for generating instances. There are lots of real cases that are structured.

This proposition is rather destructive regarding the existing methods used to generate quality maps. It does not supply any method to produce a good set of instances, but it is helpfull to avoid what we call "the pitfall of the random approach".

A link with computational complexity What is interesting in Kolmogorov complexity theory, sometimes called "algorithmic theory of information", is that it gives some results on the behavior of the algorithms in function of the complexity of the instances.

First we have to precise the notions of *average case* and *worst-case complexity*[KLV97]. When we say *on the average case*, we assume that all inputs are equally likely, that is the probability is uniform, then the average computation time is an "uniform average" time on every instances. On the contrary, the *worst-case* is independant of any distribution and represents one or several instances that force the algorithm to use maximal computation time. Then, if we assume that the distribution is not uniform anymore, and that inputs with low Kolmogorov complexity are more likely than inputs with high complexity, the running time we *expect* under this (universal) distribution is (essentially) the worst-case running time of the algorithm.

Again this is not hard to see in the case of quicksort, for which the worst-case is given by a sorted list, that is a highly structured input. Under the universal distribution, the already sorted input is far more likely than the unsorted input, so we expect that quicksort will require n^2 steps. It is true in general when an algorithm runs fast on some inputs and slowly on others. Intuitively, the universal distribution assigns high enough probability to simple strings "to slow" the average running time of the algorithm to its worst case running time.

The following theorem due to LI and VITÀNYI [LV97] makes the link between Kolmogorov complexity and computational complexity, and give a formal argument to what we said above.

Théorème A.2 *(LI and VITÀNYI) [LV97]*

If the inputs to any complete algorithm are distributed according to m, then the average time complexity is of the same order of magnitude as the worst-case time complexity

In a few words, theorem A.2 shows that the inputs having small Kolmogorov complexity contain the worst cases with high probability. Conversely, VLASIE studies under which conditions the reciprocal of theorem A.2 holds, *i.e.* almost all the worst cases admit short descriptions.

Théorème A.3 *(VLASIE) [Vla97]*

If under a given enumerable input distribution, a NP-complete problem admits an algorithm with polynomial average-case complexity, then all but finitely many superpolynomial input cases, if they exist, are compressible.

The intuition of theorem A.3 is that for problems with a low average complexity, there is very few cases with a high computational cost. Thus the hard cases can be described by giving their index in this small set. These results have been used to build a method for generating very hard graph fo the 3-COL problem [Vla95] with the Brélaz algorithm [Bre79]. In this study, VLASIE showed with experimental results that the very hard instances occurs *only* when the graph are highly structured. By the Kolmogorov

complexity approach, he confirms a phase transition point given by experiments that distinctly separates an easy region of graphs from a hard one.

Even if there is no general result concerning polynomial problems, similar considerations have been made in [LV92] to show that only non-random lists can achieve the worst-case complexity for Quicksort (for instance, the sorted, almost sorted and reversed lists).

There is one important point to insist on: the notion of hard inputs created are specific to an algorithm[1]. The aim of VLASIE was clearly to create hardness for the Brélaz algorithm. One could argue that theorem A.3 does not hold anymore it there is a specific method that could solve the previous inputs in polynomial time. As VLASIE said, "one should not forget that there are many other ways to get compressible instances, eventually hard for the algorithm in question and on which a specific method would be of little interest. Moreover, we remember that the Kolmogorov complexity as a function of binary strings is not computable and the incompressibility is not a decidable property, so there is little hope that one can imagine a general method for solving the hard cases by detecting and exploiting their regularities." That means that the fact that Kolmogorov complexity of a hard instance is small is independant of the algorithm. It only depends on the instance itself.

Contributions of Kolomogorov complexity to our problem The particularity of Kolmogorov complexity in algorithms analysis, like in other various domains (see [Del98b] and [Del98a] for more applications), it supplied a clearer framework and point of view but does not directly give any clue for producing any algorithmic method.

So what can be exploited to solve our problem of generating representative inputs for our anytime algorithms?

The question of randomness have been largely discussed for several centuries, from Epicurius to Laplace [KLV97] and Kolmogorov complexity finally formalized it. But random objects have never been considered as real reflect of reality. With proposition A.1, we claim that random generation of inputs is not the right method to apply if we desire representative quality maps. The simple reason, but not evident at first sight, is that uniform distribution of probability is not the right distribution to represent real appearance probability of inputs.

In a second part, we discovered that this last result is all the more important because theorems A.2 and A.3 stated that "hard cases are the structured data". As a consequence, this part of the instances set can not be neglected if it forces the algorithm to the the worst-case behavior. This is especially true for NP-complete problems. We also saw that hard instances are specific to an algorithm and sometimes we prefer to call them hard inputs to insist on this fact.

Finally, we do not get any generation method of instances in general for building, because this problem is problem specific and more, algorithm specific. But we get some

1. That is why we should talk about inputs (of an algorithm) rather than instances (of a problem)

information on what not to do (random generation) and "where" the hardness is in the set of intances. As conclusion, we need a precise "structured method" to systematically generate instances to ensure a representative quality map.

The question of neglecting some parts of the set of instances that we think not representative is difficult to determine. The universal distribution could give us a possible approximation of this risk by considering the probability of these neglected instances.

A.4 Conclusion

In this chapter, we focus our attention on practical problems encountered in constructing anytime algorithms. Simple considerations on the TSP first lead to doubts concerning the easiness of building performance profiles. That is why we adopt a user-view (one can say naive) approach to try to solve the real problems in designing anytime algorithms.

In this study, we exhibited two questions that are usually not solved in general, and not even considered, in the studies concerning anytime algorithms design. These questions are:

1. choosing the quality criterion
2. collecting data for building the quality map and the performance profiles

These problems are generally considered as a work that the user of anytime techniques must define himself. As a matter of fact, it could seem natural since both these problems are strongly application dependent.

But we chose to focus on these questions because we thought that, even if only the designer of anytime algorithms can solve them, there are some pitfalls to avoid. With this practical approach to design of anytime algorithms, we exhibited some of these pitfalls with elementary but representative examples of current problems (sort problem and TSP) and tried to give some piece of advice.

As for the choice of the quality criterion, we saw that it had to be specific to the algorithm simply to have a sense for the treatment applied but that it also depends on the goal of the designer. As a matter of fact, the result of the algorithm is to be used in a certain context and can be reused as input to another algorithm. The choice of the quality criterion is always a trade-off between these two points of view.

The problem of building a performance profile is not a trivial one either. Obviously, the best data for constructing performance profiles is a good sample of actual input instances, but we considered the case where we have poor information about what sort of data will be used by the algorithm. Some considerations on realistic cases of instances showed us that the random choice of instances is rarely the right method to obtain a performance profile that predict the realistic behavior of an algorithm. The issue of using random problem instances for the empirical study of algorithms has already been critized in the past. See for example [Joh96], where JOHNSON explains that some TSP algorithms can be very efficient on large random instances, while they have great difficulty with structured

instances containing just 53 cities.

We introduced the Kolmogorov complexity theory to confirm this fact. It shows that strutured instances, *i.e.* not random instances, are the most current in the realistic cases. This shows that we need a fine analysis of the algorithm and of the generating method of the instances before generating the quality map. Kolmogorov Complexity shows a little more than "random is not ideal choice of input data". The combinatorial complexity results linked to Kolmogorv complexity show that very structured cases can lead with high probability to worst cases (in time). This is especially interesting in the case of TSP where the average case is far from the worst case. Kolmogorov complexity of a data (compared to another one, as absolute Kolmogorov complexity is not computable) can be used to say if it is an "average" case or if it is a "worst" case, so as the user of the algorithm can be sure that his algorithm will or will not have a worse behavior in reality. This means that Kolmogorov complexity can be used to analyse input data structure and to say if this data is realistic.

We have to mention that HORVITZ has studied anytime sorting in [Hor90]. This study enables to examine how the different sort algorithms behave and what a partial sot could mean. He defines four main properties of the random list that could transform into criteria to select a sort algorithm and into quality criteria. Our aim here is to show how difficult it can be to choose a quality criterion and we used sort, as HORVITZ did. Our study concerns the selection a quality criteria as his study, but we consider one more dimension that is the choice of inputs and the influence on the quality maps. More over, it seems that HORVITZ only examined cases of random lists. We showed that some structured list could provide "strange" behaviors (*non monotonic quality* for example). However, it is worth mentioning that the representation used ("Protos/Algo"), consisting in representing partially sorted lists, is an alternative to the quality graphs for analysis of the behavior of sort algorithms. This study can be considered to be complementary to ours on quality criteria.

Even if we did not produce any complete method for building performance profiles, Kolmogorov complexity helped us to produce a more formal explaination of phenomena observed on random data by HORVITZ [Hor90] and JOHNSON [Joh96] for example. From this point of view, the contribution of this theory is rather limited but we believe that Kolmogorov complexity can be a new approach to do this analysis and to generate representative quality maps and performance profiles. A lot of studies about generating "hard" instances and sorting instances in function of their Kolmogorov complexity are in progress, especially on NP-hard problems (see, for example, [Vla97] about the 3-COL problem). The analysis of the behavior of algorithms and the Kolmogorov complexity of the instances is very problem dependent. But the advantage of Kolmogorov complexity is mainly that it authorizes individual analysis of the instances of our problems, when we only disposed of statistical results before. On the condition of using this theory, we think that a "beginner" in anytime algorithms could avoid some pitfalls. However, this problem of analysing input data to predict the beahvior of an algorithm remains hard.

Annexe B

Considerations on Software environment

Anytime algorithms are particulary presented as a solution for real-time AI problems. As a consequence, it seems natural to introduce a discussion on the software environment we use to implement these algorithms. In this appendix, we present the influence of the different possible operating systems on the implementation and especially on time measure.

B.1 About the importance of a good time measure

The first parameter for evaluating performance of the anytime algorithms is obviously time as the framework of these algorithms is real-time. As we already said, the term "real-time" does not mean that our algorithms have to be fast, but that they will give results regularly, at precise time. As a consequence, it is necessary to be regular too when taking measures of quality. This regularity is essential to get the same computation time for the same operations. Without this, we immediatly loose the prediction capability of the performance profile. As a matter of fact, it is impossible in this case to reproduce in action results that we got with the performance profile. Moreover, remember that our objective is to exhibit interesting properties of anytime algorithms. If the time measure is not precise enough, we can not be sure observing all phenomena on the performance profile.

Instead of measuring real-time, we could have analyzed our algorithms like SEDGEWICK and FLAJOLET [SF96]. We decided not to do this analysis for two main reasons:

- with anytime algorithms we get one supplementary dimension to the analysis because of intermediate results when SEDGEWICK and FLAJOLET just analysis total computation times. This is a very complex approach, even for very simple algorithms like sort algorithms. This approach should have been very heavy.
- this is not the approach that common programmer (it. industry) will adopt because

of its complexity. It is simplier to test algorithms with efficient tools for real which the programmer disposes, especially if the algorihms to test are numerous. Finally, the programmer has to implement his algorithm. We think it is better for him (and here for us) to be confronted to real technical problems like the choice of an operating system, of a compiler, etc.

That are the reasons why we insisted on implementation of a very precise measure of time. The operating system to use to implement anytime algorithms must take this facts into account.

B.2 Possible operating systems

Here we describe the different possibilities we have in the set of all types of operating systems to ensure precise time measure. A quality we are looking for is the portability of our source code. That is why we will examine the case of Unix and similar operating systems. We will consider other operating systems like MS-DOS, for instance, to discuss about our requirements of our final choice.

B.2.1 Unix

We use two different implementations of Unix to do our tests: the one is Solaris, a product of Sun company, the other is Linux (kernel version 2.2). We chose these two implementations to observe two differents ways of programming the Unix standard. Linux and Solaris conform to the POSIX specifications [Dra87]. Remember that the main aim of POSIX is the portability of source code and that is what we hope by choosing Unix as our software environment. The POSIX documents describe the interfaces of the C language with the operating system. Losts of standard interfaces and ways to implement them can be found in [CDM98].

B.2.1.1 Unix theoretical capabilities

Traditionaly there exists two ways of measuring time with Unix [Ste92], corresponding to the two possible different natures of time for a process. These two types are: calendar time and CPU time. The first one is the physical time that pass, and the seconde one is the real consumption of the ressource, the CPU. Knowing that Unix does time sharing, that is shares the CPU between several processes, it is obvious that calendar time can not be appropriate for our measure of time. As a matter of fact, what we get by measuring calendar time is the sum of CPU time of all the processes. That leads us to choose CPU time that reflects the real comsumption of computation of the process. If the process were alone, the calendar time would be equal to the CPU time.

First, let explain the notions of precision and resolution. What we mean with resolution is the number of digit available to express our measure. The precision is the number of

digits that are significant for the measure. That is, for a unique number of computational operations, the number of digits in a computation time value that do not vary from a measure to an other. The precision is obviously always lower than the resolution. It is often expressed with an error (a percentage). For instance, we can hope a precision of 10ms in time measure with an error of 10%, *i.e.* a variation of +/- 0.5ms.

One C function, namely *times* [Lew93], defined in POSIX, that gives system CPU time and user CPU time. The system mode corresponds to the system commands (communication with the machine) and the user mode corresponds to the computation of the algorithms programmed by the user [CDM98]. Typically, a process that only contains pure calculus will mainly work in user mode. The time we measure will be the sum of user and system time. The *times* gives CPU time in function of the number of ticks in a second, defined by the value of *CLK_TCK*. But one must very carefully manipulate this value. If one compiles one's program on each platform working with Unix, the value may change [Ste92] [Lew93]. For instance, on SunOS 4.4.1, *CLK_TCK* is 60, on UNIX SVR4, it is equal to 100, on Linux 2.2 for x86 processors to 100, and for DEC Alpha processor to 1000. As a consequence, the theoretical precision we can hope on these different operating systems may vary too. That why POSIX theoretically ensures that the mimimal resolution of time measure is 10 milliseconds.

Note that there exists another more powerful[2] function, namely *getrusage*, that gives all sorts of information about the process and the ressources consumed, and particularly the CPU time. According to the handbook [CDM98], and it would be a serious advantage compared to *times*. *getrusage* gives times with a resolution of one microsecond.

Now we have to verify in practice that Unix is really able to be precise in giving the CPU time, at least with the POSIX precision of 10 ms. We test our programs on both Linux and Solaris.

B.2.1.2 Results of time measure with Unix

To verify if the effective precision of Unix (Linux or Solaris) is 10ms, we use a program that does a fixed number of elementary operations in C (for example, several sums and products). Then we launch the program several times and measure its execution time. The principle of this program is described by algorithm B.6.

0.29	0.30	0.29	0.30	0.29	0.29	0.30	0.29	0.29	0.29

TAB. B.1 - *Execution time measures of a program with constant number of operations*

If we do not take precautions, that is let run several other heavy processes like a word processor, a web navigator and all you can imagine, the results of the measure are far from what we expected. As a matter of fact, we can observe on table B.1 that the time measure is not constant.

2. When it will be fully implemented

ALG B.6 Program with constant number of operations

```
program algo_constant()
  begin
    t0 ← mesure_temps()
    computation()
    t1 ← mesure_temps()
    print(t1 − t0)
  end
```

	Measure 1	Measure 2	Measure 3
Points Nr.	100	100	100
Average value (s)	1.1525	17.0794	130.2302
Medium value (s)	1.15	17.08	130.22
Minimum (s)	1.15	17.07	130.22
Maximum (s)	1.16	17.13	130.45
Max-Min (s)	0.01	0.06	0.23
Error/medium value(%)	0.870	0.351	0.177
Std. Deviation	0.0000189	0.0000421	0.00652485

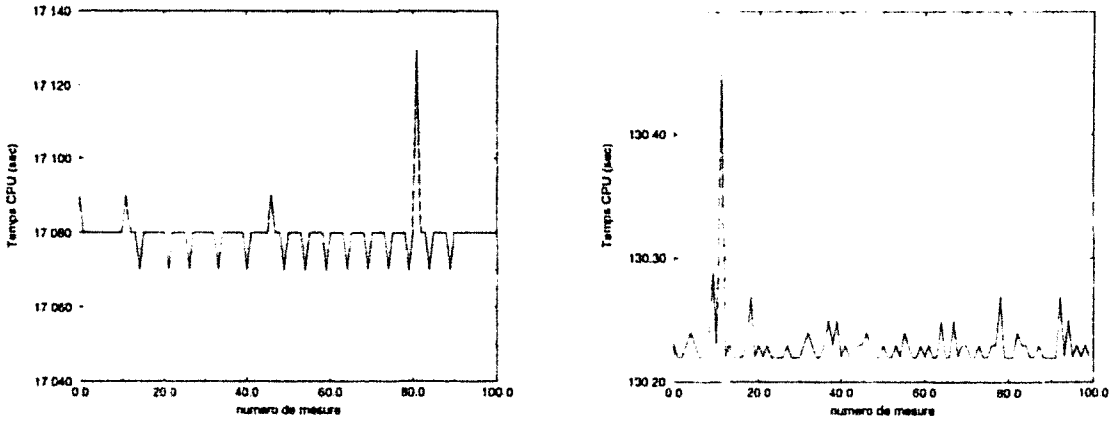
TAB. B.2 - *Execution time measures of a program with increasing number of operations with Unix*

By taking precautions and authorizing minimum number of processes running, we redo the experiment to get measures of increasing execution time. We simply did it by increasing the number of operations in the program. The results are on table B.2. This measures enable us to guess that error made in measuring increasing execution time is not an absolute error as it increases too.

Even if the relative error (Max-min)/Medium, is decreasing, the CPU time measure under Unix systems seems not reliable for two reasons:

- surprisingly CPU time can be disrupted by other processes running at the same time, especially when they are heavy and force the system to use swapping. It seems that this swapping time is taken into account in the total CPU time (in system AND in user mode)
- the CPU time measured doe not have a absolute error than could be substracted.

One more observation can be made on figure B.1. It shows the evolution, trial after trial, of the value of the CPU time measured so as to show its different values. Note that in table B.2, the medium value is always close to the mimimum value. That means that the picks of excess consumption of CPU are rare as observed on figure B.1. We explain this by the fact that standard Unix is not real-time and the process is exposed to interruptions by the scheduler and to overload, that make CPU time increase. And this picks phenomenon is all the more important as the total CPU time is large, because the chance to be interrupted increases.



(a) medium value = 17 sec

(b) medium value = 130 sec

FIG. B.1 – *Evolution with measures of execution time with Unix*

B.2.1.3 Conclusion on Unix

As a consequence of the lack of precision we observed, we can hardly choose a standard Unix platform to implement our programs. Remember that our primary objective is to construct the best prediction of performance of the anytime algorithm upon time and we need very precise tools to avoid any bias. One can propose to reiterate the same measures hundreds of times and to take the average value. The problem is that it can imply enormous amount of measure time, especially for complex problems. That is why we now consider other possibilities.

B.2.2 A real-time operating system

The problem we have with a standard Unix is not only a problem of precision, but also a problem of regularity. In a word, we need a system that can respect real-time deadlines. The first possibility that we examine was RT-Linux, a Linux 2.0 system extended with hard real-time functions. The principle is a real-time scheduler that consider the Linux kernel as a simple task with very low priority so as the real-time tasks that we program can be executed without any disrupt from the classic tasks. The advantage of RT-Linux is the portability as it respects real-time POSIX specifications. And that is why we found this solution interesting. But the drawback is that it needs a serious revision of our programming methods and it seems too heavy for just building performance profiles. The RT-Linux solution will be suitable for a entire platform that controls anytime algorithms because of its multitasking capabilities.

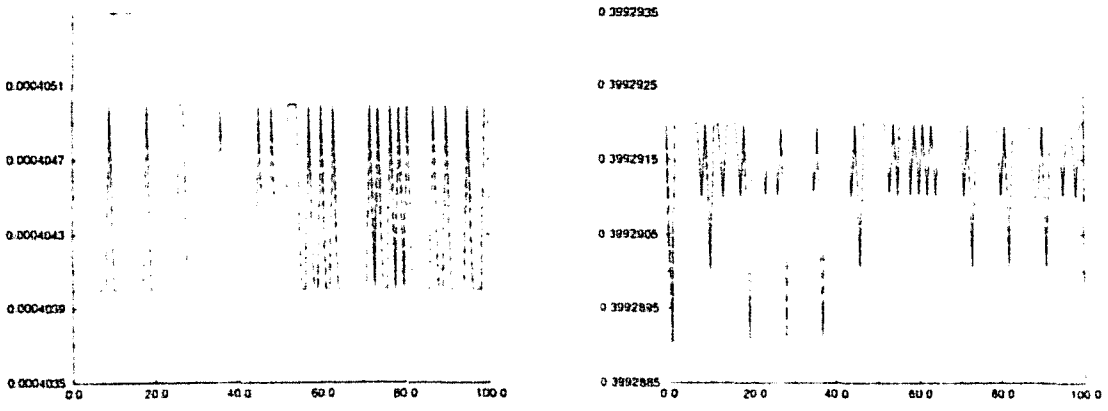
But at this time, our objective is more simple and we finally chose MS-DOS. All we need to build performance profiles is to interrupt the algorithm at precise time and always

get the same measure for the same process. That is what MS-DOS can do as:

- it is a monotask operating system. As a consequence our process can not be disrupted by another one
- you can have access to hardware to program a high resolution and, we hope, a high precision timer

	Measure 1	Measure 2	Measure 3	Measure 4
Points Nr.	100	100	100	100
Average value (s)	0.00040422	0.0039972	0.39929	-
Medium value (s)	0.0004045	0.0039975	0.399291	17.965880
Minimum (s)	0.0004040	0.0039970	0.399289	17.96587775
Maximum (s)	0.0004050	0.0039980	0.399293	17.96588575
Max-Min (s)	1e-6	1e-6	4e-6	6e-6
Error/medium value(%)	0.0247	0.025	0.001	3e-5
Std. Deviation	4.14e-7	3.8419e-7	7.8607e-7	1.7797e-6

TAB. B.3 - *Execution time measures of a program with increasing number of operations with MS-DOS*



(a) medium value = 4.045 ms

(b) medium value = 0.399291 s

FIG. B.2 - *Evolution with measures of execution time with MS-DOS*

In a word, MS-DOS is a real-time monotask system. And the time measure we made with the same method as with Unix systems give us very satisfactory precision as it can be observed on figure B.2 and table B.3. This precision is far better than what we get with Unix and is very regular. The maximum error we have is 6 μs. We can assume that we have a precision of 10 μ s with $\pm$ 5 μs of maximum absolute error.

The principle and the source code of our timer can be found in Appendix D. In the next section, we then present our measure protocol with this timer.

B.3 Description of our choice for experimentation

The principle of our program is to insert an algorithm between two time measure points. We assume that anytime algorithms do an iterative work (by definition, they do iterative improvements) and to stop the algorithm at constant time, we insert a counter for the number of iterations. Before starting the algorithm, we take a measure of time and fix a limit for the counter. Then the algorithm runs until the counter reaches the limit. At this moment, the algorithm exits and the second time measure is taken. Note that time measured at both points is time that passes, but the difference between the two measures represents the exact CPU time because MS-DOS is monotask. To get different execution times, we change the value of the counter. The program is able to repeat the same series of measure and to control the counter. As a consequence, with this program, we can get a set of graphs describing the quality as a function of time. This set will be used to build the performance profile. Our measure protocol first depends on the precautions we take in programming to ensure a good measure of execution time.

We assumed that the algorithms do an iterative work, but that does not mean that they are iterative algorithms. Lots of recursive algorithms do an iterative work too. The interruption to take the time measure is then executed by a special function in C, namely *longjmp*, that enables an immediate jump to the context saved before starting the algorithm. This results in negligible overheads for exiting the C function containing the algorithm. In particular, for a recursive algorithm, the interruption does not imply to go back all the recursion.

Another precaution consists in restarting the algorithm after each measure to avoid counting the interruption time in the execution time, and especially to avoid cumulative error due to repetitive interruption. When we take time measure, we do not use algorithms as interruptible ones but as contract ones. We always restart from scratch to avoid this cumulative error by interrupting and restarting from the point of interruption.

The following algorithms (B.7 and B.8) explain how we program the main part of our application and the modifications in the original algorithm to measure its execution time.

We assume in this program that quality must be calculated outside the algorithm. This point enables us to use all type of quality criteria and especially those that are expensive in term of computation time with regards to the one iteration step of the algorithm. The aim here is to discuss on the quality criteria and it is obvious that in normal use of anytime algorithms, the shortest is the computation of the quality, the best it is.

ALG B.7 Main program for measure of computation time of an anytime algorithm

```
program mesure_temps()
  begin
    for iter from 1 to iter_limit do
      init_données()
      for pt_mesures from 1 to pt_mesures_limit do
        compteur ← nb_boucles * mesures
        t0 ← mesure_temps()
        if (save_context(↑context) = 0) then
          algorithme()
        endif
        t1 ← mesure_temps();
        temps[pt_mesures] ← t1 − t0
        qualité[pt_mesures] ← calcul_qualité()
      endfor
    endfor
  end
```

ALG B.8 Modified algorithm for computation time measure

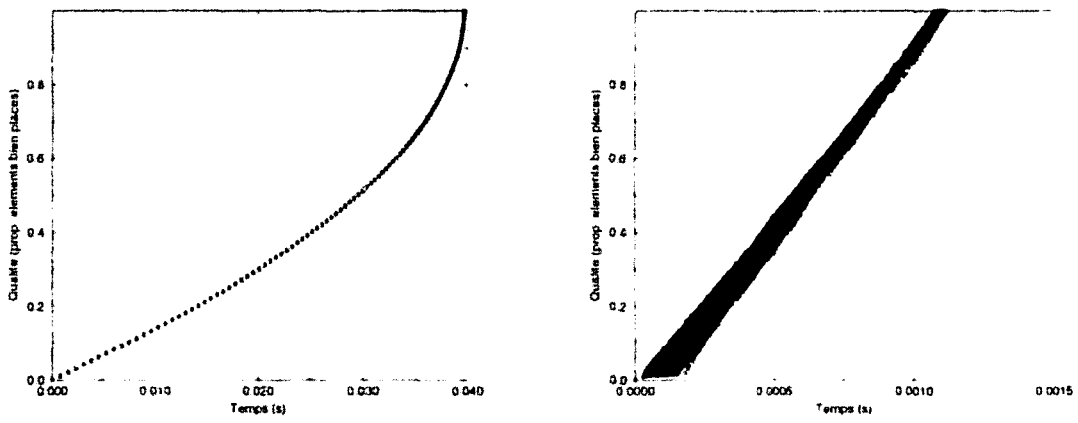
```
procedure algorithm()
  begin
    while (computation_end = False) do
      if (compteur = 0) then
        jump(↑context)
      endif
      compteur ← compteur − 1
      ALGORITHM MAIN PART
    endwhile
  end
```

Annexe C

Tests complémentaires de construction de cartes de qualité pour les tris

C.1 Tris sur des listes aléatoires

C.1.1 Critère q_{bp}

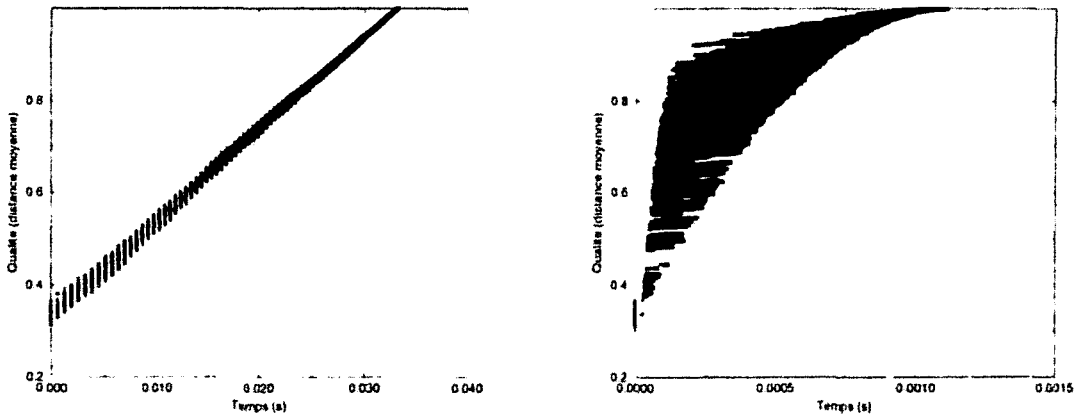


(a) Tri par sélection ordinaire

(b) Quicksort

FIG. C.1 - *Comportement d'un tri sur une liste de 1400 entiers pris entre 1 et 1400, mélangés au hasard - qualité mesurant la proportion d'éléments bien placés - 100 points de mesure, 200 listes*

C.1.2 Critère q_d

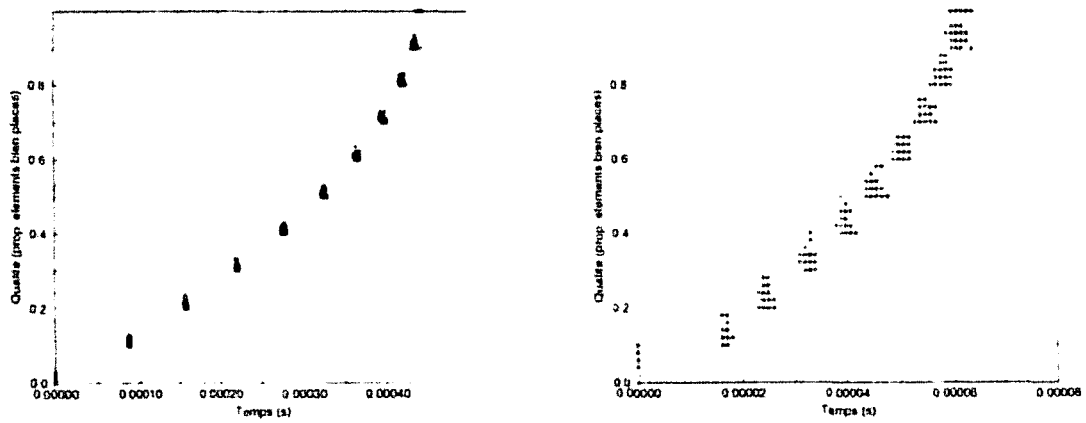


(a) Tri par sélection ordinaire (b) Quicksort

FIG. C.2 - *Comportement d'un tri sur une liste de 1400 entiers pris entre 1 et 1400, mélangés au hasard - qualité mesurant la distance moyenne normée à la position finale des éléments - 100 points de mesure, 200 listes*

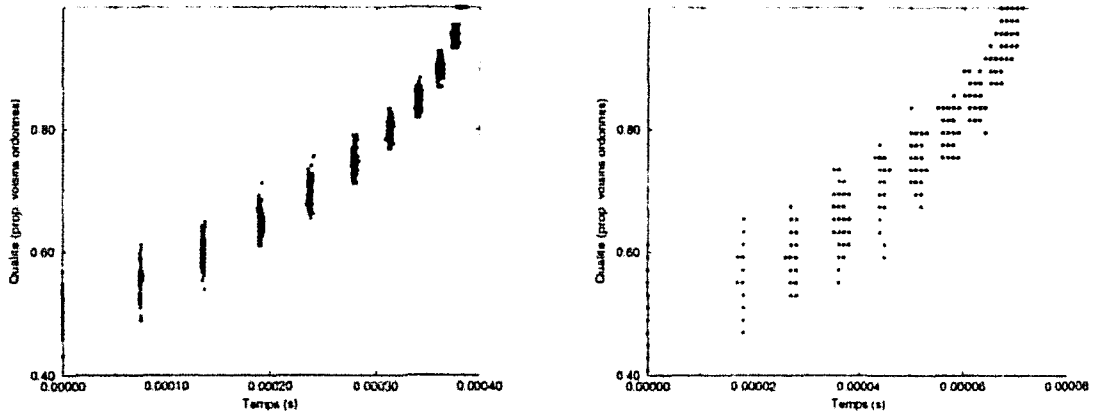
C.2 Influence de la taille de l'entrée

C.2.1 Tri par sélection ordinaire



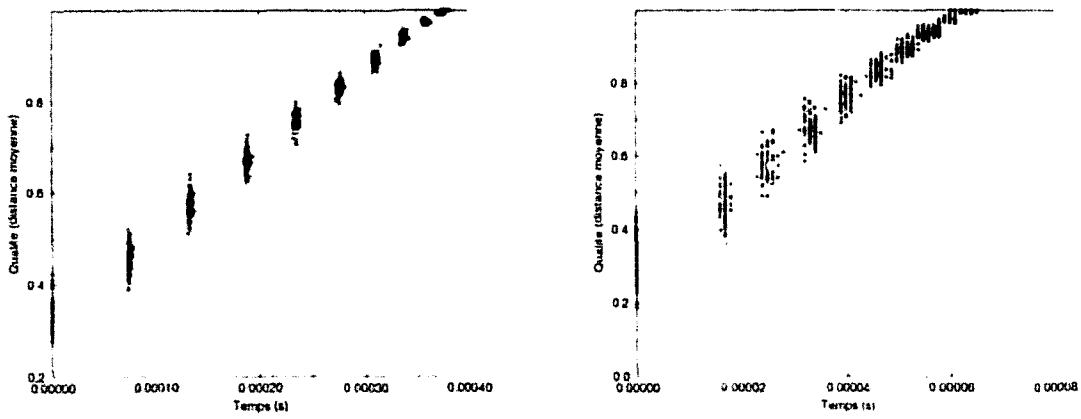
(a) n = 140 (b) n = 50

FIG. C.3 - *Comportement d'un tri par sélection ordinaire sur une liste de n entiers pris entre 1 et n, mélangés au hasard - qualité q_{bp} - 100 listes*



(a) n = 140 (b) n = 50

FIG. C.4 - *Comportement d'un tri par sélection ordinaire sur une liste de n entiers pris entre 1 et n, mélangés au hasard - qualité q_v - 100 listes*

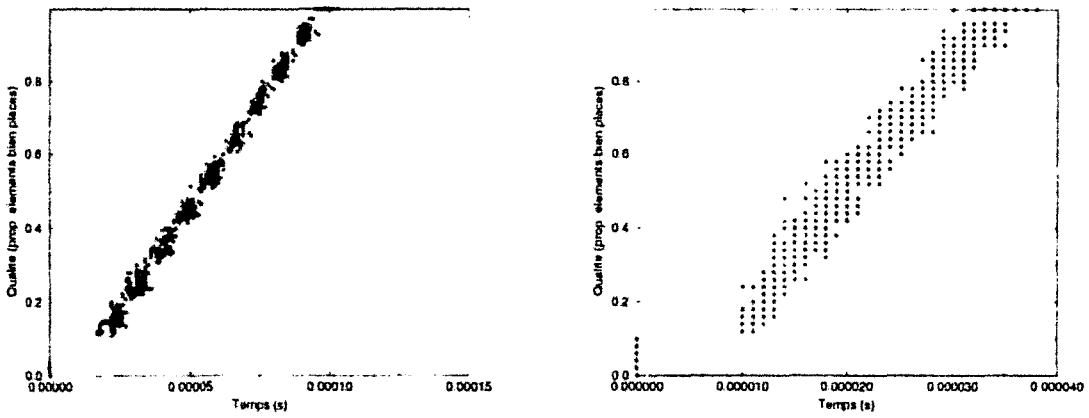


(a) n = 140 (b) n = 50

FIG. C.5 - *Comportement d'un tri par sélection ordinaire sur une liste de n entiers pris entre 1 et n, mélangés au hasard - qualité q_d - 100 listes*

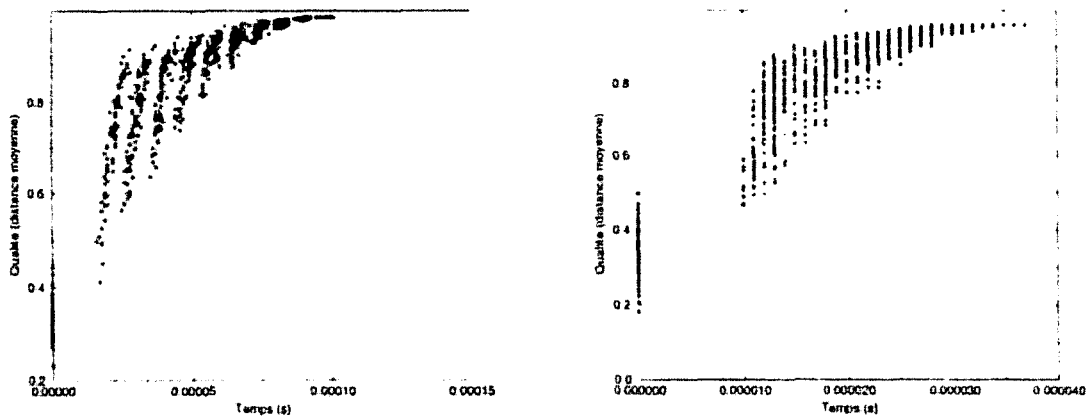
C.2.2 Tri rapide

Notons que pour le Quicksort, nous avons établi que le critère q_v n'était pas utilisable tel quel. Nous avons pu observer des décroissances de qualité en utilisant le quicksort avec ce critère de qualité. Aussi nous ne représenterons pas ici les compléments de résultats expérimentaux avec q_v.



(a) n = 140 (b) n = 50

FIG. C.6 - *Comportement d'un tri rapide sur une liste de n entiers pris entre 1 et n, mélangés au hasard - qualité q_{bp} - 100 listes*



(a) n = 140 (b) n = 50

FIG. C.7 - *Comportement d'un tri rapide sur une liste de n entiers pris entre 1 et n, mélangés au hasard - qualité q_d - 100 listes*

Annexe D

Timer pour la mesure sous DOS

Suite à nos essais dans un environnement multi-tâche (Unix), non-déterministe donc qui sied peu au temps réel, nous avons obtenu une précision de mesure du temps qui variait de quelques dizaines de millisecondes à deux ou trois centaines de millisecondes. L'erreur sur le temps CPU mesuré croissait avec la longueur du processus dont nous mesurions la longueur d'exécution. Nous avons voulu explorer la voie du DOS sur PC car c'est un système monotâche et nous espérions obtenir une régularité des mesures de temps bien plus grande, et ainsi augmenter notre précision. Dans cette annexe, nous exposons la solution adoptée pour programmer un *timer* sous DOS afin d'obtenir une précision d'un ordre mille fois plus petit que ce que nous obtenions avec Unix, et aussi une plus grande régularité dans les temps mesurés. Nous exposons donc ici les principes généraux de ce timer et donnons le code source correspondant.

D.1 Principe

Nous utilisons des ordinateurs basés sur l'architecture du PC-AT. Le principe du timer que nous voulons programmer est d'utiliser un circuit servant de référence pour la mise à jour de l'horloge dans ce type de machine. Chaque PC comprend au moins un Timer d'intervalle programmable i8254 ou équivalent. Ce circuit comprends trois timers 16 bits indépendants. Ce sont uniquement les timers 0 et 2 qui vont nous servir. Le timer 0 sert de timer principal (pour le rafraîchissement de l'horloge) et le timer 1 et le timer 2 s'utilise de façon générique dans les applications. Le schéma de fonctionnement de ces deux timers sont en figure D.1.

Nous allons donc utiliser les deux timers décrits plus avant. Comme nous pouvons l'observer sur la figure D.1, le timer 2 reçoit en entrée une fréquence dépassant le mégahertz. Il est capable de décompter de 1 à chaque tick d'horloge et nous pouvons donc espérer obtenir des mesures d'une précision inférieure à la microsecondes en comptant le nombre de ticks en entrée de ce timer. Malheureusement, le timer 2 seul n'est pas suffisant car c'est un compteur 16 bits et ne nous permet donc de compter que 65536 ticks d'horloge, après quoi il repart à zéro. La fréquence de remise à zéro du timer 2, comptant 65536 ticks à 1,193180 Mhz. est donc de 18,206481 Hz. Ceci correspond exactement à la fréquence du timer 0 et de l'apparition de l'interruption 08h. Nous pouvons donc exploiter l'apparition

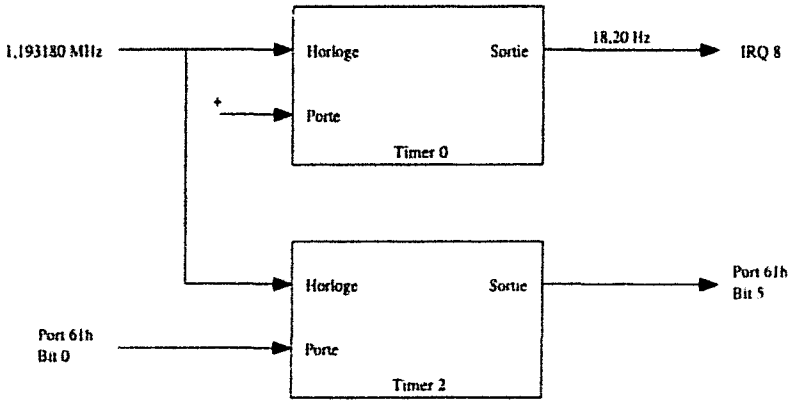


FIG. D.1 - *Système de synchronisation du PC*

de cette interruption pour incrémenter une variable qui comptera le nombre de remise à zéro du timer 2. La récupération du compteur du timer 2 complétera la mesure de temps. La formule de calcul de temps sera donc la suivante, avec cpt_{tmr0} et cpt_{tmr2} les compteurs du timer 0 et du timer 2 respectivement:

$$temps = \frac{cpt_{tmr0}}{18{,}206481} + \frac{(65636 - cpt_{tmr2})}{1193180}$$

Mais dérouter l'interruption 08h peut causer des comportements de la machine imprévisibles si de grandes précautions ne sont pas prises. Comme l'interruption 1Ch est déclenchée au même moment (ou plutôt juste après 08h) et est utilisable tout à loisir par le programmeur, nous utiliserons cette dernière interruption à la place de la première. Finalement, il restera à synchroniser le départ de l'algorithme avec une mise à zéro du timer 2. Ceci se fera simplement en attendant une première interruption 1Ch du timer 0.

Nous n'entrons pas plus dans les détails techniques. Le lecteur intéressé par les modes de fonctionnement des timers et les principes d'interruption entre autre se référera à [Edi87] et [Gil99].

D.2 Code Source

```
\************* Timer pour DOS en Turboc 2.01 *************/
#include <stdio.h>
#include <dos.h>
#include <conio.h>

long cpt_tmr0 = -1;
unsigned mot;
int flag = 0, octet;

void interrupt far IT_timer0(){
```

```
   /* pulse positif sur la porte du timer 2 */
   octet = inp(0x61);  /* lit la valeur courante de la porte */
   octet = octet | 1;  /* positionne le bit 0 a 1            */
   outp(0x61, octet);  /* active le compteur                 */
   octet = octet & ~1; /* positionne le bit 0 a 0            */
   outp(0x61, octet);  /* desactive le compteur              */
   /* flag pour indiquer le depart a reception de l'IT 1C */
   flag = 1;
   /* increment du compteur de ticks du timer 0 */
   cpt_tmr0++ ;
}

double main() {
   int octet;
   long i, j, k;
   double temps;
   void (interrupt far *gest_original)();

   /* desactivation du compteur du timer 2 */
   octet = inp(0x61);   /* lit la valeur courante de la porte */
   octet = octet & ~1;  /* positionne le bit 0 a 0 */
   outp(0x61, octet);   /* desactive le compteur */


   /* programmation du mode et de la valeur du timer 2 */
   outp(0x43,0xB2);
   outp(0x42, 0);   /* LSB */
   outp(0x42, 0);   /* MSB */

   /* Sauvegarde de l'ancien vecteur d'IT */
   gest_original = getvect(0x1C);

   /* Derouter l'interruption 1C sur routine_IT */
   disable(); /* pas d'IT pendant setvect */
   setvect(0x1C, IT_timer0);
   enable(); /* IT de nouveau autorisees */

   /* init et attente de la premiere IT 1C */
   cpt_tmr0 = -1;
   mot = 0;
   while(flag != 1);

   /* demarrage de l'algorithme à mesurer */
   for (i = 0; i < 10000; i++){
      for(j = 0; j < 10000; j++) {
```

```
            k= j - i;
        }
    }
    /* fin de l'algorithme */

    /* lecture du timer 2 */
    outp(0x43, 0x80); /* latch sur le compteur */
    mot = inp(0x42)&0xFF; /* LSB */
    mot |= inp(0x42)<<8;  /* MSB */

    /* Restitution de l'ancien vecteur d'IT */
    disable();
    setvect(0x1C, gest_original);
    enable();

    temps = 54.92549e-3 * cpt_tmr0 + (65536L-mot)/1.19318e6;
    printf("cpt_tmr0 = %ld\t65536-mot = %ld\n", cpt_tmr0,
                        65536L-(unsigned long)mot);
    printf("%f secondes\n", temps) ;

    return (0);
}
```

Annexe E

Article publié à IJCAI'99

Arnaud Delhay, Max Dauchet, Patrick Taillibert, Philippe Vanheeghe. *Maximization of the Average Quality of Anytime Contract Algorithms over a Time Interval*, Proceedings of International Joint Conference on Artificial Intelligence 1999, pp. 212-217, Stockholm, Sweden.

Maximization of the Average Quality of Anytime Contract Algorithms over a Time Interval

Arnaud Delhay*
LIFL - ISEN
F-59046 Lille
adel@isen.fr

Max Dauchet
LIFL
F-59655 Villeneuve
d'Ascq
Max.Dauchet@lifl.fr

Patrick Taillibert
Thomson-CSF Detexis
F-78190 Trappes
Patrick.Taillibert@detexis.thomson-csf.com

Philippe Vanheeghe
ISEN
F-59046 Lille
pva@isen.fr

Abstract

Previous studies considered quality optimization of anytime algorithms by taking into account the quality of the final result. The problem we are interested in is the maximization of the average quality of a contract algorithm over a time interval. We first informally illustrate and motivate this problem with few concrete situations. Then we prove that the problem is NP-hard, but quadratic if the time interval is large enough. Eventually we give empirical results.

1 Introduction

Hard problems like planning or decision making cannot be reasonably treated by complete methods. That is the reason why [Dean and Boddy, 1988] first considered anytime algorithms, also called flexible algorithms in [Horvitz, 1988]. These algorithms offer a trade-off between time and performance. They are characterized by a *Performance Profile* [Grass, 1996] that enables a prediction about the quality of the results given by the algorithm depending on the execution time duration. This method has been used to solve several problems in various domains like robot control [Zilberstein and Russel, 1993], knowledge-based computation [Mouaddib and Zilberstein, 1995] and reactive agents [Adelantado and de Givry, 1995].

Quality is not the essential characteristic of a computation result: what really matters is its utility. The intuitive idea is that, in many situations, the utility of a result decreases over time, and a result of medium quality rapidly obtained is more useful than a result of high quality obtained after a long time [Zilberstein and Russel, 1996]. But, when the algorithm operates on an uncertain environment, utility can be of another nature. This is the case in the following examples, which are all associated to a "crisis situation":

- a person (P) has to give his boss (B) a report in the morning. P only knows that B will ask for the report at some time between 8 a.m. and 11 a.m. The problem for P is that trying to achieve the best quality would be a good strategy only if the report were claimed at 11. If it is not the case, no report at all is available! Hence it seems better to ensure a medium quality draft for 8 a.m. and, once the draft is ready, to start to write a better quality report, expecting that the claim will occur late in the morning.
- Every time the enemy is going to launch a satellite, only the temporal window on which the event will occur is known in advance. To perturb the launching, some electromagnetic jamming action must be set up (planes have to take-off, lures must be activated...). All these actions take time and the best jamming is useless if achieved after the launch. What could be considered is to set up the jamming in order to ensure the best *average* quality on the time interval, then to maximize the utility (in the long term).
- When a tornado is announced, very little time is available to prepare oneself (and one's house) for any possible destruction. Hence it is important to achieve the best "utility" of the protections set in place, e.g. to ensure that minimal actions have been taken first (securing the kids), before improving the quality of the protections (nailing down the shutters).

In the previously described situations, an interruptible algorithm would be the best solution: at the time of the event, the best possible quality would be achieved. Unfortunately, interruptible algorithms are not always available since:

- none of the available algorithms might be interruptible,
- anytime algorithms might result from the composition of elementary interruptible algorithms. In that case, the result is of the contract kind [Zilberstein and Russel, 1996],
- contract algorithms can be transformed into interruptible ones [Zilberstein and Russel, 1996] but, notwithstanding the fact that the execution time is 4 times longer, this method only applies if the contract durations can be chosen freely (exponential series) which is generally not the case.

Finally, even with a genuine interruptible algorithm, situations exist in which the contract case re-occurs: if applying the results of the algorithm causes a change in the environment, it is no longer possible to let the algorithm continue in order to improve the solution from the previously

* This author is supported by the *Délégation Générale pour l'Armement (DGA)*, French Ministry of Defense

obtained one; it must be restarted from scratch. For instance, this might occur in counter-measure applications, if the result of the interruptible algorithm is a jamming action which itself provokes a counter-jamming decision from the enemy.

In [Delhay *et al.* 1998], we proposed partial solutions to solve these kinds of problems consisting of maximizing the average quality over a time interval. But the results achieved are limited to convex quality functions (whose second derivative is positive) which are the less probable ones in real applications.

In this paper, after restating the problem of maximizing the average quality of an anytime contract algorithm over a time interval (section 2), we present general results about any kind of quality function approximated by a stepwise function. We prove that the problem is NP-hard, but becomes quadratic if the time interval is large enough (section 3). We then present empirical results which augurs well for the practical applicability of the approach (section 4).

2 Maximizing the average quality over a time interval

We use the notion of contract algorithms which was first coined by Zilberstein [Zilberstein, 1993], even though they appeared before, like RTA* [Korf, 1985]. Contract algorithms can also result from the composition of anytime modules. In this paper, we assume that the performance profiles are deterministic functions of time. It is sometimes difficult to construct them with such a confidence, but they are good approximations of the performance profiles used in real situations.

2.1 Informal presentation

A typical example is the following: an attack might happen with a uniform probability over a given time interval, and a contract algorithm, whose performance profile is increasing over time (figure 1) is available to counter-attack. The problem then consists in determining how to best prepare the counter-attack in order to get the best chance of survival over the time interval.

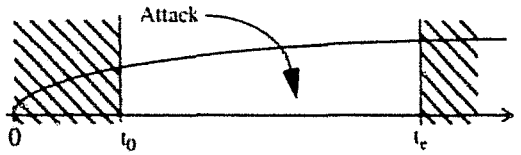


figure 1: The attack might happen at any time over $[t_0,t_e]$

An answer to the attack must be given between t_0 and t_e and it is possible to begin the computation at time t = 0. To answer to this problem, several solutions can be considered.

First (figure 2) we activate the algorithm for a short contract (t_1): in that case, the result has a relatively poor quality but this quality is available on a relatively long time interval ($t_e - t_1$).

figure 2: short but bad preparation

To get a better quality, we have to start the algorithm with a longer contract (t_2): the quality of the result is better, but most of the time, on $[t_0, t_2)$, no "quality" (that is, no protection from the attack) is available (figure 3).

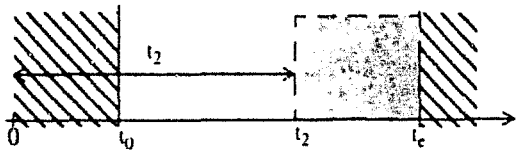


figure 3: good but slow preparation

These two cases lead to a simple observation: if the algorithm starts with a (sufficiently) short contract (t_1), then the remaining time can be used to restart the algorithm with a contract t'_2, to get a better result

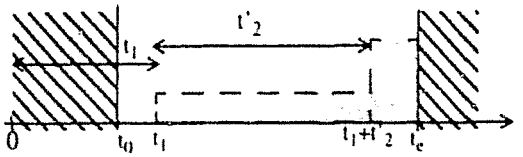


figure 4: mixed preparation

Hence an effective method is to start the contract algorithm several times to get a good cover of the time interval and a minimal quality early. We only have to respect two conditions:

- The sum of all contracts must be strictly lower than the length of the interval where it is possible to compute
- Every new contract must be longer than the previous one to improve the quality of the result

Remark:

Note that to maximize the average quality, we never execute more than one contract before t_0. Should we do that, all contracts before t_0, except the last, would be useless.

2.2 Formalization

Now let us give the definition of the average quality over an interval where $\langle\Theta_n\rangle = \{\theta_1, \theta_2, \ldots, \theta_n\}$ denotes the duration of successive runs of the algorithm (contracts).

definition 1 : *integral quality*

Let f be the performance profile of a contract algorithm A. The integral quality *of A over a time interval $[t_0, t_e]$, relative to a choice $\langle\Theta_n\rangle$ of n contracts with performance profile f, is defined as follows:*

$$Q(f,(\Theta_n)) = min(0,\theta_1 - t_0)\ f(\theta_1) + \sum_{i=1}^{n-1} f(\theta_i)\cdot\theta_{i+1} + f(\theta_n)\left(t_e - \sum_{i=1}^{n}\theta_i\right)$$

The above definition is only usable if the contracts respect the sum constraint, that is: $\sum_{i=1}^{n}\theta_i < t_e$

The average quality $\overline{Q}(f,<\Theta_n>)$ *is equal to the integral quality divided by the length of the interval* $[t_0, t_e]$. $Q(f)$ *is the supremum of the integral quality, and* $\overline{Q}(f)$ *the supremum of the average quality.*

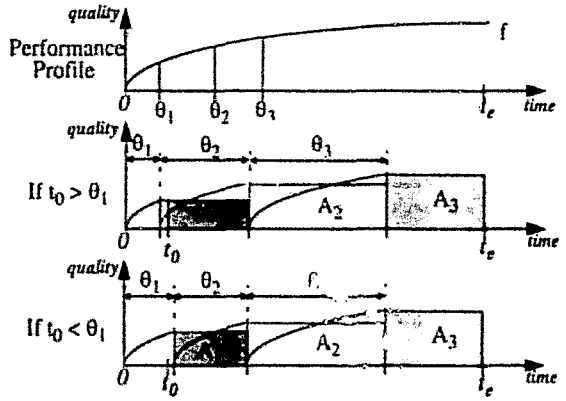


figure 5: Performance Profile and Average Quality

2.3 The analytic case

In [Delhay *et al.*, 1998], we studied the maximization of the average quality for continuous and derivable performance profiles, leading to preliminary analytic results that partially cover functions with constant curvature, that is the convex case and the concave case for one contract. These results are shown below.

A first theorem gives the value of a single contract. The linear performance profile is a limit and the single contract equals $t_e/2$. For convex performance profiles, the single contract is greater than $t_e/2$. In the concave case, it is lower than $t_e/2$. A second theorem states that, for convex performance profiles, there is no need to start several contracts over $[0,t_e]$, as a single contract always gives the best average quality. The value of this contract θ_1 is analytically defined by a simple equation.

$$\theta_1 = t_e - \frac{f(\theta_1)}{f'(\theta_1)}$$

The point we can add is that there is at least two contracts in the concave case because of the first theorem. As a matter of fact, as the single contract is lower than $t_e/2$, it is possible to add a contract greater than the first one, but respecting the sum constraint, that improves the average quality.

3 Average quality for stepwise constant performance profiles

Because of the difficulties involved in solving this problem in the continuous and derivable case, we chose to solve the problem by approximating the performance profile in order to get a discrete problem. We first tried using a stepwise linear function. Even with this approximation, we did not manage to exhibit interesting properties. Then we approximated the performance profiles with stepwise constant functions. This approach not only enables us to avoid difficulties due to the continuous and derivable performance profiles, but also makes sense in the common representation of performance profiles, that is the discrete tabular representation. Moreover, lemma 1 states that the average quality is approximated with the same error as the error on performance profile itself. In particular, this is true when approximating the continuous performance profile f with a stepwise function.

lemma 1 *: approximation lemma*

if $|f-g| < \varepsilon$ *then* $|\overline{Q}(f) - \overline{Q}(g)| < \varepsilon$

Proof: This is a property of integrals applied to definition 1.

So, hereafter the performance profiles are stepwise constant functions, either originally, or by approximation. A stepwise constant function is a function such as, for a finite set of thresholds $\{\theta_1,\ldots,\theta_n\}$, f is constant on every interval $[\theta_i, \theta_{i+1})$. The following lemma allows us to treat the maximization problem as a discrete problem, by only examining the steps of the performance profile instead of all values in the interval $[0,t_e]$.

lemma 2 *: (stepwise function lemma)*

To maximize average quality, it is sufficient to choose all contracts θ_i *in the set of thresholds of the stepwise function.*

Proof:

If a contract θ is in the interval (θ_i,θ_{i+1}), replacing θ by θ_i increases the average quality. It can be checked by calculating the difference between the two average qualities (with and without θ).

Thanks to lemma 2, we call this (discrete) problem MAXQSF (MAXimization of the average Quality for a Stepwise Function). MAXQSF(f,t_e) denotes the maximum of the integral quality of an increasing stepwise function f over the time interval $[0,t_e]$. In the next subsections, the tractability of MAXQSF is considered.

3.1 MAXQSF is NP-hard

The following theorem states that the MAXQSF problem is intractable in general.

theorem 1 *: MAXQSF(f, t_e) is NP-hard*

The proof consists in reducing polynomially the Knapsack problem to MAXQSF. The Knapsack (<A>,b) problem is to find a part of $<A>=\{a_1,a_2,\ldots,a_n\}$, a set of naturals, whose sum of its elements is maximal and lower than a natural b. We

assume that the a_i are sorted and distinct. This restriction is still NP-complete. The reduction consists in building a stepwise constant function $f_{<A>}$ of 2n+1 steps that gives an instance of MAXQSF(f,t_e) for any instance of Knapsack(<A>,b).

To do so, we use the fact that the increase of average quality obtained by adding a contract θ' between two consecutive contracts θ and θ'' only depends on θ and θ'' and θ_φ (the last contract).

The variation of the average quality induced by the choice of introducing a threshold as a contract depends on the other choices and is not easy to evaluate. That is why $f_{<A>}$ is constructed by alternating the thresholds (odd numbers) that corresponds to the elements of <A>, and thresholds (even numbers) **that are chosen to necessary belong to the optimal choice of the best average quality**. The even thresholds "isolate" the effects of a choice in the set of odd thresholds; hence, the improvement of the average quality obtained by adding an odd threshold (element of <A>) only depends on the two even thresholds surrounding it[1]. That is the reason why we chose them so that if the corresponding element α of <A> is in the solution, the improvement of the average quality is α. As a consequence, maximizing the average quality will maximize the load of the knapsack.

The reduction also requires that optimizing the average quality satisfies the sum constraint of the Knapsack problem. This is obtained by choosing t_e, the end of the time interval, such that:

$$t_e = b + \sum_{i \text{ even}} s_i + 0.5$$

Since the sum of all contracts in MAXQSF must be less than t_e and that all the even thresholds belong to the solution by construction, we therefore have:

$$\sum_{i \text{ odd}} s_i \le b$$

which is the sum constraint of the Knapsack problem.

We do not have enough space to include all the steps necessary to perform the reduction we have presented. Let's just say that the choice of the f function is done such that $f_i - f_{i-1} = K^{4n-i}$ with K great enough (but polynomially linked to the size of <A>) and that the s_{2i} are chosen such that $s_{2i} < a_i + 1/(3nK^{2n})$. This construction is polynomial in the size of <A> which proves that MAXQSF is NP-hard. For a detailed proof, see [Delhay and Dauchet, 1999].

Theorem 1 looks like an instance of theorem 4.2 presented by [Zilberstein and Russel, 1996] in the framework of composition of anytime algorithms. However, it is not the case: we look for the best average quality, whereas they look for the best final quality after a fixed contract, the most important difference being that we do not know the number of contracts necessary to get the best average quality.

[1] *also on the last contract (θ_φ), but, since it's an even one, it belongs to the optimal choice*

3.2 MAXQSF with no sum constraint is quadratic

Our problem is intractable in general. The main difficulty comes from the search over the set of thresholds assuming that the sum constraint does not allow to take any subset of thresholds. As in Knapsack, it is necessary to judiciously choose the candidate contracts. So we could suppose that:

$\sum_{i=1}^{p} s_i < t_e$ where s_i is a threshold of the performance profile f.

Hence it is possible to add any contract because the time interval is large enough to contain all contracts. That restricts MAXQSF and leads to a lower complexity algorithm.

The idea of this dynamic programming algorithm is founded on lemma 3 which allows the division of the set of combinations of thresholds into distinct subsets composed of the combinations finishing by a fixed threshold s_i. The lemma allows to iteratively compute $Q(f,<\Theta|\{s_i\}>)$, noted $MAXQ(s_i)$ for subset of combinations finishing by s_i, with s_i from s_1 to s_p, to finally give $MAXQ(s_p) = Q(f,<\Theta|\{s_i\}>)$ is the ordered set Θ finishing by s_i.

lemma 3 :

Let s_i and s_j be two thresholds of <s>, the set of thresholds of f, with $j<i$.
For any s_i, let $MAXQ(s_i)$ be the maximum of integral quality with s_i as the last contract and $TR(s_i)=(t_e-\theta_1-\ldots-s_i)$ the remaining time for the optimal choice of contracts with s_i as the last contract.
Then $MAXQ(s_i) = max_j(MAXQ(s_j) + (TR(s_j) - s_i) \cdot (f(s_i) - f(s_j)))$

and especially, $MAXQ(s_p) = Q(f)$, where s_p is the last threshold of <s>.

Proof:

- It can be immediately proved that $Q(f,<\Theta|\{s_j,s_i\}>) = Q(f,<\Theta|\{s_j\}>)+(TR(s_j)-s_i).(f(s_i)-f(s_j))$. By using the maximum for both members, the result of the lemma comes as $TR(s_i)$ only depends on s_j. Indeed, $TR(s_j)$ is the remaining time for the optimal choice of contracts with s_j as the last contract.
- For the second part of the lemma, note that s_p is necessary in the optimal choice of contracts. It could always be added to the choice of contracts (as there is no sum constraint) and improves the quality. By definition, $MAXQ(s_p)$ gives the maximal integral quality at the last step of iteration.

The complexity of the algorithm is $O(n^2)$, with n the number of thresholds.

Algorithm:

Only one threshold $\theta_1 \leftarrow s_1$ and $TR(s_1) \leftarrow t_e - max_i(t_0, s_1)$
$MAXQ(s_1) \leftarrow TR(s_1) \cdot f(s_1)$
$CHOICE(s_i)$ denotes the optimal choice of contracts with s_i as the last contract.

For i *from* 2 *to* p *Do*
  $Q_{temp} \leftarrow 0$
  $TR_{temp} \leftarrow 0$
  $CHOICE_{temp} \leftarrow \emptyset$
  For j *from* 1 *to* i-1 *Do*
    If $MAXQ(sj) + (TR(s_j) - s_i) \cdot (f(s_i) - f(s_j)) > Q_{temp}$
    Then
    $Q_{temp} \leftarrow MAXQ(s_j) + (TR(s_j) - s_i) \cdot (f(s_i) - f(s_j))$
    $TR_{temp} \leftarrow TR(s_j) - s_i$
    $CHOICE_{temp} \leftarrow CHOICE(s_j) \cup \{s_i\}$
    Endif
  Endfor
  $MAXQ(s_i) \leftarrow Q_{temp}$
  $TR(s_i) \leftarrow TR_{temp}$
  $CHOICE(s_i) \leftarrow CHOICE_{temp}$
Endfor
$Q(f) \leftarrow MAXQ(s_i)$
$CHOICEMAX \leftarrow CHOICE(s_i)$

4 Empirical results

The results obtained so far do not take into account the duration of the deliberation itself. It might be a serious impediment to the practical application of our approach since we proved that the general problem is NP-hard! Fortunately, achieving the optimal solution is not a necessary condition for applying our method: a set of contracts approaching the optimal value of the average quality on the time interval is sufficient. That is the reason why we conducted a set of experiments designed to estimate the practical complexity of our problem. And the result was far better than expected: for all the cases we studied, the average quality achieved by the best choice of a set of 2 contracts was always at a distance lower than 2.75% from the quality of the optimal choice.

These results concern a family of monotonic functions which approximate the quality functions most often encountered in practical cases. That is the reason why we think that these results can be of some general interest.

The experiment was twofold. First, we studied the contribution of the shape of the quality function to the value of the average quality. For that, we considered the family of functions defined by the following equation where the parameter "a" permits the control of the curvature of the function as shown in the associated graphics.

$$f(t) = \frac{a \cdot t}{(a-1) \cdot t + 1}$$

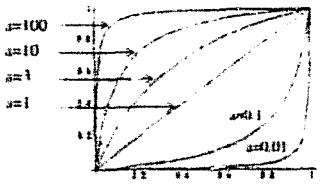


figure 6: performance profile for experiments

Every function ("a" varying from 1 to 205 incremented by 5) was approximated by a stepwise function of 50 steps for which we computed the optimal average quality; we also computed the error w.r.t. this optimal value when considering smaller sets of contracts. We relied upon integer programming in order to avoid the classic problems of floating point numbers and took advantage of the sum constraint to limit the computation time. The results are summarized in the following table where a is the curvature, n is the number of contracts and ● is the optimal solution:

a \ n	1	2	3	4	5	6
0.1	●					
1	●					
5	10.83%	0.23%	●			
25	12.33%	2.47%	0.25%	●		
45	11.59%	2.71%	0.59%			
65	10.75%	2.63%	0.6%	0.01%	●	
85	9.97%	2.44%	0.53%	●		
105	9.52%	2.36%	0.46%	0.01%	●	
125	8.98%	2.29%	0.45%	0.02%	●	
145	8.55%	2.22%	0.39%	0.02%	●	
165	8.29%	2.02%	0.35%	0.01%	●	
185	7.96%	1.83%	0.28%	●		
205	7.54%	1.68%	0.31%	●		

table 1: Error for limited number of contracts

For $a \leq 1$ the optimal quality is obtained with only one contract which was the expected result because of theorem 2. In the other cases, it is clear that the optimal quality obtained with 2 contracts is very close to the best result obtained notwithstanding the number of contracts.

We also investigated the influence of the temporal location of the time interval (t_0), which was set to 0 in the first experiment, while t_c-t_0 remained equal to 1. This might be of importance for many applications when time is available before the "attack" might occur. Here again, as shown on the following figure, restricting the deliberation to the computation of only 2 contracts gives very accurate results (even if only one contract rapidly gives very good results):

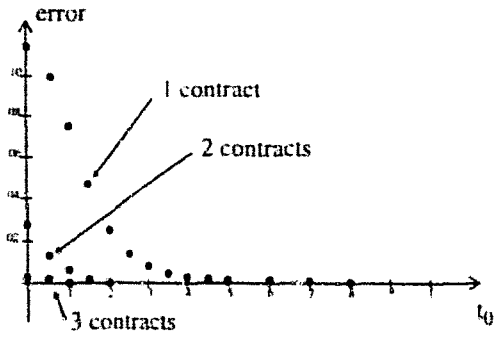


figure 7: Influence of t_0 on the number of contracts

In that experiment, the curvature was set to 51 and, since t_e was increased, the "sum constraint heuristic" became less efficient; hence we operated with a stepwise function of only 30 steps to keep the computation of optima tractable.

5 Conclusion

The aim of this paper was to present an **off-line** optimization of the time allocation for a contract algorithm so as to get the best chance of survival over a time interval. So far, no analytical method for solving the problem of maximizing the average quality over a time interval is known in the general case. That is the reason why we proposed solutions for discrete performance profiles. This restriction is not a real impediment since:

- a discrete tabular representation for the performance profiles is not a rare occurrence,
- the average quality resulting from a discrete performance profile can be as close as necessary to a given continuous performance profile (lemma 1).

We showed that the so-called MAXQSF problem is NP-hard in general, and quadratic if the time interval is large enough. This, unfortunately, is not frequently encountered in practical applications. Nonetheless, the experiments we carried out lead us to think that the "practical" complexity of that problem is quite low, which makes it possible to use the approach in real-time applications. Further studies to find a theoretical explanation of this behavior could prove very interesting.

The average quality of a contract algorithm A can be defined by:

$$\frac{1}{t_e - t_0} \int_{t_0}^{t_e} f^*(t)dt$$

where f^* is the quality of an interruptible algorithm A^* associated to A by the relation: $A^*(t) = A(\theta_i)$ where θ_i is the last contract executed at t.

[Zilberstein and Russel, 1996] gives a simple construction to transform a contract algorithm A of quality f into an interruptible algorithm A^* of quality f^* such as $f^*(4t)>f(t)$. The first difference with our study is that we consider the case of a given time interval to optimize the average quality. In our case, this leads to a construction of A^* which optimizes the average quality. The second difference is that our problem permits cases where the length of each contract is imposed by a method, when Zilberstein and Russel assume that the length of each contract can be arbitrarily chosen.

Note that a uniform probability of appearance of the attack over the interval has been considered. There are situations where this probability is not constant, such as with a gaussian. A future study could concentrate on this point.

Another extension could concentrate on our evaluation criteria, that is the average quality. Even if this criteria gives the best statistical results, there could exist others, for example that could take the number of contracts into account (a good solution therefore is a choice of few contracts).

Acknowledgments

I am especially appreciative of the discussions I had with Dominique Lohez at different stages of this work. His remarks and suggestions were very valuable.

References

[Adelantado and De Givry, 1995] M. Adelantado, S. De Givry, *Reactive/anytime Agents, Towards Intelligent Agents with Real-Time Performance*, IJCAI'95, Workshop on Anytime Algorithms and Deliberation Scheduling, August 21-25, 1995, Montreal, Canada.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy, *An Analysis of Time-Dependent Planning*, AAAI 88, August 1988, Saint Paul Minnesota.

[Delhay *et al.*, 1998] Arnaud Delhay, Max Dauchet, Patrick Tabaste, Philippe Vanheeghe, *Optimization of the Average Quality of Anytime Algorithms*, Workshop on Monitoring and control of real-time intelligent systems, ECAI'98, August 1998, Brighton, pp. 1-4.

[Delhay and Dauchet, 1999] Arnaud Delhay and Max Dauchet, *Complexity of the maximization of the average quality of anytime contract algorithms*, Internal Report LIFL-99-09, Laboratoire d'Informatique Fondamentale de Lille, 1999, ftp://ftp.lifl.fr/pub/reports/internal/1999-09.ps.gz

[Grass, 1996] Joshua Grass, *Reasoning about computational resource allocation, An introduction to anytime algorithms*, Crossroads, ACM student magazine, University of Massachusetts, September 1996, http://anytime.cs.umass.edu/~jgrass/school/papers/racra.html.

[Horvitz, 1988] Eric J. Horvitz, *Reasoning under varying and uncertain resource constraints*, Proceedings of the Seventh National Conference on Artificial Intelligence, Minneapolis, MN. August 1988, Morgan Kaufmann, San Mateo, CA. pp. 111-116

[Korf, 1985] Richard E. Korf, *Real-Time Heuristic Search*, Artificial Intelligence vol. 42, 1985, pp. 189-211.

[Mouaddib and Zilberstein, 1995] Abdel-Illah Mouaddib, Shlomo Zilberstein, *Knowledge-Based Anytime Computation*, in Proceedings of the 14th IJCAI, Montreal, Canada, 1995, pp 775-781.

[Zilberstein, 1993] Shlomo Zilberstein, *Operational Rationality through Compilation of Anytime Algorithms*, Ph.D. dissertation, Computer Science Division, University of California at Berkeley, 1993

[Zilberstein and Russel, 1993] Shlomo Zilberstein, Stuart J. Russel, *Anytime Sensing, Planning and Action: A Practical Model for Robot Control*, in Proceedings of the 13th IJCAI, Chambery, France, 1993.

[Zilberstein and Russel, 1996] Shlomo Zilberstein, Stuart J Russel, *Optimal composition of real-time systems*, Artificial Intelligence vol 82, 1996, pp. 181-213.