Ordonnancement en temps-réel des activités des radars

Cyril Duron

année : 2003

N° attribué par la bibliothèque

THÈSE

présentée à

L'UNIVERSITÉ DE METZ - UFR MIM
École Doctorale IAE+M

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

Automatique

soutenue par

Cyril DURON

le 20 décembre 2002

Titre

Ordonnancement en Temps-Réel des Activités des Radars

Directeur de thèse : Jean-Marie Proth

Jury

Président	Bernard Mutel
Rapporteurs	Alexandre Dolgui
	Jean-Pierre Elloy
	Jean-Claude Hennet
	Rakesh Nagi
ateurs	Jean-Marie Proth
	Ibrahima Sakho

année : 2003

N° attribué par la bibliothèque

THÈSE

présentée à

L'UNIVERSITÉ DE METZ - UFR MIM
École Doctorale IAE+M

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

Automatique

soutenue par

Cyril DURON

le 20 décembre 2002

Titre

Ordonnancement en Temps-Réel des Activités des Radars

Directeur de thèse : Jean-Marie Proth

Jury

Président	Bernard Mutel
Rapporteurs	Alexandre Dolgui
	Jean-Pierre Elloy
	Jean-Claude Hennet
	Rakesh Nagi
Examinateurs	Jean-Marie Proth
	Ibrahima Sakho

Résumé

Directeur de Thèse : Pr. Jean-Marie PROTH

L'objectif général de cette thèse, suggéré par le contrôle des radars de combat, consiste à intercaler en temps réel une tâche aléatoire dans un ordonnancement existant tout en limitant autant que possible l'augmentation de la valeur du critère. Dans notre cas, le critère que nous considérons est la somme des dépassements des délais des tâches déjà ordonnancées. Ces délais sont supposés quelconques : cette contrainte est plus dure que dans le cas des radars de combat où un certain nombre de tâches de surveillance doivent être effectuées de manière répétitive au cours d'une période donnée à l'intérieur de laquelle leur ordonnancement est libre, ce qui équivaut à un délai unique pour l'ensemble des tâches. La tâche à intercaler apparaît à un instant quelconque (c'est l'instant que nous considérons comme l'instant zéro). Sa durée n'est connue qu'au moment de son apparition. Il en est de même de son délai, qui est impératif. Nous considérons d'abord le cas d'une tâche aléatoire unique, puis le cas d'une tâche aléatoire composée de deux sous-tâches séparées par une période donnée. Enfin, nous proposons une amélioration de l'approche actuellement utilisée dans ce domaine.

Le contenu de cette thèse se limite aux aspects publiables.

Abstract

Ph. D. Supervisor : Pr. Jean-Marie PROTH

The goal of this thesis, inspired by the battle radar management, is to insert in real-time a random task in the current schedule while minimizing the criterion. In our case, the criterion is the minimization of the sum of the delays of the already scheduled tasks. There is no leading rule for these delays. It is a stronger constraint than in the battle radars case, because they have to perform an amount of repetitive tasks in a given period. This may be considered as a unique due-date for all the tasks. The task we have to insert may appear at any time (For the sake of simplicity, the time at which a random task appears is time zero). Its processing time and its delay, which cannot be violated, are known only at time zero. We firstly take into consideration the case of a single random task, then we consider the case of a task made of two subtasks separated by a given period. Finally, we propose an improvement of the approach currently used to manage battle radars.

Remerciements

Cette thèse a été élaborée au sein du projet Sagep de l'Inria-Lorraine. Cette équipe est animée par Monsieur Jean-Marie Proth, directeur de recherches. Je tiens à lui exprimer toute ma reconnaissance non seulement pour avoir rendu cette thèse possible, mais aussi pour sa disponibilité, ses nombreuses relectures du manuscrit, ses conseils et son soutien tout au long de ces trois années de préparation de thèse.

Je tiens à remercier les membres de l'équipe Sagep : Monsieur Satyaveer Singh Chauhan et sa femme Madame Anjali Awasthi pour leur amitié et leur soutien.

Je désire témoigner ma reconnaissance et mes remerciements à :

- la Direction Générale de l'Armement qui a supporté financièrement ce travail, et en particulier à Monsieur Michel Granger qui a suivi ce travail,
- Monsieur Alexandre Dolgui, Professeur à l'Université de Technologie de Troyes,
- Monsieur Jean-Pierre Elloy, Professeur à l'Ecole Centrale de Nantes,
- Monsieur Jean-Claude Hennet, Directeur de Recherches CNRS, LAAS de Toulouse,
- Monsieur Bernard Mutel, Professeur à l'ENSAIS à Strasbourg,
- Monsieur Rakesh Nagi, Professeur à l'Université de Buffalo,
- Monsieur Ibrahima Sakho, Professeur à l'Université de Metz.

à mes parents,

et à tous ceux que je ne nomme pas, mais qui se reconnaitront

Table des matières

Liste des tableaux

Table des figures

Computers do not dream,any more than they play. We are far from certain what dreams are good for but we know what they indicate : a great deal of information processing goes on far beneath the surface of man's purposive behavior, in ways and for reasons that are only very indirectly reflected in his overt activity.

—Alan M. Turing

There are reports that many executives make their decisions by flipping a coin or by throwing darts,etc...It is also rumored that some college professors prepare their grades on such a basis. Sometimes it is important to make a completely 'unbiaised' decision; this ability is occasionally useful in computer algorithms, for example in situations where a decision made each time would cause the algorithm to run more slowly.

—Donald E. Knuth

Chapitre 1

Introduction

1.1 Généralités

L'ordonnancement vise à définir les dates d'exécution d'un ensemble d'opérations en tenant compte de la disponibilité des ressources (matérielles, humaines, etc) nécessaires, de manière à optimiser un critère donné. Un des critères d'optimisation les plus utilisés pour un ordonnancement consiste à minimiser le temps d'exécution des tâches, chaque tâche étant composée d'un ensemble d'opérations : on parle alors de minimisation du *makespan*. D'autres critères existent comme la minimisation de la somme des retards (lorsque des délais interviennent), la minimisation des en-cours, la minimisation du coût de production, ou encore une répartition équilibrée des charges des ressources dans le cas de ressources fonctionnant en parallèle. Le champ d'applications de l'ordonnancement est large : la gestion de la production dans l'industrie, la gestion de projet, l'organisation des emplois du temps, la gestion de la charge de processeur en informatique, etc. Ainsi, les ouvrages et articles consacrés à l'ordonnancement (scheduling en anglais) sont nombreux comme le prouvent ces quelques références : Conway et *al.* [1], Baker [2], Coffman [3], Rinnooy Kan [4], Dempster et *al.* [5], French [6], Carlier et Chrétienne [7], Blazewicz et *al.* [8], Morton et Pentico [9], Feldmann et *al.* [10], Tanaev et *al.* [11], Tanaev et *al.* [12], Zweben et Fox [13], Brucker [14], Chrétienne et *al.* [15], Pinedo [16], Chu et Proth [17], Koole [18], Lai et *al.* [19], Shmelev [20], et Esquirol et Lopez [21].

Les problèmes d'ordonnancement peuvent être partitionnés en deux classes distinctes : ceux qui obéissent à la contrainte "temps-réel" et les autres. Cette

thèse s'intéresse à un problème de type "temps-réel" qui a la particularité de ne concerner qu'une seule machine (un radar, en l'occurrence) et a pour objectif d'insérer une tâche qui apparait de manière aléatoire dans un ordonnancement existant en le perturbant aussi peu que possible.

Dans la pratique, cette problématique appartient au monde des radars de combat modernes dont l'objectif est d'exécuter des tâches de surveillance répétitives tout en réagissant à l'apparition aléatoire d'objets volants tels que avions ou missiles.

Avant d'aborder ce problème spécifique, nous allons rappeler les principaux travaux effectués sur les problèmes à une machine puis sur les problèmes "temps-réel".

1.2 Les problèmes à une machine

Dans cette section, nous allons énumérer différentes méthodes permettant l'insertion d'une tâche dans un ordonnancement sur une ressource unique. Les méthodes les plus classiques seront simplement citées au travers de références, et d'autres, moins répandues, feront l'objet d'une étude plus approfondie au moyen de commentaires d'articles.

De nombreux travaux de recherche ont déjà été consacrés à la résolution de ce problème : Albers et Brucker [22], Beauquier *et al.* [23], Ben-David *et al.* [24], Brucker et Knust [25], Chauvet et Proth [26], Chauvet *et al* [27], Gondran et Minoux [28], Nowicki [29], Panwalkar et Rajagopalan [30].

De nombreuses heuristiques générales permettent de résoudre ce type de problème, comme le recuit simulé [31], les méthodes tabous [32, 33], les algorithmes génétiques [31, 34], une méthode basée sur le comportement des colonies de fourmis [35], pour ne citer que quelques unes d'entre elles.

D'autres algorithmes plus spécifiques ont été proposés pour l'ordonnancement d'un ensemble de tâches. L'idée maîtresse de quelques-uns de ces algorithmes est donnée ci-dessous.

- EDF (Earliest Deadline First) : cet algorithme choisit parmi les tâches exécutables celle dont le délai sera échut le plus tôt. Si aucune tâche n'est disponible, alors un temps libre est généré. (voir [36, 37])
- LLF ou LL (Last Laxity First) : cet algorithme choisit parmi les tâches

exécutables celle dont la distance entre la fin de son exécution et son délai est le plus court. Si aucune tâche n'est disponible, alors un temps libre est généré. (voir [38])

- LRF (Last Release Time First) : cet algorithme choisit parmi les tâches exécutables celle dont le temps écoulé depuis son apparition est le plus grand. Si aucune tâche n'est disponible, alors un temps libre est généré. (voir [39])
- HPF (High Priority First) : cet algorithme choisit parmi les tâches exécutables celle dont la priorité est la plus grande. Si aucune tâche n'est disponible, alors un temps libre est généré.

Il a été démontré que l'algorithme EDF (Earliest Deadline First) [36, 37], qui ordonnance les tâches dans l'ordre croissant de leur délai d'exécution, et l'algorithme LL (Last Laxity) [38, 40], sont optimaux dans le cas de tâches indépendantes et préemptives exécutées sur une seule ressource. Malheureusement, si la préemption est possible lorsque la ressource est un processeur, elle est impossible lorsqu'il s'agit d'un radar, ou d'un système de production.

Dans leur article *Single Machine Scheduling Subject to Precedence Delays*, (voir [41]) Finta et Liu proposent une méthode (désignée par *Lexicographic Order Schedule*) pour trouver un ordonnancement qui minimise le makespan dans le cas d'un ensemble de n tâches ayant la caractéristique suivante : une tâche ne peut commencer son exécution avant que ses prédécesseurs aient terminé leur propre exécution et qu'une période supplémentaire donnée se soit écoulée. Dans le cas où les temps d'exécution des tâches sont unitaires, et les durées des délais supplémentaires sont entières le problème est NP-Difficile dans le sens fort. Par contre, dans le cas où les temps d'exécution des tâches sont entiers, et les durées des délais supplémentaires sont unitaires, le problème est polynomial, et Finta et Liu proposent un algorithme optimal de complexité $O(n^2)$. Pour modéliser, ils emploient un graphe orienté acyclique $G = (V, E)$ où l'ensemble des noeuds V correspond à l'ensemble des tâches, et l'ensemble des arcs E correspond aux délais de précédence. Pour chaque paire de tâches $i, j \in V$, si $(i, j) \in E$, la tâche j ne peut commencer que l_{ij} unités de temps après que la tâche i ait fini son exécution. L'algorithme est le suivant : on

numérote arbitrairement les k noeuds terminaux du graphe, de 1 à k. Puis, on considère tous les noeuds non numérotés dont tous les successeurs ont déjà été numérotés. On leur donne arbitraitrement un numéro unique supérieur à k. On itère ensuite de la même façon, jusqu'à ce qu'il ne reste plus de noeud non numéroté. L'ordonnancement est alors obtenu en exécutant dans l'ordre décroissant des numéros des sommets du graphe, les tâches associées.

Une approche utilisant la méta-heuristique "colonie de fourmis" (Ant Colony Optimization ou ACO) est développée dans le papier *Minimizing Total Tardiness on a Single Machine Using Ant Colony Optimization* écrit par A.Bauer, B. Bullnheimer, R.F. Hartl et C. Strauss. (voir [35]). Cette heuristique est basée sur l'observation du comportement naturel des fourmis. Lorsqu'une fourmi, partie à la recherche de nourriture revient vers la fourmillière, elle laisse des phéromones. Ces phéromones (qui sont des marques odorantes) codent des informations comme la distance à parcourir, la direction à suivre et la qualité de la nourriture et se dissipent avec le temps. Les auteurs ont fait un parallèle entre la recherche du meilleur chemin pour trouver de la nourriture, et la recherche d'un ordonnancement de tâches non préemptives (modélisé comme un chemin dans un graphe orienté acyclique) minimisant la somme des délais dans le cas d'une machine unique. Le modèle de graphe proposé pour modéliser un ordonnancement de n tâches est un graphe top-bottom. Chaque noeud i ($1 \leq i \leq n+1$) indique quel est le nombre de tâches déjà exécutées. Par conséquent, le noeud i indique qu'il y a eu $i-1$ tâches exécutées. Chaque arc est associé à un nom de tâche k ($1 \leq k \leq n$), et un poids p (modélise les phéromones). Entre chaque noeud i et $i+1$, il existe n arcs distincts représentant chaque tâche qu'il est possible d'exécuter. Le graphe est initialisé avec, pour chaque arc, une valeur de phéromone $p = 0$. La méta-heuristique ACO utilise un algorithme itératif. Chaque itération correspond à une population de fourmis. Chacune de ces fourmis construit tâche après tâche (nous verrons comment ci-dessous) sa propre solution. A la fin de l'itération, les fourmis ayant trouvé les meilleurs chemins (au sens du critère) marquent les arcs correspondant au chemin qu'elles ont parcouru en augmentant la valeur de p associée. A la fin de chaque itération, on remet à jour la valeur des phéromones (dissi-

pation) pour éviter de bloquer l'algorithme dans un minimum local. Chaque fourmi génère son chemin (indépendamment des fourmis de son itération) en choisissant, tâche après tâche, quel est le meilleur ordonnancement au sens d'une probabilité de transition P_{ij} (j est le nom de la tâche considérée, et i le numéro de la tâche que nous considérons dans l'ordonnancement, voir [35]). P_{ij} est constitué de deux indicateurs pondérés :

- $\eta_{ij} \in]0,1]$. Plus sa valeur tend vers 1, plus la solution j semble bonne *a priori.*
- $\tau_{ij} \in]0,1]$ calculé en fonction des phéromones. Plus sa valeur tend vers 1, plus la solution j semble bonne *a posteriori.*

La conclusion des auteurs est que cette méta-heuristique est plus performante que les autres heuristiques (branch and bounds, recuit simulé, ...) en utilisant une heuristique MDD (Modified Due Date, voir [35]) pour η_{ij} et une heuristique prenant en compte les meilleures solutions calculées à partir du critère EDF (voir [34]) pour τ_{ij}. En sus, avant chaque mise à jour des phéromones, les auteurs recommandent d'appliquer une stratégie de recherche locale basé sur l'échange de deux tâches dans l'ensemble des tâches déjà ordonnancées.

Dans leur article *Genetic and Local Search Algorithms as Robust and Simple Optimization Tools*, M. Yagiura et T. Ibaraki (voir [34]) décrivent plusieurs approches méta-heuristiques pour résoudre des problèmes NP-difficiles et en particulier les ordonnancements. Plus précisément, ils comparent ces méta-heuristiques en fonction de leur capacité à résoudre le problème de l'ordonnancement de n tâches non préemptives sur une machine unique. Tout d'abord, ils considèrent la méta-heuristique LS (Local Search - Recherche Locale). Ils l'initialisent en proposant une solution S de départ via une heuristique adaptée au problème à résoudre. Ensuite, ils cherchent dans le voisinage de S s'il existe une solution S' meilleure que S (au sens du critère que l'on cherche à optimiser). Si ce n'est pas le cas, alors S est au moins un extremum local, et la recherche se termine. Si oui, on pose $S = S'$, et on recommence le processus.

Il existe aussi une méthode directement dérivée de la LS : c'est la MLS (Multi Local Search -Recherche locale multiple-). Cette méta-heuristique consiste à mener une recherche LS pour plusieurs points de départ (générés aléatoire-

ment, et non à l'aide d'une une heuristique), et à ne garder que le meilleur des résultats obtenus.

La méta-heuristique GRASP (recherche gloutonne), est une variante du MLS. La différence réside dans le fait que les solutions initiales sont générées par des heuristiques. Les AE (algorithmes évolutionnistes) peuvent être vus comme une généralisation du LS. L'algorithme est initialisé en construisant au hasard un ensemble de solutions P. Ils cherchent alors à atteindre un objectif, appelé critère d'arrêt. Un ensemble Q de solutions voisines de P est généré. Pour cela, ils utilisent deux méthodes :

- par *mutation* qui consiste à générer de nouvelles solutions en faisant subir une permutation à un élément de P
- par *cross-over* qui consiste à combiner deux éléments de P, pour ne produire qu'un élément à priori nouveau.

Ils mélangent alors les deux ensembles P et Q, et en extraient $|P|$ éléments au hasard ou à l'aide d'une heuristique. Cette sélection est le nouvel ensemble P. Ils réitèrent alors le processus jusqu'à ce que la condition d'arrêt soit atteinte.

Une amélioration de cet algorithme est le GLS, qui est une combinaison de GA et de LS. La différence réside dans le fait que chaque solution de Q est améliorée à l'aide d'une technique de recherche locale. La conclusion du papier est que si l'on recherche un algorithme simple pour la résolution de notre problème, il est préférable d'utiliser MLS. Par contre, si l'on désire une qualité plus élevée des solutions obtenues, il est préférable d'utiliser GLS.

L'article "*Evaluation of greedy, myopic and less-greedy heuristics for the single machine, total tardiness problem*", écrit par R.M. Russel et J.E. Holsenback (voir [42]), décrit l'heuristique NBR (Net Benefit of Relocation). Cette heuristique est utilisée pour résoudre le problème d'ordonnancement de n tâches de durée entière, non préemptives, sur une machine unique. Le critère que les auteurs cherchent à minimiser est la somme des retards. Les auteurs décrivent l'algorithme original, puis introduisent deux propriétés du problème, ce qui leur permet de simplifier leur algorithme. L'heuristique initiale est la suivante : on classe les tâches par ordre croissant des délais. Les tâches sont ensuite notées $1, 2, \ldots, N$ où N est le nombre total de tâches). On marque

le N. On parcourt ensuite successivement toutes les tâches de $N-1$ à 1 et on marque les tâches ayant une durée d'exécution supérieure à la dernière tâche marquée. On évalue alors le critère NBR pour les tâches j marquées : $NBR_j = max(d_j - C_j, 0) + \sum_{i=j-1}^{k} (min(p_j - max(0, \sum_{l=1}^{j} p_l - d_j)) - p_i)$ où p_j est la durée de la tâche j, d_j le délai de la tâche j et C_j l'instant de fin de la tâche j. On permute la tâche N avec celle ayant le plus grand NBR_j. La première propriété indique que si une tâche avec un retard réductible est précédée par une tâche de durée supérieure, l'ordonnancement n'est pas optimal. La seconde propriété rappelle que si on place une tâche j immédiatement après une tâche k, et que si, après placement, le retard de la tâche k est strictement plus grand que la durée de cette même tâche, alors j doit posséder un retard réductible. Les auteurs en déduisent les modifications suivantes de l'algorithme NBR : la première propriété est utilisée pour réduire l'espace des solutions (tâches marquées) et la seconde propriété dit que si après une relocalisation à la place $k+1$, la tâche k continue à posséder un retard réductible, alors, les tâches candidates à la relocalisation en k sont les tâches avec une durée supérieure à celle de la tâche k et inférieure à celle de la tâche $k+1$.

On notera l'algorithme NBR modifié, M-NBR. Les auteurs ont implémenté un logiciel pour comparer les performances des algorithmes NBR, M-NBR et PSK.

L'algorithme PSK a été développé par Panwalkar *et al* (voir [43]). Cet algorithme, comme NBR, fonctionne en une seule passe et possède une complexité de calcul assez faible. Il séquence les tâches à partir de la première position, en utilisant le critère EDF (Earliest Deadline First). L'ordonnancement s'arrête lorsque le premier retard se présente. De cette façon, une tâche (si son exécution peut être complétée à temps) avec un temps d'exécution élevé et un délai court peut se trouver exécutée en début d'ordonnancement même si dans le cas optimal elle devait se placer en dernier.

Les temps de calcul ne sont pas pris en compte car d'une part ils sont négligeables, et d'autre part il ne peuvent être mesurés de façon fiable. Les jeux de tests ont été créés à partir des normes introduites par Potts et Van Wassenhove (voir [44] et le paragraphe suivant).

En conclusion, pour les problèmes de moins de 50 tâches, NBR est plus

puissant que PSK, mais, au delà de 100 tâches, NBR est est un algorithme trop glouton pour obtenir des résultats intéressants. Cependant, M-NBR permet de passer ce cap. Et finalement, la complémentarité de PSK et de M-NBR permet, en les associant, de trouver des résultats quasi-optimaux pour des problèmes plus larges encore.

Dans leur article "*A decomposition Algorithm for the single machine total tardiness problem*", C.N. Potts et L.N. Van Wassenhove (voir [44]) proposent un algorithme permettant de traiter le problème de l'ordonnancement de n ($n > 50$) tâches non préemptives sur une machine unique. Cet algorithme s'appuie sur deux théorèmes permettant de partitionner le problème en sous-problèmes de taille suffisamment petite pour pouvoir être résolus par des méthodes déjà connues. Ici, les sous-problèmes sont résolus en utilisant l'algorithme de programmation dynamique de Schrage et Baker (voir [45]).

Pour effectuer la décomposition, les auteurs supposent que les tâches sont classées par ordre croissant de leur délai, et en cas d'égalité des délais, par ordre croissant de leur durée. Les deux théorèmes sur lesquels est fondée l'approche par partitionnement sont les suivants : le premier est de Lawler (1977) et dit que si la tâche j (j aussi grand que possible) possède la durée la plus grande dans un ensemble des n tâches, alors le problème se décompose et la tâche j peut être permutée avec la tâche k avec $k \in [j, \ldots, n]$. Le second théorême dit que si la tâche j (j aussi grand que possible) possède la durée la plus grande d'un ensemble de n tâches, et si la somme des durées des tâches 1 à k pour $k \in [j, n-1]$ est plus grande que le délai de la tâche $k+1$, ou si la somme des durées des tâches 1 à $k-1$ pour $k \in [j+1, n]$ est strictement inférieure au délai de la tâche k, alors le problème ne se décompose pas et la tâche j ne peut pas être permutée avec la tâche k avec $k \in [j, \ldots, n]$.

Les auteurs ont testé cette heuristique sur des problèmes comprenant de 60 à 100 tâches. Chaque tâche a une durée entière, générée suivant une distribution uniforme entre 0 et 100. La difficulté du problème est quantifiée par deux paramètres : tout d'abord le RDD (Relative Range of Dates) et le ATF (Average Tardiness Factor). Les auteurs calculent le délai de chaque tâche en fonction de ces paramètres.

L'algorithme proposé est efficace jusqu'à environ 100 tâches et est supérieur aux autres algorithmes proposés dans ce domaine jusqu'en 1982. Mais cet algorithme n'est pas généralisable et ne permet pas de prendre en compte des retards pondérés.

1.3 Les problèmes à une machine en temps-réel

Dans une application en temps réel, on dit aussi en ligne (et, en anglais, real-time ou on-line), il est obligatoire de fournir une solution rapidement, sans pouvoir attendre les informations sur les demandes à long terme. Les premiers problèmes importants traités en temps réel ont concerné le pilotage de systèmes complexes (avion, centrale nucléaire, etc). De tels problèmes se sont également posés pour l'ordonnancement de radars (Orman et *al.*, [46, 47, 48]), ou de processeurs en informatique (Chrétienne et *al.*, [15]). L'ordonnancement en temps réel permet aussi de fournir un délai de fabrication précis au client, à l'instant où il passe sa commande. Sgall [49] décrit la littérature déjà abondante relative à l'ordonnancement "en temps réel". Aujourd'hui, les applications en temps réel de l'ordonnancement se développent parce qu'elles permettent de réagir dynamiquement aux aléas de l'environnement. Les applications en temps réel impliquent de développer des techniques rapides dans tous les cas de figures. La rapidité d'obtention souhaitée des résultats dépend du problème traité : pour une entreprise, il peut s'agir de quelques minutes, pour un radar, de quelques millisecondes. Le terme temps-réel signifiant que l'application considérée est capable de fournir une solution à un problème avant que d'autres évènements ne viennent modifier l'état du système. La rapidité d'un algorithme en temps réel est mesurée par sa "complexité", c'est-à-dire l'évaluation du nombre d'opérations élémentaires effectuées par l'algorithme dans le pire des cas. Elle est donnée en fonction de la taille des problèmes traités (Knuth,[50], Gaudel et *al*,[51]). Pour garantir que le temps d'exécution d'un algorithme ne dégénère pas dans des cas d'application en temps réel, il est souhaitable que sa complexité soit polynomiale (i.e. que le temps de calcul s'exprime comme un polynôme fonction de la taille des données). A l'opposé, les problèmes NP-difficiles au sens fort forment une classe pour laquelle on ne connaît pas d'algorithme polynomial (voir Garey et Johnson,[52]). Pour résoudre de tels problèmes de

manière optimale, les seules méthodes connues consistent à explorer explicitement ou implicitement toutes les solutions, ce qui n'est possible que pour des exemples de taille très limitée. Ainsi, dans le cas d'applications réelles et quand la recherche de la solution optimale devient "combinatoire", on utilise des "heuristiques" rapides qui fournissent, au mieux, une solution proche de la solution optimale.

Dans leur article *On-line Algorithms for a Single Machine Scheduling Problem*, Mao W., Kincaid R.K., et Rifkin A. (voir [53]) présentent deux approches temps-réel pour résoudre le problème de l'ordonnancement de tâches non préemptives sur une machine à ressource unique. Par temps-réel, ils supposent que toutes les tâches qui vont être exécutées lors de l'utilisation de la machine ne sont pas connues à l'instant de départ $t = 0$. Autrement dit, des tâches apparaissent de manière asynchrones pour $t > 0$ et doivent être prises en compte dans l'ordonnancement déjà établi.

Le premier algorithme proposé est appelé $FCFS$ (pour First Came, First Served : premier arrivé, premier servi). Il consiste à exécuter les tâches dans l'ordre croissant de leur date de début (si plusieurs tâches ont des dates de début identiques, elles sont classées dans l'ordre croissant de leur durée d'exécution.). Chaque nouvelle tâche arrivant en temps réel est classée dans cette liste. Lorsque la machine devient disponible, elle exécute la première tâche de la liste d'attente. Chaque tâche exécutée est retirée de la liste.

Le deuxième algorithme proposé est appelé $SAJF$ (Shortest available job first : exécution prioritaire de la tâche ayant la durée la plus brève). Cet algorithme utilise aussi une file d'attente, et le même mécanisme d'ajout et de retrait de tâches de la liste que $FCFS$.

La différence se situe au niveau de la méthode de classement des tâches dans la liste d'attente : $SAJF$ classe les tâches dans l'ordre croissant de leur durée d'exécution. La comparaison entre les algorithmes temps-réel $FCFS$ et $SAJF$ est effectuée suivant le critère de la c-compétitivité. Les auteurs considèrent comme c-compétitif un algorithme temps-réel capable d'effectuer le même travail qu'un algorithme optimal hors-ligne en moins de c fois le temps que met cet algorithme optimal. Ici, cependant, les auteurs résolvent un problème NP-difficile, et à la place de l'algorithme optimal, ils utilisent une heuristique pour

les problèmes comprenant plus de 50 tâches, car alors, le temps de calcul requis rend le problème insoluble. Le choix a été fait d'utiliser l'heuristique de la recherche tabou (voir [32, 33]) qui donne d'excellents résultats. Les auteurs ont prouvé que les deux algorithmes $FCFS$ et $SAJF$ sont n-compétitifs (n est le nombre total de tâches ordonnancées), et que $SAJF$ est meilleur que $FCFS$ pour minimiser le temps d'exécution sur la machine.

1.4 Ordonnancement à flot continu

1.4.1 Les radars multifonctions

Les radars de combat effectuent plusieurs types d'activités. L'activité la plus connue de ces appareils est certainement celle consistant à détecter des objets dans le ciel. Mais les radars ont beaucoup d'autres applications : ils peuvent par exemple exploiter l'effet Doppler, afin de connaître la vitesse et la direction de déplacement d'une cible. Ils peuvent également former une image d'un objet se déplaçant dans le ciel et en déduire (à l'aide de bases de données) quelles en sont les caractéristiques techniques. Un radar peut aussi permettre de guider un missile vers une cible hostile.

Pour les systèmes de radars classiques, la concentration de plusieurs de ces compétences n'est pas possible, et nécessite la mise en place de réseaux constitué de radars différenciés : chaque activité (surveillance, pistage, etc,...) dispose de son antenne de radar dédiée. Le grand avantage de ce système est la relative simplicité à ordonnancer les tâches sur les différents radars. Par contre la synchronisation de l'ensemble est difficile, non seulement au niveau de l'échange des données, mais aussi au niveau des fréquences d'émission des différentes antennes, nécessaire pour éviter des interférences génératrices d'erreurs d'interprétation.

Une des caractéristiques des radars de combat modernes est de pouvoir supporter plusieurs types d'activité simultanément, et ce grâce aux progrès de la micro-informatique et de la technique des composants radio. Plus particulièrement, nous aller nous intéresser à un type de radar de détection aérienne appelé *radar multifonctions*. Ce radar est capable de procéder régulièrement à une inspection complète de son domaine de visibilité (envoi d'ondes vers diffé-

rentes parties du ciel), de suivre les cibles qui ont déjà été identifiées (calcul des trajectoires, doppler, etc) et, le cas échéant, de guider un missile vers une cible hostile. Ce radar se présente sous la forme d'une antenne rotative unique. Le fait que toutes les activités soient concentrées sur cet appareil unique permet d'éliminer le problème de synchronisation des radars classiques (bien entendu, ça ne supprime pas le problème des brouillages ennemis). Par contre, la sollicitation que subit cet appareil afin de remplir toutes ses missions rend plus complexe l'ordonnancement d'une tâche aléatoire lorsqu'elle surgit. Les conditions d'apparition d'une tâche aléatoire seront décrites ultérieurement.

Fondamentalement, un radar est capable d'effectuer deux types de tâches : l'émission et la réception d'ondes. Ces tâches sont non préemptives : lorsqu'elles sont initiées, elles doivent être complétées. Le radar n'est pas continuellement en train de recevoir ou d'émettre des ondes : il existe aussi des plages de temps disponibles, où le radar est en attente. L'origine de ces temps disponibles est triple :

- Les contraintes temporelles liant les tâches déjà ordonnancées entre elles ne permettent pas d'utiliser ce temps disponible.
- Le système d'émission a été trop sollicité et doit rester inactif temporairement (cette problématique de temps maximal d'émission est abordé dans le chapitre 7 de cette thèse).
- Il n'y a pas de tâches dont les créneaux temporaires soient compatibles. Par exemple, la position actuelle de l'antenne du radar ne permet pas d'accéder à la zone du ciel qui doit être vérifiée.

Toutes les activités exécutées par un radar ont un schéma de base semblable : tout d'abord une émission d'ondes, puis un temps mort (temps nécessaire à l'onde pour parcourir la distance entre le radar et la cible supposée, et retour), et enfin une réception éventuelle de l'onde. Suivant le type de tâche exécutée (surveillance, pistage, etc), et suivant l'environnement de travail (mer, orage, etc), ce schéma de base peut être répété jusqu'à plusieurs dizaines de fois, afin de diminuer les risques d'erreurs dues au brouillage naturel.

Revenons plus précisément sur les activités du radar. Elles sont de deux types : les tâches périodiques (surveillance du ciel, suivi de cible, etc) et les tâches apériodiques (confirmation d'un écho, perte d'une cible, etc). Les tâches

apériodiques ont un lien de causalité avec le résultat d'une tâche périodique. En d'autres termes, si une tâche de surveillance du ciel est positive (une émission a été réfléchie sur un objet et a été renvoyée vers le radar) alors le radar doit confirmer cet écho dans un délai donné (fenêtre de visibilité). Si cette tâche de confirmation est elle aussi positive, le radar évalue (par application du principe de l'effet Doppler) la direction et la vitesse de la nouvelle cible. Le suivi de cette cible est une nouvelle tâche à ajouter à la liste des tâches périodiques.

Actuellement, l'ordonnancement des radars ne tire pas avantage des temps morts entre émission et réception. Pour une simplification des calculs d'ordonnancement d'une tâche aléatoire, le schéma de base est considéré comme étant une seule et unique tâche non préemptive. Nous examinons une amélioration de cette approche dans la première partie de cette thèse. Dans la seconde partie, nous considèrons toujours que les tâches de base du radar (émission et réception d'ondes) sont non préemptives, mais par contre nous essayons d'utiliser les temps morts afin d'ordonnancer d'autres tâches. De plus, nous exploitons la conséquence directe de cette approche : au lieu d'exécuter les activités les unes à la suite des autres, nous entrelaçons (par exemple nous exécutons plusieurs tâches d'émission avant d'exécuter une tâche de réception). Enfin, dans la troisième partie de, nous proposons une amélioration de l'approche actuellement utilisée dans les radars de combat.

A notre connaissance, le premier auteur à avoir publié un article sur les radars multifonctions est Billeter [54]. Plus récemment, Orman ([46, 47, 48]) dans deux articles et dans sa thèse, a décrit le fonctionnement technique des radars multifonctions et essayé de proposer un modèle d'ordonnancement temps-réel des tâches du radar. Plusieurs règles d'ordonnancement y sont testées, et un logiciel de simulation y est proposé. Cependant, ni ces articles, ni la thèse ne fournissent d'information nous permettant d'effectuer une comparaison avec l'approche développée dans les pages qui suivent. En particulier, les données ne sont fournies que partiellement.

1.4.2 Problématique

Dans ce chapitre, nous avons présenté un état de l'art des problèmes d'ordonnancement en général, et plus précisément des problèmes à une machine,

à tâches non préemptives et en *temps réel.*

Cette thèse se divise en trois parties. La première partie traite du problème de l'insertion d'une tâche unique sur une ressource unique. Nous faisons l'hypothèse que nous disposons d'un ordonnancement, et que l'on cherche à y insérer une nouvelle tâche caractérisée par un instant minimal de départ, un temps d'exécution et un délai. Il nous est possible de retarder l'exécution de tâches, mais pas de changer leur ordre d'exécution. Le critère que nous cherchons à minimiser est l'augmentation de la somme des retards. Nous définissons dans les chapitres 2 et 3 trois algorithmes différents pour arriver à ce résultat.

Dans la deuxième partie, nous considérons l'insertion non plus d'une tâche simple, mais d'une tâche composée de deux sous-tâches séparées par une période fixée, que nous appellerons bitâche. Cette bitâche doit être exécutée avant un délai, qui est impératif. Cette période peut être utilisée pour l'exécution d'autres tâches. Il nous est possible de retarder l'exécution de tâches (voire de bitâches), mais pas de changer leur ordre d'exécution. Le chapitre 4 présente le problème et étudie quelles sont les variations du critère suivant les différents paramètres de la tâche aléatoire. Le chapitre 5 est dédié à la description et l'évaluation de trois algorithmes pour l'insertion d'une bitâche dans un ordonnancement de tâches simples. Finalement, le chapitre 6 propose une méthode permettant d'insérer une bitâche aléatoire dans un ordonnancement de bitâches.

La dernière partie décrit techniquement le fonctionnement d'un radar de combat, ainsi que deux algorithmes simulant le fonctionnement d'un modèle précis de radar (le radar multifonctions). Un radar de combat exécute des bitâches. Les algorithmes proposés permettent de gérer l'apparition de tâches aléatoires en les insérant dans un ordonnancement préexistant sans remettre en question ni les dates ni l'ordre d'exécution des tâches déjà ordonnancées. Le premier algorithme ne permet pas d'utiliser le temps libre des tâches (cas non entrelacé), alors que c'est le cas pour le second (cas entrelacé). Les résultats expérimentaux montrent que la méthode permettant l'entrelacement permet d'obtenir des résultats de bien meilleure qualité. Les algorithmes utilisés pour résoudre ce problème de taille réelle sont des heuristiques. Les programmes ont été écrit en Java.

Il y a trois annexes. Le premier annexe est un article en anglais (voir [55]) consacré à l'ordonnancement d'une polytâche dans un ordonnancement. Une polytâche est une tâche constituée d'un nombre donné de sous-tâches séparées deux à deux par un temps libre. La durée de chaque sous-tâche, ainsi que celle de chaque temps libre est arbitraire. Deux heuristiques sont proposées pour résoudre ce problème. Le temps réel n'y est pas abordé.

Le second annexe contient la description détaillée d'un algorithme d'insertion optimale d'une bitâche dans un ordonnancement de bitâches. Cet algorithme n'est pas exécutable en temps réel, et est utilisé comme étalon. Il est décrit dans le chapitre 5.

Le troisième annexe contient le listing et des captures d'écrans de l'algorithme de simulation de radars multifonctions décrit dans le chapitre 7.

L'étude proposée dans cette thèse déjà fait l'objet de communications dans les articles suivants :

- Duron et Proth (2002) [55],
- Duron (2000) [56],
- Duron et Proth (2002) [57],
- Duron et *al* (2001) [58],
- Duron et *al* (2002) [59],
- Duron et *al* (2001) [60],
- Duron et Proth (2002) [61],
- Duron et Proth (2002) [62].

Première partie

Insertion d'une tâche simple dans un ordonnancement

Présentation de la première partie

Cette thèse concerne un type particulier de problème d'ordonnancement à une machine issu du contrôle des radars de combat. Nous cherchons à insérer une nouvelle tâche, dite tâche aléatoire, dont l'arrivée est aléatoire, et dont les caractéristiques ne sont connues qu'au moment de son apparition, dans un ordonnancement préexistant. Cette première partie s'intéresse à une tâche aléatoire caractérisée par sa durée, et par un délai d'exécution impératif. La ressource sur laquelle les tâches sont exécutées ne permet pas de suspendre temporairement une tâche en cours d'exécution afin d'en exécuter une autre (non préemptivité). Lorsque nous cherchons à inclure une nouvelle tâche dans l'ordonnancement existant, il nous est permis de retarder les tâches existantes pour intercaler la tâche aléatoire, mais pas de changer l'ordre d'exécution de ces tâches. La plus grande difficulté de ce problème consiste à effectuer ce travail d'insertion en *temps réel*. Pour atteindre ce but, nous allons considérer des heuristiques agissant en deux temps :

- Une partie en différé : le plus gros des calculs est effectué lorsque la ressource de calcul est disponible afin de minimiser le temps de calcul lorsque surgit une nouvelle tâche aléatoire. Les calculs sont effectués sur la base d'hypothèses qui compensent partiellement le manque d'information concernant la tâche aléatoire.
- Une partie en *temps réel* : on doit, avec un temps de calcul minimal, pouvoir placer la tâche aléatoire qui vient d'apparaître. Pour celà, on utilise les calculs effectués lors de la partie en différé de l'heuristique.

Le chapitre 2 aborde le problème sous l'angle des temps libres entre les tâches de l'ordonnancement initial et des délais attachés aux tâches. Deux heuristiques y sont proposées. Dans le chapitre 3 nous aurons une approche basée sur une fonction directement dérivée du critère. Ce chapitre commence par quelques

remarques sur la structure du problème, puis se poursuit par l'exposé d'un algorithme temps-réel permettant un placement optimal de la tâche aléatoire suivant le critère "minimisation de la somme des retards".

Chapitre 2

Insertion d'une tâche simple en temps réel : exploitation des temps libres entre les tâches et des délais.

2.1 Introduction.

Dans ce chapitre, nous effectuons une première approche du problème en considérant uniquement l'exploitation des données relatives aux délais et aux temps libres entre les tâches.

Nous définissons ce problème dans le paragraphe 2.2. Les formules de base sont données dans le paragraphe 2.3. La manière de les utiliser dans les algorithmes que nous proposons, et qui consiste à effectuer un maximum de calculs préparatoires en différé, est exposée dans le paragraphe 2.4. Le paragraphe 2.5 est consacré à la présentation de deux algorithmes. Le paragraphe 2.6 donne des résultats numériques et compare l'efficacité des algorithmes. La conclusion est donnée dans le paragraphe 2.7.

2.2 Position du problème.

Nous disposons d'un ordonnancement de n tâches $a_1, a_2, \ldots, a_n$ de durées respectives $\theta_1, \theta_2, \ldots, \theta_n$. Les instants de début de ces tâches sont notés $\mu_1, \mu_2, \ldots, \mu_n$. Bien entendu, $\mu_i + \theta_i \leq \mu_{i+1}$ pour $i = 1, 2, \ldots, n-1$ car la ressource exécute au plus une tâche à chaque instant. Les délais associés à ces

tâches sont respectivement $d_1, d_2, \ldots, d_n$. Dans la suite, nous noterons :

$$\Delta_i = \mu_{i+1} - (\mu_i + \theta_i) \text{ pour } i = 1, 2, \ldots, n-1 \tag{2.1}$$

Δ_i est l'intervalle de temps qui sépare la fin d'une tâche du début de la tâche suivante, et :

$$f_i = [d_i - (\mu_i + \theta_i)]^+ \text{ pour } i = 1, 2, \ldots, n-1 \tag{2.2}$$

f_i représente la flexibilité attachée à la tâche a_i, c'est-à-dire la durée dont nous pouvons retarder le début de la tâche a_i sans augmenter le critère attaché à l'ordonnancement, lequel s'écrit :

$$C_n = \sum_{i=1}^{n} (\mu_i + \theta_i - d_i)^+ \tag{2.3}$$

Dans la figure 2.1, nous avons représenté deux tâches a_i et a_{i+1} ainsi que les valeurs f_i et Δ_i asssociées à a_i.

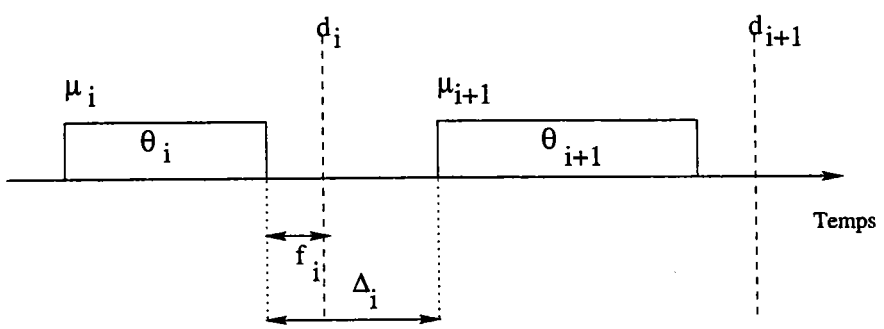


FIG. 2.1 – Relations de base

Nous voulons insérer dans l'ordonnancement initial, en *temps réel*, une tâche A apparaissant à l'instant 0. Cette tâche A est définie par :

- sa date de fin d'exécution D qui ne peut être violée,
- sa durée, notée θ.

L'objectif est de minimiser l'augmentation de la valeur du critère C_n.

2.3 Relations de base.

Supposons que la tâche aléatoire A de durée θ débute à la fin de la tâche a_k, c'est-à-dire au temps $\mu_k + \theta_k$. Alors, si $\theta \leq \Delta_k$, la tâche A peut être

insérée entre a_k et a_{k+1} sans retarder a_{k+1}. Si $\Delta_k < \theta \leq \Delta_k + \Delta_{k+1}$, alors nous devons déplacer a_{k+1} dont l'instant de début devient $\mu_{k+1} + (\theta - \Delta_k)$, mais a_{k+1} est la seule tâche qui sera déplacée. De même, si $\Delta_k + \Delta_{k+1} < \theta \leq \Delta_k + \Delta_{k+1} + \Delta_{k+2}$, à la fois a_{k+1} et a_{k+2} vont être déplacées : l'instant de début de a_{k+1} devient $\mu_{k+1} + (\theta - \Delta_k)$ alors que l'instant de début de a_{k+2} devient $\mu_{k+2} + (\theta - \Delta_k - \Delta_{k+1})$, et ainsi de suite. Pour résumer :

$$\text{Si } \sum_{s=0}^{n_k-1} \Delta_{k+s} < \theta \leq \sum_{s=0}^{n_k} \Delta_{k+s}, \text{ alors :}$$

- Les tâches a_{k+s}, $s = 1, 2, \ldots, n_k$ sont les seules tâches à être déplacées si la tâche aléatoire commence lorsque a_k finit.
- L'instant de début de a_{k+s} devient $\mu_{k+s} + (\theta - \sum_{i=0}^{s-1} \Delta_{k+i})$ pour $s = 1, 2, \ldots, n_k$.

(2.4)

De plus, si $\theta \leq \Delta_k$ aucune des tâches déjà ordonnancées n'est déplacée.

Une conséquence des relations (2.4) est que si le critère (2.3) augmente après avoir inséré la tâche A après la tâche a_k, l'augmentation est due aux n_k tâches qui sont déplacées.

L'augmentation de C_n résultant de la translation de a_{k+1} sur l'axe des temps est :

$$R_{k+1} = \begin{cases} 0 & \text{si} \quad \theta \leq \Delta_k \\ (\theta - \Delta_k - f_{k+1})^+ & \text{si} \quad \theta > \Delta_k \end{cases} \tag{2.5}$$

L'explication de (2.5) est immédiate. Si $\theta \leq \Delta_k$, alors la tâche A peut être insérée après la tâche a_k et complétée avant que la tâche a_{k+1} ne débute : l'ordonnancement d'origine n'est pas modifié. Si $\theta > \Delta_k$, la tâche a_{k+1} est déplacée de $\theta - \Delta_k$. Alors, si $\theta - \Delta_k \leq f_{k+1}$, le délai d'exécution de a_{k+1} n'est pas violé, et l'augmentation de C_n est égale à zéro, sinon l'augmentation de C_n est $\theta - \Delta_k - f_{k+1}$. Ceci conclut l'explication.

L'augmentation de C_n résultant du déplacement de a_{k+2} sur l'axe des temps est :

$$R_{k+2} = \begin{cases} 0 & \text{if} \quad \theta \leq \Delta_k + \Delta_{k+1} \\ (\theta - \Delta_k - \Delta_{k+1} - f_{k+2})^+ & \text{si} \quad \theta > \Delta_k + \Delta_{k+1} \end{cases} \tag{2.6}$$

L'explication de (2.6) est la suivante. Si $\theta \leq \Delta_k + \Delta_{k+1}$, l'instant de départ de la tâche a_{k+2} n'est pas modifié, tout comme celui des tâches suivantes. Si

$\theta > \Delta_k + \Delta_{k+1}$, alors a_{k+2} est déplacée de $\theta - \Delta_k - \Delta_{k+1}$. Si $\theta - \Delta_k - \Delta_{k+1} \leq f_{k+2}$, le délai d'exécution de a_{k+2} n'est pas violé et l'augmentation de C_n résultant de a_{k+2} est égale à zéro, autrement, l'augmentation de C_n est $\theta - \Delta_k - \Delta_{k+1} - f_{k+2}$. Ceci termine l'explication de (2.6).

La première tâche à ne pas être retardée est la tâche a_{k+n_k+1}, où n_k est le plus petit entier tel que $\sum_{s=0}^{n_k} \Delta_{k+s} \geq \theta$. Si $\Delta_k \geq \theta$, alors aucune tâche n'est retardée.

Finalement, l'augmentation de la valeur du critère est :

$$L_k(\theta) = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta \\ \sum_{s=1}^{n_k} R_{k+s} & \text{si} \quad \Delta_k < \theta \end{cases} \tag{2.7}$$

Cette formule se réécrit :

$$L_k(\theta) = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta \\ \sum_{s=1}^{n_k} (\theta - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ & \text{si} \quad \Delta_k < \theta \end{cases} \tag{2.8}$$

Nous sommes donc en mesure de calculer l'augmentation de la valeur du critère associé à l'ordonnancement initial dès que l'on connait k, c'est-à-dire l'endroit où l'on va intercaler la tâche A, et n_k, c'est-à-dire le nombre de tâches successives qui vont être décalées.

Bien entendu, nous ne pouvons pas essayer toutes les positions possibles de A et conserver la meilleure : ce ne serait pas compatible avec la contrainte "*temps réel*". Nous devons donc trouver un moyen d'évaluer $L_k(\theta)$ de manière rapide.

2.4 Utilisation des relations de base.

Lorsqu'une tâche aléatoire A intervient, nous connaissons sa durée θ et son délai impératif D. Nous devons alors décider, en *temps réel*, de la placer à la suite de l'une des tâches a_i déjà ordonnancées, l'objectif étant d'augmenter aussi peu que possible le critère C_n attaché à l'ordonnancement initial. Un certain volume de calculs est nécessaire pour parvenir à cette décision. L'objectif de ce paragraphe est de présenter des approches permettant d'effectuer la plupart des calculs en différé, c'est-à-dire dès que l'on connait l'ordonnancement

et avant qu'apparaîsse la tâche aléatoire, et de ne réserver à la démarche *temps réel* qu'un volume limité de calculs.

2.4.1 Première approche.

La première approche que nous présentons conduira à l'algorithme $TR1$ qui fournit des résultats relativement précis. Le lecteur observera cependant que le volume des calculs à effectuer en temps réel nécessite l'exploration de plusieurs positions de la tâche aléatoire.

Les tableaux de référence.

Les calculs en différé consistent à construire ce que nous appellerons des tableaux de référence pour décider du positionnement de la tâche A. Considérons l'ordonnancement initial de la figure 2.2. Les délais ne sont pas portés sur la figure.

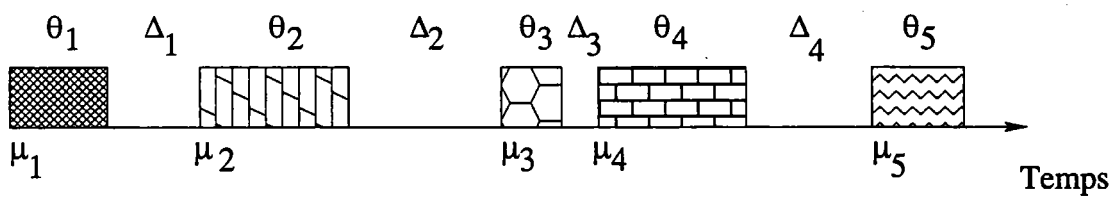


FIG. 2.2 – Un exemple d'ordonnancement

La tâche aléatoire A, de paramètres θ et D, se manifeste à l'instant 0. Considérons, par exemple, l'instant de début μ_4 de la tâche a_4. Si $\theta \leq \mu_4 - (\mu_3+\theta_3)$, alors placer la tâche aléatoire A après a_3 garantit qu'elle se terminera au plus tard à l'instant μ_4. Si par contre, $\mu_4 - (\mu_3 + \theta_3) < \theta \leq \mu_4 - (\mu_2 + \theta_2)$, il faudra placer A après a_2 pour être assuré qu'elle se terminera au plus tard à l'instant μ_4. De même, si $\mu_4 - (\mu_2 + \theta_2) < \theta \leq \mu_4 - (\mu_1 + \theta_1)$ alors il faudra placer A après a_1 pour être assuré qu'elle se terminera au plus tard à l'instant μ_4. Enfin, si $\theta > \mu_4 - (\mu_1 + \theta_1)$, il est impossible que A se termine au plus tard à l'instant μ_4.

De plus, comme nous l'avons vu sans le paragraphe 2.3 :

- si $\theta \leq \Delta_3$, aucune tâche de l'ordonnancement initial n'est déplacée si l'on place A après a_3.
- si $\Delta_3 < \theta \leq \Delta_3 + \Delta_2$, la tâche a_3 sera retardée de $\theta - \Delta_2$ si l'on place A après a_2.

- si $\Delta_3 + \Delta_2 < \theta \leq \Delta_3 + \Delta_2 + \Delta_1$, la tâche a_2 sera retardée de $\theta - \Delta_1$ et la tâche a_3 sera retardée de $\theta - \Delta_1 - \Delta_2$ si l'on place A après a_1.

Si donc on connait un instant de début μ_k de la tâche a_k avant lequel on souhaite que la tâche aléatoire A se termine, il est possible de donner la liste des tâches à la suite desquelles placer A pour atteindre ce but.

Le tableau de référence X est déduit de ces remarques. Il comporte n colonnes et, en théorie, n lignes. Considérons la ligne i et la colonne j du tableau X.

L'élément x_{ij} de X est défini de la manière suivante :

$$x_{ij} = \begin{cases} 0 & \text{si} \quad j \leq i \\ \sum_{k=i}^{j-1} \Delta_k & \text{si} \quad j > i \end{cases} \quad \text{pour } i = 1, 2, \ldots, n. \tag{2.9}$$

x_{ij} représente la valeur maximale de θ qui permet de ne déplacer que les tâches $a_{i+1}, a_{i+2}, \ldots, a_{j-1}$ si A est placé après a_i.

Le tableau de référence X qui correspond à l'exemple de la figure 2.2 est le tableau 2.1.

TAB. 2.1 – Tableau de référence X relatif à l'ordonnancement représenté dans la figure 2.2.

	μ_1	μ_2	μ_3	μ_4	μ_5
a_1	0	Δ_1	$\sum_{i=1}^{2} \Delta_i$	$\sum_{i=1}^{3} \Delta_i$	$\sum_{i=1}^{4} \Delta_i$
a_2	0	0	Δ_2	$\sum_{i=2}^{3} \Delta_i$	$\sum_{i=2}^{4} \Delta_i$
a_3	0	0	0	Δ_3	$\sum_{i=3}^{4} \Delta_i$
a_4	0	0	0	0	Δ_4

De ce tableau initial, nous déduisons le tableau de références Y et défini comme suit (ses éléments sont notés y_{ij}) :

$$y_{ij} = \begin{cases} 0 & \text{si} \quad j \leq i \\ x_{ij} + f_j & \text{si} \quad j > i \end{cases} \quad \text{pour } i = 1, 2, \ldots, n. \tag{2.10}$$

Le tableau de référence Y , appliqué à l'exemple précédent, est représenté dans la figure 2.2.

Au tableau de référence Y, qui va nous permettre de calculer l'augmentation de critère, s'ajoute le tableau de référence V, et dont les éléments v_{ij} se définissent comme suit.

$$v_{ij} = \begin{cases} 0 & \text{si} \quad j \leq i \\ \mu_j - (\mu_i + \theta_i) & \text{si} \quad j > i \end{cases} \quad \text{pour } i = 1, 2, \ldots, n. \tag{2.11}$$

Le tableau de références V, pour l'exemple ci-dessus, est représenté dans le tableau 2.3.

Lorsque $j > i$, ce tableau fournit la valeur maximale de la durée θ de la tâche T pour qu'elle puisse être terminée avant l'instant μ_j si elle débute dès que a_i se termine.

Les tableaux de référence, ainsi que le tableau initial, ne dépendent que de l'ordonnancement initial. Ils peuvent donc être calculés en différé. Comme seules les parties triangulaires supérieures contiennent les informations qui nous intéressent, nous aurions à stocker $(n-1)(n-2)/2$ éléments pour chaque tableau. Pour n grand, (plus grand que 100 par exemple), nous risquons d'aboutir à des tableaux d'une dimension telle qu'ils ne soient pas gérables en temps réel. Nous avons donc décidé de limiter le nombre de lignes à Z. Le nombre Z dépend évidemment du problème à traiter : il est d'autant plus grand que le nombre de Δ_i successifs dont la somme dépasse θ est grand. Pour cette valeur, le nombre d'éléments utiles (différents de zéro) du tableau de référence est $\frac{Z(2n-Z-1)}{2}$.

TAB. 2.2 – Tableau de référence Y relatif à l'ordonnancement représenté dans la figure 2.2

	μ_1	μ_2	μ_3	μ_4	μ_5
a_1	0	$\Delta_1 + f_2$	$\sum_{i=1}^{2} \Delta_i + f_3$	$\sum_{i=1}^{3} \Delta_i + f_4$	$\sum_{i=1}^{4} \Delta_i + f_5$
a_2	0	0	$\Delta_2 + f_3$	$\sum_{i=2}^{3} \Delta_i + f_4$	$\sum_{i=2}^{4} \Delta_i + f_5$
a_3	0	0	0	$\Delta_3 + f_4$	$\sum_{i=3}^{4} \Delta_i + f_5$
a_4	0	0	0	0	$\Delta_4 + f_5$

TAB. 2.3 – Tableau de référence V relatif à l'ordonnancement représenté dans la figure 2.2.

	μ_1	μ_2	μ_3	μ_4	μ_5
a_1	0	$\mu_2-(\mu_1+\theta_1)$	$\mu_3-(\mu_1+\theta_1)$	$\mu_4-(\mu_1+\theta_1)$	$\mu_5-(\mu_1+\theta_1)$
a_2	0	0	$\mu_3-(\mu_2+\theta_1)$	$\mu_4-(\mu_2+\theta_2)$	$\mu_5-(\mu_2+\theta_2)$
a_3	0	0	0	$\mu_4-(\mu_3+\theta_3)$	$\mu_5-(\mu_3+\theta_3)$
a_4	0	0	0	0	$\mu_5-(\mu_4+\theta_4)$

Utilisation des tableaux de référence.

Supposons qu'une tâche aléatoire A se manifeste à l'instant 0 et que $D = \mu_k$. Alors les positions admissibles pour A, c'est-à-dire les positions qui garantissent que D n'est pas violé, s'obtiennent en retenant dans la k-ième colonne du tableau V les indices des lignes $i = 1, 2, \ldots, p$ telles que $v_{ik} \geq \theta$. Soit I_k l'ensemble de ces indices. Alors pour tout $i \in I_k$, la relation (2.8) s'applique pour $L_i(\theta)$, avec $n_i = k - i$:

$$L_i(\theta) = \sum_{s=1}^{n_i} (\theta - \sum_{p=0}^{s-1} \Delta_{i+p} - f_{i+s})^+ \tag{2.12}$$

avec toujours la même convention sur les sommes, à savoir $\sum_{i=n_1}^{n_2} . = 0$ si $n_2 < n_1$

En utilisant les éléments du tableau X, (2.12) se réécrit :

$$L_i(\theta) = \sum_{s=1}^{n_i} (\theta - x_{i,i+s} - f_{i+s})^+ \text{ pour tout } i \in I_k \tag{2.13}$$

Par conséquent, en utilisant les éléments du tableau Y :

$$L_i(\theta) = \sum_{s=1}^{n_i} (\theta - y_{i,i+s})^+ \text{ pour tout } i \in I_k \tag{2.14}$$

La procédure précédente s'applique non seulement pour $D = \mu_k$, mais aussi pour $D \in [\mu_k, \mu_k + \theta_k]$. Si $D \in [\mu_k + \theta_k, \mu_{k+1})$, alors la procédure s'applique en choisissant $\mu_{k+1} = D$ et $\Delta_k = D - (\mu_k + \theta_k)$. Dans l'algorithme qui suit, nous ne considérons pas ce cas et nous poserons $D = \mu_k$ tant que $D \in [\mu_k, \mu_{k+1})$: c'est la raison pour laquelle notre approche est qualifiée d'heuristique, ce qui explique que les résultats obtenus ne soient pas tous optimaux.

Enfin, si $I_k = \emptyset$, il n'existe aucune position de A qui lui permette de se terminer au plus tard à l'instant μ_k : le problème n'a pas de solution.

Bien entendu, le fait que le tableau de référence soit limité à Z lignes peut conduire à éliminer des solutions admissibles parmi lesquelles pourraient se trouver la solution optimale.

2.4.2 Seconde approche.

Cette seconde approche est particulièrement simple. Elle consiste à calculer, pour $k = 1, 2, \ldots, n$, T_k qui est la somme maximale de k périodes Δ_i consécutives. En d'autres termes :

$$T_k = \max_{i=1,2,\ldots,n-k+1} \sum_{s=0}^{k-1} \Delta_{i+s} \tag{2.15}$$

On désigne par $i^*(k)$ l'indice i qui conduit au maximum. L'idée de cette seconde approche est de rechercher la valeur minimale de k telle que $T_k > \theta$ lorsque la tâche aléatoire A apparait. Soit k^* cette valeur. Alors deux cas peuvent se présenter :

1. Si $i^*(k^*)$ est tel que
   $\mu_{i^*(k^*)} + \theta_{i^*(k^*)} + \theta \leq D$, c'est-à-dire s'il est possible de terminer la tâche A avant D en la plaçant après $i^*(k^*)$, alors on place A après $i^*(k^*)$.
2. Si $\mu_{i^*(k^*)} + \theta_{i^*(k^*)} + \theta > D$, alors on place la tâche A après la tâche 1 si $\mu_1 + \theta_1 + \theta \leq D$, sinon le problème n'a pas de solution.

On remarquera que cette approche ignore totalement la flexibilité des tâches. elle cherche simplement à placer la tâche A à un endroit qui perturbe un nombre minimal de tâches déjà ordonnancées. Si cette position conduit à violer le délai D, alors on essaie de placer A après la première tâche. Si cette position conduit encore à violer D, alors le problème n'a pas de solution. Le souhait de ne déplacer qu'un minimum de tâches s'explique par la nécessité de mettre à jour l'ordonnancement, problème qui n'est pas examiné ici.

2.5 Les algorithmes heuristiques.

2.5.1 L'algorithme $TR1$.

C'est l'algorithme déduit de la première approche. A la lumière du paragraphe précédent, nous voyons que les calculs se divisent clairement en deux parties : la première regroupe les calculs qui peuvent se dérouler en différé, c'est-à-dire dès que l'on connait l'ordonnancement et avant que n'apparaisse une tâche aléatoire, alors que la seconde partie se déclenche dès qu'apparait une tâche aléatoire.

Remarque : Les tableaux X,Y et V précédents sont calés sur le début de la tâche a_1. Le temps s'écoulant, la tâche a_1 devient une tâche du passé, et la première tâche à exécuter devient la tâche a_2, et ainsi de suite. Il est donc nécessaire de modifier ces tableaux au cours du temps. Une approche simple consiste :

- à conserver ces tableaux constants entre le début d'une tâche et le début de la tâche suivante : la première tâche reste la même tout au long de cette période.
- à ajuster les tableaux X, Y et V en fin de période en soustrayant Δ_i à chaque élément de la i-ème ligne.
- à recalculer le tableau Y à partir du nouveau tableau de référence X.

Ces aspects ne sont pas considérés dans ce chapitre. Une description de cet algorithme est donné dans la figure 2.3.

2.5.2 L'algorithme $TR2$.

Cet algorithme est déduit de la seconde approche présentée dans le paragraphe 2.4.2. Il comporte également deux parties : celle qui concerne les calculs en différé, et celle qui concerne les calculs en *temps réel*, lesquels s'effectuent lorsqu'apparait la tâche aléatoire. Cet algorithme est détaillé dans la figure 2.4.

1. Partie 1 : Calculs en différé.
   (a) Calcul du tableau X à l'aide de la relation (2.9)
   (b) Calcul du tableau V à l'aide de la relation (2.11)
   (c) Calcul du tableau Y à l'aide de la relation (2.10)
2. Partie 2 : Calculs en *temps réel* :
   (a) Définir μ_k tel que $D \in [\mu_k, \mu_{k+1})$.
   (b) Faire $I_k = \emptyset$
   (c) Pour $i = 1, 2, \ldots$ faire :
   Si $v_{i,k} \geq \theta$ faire $I_k = I_k + \{i\}$, sinon aller en 2.(d)
   (d) Si $I_k = \emptyset$, pas de solution. Fin.
   (e) Pour tout $i \in I_k$, calculer $L_i(\theta)$ (voir (2.14))
   (f) Calculer i^* tel que $L_{i^*}(\theta) = \min_{i \in I_k} L_i(\theta)$
   (g) A sera placé après a_{i^*}

FIG. 2.3 – *Algorithme TR1*

1. Partie 1 : Calculs en différé.
   (a) Pour $k = 1, 2, \ldots, n$, calculer T_k et $i^*(k)$. T_k est calculé à partir de la relation (2.15)
2. Partie 2 : Calculs en *temps réel* :
   (a) Pour $k = 1, 2, \ldots, n$,
      i. Si $T_k < \theta$, aller en 2.(b)
      ii. Si $T_k \geq \theta$, stocker $i^*(k)$ et aller en 2.(d)
   (b) Fin de la boucle k
   (c) Pas de solution : FIN.
   (d) Si $\mu_{i^*(k)} + \theta_{i^*(k)} + \theta \leq D$, on place la tâche après la tâche $i^*(k)$. FIN.
   (e) Si $\mu_{i^*(k)} + \theta_{i^*(k)} + \theta > D$
      i. Si $\mu_1 + \theta_1 + \theta \leq D$, on place A après la tâche 1. FIN.
      ii. Si $\mu_1 + \theta_1 + \theta > D$, pas de solution. FIN.

FIG. 2.4 – *Algorithme TR2*

2.6 Le programme de simulation.

2.6.1 Présentation du programme

Dans cette section nous présentons le protocole de simulation que nous avons appliqué afin de tester les performances des deux heuristiques $TR1$ et $TR2$. Pour cela, nous générons aléatoirement des jeux de test (voir paragraphe suivant). Pour chacun de ces jeux de test, nous calculons à l'aide d'une exploration systématique (voir 2.6.2) quel est l'emplacement d'insertion optimal (notons qu'il peut y avoir plusieurs emplacements d'insertion optimaux). Ensuite, nous appliquons successivement les heuristiques $TR1$ et $TR2$, et nous comparons les résultats à ceux fournis par l'exploration systématique. Nous distinguerons trois types de résultats :

- Les résultats fournis par une heuristique qui conduisent à une solution optimale.
- Les résultats qui excèdent au plus de 10% le résultat optimal.
- Les résultats qui excèdent au plus de 15% le résultat optimal.

2.6.2 Algorithme systématique

Cet algorithme effectue les calculs correspondant à l'insertion de la tâche aléatoire A de durée θ et de délai impératif D après chaque tâche a_k, $k \in H_Q$, de l'ordonnancement initial avec $H_Q = \{k \setminus k \in [1, 2, \ldots, Q]$ et $\mu_Q + \theta_Q + \theta \leq D < \mu_{Q+1} + \theta_{Q+1} + \theta\}$. On calcule alors l'augmentation de critère associée à chacun de ces cas de figure. L'emplacement d'insertion optimal est celui qui conduit à la plus faible augmentation de la valeur du critère. Cet algorithme est donné dans la figure 2.5.

2.6.3 Algorithme de simulation

L'algorithme du programme de simulation est donné dans la figure 2.6.

Chaque ordonnancement est généré de manière aléatoire. On génère 35 tâches séparées éventuellement par un temps libre.

- Le temps libre qui suit une tâche a_k est généré par la formule :
  $T = 15+10*(12*\text{random}(1.0)-6)^+$ où $(x)^+ = \max(0, x)$ et $random(1.0)$ est un nombre aléatoire réel compris dans l'intervalle $[0, 1[$.

1. Calculer Q, le plus grand entier tel que $\mu_Q + \theta_Q + \theta \leq D$.
   a_Q est la dernière tâche après laquelle A peut démarrer sans violer D
2. Poser $K^* = \emptyset$.
   K^* Contient la liste des endroits d'insertion menant à la plus faible augmentation de la valeur du critère
3. Pour $k \in H_Q$ faire :
   Ainsi que défini préalablement,
   $H_Q = \{k \setminus k \in \{1, 2, \ldots, Q\}$ *et* $\mu_Q + \theta_Q + \theta \leq D < \mu_{Q+1} + \theta_{Q+1} + \theta\}$
   (a) Etablir le nouvel ordonnancement O_k résultant de l'insertion de A après a_k.
   (b) Calculer l'augmentation de critère de l'ordonnancement O_k par rapport au critère associé à l'ordonnancement original en utilisant (2.8).
   (c) Si $L_p(\theta) = L_k(\theta)$, avec $p \in K^*$ alors $K^* = K^* \cup \{k\}$.
   (d) Si $L_p(\theta) < L_k(\theta)$, avec $p \in K^*$ alors $K^* = \{k\}$.

FIG. 2.5 – *Algorithme systématique*

- Chacune des tâches est caractérisée par une date de début, un temps d'exécution et un délai.

  1. Le temps de début de la première tâche est zéro. Ensuite, pour obtenir le temps de début de la tâche que l'on est en train de générer, on additionne les temps d'exécution des tâches précédentes et les temps libres.
  2. Le temps d'exécution d'une tâche est généré aléatoirement, et de manière équiprobable entre 1 et 50.
  3. Le délai est calculée en additionnant la date de début de la tâche, le temps d'exécution et un nombre généré aléatoirement et de manière non équiprobable entre -50 et 50. Il y a 20% de chances pour que le nombre aléatoire prenne une valeur comprise entre -50 et 0.

1. Pour $\theta = 20$ à 100 par pas de 20 faire :
2. (a) Pour $i = 1$ à 400 faire :
   (b) i. Générer un ordonnancement.
   ii. Générer D : On le génère après le temps 700 et avant le temps de commencement de la dernière tâche schedulée.
   iii. Trouver l'insertion optimale pour θ et$D$ (voir algorithme 2.6.2)
   iv. Appliquer l'heuristique étudiée.

FIG. 2.6 – *Algorithme de simulation*

2.6.4 Complexité

Algorithme $TR1$.

Examinons la complexité de l'algorithme $TR1$. Pour cela, nous regardons d'abord quelle est sa complexité pour la partie concernant les calculs en différé. Nous partons des équations (2.9), (2.10) et (2.11), dont nous détaillons les algorithmes ainsi :

Notons bien que nous calculons les valeurs de Δ_i et de f_i en même temps que les tableaux de référence.

Calcul de X :

1. Pour $i = 1$, *nbtask* faire
   (a) Pour $j = 1$, i faire $X[i][j] = 0$;
   (b) Pour $j = i+1$, *nbtask* Faire $X[i][j] = X[i][j\text{-}1] + \Delta_{j-1}$

On associe à ce calcul la complexité :

$$C_X = \sum_{i=1}^{n}\{\sum_{j=1}^{i} 1 + \sum_{j=i+1}^{n} 2\} = \frac{3n^2 - n}{2}$$

Calcul de Y :

1. Pour $i = 1$, *nbtask* faire
   (a) Pour $j = 1$, i faire $Y[i][j] = 0$;
   (b) Pour $j = i+1$, *nbtask* Faire $Y[i][j] = X[i][j\text{-}1] + f_{j-1}$

On associe à ce calcul la complexité :

$$C_Y = C_X = \sum_{i=1}^{n}\{\sum_{j=1}^{i} 1 + \sum_{j=i+1}^{n} 2\} = \frac{3n^2 - n}{2}$$

Calcul de V :

1. Pour $i = 1$, *nbtask* faire
   (a) Pour $j = 1$, i faire $V[i][j] = 0$;
   (b) Pour $j = i+1$, *nbtask* Faire $V[i][j] = \mu_j - \mu_i - \theta_i$

On associe à ce calcul la complexité :

$$C_V = \sum_{i=1}^{n}\{\sum_{j=1}^{i} 1 + \sum_{j=i+1}^{n} 4\} = \frac{5n^2 - 3n}{2}$$

Ainsi, nous obtenons le nombre d'instructions à exécuter :

$$\begin{aligned} C_{hors-ligne} &= 2C_X + C_V \\ &= 2 * \{\frac{3n^2 - n}{2}\} + \frac{5n^2 - 3n}{2} \\ &\leq \frac{11n^2}{2} \end{aligned}$$

Considérons maintenant la partie des calculs en *temps réel* en détaillant pour chaque ligne de l'algorithme :

- ligne 2.a : au plus n opérations
- ligne 2.b : 1 opérations
- ligne 2.c : au plus n fois 3 opérations (test, addition, affectation).
- ligne 2.e : au plus n fois la formule (13), qui comporte au plus 3 * (n-2) opérations
- ligne 2.f : au plus n opérations.

En sommant, on obtient :

$$C_{en-ligne} = n + 1 + 3n + \sum_{i=1}^{n} \sum_{j=1}^{n-2} 3 + n = 3n^2 - n + 1 \leq 3n^2$$

Algorithme $TR2$.

Calculons maintenant la complexité de l'algorithme $TR2$. Pour cela, nous évaluons d'abord sa complexité pour la partie concernant les calculs en différé. Nous partons de l'équation (2.15), nous en dédu.sons l'algorithme :

1. Pour $i = 2$, *nbtask* faire $T[1][i] = \Delta_{i-1}$;
2. Pour $i = 2$, *nbtask* $- 1$ faire
   (a) Pour $i =$ i, *nbtask* faire $T[i][j] = T[i - 1][j] + \Delta_{j-i}$

et nous obtenons le nombre d'instructions à exécuter :

$$C_{hors-ligne} = \sum_{i=1}^{n} 1 + \sum_{i=2}^{n-1} \sum_{j=i}^{n} 2 = n^2 - 3 \leq n^2$$

Considérons maintenant la partie des calculs en *temps réel* :

- ligne 2.a : au plus n opérations.
- le reste : au plus 5 opérations.

En sommant, on obtient :

$$C_{en-ligne} = \sum_{i=1}^{n} 1 + 5 = n + 5 \leq 2n$$

Le tableau 2.4 résume les résultats de complexité des deux approches heuristiques $TR1$ et $TR2$.

TAB. 2.4 – Complexité des heuristiques $TR1$ et $TR2$.

	$TR1$	$TR2$
Partie différé	$O(n^2)$	$O(n^2)$
Partie *temps réel*	$O(n^2)$	$O(n)$

2.6.5 Les résultats numériques.

Nous avons traité deux mille exemples avec chacun des algorithmes $TR1$ et $TR2$. Dans chacun de ces exemples, nous avons recherché la (ou les) solution(s) optimale(s) à l'aide d'un balayage systématique.

On donne dans le tableau 2.5, pour chacune de ces heuristiques :

- le pourcentage d'exemples qui conduisent à une solution optimale,
- le pourcentage d'exemples qui conduisent à des solutions dont le critère se situe à moins de 10% du critère optimal,
- le pourcentage d'exemples qui conduisent à des solutions dont le critère se situe à moins de 15% du critère optimal.

TAB. 2.5 – Résultats

	0%	$\leq$ 10%	$\leq$ 15%
$TR1$	92%	95%	97%
$TR2$	84.5%	84.7%	89.6%

2.7 Conclusion.

L'objectif de ce chapitre était d'intercaler une tâche aléatoire de délai impératif dans un ordonnancement donné en minimisant l'accroissement de la valeur du critère, qui est la somme des retards des tâches préalablement ordonnancées. Cet objectif devrait être atteint en temps réel.

L'heuristique $TR1$ nous fournit des résultats de très bonne qualité, mais au prix d'une complexité élevée. L'heuristique $TR2$ nous fournit des résultats qui restent de bonne qualité par rapport à l'heuristique $TR1$, mais pour une durée de calcul *temps réel* plus acceptable.

Il faut bien remarquer que D est petit par rapport à la durée de l'ordonnancement dans lequel on veut insérer la tâche aléatoire, aussi, seul un nombre $n^* \ll n$ de tâches sera pris en compte dans les calculs.

Résumé

Le problème que nous avons abordé concerne une ressource unique. Nous avons considéré un ordonnancement supposé optimal, et dont le critère est la somme des retards par rapport à des délais connus. Une tâche aléatoire intervient à un instant donné, sa durée est connue, et son délai est impératif. L'objectif est de placer cette tâche en *temps réel* de manière à augmenter aussi peu que possible le critère attaché à l'ordonnancement initial. Nous avons fourni deux heuristiques, chacune possédant une partie exécutée en différé et une partie exécutée en *temps réel.*

La première heuristique consiste à :

- Partie en différé :
  1. Calcul des tableaux de référence.
- Partie en *temps réel* :
  1. Calcul de la dernière tâche θ_{k_2} de l'ordonnancement initial à commencer avant le délai D
  2. Calcul de la première tâche θ_{k_1} à partir de laquelle l'insertion de la tâche aléatoire est possible sans bouger θ_{k_2}.
  3. L'insertion de la tâche aléatoire se fait juste après la tâche θ_i, $i \in [k_1, k_2]$, de façon que cette insertion conduise à une augmentation minimale du critère. (Notons bien qu'il peut ne pas y avoir de solution)

(Voir les paragraphes 2.4.1 et 2.5.1.)

La seconde heuristique recherche (si il elle existe) l'insertion de la tâche aléatoire minimisant le nombre de tâches à déplacer (Voir les paragraphes 2.4.2 et 2.5.2.) .

Chapitre 3

Insertion d'une tâche simple en temps réel : Méthode dérivée de l'analyse du critère.

3.1 Introduction.

Dans le chapitre précédent, nous avons cherché à insérer une tâche aléatoire en essayant de perturber le moins possible l'ordonnancement de départ, et en supposant que cette manière de procéder allait permettre d'atteindre un résultat satisfaisant.

Dans ce chapitre, nous cherchons à insérer la tâche aléatoire de façon à avoir la plus petite augmentation de la somme des retards des tâches. En effet, il est aisé de calculer pour chaque tâche de l'ordonnancement de départ quelle serait l'augmentation de critère si on insérait à sa suite une tâche aléatoire de longueur $\theta \in [1, \infty)$. En prenant le minimum de ces fonctions pour chaque valeur de θ, il est alors facile de connaître l'instant optimal d'insertion d'une tâche aléatoire de longueur $\theta^* \in [1, \infty)$.

Nous donnons une définition de ce problème dans le section 3.2. L'approche proposée dans ce chapitre est donnée dans la section 3.3. Un exemple illustratif est donné dans la section 3.4. La section 3.5 est la conclusion. Un résumé est donné en fin de chapitre.

3.2 Définition du problème et propriétés

Nous définissons les notations utilisées dans ce chapitre dans la première sous-section. Les relations fondamentales qui seront exploitées dans notre approche sont données dans la sous-section 3.3. La troisième sous-section est consacrée à la présentation du problème.

3.2.1 Notations et définitions

Les notations et définitions utilisées dans ce chapitre sont définies dans le paragraphe 2.2

3.2.2 Relations fondamentales

La définition de la fonction augmentation de critère $L_k(\theta)$ est donnée dans le paragraphe 2.3. Pour ce chapitre, pour des fins temps-réel, nous allons devoir réécrire cette dernière fonction en tenant compte des remarques suivantes :

1. Arrêter la sommation à $s = n_k + 1$ garantit que nous nous limitons aux tâches qui sont effectivement retardées par l'introduction de la tâche aléatoire de durée θ à la suite de la tâche a_k. Il est cependant possible de remplacer $n_k + 1$ par $+\infty$ car tout élément de la somme d'indice s supérieur à $n_k + 1$ est égal à zéro. En effet, la seconde inégalité de (2.4) montre que :

$$\theta - \sum_{p=0}^{n_k+1} \Delta_{k+p} \leq 0$$

et donc que :

$$\theta - \sum_{p=0}^{r} \Delta_{k+p} \leq 0 \text{ pour tout } r \geq n_k + 1$$

A fortiori :

$$\theta - \sum_{p=0}^{r} \Delta_{k+p} - f_{k+r+1} \leq 0 \text{ pour tout } r \geq n_k + 1$$

Si bien que $L_k(\theta)$ peut encore s'écrire :

$$L_k(\theta) = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta \\ \sum_{s=1}^{+\infty} (\theta - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ & \text{si} \quad \Delta_k < \theta \end{cases} \tag{3.1}$$

2. Si n est le nombre de tâches ordonnancées, et $n-k$ le nombre maximal de tâches qui peuvent être perturbées par l'introduction de la tâche aléatoire après la tâche a_k, alors $L_k(\theta)$ peut encore s'écrire :

$$L_k(\theta) = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta \\ \sum_{s=1}^{n-k+1} (\theta - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ & \text{si} \quad \Delta_k < \theta \end{cases} \tag{3.2}$$

C'est cette formulation qui sera utilisée dans l'algorithme "temps réel".

3.2.3 Réduction de la charge de calcul

Le résultat qui est présenté ci-dessous peut amener à une large réduction de la charge de calculs.

Résultat 1

Si $\Delta_k = 0$ et si $\mu_{k+1} + t_{k+1} + \theta < D$, alors l'augmentation du critère sera plus petite si nous faisons démarrer la tâche aléatoire lorsque a_{k+1} se termine plutôt que lorsque a_k se termine.

Preuve du résultat 1

$\Delta_k = 0$ signifie que $\mu_{k+1} = \mu_k + t_k$ ou, en d'autres termes, que a_{k+1} démarre aussitôt que a_k se termine.

Soit C_n^{k+1} l'augmentation de critère si nous démarrons la tâche aléatoire lorsque a_{k+1} se termine. Cette augmentation provient du déplacement de l'instant de début de a_{k+2} qui devient :

$$\mu'_{k+2} = \mu_{k+2} + (\theta - \Delta_{k+1})^+$$

Bien entendu, cette augmentation peut être égale à zéro.
Considérons maintenant le cas où la tâche aléatoire démarre aussitôt que a_k se termine. Dans ce cas :

- L'augmentation du critère C_n due au déplacement de l'instant de départ de a_{k+1} est $\Delta(R_{k+1}) = [d_{k+1} - (\mu_{k+1} + \theta + t_{k+1})]^+ - [d_{k+1} - (\mu_{k+1} + t_{k+1})]^+ \geq 0$. Cette égalité provient du fait que l'instant de début de a_{k+1} devient $\mu_{k+1} + \theta$ puisque $\Delta_k = 0$

- L'augmentation du critère C_n due au déplacement de l'instant de début de a_{k+2}, qui est toujours μ'_{k+2}.
  En fin de compte : $C_n^{k+2} = C_n^{k+1} - \Delta(R_{k+1})$ et puisque $\Delta(R_{k+1}) \geq 0$, $C_n^{k+2} \leq C_n^{k+1}$. Ceci termine la démonstration. □

La figure 3.1 illustre le Résultat 1.

Comme nous pouvons le constater, les mêmes tâches subissent les mêmes déplacements dans les cas b. et c., exception faite de la tâche a_{k+1} qui est déplacée dans le cas b. mais non dans le cas c. L'augmentation de C_n^{k+1} est due au fait que a_{k+2} et a_{k+3} sont déplacées, tandis que $\Delta(R_{k+1})$ est l'augmentation de critère résultant du fait que a_{k+1} est déplacée dans la cas b.
Le résultat 1. amène à un corollaire intéressant qui permet de réduire la charge de calcul dans la majorité des cas.

Corollaire :

Soit 0 l'instant où apparaît la tâche aléatoire, et soit $a_1, a_2, \ldots, a_Q$ les tâches préalablement ordonnancées, où Q et le plus grand nombre entier tel que $\mu_Q + t_Q + \theta \leq D$.

1. Supposons qu'il existe au moins un $k \in \{1, 2, \ldots, Q\}$ tel que $\Delta_k > 0$ et soit $W_Q = \{k \setminus k \in \{1, 2, \ldots, Q\} \text{ et } \Delta_k > 0\} \cup \{Q\}$. Alors, la position optimale de A se trouve juste après l'un des $a_k, k \in W_Q$. En d'autres termes, il est inutile de considérer les emplacements situés après les tâches a_k telles que $\Delta_k = 0$, sauf peut-être après a_Q.

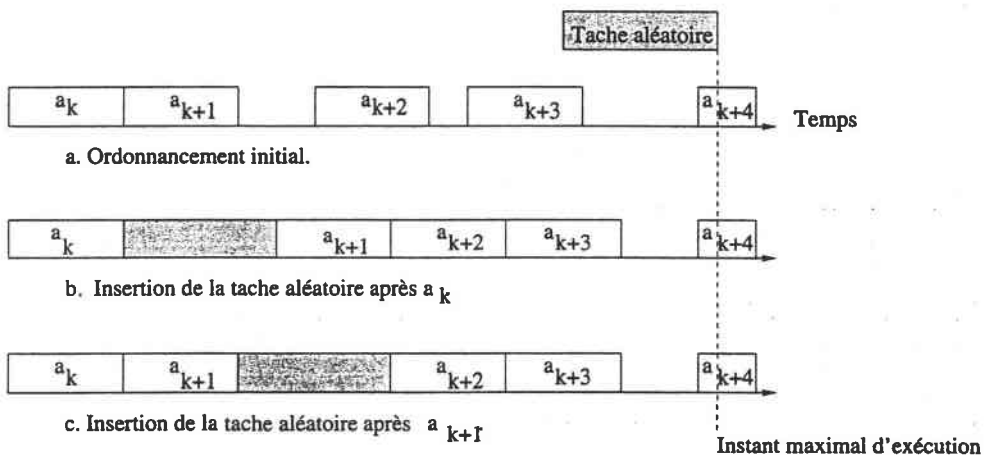


FIG. 3.1 – Exemple illustratif du Résultat 1.

2. Si $W_Q \setminus \{Q\} = \emptyset$, démarrer la tâche A aussitôt que a_Q se termine est optimal.

Le corollaire est une conséquence directe du résultat 1.

3.2.4 Approche non contrainte

En utilisant les résultats ci-dessus, et en supposant que les contraintes temps-réel ne s'appliquent pas, l'algorithme nous donnant la meilleure insertion d'une tâche aléatoire dans un ordonnancement existant est appelé algorithme d'approche non contrainte et est décrit dans la figure 3.2, et l'on suppose que :

- la tâche aléatoire apparaît à l'instant 0.
- sa durée est θ.
- son délai est D.

Cet algorithme calcule quelles sont les tâches de l'ordonnancement initial après lesquelles il est judicieux d'insérer une tâche aléatoire (tâches suivies d'un temps libre, et dont l'emplacement temporel permet à la tâche aléatoire de ne pas violer son délai impératif). Pour chacun de ces cas, on calcule quelle est l'augmentation de critère associé. Le (ou les) cas conduisant à une valeur minimale de l'augmentation du critère sont la solution. Enfin, on calcule à partir de la solution, le nouvel emplacement des tâches ayant dû être déplacées.

1. Calculer Q, le plus grand entier tel que $\mu_Q + t_Q + \theta \leq D$.
   a_Q est la dernière tâche après laquelle A peut démarrer sans violer D
2. Calculer Δ_k et f_k pour $k \in \{1, 2, \ldots, Q\}$. (voir relations (2.1) et (2.2))
3. Pour $k \in W_Q$ faire :
   Ainsi que défini préalablement, $W_Q = \{k \backslash k \in \{1, 2, \ldots, Q\}$ et $\Delta_k > 0\} \cup \{Q\}$
   (a) Calculer n_k en utilisant (2.4)
   (b) Calculer $L_k(\theta)$ en utilisant (2.8)
4. Calculer k^* tel que :
   $L_{k^*}(\theta) = \min_{k \in W_Q} L_k(\theta)$
   La tâche aléatoire devrait démarrer lorsque a_{k^*} finit si nous voulons minimiser l'augmentation du critère
5. Calculer les nouveaux instants de départ des tâches déplacées, s'il y en a, c'est-à-dire :
   $\mu'_{k^*+s} = \mu_{k^*+s} + (\theta - \sum_{i=0}^{s-1} \Delta_{k+i})$ pour $s = 1, 2, \ldots, n_k$

FIG. 3.2 – *Algorithme d'approche non contrainte*

Note : Cet algorithme est différent de celui décrit dans le paragraphe 2.6.2. Dans cet algorithme, nous effectuions les calculs pour toutes les tâches de l'ordonnancement ; ici, nous appliquons les résultats du paragraphe précédent et n'effectuons des calculs que pour les tâches amenant une augmentation minimale de la valeur du critère.

3.2.5 Complexité de l'algorithme d'approche non contrainte

La complexité de cet algorithme est en $O(n^2)$ (voir paragraphe 3.3.5 pour les détails du calcul). Visiblement, à cause de sa complexité, cet algorithme n'est pas applicable en temps réel. C'est pourquoi nous allons utiliser une approche sensiblement différente afin d'effectuer la majorité des calculs en différé.

3.3 L'approche temps-réel

3.3.1 Calculs en différé

La partie la plus complexe du calcul concerne la sélection de la tâche a_{k^*} après laquelle la tâche A devrait démarrer. La plupart de ces calculs peuvent être faits en différé.

A ce stade nous ne connaissons pas la tâche aléatoire qui risque d'intervenir. Nous ne pouvons donc pas déterminer l'ensemble des tâches ordonnancées admissibles, c'est-à-dire l'ensemble des tâches ordonnancées après lesquelles nous allons pouvoir placer la tâche aléatoire. Cependant, nous pouvons calculer $L_k(\theta)$ pour $k = 1, 2, \ldots, n$ (rappelons que n est l'ensemble des tâches ordonnancées) et pour $\theta \in [0, +\infty)$. Considérons la relation (3.2) et posons, pour $k \in \{1, 2, \ldots, n\}$:

$$z_{k,s} = \sum_{p=0}^{s-1} \Delta_{k+p} + f_{k+s} \text{ pour } s = 1, 2, \ldots, n-k+1 \qquad (3.3)$$

Notons : $z^*_{k,s}$, $s = 1, 2, \ldots, n-k+1$ l'ensemble des $z_{k,s}$ classés dans l'ordre croissant. A ce stade, il convient de noter que les $z_{k,s}$ peuvent ne pas être naturellement classés dans l'ordre croissant du fait de la présence des termes f_{k+s} qui sont positifs, mais pas forcément croissants par rapport à s.
Alors la relation (3.2) se réécrit, pour $k = 1, 2, \ldots, n$:

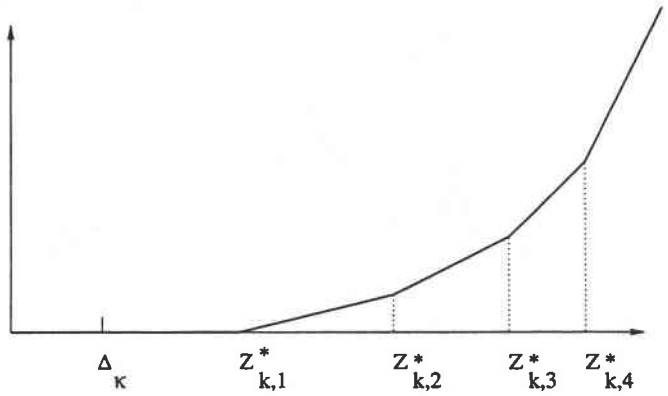


FIG. 3.3 – La fonction $L_k(\theta)$.

$$L_k(\theta) = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta \\ \sum_{s=1}^{n-k+1} (\theta - z^*_{k,s})^+ & \text{si} \quad \Delta_k < \theta \end{cases} \tag{3.4}$$

La fonction $L_k(\theta)$ est nulle sur $[0, z^*_{k,1}]$.
Pour $\theta \in [z^*_{k,1}, z^*_{k,2}]$, nous obtenons :
$L_k(\theta) = \theta - z^*_{k,1}$, qui est une fonction linéaire de pente 1.
De manière générale, si $\theta \in [z^*_{k,s}, z^*_{k,s+1}]$, alors :

$$L_k(\theta) = s.\theta - \sum_{p=1}^{s} z^*_{k,p} \tag{3.5}$$

Résultat 2

La fonction $L_k(\theta)$ est donc continue et linéaire par morceaux, comme représenté sur la figure 3.3.

Preuve du résultat 2

La fonction $L_k(\theta)$, sous la forme (3.5) est de la forme $a.x + b$, ce qui est l'écriture caractéristique d'une fonction linéaire (par conséquent continue). Cependant, ceci n'est vrai que sur un intervalle $[z^*_{k,s}, z^*_{k,s+1}]$, $s = 1, \ldots, n-k$: la pente de la fonction dépend du nombre de tâches déplacées.

Enfin, calculer $L_k(\theta)$ consiste à stocker les points :

$$(0,0), \{z^*_{k,i}, L_k(z^*_{k,i})\}_{i=1,\ldots,n-k+1}$$

Il faut en outre se souvenir que, pour $\theta > z^*_{k,n-k+1}$, l'équation de $L_k(\theta)$ est :

$$L_k(\theta) = (n-k+1).\theta - \sum_{p=1}^{n-k+1} z^*_{k,p}$$

Ceci termine la démonstration. □

Exemple 1 : Considérons un ordonnancement de quatre tâches a_1, a_2, a_3 et a_4 définies comme suit :

$$\begin{array}{lll} \mu_1 = 0, & t_1 = 1, & d_1 = 12 \\ \mu_2 = 4, & t_2 = 3, & d_2 = 18 \\ \mu_3 = 9, & t_3 = 2, & d_3 = 10 \\ \mu_4 = 14, & t_4 = 4, & d_4 = 21 \end{array}$$

Cet ordonnancement est représenté dans la figure (3.4).

En nous référant à la relation (2.1), nous obtenons :

$$\Delta_1 = 3, \Delta_2 = 2, \Delta_3 = 3, \Delta_4 = +\infty$$

Nous déduisons de la relation (2.2) :

$$f_1 = 11, f_2 = 11, f_3 = 0, f_4 = 3.$$

En appliquant la relation (3.3), nous obtenons :

$$\begin{array}{llll} z_{1,1} = 14 & z_{1,2} = 5 & z_{1,3} = 11 & z_{1,4} = +\infty \\ z_{2,1} = 2 & z_{2,2} = 8 & z_{2,3} = +\infty & \\ z_{3,1} = 6 & z_{3,2} = +\infty & & \\ z_{4,1} = +\infty & & & \end{array}$$

Après classement des $z_{k,s}$ par ordre croissant, il vient :

$$\begin{array}{llll} z^*_{1,1} = 5 & z^*_{1,2} = 11 & z^*_{1,3} = 14 & z^*_{1,4} = +\infty \\ z^*_{2,1} = 2 & z^*_{2,2} = 8 & z^*_{2,3} = +\infty & \\ z^*_{3,1} = 6 & z^*_{3,2} = +\infty & & \\ z^*_{4,1} = +\infty & & & \end{array}$$

Les fonctions $L_1(\theta)$,$L_2(\theta)$, et $L_3(\theta)$ sont représentées dans la figure (3.5). La fonction $L_4(\theta)$ n'est pas représentée car elle est identiquement égale à 0.

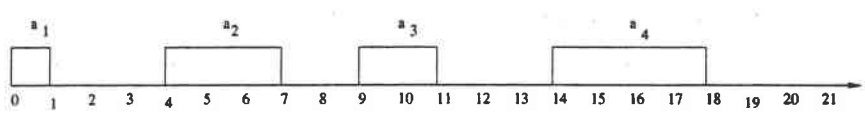


FIG. 3.4 – Ordonnancement initial (exemple 1).

Ceci est dû au fait qu'aucune tâche ne se trouve après a_4. Par conséquent, placer la tâche aléatoire après a_4 sera toujours optimal (Si le délai impératif de la tâche aléatoire le permet).

3.3.2 Accroîssement minimal du critère

Nous voulons extraire de ces fonctions $L_k(\theta)$ la fonction "augmentation du critère" notée $z_A(\theta)$. Pour celà, utilisons le fait que les fonctions $L_k(\theta)$ sont linéaires sur chacun des intervalles $[0, \Delta_k], [\Delta_k, z^*_{k,1}]\{[z^*_{k,i-1}, z^*_{k,1}]\}_{i=2,3,...,H_k}, k \in W_Q$.

Pour simplifier, nous notons ces ensembles d'intervalles $S_k = \{[s_{k,i}, s_{k,i+1}]\}_{i=0,1,...}$, où $s_{0,i} = 0$. Nous considérons les intersections des intervalles $\{S_k\}_{k \in W_Q}$ et obtenons un ensemble de sous-intervalles où toutes les fonctions $L_k(\theta)$, $k \in W_Q$ sont linéaires simultanément : $S^* = \{[s^*_g, s^*_{g+1}]\}_{g=0,1,...}$. Dans la figure 3.6, nous exhibons un de ces sous-intervalles dans le cas où $Q = 4$ et $W_Q = \{1, 2, 3, 4\}$. Dans cette figure, $Z_4(\theta)$ est représentée en gras.

Nous concluons cet exemple en remarquant que chaque sous-intervalle $[s^*_g, s^*_{g+1}]$ peut se diviser en sous-intervalles plus petits sur lesquels $Z_a(\theta)$ est linéaire et correspond à l'un des $L_k(\theta)$, $k \in W_Q$.

Dans l'exemple proposé figure 3.6, le sous-intervalle $[s^*_g, s^*_{g+1}]$ sera divisé en trois segments :

- Le sous-intervalle $[s^*_g, x(P_1)]$, où $x(P_1)$ est l'abscisse de P_1. Dans ce sous-intervalle, $Z_Q(\theta) = L_2(\theta)$.
- Le sous-intervalle $[x(P_1), x(P_2)]$, où $Z_Q(\theta) = L_1(\theta)$.
- Le sous-intervalle $[x(P_2), s^*_{g+1}]$, où $Z_Q(\theta) = L_3(\theta)$.

Dans la suite de ce paragraphe, nous allons définir un algorithme nous permettant de construire les sous-intervalles $\{[s^*_g, s^*_{g+1}]\}_{g=0,1,...}$ de $Z_Q(\theta)$ à partir des fonctions $L_i(\theta)$, $i \in \{1..Q\}$. Nous l'appellons algorithme de construction de $Z_Q(\theta)$.

Il commence par l'intialisation de la fonction $L_{k^*_1}(\theta)$ qui concerne le premier intervalle. C'est la fonction définie par :

$$L_{k^*_1}(s^*_g) = \min_{k \in W_Q} L_k(s^*_g)$$

Ensuite, l'algorithme calcule E_1, l'ensemble des index $s \in W_Q$ tels que $L_{k^*_1}(\theta) = L_s(\theta)$ pour un $\theta \in [s^*_g, s^*_{g+1}]$. Si E_1 est vide, alors $L_{k^*_1}(\theta) = Z_Q(\theta)$ sur

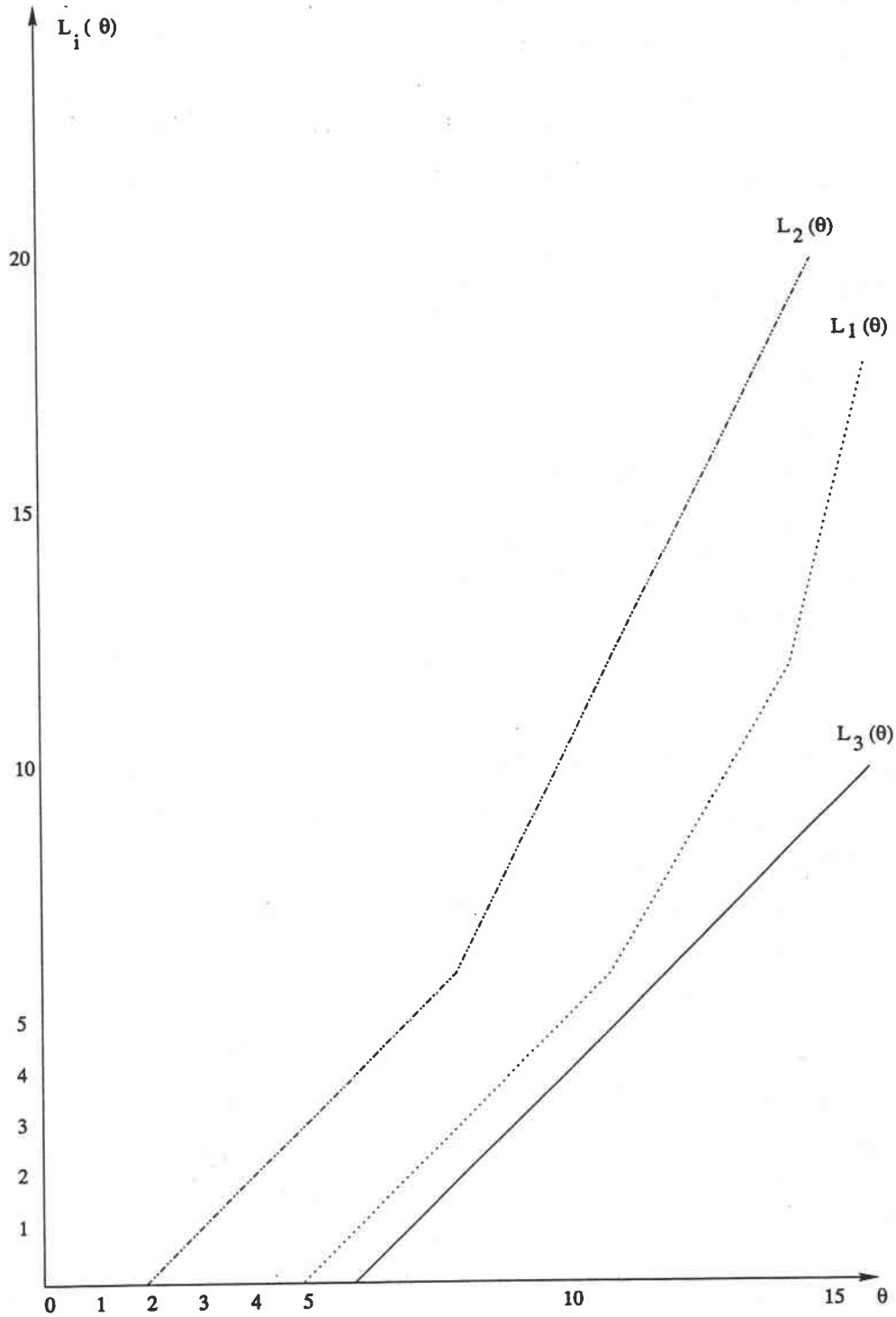


FIG. 3.5 – Fonctions L_1, L_2 et L_3.

l'intervalle $[s_g^*, s_{g+1}^*]$, et le calcul est fini. Sinon, une ou plusieurs des fonctions $L_s(\theta)$, $s \in E_1$ peuvent être minimales sur de futurs intervalles inclus dans $[s_g^*, s_{g+1}^*]$. Si θ_s est la solution de $L_{k_1^*}(\theta) = L_s(\theta)$ pour $s \in E_1$, l'extrémité du premier intervalle inclus dans $[s_g^*, s_{g+1}^*]$ est $\theta_1^* = \min_{s \in_1} \theta_s$ et $Z_Q(\theta) = L_{k_1^*}(\theta)$ pour $\theta \in [s_g^*, \theta_1^*]$.

Nous considérons maintenant $L_{k_2^*}(\theta)$ tel que $L_{k_2^*}(\theta_1^*) = L_{k_1^*}(\theta_1^*)$ et calculons E_2, ensemble d'index $s \in E$ tels que $L_{k_2^*}(\theta) = L_s(\theta)$ a une solution sur $[\theta_1^*, s_{g+1}^*]$.

Si E_2 est vide, alors $Z_Q(\theta) = L_{k_2^*}(\theta)$ pour $\theta \in [\theta_1^*, s_{g+1}^*]$, et l'algorithme s'arrête. Sinon, si θ_s est la solution de $L_{k_2^*}(\theta) = L_s(\theta)$ pour $s \in E_2$ et $\theta_2^* = \min_{s \in E_2} \theta_s$, on pose $Z_Q(\theta) = L_{k_2^*}(\theta)$ pour $\theta \in [\theta_1^*, \theta_2^*]$, et ainsi de suite. Nous formalisons l'algorithme de construction de $Z_Q(\theta)$ dans la figure 3.7. Il concerne un sous-intervalle $[s_g^*, s_{g+1}^*]$.

L'algorithme de construction de $Z_Q(\theta)$ est appliqué à chacun des sous-intervalles $\{[s_g^*, s_{g+1}^*]\}_{g=0,1,2,\ldots}$. Ceci amène finalement à une séquence d'intervalles dont les extrémités sont connues. De plus, un des $L_k(\theta)$ est associé à chacun des intervalles : cette fonction est $Z_Q(\theta)$.

3.3.3 Calculs en temps-réel

Lorsque la tâche aléatoire A définie par le couple (D, θ) apparait, nous sommes en mesure de calculer Q, plus grand entier tel que $\mu_Q + t_Q + \theta \leq D$.

Nous savons (voir résultat 1) que la position optimale de A est de le placer après un a_k, $k \in W_Q$, où $W_Q = \{k \setminus \Delta_k > 0 \text{ et } k \leq Q\} \cup \{Q\}$.

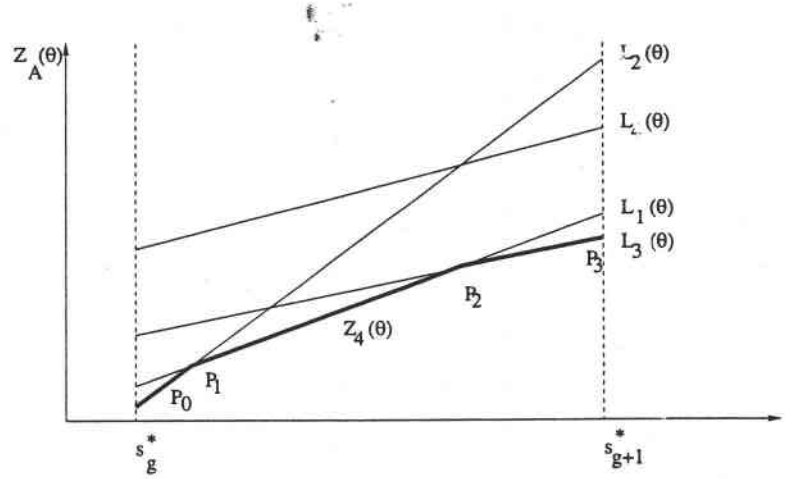


FIG. 3.6 – La fonction $z_A(\theta) = Min(L_k(\theta))$, $\theta \in [s_g^*, s_{g+1}^*]$, $k \in [1, 4]$.

1. On pose $i = 0$, $\theta_i^* = s_g^*$
2. On pose $E_0 = W_Q$.
3. Calculer $v_k = L_k(\theta_i^*)$ pour $k \in E_0$.
4. Calculer k_{i+1}^* tel que $v_{k_{i+1}^*} = \min_{k \in E_i} v_k$
   Si $v_k = v_{k_{i+1}^*}$ pour plusieurs $k \in W_Q$, on sélectionne comme k_{i+1}^* l'index tel que $L_{k_{i+1}^*}(\theta_i^* + \epsilon) = \min_{k \in E_i} L_k(\theta_i^* + \epsilon)$, où ϵ est positif. Si plusieurs k vérifient cette égalité, l'un deux est choisi au hasard comme k_{i+1}^*.
5. Calculer E_{i+1}, ensemble d'index s appartenant à $E_i \backslash \{k_{i+1}^*\}$, tel que l'équation $L_{k_{i+1}^*}(\theta) = L_s(\theta)$ ait une solution $\theta_s \in [\theta_i^*, s_{g+1}^*]$
6. Si $E_{i+1} = \emptyset$ alors
   (a) $\theta_{i+1}^* = s_{g+1}^*$ et $Z_Q(\theta) = L_{k_{k+1}^*}(\theta)$ pour $\theta \in [\theta_i^*, \theta_{i+1}^*]$.
   (b) Fin de l'algorithme.
7. Si $E_{i+1} \neq \emptyset$, alors :
   (a) On pose $\theta_{i+1}^* = \min_{s \in E_{i+1}} \theta_s$
   (b) Calculer k_{i+2}^* tel que $L_{k_{i+2}^*}(\theta_{i+1}^*) = L_{k_{i+1}^*}(\theta_{i+1}^*)$
   Si plusieurs $k \in E_{i+1}$ sont tels que $L_k(\theta_{i+1}^*) = L_{k_{i+1}^*}(\theta_{i+1}^*)$, alors nous sélectionnons celui qui correspond à la fonction ayant la dérivée minimale.
   (c) On pose $i = i + 1$
   (d) Aller en 5.

FIG. 3.7 – *Algorithme de construction de $Z_Q(\theta)$*

Le calcul temps-réel se limite donc au calcul de :

$$Z_A(\theta) = \min_{k \in W_Q} L_k(\theta)$$

où une seule valeur de θ est prise en compte. Si k^* est la valeur de k qui permet d'atteindre ce minimum, alors A est à placer après a_{k^*}.

Note : Pour le temps réel, il est hors de question de calculer toute les valeurs de la fonction $Z_A(\theta)$ (i.e. d'appliquer l'algorithme défini en 3.3.2) : seul le point $Z_A(\theta^*)$ où θ^* est la durée de la tâche aléatoire nous intéresse.

Le calcul des nouveaux instants de début des tâches s'effectue également en temps réel.(voir point 5 de l'algorithme d'approche non contrainte (voir 3.2.4))

Exemple 2 : Reprenons les données de l'exemple 1 et supposons que $D = 20$ et $\theta = 8$. Alors, nous obtenons $Q = 3$ et $W_Q = \{1, 2, 3\}$. Alors $Z_A(8) = \min_{k \in W_Q} L_k(8)$.

Le minimum est obtenu pour $L_3(8)$ et $Z_A(8) = 2$ (voir figure (3.5)). La solution optimale est donc de placer la tâche aléatoire A après a_3. Cette localisation

retarde a_4 de $8 - 3 = 5$ unités. Comme le délai de a_4 est 21, cette translation de a_4 amènera la fin de la tâche a_4 à l'instant 23, d'où une augmentation du critère de 23-21 = 2.

3.3.4 Algorithme général

Partant des remarques précédentes, et pour résoudre le problème en temps réel, nous définissons un algorithme appelé algorithme général donné dans la figure 3.8. Il se décompose en deux parties. La première partie (partie en différé) est appliquée dès que l'ordonnancement est modifié. La seconde (partie "temps réel") est appliquée dès qu'apparaît une tâche aléatoire.

1. Calculs en différé.
   Calcul de $L_k(\theta)$ pour $k = 1, 2, \ldots, n$ et $\theta \in [0, +\infty)$ comme indiqué dans le paragraphe 3.3.1.
2. Calculs en temps-réel.(A l'apparition de la tâche aléatoire de délai D et de durée θ).
   (a) Calcul de Q, plus grand entier tel que $\mu_Q + t_Q + \theta \leq D$
   (b) Calcul de $W_Q = \{k \setminus k \in \{1, 2, \ldots, Q\} \text{ et } \Delta_k > 0\} \cup \{Q\}$.
   (c) Calcul de l'accroîssement $z_A(\theta) = \min_{k \in W_Q} L_k(\theta)$
   Soit k^* la valeur de $k \in W_Q$ qui conduit au minimum.
   (d) Réajustement de l'ordonnancement :
   $$\mu'_{k^*+s} = \mu_{k^*+s} + (\theta - \sum_{i=0}^{s-1} \Delta_{k^*+i})^+ \text{ pour } s = 1, 2, \ldots, n - k^* + 1.$$

FIG. 3.8 – *Algorithme général*

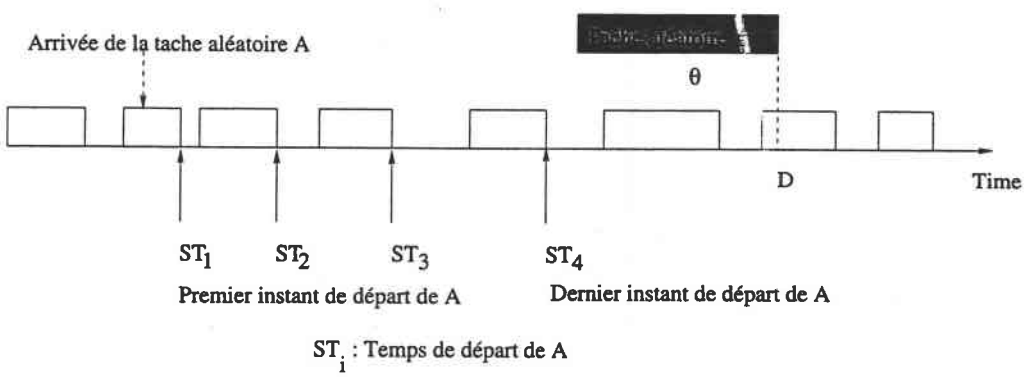


FIG. 3.9 – Instants possibles de départ de A.

3.3.5 Complexité de l'algorithme général

Pour le calcul de la complexité, nous considérons que l'ordonnancement existant comporte n éléments.

Complexité hors ligne.

Le calcul hors-ligne, c'est-à-dire le calcul des valeurs de la fonction $L_k(\theta)$ pour les valeurs de θ correspondantes aux bornes des segments sur lesquels $L_k(\theta)$ est linéaire se décompose en quatre parties :

1. Calcul des valeurs de Δ_i pour $i \in \{1, \ldots, n-1\}$ et de f_i pour $i \in \{1, \ldots, n\}$. (voir resp. équations 2.1 et 2.2). La complexité de ce calcul est en $O(n)$
2. Calcul $z_{k,s}$, pour $k \in \{1, \ldots, n\}$ et $s \in \{1, \ldots, n-k+1\}$ (utilisation de la formule 3.3).
   La complexité sera de : $\sum_{k=2}^{n} \sum_{s=1}^{n-k+1} 2$. Cette complexité est en $O(n^3)$
3. Le tri des $z_{k,s}$ par ordre croissant. Nous notons le résultat de ce tri $z^*_{k,s}$. Un tri efficace (par exemple tri à bulles) a une complexité en $O(n^2)$.
4. Calcul de $L_k(\theta)$, $k \in \{1, \ldots, n\}$ (utilisation de la formule 3.4). θ n'étant pas connu, le calcul se résume à sommer des valeurs de $z^*_{k,s}$. La complexité de cette étape est en $O(n^2)$.

La complexité associée à cet algorithme est donc en $O(n^3)$.

Complexité en ligne.

Nous réécrivons ici l'algorithme général (voir 3.3.4) de manière plus détaillée.

1. Q = 1
   complexité : 1
2. si $\mu_Q + t_Q + \theta \leq D$ alors
   Q=Q+1
   Aller en 2
   complexité : $\sum_{i=1}^{n} 6 = 6n$

3. $Q = Q - 1$
   $W_Q = \emptyset$
   complexité : 3

4. pour k = 1 à Q-1 faire
   si (Δ_k>0) alors $W_Q = W_Q \cup \{a_k\}$
   complexité : $\sum_{i=1}^{n-1} 4 = 4(n-1)$

5. $W_Q = W_Q \cup \{a_Q\}$
   $Min = L_{W_Q^1}(\theta)$
   $k^* = 1$
   complexité : 3

6. pour k = 2 à $CARD(W_Q)$ faire
   si ($Min > L_{W_Q^k}(\theta)$) alors
   $Min = L_{W_Q^k}(\theta)$
   $k^* = k$
   complexité : $\sum_{k=2}^{n} 3 = 3(n-1)$

7. pour $s = 1$ à Q faire
   $\mu'_{k^*+s} = \mu_{k^*+s} + (\theta - \sum_{i=0}^{s-1} \Delta_{k^*+i})^+$
   complexité : Les sommations sur les Δ_k ont déjà été effectuées lors des calculs hors-ligne, aussi, l'expression de la complexité va être : $\sum_{i=1}^{n} 4 = 4n$

La complexité des calculs en ligne va donc être en $O(n)$.

3.3.6 Optimalité

Le résultat suivant indique que l'algorithme général (voir 3.3.4) proposé est optimal.

Résultat 3

La solution obtenue en appliquant l'algorithme général est optimale, si Q est le plus grand entier tel que $\mu_Q + t_Q + \theta \leq D$.

Preuve du résultat 3

La preuve est immédiate si l'on considère les remarques suivantes :

1. Supposons d'abord qu'il soit optimal de commencer la tâche A sur $[\mu_k + t_k, \mu_{k+1}]$. Alors l'instant de départ qui conduit à une augmentation minimale du critère C_n est $\mu_k + t_k$. En effet, on observe que démarrer A après $\mu_k + t_k$ (et avant μ_{k+1}) augmenterait la valeur des instants de début des tâches déjà repoussées lorsque A démarre au temps $\mu_k + t_k$, et amènerait même à repousser d'autres tâches, ce qui augmenterait la valeur du critère, ou, dans le meilleur des cas, la laisserait inchangée.
2. Remarque similaire, si nous décidons de démarrer A sur $[\mu_{k+1}, \mu_{k+1} + t_{k+1}]$: dans ce cas nous avons à retarder a_{k+1} d'une période plus importante que si A démarrait au temps $\mu_k + t_k$.

En conséquence, la solution optimale est obtenue en démarrant A quand une des tâches se termine. Et compte tenu du fait que nous testons tous les instants de départ possibles, la solution est optimale. Ceci termine la démonstration. □

3.4 Un exemple

Dans cette section, nous présentons un exemple illustratif. Nous considérons un ensemble de tâches dont les caractéristiques (instants de départ, durée, instants maximaux d'exécution) sont définies dans le tableau 3.1.

Nous avons exécuté la partie hors-ligne de l'algorithme général pour les cinquante tâches constituant l'ordonnancement initial. Puis, nous avons considéré l'insertion d'une tâche aléatoire A définie par (D=600, $\theta = 20$). La partie en temps-réel de l'algorithme commence alors. Pour satisfaire la contrainte de délai, seules les fonctions $L_k(\theta)$ pour $k = 1, \ldots, 30$ seront prises en compte. On effectue le calcul nous donnant la fonction $z_A(\theta)$ correspondant à l'accroîssement minimal du critère. Le lecteur remarquera que bien que les fonctions $L_k(\theta)$ soient continues et linéaires par morceaux, ce n'est pas le cas de la fonction $z_A(\theta)$. L'explication est la suivante : le domaine de définition de chaque fonction $L_k(\theta)$ n'est pas le même. Ce domaine de définition est fonction du délai D de la tâche aléatoire. Aussi, bien que le calcul initial d'une fonction $L_k(\theta)$ soit effectué pour $\theta \in [0, \infty)$, seul le domaine $\theta \in [0, D - (\mu_k + t_k)[$ sera pris en considération pendant les calculs en temps-réel. La fonction $z_A(\theta)$ est

TAB. 3.1 – Tâches ordonnancées.

Tâche k	Instant de début	Durée	Délai	Δ_k	Tâche k	Instant de début	Durée	Délai	Δ_k
1	0	13	3	18	26	499	15	517	19
2	31	20	47	0	27	533	7	541	0
3	51	2	43	0	28	540	20	561	0
4	53	20	71	15	29	560	13	566	13
5	88	3	92	16	30	586	15	599	19
6	107	15	120	19	31	620	18	636	7
7	141	2	153	0	32	645	14	654	10
8	143	20	153	1	33	669	7	686	10
9	164	20	191	9	34	686	18	708	0
10	193	13	199	0	35	704	15	729	0
11	206	15	231	0	36	719	20	733	0
12	221	15	233	19	37	739	13	749	19
13	255	8	261	0	38	771	8	777	0
14	263	20	277	0	39	779	3	789	0
15	283	7	294	10	40	782	6	781	7
16	300	9	304	5	41	795	9	814	17
17	314	8	332	19	42	821	8	822	0
18	341	2	333	3	43	829	20	849	10
19	346	20	373	0	44	859	8	867	1
20	366	20	396	0	45	868	12	870	18
21	386	15	391	19	46	898	7	912	0
22	420	8	418	10	47	905	15	930	10
23	438	19	458	0	48	930	20	943	0
24	457	13	470	0	49	950	17	962	0
25	470	13	478	16	50	967	8	965	19

donc une fonction restant continue et linéaire par morceaux, mais qui ne sera en général pas concave.

La figure 3.10 représente la fonction $z_A(\theta)$ pour l'exemple (Notons que, dans la réalité, une seule valeur de la fonction doit être calculée). Sous l'axe des abscisses, nous avons indiqué quelle est la tâchea après laquelle il faut insérer suivant la valeur de θ.

Ici, la durée de la tâche aléatoire A considérée est de $\theta = 20$. L'insertion optimale se fait donc après la tâche 5 et le surcoût engendré est alors de 1.

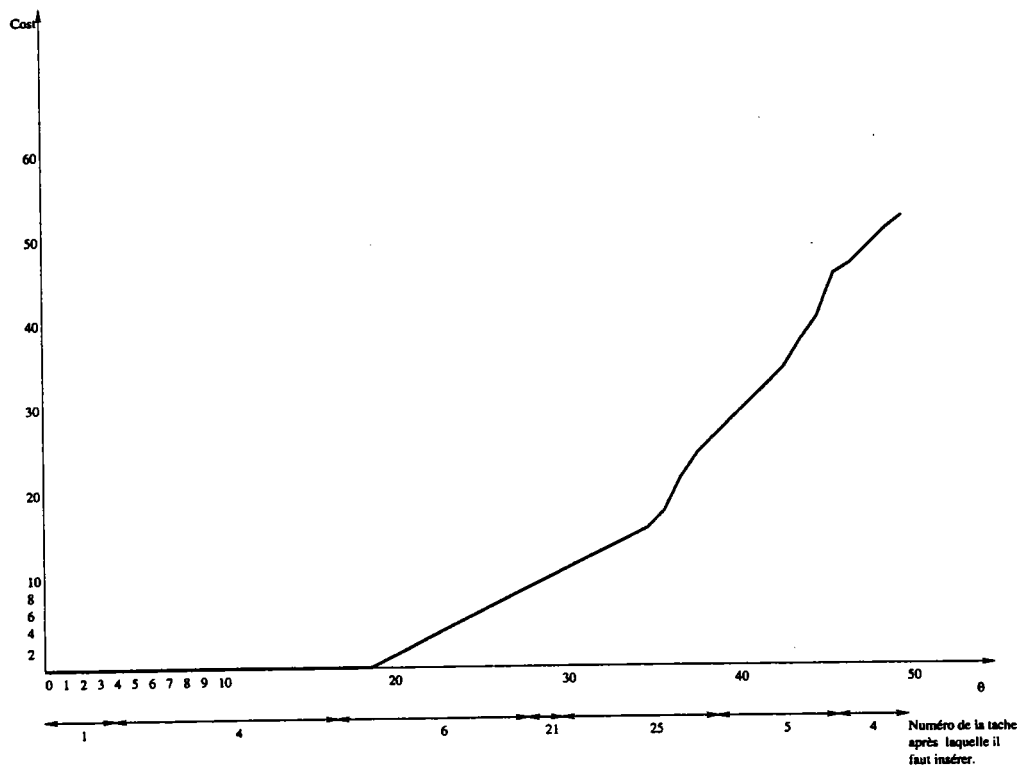


FIG. 3.10 – La fonction $z_A(\theta)$ pour l'exemple du paragraphe 3.4.

3.5 Conclusion

L'objectif de ce chapitre était de produire un algorithme permettant d'intercaler une tâche dans un ordonnancement existant, sous la contrainte "temps réel", et en minimisant le coût supplémentaire dû à l'insertion de cette tâche. C'est ce que nous avons fait en proposant l'algorithme général. Cet algorithme est optimal, et, de plus il est de complexité comparable aux heuristiques proposées dans le chapitre précédent. Cependant, la complexité de la partie des

calculs effectués en différé est supérieure.

L'objectif de la partie suivante de cette thèse est de produire un algorithme permettant d'insérer en temps réel dans les mêmes conditions de minimisation du coût supplémentaire un couple de tâches séparées par un temps fixe donné.

Résumé

Nous avons considéré le cas d'un ensemble de tâches définies par leur durée et par leur délai d'exécution. Elle sont exécutées de façon non préemptive sur une seule ressource et ordonnancées de façon à minimiser la somme des retards.
Une tâche apparaît dans le système à l'instant 0. Sa durée est aléatoire. Elle possède aussi une date limite d'exécution aléatoire, qui est inviolable. Notre but est d'insérer cette tâche aléatoire dans l'ordonnancement considéré, en faisant en sorte d'augmenter aussi peu que possible le critère de l'ordonnancement initial, qui est, dans notre cas la somme des retards. Cet algorithme doit permettre l'insertion de la tâche en temps réel. Nous proposons un algorithme en deux parties :

- Une partie en différé (à ce stade, on ne connait pas les paramètres de la tâche à insérer), calculant une fonction accroîssement du critère $L_k(\theta)$ pour l'insertion d'une tâche de longueur θ après la k^{ieme} tâche de l'ordonnancement initial. (voir section 3.3.4)
- Une partie en temps-réel (à ce stade, on connait les paramètres de la tâche à insérer : sa durée θ^* et son délai D), sélectionnant parmi les tâches de l'ordonnancement initial l'ensemble S de celles qui permettent de d'insérer la nouvelle tâche de façon à ce qu'elle finisse son exécution avant son délai D. Enfin, La tâche l après laquelle on va insérer la tâche aléatoire sera telle que , $L_l(\theta^*) = Min(L_k(\theta^*))$, $k \in S$. (voir sections 3.3.4 et 3.4)

Deuxième partie

Insertion d'une bitâche dans un ordonnancement

Présentation de la seconde partie

Dans la première partie de cette thèse, nous avons étudié l'insertion d'une tâche simple dans un ordonnancement de tâches simples, en spécifiant que ce problème était issu de la gestion des radars de combat. En fait, bien que pour des raisons de complexité ce soit ce premier modèle qui prévale actuellement, les tâches exécutées sur les radars sont de nature plus complexe.

Le schéma de base d'une requête exécutée par un radar est en effet composé d'une séquence d'émission d'ondes, suivie d'un temps d'inactivité fixé, et enfin d'une séquence de réception. Le temps d'exécution relatif à ces trois étapes est déterminé par la distance entre le radar et l'endroit que l'on désire sonder. Ces informations sont détaillées dans le chapitre 7. Techniquement, rien ne nous empêche d'exploiter le temps d'inactivité entre la phase d'émission et la phase de réception d'une même requête afin d'exécuter totalement ou partiellement une ou plusieurs autres requêtes. C'est pourquoi, dans la seconde partie de cette thèse, nous nous intéressons à l'insertion de tâches ayant une structure particulière, appelées bitâches (une bitâche est constituée de deux tâches simples non-préemptives séparées par une période de disponibilité), dans un ordonnancement que nous considèrerons constitué dans un premier temps de tâches simples, et dans un second temps de bitâches.

Comme dans la première partie de cette thèse nous cherchons à insérer une bitâche, dont les paramètres ne sont connus qu'au moment de son apparition, dans un ordonnancement préexistant. Cette nouvelle bitâche a un délai impératif. Nous considérons que les (bi)tâches constituant cet ordonnancement peuvent être retardées, mais qu'en aucun cas leur ordre d'exécution ne peut être altéré. Le lecteur aura remarqué qu'à cause du délai impératif, l'insertion de la nouvelle bitâche doit être effectuée en temps réel.

Bien entendu, les heuristiques proposées dans la suite sont décomposés en

deux parties : une partie en différé préparant l'insertion d'une bitâche aléatoire dont on ignore les paramètres, et une partie en temps réel, permettant l'insertion de la bitâche aléatoire lorsque ses paramètres sont connus.

Le chapitre 4 débute par l'étude des propriétés de la fonction *augmentation du critère* en fonction des trois paramètres suivants :

- Le temps d'exécution de la première tâche simple (correspondant au temps d'émission).
- La période d'attente(correspondant à la durée du voyage aller-retour entre le radar et la cible supposée).
- Le temps d'exécution de la seconde tâche simple (correspondant au temps de réception)

Le chapitre 5 propose trois algorithmes heuristiques permettant l'insertion d'une bitâche aléatoire dans un ordonnancement de tâches simples. Ces trois approches sont comparées à un algorithme fournissant un résultat optimal.
Le chapitre 6 est dédié à l'insertion d'une bitâche aléatoire dans un ordonnancement préexistant de bitâches, et propose un algorithme temps-réel pour parvenir à ce but. Cet algorithme est optimisé au niveau de son temps d'exécution en faisant appel à des techniques de calcul parallèle.

Chapitre 4

Analyse du critère dans le cas d'une bitâche

4.1 Introduction

Dans ce chapitre, nous ne considérons plus l'insertion d'une tâche simple dans un ordonnancement préétabli de tâches simples, mais celui d'une bitâche. L'objectif est d'augmenter aussi peu que possible le critère lié à cet ordonnancement et qui est l'augmentation de la somme des retards. Dans la continuité du chapitre 3, nous allons faire l'analyse de l'augmentation du critère afin de déterminer quelles sont les meilleures insertions possibles pour la bitâche aléatoire. Il est clair que, dans ce cas, le problème se complique considérablement du fait que la première sous-tâche peut retarder des tâches qui seront à nouveau retardées par l'introduction de la seconde sous-tâche.

La section 4.2 de ce chapitre donne la définition du problème. Dans la section 4.3 nous démontrons quelques résultats de base. La section 4.4 propose un algorithme permettant le calcul du critère (cet algorithme n'est pas utilisable en temps réel) et se termine par des exemples illustratifs. La section 4.5 est dédiée à l'analyse de l'augmentation du critère. La section 4.6 exploite les résultats de la section précédente pour proposer un algorithme temps-réel. Une analyse de la complexité est fournie 4.7. La section 4.8 contient une évaluation de l'algorithme temps-réel. La section 4.9 est la conclusion et est suivie d'un résumé du chapitre.

4.2 Formulation du problème

Une bitâche est composée de deux sous-tâches A_1 et A_2, séparées par un temps disponible de durée L. Une bitâche est représentée sur la figure 4.1.

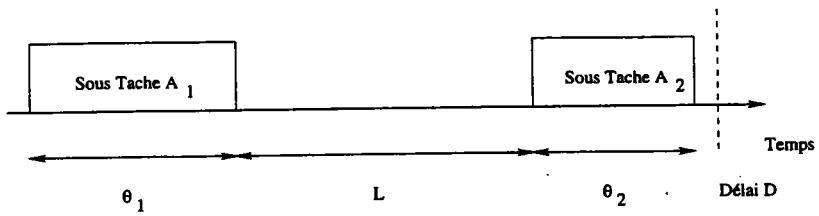


FIG. 4.1 – Représentation d'une bitâche A.

Les notations suivantes sont utilisées :

n : Nombre de tâches simples déjà ordonnancées et numérotées de 1 à n suivant l'ordre croissant de leur instant de départ.

μ_i : Instant de départ initial de la tâche i

μ_i' : Instant de départ de la tâche i après insertion de la bitâche

t_i : Temps d'exécution de la tâche i

d_i : Délai de la tâche i

θ_1 : Temps d'exécution de la sous-tâche 1

θ_2 : Temps d'exécution de la sous-tâche 2

L : Temps disponible entre les sous-tâches 1 et 2

D : Délai de la bitâche. Ce délai ne peut être violé.

Tout comme dans la première partie de la thèse, la ressource où les tâches sont exécutées est unique et les tâches sont non préemptives.

L'expression de l'augmentation du critère après l'insertion de la bitâche est donnée dans la section 2.2.

La fin de la sous-tâche A_1, de durée θ_1, est séparée du début de la sous-tâche A_2, de durée θ_2, par une période de longueur L. Cette période peut être utilisée pour exécuter certaines des tâches déjà ordonnancées. Enfin, le délai de la bitâche aléatoire, noté D, est connu et ne peut être violé.

Pour le moment, nous ferons l'hypothèse simplificatrice suivante : le début de la sous-tâche A_1 sera systématiquement choisi à la fin d'une des sous-tâches déjà ordonnancées.

Notons que, dans le cas d'une bitâche, cette contrainte n'est pas nécessairement satisfaite par la solution optimale, contrairement à ce qui se passait dans le cas d'une tâche simple.

4.3 Résultats de base

Nous considèrerons les trois cas suivants, quelle que soit la tâche a_k à la suite de laquelle nous plaçons la tâche aléatoire :

- Cas 1 : La somme des durées des tâches déplacées à la suite de l'introduction de la sous-tâche A_1 est inférieure à L, et au moins une tâche non déplacée se situe dans la période de longueur L après placement de A_1.
- Cas 2 : La somme des durées des tâches déplacées à la suite de l'introduction de la sous-tâche A_1 est inférieure ou égale à L, et aucune des tâches non déplacées ne peut s'intercaler dans la période de longueur L.
- Cas 3 : La somme des durées des tâches déplacées est supérieure à L.

Le nombre de tâches retardées du fait de l'introduction de la sous-tâche A_1 de durée θ_1 après la tâche a_k est l'entier n_k^1 tel que :

$$\sum_{i=0}^{n_k^1-1} \Delta_{k+i} < \theta_1 \leq \sum_{i=0}^{n_k^1} \Delta_{k+i} \tag{4.1}$$

Les tâches déplacées sont les tâches $a_{k+1}, \ldots, a_{k+n_k^1}$. Notons que si $\Delta_k \geq \theta_1$, alors A_1 peut être placée après a_k sans perturber l'ordonnancement.

4.3.1 Premier cas

Les hypothèses sont : $\sum_{i=1}^{n_k^1} t_{k+i} < L$ et au moins la tâche $a_{k+n_k^1+1}$ peut être placée entre A_1 et A_2.

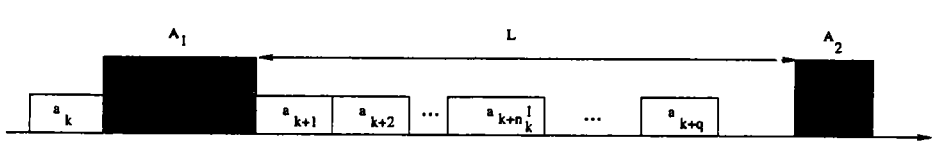


FIG. 4.2 – Exemple d'un couple de tâches A_1 et A_2 dans le cas 1

Nous considérons le plus grand entier q tel que $\mu_{k+q}+t_{k+q} \leq \mu_k+t_k+\theta_1+L$. La tâche a_{k+q} est donc la dernière tâche qui se situera entre les sous-tâches A_1

et A_2 si nous plaçons A_1 après a_k. De plus, a_{k+q} ne sera pas retardée.
L'augmentation de coût due au déplacement des tâches $a_{k+1}, \ldots, a_{k+n_k^1}$ est :

$$C_1^k = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta_1 \\ \sum_{s=1}^{n_k^1} (\theta_1 - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ & \text{si} \quad \Delta_k < \theta_1 \end{cases} \tag{4.2}$$

Une explication de cette formule a été donnée dans les chapitres 2 et 3.
Pour la deuxième sous-tâche, il est équivalent de dire que nous intégrons, après a_{k+q}, une tâche de durée :

$$\theta_2^* = \mu_k + t_k + \theta_1 + L + \theta_2 - (\mu_{k+q} + t_{k+q}) \tag{4.3}$$

Le coût résultant de l'intégration de la dernière sous-tâche est :

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_{k+q} \geq \theta_2^* \\ \sum_{s=1}^{n_k^2} (\theta_2^* - \sum_{p=0}^{s-1} \Delta_{k+q+p} - f_{k+q+s})^+ & \text{si} \quad \Delta_{k+q} < \theta_2^* \end{cases} \tag{4.4}$$

où n_k^2 est l'entier tel que :

$$\sum_{i=0}^{n_k^2-1} \Delta_{k+q+i} < \theta_2^* \leq \sum_{i=0}^{n_k^2} \Delta_{k+q+i} \tag{4.5}$$

Cette formule est identique à (4.1) La formule est identique à la précédente, seule la durée de la tâche A_2 a été modifiée. L'entier n_k^2 est le nombre de tâches situées après la tâche a_{k+q} et qui seront retardées du fait de l'introduction, après a_{k+q}, de la tâche virtuelle de durée θ_2^*. C'est, plus simplement, le nombre de tâches situées après la fin de tâche aléatoire qui sont perturbées par l'introduction de cette tâche.

4.3.2 Deuxième cas

Les hypothèses sont sensiblement différentes du cas précédent : $\sum_{i=1}^{n_k^1} t_{k+i} \leq L$ et la tâche $a_{k+n_k^1+1}$ ne peut être placée entre A_1 et A_2.
En d'autres termes, la fin de la tâche $a_{k+n_k^1+1}$ se situe après le début de la sous-tâche A_2, ce qui s'écrit :

$$\mu_{k+n_k^1+1} + t_{k+n_k^1+1} > \mu_k + t_k + \theta_1 + L$$

La figure 4.3. présente le cas où le début de la tâche $a_{k+n_k^1+1}$ est postérieur à la fin de la sous-tâche A_2, soit : $\mu_{k+n_k^1+1} \geq \mu_k + t_k + \theta_1 + L + \theta_2$. Il se pourrait également que cette condition ne soit pas réalisée. Dans ce cas, l'instant de début de $a_{k+n_k^1+1}$ serait retardé jusqu'à coïncider avec l'instant de fin de A_2.

Le coût C_1^k reste valable (voir la relation (4.2)).

En notant $\Delta_k^* = \mu_{k+n_k^1+1} - (\mu_k + t_k)$, C_2^k se calcule comme si :

- nous plaçions une tâche de durée $\theta_2^{**} = \theta_1 + L + \theta_2$ après a_k.
- les tâches $a_{k+1}, \ldots, a_{k+n_k^1}$ n'existaient pas.

Alors, toujours avec la même approche :

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_k^* \geq \theta_2^{**} \\ \sum_{s=1}^{n_k^3} (\theta_2^{**} - \Delta_k^* - \sum_{p=1}^{s-1} \Delta_{k+n_k^1+p} - f_{k+n_k^1+s})^+ & \text{si} \quad \Delta_k^* < \theta_2^{**} \end{cases} \tag{4.6}$$

où n_k^3 est l'entier défini par :

$$\Delta_k^* + \sum_{i=1}^{n_k^3-1} \Delta_{k+n_k^1+i} < \theta_2^{**} \leq \Delta_k^* + \sum_{i=1}^{n_k^3} \Delta_{k+n_k^1+i} \tag{4.7}$$

La suppression virtuelle des tâches $a_{k+1}, \ldots, a_{k+n_k^1}$ conduit à considérer que la période Δ_k^* entre la fin de a_k et le début de $a_{k+n_k^1+1}$ est la période libre qui suit la tâche a_k. Comme dans le cas précédent, les tâches retardées par A_1 ne sont pas retardées par A_2, mais il y a continuité entre les tâches retardées par A_1 et les tâches retardées par A_2. Nous remarquons que les mêmes formules s'appliquent encore, moyennant une correction de la durée de la tâche intercalée et une correction de la période libre qui suit a_k.

4.3.3 Troisième cas

A présent, les hypothèses sont les suivantes : $\sum_{i=1}^{n_k^1} t_{k+i} > L$.

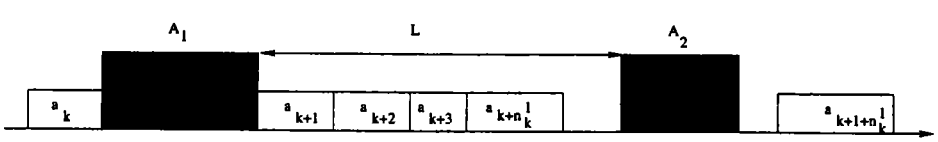


FIG. 4.3 – Exemple d'un couple de tâches A_1 et A_2 dans le cas 2

Nous sommes dans le cas où la fenêtre de temps située entre la fin de la sous-tâche A_1 et le début de la sous-tâche A_2 n'est pas en mesure de contenir l'ensemble des tâches retardées du fait de l'introduction de A_1. Notons que, dans ce cas de figure :

- Pour toutes les tâches qui peuvent être situées entre la fin de A_1 et le début de A_2, une tâche débute dès que la précédente se termine.
- Il se peut que l'instant de fin de la dernière de ces tâches soit strictement inférieur à l'instant de début de A_2

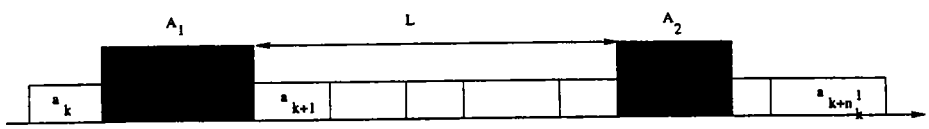


FIG. 4.4 – Exemple d'un couple de tâches A_1 et A_2 dans le cas 3

Soit r_k^1 tel que $\sum_{i=1}^{r_k^1} t_{k+i} \leq L < \sum_{i=1}^{r_k^1+1} t_{k+i}$.

L'entier r_k^1 est le nombre de tâches déplacées par l'introduction de A_1 et qui vont être placées avant A_2. Nous sommes dans le cas où certaines tâches, les tâches $a_{k+r_k^1+1}$ à $a_{k+n_k^1}$, sont retardées successivement par A_1 et A_2.
L'augmentation du critère dûe au seul effet de l'introduction de A_1 est :

$$C_1^k = \sum_{s=1}^{r_k^1} (\theta_1 - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ \tag{4.8}$$

En notant $\Delta_k^{**} = \mu_{k+r_k^1+1} - (\mu_k + t_k)$, C_2^k se calcule comme si :

- nous plaçions une tâche de durée $\theta_2^{**} = \theta_1 + L + \theta_2$ après a_k.
- les tâches $a_{k+1}, \ldots, a_{k+r_k^1}$ n'existaient pas.

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_k^{**} \geq \theta_2^{**} \\ \sum_{s=1}^{n_k^4} (\theta_2^{**} - \Delta_k^{**} - \sum_{p=1}^{s-1} \Delta_{k+r_k^1+p} - f_{k+r_k^1+s})^+ & \text{si} \quad \Delta_k^{**} < \theta_2^{***} \end{cases} \tag{4.9}$$

où n_k^4 est l'entier tel que :

$$\Delta_k^{**} + \sum_{i=1}^{n_k^4-1} \Delta_{k+r_k^1+i} < \theta_2^{**} \leq \Delta_k^{**} + \sum_{i=1}^{n_k^4} \Delta_{k+r_k^1+i} \tag{4.10}$$

4.4 Algorithme de calcul du critère

Les résultats précédents sont résumés dans l'algorithme de calcul d'augmentation du critère, donné figure 4.5. Nous supposons que nous plaçons la tâche aléatoire après a_k.

4.4.1 Exemples

Exemple 1

Considérons l'ordonnancement des tâches a_i données par le tableau 4.1 :

TAB. 4.1 – Données de l'ordonnancement 1

tâches i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
t_i	2	3	1	4	5	3	2	4
d_i	3	8	14	12	23	27	28	25
μ_i	0	5	9	10	15	21	27	29

De ces données, nous tirons les informations consignées dans le tableau 4.2 :

TAB. 4.2 – Autres données de l'ordonnancement 1.

tâches i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
Δ_i	3	1	0	1	1	3	0	∞
f_i	1	0	4	0	3	3	0	0

La tâche aléatoire qui apparait à l'instant 0 est caractérisée par :

$$\theta_1 = 1 \quad L = 18 \quad \theta_2 = 2$$

Nous souhaitons calculer l'augmentation du critère si nous plaçions cette tâche après a_2. Comme $\Delta_2 = \theta_1 = 1$, nous pouvons introduire la tâche aléatoire entre a_2 et a_3 sans perturber cette dernière. (voir figure 4.6). Nous sommes donc dans le **cas1** de l'algorithme avec $n_2^1 = 0$. Déjà, $C_1^2 = 0$. Nous calculons ensuite l'entier q tel que :

$$\begin{aligned} \mu_{2+q} + t_{2+q} &\leq \mu_2 + t_2 + \theta_1 + L \\ \mu_{2+q} + t_{2+q} &\leq 27 \end{aligned}$$

1. Calcul de n_k^1 à l'aide de (4.1))
2. Cas 1 : si $\sum_{i=1}^{n_k^1} t_{k+i} < L$ et $a_{k+n_k^1+1} + t_{k+n_k^1+1} \leq a_k + t_k + \theta_1 + L$, faire :
   (a) Calcul de C_1^k à l'aide de (4.2))
   (b) Calcul du plus grand entier q tel que :
   $\mu_{k+q} + t_{k+q} \leq \mu_k + t_k + \theta_1 + L$
   (c) Calcul de θ_2^* à l'aide de (4.3))
   (d) Calcul de n_k^2 à l'aide de (4.5)
   (e) Calcul de C_2^k à l'aide de (4.4)
   (f) Calcul de $C_k^* = C_1^k + C_2^k$. FIN
3. Cas 2 : si $\sum_{i=1}^{n_k^1} t_{k+i} < L$ et $a_{k+n_k^1+1} + t_{k+n_k^1+1} > a_k + t_k + \theta_1 + L$, faire :
   (a) Calcul de C_1^k à l'aide de (4.2)
   (b) Calcul de $\Delta_k^* = \mu_{k+n_k^1+1} - (\mu_k + t_k)$
   (c) Calcul de $\theta_2^{**} = \theta_1 + L + \theta_2$
   (d) Calcul de n_k^3 à l'aide de (4.7)
   (e) Calcul de C_2^k à l'aide de (4.6)
   (f) Calcul de $C_k^* = C_1^k + C_2^k$. FIN
4. Cas 3 : si $\sum_{i=1}^{n_k^1} t_{k+i} > L$
   (a) Calcul de r_k^1 tel que :
   $\sum_{i=1}^{r_k^1} t_{k+i} \leq L < \sum_{i=1}^{r_k^1+1} t_{k+i}$
   (b) Calcul de C_1^k à l'aide de (4.8)
   (c) Calcul de $\Delta_k^{**} = \mu_{k+r_k^1+1} - (\mu_k + t_k)$
   (d) Calcul de $\theta_2^{**} = \theta_1 + L + \theta_2$
   (e) Calcul de n_k^4 à l'aide de (4.10)
   (f) Calcul de C_2^k à l'aide de (4.9)
   (g) Calcul de $C_k^* = C_1^k + C_2^k$. FIN

FIG. 4.5 – *Algorithme de calcul d'augmentation du critère*

Nous trouvons $q = 4$.

Nous intégrons donc après a_6 une tâche fictive de durée :

$$\begin{aligned} \theta_2^* &= \mu_2 + t_2 + \theta_1 + L + \theta_2 - (\mu_6 + t_6) \\ &= 5 \end{aligned}$$

L'entier n_2^2 est tel que :

$$\sum_{i=0}^{n_2^2-1} \Delta_{6+i} < 5 \leq \sum_{i=0}^{n_2^2} \Delta_{6+i}$$

Nous trouvons $n_2^2 = 2$.

Nous calculons alors :

$$\begin{aligned} C_2^2 &= \sum_{s=1}^{2} (5 - \sum_{p=0}^{s-1} \Delta_{6+p} - f_{6+s})^+ \\ &= (5-3)^+ + (5-3)^+ \\ &= 4 \end{aligned}$$

Finalement : $C_2^* = 4$

Exemple 2

A présent nous allons étudier l'ordonnancement dont les caractéristiques sont données dans le tableau 4.3

TAB. 4.3 – Données de l'ordonnancement 2

tâches i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
t_i	1	2	3	2	4	2	1	3	3	3
d_i	2	5	8	14	20	22	26	30	35	42
μ_i	0	2	6	10	15	21	24	28	33	38

A nouveau, nous utilisons le tableau 4.3 pour obtenir les informations données dans le tableau 4.4.

Pour cet exemple, la tâche aléatoire qui apparait à l'instant 0 a les caractéristiques suivantes :

$$\theta_1 = 4 \quad L = 9 \quad \theta_2 = 6$$

TAB. 4.4 – Autres données de l'ordonnancement 2

tâches i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
Δ_i	1	2	1	3	2	1	3	2	2	∞
f_i	1	1	0	2	1	0	1	0	0	1

Comme dans l'exemple précédent, nous cherchons à calculer l'augmentation du coût lorsque nous introduisons la tâche aléatoire après a_2. Pour cela, il faut calculer n_2^1 en utilisant la relation (4.1). Nous trouvons $n_2^1 = 2$: deux tâches seront donc perturbées par l'insertion de A_1. (voir figure 4.7). Le calcul donne aussi :

$$\sum_{i=1}^{2} t_{2+i} = 5$$

Comme en plus la tâche a_5 ne peut être insérée entre A_1 et A_2, nous sommes donc dans les hypothèses du **cas 2** de l'algorithme. Le coût supplémentaire due à l'introduction de A_1 est donc :

$$\begin{aligned} C_1^2 &= \sum_{s=1}^{2}(4 - \sum_{p=0}^{s-1} \Delta_{2+p} - f_{2+p})^+ \\ &= (4-2-0)^+ + (4-2-1-2)^+ \\ &= 2 \end{aligned}$$

Pour calculer le second coût supplémentaire, nous devons calculer Δ_2^*, temps libre après la tâche a_2, et θ_2^{**}, durée de la tâche fictive. Les calculs donnent respectivement :

$$\begin{aligned} \Delta_2^* &= \mu_5 + t_5 - (\mu_2 + t_2) \\ &= 11 \\ \theta_2^{**} &= 19 \end{aligned}$$

Nous devons également déterminer l'entier n_2^3 tel que :

$$11 + \sum_{i=1}^{n_2^3-1} \Delta_{4+i} < 19 \leq 11 + \sum_{i=1}^{n_2^3} \Delta_{4+i}$$

nous trouvons dans le cas présent $n_2^3 = 4$. Il y aura donc quatre tâches qui seront perturbées par A_2. Nous pouvons maintenant calculer le second coût supplémentaire :

$$\begin{aligned} C_2^2 &= \sum_{s=1}^{4}(19 - 11 - \sum_{p=1}^{s-1}\Delta_{4+p} - f_{4-p})^+ \\ &= (8-1)^+ + (8-2-0)^+ + (8-2-1-1)^+ + (8-2-1-3-0)^+ \\ &= 19 \end{aligned}$$

Finalement, en additionnant les deux coûts, nous trouvons $C_2^* = 21$.

Exemple 3

Pour ce dernier exemple, nous conservons les données de l'ordonnancement précédent. La seule modification apportée concerne les caractéristiques de la tâche aléatoire. Les nouvelles données de cette tâche sont dorénavant :

$$\theta_1 = 8 \quad L = 5 \quad \theta_2 = 6$$

Le problème est toujours le même, à savoir calculer le coût supplémentaire lié à l'introduction de la tâche aléatoire après a_2. (voir figure 4.8)
Nous commençons comme précédemment par calculer le nombre n_2^1 défini par la relation (4.1). Dans cet exemple :

$$\Delta_2 + \Delta_3 + \Delta_4 < \theta_1 \leq \Delta_2 + \Delta_3 + \Delta_4 + \Delta_5$$

D'où $n_2^1 = 3$. En calculant $\sum_{i=1}^{3} t_{2+i}$, nous trouvons une valeur de 9, ce qui est strictement supérieur à la période de longueur L située entre les tâches A_1 et A_2. Nous nous situons donc dans le cadre du **cas 3** de l'algorithme.
Nous devons tout d'abord calculer le nombre r_2^1 de tâches déplacées suite à l'introduction de A_1. L'entier r_2^1 est tel que :

$$\sum_{i=1}^{r_2^1} t_{2+i} \leq 8 < \sum_{i=1}^{r_2^1+1} t_{2+i}$$

Ce qui nous donne $r_2^1 = 2$.
Nous pouvons donc en déduire le coût supplémentaire C_1^2 en utilisant la formule

(4.8) :

$$\begin{aligned} C_1^2 &= \sum_{s=1}^{2}(8 - \sum_{p=0}^{s-1}\Delta_{2+p} - f_{2+s})^+ \\ &= (8-2-0)^+ + (8-2-1-2)^+ \end{aligned}$$

Soit $C_1^2 = 9$.

Pour calculer le second coût, nous devons au préalable déterminer certaines données comme Δ_2^{**}, θ_2^{**} et n_2^4. Nous trouvons respectivement :

$$\begin{aligned} \Delta_2^{**} &= \mu_5 - (\mu_2 + t_2) \\ &= 11 \\ \theta_2^{**} &= \theta_1 + L + \theta_2 \\ &= 19 \\ n_2^4 &= 4 \end{aligned}$$

L'entier n_2^4 a été calculé en utilisant la relation (4.10), le calcul étant similaire à ceux de n_2^2 et n_2^3 et qui ont été menés dans les exemples précédents. La seule différence concerne la borne supérieure de la somme puisque, dans le cas présent, cette borne dépend de r_2^1 et non de n_2^1.
Le calcul du second coût C_2^2 est le suivant :

$$\begin{aligned} C_2^2 &= \sum_{s=1}^{4}(19 - 11 - \sum_{p=1}^{s-1}\Delta_{4+p} - f_{4+s})^+ \\ &= (8-1)^+ + (8-2-0)^+ + (8-2-1-1)^+ + (8-2-1-3-0)^+ \\ &= 7+6+4+2 \end{aligned}$$

Soit $C_2^2 = 19$.
Finalement, le coût supplémentaire total dû à l'introduction de la tâche aléatoire est de $C_2^* = 28$.

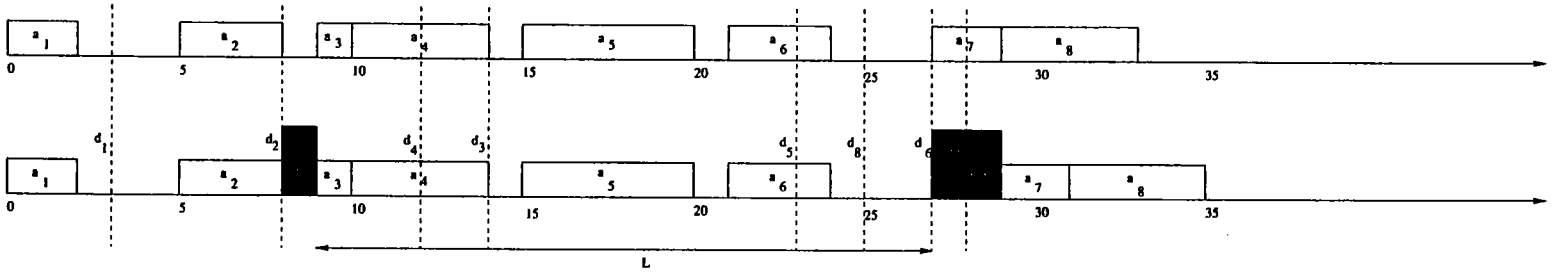


FIG. 4.6 – Exemple pour le cas 1.

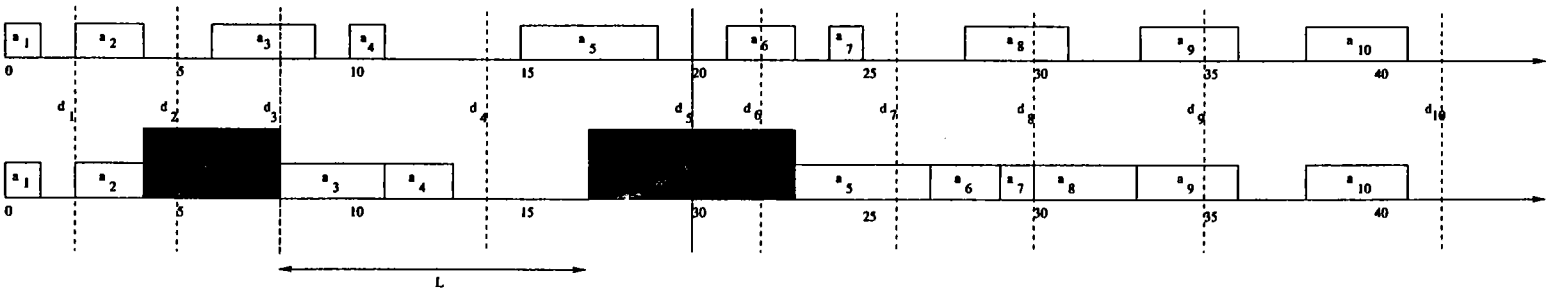


FIG. 4.7 – Exemple pour le cas 2.

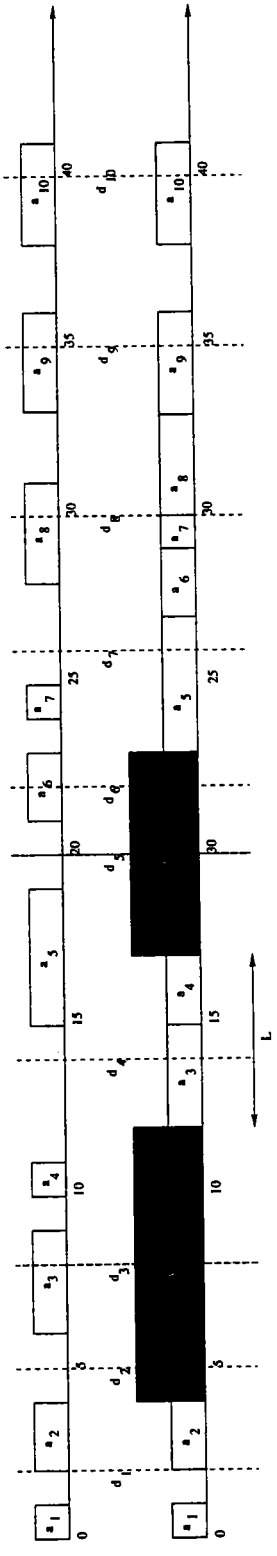


FIG. 4.8 – Exemple pour le cas 3.

4.5 Etude de la fonction "accroîssement du critère"

L'accroîssement du critère, lorsque nous plaçons la tâche aléatoire après la tâche a_k, est la somme de deux termes :

- C_1^k, accroîssement dû à la sous-tâche A_1.
- C_2^k, accroîssement dû à la sous-tâche A_2.

Les valeurs sont fonction de la position de la tâche aléatoire, c'est-à-dire de l'indice k, mais aussi de θ_1, durée de la sous-tâche A_1, de θ_2, durée de la sous-tâche A_2, et de L, période séparant la fin de la sous-tâche A_1 du début de la sous-tâche A_2. Nous noterons donc ces termes respectivement $C_1^k(\theta_1, L, \theta_2)$ et $C_2^k(\theta_1, L, \theta_2)$.

La fonction "accroîssement du critère" lorsque la tâche aléatoire débute lorsque a_k se termine s'écrira :

$$C^k(\theta_1, L, \theta_2) = C_1^k(\theta_1, L, \theta_2) + C_2^k(\theta_1, L, \theta_2)$$

C'est cette fonction que nous étudions dans la suite.

4.5.1 Variation de $C_1^k(\theta_1, L, \theta_2)$ lorsque θ_1 croît

Supposons que nous soyons dans le premier cas décrit dans le paragraphe 4.2 pour le triplet $(\theta_1^*, L^*, \theta_2^*)$.

Le résultat suivant rappelle le résultat 2 (voir section 3.2).

Dans le résultat 4, q est la première tâche qui, dans l'ordonnancement initial, se termine après que la seconde sous-tâche a débuté.

Résultat 4

Si $\sum_{i=1}^{q+1} t_{k+i} > L^*$, alors $C^k(\theta_1, L^*, \theta_2^*)$ est une fonction continue et linéaire par morceaux de θ_1 pour $\theta_1 \geq \theta_1^*$.

Preuve du résultat 4

En nous reportant aux relations (4.2), (4.3) et (4.4), et en tenant compte du fait que l'inégalité $\sum_{i=1}^{q+1} t_{k+i} > L^*$ nous permet de dire que seules les tâches

$a_{k+1}, \ldots, a_{k+q}$ peuvent être placées entre la fin de A_1 et le début de A_2 quelque soit $\theta_1 \geq \theta_1^*$, alors :

$$\mathcal{C}^k(\theta_1, L^*, \theta_2^*) = \sum_{s=1}^{q}(\theta_1 - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+$$

$$+\sum_{s=1}^{+\infty}(\mu_k + t_k + \theta_1 + L^* + \theta_2^* - \mu_{k+q} - t_{k+q} - \sum_{p=0}^{s-1}\Delta_{k+q+p} - f_{k+q+s})^+ \tag{4.11}$$

En posant : $z_{k,s} = \sum_{p=0}^{s-1}\Delta_{k+p} + f_{k+s}$ pour $s = 1, 2, \ldots, n_k^1$

et

$z_{k+q,s} = \mu_k - t_k - L^* - \theta_2^* + \mu_{k+q} + t_{k+q} + \sum_{p=0}^{s-1}\Delta_{k+q+p} - f_{k+q+s}$, pour $s = 1, 2, \ldots$

L'accroîssement de critère s'écrit :

$\mathcal{C}^k(\theta_1, L^*, \theta_2^*) = \sum_{s=1}^{q}(\theta_1 - z_{k,s})^+ + \sum_{s=1}^{+\infty}(\theta_1 - z_{k+q,s})^+$

Notons $z_{k,u}^*, u = 1, 2, \ldots$ l'ensemble $\{z_{k,s}\}_{s=1,2,\ldots,n_k^1} \cup \{z_{k+q,s}\}_{s=1,2,\ldots}$ classés par ordre croissant. Alors :

$$\mathcal{C}^k(\theta_1, L_1^*, \theta_2^*) = \sum_{u=1}^{+\infty}(\theta_1 - z_{k,u}^*)^+$$

Cette fonction est égale à :

$$\mathcal{C}^k(\theta_1, L_1^*, \theta_2^*) = \begin{cases} 0 & \text{si} \quad \theta_1 < z_{k,1}^* \\ (u-1)\theta_1 - \sum_{i=1}^{u-1} z_{k,i}^* & \text{si} \quad \theta_1 \in [z_{k,u-1}^*, z_{k,u}^*[\end{cases} \tag{4.12}$$

Ceci termine la démonstration. □

Le résultat que nous présentons maintenant complète le précédent :

Résultat 5

Nous posons $w_i = \mu_{k+q+i} + t_{k+q+i} - (\mu_k + t_k + L)$.

Supposons que $\sum_{i=1}^{k+r} t_{k+i} \leq L^* < \sum_{i=1}^{k+r+1} t_{k+i}$ pour $r \geq 1$. Alors la fonction $\mathcal{C}^k(\theta_1, L^*, \theta_2^*)$ est :

1. discontinue pour $\theta_1 = w_i, i = 1, 2, \ldots, r$,
   avec $\mathcal{C}^k(w_i^-, L^*, \theta_2^*) \geq \mathcal{C}^k(w_i^+, L^*, \theta_2^*)$

2. continue et linéaire par morceaux sur chacun des intervalles suivant : $I_0 = [0, w_1)$,$I_i = [w_i, w_{i+1})$, $i = 1, 2, \ldots, r-1$ et $I_r = [w_r, +\infty)$.

Preuve du résultat 5

1. Nous montrons d'abord que $C^k(\theta_1, L^*, \theta_2^*)$ est une fonction discontinue de θ_1 aux points $w_i, i = 1, 2, \ldots, r$ et nous précisons la discontinuité.

   (a) Lorsque $\theta_1 = w_i^-$, $\theta_1 < \mu_{k+q+i} + t_{k+q+i} - (\mu_k + t_k + L^*)$, soit $\mu_k + t_k + \theta_1 + L^* < \mu_{k+q+i} + t_{k+q+i}$. Cette inégalité indique que, dans sa position initiale, la tâche a_{k+q+i} se terminerait après que la sous-tâche A_2 a débuté. Par conséquent, après introduction de la tâche aléatoire :

   - Les tâches $a_{k+1}, a_{k+2}, \ldots, a_{k+q+i-1}$ se situeront entre la fin de la sous-tâche A_1 et le début de la sous-tâche A_2.
   - Les tâches $a_{k+q+i}, a_{k+q+i+1}, \ldots$ se situeront après la fin de la sous-tâche A_2.

   L'augmentation du critère associée à cette situation s'écrit :

$$C^k(w_i^-, L^*, \theta_2^*) = \sum_{s=1}^{q+i-1} (w_i - \sum_{p=0}^{s-1} \Delta_{k+p} - f_{k+s})^+ +$$

$$\sum_{s=1}^{+\infty} (\mu_k + t_k + w_i + L^* + \theta_2^* - \mu_{k+q+i-1} - t_{k+q+i-1} - \sum_{p=0}^{s-1} \Delta_{k+q+i+p-1} - f_{k+q+i+s-1})^+$$

   (b) Considérons maintenant le cas où $\theta_1 = w_i$. Cette nouvelle situation conduit à $\mu_k + t_k + \theta_1 + L > \mu_{k+q+i} + t_{k+q+i}$. Dans ce cas, la tâche a_{k+q+i}, considérée dans sa position initiale, se termnerait avant que la sous-tâche A_2 débute.

   Par conséquent, après introduction de la tâche aléatoire :

   - Les tâches $a_{k+1}, a_{k+2}, \ldots, a_{k+q+i-1}, a_{k+q+i}$ se situeront entre la fin de la sous-tâche A_1 et le début de la sous-tâche A_2.
   - Les tâches $a_{k+q+i+1}, a_{k+q+i+2}, \ldots$ se situeront après la fin de la sous-tâche A_2.

L'augmentation de critère associée s'écrit :

$$C^k(w_i^+, L^*, \theta 2^*) = \sum_{s=1}^{q+i}(w_i - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+ +$$

$$\sum_{s=1}^{+\infty}(\mu_k + t_k + w_i + L^* + \theta_2^* - \mu_{k+q+i} - t_{k+q+i} - \sum_{p=0}^{s-1}\Delta_{k+q+i+p} - f_{k+q+i+s})^+$$

Nous allons montrer maintenant que $C^k(w_i^-, L^*, \theta_2^*) \geq C^k(w_i, L^*, \theta_2^*)$

i. La contribution de la tâche a_{k+q+i} à l'accroîssement de la valeur du critère est :

- Lorsque $\theta_1 = w_i^-$ c'est-à-dire lorsque a_{q+k+i} commence après que la sous-tâche A_2 s'est terminée, cet accroîssement est $\sigma^-(a_{k+q+i})$, avec
  $\sigma^-(a_{k+q+i}) = \mu_k + t_k + w_i + L^* + \theta_2^* - \mu_{k+q+i-1} - t_{k+q+i-1} - \Delta_{k+q+i-1} - f_{k+q+i}$.
- Lorsque $\theta_1 = w_i$, c'est-à-dire lorsque a_{k+q+i} peut se terminer avant que la sous-tâche A_2 ne débute, cet accroîssement est $\sigma^+(a_k + q + i)$, avec :
  $\sigma^+(a_{k+q+i}) = w_i - \sum_{p=0}^{q+i-1}\Delta_{k+p} - f_{k+q+i}$

  Alors :

$$\begin{aligned}\sigma^-(a_{k+q+i}) - \sigma^+(a_{k+q+i}) &= \mu_k + t_k + L^* + \theta_2^* - \mu_{k+q+i-1} - t_{k+q+i-1} \\ &\quad -\Delta_{k+q+i-1} + \sum_{p=0}^{q+i-1}\Delta_{k+p}\end{aligned}$$

  Mais :

$$\mu_{k+q+i-1} + t_{k+q+i-1} - (\mu_k + t_k) = \sum_{s=1}^{q+i-2}\Delta_{k+s} + \sum_{s=1}^{q+i-1} t_{k+s}$$

  Donc :

$$\begin{aligned}\sigma^-(a_{k+q+i}) - \sigma^+(a_{k+q+i}) &= L^* + \theta_2^* - \sum_{s=0}^{q+i-2}\Delta_{k+s} - \\ &\quad \sum_{k=1}^{q+i-1} t_{k+s} - \Delta_{k+q+i-1} + \sum_{s=0}^{q+i-1}\Delta_{k+p}\end{aligned}$$

$$= L^* + \theta_2^* - \sum_{k=1}^{q+i-1} t_{k+s}$$

Mais $\sum_{k=1}^{q+i-1} t_{k+s} \leq L^*$ pour $i = 1, 2, \ldots, r$

Finalement : $\sigma^-(a_{k+q+i}) - \sigma^+(a_{k+q+i}) \geq 0$

$$\text{et } \sigma^-(a_{k+q+i}) \geq \sigma^+(a_{k+q+i}) \tag{4.13}$$

ii. Nous considérons maintenant la contribution d'une tâche $a_{k+q+j}, j > i$ à l'accroîssement de la valeur du critère.

- Lorsque $\theta_1 = w_i^-$ cette contribution est $\sigma^-(a_{k+q+j})$, avec :

$$\begin{aligned}\sigma^-(a_{k+q+j}) &= \mu_k + t_k + w_i + L^* + \theta_2^* - \mu_{k+q+i-1} - t_{k+q+i-1} \\ &\quad - \sum_{p=0}^{j-1} \Delta_{k+p+i-1+p} - f_{k+q+j}\end{aligned}$$

De manière analogue, lorsque $\theta_1 = w_i$, cette contribution est $\sigma^+(a_{k+q+j})$, avec :

$$\begin{aligned}\sigma^+(a_{k+q+j}) &= \mu_k + t_k + w_i + L^* + \theta_2^* - \mu_{k+q+i} - t_{k+q+i} \\ &\quad - \sum_{p=0}^{j-i-1} \Delta_{k+p+i+p} - f_{k+q+j}\end{aligned}$$

Des relations précédentes, nous déduisons :

$$\begin{aligned}\sigma^-(a_{k+q+j}) - \sigma^+(a_{k+q+j}) &= \mu_{k+q+i} + t_{k+q+i} - (\mu_{k+q+i-1} + t_{k+q+i-1}) \\ &\quad + \sum_{p=0}^{j-i-1} \Delta_{k+q+i+p} - \sum_{p=0}^{j-1} \Delta_{k+q+i-1+p} \\ &= mu_{k+q+i} + t_{k+q+i} - (\mu_{k+q+i-1} + t_{k+q+i-1}) - \Delta_{k+q+i-1} \\ &= \Delta_{k+q+i-1} + t_{k+q+i} - \Delta_{k+q+i-1} \\ &= t_{k+q+i} > 0\end{aligned}$$

Finalement : $\sigma^-(a_{k+q+j}) - \sigma^+(a_{k+q+j}) = t_{k+q+i} > 0$ pour $j = i+1, i+2, \ldots$

et :

$$\sigma^-(a_{k+q+j}) > \sigma^+(a_{k+q+j}) \tag{4.14}$$

Pour $j = i+1, i+2, \ldots$

En additionnant membre à membre la relation (4.13) aux relations (4.14), il vient :

$$C^k(w_i^-, L^*, \theta_2^*) \geq C^k(w_i, L^*, \theta_2^*) \tag{4.15}$$

2. Dans chacun des intervalles $I_i, i = 0, 1, \ldots, r$, nous nous retrouvons dans le cas du résultat 4., et donc $C^k(\theta_1, L^*, \theta_2^*)$ est une fonction continue et linéaire par morceaux.

Supposons que $\theta_1 \in I_i$ c'est-à-dire : $\mu_{k+q+i} + t_{k+q+i} - (\mu_q + t_k + L^*) \leq \theta_1 < \mu_{k+q+i+1} + t_{k+q+i+1} - (\mu_k + t_k + L^*)$ alors, les tâches situées entre les sous-tâches A_1 et A_2 sont : $a_{k+1}, a_{k+2}, \ldots, a_{k+q}, a_{k+q+1}, \ldots, a_{k+q+i}$, l'accroîssement du coût s'écrit alors :

$$C^k(\theta_1, L^*, \theta_2^*) = \sum_{s=1}^{q+i}(\theta_1 - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+ +$$

$$\sum_{s=1}^{+\infty}(\mu_k + t_k + \theta_1 + L^* + \theta_2^* - \mu_{k+q+i} - t_{k+q+i} - \sum_{p=0}^{s-1}\Delta_{k+q+i+p} - f_{k+q+i+s})^+ \tag{4.16}$$

Soit, en posant :

$z_{k,s} = \sum_{p=0}^{s-1}\Delta_{k+p} + f_{k+s}$ pour $s \in \{1, 2, \ldots, n_k^1\}$

et

$$z'_{k+q+i,s} = -\mu_k - t_k - \theta_2^* - L^* + \mu_{k+q+i} + t_{k+q+i} + \sum_{p=0}^{s-1}\Delta_{k+q+i+p} + f_{k+q+i+s}$$

pour $s = 1, 2, \ldots$.

$$C^k(\theta_1, L^*, \theta_2^*) = \sum_{s=1}^{q+i}(\theta_1 - z_{k,s})^+ + \sum_{s=1}^{+\infty}(\theta_2^* - z'_{k+q+i,s})^+$$

Nous définissons $z_{k,u}^*$, $u = 1, 2, \ldots, +\infty$ l'ensemble $\{z_{k,s}\}_{s=1,2,\ldots,n_k^1} \cup \{z'_{k+q+i,s}\}_{s=1,2,\ldots}$ ordonné par ordre croissant. Alors :

$$C^k(\theta_1, L^*, \theta_2^*) = \sum_{u=1}^{+\infty}(\theta_1 - z_{k,u}^*)^+ \tag{4.17}$$

Soit m_1 le plus grand entier tel que :

$$z_{k,m_1}^* \leq w_i = \mu_{k+q+i} + t_{k+q+i} - (\mu_k + t_k + L^*)$$

Soit M_1 le plus petit entier tel que :

$$z^*_{k,M_1} \geq w_{i+1} = \mu_{k+q+i+1} + t_{k+q+i+1} - (\mu_k + t_k + L^*) \tag{4.18}$$

Alors,

$$\mathcal{C}^k(\theta_1, L^*, \theta_2^*) = \sum_{u=m_1}^{M_1-1} (\theta_1 - z^*_{k,u})^+ \tag{4.19}$$

La fonction $\mathcal{C}^k(\theta_1, L^*, \theta_2^*)$ est égale à :

$$\mathcal{C}^k(\theta_1, L^*, \theta_2^*) = \begin{cases} 0 & \text{si} \quad \theta_1 < z^*_{k,1} \\ (M_1 - 1 - m_1)\theta_1 - \sum_{i=m_1}^{M_1-1} z^*_{k,i} & \text{si} \quad \theta_1 \in [z^*_{k,u-1}, z^*_{k,u}[\end{cases} \tag{4.20}$$

Ceci termine la démonstration. □

Résultat 6

Soit M_1 l'entier défini dans la preuve précédente. Alors $M_1 \leq M^*$, où M^* est le plus petit entier tel que : $-\mu_k - t_k - \theta_2^* - L^* + \mu_{k+q+i} + t_{k+q+i} + \sum_{p=0}^{M^*-1} \Delta_{k+q+i+p} > w_{i+1}$

Preuve du résultat 6

De l'inégalité (4.18), nous déduisons que : $z'_{k+q+i,s} > w_{i+1}$ pour $s \geq M^*$
Le résultat 6 nous donne une limite supérieure pour M_1. Il nous donne également l'ensemble des $z'_{k+q+i,s}$ à considérer pour le classement, c'est-à-dire pour le calcul de l'ensemble des $z^*_{k,u}$: nous ne considèrons, dans le classement, que l'ensemble $\{z'_{k+q+i,s}\}_{s=1,2,\ldots,M^*}$. Ceci termine la démonstration. □

4.5.2 Exemple

Considérons l'ordonnancement initial des tâches $a_1, a_2, \ldots, a_{10}$ défini comme suit (Le premier terme du triplet est μ_i, le second t_i et le dernier d_i) :
$a_1(0,2,5), a_2(4,3,12), a_3(10,2,14), a_4(13,1,14), a_5(17,2,22), a_6(20,4,29),$

$a_7(25,1,28), a_8(28,2,31), a_9(34,4,41), a_{10}(39,1,40)$

Cet ordonnancement initial est représenté dans la figure 4.9.

Considérons la tâche aléatoire A définie par le triplet $(\theta_1, L^* = 9, \theta_2^* = 3)$. Nous souhaitons commencer la sous-tâche A_1 dès que a_1 se termine, c'est-à-dire à l'instant 2, et nous cherchons à évaluer l'accroîssement du critère en fonction de θ_1, soit $\mathcal{C}^1(\theta_1, 9, 3)$. Quelle que soit la valeur de θ_1, au plus les tâches a_2, a_3, a_4 et a_5 pourront se situer entre la fin de la sous-tâche A_1 et le début de la sous-tâche A_2.

Si $\theta_1 = 0$, seul a_2 est situé entre A_1 et A_2, comme l'indique la figure 4.10. Donc $q = 1$. Dans ce cas, en appliquant par exemple la formule (4.11), nous obtenons : $\mathcal{C}^1(0, 9, 3) = 5$. De plus, $r = 3$ (voir les notations du résultat 5.)

D'après la définition de w_i, $i = 1, 2, 3, 4$, il vient :

$$w_1 = \mu_3 + t_3 - (\mu_1 + t_1 + L^*) = 12 - (11) = 1$$

$$w_2 = \mu_4 + t_4 - (\mu_1 + t_1 + L^*) = 14 - (11) = 3$$

$$w_3 = \mu_5 + t_5 - (\mu_1 + t_1 + L^*) = 19 - (11) = 8$$

Les points de discontinuité de la fonction $\mathcal{C}^k(\theta_1, 9, 3)$ sont donc $\theta_1 = 1, \theta_1 = 3$ et $\theta_1 = 8$.

Il est facile de voir que $\mathcal{C}^1(\theta_1, 9, 3)$ est linéaire pour $\theta_1 \in [0, 1)$ et $\mathcal{C}^1(1^-, 9, 3) = 7$. La situation, lorsque $\theta_1 = 1^-$, est représentée sur la figure 4.11 .

Lorsque $\theta_1 = 1$, la tâche a_3 précède la sous-tâche A_2 comme représenté dans la figure 4.12. En appliquant la formule (4.11) à cette nouvelle situation, nous obtenons :

$\mathcal{C}^1(1, 9, 3) = 2$

Le point de discontinuité suivant est $\theta_1 = 3$. La situation, lorsque $\theta_1 = 3^-$, est représentée sur la figure 4.13 . Dans ce cas, $\mathcal{C}_1^1(3^-, 9, 3) = 0$ et $\mathcal{C}_2^1(3^-, 9, 3) = 4$, donc : $\mathcal{C}^1(3^-, 9, 3) = 4$.

De plus, la fonction $\mathcal{C}^1(\theta_1, 9, 3)$ est linéaire sur $[1, 3)$.

Dans la figure 4.14, nous représentons la situation lorsque $\theta_1 = 3$. Dans ce cas nous obtenons :

$\mathcal{C}^1(3, 9, 3) = 0$

Le dernier point de discontinuité est $\theta_1 = 8$. La situation, lorsque $\theta_1 = 8^-$, est représentée dans la figure 4.15 .

Compte tenu des résultats précédents :

$z_{1,s} = \sum_{p=0}^{s-1} \Delta_{1+p} + f_{1+s}$ pour $s = 1, 2, 3, 4$.

Il vient :

$z_{1,1} = \Delta_1 + f_2 = 2 + 5 = 7$

$z_{1,2} = \Delta_1 + \Delta_2 + f_3 = 2 + 3 + 2 = 7$

$z_{1,3} = \Delta_1 + \Delta_2 + \Delta_3 + f_4 = 2 + 3 + 1 + 0 = 6$

$z_{1,4} = \Delta_1 + \Delta_2 + \Delta_3 + \Delta_4 + f_5 = 2 + 3 + 1 + 3 + 3 = 12 > 8$

Toujours compte-tenu des résultats précédents :

$M^* = 4$ et $z'_{q+k+i,s} = -\mu_k - t_k - L^* - \theta_2^* + \mu_{k+q+i} + t_{k+q+i} + \sum_{p=0}^{s-1} \Delta_{k+q+i+p} + f_{k+q+i+s}$

et :

$z'_{1+1+2,1} = -0 - 2 - 9 - 3 + 13 + 1 + 3 + 3 = 6$

$z'_{1,2} = 3 + 1 + 5 = 9$

$z'_{1,3} = 3 + 1 + 1 + 2 = 7$

$z'_{1,4} = 3 + 1 + 1 + 2 + 1 = 8$

Finalement, les points remarquables entre 3 et 8 sont $\theta_1 = 6$ et $\theta_2 = 7$. Au point $\theta_1 = 6$, la tâche a_4 commence à violer son délai, de même que a_5. Par conséquent :

$\mathcal{C}^1(\theta_1, 9, 3) = 0$ pour $\theta_1 \in [3, 6)$

et

$\mathcal{C}^1(\theta_1, 9, 3) = 2\theta_1 - 12$ pour $\theta_1 \in [6, 7)$

Au point $\theta_1 = 7$, les tâches a_2, a_3 et a_7 commencent à violer leur délai.

Par conséquent : $\mathcal{C}^1(\theta_1, 9, 3) = 5\theta_1 - 33$.

Finalement :

$\mathcal{C}^1(8^-, 9, 3) = 7$.

Pour $\theta_1 = 8$, la tâche a_5 se situe entre les sous-tâches A_1 et A_2, comme l'indique la figure 4.16 .

Le calcul de $\mathcal{C}^1(8, 9, 3)$ se fait alors en utilisant la formule (4.16). avec $k = 1$ et $q + i = 4$. Nous obtenons :

$\mathcal{C}^1(8, 9, 3) = 4$.

A partir de $\theta_1 = 8$, plus aucune discontinuité n'apparaitra dans la fonction $\mathcal{C}^1(\theta_1, 9, 3)$, ce qui signifie que les tâches situées entre les sous-tâches A_1 et A_2 y resteront, et qu'aucune autre tâche ne s'y ajoutera.

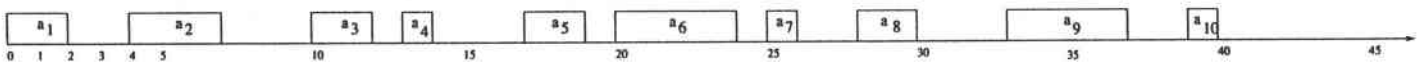


FIG. 4.9 – Ordonnancement initial

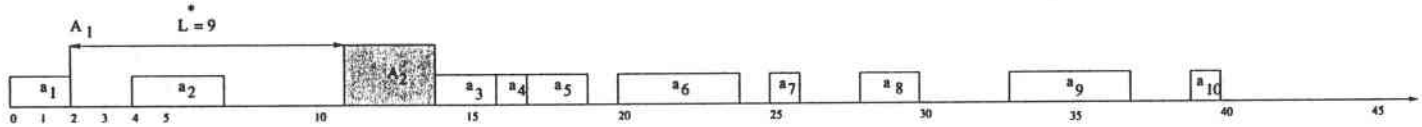


FIG. 4.10 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 0$

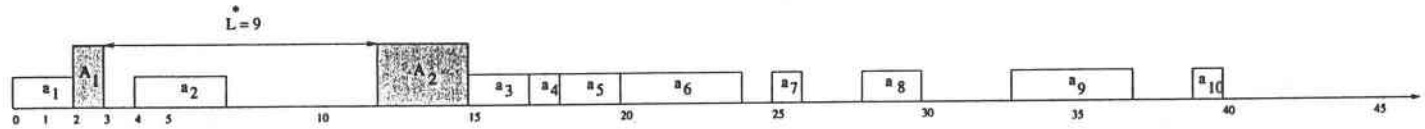


FIG. 4.11 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 1^-$

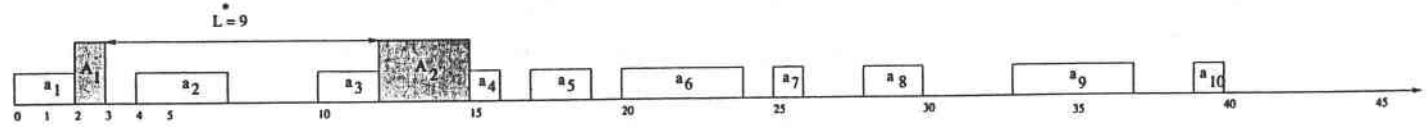


FIG. 4.12 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 1$

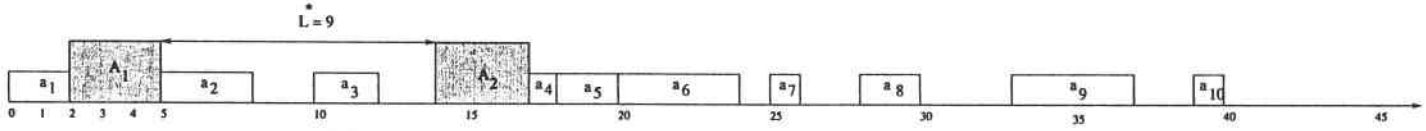


FIG. 4.13 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 3^-$

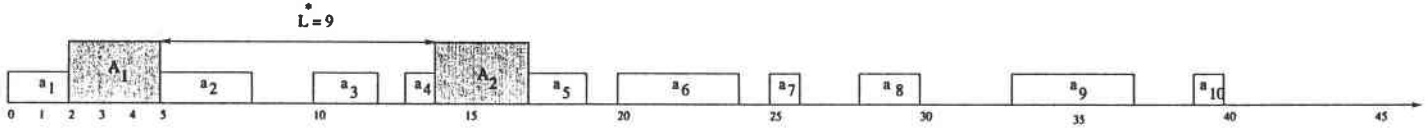


FIG. 4.14 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 3$

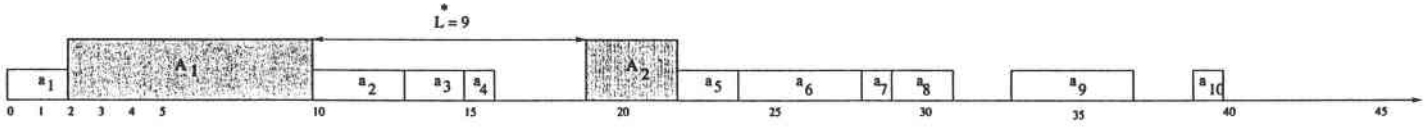


FIG. 4.15 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 8^-$

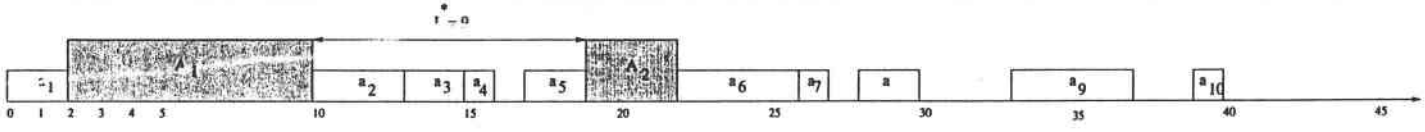


FIG. 4.16 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 8$

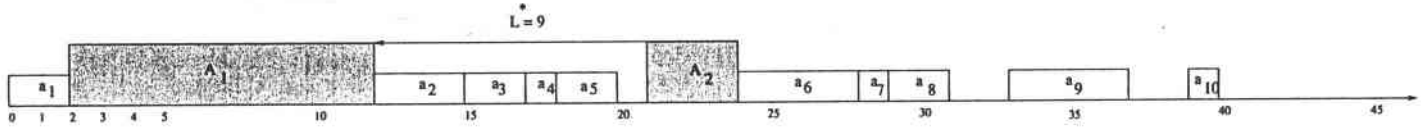


FIG. 4.17 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 10$

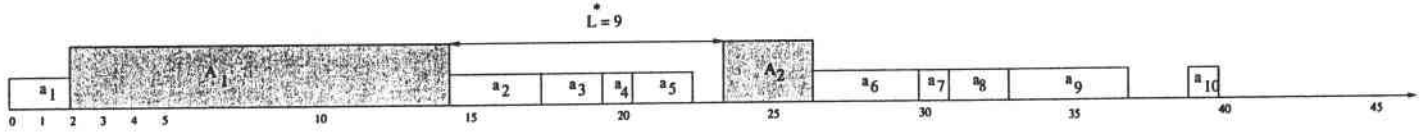


FIG. 4.18 – Introduction de la tâche aléatoire après a_1, avec $\theta_1 = 12$

L'application de la formule (4.16) avec $k = 1$ et $q + i = 4$ conduit aux formulations suivantes :

$C^1(\theta_1, 9, 3) = 3\theta_1 - 20$ si $\theta_1 \in [8, 9]$

$C^1(\theta_1, 9, 3) = 4\theta_1 - 29$ si $\theta_1 \in [9, 10]$

La figure 4.17 présente l'ordonnancement lorsque $\theta_1 = 10$. Nous trouvons ensuite :

$C^1(\theta_1, 9, 3) = 5\theta_1 - 39$ si $\theta_1 \in [10, 11]$

$C^1(\theta_1, 9, 3) = 6\theta_1 - 50$ si $\theta_1 \in [11, 12]$

La figure 4.18 présente l'ordonnancement lorsque $\theta_1 = 12$.

$C^1(\theta_1, 9, 3) = 7\theta_1 - 62$ si $\theta_1 \in [12, 14]$

$C^1(\theta_1, 9, 3) = 8\theta_1 - 76$ si $\theta_1 \in [14, 15]$

$C^1(\theta_1, 9, 3) = 9\theta_1 - 91$ si $\theta_1 \in [15, +\infty]$

La fonction $C^1(\theta_1, 9, 3)$ est représentée sur la figure 4.19 .

4.5.3 Variation de $C^k(\theta_1, L, \theta_2)$ lorsque L varie

Supposons que $\theta_1 = \theta_1^*$ et $\theta_2 = \theta_2^*$ soient fixés, et étudions la variation de $C^k(\theta_1^*, L, \theta_2^*)$ lorsque L varie de 0 à $+\infty$. Nous supposons que l'ordonnancement initial concerne les tâches $a_1, a_2, \ldots, a_n$, et nous posons :

$l_0 = 0$

$l_i = \max(\mu_{k+i} + t_{k+i} - \mu_k - t_k - \theta_1^*, \sum_{p=1}^{i} t_{k+p})$ pour $i = 1, 2, \ldots, n$

$l_{n+1} = +\infty$

Alors le résultat ci-dessous s'applique.

Résultat 7 :

La fonction $C^k(\theta_1^*, L, \theta_2^*)$ est :

- discontinue pour $L = l_i$, $i = 1, 2, \ldots, n$,
- linéaire par morceaux et continue sur tout intervalle $[l_i, l_{i+1}), i = 0, \ldots, n$,
- croissante sur tout intervalle $[l_i, l_{i+1}), i = 0, 1, \ldots, n-1$,
- constante sur $[l_n, l_{n+1})$.

Preuve du résultat 7 :

1. Si $L \in [l_0, l_1)$ alors $0 \leq L < \max(\mu_{k+1} - \mu_k - t_k - \theta_1^*, t_{k+1})$.
   L'inégalité de droite implique :

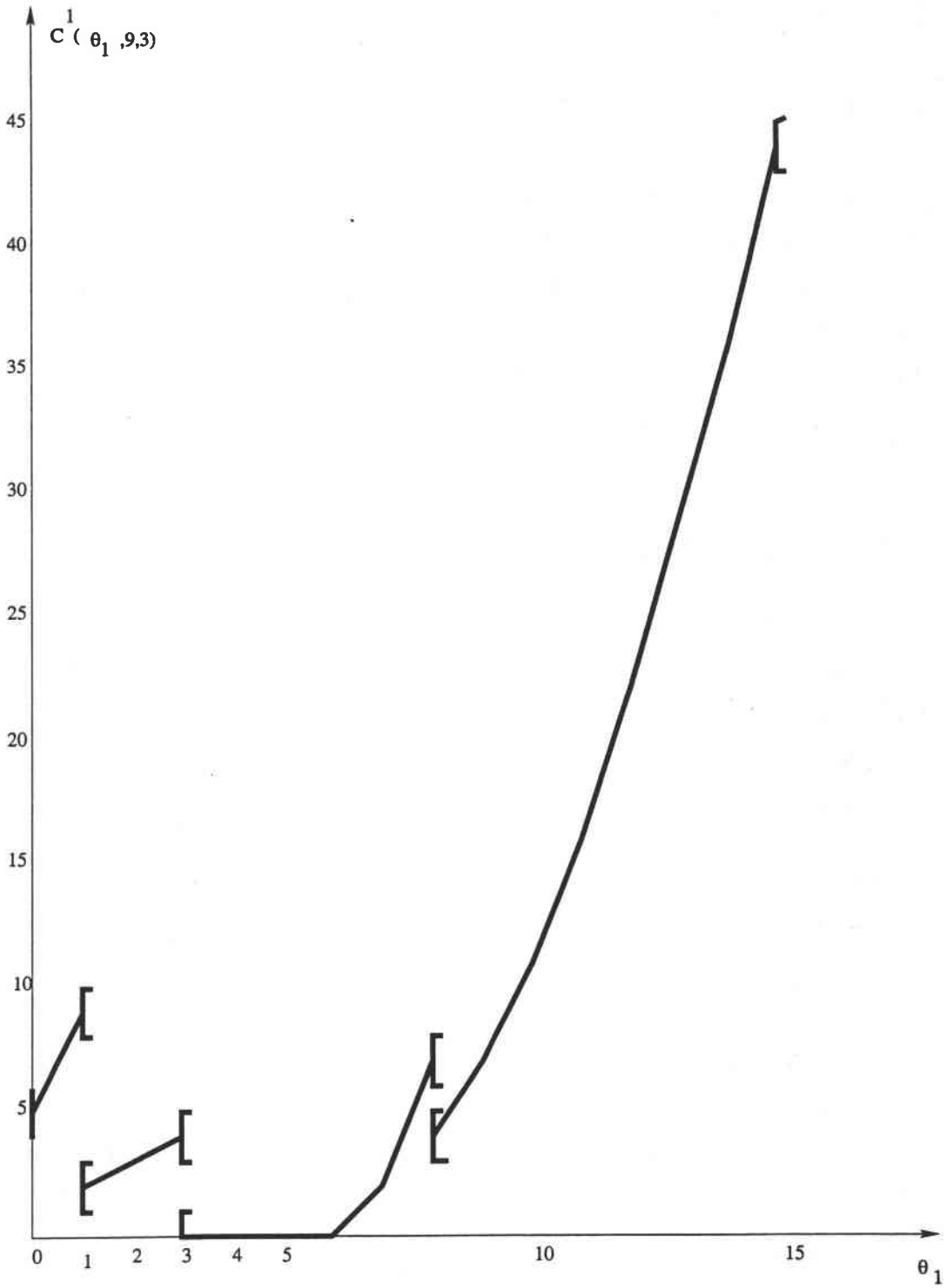


FIG. 4.19 – Fonction $\mathcal{C}^1(\theta_1, 9, 3)$

- soit que $\mu_{k+1}+t_{k+1}-\mu_k-t_k-\theta_1^* < L$, qui s'écrit encore : $\mu_k+t_k+\theta_1^*+L > \mu_{k+1}+t_{k+1}$. Cette inégalité signifie que la tâche a_{k+1} se termine au plus tôt après que la sous-tâche A_2 a commencé.
- soit que $L < t_{k+1}$, ce qui signifie que la durée de la tâche a_{k+1} est trop importante pour placer cette tâche entre les sous-tâches A_1 et A_2.

Par conséquent, quel que soit le minimum de l'inégalité de droite, $L \in [l_0, l_1)$ implique qu'une au moins des conditions requises pour que a_{k+1} puisse être placée entre A_1 et A_2 n'est vérifiée. La relation (4.16) s'écrit donc dans ce cas :

$$C^k(\theta_1^*, L, \theta_2^*) = \sum_{s=1}^{n}(\theta_1^* + L + \theta_2^* - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+ \quad (4.21)$$

Posons :

$z_{k,s} = -\theta_1^* - \theta_2^* + \sum_{p=0}^{s-1}\Delta_{k+p} + f_{k+s}$

et soit :

$Z_k^0 = \{z_{k,s}, s = 1, 2, \ldots, n-k \setminus z_{k,s} \geq 0 \text{ et } z_{k,s} < l_1\}$.

Nous désignons par Z_k^{0*} l'ensemble des éléments de Z_k^0 classés par ordre croissant, et nous les désignons par $z_{k,W}^*$.

Le lecteur notera que $card(Z_k^0) = card(Z_k^{0*})$ est inférieur ou égal au plus petit entier s tel que $\sum_{p=0}^{s-1}\Delta_{k+p} - \theta_1^* - \theta_2^* \geq l_1$.

Finalement :

$$C^k(\theta_1^*, L, \theta_2^*) = \sum_{W=1}^{card(Z_k^{0*})}(L - z_{k,W}^*)^+$$

Cette solution montre que la fonction est linéaire par morceaux, continue et croissante sur $[l_0, l_1)$.

2. Supposons maintenant que $L \in [l_i, l_{i+1}), i = 1, 2, \ldots, n$. Alors :

$\max(\mu_{k+i} + t_{k+i} - \mu_k - t_k - \theta_1^*, \sum_{p=1}^{i} t_{k+p}) \leq L < \max(\mu_{k+i+1} \div t_{k+i+1} - \mu_k - t_k - \theta_1^*, \sum_{p=1}^{i+1} t_{k+p})$

Comme ci-dessus, l'inégalité de droite montre que a_{k+i+1} ne peut être placée entre A_1 et A_2. L'inégalité de gauche conduit à :

$\mu_k + t_k + \theta_1^* + L \geq \mu_{k+i} + t_{k+1}$ et $L \geq \sum_{p=1}^{i} t_{k+p}$

La première inégalité indique que la seconde sous-tâche A_2 commence après que a_{k+i} s'est terminée. La seconde inégalité indique que la période entre la fin de A_1 et le début de A_2 est suffisante pour placer $a_{k+1}, \ldots, a_{k+i}$.

Par conséquent, $L \in [l_i, l_{i+1})$ conduit à (voir la relation (4.16)) :

$$C^k(\theta_1^*, L^*, \theta_2^*) = \sum_{s=1}^{i}(\theta_1^* - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+ +$$

$$\sum_{s=1}^{n-i}(\mu_k + t_k + \theta_1^* + L + \theta_2^* - \mu_{k+i} - t_{k+i} - \sum_{p=0}^{s-1}\Delta_{k+i+p} - f_{k+i+s})^+ \quad (4.22)$$

Le premier terme du second membre de cette égalité est une constant par rapport à L. Nous la notons K. Pour ce qui concerne le second terme, posons :

$$z_{k,s} = -\mu_k - t_k - \theta_2^* - \theta_1^* + \mu_{k+i} + t_{k+i} + \sum_{p=0}^{s-1}\Delta_{k+i+p} + f_{k+i+s}$$

Alors :

$$C^k(\theta_1^*, L, \theta_2^*) = K + \sum_{s=1}^{n-i}(L - z_{k,s})^+$$

Comme précédemment, nous définissons : $Z_k^i = \{z_{k,s}, s = 1, 2, \ldots, n - 2 / z_{k,s} \geq l_i \text{ et } z_{k,s} < l_{i+1}\}$, et nous désignons par Z_k^{i*} l'ensemble des éléments de Z_k^i classés par ordre croissant. Nous notons ces éléments $z_{k,W}^*, W = 1, 2, \ldots, card(Z_k^{i*})$.

Finalement :

$$C^k(\theta_1^*, L, \theta_2^*) = K + \sum_{W=1}^{card(Z_k^{i*})}(L - z_{k,W}^*)^+$$

Cette relation montre que $C^k(\theta_1^*, L, \theta_2^*)$ est linéaire par morceaux, continue et croissante sur $[l_i, l_{i+1})$.

3. Supposons enfin que $L \in [l_n, l_{n+1})$, alors la relation (4.16) se réduit à :

$$C^k(\theta_1^*, L, \theta_2^*) = \sum_{s=1}^{n}(\theta - \sum_{p=0}^{s-1}\Delta_{k+p} - f_{k+s})^+ \quad (4.23)$$

Le second membre n'étant pas défini en fonction de L, nous en déduisons que $C^k(\theta_1^*, L, \theta_2^*)$ est constant par rapport à L. Ceci conclut la preuve □

Résultat 8 :

Avec les notations précédentes :

$$C^k(\theta_1^*, l_i^-, \theta_2^*) \geq C^k(\theta_1^*, l_i, \theta_2^*) \text{ pour } i = 1, \ldots, n$$

Preuve du résultat 8 :

Le passage de $L = l_i^-$ à $L = l_i$, pour $i = 1, 2, \ldots, n$ se traduit par :

- La relocalisation de la tâche a_i. Pour $L = l_i^-$, a_i débute après que la sous-tâche A_2 se termine, alors que a_i se retrouve entre A_1 et A_2 lorsque $L = l_i$. L'apport de a_i à l'augmentation de critère est donc moins importante lorsque $L = l_i$ que lorsque $L = l_i^-$.
- du fait de la relocalisation de la tâche a_i, le début des tâches $a_{i+1}, a_{i+2,\ldots,a_n}$ lorsque $L = l_i$ est inférieur ou égal au début de ces mêmes tâches lorsque $L = l_i^-$. Leur apport à l'augmentation du critère est donc inférieur ou égal lorsque $L = l_i$ que lorsque $L = l_i^-$
- enfin, l'apport des tâches $a_1, a_2, \ldots, a_{i-1}$ (lorsqu'elles existent), est le même dans les deux cas.

Ceci conclut la preuve. □

4.5.4 Exemple

Nous conservons le même ordonnancement que ci-dessus (voir figure 4.19) et, comme dans le cas précédent, nous introduisons la tâche aléatoire après a_1. Cette tâche aléatoire est initialement définie par le triplet $(\vartheta_1^* = 3, L = 0, \theta_2^* = 3)$. En fait, cette situation revient à introduire après a_1 une tâche aléatoire simple de durée $\theta_1^* + \theta_2^*$=6. Nous appliquons la relation (4.16) en tenant compte du fait que toutes les tâches sauf a_1 sont repoussées après A_2. On obtient :

$$\begin{aligned} C^k(3,0,3) &= \sum_{s=1}^{10}(\mu_1 + t_1 + \theta_1^* + \theta_2^* - \mu_1 - t_1 - \sum_{p=0}^{s-1}\Delta_{1+p} - f_{1+s})^+ \\ &= \sum_{s=1}^{10}(6 - \sum_{p=0}^{s-1}\Delta_{1+p} - f_{1+s})^+ \\ &= 0 \end{aligned}$$

L'ordonnancement obtenu après avoir intercallé la tâche aléatoire est représenté dans la figure 4.20 .

Calculons les l_i tels que définis au début de ce paragraphe.

$l_1 = \max(\mu_2 + t_2 - \mu_1 - t_1 - 3, t_2) = \max(7 - 2 - 3, 3) = 3$

$l_2 = \max(12 - 2 - 3, 5) = 7$

$l_3 = \max(14 - 2 - 3, 6) = 9$

$l_4 = \max(19 - 2 - 3, 8) = 14$

$l_5 = \max(24 - 2 - 3, 12) = 19$

$l_6 = \max(26 - 2 - 3, 13) = 21$

$l_7 = \max(30 - 2 - 3, 15) = 25$

$l_8 = \max(37 - 2 - 3, 19) = 32$

$l_9 = \max(40 - 2 - 3, 20) = 35$

La première discontinuité apparait pour $L = l_1 = 3$

Calculons maintenant les $z_{1,s}$ pour $s = 1, 2, \ldots, 9$

$z_{1,1} = -6 + \Delta_1 + f_2 = -6 + 2 + 5 = 1$

$z_{1,2} = -6 + \Delta_1 + \Delta_2 + f_3 = -6 + 2 + 3 + 2 = 1$

$z_{1,3} = -6 + \Delta_1 + \Delta_2 + \Delta_3 + f_4 = -6 + 2 + 3 + 1 + 0 = 0$

$z_{1,4} = -6 + \sum_{i=1}^{4} + f_5 = -6 + 2 + 3 + 1 + 3 + 3 = 6$

$z_{1,5} = -6 + \sum_{i=1}^{5} + f_6 = -6 + 2 + 3 + 1 + 3 + 1 + 5 = 9$

$z_{1,6} = -6 + \sum_{i=1}^{6} + f_7 = -6 + 11 + 2 = 7$

$z_{1,7} = -6 + \sum_{i=1}^{7} + f_8 = -6 + 11 + 2 + 1 = 8$

$z_{1,8} = -6 + \sum_{i=1}^{8} + f_9 = -6 + 13 + 4 = 11$

$z_{1,9} = -6 + \sum_{i=1}^{9} + f_10 = -6 + 15 + 0 = 9$

Dans le premier intervalle qui nous intéresse, à savoir l'intervalle $[l_0, l_1] = [0, 3]$, le point 1 présente une discontinuité de la dérivée.

$\mathcal{C}^1(3, L, 3) = L$ si $L \in [0, 1]$

$\mathcal{C}^1(3, L, 3) = 3L - 2$ si $L \in [1, 3]$

Donc $\mathcal{C}^1(3, 3^-, 3) = 7$

Considérons maintenant l'intervalle $[l_1, l_2) = [3, 7)$. Nous ne présentons pas le calcul des $z_{1,i}, i = 1, \ldots, 8$, sur cet intervalle par manque de place.

Lorsque L=3, la situation est donnée par la figure 4.21 et la relation (4.16)

nous conduit à :

$$\begin{aligned} \mathcal{C}^1(3,3,3) &= (3-\Delta_1-f_2)^+ \\ &\quad +\sum_{s=1}^{8}(\mu_1+t_1+3+3+3-\mu_2-t_2-\sum_{p=0}^{s-1}\Delta_{2+p}-f_{2+s})^+ \\ &= 0+(11-7-3-2)^+ +(11-12-4-0)^+ \\ &= 0 \end{aligned}$$

Sur l'intervalle $[3,7)$, seul le point 4 présente une discontinuité de la dérivée.

$\mathcal{C}^1(3,L,3) = L-3$ si $L \in [3,4]$

$\mathcal{C}^1(3,L,3) = 2L-7$ si $L \in [4,7]$

Donc $\mathcal{C}^1(3,7^-,3) = 7$

Considérons maintenant l'intervalle $[l_2,l_3) = [7,9)$. Ici encore, nous ne présentons pas le calcul des $z_{1,i}, i = 1,2,\ldots,7$.

Lorsque $L = 7$, l'ordonnancement est donné par la figure 4.22 et l'application de la relation (4.16) conduit à :

$\mathcal{C}^1(3,7,3) = 2$

Sur l'intervalle $[7,9)$, nous ne rencontrons aucune discontinuité de la dérivée. En d'autres termes, aucun des $z_{1,i}, i = 1,\ldots,7$, ne figure entre 7 et 9.

$\mathcal{C}^1(3,L,3) = L-5$ pour $L \in [7,9)$

et $\mathcal{C}^1(3,9^-,3) = 4$

Nous considèrons ensuite l'intervalle $[l_3,l_4) = [9,14)$. La figure 4.23 donne l'ordonnancement lorsque $L = 9$. La relation (4.16) conduit à :

$\mathcal{C}^1(3,9,3) = 0$

Sur l'intervalle $[9,14)$, le calcul des $z_{1,i}$ correspondant nous donnerait un seul point de discontinuité de la dérivée : le point 12.

Nous obtenons :

$\mathcal{C}^1(3,L,3) = 0$ pour $L \in [9,12)$

$\mathcal{C}^1(3,L,3) = L-12$ pour $L \in [12,14)$

et $\mathcal{C}^1(3,14-,3) = 2$

L'intervalle suivant est l'intervalle $[l_4,l_5) = [14,19)$. La figure 4.24 nous donne l'ordonnancement lorsque $L = 14$ et nous obtenons :

$\mathcal{C}^1(3,14,3) = 0$

Les points 15,16,17 sont des points de discontinuité de la dérivée et :

$\mathcal{C}^1(3,L,3) = 0$ pour $L \in [14,15)$

$\mathcal{C}^1(3, L, 3) = L - 15$ pour $L \in [15, 16)$

$\mathcal{C}^1(3, L, 3) = 2L - 31$ pour $L \in [16, 17)$

$\mathcal{C}^1(3, L, 3) = 3L - 48$ pour $L \in [17, 19)$

et $\mathcal{C}^1(3, 19-, 3) = 9$

L'intervalle suivant est l'intervalle $[l_5, l_6) = [19, 21)$.

La figure 4.25 nous donne l'ordonnancement lorsque $L = 19$ et nous obtenons :

$\mathcal{C}^1(3, 19, 3) = 0$

Le point 20 est un point de discontinuité de la dérivée de $\mathcal{C}^1(3, L, 3)$ et :

$\mathcal{C}^1(3, L, 3) = L - 19$ pour $L \in [19, 20)$

$\mathcal{C}^1(3, L, 3) = 2L - 39$ pour $L \in [20, 21)$

et $\mathcal{C}^1(3, 21^-, 3) = 3$

Considérons maintenant l'intervalle $[l_6, l_7) = [21, 25)$.

La figure 4.26 nous donne l'ordonnancement pour $L = 21$ et $\mathcal{C}^1(3, 21, 3) = 0$. Il n'y a pas de point de discontinuité entre 21 et 25 et :

$\mathcal{C}^1(3, L, 3) = L - 21$ pour $L \in [21, 25)$

et $\mathcal{C}^1(3, 25^-, 3) = 4$

L'intervalle suivant est l'intervalle $[l_7, l_8) = [25, 32)$. Pour L=25, l'ordonnancement est donné par la figure 4.27 et $\mathcal{C}^1(3, 25, 3) = 0$

Les points 27 et 29 sont des points de discontinuité pour la dérivée première

$\mathcal{C}^1(3, L, 3) = 0$ pour $L \in [25, 27)$

$\mathcal{C}^1(3, L, 3) = L - 27$ pour $L \in [27, 29)$

$\mathcal{C}^1(3, L, 3) = 2L - 56$ pour $L \in [29, 32)$

Considérons maintenant l'intervalle $[l_8, l_9) = [32, 35)$.

L'ordonnancement pour $L = 32$ est donné dans la figure 4.28 et $\mathcal{C}^1(3, 32, 3) = 1$. Aucun point de discontinuité de la dérivée première ne figure entre 32 et 35 et :

$\mathcal{C}^1(3, L, 3) = L - 31$ pour $L \in [32, 35)$

De plus, $\mathcal{C}^1(3, 35^-, 3) = 4$

Enfin, il est facile de voir que, pour $L \in [35, +\infty)$, $\mathcal{C}^1(3, L, 3) = 0$. La fonction $\mathcal{C}^1(3, L, 3)$ est représentée sur la figure 4.29 et $\mathcal{C}^1(3, 32^-, 3) = 8$

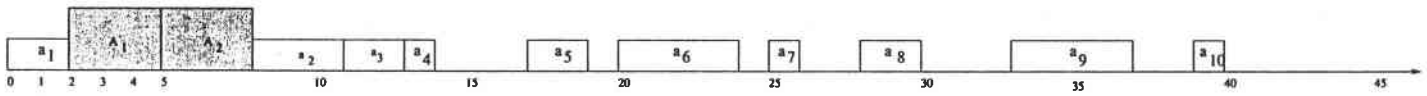


FIG. 4.20 – Introduction de la tâche aléatoire (3,0,3) après a_1

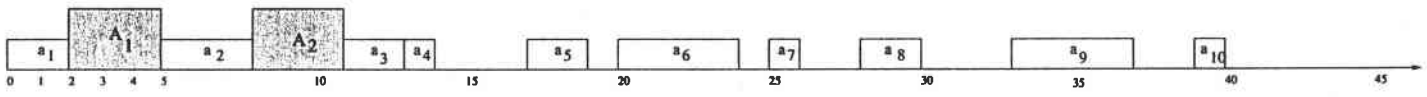


FIG. 4.21 – Introduction de la tâche aléatoire (3,3,3) après a_1

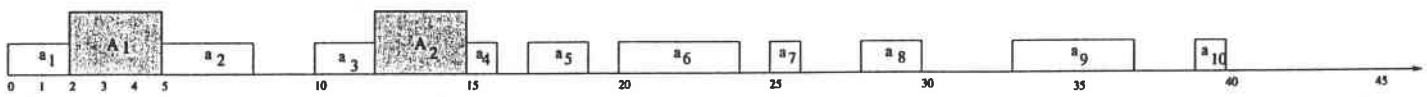


FIG. 4.22 – Introduction de la tâche aléatoire (3,7,3) après a_1

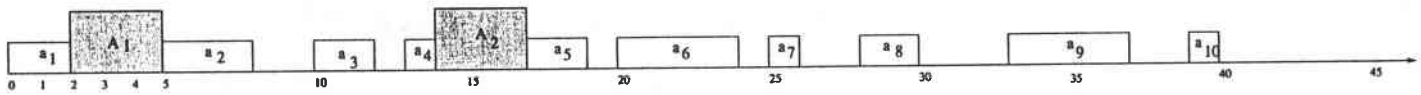


FIG. 4.23 – Introduction de la tâche aléatoire (3,9,3) après a_1

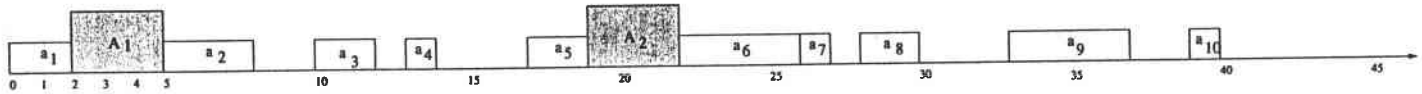


FIG. 4.24 – Introduction de la tâche aléatoire (3,14,3) après a_1

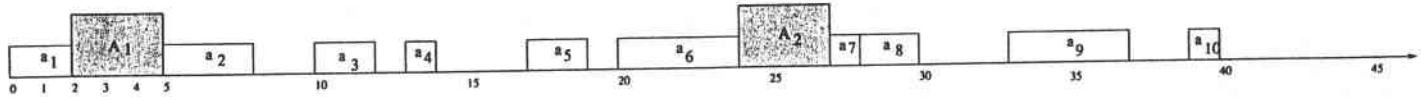


FIG. 4.25 – Introduction de la tâche aléatoire (3,19,3) après a_1

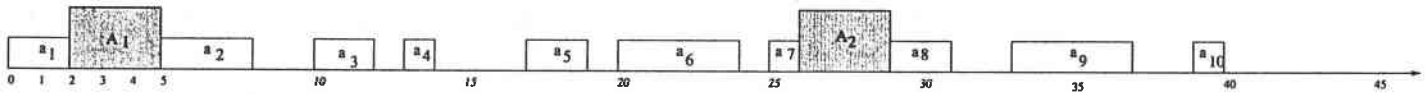


FIG. 4.26 – Introduction de la tâche aléatoire (3,21,3) après a_1

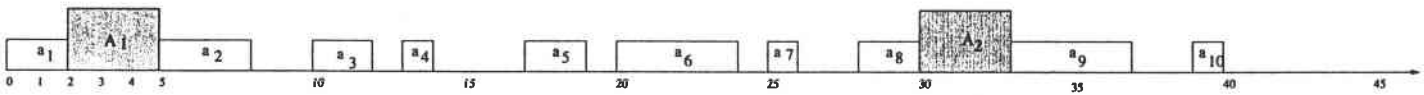


FIG. 4.27 – Introduction de la tâche aléatoire (3,25,3) après a_1

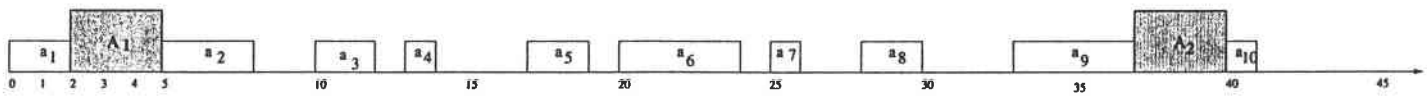


FIG. 4.28 – Introduction de la tâche aléatoire (3,32,3) après a_1

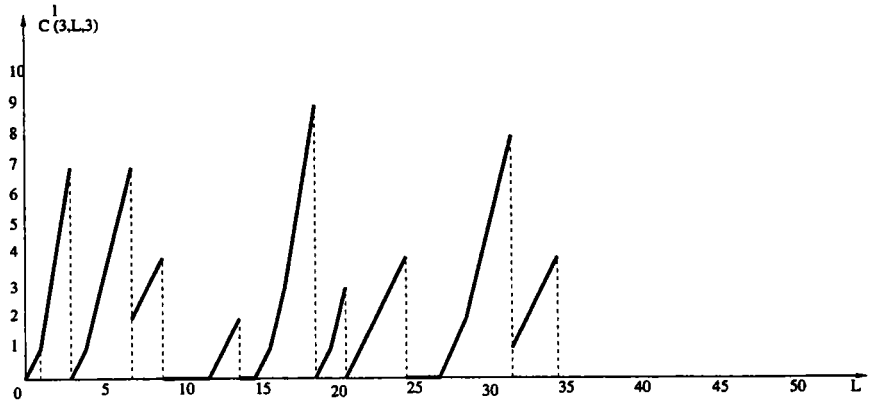


FIG. 4.29 – Fonction $C^1(3, L, 3)$

4.5.5 Variation de $C^k(\theta_1, L, \theta_2)$ lorsque θ_2 varie

La fonction obtenue est linéaire, continue par morceaux. En effet, puisque L et θ_1 sont fixés, faire varier θ_2 revient à insérer une seule tâche. (i.e. θ_1 et L sont connus, donc nous savons quelles sont les tâches que nous allons devoir déplacer pour insérer A_2 : c'est exactement le problème de l'insertion d'une tâche.)

4.6 Algorithme temps-réel

L'algorithme que nous présentons est un algorithme en ligne. Nous faisons donc en sorte d'effectuer un maximum de calculs en différé. Pour y parvenir, nous faisons l'hypothèse que le délai D de la tâche aléatoire correspond au début d'exécution d'une des tâches déjà ordonnancées. En d'autres termes, si nous avons $\mu_k \leq D < \mu_{k+1}$, nous admettons que $D = \mu_k$.
Notre tableau de référence Y sera identique au tableau fourni dans le cas d'une tâche simple, comme indiqué dans tableau 4.5. Nous nommons les éléments constituant ce tableau $y_{i,j}$ où i est l'indice de la ligne et j l'indice de la colonne.

Nous utilisons également le tableau de référence V identique au tableau de référence du cas d'une tâche aléatoire unique (voir figure 4.6). Les éléments de ce tableau sont nommés $v_{i,j}$ où i est l'indice de la ligne et j l'indice de la colonne. La seule différence réside dans son utilisation : si le délai a été corrigé à μ_n, alors la dernière tâche après laquelle nous pouvons placer la tâche aléatoire

TAB. 4.5 – Exemple de tableau de référence Y.

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8
a_1	0	$\Delta_1 + f_2$	$\sum_{i=1}^{2} \Delta_i + f_3$	$\sum_{i=1}^{3} \Delta_i + f_4$	$\sum_{i=1}^{4} \Delta_i + f_5$	$\sum_{i=1}^{5} \Delta_i + f_6$	$\sum_{i=1}^{6} \Delta_i + f_7$	$\sum_{i=1}^{7} \Delta_i + f_8$
a_2	0	0	$\Delta_2 + f_3$	$\sum_{i=2}^{3} \Delta_i + f_4$	$\sum_{i=2}^{4} \Delta_i + f_5$	$\sum_{i=2}^{5} \Delta_i + f_6$	$\sum_{i=2}^{6} \Delta_i + f_7$	$\sum_{i=2}^{7} \Delta_i + f_8$
a_3	0	0	0	$\Delta_3 + f_4$	$\sum_{i=3}^{4} \Delta_i + f_5$	$\sum_{i=3}^{5} \Delta_i + f_6$	$\sum_{i=3}^{6} \Delta_i + f_7$	$\sum_{i=3}^{7} \Delta_i + f_8$
a_4	0	0	0	0	$\Delta_4 + f_5$	$\sum_{i=4}^{5} \Delta_i + f_6$	$\sum_{i=4}^{6} \Delta_i + f_7$	$\sum_{i=4}^{7} \Delta_i + f_8$
a_5	0	0	0	0	0	$\Delta_5 + f_6$	$\sum_{i=5}^{6} \Delta_i + f_7$	$\sum_{i=5}^{7} \Delta_i + f_8$
a_6	0	0	0	0	0	0	$\Delta_6 + f_7$	$\sum_{i=6}^{7} \Delta_i + f_8$
a_7	0	0	0	0	0	0	0	$\Delta_7 + f_8$

est la tâche a_{k^*} telle que :

$$v_{k^*,n} \geq \theta_1 + L + \theta_2$$

Nous ne reviendrons pas sur cet aspect. Notons que si a_{k^*} est la dernière tâche après laquelle nous pouvons placer la tâche aléatoire, alors nous pouvons également placer cette tâche après a_k pour $k = 1, \ldots, k^*$.

TAB. 4.6 – Exemple de tableau de référence V.

	μ_1	μ_2	μ_3	μ_4	μ_5
a_1	0	$\mu_2 - (\mu_1 + \theta_1)$	$\mu_3 - (\mu_1 + \theta_1)$	$\mu_4 - (\mu_1 + \theta_1)$	$\mu_5 - (\mu_1 + \theta_1)$
a_2	0	0	$\mu_3 - (\mu_2 + \theta_1)$	$\mu_4 - (\mu_2 + \theta_2)$	$\mu_5 - (\mu_2 + \theta_2)$
a_3	0	0	0	$\mu_4 - (\mu_3 + \theta_3)$	$\mu_5 - (\mu_3 + \theta_3)$
a_4	0	0	0	0	$\mu_5 - (\mu_4 + \theta_4)$

Nous examinons maintenant comment utiliser le tableau de référence Y, calculé en différé, pour évaluer l'augmentation du critère dans chacun des cas examinés précédemment.

Le tableau de référence V nous donne donc la liste des tâches à la suite desquelles nous pouvons placer la tâche aléatoire.

Soit i une de ces tâches. Il convient alors de se servir au mieux du tableau 4.5 pour définir l'augmentation du critère suite à ce positionnement.

Nous calculons d'abord n_k^1 à l'aide de (4.1), la suite des calculs dépend du cas dans lequel nous nous plaçons.

4.6.1 Cas 1 de l'algorithme

Les éléments du tableau de référence Y sont désignés par $y_{k,j}$.
Nous calculons C_1^k de la manière suivante, en nous référant à (4.2) :

$$C_1^k = \begin{cases} 0 & \text{si} \quad \Delta_k \geq \theta_1 \\ \sum_{s=1}^{n_k^1} (\theta_1 - y_{k,k+s})^+ & \text{si} \quad \Delta_k < \theta_1 \end{cases} \tag{4.24}$$

Puis nous calculons q, θ_2^* à l'aide de (4.3) et n_k^2 à l'aide de (4.6) et nous obtenons (voir (4.4)) :

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_{k+q} \geq \theta_2^* \\ \sum_{s=1}^{n_k^2} (\theta_2^* - y_{k,k+q+s})^+ & \text{si} \quad \Delta_{k+q} < \theta_2^* \end{cases} \tag{4.25}$$

4.6.2 Cas 2 de l'algorithme

Le coût supplémentaire C_1^k se calcule comme précédemment, en utilisant la relation (4.11).
Nous calculons ensuite Δ_k^*, θ_2^{**} et n_k^3 à l'aide de (4.7) :

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_k^* \geq \theta_2^{**} \\ \sum_{s=1}^{n_k^3} (\theta_2^{**} - \Delta_k^* - y_{k,k+n_k^1+s} + \Delta_{k+n_k^1})^+ & \text{si} \quad \Delta_k^* < \theta_2^{**} \end{cases} \tag{4.26}$$

4.6.3 Cas 3 de l'algorithme

Nous commençons par calculer r_k^1, puis C_1^k avec la formule suivante :

$$C_1^k = \sum_{s=1}^{r_k^1} (\theta_1 - y_{k,k+s})^+ \tag{4.27}$$

Nous calculons ensuite Δ_k^{**}, θ_2^{**} et n_k^4 à l'aide de (4.10), puis C_2^k en utilisant la formule (4.9) :

$$C_2^k = \begin{cases} 0 & \text{si} \quad \Delta_k^{**} \geq \theta_2^{**} \\ \sum_{s=1}^{n_k^4} (\theta_2^{**} - \Delta_k^{**} - y_{k,k+r_k^1+s} + \Delta_{k+r_k^1})^+ & \text{si} \quad \Delta_k^{**} < \theta_2^{**} \end{cases} \tag{4.28}$$

4.6.4 Algorithme TR-Bitâche-Augmentation du critère

L'algorithme temps-réel que nous venons de décrire est formalisé dans la figure 4.30.

Il se décompose comme toujours en deux parties :

- Une partie en différé : elle est constituée du calcul des valeurs Δ_i, $i \in \{1, \ldots, n-1\}$ et f_j, $j \in \{1, \ldots, n\}$ (voir chapitre 2). A l'aide de ces valeurs, nous calculons le tableau de référence Y, qui va contenir le résultat de calculs nécessaires pour exécuter la partie en ligne de l'algorithme, et que nous pouvons calculer par avance. Enfin, nous calculons le tableau de références V qui va nous permettre de connaitre quelle est la dernière tâche de l'ordonnancement actuel derrière laquelle nous pouvons insérer la bitâche aléatoire.
- Dans la partie en ligne, nous déterminons quelles sont les tâches après lesquelles nous pouvons insérer la bitâche aléatoire à l'aide du tableau V. La suite des calculs dépend du cas dans lequel nous nous trouvons (ces trois cas sont décrits dans la section 4.6).

4.7 Complexité de l'algorithme TR-Bitâche-Augmentation du critère

Cette section donne la complexité, dans le pire des cas, de l'algorithme proposé dans la section précédente. Le premier paragraphe donne la complexité des calculs effectués en différé, et le second paragraphe donne la complexité des calculs exécutés en ligne.

4.7.1 La complexité de la partie en différé

Dans cette partie, il y a trois entités à calculer :

- Les Δ_i et les f_i seront calculés en même temps que les tableaux de référence.
- Le tableau de référence Y. Cette complexité a été évaluée dans la section 2.6.4. Elle est de $\mathcal{C}_1 = 3n^2 - n$.
- Le tableau de référence V. Cette complexité a été évaluée dans la section 2.6.4. Elle est de $\mathcal{C}_2 = \frac{5n^2-3n}{2}$.

- Partie en différé
  1. Calcul des Δ_i, $i \in \{1,2,\ldots,n-1\}$ et des f_j, $j \in \{1,2,\ldots,n\}$.
  2. Calcul du tableau de référence Y.
  3. Calcul du tableau de référence V.
- Partie en temps-réel
  1. Utiliser le tableau V pour savoir quelle est la dernière tâche de l'ordonnancement après laquelle on peut insérer la bitâche aléatoire sans violer son délai. Nous nommons k^+ l'indice de cette tâche.
  2. Pour chaque valeur de k, $k \in \{1,2,\ldots,k^+\}$ faire
     (a) Calcul de n_k^1 à l'aide de (4.1))
     (b) Cas 1 : si $\sum_{i=1}^{n_k^1} t_{k+i} < L$ et $a_{k+n_k^1+1} + t_{k+n_k^1+1} \leq a_k + t_k + \theta_1 + L$, faire :
        i. Calcul de C_1^k à l'aide de (4.24))
        ii. Calcul du plus grand entier q tel que : $\mu_{k+q} + t_{k+q} \leq \mu_k + t_k + \theta_1 + L$
        iii. Calcul de θ_2^* à l'aide de (4.3))
        iv. Calcul de n_k^2 à l'aide de (4.5)
        v. Calcul de C_2^k à l'aide de (4.25)
        vi. Calcul de $C_k^* = C_1^k + C_2^k$.
     (c) Cas 2 : si $\sum_{i=1}^{n_k^1} t_{k+i} < L$ et $a_{k+n_k^1+1} + t_{k+n_k^1+1} > a_k + t_k + \theta_1 + L$, faire :
        i. Calcul de C_1^k à l'aide de (4.24)
        ii. Calcul de $\Delta_k^* = \mu_{k+n_k^1+1} - (\mu_k + t_k)$
        iii. Calcul de $\theta_2^{**} = \theta_1 + L + \theta_2$
        iv. Calcul de n_k^3 à l'aide de (4.7)
        v. Calcul de C_2^k à l'aide de (4.26)
        vi. Calcul de $C_k^* = C_1^k + C_2^k$.
     (d) Cas 3 : si $\sum_{i=1}^{n_k^1} t_{k+i} > L$
        i. Calcul de r_k^1 tel que : $\sum_{i=1}^{r_k^1} t_{k+i} \leq L < \sum_{i=1}^{r_k^1+1} t_{k+i}$
        ii. Calcul de C_1^k à l'aide de (4.27)
        iii. Calcul de $\Delta_k^{**} = \mu_{k+r_k^1+1} - (\mu_k + t_k)$
        iv. Calcul de $\theta_2^{**} = \theta_1 + L + \theta_2$
        v. Calcul de n_k^4 à l'aide de (4.10)
        vi. Calcul de C_2^k à l'aide de (4.28)
        vii. Calcul de $C_k^* = C_1^k + C_2^k$.
     (e) Placer dans C_k^{**} la plus petite valeur de critère calculée jusqu'à maintenant. Consigner dans k^{**} la valeur de k menant à ce critère.
  3. Placement de la bitâche après la tâche k^{**}.

FIG. 4.30 – *Algorithme TR-Bitâche-Augmentation du critère*

La complexité totale de la partie en différée de l'algorithme est donc de : $C_{diff} = C_1 + C_2 = 3n^2 - n + \frac{5n^2-3n}{2} = \frac{11n^2-5n}{2} \leq \frac{11n^2}{2}$.
Cette complexité est en $O(n^2)$.

4.7.2 La complexité de la partie en ligne

L'évaluation du nombre de tâches à considérer pour les calculs en ligne nécessite l'exploration du tableau de référence V. Cette exploration a une complexité de n. Ensuite quel que soit le cas où nous nous plaçons, l'utilisation des calculs effectués lors de la partie en différé permet de supprimer une grande partie de la complexité : les calculs restants sont de simples sommations. La complexité de la partie en ligne de l'algorithme est donc en $O(n)$.

4.8 Evaluation de l'algorithme TR-Bitâche-Augmentation du critère

Cette section se divise en deux parties. La première partie décrit la génération des jeux de tests, et la seconde partie donne les résultats des simulations.

4.8.1 Protocole

Nous supposons que nous disposons d'un ordonnancement existant de tâches simples, et qu'une bitâche aléatoire vient d'apparaitre. Nous voulons évaluer les performances de l'algorithme TR-Bitâche-Augmentation du critère. Pour cela nous comparons les résultats qu'il nous fournit par rapport à ceux fournis par un algorithme de nous considérons comme algorithme de référence. Cet algorithme de référence calcule de façon systématique quel est le meilleur instant d'insertion de la bitâche aléatoire au sens du critère (la minimisation de la somme des délais). Pour cela, il calcule la valeur du critère pour chaque instant d'insertion possible de la bitâche, c'est-à-dire immédiatement après la fin de chaque tâche de l'ordonnancement initial.

Des jeux de test sont générés au hasard afin de permettre une comparaison des résultats fournis par l'algorithme de référence et par l'algorithme TR-Bitâche-Augmentation du critère.

Pour $i = 1$ à N faire
- Calculer un ordonnancement de base de k tâches simples. Les paramètres de chaque tâche simple sont générés de façon aléatoire.
- Génération aléatoire des paramètres de la bitâche.
- Calcul de la meilleure solution à l'aide de l'algorithme de référence.
- Calcul de la meilleure solution fournie par l'algorithme TR-Bitâche-Augmentation du critère.
- Comparaison des résultats fournis par les deux algorithmes. On considère comme acceptable le résultat fourni par l'algorithme TR-Bitâche-Augmentation du critère, si le critère de la solution fournie est inférieure ou égal au critère de la solution fournie par l'algorithme de référence à p % près.

FIG. 4.31 – *Algorithme de comparaison*

Un jeu de test est conçu de la manière suivante :

- Nous génèrons à l'aide de nombres aléatoires les paramètres de k tâches (c'est-à dire l'instant de début, la durée et le délai) permettant de bâtir un ordonnancement de base.
- Nous définissons une bitâche aléatoire. A l'aide de nombres aléatoires nous générons les durées, de la première sous-tâche, du temps disponible entre les deux sous-tâches, de la seconde sous-tâche, et enfin son délai.

Le jeu de test est alors soumis aux deux algorithmes.

Nous répètons ce processus N fois de façon à connaitre les performances de l'algorithme TR-Bitâche-Augmentation du critère. Dans la suite, nous donnerons à N la valeur 1000.

Les étapes de la simulation sont décrites dans la figure 4.31.

4.8.2 Résultats

Nous choisissons de définir les tâches constituant les jeux de test de la façon suivante :

- La durée d'une tâche est comprise entre 1 et *Durée-Tâche* unités de temps. Cette valeur sera choisie entre 5 et 50.
- Le délai d'une tâche est situé au hasard entre 25 unités de temps avant son achèvement et 25 unités de temps après son achèvement.
- Le temps libre séparant la fin de l'exécution de la tâche courante et le début de la suivante est compris entre 1 et 25 unités de temps.

La bitâche aléatoire possède les caractéristiques suivantes :

- La durée de la première tâche est comprise entre 1 et 50 unités de temps

TAB. 4.7 – Résultats des simulations.

p / Durée-Tâche	5	10	20	50
0.00	978	951	905	774
0.05	978	953	909	781
0.10	981	957	915	782
0.15	982	965	925	840
0.20	983	968	926	819

- La durée de la seconde tâche est comprise entre 1 et 50 unités de temps
- La durée du temps diponible entre les deux tâches (i.e. la fenêtre de disponobilité) est comprise entre 1 et 50 unités de temps.
- Le délai est un instant faisant partie de l'intervalle de temps de l'ordonnancement initial.

Nous choisissons de créer des jeux de test composés de cinquante tâches simples.

Le nombre de simulations effectuées est de 1000.

Dans le tableau 4.7, nous consignons les résultats des simulations. Nous faisons varier deux paramètres :

- *Durée-Tâche* : durée maximale des tâches simples. Dans le tableau 4.7, ce paramètre est choisi dans l'ensemble $\{5, 10, 20, 50\}$.
- Chaque élément du tableau fournit le nombre de tests qui, en appliquant l'algorithme TR-Bitâche-Augmentation du critère, ont conduit à un critère compris entre une fois et (p+1) fois le critère de référence fourni par l'algorithme de référence.

Les résultats indiquent que notre algorithme est performant, surtout dans le cas où les tâches constituant l'ordonnancement de base ont des durées plus faibles que les durées des tâches de la bitâche aléatoire.

4.9 Conclusion

Les simplifications apportées proviennent :

- de l'approximation sur D (qui dégrade le résultat).
- de l'utilisation du tableau de référence Y, calculé en différé, et qui fournit une partie des calculs.

La suite consistera à trouver d'autres approximations afin de simplifier encore les calculs "temps réels" tout en contrôlant la perte de précision des résultats.

Résumé

Analyse de la fonction augmentation du critère :

- *Variation de $C^k(\theta_1, L, \theta_2)$ lorsque θ_1 varie*
  Supposons que nous commençions la première sous-tâche dès que a_i se termine. Soit r le plus petit entier tel que $\mu_r + t_r > \mu_i + t_i + \theta_1 + L$.
  Soit s le plus grand entier inférieur à r tel que $\sum_{j=i+1}^{s} t_j \leq L$.
  Alors, après placement de la tâche $a_{i+1}, a_{i+2}, a_{i+3}, \ldots, a_s$ se situeront entre A_1 et A_2, et $a_{s+1}, a_{s+2}, a_{s+3}, \ldots, a_n$ se situeront après A_2.
  Les valeurs de θ_1 qui introduisent des discontinuités dans la fonction accroîssement du coût sont $\theta_1^s = (\mu_s + t_s - \mu_i - t_i - L)^+$pour s tel que $\sum_{j=i+1}^{s} t_j \leq L$
  A l'intérieur des intervalles définis par les θ_1^s, et au-delà de la plus grande de ces valeurs, les coûts sont continus et linéaires par morceaux. Lorsque θ_1 franchit une de ces valeurs, l'accroîssement du coût décroit (discontinuité).
- *Variation de $C^k(\theta_1, L, \theta_2)$ lorsque θ_2 varie*
  Cas de l'insertion d'une tâche simple, voir les chapitres 2 et 3.
- *Variation de $C^k(\theta_1, L, \theta_2)$ lorsque L varie*
  Supposons que nous commençions la première sous-tâche dès que a_i se termine. θ_1 et θ_2 sont supposés fixes. La fonction accroîssement du critère, ici fonction de L, sera linéaire et continue par morceaux. Chaque intervalle de continuité sera donné par les bornes suivantes :
  $l_0 = 0$
  $l_i = \max(\mu_{k+i} + t_{k+i} - \mu_k - t_k - \theta_1^*, \sum_{p=1}^{i} t_{k+p})$ pour $i = 1, 2, \ldots, n$
  $l_{n+1} = +\infty$
  Sur chacun de ces intervalles, la fonction accroîssement du critère sera monotone croissante, et si B est une borne de l'un de ces intervalles, nous aurons $\lim_{\epsilon \to 0}(B - \epsilon) \geq B$

L'algorithme temps-réel fait les mêmes préparatifs en différé que pour l'algorithme du chapitre 2. En temps réel, il effectue le calcul des variations de coût pour déterminer une solution bonne mais non optimale d'insertion (voir section 4.6)

Chapitre 5

Insertion d'une bitâche dans un ordonnancement de tâches simples

5.1 Introduction

Dans ce chapitre, nous développons d'autres approches heuristiques pour l'insertion d'une bitâche dans un ordonnancement préexistant de tâches simples, et voulons montrer la supériorité de la méthode consistant à utiliser la période de disponibilité de la bitâche pour y insérer des tâches simples déjà ordonnancées.

La section 5.2 pose le problème. Dans les sections 5.3 et 5.4, nous fournissons une expression des instants optimaux de départ des tâches de l'ordonnancement après insertion de la bitâche aléatoire. Un algorithme de placement optimal de la bitâche est proposé dans la section 5.5. Cet algorithme sera utilisé pour l'étalonnage des heuristiques proposées dans les sections suivantes. La section 5.6 propose une heuristique basée sur le remplacement de la bitâche aléatoire par une tâche simple. Une méthode de résolution optimale de ce problème est donnée dans le chapitre 3. Cette solution étant calculée, nous tirons partie de la période de disponobilité de la bitâche pour y insérer des tâches de l'ordonnancement existant. La seconde méthode proposée dans la section 5.7 est une simplification de l'algorithme optimal. Le paragraphe 5.8 est dédié à un exemple numérique. Des comparaisons sont faites entre les résultats fournis par les deux algorithmes proposés dans ce chapitre, ainsi qu'entre ces algorithmes et l'algorithme général (voir figure 3.8). La complexité des heuristiques est évaluée dans la section 5.9. Le paragraphe 5.10 est la conclusion. Un résumé clôt le chapitre.

5.2 Position du problème

Les notations utilisées dans ce chapitre sont les mêmes que celles définies dans le paragraphe 4.2.

Pour le cas d'une bitâche, et lorsque nous décidons d'insérer A_1 après a_k, il est possible que commencer à exécuter A_1 aussitôt que a_k se termine ne conduise pas à la solution optimale. La raison est que la position de la sous-tâche A_2 est contrainte par la position de la sous-tâche A_1 et que décaler A_1 légèrement peut permettre d'insérer une tâche supplémentaire entre la fin de A_1 et le début de A_2, ce qui peut conduire à une diminution significative du critère. Ainsi, nous introduisons z, instant de départ de A_1. Si nous voulons insérer A_1 après a_k, alors, nous devons avoir $z \geq \mu_k + t_k$.

Les formules donnant les nouveaux instants de départ μ_i', $i \in [k+1, k+r]$ pour les tâches décalées sont données dans la section 5.3

5.3 Une borne supérieure pour l'instant de départ de la nouvelle bitâche.

Résultat 9

Si nous décidons d'insérer A_1 après a_k, la solution optimale consiste à insérer A_1 à l'instant z tel que $\mu_k + t_k \leq z \leq z_1$ avec $z_1 = \min_r(\mu_{k+r} + t_{k+r} - \theta_1 - L, D - \theta_1 - L - \theta_2)$, où r est le plus grand entier tel que $\sum_{i=1} t_{k+i} \leq L$. Bien entendu, il est possible que $z_1 < \mu_k + t_k$. Dans ce cas, la bitâche ne peut être placée après a_k.

Preuve du résultat 9

1. $\mu_k + t_k \leq z$ est immédiat puisque A_1 est inséré après a_k.
2. Quand $z = z_1 = \mu_{k+r} + t_{k+r} - \theta_1 - L$, les tâches $a_{k+1}, a_{k+2}, \ldots, a_{k+r}$ sont comprises dans l'intervalle de temps L entre la fin de A_1 et le début de A_2, et il est impossible d'insérer une tâche supplémentaire dans cet intervalle de temps. De plus, si $z > z_1$, les tâches $a_{k+1}, a_{k+2}, \ldots, a_{k+r}$ sont décalées vers la droite au plus d'une période $z - z_1$, et les tâches $a_{k+r+1}, a_{k+r+2}, \ldots$ peuvent aussi être décalées, ce qui signifie que le critère du problème ne peut pas être plus petit que celui obtenu pour $z = z_1$.

3. Lorsque $z = z_1 = D - \theta_1 - L - \theta_2$, alors A_1 ne peut pas commencer après z_1, sinon le délai D serait violé.

Pour simplifier la représentation, nous continueront d'appeler r le nombre de tâches insérables entre A_1 et A_2 dans ce cas. Ceci conclut la preuve. □

Résultat 10

La valeur optimale pour z est une des valeurs $\{\mu_{k+i} + t_{k+i} - \theta_1 - L\}_{i=1,2,\ldots,r}$. Si $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$, alors $a_{k+1}, a_{k+2}, \ldots, a_{k+i}$ sont insérées entre A_1 et A_2, et les tâches $a_{k+i+1}, a_{k+i+2}, \ldots$ sont exécutées après A_2. Si $r = 0$, c'est-à-dire si $t_{k+1} > L$, alors $z = \mu_k + t_k$ est optimal.

Preuve du résultat 10

1. Si $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$, $i \in \{1, 2, \ldots, r\}$, alors a_{k+i} finit lorsque A_2 commence.

   Si $z \in (\mu_{k+i} + t_{k+i} - \theta_1 - L, \mu_{k+i+1} + t_{k+i+1} - \theta_1 - L)$, seules les tâches $\{a_{k+1}, \ldots, a_{k+i}\}$ seront ordonnancées entre A_1 et A_2, et l'augmentation du critère est une fonction croissante de z dans cet intervalle. En effet, lorsque z croît :

   - Les tâches situées dans la période de disponibilité ne peuvent qu'être retardées, et leur apport au critère ne fait que croître.
   - Les tâches qui commencent après la fin de la tâche A_2 subissent le même sort.

   Si $z = \mu_{k+i} + t_{k+i} - \theta_1 - L - \epsilon$, où ϵ est aussi petit que nous le voulons, seules les tâches $\{a_{k+1}, \ldots, a_{k+i-1}\}$ peuvent être ordonnancées entre A_1 et A_2. La tâche a_{k+i} ne peut donc commencer qu'après A_2 se termine, ce qui augmente son propre apport au critère par rapport au cas où $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$. De plus, le fait que a_{k+i} commence après que A_2 se termine retarde les autres tâches qui suivent A_2.

   Nous voyons donc que l'augmentation du critère est toujours plus faible lorsque $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$ que lorsque $z = \mu_{k+i} + t_{k+i} - \theta_1 - L - \epsilon$. Définissons $s_i = \mu_{k+i} + t_{k+i} - \theta_1 - L$. La figure 5.1 illustre l'augmentation du coût comme fonction de z.

2. Si $t_{k+1} > L$, a_{k+1} ne peut pas être inséré entre A_1 et A_2, et, de plus, le

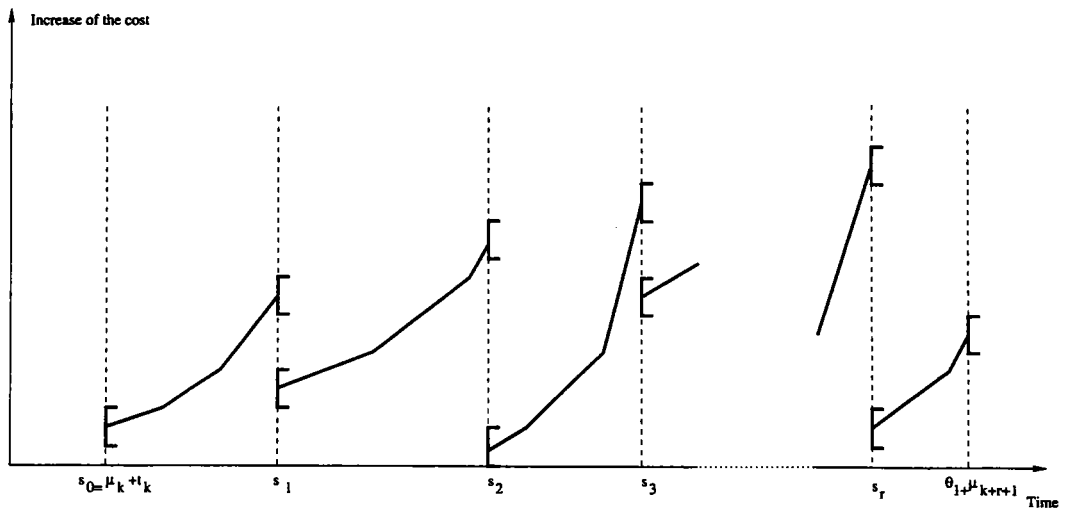


FIG. 5.1 – Augmentation du critère.

minimum du critère ne peut être obtenu que si A_1 commence aussitôt que possible, c'est-à-dire aussitôt que a_k termine.

Ceci conclut la preuve. □

5.4 Les nouveaux instants de départ

Nous considèrerons d'abord le cas des tâches insérées entre A_1 et A_2. Considérons que $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$, $i \in \{1, 2, \ldots, r\}$. En application du résultat 10, les tâches $\{a_{k+1}, \ldots, a_{k+i}\}$ seront insérés entre A_1 et A_2.

$$\begin{cases} \mu'_{k+1} = \max\{\mu_{k+1}, z + \theta_1\} \\ \mu'_{k+j} = \max\{\mu_{k+j}, \mu'_{k+j-1} + t_{k+j-1}\} \quad \text{if} \quad j = 2, \ldots, i-1 \\ \mu'_{k+i} = \mu_{k+i} \end{cases} \tag{5.1}$$

Considérons maintenant les tâches exécutées après A_2. Pour ces tâches, qui sont nommées a_{k+i+s} avec $s = 1, 2, \ldots$,

$$\begin{cases} \mu'_{k+i+1} = \max\{\mu_{k+i+1}, z + \theta_1 + L + \theta_2\} \\ \mu'_{k+i+s} = \max\{\mu_{k+i+s}, \mu'_{k+i+s-1} + t_{k+i+s-1}\} \quad \text{pour} \quad j = 2, \ldots \end{cases} \tag{5.2}$$

5.5 Algorithme de placement optimal

Afin d'obtenir la solution optimale, nous devons tester chaque instant possible d'insertion pour la nouvelle bitâche. Pour chaque tâche a_k, nous calculons

1. Calcul de k_{max} , index de la dernière tâche après laquelle on peut insérer A_1
2. Pour $k = 1$ à $k = k_{max}$
   (a) Calcul des nouveaux instants de début et surcoût lorsque nous insérons la bitâche juste après la tâche k.
   (b) Calcul du surcoût et des nouveaux instants de départ pour les tâches situées entre les deux sous-tâches de la bitâche.
   (c) Consigner l'index de la première tâche suivant la bitâche.
   (d) Calculer le surcoût et les nouveaux instants de départ pour les tâches situées après la bitâche.
   (e) Optimisation du critère (utilisation des résultats 9 et 10)
   (f) Si le surcoût obtenu lorsque la bitâche est insérée après a_k est le plus faible obtenu jusqu'à maintenant, alors nous le consignons.

FIG. 5.2 – *Algorithme de placement Optimal*

l'augmentation de critère si nous insérons A_1 aussitôt que la tâche a_k se termine. Ensuite nous vérifions les instants de départ calculés d'après le résultat 9. Cette étape nous permet d'optimiser le critère et d'insérer la bitâche aléatoire à un instant optimal. L'algorithme de placement optimal est décrit dans la figure 5.2. Une version plus détaillée de cet algorithme figure dans l'annexe B.

5.6 Premier algorithme heuristique : TR-Bitâche-1

Le premier algorithme temps-réel est inspiré de l'algorithme général décrit dans le chapitre 3. Cet algorithme nous permettait d'insérer une tâche simple dans un ordonnancement composé de n tâches simples.

Cet algorithme temps-réel est exécuté en deux parties :

- Une partie en différé, qui est exécutée pour préparer l'apparition éventuelle d'une bitâche aléatoire. Pour cela, on calcule les fonctions *augmentation de critère* $L_k(\theta)$ avec $k \in \{1, 2, \ldots, n\}$ (voir section 2.3). Les valeurs de θ sont comprises entre 0 et θ_{max}, où θ_{max} représente la durée maximale de la tâche à insérer.
- Une partie en temps-réel, qui effectue si possible l'insertion de la bitâche aléatoire. Elle se déroule en deux étapes :

  1. Nous considèrons d'abord une tâche simple, d'une durée égale à $\theta_1 + L + \theta_2$. L'instant optimal I pour l'insertion de cette tâche simple

dans l'ordonnancement courant est connu grâce aux calculs effectués sur la fonction accroîssement de critère (voir chapitre 3).

2. Nous insérons alors A_1 à l'instant I de l'ordonnancement actuel. Nous insérons A_2 à l'instant $I + \theta_1 + L$. La période L est utilisée pour exécuter le plus possible de tâches de l'ordonnancement initial dont les instants de départ étaient compris entre I et $I + \theta_1 + L$.

Cet algorithme est décrit dans la figure 5.3. Il consiste à travailler d'abord comme si la bitâche était une tâche simple, puis à faire passer un maximum de tâches dans la période de disponibilité.

1. Calculs hors ligne.
   (a) Calculer $L_k(\theta)$ pour $k = 1, \ldots, n$ et $\theta \in [0, +\infty)$ (voir 3.3)
2. Calculs en ligne.
   (a) Calculer le plus grand entier Q tel que $\mu_Q + t_Q + \theta_1 + L + \theta_2 \leq D$
   (b) Calculer $W_Q = \{k \setminus k \in \{1, 2, \ldots, Q\} \text{ et } \Delta_k > 0\} \cup \{Q\}$
   (c) Calculer l'augmentation $z_A(\theta_1 + L + \theta_2) = \min_{k \in W_Q} L_k(\theta_1 + L + \theta_2)$ Soit k^* la valeur de $k \in W_Q$ qui conduit au minimum d'augmentation (voir section 3.3).
   (d) Mettre l'ordonnancement à jour en utilisant les équations (5.1) et (5.2).

FIG. 5.3 – *Algorithme TR-Bitâche-1*

5.7 Second algorithme heuristique : TR-Bitâche-2

L'algorithme TR-Bitâche-2 est basé sur l'algorithme de placement optimal (voir paragraphe 5.4). Une expression résumée de cet algorithme est dans la figure 5.2, et une expression détaillée dans l'annexe B. Cet algorithme calcule d'abord quelle est l'augmentation de critère pour chaque point d'insertion possible de la bitâche aléatoire. Cela signifie qu'il calcule cette augmentation de critère lorsqu'on place la bitâche aléatoire directement après la fin d'une tâche de l'ordonnancement actuel, mais aussi si on laisse un temps disponible entre la fin d'une tâche de l'ordonnancement actuel et le début de la bitâche aléatoire. Cette dernière caractéristique permet d'avoir un résultat optimal quand au critère (voir résultats 9 et 10). Par contre, les calculs qui y sont associés augmentent considérablement la complexité de l'algorithme (voir 5.9). Aussi,

nous allons supprimer l'étape d'optimisation du placement de la bitâche aléatoire et ne prendre en compte que les points d'insertion situés juste après la fin des tâches de l'ordonnancement initial. Le point d'insertion retenu sera celui menant à la plus faible augmentation du critère. Le code de cet algorithme est le même que celui de l'algorithme de placement optimal (voir figure 5.2), excepté l'étape (e) qui est supprimée. Dans la représentation détaillée de l'algorithme de placement optimal donnée dans l'annexe B, cela correspond à la supression de l'étape 2e.

5.8 Exemples numériques

Dans cette section, nous comparons les performances des heuristiques TR-Bitâche-1,TR-Bitâche-2 et l'Algorithme Général (voir 3.8) en termes de qualité des résultats. Une comparaison est effectuée avec les résultats optimaux fournis par l'Algorithme de placement Optimal (voir 5.5).

Nous commençons par fournir des informations sur le protocole de génération des jeux de test, avant de fournir les résultats numériques.

Nous générons un ordonnancement au hasard en suivant les règles suivantes :

- Les temps d'exécution des tâches sont générées au hasard comme suit :
  - L'intervalle [1,8] est sélectionné avec une probabilité de 0.3.
  - L'intervalle [9,13] est sélectionné avec une probabilité de 0.2.
  - L'intervalle [14,20] est sélectionné avec une probabilité de 0.5.

  Aussitôt que l'intervalle est sélectionné, le temps d'exécution est choisi au hasard dans cet intervalle.
- Chaque tâche est suivie par une période de temps non utilisé dont la durée est générée comme suit :
  - 0 avec une probabilité de 0.5.
  - choisi au hasard dans l'intervalle [1,20] avec une probabilité de 0.5.
- Le délai de la tâche est sélectionné au hasard dans l'intervalle $[z-1, z+10]$ où z est l'instant de fin d'exécution de la tâche.

Cet ordonnancement est donné dans la table 5.1

Nous tentons d'insérer des bitâches dans cet ordonnancement. Ces bitâches sont générées de façon aléatoires. θ_1, L, θ_2 sont générées respectivement sur

les intervalles de valeurs [1,30], [1,50] et [1,20], en utilisant une répartition uniforme. Le délai $D = 600$ est le même pour toutes les bitâches.

Nous considérons 10 cas. Ils sont décrits dans le tableau 5.2

Nous appliquons les algorithmes TR-Bitâche-1, TR-Bitâche-2, Général et Optimal dans chacun de ces cas. Les résultats sont présentés dans le tableau 5.4.

5.9 Complexité

Dans cette section, nous calculons la complexité de chacun des algorithmes proposés. Nous supposons que chaque évènement a la même complexité. Nous considérons le pire cas. Nous supposons que l'ordonnancement dans lequel nous voulons insérer la bitâche aléatoire comporte n tâches.

5.9.1 Complexité de l'algorithme général

Cette complexité est évaluée dans le paragraphe 3.3.5. Nous rappelons les résultats pour mémoire :

- Partie en différé : complexité en $O(n^3)$.
- Partie en temps-réel : complexité en $O(n)$.

5.9.2 Complexité de l'algorithme de placement Optimal

Il n'y a pas de complexité hors-ligne à évaluer. La complexité de l'algorithme de placement Optimal va être évaluée pas à pas.

- Etape 1 : 3
- Etape 2 : 10n+8
- Etape 3 : 3
- Etape 4 : 32
- Etape 5 : 3
- Etape 6 : 29
- Etape 7 : 8+(n-k)(60+20n)
- Etape 8 : 2n+10

Nous n'évaluons que la complexité basique de chaque étape. Le nombre de fois où chaque étape est exécutée est fonction du cas concerné. Ici, le pire cas est obtenu lorsque il n'y a aucune tâche qui puisse être insérée entre les deux

TAB. 5.1 – Exemple d'ordonnancement

Tâche #i	Instant de départ	Temps d'exéc.	Délai	Temps libre
1	0	17	11	0
2	17	8	32	0
3	25	20	41	0
4	45	13	53	0
5	58	15	83	0
6	73	8	76	0
7	81	17	105	0
8	98	14	122	19
9	131	17	158	3
10	151	13	165	8
11	172	13	194	0
12	185	8	186	0
13	193	20	220	14
14	227	13	236	0
15	240	7	257	15
16	262	20	278	15
17	297	13	305	19
18	329	10	344	0
19	339	8	337	0
20	347	20	377	15
21	382	15	407	0
22	397	8	415	0
23	405	2	403	19
24	426	8	424	18
25	452	13	468	0
26	465	20	495	0
27	485	20	514	15
28	520	3	531	0
29	523	13	543	0
30	536	15	547	11
31	562	20	592	19
32	601	12	606	15
33	628	8	639	1
34	637	8	643	8
35	653	19	672	19
36	691	20	707	0
37	711	13	717	10
38	734	17	742	14
39	765	15	781	0
40	780	13	794	16
41	809	9	825	3
42	821	20	836	15
43	856	12	875	19
44	887	15	903	0
45	902	13	912	18
46	933	5	939	0
47	938	13	952	0
48	951	12	973	0
49	963	13	976	0
50	976	8	994	0

TAB. 5.2 – Les paramètres des bitâches.

indice de la bitâche i	θ_1	θ_2	L
1	30	19	13
2	30	9	55
3	1	17	36
4	5	12	41
5	30	12	10
6	6	12	31
7	22	11	9
8	26	12	16
9	30	10	41
10	14	10	47
11	30	1	23

sous-tâches de la bitâche, et que le temps d'exécution de toutes les tâches à exécuter après a_k est inférieur ou égal à L.

Ainsi la complexité sera :

$$\begin{aligned} C^2_{on} &= 3 + \sum_{i=1}^{n-1} 32 + 3 + \sum_{k=1}^{n} (8 + (n-k)(60 + 20n)) + 2n + 10 \\ &= 34n^2 - 6n + 11 + 30n^3 + 20n^2 + 38n \\ &= 30n^3 + 54n^2 + 32n + 11 \end{aligned}$$

En résumé :

- Partie en différé : 0.
- Partie en temps-réel : complexité en $O(n^3)$.

5.9.3 Complexité de l'algorithme TR-Bitâche-1

Cet algorithme a la même complexité hors-ligne que l'algorithme Général qui est de : $C^3_{off} = C^1_{off} \leq 3n^2$

La complexité en ligne est donnée par la complexité des étapes 2.a à 2.c de la partie en ligne de l'algorithme général, c'est-à-dire $13n$. Puis, nous exécutons les étapes 3,4,5,6 et 8 de l'algorithme de placement Optimal pour une valeur unique de k. La complexité obtenue est de $3 + \sum_{i=1}^{n-1} 32 + 3 + 2n + 10 = 34n - 19$

Et, finalement $C^3_{on} = 13n + 34n - 19 = 47n - 19 \leq 47n$

En résumé :

- Partie en différé : complexité en $O(n^2)$.
- Partie en temps-réel : complexité en $O(n)$.

5.9.4 Complexité de l'algorithme TR-Bitâche-2

Il n'y a pas de complexité hors-ligne à évaluer. L'évaluation de la complexité en ligne est donnée dans la section 5.9.2. L'étape 7 de l'algorithme doit cependant être ignorée.

$$C^4_{on} = 3 + 10n + 8 + \sum_{k=1}^{n}(3 + \sum_{i=1}^{n-k} 32 + 3 + 2n + 10)) = 34n^2 - 6n + 11$$

En résumé :

- Partie en différé : complexité en 0.
- Partie en temps-réel : complexité en $O(n^2)$.

Un résumé des complexité obtenues est donné dans la table 5.3

TAB. 5.3 – Complexité des algorithmes.

Algorithme	Complexité *Hors-line*	Complexité *En-line*
Général	$O(n^3)$	$O(n)$
Optimal	0	$O(n^3)$
TR-Bitâche-1	$O(n^2)$	$O(n)$
TR-Bitâche-2	0	$O(n^2)$

5.10 Conclusions

Les conclusions de ce chapitre sont les suivantes :

- L'Algorithme TR-Bitâche-1 fournit une valeur de critère qui est toujours significativement meilleure que celle fournie par l'algorithme Général. Cela signifie qu'il est toujours plus avantageux de tirer partie du temps libre entre les deux sous-tâches de la bitâche. La méthode permettant d'exploiter la période de disponibilité est donc meilleure que celle consistant à approximer en ne considérant que des insertions de tâches simples.
- Les résultats donnés par l'Algorithme TR-Bitâche-2 sont légèrement meilleurs que ceux donnés par l'Algorithme TR-Bitâche-1.

Un bon compromis entre la complexité et la qualité de la solution est l'Algorithme TR-Bitâche-1.

Le sujet du chapitre suivant va être la recherche d'une méthode efficace pour effectuer l'insertion d'une bitâche dans un ordonnancement existant de bitâches.

TAB. 5.4 – Résultats

Ident. bitâche	Algorithme Général			Algorithme de placement Optimal		
	Insérée après	Instant de départ	Critère	Insérée après	Instant de départ	Critère
1	15	247	103	23	407	42
2	15	247	344	8	112	14
3	23	407	66	6	81	0
4	15	247	84	12	194	0
5	23	407	56	23	407	15
6	23	407	44	6	81	0
7	23	407	25	23	407	3
8	23	407	66	23	407	13
9	15	247	229	15	247	15
10	15	247	155	17	310	0
11	23	407	66	23	412	16

Ident. bitâche	Algorithme TR-Bitâche-1			Algorithme TR-Bitâche-2		
	Insérée après	Instant de départ	Critère	Insérée après	Instant de départ	Critère
1	15	247	103	23	407	42
2	15	247	36	8	112	14
3	23	407	6	6	81	0
4	15	247	8	20	367	0
5	23	407	15	23	407	15
6	23	407	1	6	81	0
7	23	407	3	23	407	3
8	23	407	13	23	407	13
9	15	247	15	15	247	15
10	15	247	23	17	310	0
11	23	407	17	23	407	17

Résumé

Contrairement aux tâches simples, placer une bitâche juste après la fin d'une tâche de l'ordonnancement peut ne pas conduire à un résultat optimal au niveau du critère (voir résultats 9 et 10).
Tenant compte de cette remarque un algorithme de placement optimal est fourni pour l'insertion d'une bitâche dans un ordonnancement de tâches simples (voir section 5.5). Cet algorithme sert de référence mais sa complexité est trop élevée pour qu'il soit utilisable en temps réel.
Deux algorithmes heuristiques sont proposés pour l'approche temps-réel :

- Le premier algorithme commence par utiliser l'approche tâche unique de la première partie de la thèse en tentant l'insertion d'une tâche de longueur $\theta_1 + L + \theta_2$. Une fois l'emplacement d'insertion optimal connu, on tient compte de l'existence de la période L en y insérant le plus possible de tâches de l'ordonnancement initial.
- Le second algorithme est une simplification de l'algorithme de placement optimal : on n'optimise pas le critère suivant l'approche donnée dans les résultats 9 et 10.

Une comparaison entre les algorithmes proposés dans ce chapitre et un algorithme effectuant l'insertion d'une tâche simple (voir section 3.8) démontre la supériorité de l'approche permettant d'exploiter la période disponibilité par rapport à celle qui consiste à approximer les bitâches par des tâches simples.

Chapitre 6

Insertion d'une bitâche dans un ordonnancement constitué de bitâches.

6.1 Introduction

Dans ce chapitre, nous allons insérer une bitâche, dont les caractéristiques ne sont connues qu'au moment de son apparition, dans un ordonnancement existant de bitâches. Cette bitâche sera appelée bitâche aléatoire.

Nous considérons cette bitâche comme composée de deux tâches de même durée. Cette hypothèse correspond à la réalité des radars qui sont à l'origine de ce travail. Comme toujours cette bitâche possède un délai impératif qui ne peut être violé. Le critère de cette optimisation reste la minimisation de la somme des retards. Ce problème sera bien sûr abordé sous l'angle du temps réel.

Nous allons utiliser la technique de l'entrelacement de tâches pour effectuer cette insertion.

Le problème est présenté dans la section 6.2. La stratégie temps-réel est proposée dans les sections 6.3 et 6.4. La section 6.5 donne une approche permettant une résolution plus rapide de la partie en différé des calculs. Une évaluation de la complexité des deux parties de l'algorithme proposé est donnée dans la section 6.6. La section 6.7 est dédiée à un exemple numérique. La section 6.8 est la conclusion. Elle est suivie d'un résumé du chapitre.

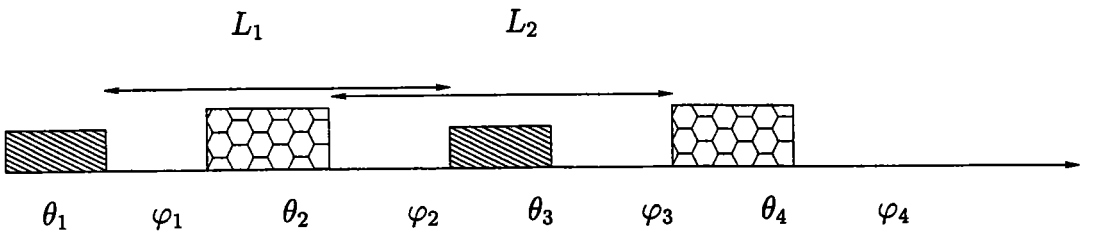


FIG. 6.1 – Exemple d'ordonnancement 1

6.2 Position du problème

Nous conservons la modélisation des bitâches définie dans le paragraphe 4.2.

Nous considérons toujours que les tâches sont non-préemptives. Dans ce chapitre, nous considèrerons que le temps d'exécution des deux tâches de la même bitâche est le même et est représenté par θ. Les deux tâches sont séparées par une période de temps (période de disponibilité) notée L durant laquelle la ressource est disponible pour l'exécution d'autres bitâches. La bitâche doit être exécutée avant son délai impératif D.

L'ordonnancement dans lequel nous désirons insérer la nouvelle bitâche est constitué de bitâches ayant déjà été ordonnancées de façon à commencer leur exécution le plus tôt possible. La plupart du temps, deux tâches consécutives ne font pas partie de la même bitâche dans l'ordonnancement donné.

Il est possible de commencer l'exécution des bitâches à des instants ultérieurs à ceux prévus initialement, afin de pouvoir en insérer une nouvelle, mais celà à condition de ne pas changer l'ordre d'exécution ni la période L qui lie les deux tâches d'une bitâche.

Aussi, afin d'alléger la forme des équations décrivant le problème, nous appellerons θ_s le temps d'exécution de la $s^{ième}$ tâche devant être exécutée suivant l'ordonnancement. De la même façon, nous appellerons φ_s le $s^{ième}$ temps libre de l'ordonnancement. φ_s peut être nul. Comme les bitâches peuvent être entrelacées, une période de temps libre ne correspond généralement pas à une période de temps libre séparant les deux tâches d'une même bitâche. L_i sera la durée du temps libre correspondant à la $i^{ième}$ bitâche. La $i^{ième}$ bitâche est celle dont la première tâche est la $i^{ième}$ de l'ordonnancement. Cet ordre n'est généralement pas celui de la deuxième tâche des mêmes bitâches. Deux illustrations de ces notations sont données dans les figures 6.1 et 6.2.

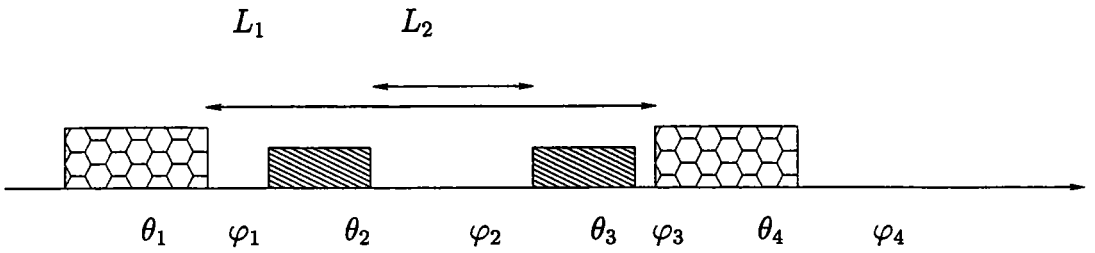


FIG. 6.2 – Exemple d'ordonnancement 2

D'après les hypothèses données ci-dessus, $\theta_1 = \theta_3$ et $\theta_2 = \theta_4$ pour la Figure 6.1. Dans le cas de l'exemple de la Figure 6.2, $\theta_1 = \theta_4$ et $\theta_2 = \theta_3$.

Le temps disponible L_i est donné par la somme des temps disponibles et opératoires situés entre la fin de la première tâche et le début de la seconde tâche de la bitâche i. Soit k_i l'index de la première tâche de la bitâche i et l_i l'index de la dernière tâche de la bitâche i. La numérotation est faite en fonction de l'ordre d'apparition de la tâche dans l'ordonnancement considéré. La contrainte liant les deux tâches de la bitâche i peut s'écrire :

$$\sum_{s=k_i}^{l_i-1} \varphi_s + \sum_{s=k_i+1}^{l_i-1} \theta_s = L_i, \ i = 1, \ldots, n \tag{6.1}$$

où n est le nombre de bitâches concernées de l'ordonnancement préexistant.

Le lecteur remarquera que L_i et θ_i sont donnés. Aussi, les seules variables de notre problème seront les φ_i.

Par exemple, dans la figure 6.1 :

$$L_1 = \varphi_1 + \theta_2 + \varphi_2$$
$$L_2 = \varphi_2 + \theta_3 + \varphi_3$$

Pareillement, dans la figure 6.2 :

$$L_1 = \varphi_1 + \theta_2 + \varphi_2 + \theta_3 + \varphi_3$$
$$L_2 = \varphi_2$$

La relation (6.1) peut être réécrite comme suit :

$$\sum_{s=k_i}^{l_i-1} \varphi_s = L_i - \sum_{s=k_i+1}^{l_i-1} \theta_s, \ i = 1, \ldots, n \tag{6.2}$$

Un autre ensemble de contraintes est introduit dans notre problème, de façon à garantir que l'instant de commencement d'une bitâche ne soit pas antérieur à son instant de commencement dans l'ordonnancement initial. Soit E l'ensemble des index correspondants aux premières sous-tâches des bitâches,

excepté pour le premier car la première bitâche est supposée commencer à l'instant 0. Soit φ_i^* les valeurs initiales des périodes de temps disponible dans l'ordonnancement initial. Les contraintes suivantes doivent être satisfaites :

$$\sum_{j=1}^{i-1} \varphi_j \geq \sum_{j=1}^{i-1} \varphi_j^* \text{ pour } i \in E \tag{6.3}$$

Nous introduisons une variable positive ρ_i, appelée variable d'écart dans chacune des inéquations (6.3) de façon à pouvoir les exprimer sous forme d'équations.

$$\sum_{j=1}^{i-1} \varphi_j - \rho_i = \sum_{j=1}^{i-1} \varphi_j^* \text{ pour } i \in E \tag{6.4}$$

Ainsi, toutes les contraintes relatives à un ordonnancement donné de n bitâches peut être exprimé sous la forme d'un système d'au plus $2n-1$ équations et $3n-2$ variables, en tenant compte des variables d'écart. Ce système inclut à la fois les équations (6.2) et (6.4).

La façon traditionnelle de représenter ce genre de système est une équation matricielle, représentée en (6.5) :

$$A.\Phi = K \tag{6.5}$$

où A est une matrice de $2n-1$ lignes et $3n-2$ colonnes, incluant ρ_i, $i \in \{2,\ldots,n\}$. Φ est une matrice-colonne de $3n-2$ éléments. Φ contient les variables du système. La matrice K est une matrice-colonne de $2n-1$ éléments. Cette dernière matrice est constante et ses éléments sont calculés à partir des membres gauches des équations (6.2) et (6.4).

Par exemple, dans le cas de la figure 6.1 :

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix} \qquad \Phi = \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \rho_1 \end{bmatrix} \qquad K = \begin{bmatrix} L_1 - \theta_2 \\ L_2 - \theta_3 \\ \varphi_1^* \end{bmatrix}$$

Et, dans le cas de la figure 6.2 :

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix} \qquad \Phi = \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \rho_1 \end{bmatrix} \qquad K = \begin{bmatrix} L_1 - \theta_2 - \theta_3 \\ L_2 \\ \varphi_1^* \end{bmatrix}$$

6.3 Calculs en différé pour l'ordonnancement.

Supposons que nous tentions d'insérer la première sous-tâche de la bitâche aléatoire dans la période d'inactivité d'indice k, et la seconde sous-tâche dans la période d'inactivité indicée l. Considérons de même que $k < l$. Dans ce cas le problème à résoudre peut s'écrire comme suit :

$$Max[Min(\varphi_k, \varphi_l) - \lambda \sum_{s=k+1}^{l-1} \varphi_s], \lambda \in (0,1) \tag{6.6}$$

sous les contraintes (6.5) et :

$$Min(\varphi_k, \varphi_l) \leq H \tag{6.7}$$

où H est une borne supérieure du temps de travail de la sous-tâche.

L'explication de cette formulation (6.6) est double :

- Ajuster les périodes φ_k et φ_l de façon à ce que leurs valeurs minimales soient aussi grandes que possible, mais aussi plus petites ou égales à H conformément à (6.7). L'objectif est de disposer d'un panel aussi large que possible de temps opératoires pour les tâches qui peuvent être insérées dans les fenêtres de temps d'inactivité d'indices k et l.
- Réduire autant que possible la différence entre la borne supérieure de la période d'inactivité φ_k et la borne inférieure de la période d'inactivité φ_l, de façon à augmenter le panel des valeurs possibles des périodes d'inactivité des bitâches pouvant être insérées.

Ce problème peut être transformé en un problème linéaire en introduisant une variable additionnelle :

$$Max[Z - \lambda \sum_{s=k+1}^{l-1} \varphi_s], \lambda \in (0,1) \tag{6.8}$$

sous les contraintes (6.5) et (6.7) et :

$$Z \leq \varphi_k \tag{6.9}$$

$$Z \leq \varphi_l \tag{6.10}$$

Si le but est d'insérer la bitâche complète dans la période d'inactivité d'index s, le problème devient :

$$\text{Max } \varphi_s$$

sous les contraintes (6.5) et :

$$\varphi_s \leq H^* \tag{6.11}$$

où H^* est la valeur maximale possible de $2\theta + L$. Remarquons bien que pour ce type de problèmes, résolus en temps différé, c'est-à-dire avant que la bitâche n'apparaîsse, nous devons chercher la solution qui offre le maximum de possibilités. Le but va être de sélectionner en temps-réel une solution qui permet non seulement à la bitâche aléatoire d'être exécutée avant son délai impératif D, mais aussi qui mène à la plus faible augmentation de critère.

Les solutions doivent être calculées pour chaque paire (k, l), $k \leq l$ de temps disponibles. Les solutions obtenues sont l'énumération des temps disponibles $S = \{\varphi_s\}_{s=1,2,\ldots,2n-1}$.

Deux cas sont possibles : $k = l$ et $k < l$. Les données qui nous sont nécessaires pour les calculs sont les mêmes dans les deux cas.

- Premier cas : si $k = l$, alors la bitâche est contenue dans une seule et unique période de temps disponible, aussi n'avons nous à stocker que k, S et l'augmentation de critère. Ensuite, nous classons les données de ces résultats suivant l'ordre croissant des $\varphi_s, s = 1, \ldots, 2n-1$. Cet ensemble de solutions est appelé E_1.
- Second cas : si $k < l$ nous stockons :

  1. La paire (k, l)
  2. Z
  3. $a = \sum_{s=k+1}^{l-1} \varphi_s + \sum_{s=k+1}^{l} \theta_s$
  4. $b = \sum_{s=k}^{l} \varphi_s + \sum_{s=k+1}^{l} \theta_s - 2Z$
  5. L'augmentation du critère.

  Les résultats sont alors classés par rapport aux valeurs croissantes de Z. Il se peut que plusieurs solutions mènent aux mêmes valeurs de Z. Nous classons alors ces solutions suivant les valeurs croissantes de a.
  Enfin, pour les solutions menant aux mêmes valeurs de Z et de a, sont stockées suivant l'ordre croissant des b. Cet ensemble de solutions est appelé E_2.

Remarque : Notons qu'il est possible de résoudre ce problème pour une série $H_1, H_2, \ldots, H_M$ de valeurs de H. Nous pourrons ainsi obtenir la solution la mieux adaptée à la bitâche aléatoire qui se présentera.

6.4 Partie en ligne de l'ordonnancement.

Nous supposons que la bitâche définie par (L^*, θ^*, D^*) apparaît dans le système :

Première étape

Commençons par considérer l'ensemble des solutions E_1 et sélectionnons les solutions telles que :

1. $\varphi_k \geq 2\theta^* + L^*$
2. L'augmentation du critère est la plus petite d'entre toutes les solutions telles que la borne supérieure de la fenêtre k soit inférieure ou égale à D^*. Soit S^1 la solution et W^1 l'augmentation de critère correspondante.

Seconde étape

Nous ne considérons que l'ensemble des solutions E_2, et y sélectionnons la solution S_2 constituée de la paire (k^*, l^*) de façon à ce que :

1. $\theta^* \leq Z$
2. $a \leq L^* \leq b$
3. L'augmentation du critère est la plus faible entre toutes les solutions constituant les paires (k, l) telles que la borne supérieure de la fenêtre l soit inférieure ou égale à D^*.

Troisième étape

Si $W^1 < W^2$ alors S^1 est la solution que nous retenons pour ce problème, sinon S^2 est solution.

Nous voyons que le choix de H est crucial. Une valeur importante de H pourra conduire à un critère de mauvaise qualité. Une valeur trop faible de H conduira, par contre, a peu de solutions admissibles en moyenne. C'est la raison pour laquelle nous avons suggéré, plus haut, de faire les mêmes calculs pour des séries de valeurs de H (ou de H^*). Rappelons que ces calculs se déroulent en différé, et donc que le temps de calcul n'est pas d'une grande importance. En outre, la nécessité de trouver une solution qui permet de terminer la tâche

aléatoire avant le délai D réduit considérablement le volume des calculs à effectuer.

6.5 Parallélisme

Comme défini dans la section 6.4, la partie en différé de l'ordonnancement consiste principalement en la résolution d'un ensemble (généralement limité) de problèmes de programmation linéaire (PL en abrégé). Pour plus d'informations sur la programmation linéaire, voir [41]. Puisque $2n$ est le nombre d'intervalles disponibles, alors le nombre Z de problèmes PL à résoudre est $Z = 2n + C_{2n}^2 = n(2n+1)$. Le lecteur n'aura pas oublié que l'on cherche à insérer la bitâche aléatoire dans un ou deux intervalles de temps disponible. Bien entendu, la présence du délai D réduit considérabement le volume des calculs à effectuer.

De plus, remarquons que ces problèmes sont indépendants les uns des autres : chacun d'entre eux peut être résolu sans avoir à tenir compte des autres. En d'autres termes, il est intéressant (en terme de temps opératoires) de résoudre ces problèmes de façon concurrente, par le biais de la programmation parallèle, et plus précisément d'une technique $SPMD$ (pour *Single Program Multiple Data)*. Autrement dit, nous écrivons un seul programme dont l'application se fera sur m jeux de données. L'idée est de répartir la charge de travail entre m processeurs, de telle façon que la charge de travail de chacun de ces processeurs soit de $\frac{Z}{m}$. Le lecteur en conclut que notre but est de diviser le temps opératoire total par m.

Nous considérons que chaque processeur a un identifiant unique appelé Id. Dans la suite de ce chapitre, nous allons considérer que nous disposons d'un ensemble de m processeurs, et que leur Id sont numérotés de 1 à m. Chaque processeur exécute un processus unique, de sa création à sa mort, sans aucune interruption possible. Dans le reste de ce chapitre, nous identifierons un processus par l'Id du processeur sur lequel il est exécuté.

Ici, tous les processus sont les mêmes (l'algorithme du Simplexe, mais les problèmes PL que chaque processus aura à résoudre seront différents), excepté le cas du processus 1, dont nous parlerons plus loin. Nous décrivons quels sont les calculs précis que chaque processus va devoir exécuter dans le paragraphe suivant.

TAB. 6.1 – Identification des problèmes PL

Index des problèmes PL	Index de φ_k	Index de φ_l
1	1	1
2	1	2
3	1	3
⋮		
$2n$	1	$2n$
$2n+1$	2	2
⋮		
$n(2n+1)$	$2n$	$2n$

Le lecteur aura remarqué qu'un problème n'a pas été abordé : nous n'avons pas parlé de la gestion des données que chacun des processus requiert afin de pouvoir résoudre les problèmes PL (i.e. φ_k et θ_k,$k \in \{1,\ldots,2n\}$), ni de comment les résultats de ces calculs deviennent disponibles pour l'exécution de la partie en ligne de l'ordonnancement. Un des processus, que nous appelons processus maître (par convention le processus 1) va être en charge de la gestion des flots d'entrée et de sortie des données pour l'ensemble des m processus. En premier lieu, le processus maître encode les données fondamentales du problème : φ_k et θ_k, $k \in \{1,\ldots,2n\}$, et les communique ensuite à chacun des autres processus. Ces données représentent l'ordonnancement actuel. Après quoi le processus maître attend que les autres processus mènent à terme leurs calculs, et lui communiquent leurs résultats. Le processus maître stocke alors ces résultats suivant la méthode expliquée dans le paragraphe 6.3. Entre l'action d'envoyer les données de base à chaque processus, et la réception des solutions, le processus maître n'exécute pas d'activité. Aussi, il serait judicieux d'exploiter ce temps disponible pour résoudre une partie des problèmes PL.

Un autre problème consiste à décider quels sont les problèmes PL que chaque processus va avoir à résoudre. Fondamentalement, il est évident qu'il est optimal de donner la même quantité de travail à chaque processus, pour éviter à certains processus d'être sur ou sous chargés de travail par rapport aux autres, dans le but de minimiser le temps de travail. C'est ce que l'on appelle l'équilibrage des charges.

Comme nous ne pouvons pas prédire le temps de travail requis par la résolution de l'un des problèmes PL, nous considèrerons qu'ils consomment tous

1. Si $Id = 1$ alors initialiser le système
2. Si $Id = 1$ alors
   (a) Encoder les données (θ et φ)
   (b) Envoyer ces données codées aux processus 1 à m
3. Pour tous les processus :
   (a) Recevoir les données communiquées par le processus 1.
   (b) Calculer quels sont les indices des programmes PL qui doivent être résolus par le processus, en fonction de Id. L'ensemble de ces indices est donné par la formule : $[(\lfloor \frac{n(2n+1)}{m} \rfloor) * (Id - 1) + 1, (\lfloor \frac{n(2n+1)}{m} \rfloor) * Id]$

      La signification de ces indices est donnée dans la table 6.1
   (c) Résoudre les problèmes PL dont les indices sont donnés dans 3b
   (d) Encoder et envoyer les résultats aux processus 1.
4. Si $Id = 1$ alors
   (a) Recevoir les données des résultats communiquées par les processus 1 à m. Les classer comme décrit dans les sections 6.3 et 6.4
   (b) Exécuter la partie en temps-réel des calculs et insérer la bitâche aléatoire.

FIG. 6.3 – Code du programme

le même temps de travail. Aussi, nous allons faire exécuter $\frac{Z}{m}$ problèmes PL à chaque processus. Nous affectons à chaque problème PL un identificateur unique, ces identificateurs sont donnés dans le tableau 6.1.

Les indices des programmes PL qu'un processus doit résoudre sont fonction de l'Id du processus : par exemple, le processus 1 résoudra les problèmes indexés de 1 à $\lfloor \frac{n(2n+1)}{m} \rfloor$, le processus 2 résoudra les problèmes PL indicés de $\lfloor \frac{n(2n+1)}{m} \rfloor + 1$ à $\lfloor \frac{2n(2n+1)}{m} \rfloor$, et ainsi de suite. Comme nous l'avons dit précédemment, le même programme (algorithme du simplexe et gestion des données) est envoyé à chaque processus, mais seul le processus maître exécutera la partie gestion des données. La figure 6.3 donne un algorithme de ce programme.

Maintenant, le modèle que nous avons décrit n'est pas suffisamment raffiné pour gérer le problème réel des radars : il nous faut en effet faire face à l'arrivée successive de plusieurs bitâches aléatoires. Le modèle approprié serait alors le suivant :

1. Le processus maître initialise le système : il y a exécution de la partie en différé de l'ordonnancement. Chaque processus dont l'Id est compris entre 1 et m résout les problèmes avec lesquels il est lié. Les résultats des calculs sont envoyés au processus maître, qui les classe puis les stocke. Les calculs en différé sont finis lorsque le processus maître a reçu et fini de traiter les solutions de tous les problèmes.

TAB. 6.2 – *Id* du proc. calculant une solution du problème PL associé avec φ_k, et φ_l

$\varphi_k \setminus \varphi_l$	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8
φ_1	1	1	1	1	1	1	1	1
φ_2		1	2	2	2	2	2	2
φ_3			2	2	2	3	3	3
φ_4				3	3	3	3	3
φ_5					3	4	4	4
φ_6						4	4	4
φ_7							4	4
φ_8								4

2. Lorsque une bitâche aléatoire A surgit dans le système[1], le processus maître en utilisant les solutions calculées précédemment lors de l'étape 1., effectue l'insertion menant vers l'augmentation la plus faible du critère (le lecteur notera que les processeurs 2 à m ne son pas sollicités durant cette phase). Lorsque l'insertion est effectuée, les données servant à effectuer une insertion ne sont plus à jour, et il faut donc recommencer l'étape 1.

En pratique, la programmation des communication entre les processeurs sera prise en charge par un logiciel de communications comme par exemple *MPI* (Message Passing Interface).

Exemple : Considérons un ordinateur équipé de $m = 4$ processeurs. Nous supposons qu'il y a actuellement $n = 4$ bitâches ordonnancées. Le programme donné dans la figure 6.3 est exécuté sur les quatre processeurs. Le processeur 1 initie un calcul en différé de l'ordonnancement. La table 6.2 fournit l'identificateur du processeur devant résoudre le problème PL correspondant au cas φ_k, et φ_l, $k, l \in \{1, \ldots, 2n\}$, et $l \geq k$. Après complétion de la réception par le processus maître des résultats en provenance des quatre processeurs, ce processus passe en mode d'attente de l'apparition d'une bitâche aléatoire.

Lorsque une bitâche aléatoire apparaît, le premier processeur exécute la partie en temps-réel de l'ordonnancement afin d'insérer la bitâche, à l'emplacement conduisant à la plus faible augmentation du critère.

[1]Si une bitâche aléatoire arrive sur le système, alors que tous les calculs en différé ne sont pas effectués, on préfèrera effectuer une insertion qui ne sera pas forcément optimale en utilisant les résultats partiels, plutôt que pas d'insertion du tout.

6.6 Complexité

Cette section est divisée en deux parties. La première partie est dédiée au calcul de la complexité de la partie en différé de l'ordonnancement. La seconde partie traite de la partie temps-réel de l'ordonnancement.

Nous considérons que l'ordonnancement dans lequel nous allons tenter d'insérer la bitâche est constitué de n bitâches.

La partie en différé de l'ordonnancement est constituée de deux parties : la première partie concerne la construction des problèmes PL, et la seconde concerne la résolution desdits problèmes en utilisant l'algorithme du simplexe. Cet algorithme est semi-calculable (en d'autres termes, il n'y a pas de moyen de prédire avant le calcul si celui-ci va finir ou pas), mais dans la pratique, il est connu pour s'exécuter en temps polynomial.

Maintenant, si nous considérons l'amélioration basée sur le parallélisme de cet algorithme pour la résolution des problèmes PL, la complexité totale de la partie en différé des calculs reste la même (en fait elle augmente légèrement du fait des communications entre les processus). Par contre, la complexité des calculs à effectuer sur chaque processeur est la complexité totale, divisée par le nombre de processeurs.

La partie en temps-réel de l'ordonnancement est divisée en trois étapes. La première étape consiste à rechercher de la meilleure solution lorsque on veut insérer la bitâche aléatoire dans une seule et unique période de disponibilité de l'ordonnancement actuel. $2n$ solutions possibles sont examinées. Dans le pire des cas, chaque solution que nous testons est meilleure (au sens du critère) que la précédente, aussi aurons nous trois opérations à exécuter pour chaque solution. La complexité de la première étape est donc $6n$. La seconde étape consiste à chercher la meilleure solution lorsque nous voulons insérer les deux tâches de la bitâche aléatoire dans deux temps d'inactivité distinctes de l'ordonnancement initial. Il y a $n(2n-1)-2n = 2n^2-3n$ possibilités à prendre en compte. Le pire cas se présente là aussi lorsque chaque solution examinée est meilleure que la précédente (au vu du critère). Chaque solution nécessite alors quatre opérations de traitement. La complexité de la seconde étape est alors de $8n^2 - 12n$. La dernière étape est la comparaison des valeurs des critères des deux solutions retenues lors des deux premières étapes, et coûte deux opé-

rations. La complexité totale de la partie en temps-réel de l'ordonnancement sera alors au plus de $8n^2 - 6n + 2$. En d'autres termes, la complexité est en $O(n^2)$.

6.7 Un exemple numérique.

Dans cette section, nous allons illustrer l'algorithme défini dans les sections 6.4 et 6.5 par un exemple numérique.

Nous considérons un ordonnancement composé de $n = 4$ bitâches. La figure 6.4 donne la disposition des bitâches de cet ordonnancement. Les valeurs numériques de θ_k et φ_k, pour $k \in \{1, \ldots, 8\}$ y sont données. Pour le calcul du critère, nous fixons la valeur de λ à 0.01.

Nous exécutons la partie en différé de cet ordonnancement pour $H = 50$. Les résultats du calcul sont séparés en deux ensembles de données, tel que décrit dans le paragraphe 6.4 : E_1 et E_2. Les données de E_1 sont consignées dans le tableau 6.3, et les données associées à E_2 sont fournies dans le tableau 6.4.

TAB. 6.3 – Données de l'ensemble E_1 pour l'exemple.

Num. période utilisée	Valeurs de φ_k, $k \in \{1, \ldots, 8\}$								Critère
	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	
1	41	0	0	0	37	1	44	$\propto$	41
2	20	21	0	0	37	1	23	$\propto$	21
3	27	0	14	0	23	15	30	$\propto$	14
4	40	0	0	1	36	1	44	$\propto$	1
5	41	0	0	0	37	1	44	$\propto$	37
6	27	0	14	0	23	15	30	$\propto$	15
7	40	0	0	1	36	1	44	$\propto$	44

A ce point, nous générons une bitâche aléatoire. Ses paramètres sont $\theta = 6$, $L = 24$, et son délai impératif $D = 170$.

Premièrement, nous sélectionnons les solutions de l'ensemble E_1 qui sont compatibles avec les paramètres de la bitâche aléatoire. Seules deux sont utilisables et ont pour valeur $\varphi_1 = 41$ et $\varphi_5 = 37$.

Il n'y a pas de solutions dans le deuxième ensemble de données E_2 qui soient utilisables pour les caractériques de la bitâche. Seul le couple (φ_1, φ_5) a le bon

TAB. 6.4 – Données de l'ensemble E_2 pour l'exemple.

φ_k	φ_l	Valeurs de φ_k, $k \in \{1,\ldots,8\}$								Z	A	B	Critère
		φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8				
1	2	20.5	20.5	0	0	37	1	23.5	$\propto$	20.5	16	16	20.5
1	3	27	0	14	0	23	15	30	$\propto$	14	26	39	14
1	4	40	0	0	1	36	1	44	$\propto$	1	46	85	1
1	5	41	0	0	0	37	1	44	$\propto$	37	66	70	37
1	6	27	0	14	0	23	15	30	$\propto$	15	113	125	14.63
1	7	41	0	0	0	37	1	44	$\propto$	41	134	137	40.62
2	3	20	10.5	10.5	0	26.5	11.5	23	$\propto$	10.5	10	10	10.5
2	4	39	1	0	1	36	1	43	$\propto$	1	30	30	1
2	5	20	21	0	0	37	1	23	$\propto$	21	50	66	21
2	6	20	11	10	0	27	11	23	$\propto$	11	97	97	10.63
2	7	20	21	0	0	37	1	23	$\propto$	21	118	120	20.62
3	4	39	0	1	1	35	2	43	$\propto$	1	20	20	1
3	5	27	0	14	0	23	15	30	$\propto$	14	40	49	14
3	6	27	0	14	0	23	15	30	$\propto$	14	73	74	13.77
3	7	27	0	14	0	23	15	30	$\propto$	14	108	124	13.62
4	5	40	0	0	1	36	1	44	$\propto$	1	20	55	1
4	6	27	0	13	1	23	14	31	$\propto$	1	53	66	0.77
4	7	40	0	0	1	36	1	44	$\propto$	1	87	130	0.63
5	7	41	0	0	0	37	1	44	$\propto$	37	31	38	36.99
6	7	27	0	14	0	23	15	30	$\propto$	15	20	35	15

panel de valeurs pour la compatibilité avec le paramètre L, mais l'insertion des θ n'est pas possible ($Z = 1 < 6$).

Maintenant, nous avons à sélectionner la meilleure solution entre les deux solutions admissibles de E_1. Ces deux solutions mènent à une augmentation égale du critère (voir équation 6.8), nous choisissons d'insérer la bitâche aléatoire dans la cinquième période de temps disponible, c'est-à-dire $\varphi_5 = 37$.

Le nouvel ordonnancement, après insertion de la bitâche aléatoire est représenté dans la figure 6.5.

6.8 Conclusions

Dans ce chapitre, nous avons fourni un algorithme permettant l'insertion d'une bitâche aléatoire dans un ordonnancement préexistant de bitâches, et ce en un temps polynomial. Cet algorithme se décompose en deux parties :

- La première partie s'exécute en différé, et sert à préparer l'éventuelle arrivée d'une bitâche aléatoire. Elle consiste, pour chaque choix d'insertion possible de l'ordonnancement existant, à connaître quelles sont les caractéristiques des bitâches pouvant y être insérées. Nous modélisons ce problème sous la forme d'un ensemble de problèmes de programmation linéaire, et utilisons l'algorithme du simplexe pour les résoudre.
- Une partie en temps-réel, exécutée si un calcul de la partie en différé a été effectué pour l'ordonnancement actuel et si une bitâche aléatoire apparaît. On sélectionne parmi les solutions fournies par le calcul en différé celles qui sont compatibles avec la bitâche aléatoire et qui mènent aux plus faibles augmentation du critère.

Bien évidemment, à chaque modification de l'ordonnancement, il faut recalculer la partie en différé de l'algorithme.

Nous avons décrit une méthode permettant d'accélérer significativement le calcul de la partie en différé de l'algorithme d'ordonnancement, en tirant partie du fait que les résolutions des problèmes de programmation linéaire sont indépendantes les unes des autres. Ces calculs peuvent donc être menés suivant la technique de calcul parallèle appelée technique *SPMD*. En d'autres termes, on utilise plusieurs processeurs sur lesquels on fait tourner un algorithme du simplexe. Nous affectons à chacun de ces processeurs un sous-ensemble distinct des problèmes à résoudre. La résolution de chaque sous-ensemble se fait parallèlement.

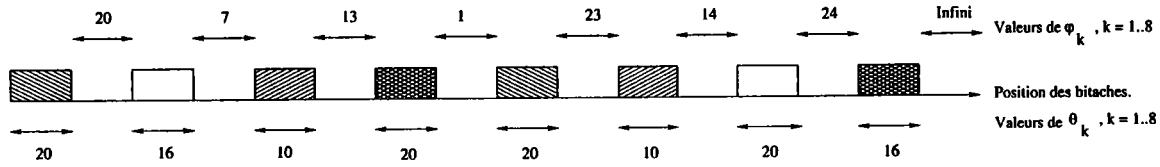


FIG. 6.4 – Ordonnancement pour l'exemple.

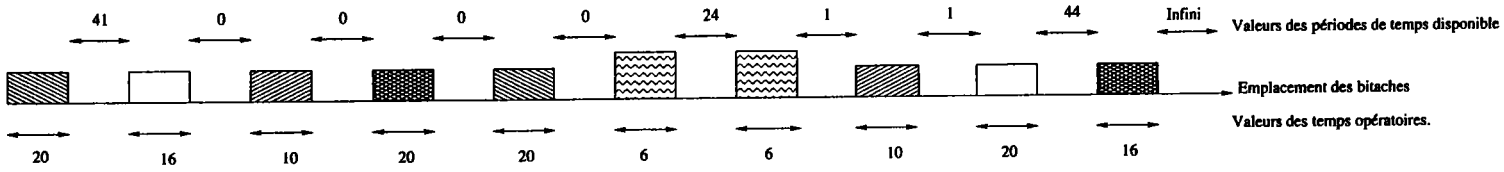


FIG. 6.5 – Ordonnancement pour l'exemple, après insertion de la bitâche aléatoire..

Résumé

Dans ce chapitre, nous avons développé une approche permettant d'insérer une bitâche dans un ordonnancement de bitâches. Pour cela, nous avons exprimé le problème sous la forme d'un problème de programmation linéaire.
Les contraintes sont fournies par l'expression des périodes d'inactivité incompressibles séparant les deux tâches d'une même bitâche, et par le fait qu'il est impossible d'avancer le temps d'exécution d'une bitâche.
Deux cas d'insertion de la bitâche dans l'ordonnancement sont possibles :
- Insèrer la bitâche dans une seule période d'inactivité de l'ordonnancement initial.
- Insèrer les deux tâches de la bitâche aléatoire dans deux périodes d'inactivité distinctes de l'ordonnancement initial.

Bien évidemment le critère d'optimisation est différent suivant le cas (voir 6.3)
L'approche heuristique proposée se décompose en deux parties :
- Une partie en différé : on résoud tous les problèmes de programmation linéaire, et on classe les résultats suivant la norme définie dans la section 6.3.
- Une partie en temps réel : on extrait des données calculées lors de l'exécution de la partie en différé de l'algorithme celles qui permettent d'insérer la bitâche aléatoire, et parmi ces dernières celles qui conduisent à l'augmentation minimale du critère.

Cet algorithme est décrit dans la figure 6.3.
Comme les problèmes de programmation linéaire que nous avons à résoudre lors de la partie en différé de l'algorithme sont indépendants les uns des autres, nous pouvons utiliser la technique de calcul parallèle dite technique gros-grain. Nous pouvons en effet exécuter en même temps m problèmes distincts sur m processeurs différents. Ceci permet de diviser le temps opératoire total par le nombre de processeurs, et ce même si la complexité totale du problème est légèrement accrue par la gestion du parallélisme.

Troisième partie

Simulation et gestion d'un radar multifonctions. Amélioration de l'existant.

Présentation de la troisième partie

Les radars multifonctions doivent remplir plusieurs types de missions : surveillance du ciel, suivi de cibles, guidage de missiles, etc, ... Certaines de ces missions sont effectuées de manière systématique et cyclique : c'est le cas des tâches de surveillance ou de guidage. D'autres sont aléatoires : elles apparaissent lorsqu'une nouvelle cible est détectée et doit être confirmée et suivie. Les tâches qui constituent un suivi de cible sont elles aussi aléatoires, car les cibles suivies peuvent ne pas se trouver là où l'extrapolation de leur trajectoire antérieure nous suggérait de les chercher. Dans ce cas, le radar est obligé de retrouver la cible perdue en faisant des recherches dans le voisinage immédiat de sa dernière position connue.

Cette partie est différente des deux premières par les aspects suivants :

- Une fois que la décision d'ordonnancement d'une tâche est prise, on ne peut plus la remettre en question. L'insertion d'une nouvelle tâche ne peut se faire que dans les périodes libres situés entre les tâches déjà ordonnancées, et il n'est pas possible de retarder une tâche déjà ordonnancée.
- Nous prenons en compte l'insertion de trains de bitâches (suite de bitâches ayant des contraintes temporelles entre elles) comme une tâche unique. En d'autres termes, nous renonçons à utiliser les périodes disponibles entre les tâches d'une même bitâche. C'est sous cette hypothèse que les radars sont gérés actuellement. Nous avons choisi ces hypothèses restrictives car elles sont actuellement retenues pour la gestion des radars de combat.
- Nous devons tenir compte d'une contrainte supplémentaire : un temps maximal d'émission sur une période donnée. Cette contrainte reflète le fait qu'il n'est pas possible de consommer plus d'une quantité donnée

d'énergie en continu. Cette quantité d'énergie dépend de la précision souhaitée pour le radar à cet instant.

- La gestion d'une priorité entre les tâches.

La suite se place dans le contexte de la gestion actuelle des radars.

Chapitre 7

Insertion d'un train de bitâches.

7.1 Introduction.

Le but de ce chapitre est de proposer une méthode de gestion pour les radars multifonctions (voir chapitre 1), et de l'évaluer. Nous nous plaçons dans le cas où l'ordonnancement existant est figé (c'est-à-dire que nous ne pouvons pas revenir sur une décision d'ordonnancement antérieure).

Un radar multifonctions est un radar qui explore systématiquement une partie de ciel afin de détecter s'il ne s'y trouve pas des cibles (avions, missiles, etc, ...). Les cibles détectées sont alors suivies jusqu'à ce qu'elles quittent l'espace d'activité du radar.

La technologie des radars est bien connue. Elle consiste en l'émission d'ondes électromagnétiques vers des cibles supposées. Une partie de ces ondes est réfléchie et renvoyée vers le radar émetteur. Des informations sont ainsi fournies au radar lors de l'éventuelle réception : distance de la cible, nature de la surface réfléchissante, vitesse, direction, etc, ...

Malheureusement, la pluie, des oiseaux, et bien d'autres corps en suspension dans l'atmosphère peuvent réfléchir ces ondes, et être prises pour des cibles potentielles. C'est pourquoi lorsqu'une tâche de surveillance détecte une cible possible, il faut immédiatement effectuer un deuxième envoi d'ondes électromagnétiques dans cette direction afin d'avoir confirmation de la réalité de la cible.

A ce point de notre description, soulignons le fait que les ondes électromagnétiques émises peuvent avoir plus ou moins de puissance énergétique. En fait, plus les ondes émises sont énergétiques, plus les informations fournies en

retour sont précises. En d'autres termes, plus le radar consomme de l'énergie, moins il y a de cibles non détectées. Bien entendu, cette médaille a son revers, et les fausses alertes liées à l'environnement sont en augmentation lorsque la consommation en énergie du radar croît. De plus, matériellement, il existe une contrainte sur la quantité d'énergie qu'un radar peut émettre dans une période de temps donnée : plus la quantité d'énergie dépensée pour émettre le faisceau d'ondes associé à une tâche est grande, plus le nombre de tâches que l'on va pouvoir exécuter est faible.

Ainsi, le but est de trouver un compromis entre la précision du radar, le nombre de tâches que l'on veut exécuter et le nombre de cibles que l'on peut suivre de manière fiable.

Dans la section 7.2, nous décrivons en détails le problème que nous cherchons à résoudre. La section 7.3 est dédiée à la description de notre approche. Dans la même section, nous proposons un résultat nous donnant une évaluation du nombre de cibles non détectées et décrivons le programme de simulation. La section 7.4 propose plusieurs illustrations numériques montrant l'efficacité de notre algorithme. La section 7.5 est la conclusion.

7.2 Formulation du problème.

Suivant les principes technologiques des radars multifonctions décrits en [54], [46] et [63], la partie du ciel qui doit être surveillée par le radar peut être représentée par une matrice à deux dimensions comptant n lignes et m colonnes. Chaque élément de cette matrice représente une partie du ciel où une tâche de surveillance doit être exécutée. Il y a donc $m.n$ zones de ciel à surveiller. Cela se fera en deux périodes de durée T. A chaque instant le radar a accès à k ($k << m$) colonnes consécutives de la matrice. L'antenne du radar pivote autour de son axe et, en permanence, des colonnes disparaissent du domaine de visibilité du radar tandis que d'autres colonnes y entrent. Nous appelons Δ le temps requis par une colonne pour entrer ou sortir du domaine de visibilité. A la fin de chaque période élémentaire Δ, la colonne située à l'extrême-gauche du domaine de visibilité disparait du domaine, tandis que la colonne située en dehors de domaine de visibilité et voisine de la colonne se trouvant le plus à droite de ce domaine y entre. Ce processus est périodique, et

la colonne 1 est définie comme celle précédant la colonne m. Le radar effectue une rotation complète dans un temps donné, et doit effectuer une vérification complète du ciel en deux tours d'antenne.

La période nécessaire pour ce processus, c'est-à-dire la période nécessaire pour parcourir les m colonnes, est désignée par T, et $\Delta = \frac{T}{m}$. Une contrainte du problème est que chaque tâche de surveillance d'une zone du ciel doit être exécutée exactement une fois toutes les deux périodes. Lorsque la surveillance d'un de ces éléments est effectuée, il est possible qu'une nouvelle cible soit détectée. Nous définissons par p_a la probabilité de détection d'une cible. Il faut alors confirmer l'existence de cette cible par une tâche dite de confirmation. Cette tâche de confirmation ne doit pas débuter moins de $50ms$ après la fin de la tâche de surveillance, et doit être achevée $100ms$ plus tard. La probabilité que cette tâche de confirmation entérine l'existence réelle d'une cible est notée p_c. Chaque cible confirmée doit ensuite être suivie une fois toute les périodes T, jusqu'à disparition de la cible de la zone de surveillance du radar. Idéalement, ces suivis devraient être effectués aux instants $T_{ec}+T, T_{ec}+2T, T_{ec}+3T, \ldots, T_{ec}+jT, \ldots$, où T_{ec} est l'instant de complétion de la tâche de confirmation. Bien entendu, la charge de travail du radar ne permet pas toujours d'exécuter ces tâches aux instants idéaux. La politique adoptée pour palliertenir compte de cette contrainte consiste à dire qu'une tâche devant idéalement commencer son exécution à l'instant $T_{ec} + jT$ peut être reportée, mais doit être achevée au plus tard à l'instant $T_{ec} + jT + 50ms$.

Il y a plusieurs causes menant à l'interruption du suivi d'une cible par le radar :

- Le radar est surchargé de travail et ne peut pas satisfaire à temps la tâche de surveillance associée à la cible.
- La cible suit des trajectoires désordonnées, et le radar n'a pas pu prédire où la cible va se trouver lors de la vérification suivante.
- La cible est hors de portée du radar (elle a quitté la zone de surveillance).

Nous nommons p_d la probabilité que la cible disparaisse du radar.

Des priorités d'exécution sont définies entre les différents types de tâches : les tâches de confirmation ont la plus grande priorité, ensuite viennent les tâches de pistage, et enfin les tâches de surveillance.

Donnons à présent quelques informations sur des caractéristiques techniques de tâches :

- Une tâche de confirmation est constituée d'une période où le radar émet des ondes électromagnétiques, suivie d'une période de disponibilité, et enfin d'une période de réception. Le radar n'est actif que durant les périodes d'émission et de réception. La figure 7.1 représente une tâche de confirmation. La période de disponibilité peut être utilisée pour exécuter d'autres tâches. Ceci correspond aux bitâches définies dans le chapitre 4.

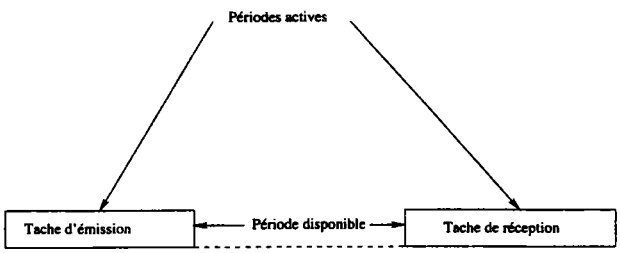


FIG. 7.1 – Une tâche de confirmation.

- Une tâche de pistage est composée d'un train de bitâches. Chaque bitâche est composée d'une période d'émission d'ondes électromagnétiques, d'une période de disponibilité et d'une période de réception. Le radar n'est actif que durant les périodes d'émission et de réception. La période de disponibilité peut être utilisée pour exécuter d'autres tâches. Le nombre de bitâches composant un train de bitâches est dépendant du domaine dans lequel opère le radar. Ici, pour simplifier, nous supposerons qu'une tâche de pistage est composée de douze bitâches. De plus, le commencement de la $(i+1)$-ème bitâche aura lieu $5\mu s$ après la fin de la i-ème bitâche. Cette période peut varier dans les limites de $\pm 10\%$.
- Une tâche de surveillance est considérée comme étant une tâche unique, et ceci en dépit du fait qu'elle contient une période d'émission, une période de disponibilité, et une période de réception. Ceci est dû au fait que les caractéristiques temporelles de ces périodes ne permettent pas d'insérer une tâche ni une bitâche dans la période de disponibilité.

Les tâches de surveillance nécessitent un temps d'émission et de réception plus long que les autres types de tâches : en effet, ces tâches explorent une partie du ciel, et servent à localiser une cible. Les autres tâches, elles, savent

exactement quel est le point du ciel visé, aussi les temps d'émission et de réception sont plus restreints.

Les équations donnant la quantification des périodes d'émission, de disponibilité et de réception pour une tâche de confirmation sont :

- Période d'émission = Période de réception = $A = kR^4p$
  - k est une constante représentant la puissance dissipée par le système
  - p est la plus petite *target radar cross section* définie par :

$$p = \frac{\text{Energie réfléchie vers la source}}{\text{Densité de l'énergie incidente}} \cdot \frac{4\pi}{\text{Angle de la cible}}$$

  - R est la distance entre le radar et la cible.
- Temps de disponibilité = $\max(0, 2.\frac{R}{c} - A)$, où c est la vitesse de la lumière.

Dans le cas d'une tâche de pistage, les formules utilisées pour calculer les valeurs des périodes de temps correspondant à chacune des bitâches sont les mêmes que celles utilisées pour calculer les périodes de temps correspondant aux tâches de confirmation.

Les tâches de surveillance sont calculées à l'aide des équations suivantes :

- Période d'émission = $k.(X+Y)^4.p$
  - X est la distance entre le radar et le point le plus proche de l'espace surveillé.
  - Y est la profondeur de l'espace surveillé : c'est-à-dire la distance entre le point le plus proche et le plus lointain de l'espace surveillé
- Période de réception = $\frac{2.Y}{c}$
- Période de disponibilité = $\max(0, \frac{2.X}{c} - k.(X+Y)^4.p)$

Notons que les temps associés aux tâches de surveillance sont toujours les mêmes, tandis que les temps associés à une tâche confirmation ou de pistage sont fonction de la position de la cible.

Plus d'informations sur les bases techniques de ces formules peuvent être trouvées dans [48].

La contrainte liée à l'énergie dépensée par l'émission d'ondes électromagnétiques est donnée par une fonction linéaire par morceaux dont un exemple est donné dans la figure 7.2. Lorsque le radar est initialisé, la valeur de cette fonction est égale à zéro. Durant une période d'émission, la fonction augmente, et sa pente est de α ($\alpha > 1$). Cette pente est fonction de l'énergie dépensée par

unité de temps. Lors d'une période de réception ou de disponibilité, la fonction décroît suivant une pente de -1. La valeur maximale que peut atteindre la fonction est définie par la valeur w. Avant la planification de toute tâche, on doit donc calculer s'il est possible de l'exécuter sans que la fonction dépasse la valeur seuil de w. Dans le cas où il n'est pas possible d'exécuter une tâche sans dépasser la valeur w, alors l'exécution de la tâche est repoussée jusqu'à l'instant où la valeur de la fonction permet son exécution (si les contraintes de l'ordonnancement le permettent). Dans la figure 7.2, c'est le cas à l'instant t_4 où l'on tente d'insérer une tâche d'émission. La simulation de l'insertion est matérialisée par une ligne pointillée. Malheureusement, insérer cette tâche au temps t_4 amènerait une valeur de la fonction supérieure à w. Le radar est donc obligé d'effectuer à l'instant t_4 soit une tâche de réception, soit une attente de façon à faire baisser la valeur de la fonction. La tâche d'émission suivante est exécutée au temps t_5.

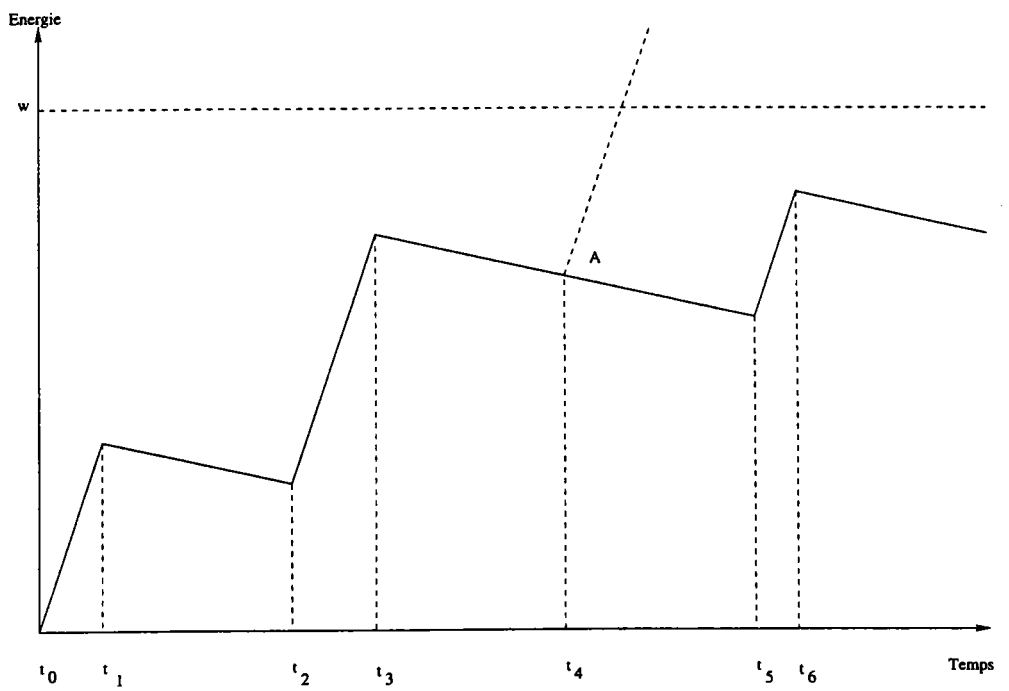


FIG. 7.2 – Contrainte sur l'énergie.

La valeur de w est constante pour un radar donné, tandis que la valeur de α est fonction de la précision du radar : plus la précision est grande plus la valeur de α est grande, ce qui réduit la période maximale d'émission sans interruption du radar.

Un autre facteur entre en ligne de compte lorsque nous examinons la précision du radar : ce sont les valeurs de n (nombre de lignes de la matrice représentant la portion de ciel contrôlable par le radar) et m (nombre de colonnes). Si l'un de ces paramètres décroît, et que l'énergie utilisée par le radar pendant les périodes d'émission demeure la même, la précision du radar décroît. En effet, l'énergie (constante) est dispersée sur un plus grand volume. Pour implanter la simulation proposée dans ce chapitre, nous avons considéré que la probabilité de confirmation, notée p_c, décroît lorsque n et/ou m décroît, alors que la probabilité de disparition d'une cible, notée p_d croît. En effet, diminuer n ou m tout en gardant constante l'énergie dépensée pour effectuer les tâches diminue la précision du radar, et donc le nombre de détections de cibles et, par voie de conséquence, le nombre de confirmations, c'est-à-dire p_c. Simultanément, la précision diminuant, la probabilité p_d de perdre des cibles augmente.

Dans la suite de ce chapitre, nous étudions deux cas distincts pour la simulation :

1. Nous considérons chaque tâche (confirmation, pistage, surveillance,...) comme un bloc unique. En d'autres termes, les périodes de temps disponibles situées entre les tâches d'émission et de réception d'une même bitâche ne peuvent être utilisées pour exécuter d'autres tâches ou bitâches. Nous appelons ce cas le *cas non entrelacé* (dénommé NE dans la suite de ce chapitre).
2. Nous considérons que le temps libre situé entre les deux tâches d'une même bitâche peut être utilisé pour exécuter d'autres activités d'émission ou de réception. Nous appelons ce cas *cas entrelacé* (dénommé E dans la suite de ce chapitre). Il s'agit d'une amélioration par rapport à l'existant.

Dans ces deux cas, nous supposons que nous ne pouvons plus revenir sur une décision d'ordonnancement prise antérieurement : une fois que l'on a décidé à quel instant une tâche doit être exécutée, il n'est plus possible de le modifier (pas de flexibilité). En d'autres termes, l'insertion d'une nouvelle tâche ne pourra se faire que dans une (ou plusieurs) fenêtre(s) de temps laissée(s) disponible(s) par les tâches déjà ordonnancées.

7.3 Résolution du problème.

Afin d'être en mesure d'évaluer la validité de notre approche, nous nous positionnons dans le cas d'un régime permanent.

Dans ce cas de figure, le nombre de nouvelles cibles apparaissant sur une période $2T$ est le même , en moyenne, que le nombre de cibles disparaissant durant la même période (un balayage complet du ciel est effectué toutes les périodes $2T$.). Cette propriété est formalisée dans l'équation (7.1) :

$$m.n.p_a.p_c = 2.N.p_d \tag{7.1}$$

où N est le nombre de cibles présentes dans le ciel.

Le premier membre de l'équation (7.1) est égal au nombre de cibles qui seraient réellement détectées si toutes les tâches de surveillance étaient effectuées chaque fois que le radar effectue deux rotations de son antenne (soit un temps de $2T$), et si le brouillage ambiant n'empêchait pas leur détection.

Le second membre représente le nombre de cibles disparaissant du ciel pendant la même période. L'équation (7.1) se réécrit :

$$N = \frac{m.n.p_a.p_c}{2.p_d} \tag{7.2}$$

Dans les deux cas E et NE, nous utilisons la même stratégie de contrôle du radar, définie comme suit.

Chaque fois qu'une nouvelle colonne apparait dans le domaine de visibilité du radar :

1. Nous explorons d'abord la liste des tâches de confirmation à effectuer, et sélectionnons celles qui doivent être exécutées entre la fin de l'apparition de la colonne courante dans le domaine de visibilité du radar et la fin de l'apparition de la suivante, c'est-à-dire durant la période élémentaire courante Δ. Le lecteur notera que cette période élémentaire peut-être utilisée pour exécuter des tâches de confirmation correspondant à des alertes antérieures, ou pour exécuter d'autres bitâches de pistage. En conséquence, il est possible que toutes les nouvelles tâches de confirmation ne puissent pas être insérées dans l'ordonnancement existant. Dans ce cas, la cible éventuelle correspondante sera perdue.

2. Nous explorons ensuite la liste des tâches de pistage qui doivent être exécutées durant la période élémentaire courante. Les tâches de pistage sont difficiles à ordonnancer car elles sont constituées de douze bitâches fortement contraintes temporellement entre elles. Si il est impossible d'exécuter une tâche de pistage tout de suite, on tente à nouveau de l'exécuter lorsque une nouvelle colonne apparaît (c'est-à-dire durant le prochain temps élémentaire Δ). Si les contraintes associées à une tâche de pistage ne permettent pas de l'insérer dans l'ordonnancement, alors la cible associée est considérée comme perdue.
3. Enfin, nous examinons les tâches de surveillance associées à la colonne qui vient d'apparaître,et tentons d'exécuter celles qui n'ont pu être prises en compte lors de la période T précédente. Il est bien entendu possible que l'on ne puisse pas exécuter toutes ces tâches à cause de la faible priorité de celles-ci et de la charge du radar. Lorsqu'une tâche de surveillance ne peut être effectuée avant que la colonne correspondante ne disparaisse du domaine de visibilité du radar, alors elle est considérée comme perdue, mais sera retentée lors de la prochaine période élémentaire T.
4. Si la période Δ concernée n'est pas entièrement utilisée, on tente d'y ordonnancer des tâches de surveillance n'ayant pu être ordonnancées lorsque la colonne leur correspondant est apparue dans le domaine de visibilité du radar.

Comme nous venons de le voir, la stratégie appliquée consiste à ordonnancer les tâches au plus tôt, suivant l'ordre de leurs priorités. Le logiciel de simulation a été développé en *JAVA*.

7.4 Exemples numériques.

Le premier jeu d'exemples est utilisé pour comparer les cas E et NE. Le second jeu se focalise sur le cas E et mesure les performances du système lorsque la précision du radar change. Nous utilisons l'équation (7.2) pour modifier le nombre de cibles présentes dans le ciel : nous fixons les valeurs de p_a et de p_c, mais faisons varier la valeur de p_d. Rappelons que plus p_d est grand, plus la valeur de N est faible.

7.4.1 Comparaison des cas E et NE.

Dans cette section nous fournissons des exemples numériques. Les résultats pour le cas E sont donnés dans la table 7.1, et les résultats pour le cas NE sont dans la table 7.2. Les probabilités p_a et p_c sont fixées, et dans nos exemples, leurs valeurs sont respectivement 0.3 et 0.1. Nous nous contentons de faire varier p_d. Comme décrit dans le paragraphe précédent (voir équation (7.2)), changer la valeur de p_d est un moyen de changer le nombre N de cibles réellement présentes dans la partie du ciel contrôlable par le radar. Dans les lignes suivantes, nous donnons la signification des abréviations employées dans les tableaux 7.1 et 7.2 :

- MB est le pourcentage de tâches de surveillance n'ayant pu être exécutées.
- MT est le pourcentage de tâches de pistage n'ayant pu être exécutées.
- MC est le pourcentage de tâches de confirmation n'ayant pu être exécutées.
- IT est le pourcentage du temps inutilisé du radar.
- PT est le pourcentage du temps total de simulation utilisé pour exécuter des tâches de surveillance.
- TT est le pourcentage du temps total de simulation utilisé pour exécuter des tâches de pistage.
- CT est le pourcentage du temps total de simulation utilisé pour exécuter des tâches de confirmation.
- La colonne NPP contient deux résultats :
  - Le nombre de cibles réellement présentes dans le domaine de visibilité du radar.
  - Le nombre moyen de cibles suivies par le radar.

TAB. 7.1 – Le cas E

p_d	MB	MT	MC	IT	PT	TT	CT	NPP
0.30	0.04	0.42	0.02	66.83	32.74	0.33	0.08	64/63
0.25	0.06	0.52	0.02	66.78	32.73	0.39	0.08	77/74
0.2	0.09	0.84	0.04	66.66	32.73	0.5	0.08	96/93
0.1	0.21	3.64	0.06	66.21	32.73	0.96	0.08	192/184
0.05	4.41	13.41	0.23	66.35	32.03	1.53	0.08	384/317

TAB. 7.2 – Le cas NE

p_d	MB	MT	MC	IT	PT	TT	CT	NPP
0.30	0.1	1.25	0.03	66.84	32.73	0.32	0.08	64/62
0.25	0.14	1.8	0.03	66.77	32.73	0.39	0.08	77/75
0.2	0.15	2.55	0.03	66.71	32.73	0.46	0.08	96/90
0.1	1.12	12.36	0.1	66.59	32.58	0.73	0.08	192/157
0.05	24.43	30.07	0.12	70.83	28.2	0.88	0.07	384/229

Le lecteur observe que le pourcentage de temps inutilisé par les tâches sur le radar se situe entre 66% et 71% dans les deux cas envisagés. Ce résultat est très important. Il résulte des contraintes qui s'appliquent aux tâches comme par exemple les délais, les liens temporels entre les bitâches d'une même tâche, etc. Ceci indique que nous ne pouvons pas tirer parti de nombreuses périodes de temps libre dans le cas E.

Des améliorations des performances sont cependant à constater dans le cas E en regard du cas NE. Ces améliorations sont beaucoup plus manifestes lorsque le radar est très sollicité (environ 4.5%) que lorsque le radar peut satisfaire à toutes les demandes (environ 0.01%).

Nous observons aussi que plus de tâches de pistage peuvent être exécutées dans le cas E que dans le cas NE : ces tâches sont celles qui profitent le plus de la stratégie d'entrelacement des tâches.

7.4.2 L'efficacité par rapport à la précision du radar.

Introduction

Nous définissons tout d'abord les activités utiles du radar comme étant :

- les tâches de surveillance,
- les tâches de confirmation ayant un retour positif (c'est-à-dire celles confirmant la découverte d'une nouvelle cible.)
- les tâches de pistage actives : c'est-à-dire celles confirmant que l'on connait toujours la position des cibles suivies.

Bien sûr, un radar est soumis aux aléas de son environnement, aussi nous définissons le temps total théorique d'utilisation du radar, qui correspond à ce que serait l'activité de ce même radar dans un environnement exempt de brouillage.

Deux critères vont être considérés dans cette sous-section :

- Le critère d'efficacité absolue (CEA en abrégé) : c'est le pourcentage du temps utilisé par le radar pour effectuer des activités utiles par rapport au temps total théorique.
- Le critère d'efficacité relative (CER en abrégé) : c'est le pourcentage de temps utilisé par le radar pour effectuer des tâches utiles par rapport au temps total où il a été utilisé.

  Ces temps sont évalués sur une période donnée kT suffisament longue pour être assuré que l'on est en régime permanent.

Le temps utilisé pour exécuter des activités utiles est obtenu lors de la simulation, alors que le temps total théorique que le radar aurait dû utiliser s'il avait suivi toutes les cibles présentes dans son rayon d'action est calculé en utilisant la relation (7.2) comme nous allons le montrer dans le paragraphe suivant.

Le lecteur notera que dans cette section, nous ne considèrerons que le cas E.

Evaluation du temps nécessaire pour exécuter les activités.

Nous considérons que p_a, p_c, p_d, m et n sont connus. Alors il est possible de calculer le nombre moyen N de cibles présentes dans le ciel en utilisant la relation (7.2).

Le lecteur remarquera que $N.p_d$ est le nombre moyen de nouvelles cibles qui apparaitront (ou disparaitront) du ciel durant une période T. En conséquence, cette valeur est aussi celle du nombre de tâches de confirmation qui devront être exécutées durant cette même période de temps T.

Le nombre moyen de tâches de pistage durant une période T est égal à N puisque chaque cible doit être pistée une fois au cours de chaque période T.

Le nombre moyen de tâches de surveillance ayant lieu durant une période T est égal à $\frac{m.n}{2}$ puisque chacune des $m.n$ parties du ciel sous contrôle du radar est examinée une fois toutes les deux périodes T.

Considérons maintenant le temps moyen t_{conf} requis pour exécuter une tâche de confirmation. Nous considérons que les cibles sont distribuées uniformément entre $D_{min} = 10000$ et $D_{max} = 300000$ mètre d'altitude. En accord avec les formules introduites dans la section 7.2, le temps d'émission plus le

temps de réception requis pour atteindre une cible située à R mètres d'altitude est $2.k.p.R^4$.

Or :

$$t_{conf} = \frac{2.k.p}{D_{max} - D_{min}} \int_{R=D_{min}}^{D_{max}} R^4 dR$$

Nous obtenons :

$$t_{conf} = \frac{2.k.p.(D_{max}^5 - D_{min}^5)}{5.(D_{max} - D_{min})} \tag{7.3}$$

Comme $k = 0.5$ et $p = 10^{-20}$, nous obtenons $t_{conf} = 16.75\mu s$.
En conséquence (l'unité de temps est la μs) :

- Le temps moyen utilisé pour exécuter des tâches de confirmation durant une période de temps T est :

$$T_c = N.p_d.t_{conf}$$

En utilisant les relations (7.2) et (7.3) et nous obtenons :

$$T_c = \frac{1}{2}.m.n.p_a.p_c.t_{conf} \tag{7.4}$$

- Le temps moyen utilisé pour exécuter des tâches de pistage durant une période de temps T est :

$$T_t = 12.N.t_{conf}$$

Le facteur douze apparait car douze bitâches, identiques à une tâche de confirmation, sont exécutées pour faire une tâche de pistage. Nous utilisons encore les relations (7.2) et (7.3), nous obtenons :

$$T_t = \frac{6.m.n.p_a.p_c.t_{conf}}{p_d} \tag{7.5}$$

- Enfin, le temps moyen utilisé pour exécuter des tâches de surveillance durant une période de temps T est :

$$T_p = \frac{m.n}{2}.2066$$

2066 μs est le temps requis pour exécuter une tâche de surveillance qui, rappelons-le, est considérée comme étant un seul bloc. Ce temps est obtenu en additionnant a_s, b_s et L_s (voir section 7.2) et est constant. Le facteur $\frac{1}{2}$ reflète le fait que chaque tâche de surveillance est effectuée une fois toutes les deux périodes de temps T. En prenant en compte la relation (7.2), nous obtenons :

$$T_p = 1033.m.n \tag{7.6}$$

Enfin, le temps moyen requis pour effectuer les activités lors de chaque période T est :

$$Tot = T_c + T_t + T_p$$

La relation précédente peut être réécrite ainsi :

$$Tot = m.n.p_a.p_c.t_{conf}.(0.5 + \frac{6}{p_d}) + 1033.m.n \tag{7.7}$$

Exemples numériques.

Le nombre moyen de cibles dans le ciel est le même pour les quatre exemples présentés dans cette sous-section, et est égal à 192. Cette valeur est dérivée de l'équation (7.2) en prenant comme paramètres $m = 256$, $n = 5$, $p_a = 0.3$, $p_c = 0.1$ and $p_d = 0.1$, qui sont des paramètres réalistes.

Nous considérons que la précision du radar décroît d'un exemple au suivant. Pour simuler cette évolution de la précision, nos réduisons le nombre m de colonnes de la matrice qui représente le ciel et augmentons $\frac{p_a.p_c}{p_d}$ de façon à avoir une valeur constante pour N. Les résultats de la simulation sont donnés dans la table 7.3. Les valeurs données en résultat sont des temps (exprimés en μs) par période de temps T. Une période T est égale à quatre secondes dans ces exemples. Nous nous restreignons aux résultats utiles pour l'évaluation des critères d'efficacité absolue et relative.

TAB. 7.3 – Résultats des simulations

Temps utilisé	Exemple 1 $m = 256$	Exemple 2 $m = 183$	Exemple 3 $m = 128$	Exemple 4 $m = 80$
Activités utiles	1,249,809	954,285	690,088	445,060
Activités inutiles	3,325,592	3,436,778	3,493,926	3,539,120

Possédant ces valeurs, nous calculons :

- Le critère d'efficacité absolue (CEA), comme suit :

$$\frac{\text{Temps utilisé pour les activités utiles}}{\text{Tot}}$$

- Le critère d'efficacité relative (CER), comme suit :

$$\frac{\text{Temps utilisé pour les activités utiles}}{\text{Temps utilisé pour les activités utiles} + \text{Temps utilisé pour les activités inutiles}}$$

Les temps consommés pour exécuter des activités utiles et inutiles sont donnés dans la table 7.3, tandis que Tot est calculé en utilisant la relation (7.7). Les résultats finaux sont donnés dans la table 7.4.

TAB. 7.4 – CEA et CER lorsque la précision du radar décroît.

Temps utilisé	Exemple 1 $m = 256$	Exemple 2 $m = 183$	Exemple 3 $m = 128$	Exemple 4 $m = 80$
CEA	0.824	0.838	0.806	0.732
CER	0.918	0.970	0.985	0.984

Le lecteur remarquera que l'efficacité maximale n'est pas obtenue en utilisant la précision maximale. En effet, dans la table 7.4, nous constatons que l'exemple 2 mène à un meilleur CEA alors que l'exemple 3 mène à un meilleur CER. L'explication est simple.

Lorsque nous augmentons la précision du radar, plus de tâches de surveillance doivent être exécutées, et le nombre de fausses alertes augmente plus vite que le nombre d'alertes confirmées. Ces fausses alertes consomment non seulement du temps radar, mais aussi de l'énergie, ce qui force le radar à stopper ses émissions d'ondes électromagnétiques, et ainsi à perdre la trace de cibles suivies et/ou à ne pas en détecter de nouvelles.

Lorsque la précision du radar baisse, le nombre de tâches de surveillance baisse aussi, ce qui libère du temps pour exécuter des tâches utiles, mais pas autant, proportionnellement parlant, que Tot et la somme (Temps utilisé pour les activités utiles + Temps utilisé pour les activités inutiles).

7.5 Conclusion.

Le problème présenté dans ce chapitre est nouveau. Son but est de permettre l'ordonnancement de plusieurs types d'activités, certaines d'entre elles

étant fortement contraintes, et ce en temps réel. Nous utilisons une approche de bon sens, qui consiste à ordonnancer aussitôt que possible les tâches tout en prenant en compte leur priorités. Cette approche nous a permis de suivre simultanément jusqu'à 300 cibles tout en continuant la surveillance du ciel.

Nous tirons deux conclusions de ce travail : premièrement, le radar est toujours sous-utilisé par rapport aux tâches devant être exécutées. La raison en est la complexité de la structure des tâches et les fortes contraintes qui s'appliquent. La seconde remarque concerne les fausses alertes, c'est-à-dire les alertes qui ne sont pas confirmées et qui sont coûteuses en temps et en énergie pour le radar. Une stratégie plus complexe devra être développée pour ajuster la précision du radar à l'état du ciel.

Résumé

Dans ce chapitre, nous avons développé la simulation du fonctionnement d'un radar multifonctions. Ce radar est capable d'effectuer plusieurs types de missions : surveillance du ciel, suivi de cibles, etc,.... Physiquement, pour le faire fonctionner, nous devons ordonnancer des tâches simples (surveillance), des bitâches (confirmation) et des trains de bitâches (pistage). Nous considérons que lorsque une décision d'ordonnancement est prise, c'est de manière irrévocable : il n'y a pas de flexibilité pour les tâches ordonnancées.

Nous avons mis en évidence deux propriétés dans nos simulations :

1. Entrelacer les tâches permet d'obtenir de meilleurs résultats (au regard du critère) que considérer toutes les tâches comme des tâches simples, mais l'amélioration reste limitée.
2. Utiliser la précision maximale du radar ne conduit pas à une qualité maximale des résultats du radar.

Chapitre 8

Conclusions

Cette thèse est dédiée à l'ordonnancement en temps réel des radars de combat : lorsque une nouvelle tâche (une nouvelle cible) apparaît de manière impromptue, il faut être en mesure de l'insérer dans l'ordonnancement des tâches à traiter. Pour décider où placer cette tâche, nous ne disposons que d'un temps de calcul très restreint dû aux contraintes matérielles du radar.

Pour effectuer cette insertion, il nous est possible de profiter de la flexibilité des tâches déjà ordonnancées. En d'autres termes, nous pouvons remettre à une date ultérieure l'exécution de tâches déjà programmées (dans les limites de leur flexibilité fixée par leur délai).

Les tâches exécutées sur un radar, et que nous appelons bitâches, sont définies par deux temps opératoires séparés par un temps disponible. Les opérations ne peuvent pas être interrompues une fois commencées. Ces deux opérations sont successivement une émission et une réception d'ondes.

Deux possibilités majeures s'offrent à nous :

- considérer que la bitâche est une tâche simple, commençant au début du premier temps opératoire et finissant à la fin du second, sans interruption possible. C'est cette approche qui est actuellement utilisée pour gérer les radars.
- Considérer que nous pouvons utiliser le temps disponible séparant les temps d'activité des bitâches pour exécuter des tâches d'émission et/ou de réception d'autres bitâches. En d'autres termes, entrelacer les opérations des différentes bitâches. Cette approche est techniquement possible, mais sa mise en place bien évidemment plus complexe.

Le critère que nous considérons pour l'optimisation de l'ordonnancement

est la minimisation de la somme des retards.

Soit $L_k(\theta)$, la fonction donnant l'augmentation de critère lorsque on insère une tâche aléatoire de durée θ ($\theta \in [0, \theta_{max}]$ où θ_{max} est la durée maximale que peut prendre une tâche aléatoire) après la $k^{ième}$ tâche de l'ordonnancement initial.

La première partie de cette thèse était consacrée à la première possibilité. Nous avons développé des algorithmes heuristiques nouveaux, permettant d'effectuer l'insertion d'une tâche simple dans un ordonnancement de tâches simples. Ces algorithmes sont basés sur les résultats suivants :

1. Si plusieurs tâches sont exécutées successivement sans interruption, alors insérer une tâche aléatoire entre deux tâches de ce bloc conduira à une plus grande augmentation du critère que si on l'insère après la fin de la dernière tâche.
2. Placer une tâche aléatoire immédiatement après la fin d'une tâche de l'ordonnancement existant conduit à avoir une augmentation de critère généralement plus faible et au plus égale à l'augmentation de critère lorsque on laisse un temps libre entre ces deux tâches.
3. La fonction $L_k(\theta)$ est croissante, continue et linéaire par morceaux
4. Soit A une tâche aléatoire, θ sa durée et D son délai. Soit t_k est l'instant de départ de la $k^{ième}$ tâche de l'ordonnancement existant, et θ_k sa durée, Le calcul de la fonction :
   $L^*(\theta) = \{k^* \backslash L_{k^*}(\theta) = Min(L_k(\theta))$ et $k \in [1, Q]$ tel que $\mu_Q + t_Q + \theta \leq D\}$
   nous donne l'emplacement optimal d'insertion de la tâche aléatoire.

L'algorithme temps-réel le plus performant que nous ayons proposé pour ce cas est l'algorithme général (voir section 3.3.4), qui exploite les propriétés de la fonction *augmentation du critère.*

La deuxième partie de la thèse est consacrée à la seconde possibilité : on considère qu'il nous est possible d'utiliser les temps disponibles entre les bitâches pour exécuter d'autres tâches. Nous notons θ_1 la durée de la première tâche A_1 d'une bitâche, L sa période de disponibilité et θ_2 la durée de la deuxième tâche A_2.

Nous avons démontré les résultats suivants :

- La fonction augmentation du critère $L_k(\theta_1, L, \theta_2)$ est linéaire et croissante par morceaux par rapport à chacune des trois variables. Elle présente cependant des points de discontinuité.
- Soit $\mathcal{D} = \{\theta_1^*, L^*, \theta_2^*)\}$ un point de discontinuité de $L_k(\theta_1, L, \theta_2)$. $\mathcal{D}$ possède la propriété suivante : $L_k(\mathcal{D}^-) \geq L_k(\mathcal{D}^+)$
- Si nous décidons d'insérer A_1 après a_k, la solution optimale consiste à insérer A_1 à l'instant z. z est tel que $\mu_k + t_k \leq z \leq z_1$ avec $z_1 = \min(\mu_{k+r} + t_{k+r} - \theta_1 - L, D - \theta_1 - L - \theta_2)$, où r est le plus grand entier tel que $\sum_{i=1}^{r} t_{k+i} \leq L$
- La valeur optimale pour z est une des valeurs $\{\mu_{k+i}+t_{k+i}-\theta_1-L\}_{i=1,2,\ldots,r}$. Si $z = \mu_{k+i} + t_{k+i} - \theta_1 - L$, alors $a_{k+1}, a_{k+2}, \ldots, a_{k+i}$ sont insérées entre A_1 et A_2, alors les tâches $a_{k+i+1}, a_{k+i+2}, \ldots$ sont exécutées après A_2. Si $r = 0$, c'est-à-dire si $t_{k+1} > L$, alors $z = \mu_k + t_k$ est optimal.

Plusieurs algorithmes temps-réel ont été développés à partir de ces résultats. L'algorithme le plus performant dans le cas de l'insertion d'une bitâche dans un environnement de bitâches est l'algorithme TR-Bitâche-1 (voir section 5.6). Cet algorithme utilise l'approche tâche unique de la première partie de la thèse en tentant l'insertion d'une tâche de longueur $\theta_1 + L + \theta_2$. Une fois l'emplacement d'insertion optimal connu, on tient compte de l'existence du temps L en y insérant le plus possible de tâches de l'ordonnancement initial.

Dans le cas de l'insertion d'une bitâche dans un ordonnancement de bitâches, nous avons proposé un algorithme décrit dans la figure 6.3. Cet algorithme utilise une modélisation du problème qui nous amène à résoudre un ensemble de problèmes de programmation linéaire (résolution effectuée en différé). Cette modélisation nous permet de connaître, pour chaque point d'insertion possible, quelles sont les caractéristiques des bitâches pouvant y être insérées. Ces calculs nécessitent un temps de calcul élevé, mais ils possèdent la propriété d'être indépendant entre eux. Aussi, il nous est possible d'introduire la notion de calcul parallèle *gros grain* : on répartit les problèmes entre m processeurs, qui fonctionnent en parallèle. Ceci permet de diviser le temps de calcul par m.

Nous avons comparé les performances des approches de la première et de la seconde partie de la thèse, sur les mêmes problèmes. Il a été établi que

l'approche permettant l'entrelacement des tâches constituant les bitâches permettait d'obtenir une qualité de résultats supérieure en terme d'optimisation du critère.

Dans la troisième partie de la thèse (chapitre 7), nous abordons le problème de l'ordonnancement du radar suivant un autre angle : nous ne permettons pas le déplacement de tâches déjà ordonnancées. Nous avons écrit un programme de simulation permettant de comparer les deux approches définies plus haut : c'est-à-dire l'approche où l'entrelacement des tâches est permis, et l'approche où ce n'est pas permis. Là aussi, l'approche permettant l'entrelacement donne des résultats meilleurs en terme d'optimisation du critère. Une autre caractéristique des radars a été testée dans ce programme. En effet, les radars, techniquement parlant, ne peuvent pas émettre en continu plus d'une période donnée de temps, sous peine d'être endommagés gravement. Le programme compare donc le rapport entre la qualité de l'image du ciel fournie par le radar, et sa précision. On entend par précision du radar le nombre de zone en lesquelles on divise le ciel : plus ce nombre de zones est grand plus le radar est précis. Il apparait au travers des simulations que l'efficacité maximale du radar n'est pas obtenue en utilisant la précision maximale. Ceci s'explique par deux raisons :

- Lorsque nous augmentons la précision, il y a plus de fausses alertes coûteuses en temps d'émission et d'occupation du radar qui sont générées.
- Lorsqu'on baisse la précision moins de tâches de surveillance sont nécessaires, ce qui libère des créneaux de temps pour les autres activités.

Les travaux qui vont poursuivre cette thèse vont se concentrer sur la recherche d'algorithmes permettant d'insérer des trains de k bitâches successives (où k varie entre un et quelques dizaines) dans des ordonnancements de même nature. Pour les radars, ceci correspond au fait que leurs performances varient suivant les environnements auxquels ils doivent faire face (orages, mer, brouillage naturel ou artificiel, etc). Ces trains de bitâches permettent d'augmenter la fiabilité des résultats.

Quatrième partie

Annexes et Bibliographie

Annexe A

Linked Task Scheduling : Algorithms for the Single Machine Case.

Abstract : We consider a system made of one resource. The execution of the tasks is non-preemptive on this resource. The tasks we consider are composed of a given number of subtasks, two consecutives subtasks being separated by an idle period. These idle periods may be used for executing other subtasks. We wish to insert a new task in a given schedule. The characteristics of this task are not known before it appears, and its execution must be completed before a given deadline. The criterion is the minimization of the increase of the sum of the delays of the tasks.

Keywords : Scheduling, Linked Tasks, NP-Hardness, Graphs.

A.1 Introduction

The problem considered in this paper takes origin in the management of radars (see [57, 46]). A radar is considered as a single resource on which we have to schedule non-preemptive tasks. A basic radar task is composed of two subtasks (emission then reception of a wave) separated by an idle period. The duration values of these subtasks and of the idle period are computed from the distance of the putative target, and cannot be changed. Depending of the kind of task (pursuit, detection, ...) this basic task has to be performed a given number of times. The atmospheric conditions may lead to increase this number.

This problem is a scheduling problem. Formulation and exemples of scheduling problems may be found in [14, 17].

The scheduling of basic radar tasks, that is tasks containing two subtasks separated by an idle period, have been solved, using two approaches. The first approach consists in merging the two subtasks and the idle time in only one processing time (see [58, 64]). The second approach consists in allowing the use of the idle period of a given task by other subtasks (see [58, 65]).

In this paper, we deal with a more general problem. We consider that the tasks are made of a given number of subtasks, separated by idle periods.

Section A.2 is dedicated to problem setting. In section A.3 we describe an algorithm that solves the problem when the weight of the tasks is an integer value. Section A.4 takes advantage of graph theory for inserting subtasks with a non-integer duration. The complexity of the two algorithms is given in section A.5. Section A.6 is the conclusion.

A.2 Problem Setting

The problem concerns non-preemptive tasks executed on a single resource. The tasks we consider in this paper are made of a given number of subtasks. Each of these subtasks is separated from the next one by an idle period in which other subtasks can be scheduled. This period is equal to zero or to a given number of time units. In the remaining of this paper, we will call such tasks *polytasks*.

Let T be a set of n polytasks, and S an optimal schedule of T, that is a schedule that minimizes the sum of the delays. Let t_i, $i \in \{1, 2, \ldots, n\}$ be the i^{th} task of T. We assume that the tasks are ordered in their increasing starting time in S. n_i is the number of subtasks that compose t_i. Let t_i^j, $j \in \{1, 2, \ldots, n_i\}$ be the substasks of t_i. The starting time of t_i^j is given by g_i^j, where $j \in \{1, 2, \ldots, n_i\}$, and its duration is given by l_i^j. The deadline associated to task t_i is D_i.

At time 0, a new polytask t_* appears and has to be inserted in the schedule S before a given deadline D_*. In the remaining of the paper we refer to this random task as t_*.

The schedule S is such that each task is scheduled as soon as possible. As a

consequence, it is possible to postpone some tasks of S, but not to bring them foreward. Some subtasks are not postponable. This is true at least for the first task executed in the order of the schedule, since this task has already been started, but it is also true for all the subtasks belonging to tasks that started before time 0.

The criterion C which is the minimization of the sum of the delays is computed as follows :

$$C = \sum_{i=1}^{n} (g_i^{n_i} + l_i^{n_i} - D_i)^+ \text{ where } (x)^+ = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad \text{(A.1)}$$

In the first approach developped in section A.3, we assume that the duration of a subtask is unitary. In section A.4; the second approach considers subtask duration as a non-integer value.

A.3 Approach using the starting periods of the polytasks.

In this section, we assume that the duration of all the substasks of the polytasks is unitary. In order to modelize the problem, we consider two data structures.

The first one is a table. This table has two columns. The first one contains the name t_i of a task, and the second column contains the list of unit periods where the subtasks of t_i are located. We refer to this data structure as the task table. Table A.1 provides the task table that corresponds to the schedule represented in figure A.1.

TAB. A.1 – The task table for the example schedule.

Task Id	subtasks execution time
t_1	1,9,14
t_2	3,5,23
t_3	4,7,20
t_4	17,22
t_5	11,13,18

The second structure is a list (see figure A.2). Each element of this list contains a pair of data. The first element of the pair is a busy unit period. The

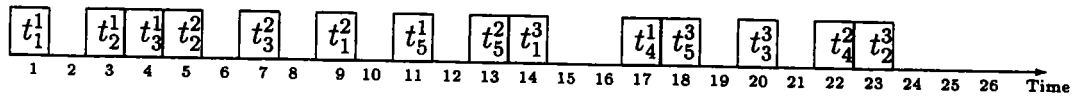


FIG. A.1 – Example of schedule.

second element is the name of the task having one of its subtasks scheduled in this unit period.

The list is built in the increasing order of the busy unit periods. We refer to this data structure as a chronological list. In figure A.2, we provide the chronological list corresponding to the schedule of figure A.1

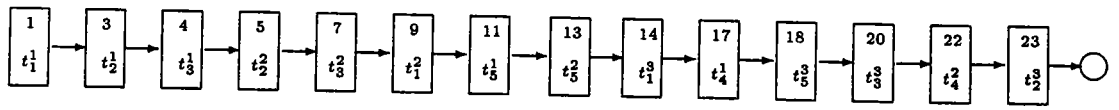


FIG. A.2 – The chronological list for the example schedule.

The references we introduce in this structure are the following :

- In the task table, the starting time of the subtasks are in fact references of the corresponding nodes in the chronological list.
- In the chronological list, the name of the task is a reference of the corresponding entry in the task table.

Some tasks are impossible to postpone. It is the case of the first task and the postponement of the tasks that started before time 0. We refer to the corresponding subtasks as blocked subtasks.

The algorithm using this structure of data is provided below. It is called Insertion Algorithm, and finds a solution, if any, when we fix the period at which the first subtask of the random task will be performed. This algorithm is given in figure A.3.

The following result states about the calculability of the insertion algorithm.

Result 1

The insertion algorithm finishes.

Proof 1

In case of conflict, at least one already scheduled task is postponed, and the random task remains at the same position.

As a consequence, the same pair of subtasks cannot be in conflict twice. Since the number of subtasks is limited, the algorithm ends. This completes the proof □

Example : We present a small example of the use of the Insertion Algorithm. We consider the schedule given in figure A.1, and we try to insert a new task t_* in the periods 3, 12 and 25. The table A.2 provides the differents steps of the computation, and the final result is provided in figure A.4.

A quality of this algorithm is that there's not always conservation of the original order of the tasks : this order may change during the computation (because of the blocked tasks).

Two obvious ways to reduce the computation burden stands in the following

1. Starting the insertion
   (a) Choose the unit period of the schedule where you want to set the first subtask of the random task.
   (b) Create a new line in the task table. In the first column, we introduce the name of the random task. In the second column we provide the list of the periods that should contain the subtasks of the random task.
   (c) Create in the chronological list the pairs corresponding to those insertion periods such that the resulting list is still chronologically ordered.
2. If all the periods of the chronological list are different from each other then go to 3
   else select in the chronological list the first reference that appears at least twice.
   (a) If none of the tasks corresponding to the reference can be postponed, then there has no solution. END.
   (b) otherwise postpone one of the non-blocked task corresponding to the reference by one unit of time (This may create new conflicts).
   (c) Go to 2
3. END

FIG. A.3 – Insertion algorithm

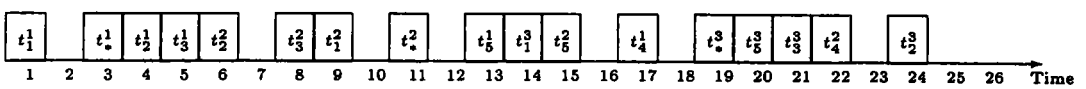


FIG. A.4 – New Schedule made with Insertion Schedule 1

remarks :

- When a task we want to postpone has k subtasks separated by zero periods of time, then we postpone it of k periods of time instead of one.
- When a task t we want to postpone is in conflict with a task (or a bunch tasks that cannot move independently from each other) that is scheduled in several consecutive periods, then it is better to postpone t of more than one period of time.

The Insertion Algorithm may be extented towards integer but non-unitary subtasks duration, considering the fact that a subtask that has a duration of k periods may be subdivided into k unitary subtasks separated by an empty idle time.

A.4 Approach using a graph

In this section, we represent the schedule by a graph called Flexibility Graph. Its goal is to provide a fast way to compute the maximum postponement of a given task in the schedule S. Each task of S is represented by a node of the graph, labelled with this task's identificator. An arc joins the nodes representing task t_{i_1} to the node representing t_{i_2} if and only if there exists at least a subtask of t_{i_1} that immediatly preceeds a subtask of t_{i_2}. The weight of the arc is the minimal idle period separating a subtask of t_{i_1} from a subtask of t_{i_2}. We denote this weight by $C(i_1, i_2)$. Furthermore, $C(i_1, i_2) = +\propto$ if there is no arc whose origin is t_{i_1} and extremity is t_{i_2}.

TAB. A.2 – Example of use of the Insertion Algorithm

Step 1	1 t_1^1	3 t_*^1	3 t_2^1	4 t_3^1	5 t_2^2	7 t_3^2	9 t_1^2	11 t_*^2	11 t_5^1	13 t_5^2	14 t_1^3	17 t_4^1	18 t_5^3	19 t_*^3	20 t_3^3	22 t_4^2	23 t_2^3
Step 2	1 t_1^1	3 t_*^1	4 t_2^1	4 t_3^1	6 t_2^2	7 t_3^2	9 t_1^2	11 t_*^2	11 t_5^1	13 t_5^2	14 t_1^3	17 t_4^1	18 t_5^3	19 t_*^3	20 t_3^3	22 t_4^2	24 t_2^3
Step 3	1 t_1^1	3 t_*^1	4 t_2^1	5 t_3^1	6 t_2^2	8 t_3^2	9 t_1^2	11 t_*^2	11 t_5^1	13 t_5^2	14 t_1^3	17 t_4^1	18 t_5^3	19 t_*^3	21 t_3^3	22 t_4	24 t_2^3
Step 4	1 t_1^1	3 t_*^1	4 t_2^1	5 t_3^1	6 t_2^2	8 t_3^2	9 t_1^2	11 t_*^2	12 t_5^1	14 t_5^2	14 t_1^3	17 t_4^1	19 t_*^3	19 t_5^3	21 t_3^3	22 t_4^2	24 t_2^3
Step 5	1 t_1^1	3 t_*^1	4 t_2^1	5 t_3^1	6 t_2^2	8 t_3^2	9 t_1^2	11 t_*^2	13 t_5^1	14 t_1^3	15 t_5^2	17 t_4^1	19 t_*^3	20 t_5^3	21 t_3^3	22 t_4^2	24 t_2^3

In figure A.5, we provide the flexibility graph corresponding to S, denoted by $G(S)$.

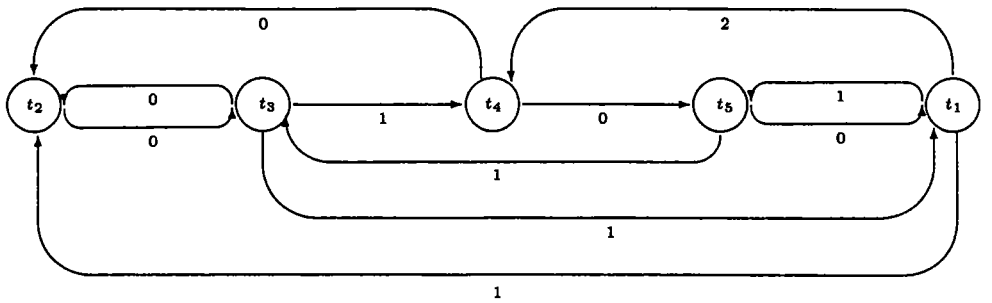


FIG. A.5 – The flexibility graph.

Indeed, if all the tasks are fixed except t_{i_1}, then t_{i_1} can be postponed by at most $\theta = \min_{i_2 \neq i_1} C(i_1, i_2)$.

A.4.1 Graph simplifications

A first simplification of the flexibility graph is made because of the blocked tasks : it is useless to keep the arcs that start from the nodes representing these tasks.

The first simplification of the flexibility graph of figure A.5 is provided in the left part of figure A.6.

The following result allows a second simplification of the flexibility graph :

Result 2

Let t_1 and t_2 be two nodes of $G(S)$. If $C(t_1, t_2) = C(t_2, t_1) = 0$ then t_1 and t_2 are mutually blocked and cannot be postponed independently from each other.

Proof 2

1. If $C(t_2, t_1) = 0$ then the smallest amount of time t_1 may be postponed without postponing t_2 is 0. So, it is not possible to postpone A without postponing t_2.
2. If $C(t_2, t_1) = 0$ then the smallest amount of time t_2 may be postponed without postponing t_1 is 0. So, it is not possible to postpone t_2 without postponing t_1.

This completes the proof □

So, if several tasks are mutually blocked (see result 2), we can merge them. Merging those tasks consists in :

1. representing these tasks by one node.
2. removing the arcs that join those tasks, and
3. taking the new node as the extremity (resp. the origin) of each arc that had one of the previous two nodes as an extremity (resp. origin).

The reader notices that the resulting graph may be divided into subgraphs that have no connections with each other (non-connex graph).

The construction algorithm of the simplified flexibility graph (SFG for short) is given below.

SFG ALGORITHM

Let BT be the set of blocked tasks.
Let PT be the set of postponable tasks. pt_i is the i^{th} element of PT. We suppose this set of task ordered in the increasing starting time of the tasks.
Let G be the set of tasks under consideration. g_i is the i^{th} element of G.
Initially, $G = \emptyset$.

1. If $G = \emptyset$ then
   (a) If $PT = \emptyset$ then go to 3
   (b) else $G = \{pt_1\}$, $PT = PT \setminus \{pt_1\}$.
2. Let $succ(g_1)$ be the set of tasks that are the successors of g_1.
   (a) $G = G \cup (succ(g_1) \cap PT)$
   (b) $PT = PT \setminus succ(g_1)$
   (c) $BT = BT \setminus succ(g_1)$
   (d) $G = G \setminus \{g_1\}$
   (e) For each t, $t \in succ(g_1)$, create an arc oriented from g_1 to t, and weight it with the minimum flexibility between the two tasks.
   (f) Go to 1
3. The nodes that are linked with a bidirectionnal arc weighted with a value of 0 must be fusionned. The arcs starting from or arriving to the original nodes must be merged too, using their minimum value.

The second simplification of the flexibility graph of figure A.5 is provided in the right part of figure A.6.

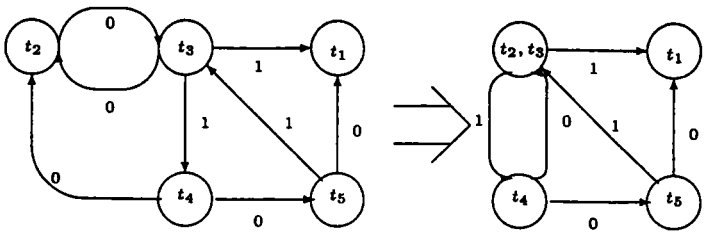


FIG. A.6 – The simplified flexibility graph of $G(S)$.

The reader will notice that the only task that may be postponed in this graph is the pair $t_2 - t_3$, for up to 1 period of time.

A.4.2 Basic result

The following result provides information about the maximum amount of time we may postpone a non-blocked task of the schedule.

Result 3

The maximum value we may postpone a non-bloked task t of S is given by the minimum of the shortest paths whose origin is t and whose extremities are blocked tasks. If no such path exists, then t can be postponed to infinity.

Proof 3

Let BT be the set of blocked tasks of S extremities of a path whose origin is t.

Let $C_{min}(a, b)$ be the weight of the shortest path between task a and task b.

1. If $BT = \emptyset$ then there exists no constraint on t, and t can be postponed to infinity.
2. Let bt_1 and bt_2 be two elements of BT. If $C_{min}(t, bt_1) > C_{min}(t, bt_2)$ (resp. $C_{min}(t, bt_2) > C_{min}(t, bt_1)$) then postponing t by $C_{min}(t, bt_1)$ (resp. $C_{min}(t, bt_2)$) would lead to postpone bt_2 (resp. bt_1), which is impossible by assumption. So, the maximum value we may postpone t is given by $\min(C(t, bt_1), C(t, bt_2)))$. Thus, the maximum postponement of task t is given by $\min_{bt \in BT} C_{min}(t, bt)$.

This completes the proof □

In our example, if we wish to postpone the task t_2, the maximum value we may postpone it is 1, and the resulting graph is given figure A.7.

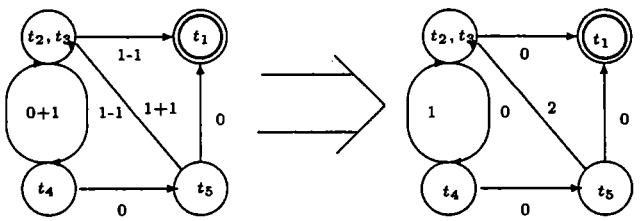


FIG. A.7 – Postponement of task t_2 by one period of time.

In result 3, we have to compute the shortest paths (in the simplified flexibility graph) between the node t associated to the task we want to postpone, and the blocked nodes extremities of paths originated in t, if any. The maximal postponement is then given by the path with the smallest weight.

The computation of the shortest path is made using the Dijkstra Algorithm (see [17, 66]).

A.4.3 Insertion algorithm 2

In this subsection, we provide an algorithm inserting a new polytask in a given schedule S. First, we use the SFG algorithm to create the graph $SFG(S)$. Then, we compute the set of possible insertion instants for the new polytask in S. The possible insertion periods are such that it does not lead the new polytask to violate its due date D_*. We test these possible instants of insertion in their chronological order. A test consists in evaluating for each subtask of the new polytask if the time available at the wished insertion instant is big enough to allow the insertion of the considered subtask. If yes then we iterate with the next subtask. If no, we use the result 3 to know whether postponing the task that should follow the considered subtask in S would solve the problem. If no then the insertion is not possible here, and we have to consider another starting instant for the insertion, if any.

These requires adjustements in $SFG(S)$. If we perform a postponement of tasks, we do not need to start over the computation of $SFG(S)$.

Let z be the maximum postponement of t_i and $k_i \leq z$. It is easy to define the new simplified flexibility graph obtained when postponing a task t_i by k_i :

- k_i is substracted from the weight of the arcs issued from the node associated to t_i.
- k_i periods of time are added to the arcs whose extremity is the node associated with t_i.

If the weights of the arcs are negative, we proceed as follows.

For each node u that is the extremity of at least one arc whose weight is negative, we compute $a = \min_{v \in Pred(u)} (C(v,u)) < 0$ where $Pred(u)$ is the set of nodes origin of the arcs whose extremity is u. We then add $-a$ to each arc whose origin is u.

We continue until the weight of all the arcs is greater than or equal to zero.

The Insertion Algorithm using the graph approach is summarized below in the Insertion Algorithm 2 (IA2 for short)

Insertion Algorithm 2

1. The new polytask t_* appears.
2. Choose an insertion period for t_* that is not already checked, and that does not lead to violate the deadline D_*. If there has none, then it is not possible to insert t_*. END.
3. For $i = 1$ to n_* do :
   (a) If it is not possible to insert t_*^i in the known schedule without postponing at least one task of S, then :
      i. Create (if needed) SFG(S) using the SFG Algorithm.
      ii. Compute the maximal amount of time m we may postpone the task of S that is in conflict with t_*^i, using result 3.
      iii. If postponing the task scheduled after t_*^i by a period m does not allow the insertion of t_*^i, then go to 2
      iv. Else postpone the task scheduled after t_*^i of the exact time needed to insert t_*^i.
      v. Update SFG(S). Go to 3.
   (b) Else insert t_*^i. Update SFG(S). Go to 3.
4. The insertion of t_* is done : END

The following result states about the calculability of the Insertion Algorithm 2.

Result 4

The Insertion Algorithm 2 finishes.

Proof 4

1. The known schedule S contains a finite number n of polytasks, and then a finite number of idle periods between the subtasks of these tasks. So the set of possible starting instants for the new polytask in S is finite.
2. The random polytask has a finite number of subtasks.

As a consequence the number of iterations of the algorithm is finite. This completes the proof □

A.4.4 Example

We consider a schedule of five tasks t_i, described in table A.3. IP stands for Idle Period. These events are given chronologically.

TAB. A.3 – Example schedule 2.

Task Id	t_1^1	IP	t_2^1	IP	t_1^2	IP	t_2^2	t_5^1	IP	t_3^1
Execution time	5.0	0.1	2.2	0.2	3.0	0.8	2.0	1.0	2.1	2.0
Task Id	IP	t_4^1	IP	t_3^2	IP	t_5^2	t_4^2	t_5^3	IP	t_1^3
Execution time	2.5	1.0	3.0	3.0	2.0	1.0	2.0	1.3	5.0	2.5

We want to insert a new polytask, that has three subtasks, in the known schedule. The first subtask of this polytask has a duration of 1.0. The second subtask starts 4 units of time after the end of the first one, and lasts 2 units of time. The third subtask starts 2.2 units of time after after the end of the second one and lasts 0.1. The deadline D_* is fixed at time 26.

We compute SFG(S), and we obtain the graph in figure A.8.

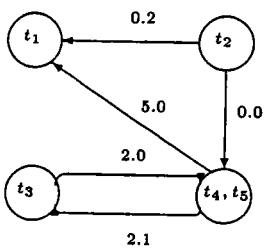


FIG. A.8 – Original SFG(S) for example 2.

We then compute the set of possible insertion points. The first one is between task t_1 and task t_2. The idle period is 0.1, which is not enough for inserting

a task that lasts 1.0. We run the result 3, and know that the maximal postponement we may apply to t_2 is 0.3. It is then not possible to set the first subtask here, neither that it is possibble in the second idle time for the same reason. The third idle period has a duration of 0.8, and the maximal postponement we may apply (0.2) allows us to insert the first subtask here. We update SFG(S), and we find one negative arc. We have to correct it by adding 0.2 to the incoming arcs of node t_4, t_5 and to remove 0.2 of the outcoming arcs of node t_4, t_5. The new SFG(S) is given in figure A.9.

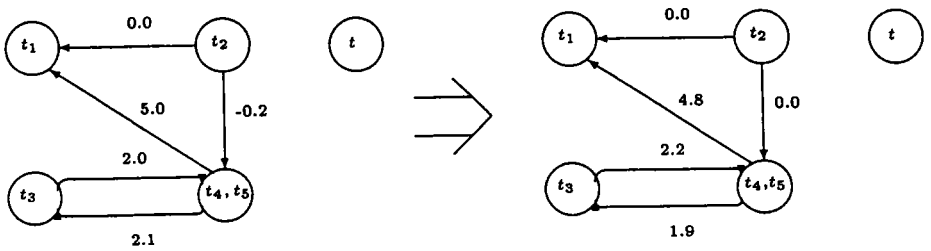


FIG. A.9 – Corrections to SFG(S).

The second subtask has to be inserted 4.0 units of time after completion of the first one. It enters in conflict with task t_3, and requires 1.1 extra units of time. The result 3 indicates that it is possible to postpone t_3 by 2.2 units of time. SFG(S) modifications are given in figure A.10

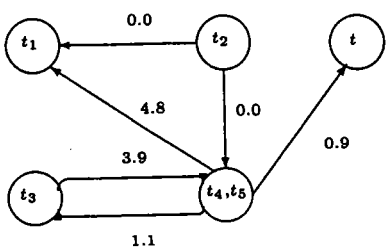


FIG. A.10 – Corrections to SFG(S).

The final subtask of the new polytask has to be placed 2.2 units of time after completion of the second subtask. It lasts 0.1 units of time, and so it may be inserted in the schedule without postponing any other tasks.

A.5 Complexity

This section is devoted to the computation of the complexity of IA1 and IA2.

A.5.1 Complexity of IA1

In this subsection, we will assume that the average number of subtasks of a polytask is m.

The average complexity of the initialization part of this algorithm (i.e. Step 1.) is $(1+m)+m = 1+2m$ operations.

The second step starts with a test whose average complexity is : $\frac{\sum_{i=1}^{n} m+m-1}{2} = \frac{(n+1)m-1}{2}$ The inner part of step 2 requires m operations, and has to be performed $(n-1)m$ times.

The total complexity is then in $O(\frac{m^2n^2}{2})$.

A.5.2 Complexity of IA2

The complexity of the Dijkstra algorithm is in $O(n * \log(\frac{m}{n}))$ (see [66]), where n is the number of nodes, and m the number of arcs.

The complexity of the SFG algorithm is computed from :

- Step 1 : 4 operations.
- Step 2 : $\sum_{k=1}^{n}(\frac{m}{n} * (n-k) + 3mn + n) = n + \frac{(7n-1)m}{2}$
- Step 3 : $3m$ operations per couple of nodes to merge.

The complexity of the SFG algorithm is then in $O(nm)$

The insertion algorithm 2 complexity is computed as follows :

- Step 1 : at most $n-1$ iterations.
- step 2 : its first execution costs $nm + nln(\frac{m}{n}) + 3m + 1$, and all the others $ln(\frac{m}{n}) + 3m + 1$

So, the total complexity is $((n-1)\sum_{i=1}^{m}) * (nln(\frac{m}{n}) + 3m + 1) + nm + nln(\frac{m}{n}) + 3m + 1$

which is in $O(mn(n\ln(\frac{m}{n}) + m))$

A.6 Conclusion

In this article, we provide two algorithms. The first one allows to insert polytasks with unitary subtask duration in a known schedule of polytasks. This algorithm may be easily extended to the case of subtask that has integer duration.

The second algorithm considers the case of subtasks with non-integer duration.

These algorithms are a first approach, because they just provide a possible insertion, if any, without optimization of a criterion. The complexity of the algorithms does not allow to use them in a real-time environment.

These two points will be the subject of further work.

Annexe B

Algorithme optimal pour l'insertion d'une bitâche dans un ordonnancement de tâches simples.

L'algorithme Optimal est présenté ici sous une forme complète (voir 5.5).

Algorithme Optimal

Données : Les variables utilisées sont décrites dans la section 5.2. Les variables temporaires sont données ci-dessous.

Variables temporaires :

$overcost$	: entier, surcoût calculé
k	: entier, index de la tâche après laquelle on insère A_1
k_{max}	: entier, index de la dernière tâche après laquelle on peut insérer A_1
r	: entier
r_{prime}	: entier, index de la première tâche exécutée après A_2
$finished$	: booléen
$beginning$	: booléen
$\mu_i^{'}$	: entier , nouvel instant de départ de la tâche i.

Résultats :

$overcost^*$	: entier, plus petit surcoût calculé
μ_i^*	: entier, instants de départ menant à la plus faible hausse du critère.

Début

1. Initialisation de l'algorithme

   $k = 1$

   $k_{max} = 1$

   $overcost^* = +\infty$

2. Calcul de k_{max}

   (a) *Tant que* $((\mu_{k_{max}} + t_{k_{max}} + \theta_1 + L + \theta_2 \leq D)$ *et* $(k_{max} < n))$ *faire*

   $k_{max} = k_{max} + 1$

(b) *si* $(\mu_{k_{max}} + t_{k_{max}} + \theta_1 + L + \theta_2 > D)$ *alors* $k_{max} = k_{max} - 1$

3. Initialisation du programme pour le calcul des nouveaux instants de début et surcoût pour le cas où l'on insère la bitâche juste après la tâche $k + r - 1$

   $r = 1$

   $finished = faux$

   $overcost = 0$

4. Calcul du surcoût et des nouveaux instants de départ pour les tâches situées entre les deux sous-tâches de la bitâche.

   (a) *Si* $((finished = vrai)$ *ou* $(\mu_{k+r} + t_{k+r} > \mu_k + t_k + \theta_1 + L))$ *Aller en* 5

   (b) *Si* $(r = 1)$ *alors* $\mu'_{k+1} = \max(\mu_{k+1}, \mu_k + t_k + \theta_1)$
   sinon $\mu'_{k+r} = \max(\mu_{k+r}, \mu'_{k+r-1} + t_{k+r-1})$

   (c) *Si* $(\mu'_{k+r} + t_{k+r} > \mu_k + t_k + \theta_1 + L)$ *alors*

      i. $finished = vrai$

   sinon

      i. $overcost = overcost + ((\mu_{k+r} + t_{k+r} - d_{k+r})^- - \mu_{k+r} + \mu'_{k+r})^+$

      ii. $r = r + 1$

      iii. *Si* $(k + r > n)$ *aller en* 5

   (d) *aller en* 4

5. Consigner l'index de la première tâche suivant la bitâche.

   $r_{prime} = r$

   $beginning = vrai$

   $finished = faux$

6. Calculer le surcoût et les nouveaux instants de départ pour les tâches situées après la bitâche.

   (a) *Si* $finished = vrai$ *Aller en* 8

   (b) *Si* $(beginning = vrai)$ *alors*

      i. $\mu'_{k+r} = \max(\mu_{k+r}, \mu_k + t_k + \theta_1 + L + \theta_2)$

      ii. $beginning = faux$

sinon $\mu'_{k+r} = \max(\mu_{k+r}, \mu'_{k+r-1} + t_{k+r-1})$

(c) *Si* $(\mu'_{k+r} = \mu_{k+r})$ *alors* $finished = vrai$
sinon

i. $overcost = overcost + ((\mu_{k+r} + t_{k+r} - d_{k+r})^- - \mu_{k+r} + \mu'_{k+r})^+$

ii. $r = r + 1$

iii. *Si* $(k + r > n)$ *alors* $finished = vrai$

(d) *Aller en* 6

7. Optimisation du critère

(a) $nbtache = 1$
$temps = 0$

(b) *Si* $((temps + \mu_{k+nbtache} < \mathrm{L}))$ *alors*

i. $temps = temps + \mu_{k+nbtache}$
$nbtache = nbtache + 1$

ii. *Aller en* 7b

(c) $nbtache = nbtache - 1$

(d) Cette partie de l'algorithme est utilisée uniquement s'il est possible d'insérer une tâche additionnelle entre A_1 et A_2 en repoussant plus tard l'instant de début de A_1. *Si* $(nbtache > origine)$ *Alors*

i. $ajout = nbtache - origine$

ii. *Pour* $cpt = 1$ *à* $ajout$

A. $maxy = origine + cpt$
$\mu'_{k+1} = \max(\mu_{k+1}, \mu_{k+maxy} + t_{k+maxy} - L)$
$ovcost2 = ((\mu_{k+1} + t_{k+1} - d_{k+1})^- - \mu_{k+1} + \mu'_{k+1})^+$
$r = 2$

B. $\mu'_{k+r} = \max(\mu_{k+r}, \mu_{k+r-1} + t_{k+r-1})$
$ovcost2 = ovcost2 + ((\mu_{k+r} + t_{k+r} - d_{k+r})^- - \mu_{k+r} + \mu'_{k+r})^+$
$r = r + 1$
Si $r \leq maxy$ *Aller en* 7(d)iiB

C. $finished = faux$
$r = nbtache + 1$
$\mu'_{k+r} = \max(\mu_{k+r}, \mu_{k+origine+1} + t_{k+origine+1} + \theta_2)$

D. *Si*($finished$ = vrai) *Aller en* 7(d)iiF

E. $\mu'_{k+r} = \max(\mu_{k+r}, \mu'_{k+r-1} + t_{k+r-1})$
Si $\mu'_{k+r} = \mu_{k+r}$ $finished$ =*vrai*
Sinon $ovcost2 = ovcost2 + ((\mu_{k+r} + t_{k+r} - d_{k+r})^- - \mu_{k+r} + \mu'_{k+r})^+$
$r = r + 1$
$finished = finished$ *ou* $(k + r \geq n)$

F. *Si* ($ovcost2 < overcost$) $overcost = ovcost2$

8. Si le surcoût obtenu lorsque la bitâche est insérée après a_k est le plus faible obtenu jusqu'à maintenant alors nous le consignons.

(a) *Si* ($overcost < overcost^*$) *alors*

i. $overcost^* = overcost$

ii. *Pour* $i = 1$ *à* k *faire* $\mu_i^* = \mu_i$

iii. $\mu_{k+1}^* = \mu_k + t_k + \theta_1$

iv. *Pour* $i = k + 1$ *à* $k + r_{prime} - 1$ *faire* $\mu_{i+1}^* = \mu'_i$

v. $\mu_{k+r_{prime}+1}^* = \mu_k + t_k + \theta_1 + L + \theta_2$

vi. *Pour* $i = k + r_{prime}$ *à* $k + r$ *faire* $\mu_{i+2}^* = \mu'_i$

vii. *Pour* $i = k + r + 1$ *à* n *faire* $\mu_{i+2}^* = \mu_i$

(b) *Si* $k < k_{max}$ *Alors*

i. $k = k + 1$

ii. *Aller en* 3

FIN

Annexe C

Listings de la simulation de radars multifonctions.

Cette annexe est composée de deux sections. La première section contient les listings des différents modules composant le programme de simulation de radars multifonctions. Les bases de fonctionnement de ce programme sont décrites dans le chapitre 7. La seconde section contient des photos d'écran du logiciel.

Ce programme a été réalisé en Java 1.3.1.

C.1 Les listings

Le programme de simulation, réalisé suivant la technique de programmation objet, se décompose en 16 modules principaux.

Le module principal est le module *simulation.java* (voir la section C.1.9).

C.1.1 barre.java

Ce module est consacré à la représentation graphique des résultats. (voir un exemple de sortie dans la section C.2.3)

```
import java.awt.*;
import java.awt.event.*;
import java.util.Vector ;
import java.awt.image.* ;

public class barre extends Frame implements ActionListener {

   private tempusFugit ligne = null ;
   private String titre = null ;
   MyCanvas temoin = new MyCanvas("Témoin") ;
   private final String OUVRIRIMAGE  = "Afficher" ;
   private final String QUITTER      = "Quitter";
```

```
public barre(String titre,tempusFugit tf) {
   ligne = tf.backup() ;
   this.titre = titre ;
   Panel RPanel = new Panel();
   setLayout(new BorderLayout());

   MenuBar mb = new MenuBar();
   Menu m = new Menu("Fichier");
   MenuItem fileMenuItem = new MenuItem(OUVRIRIMAGE);
   fileMenuItem.addActionListener(this);
   m.add(fileMenuItem);
   m.add(new MenuItem("-"));
   fileMenuItem = new MenuItem(QUITTER);
   fileMenuItem.addActionListener(this);
   m.add(fileMenuItem);
   mb.add(m);
   setMenuBar(mb);

//Création du menu de contrôle du programme
   Panel LPanel = new Panel() ;
   LPanel.setLayout(new GridLayout(1,2));
   LPanel.add(temoin);
   add("Center",LPanel) ;

   addWindowListener(
                    new WindowAdapter() {
                       public void windowClosing(WindowEvent e) {
                          System.exit(0);}});}


public void update(tempusFugit tf){
   tranche t = null ;
   for (int i = 0 ; i < tf.freeSize() ; i++)
    ligne.freeAddElement(tf.freeElementAt(i)) ;
   for (int i = 0 ; i < tf.scheSize() ; i++)
    ligne.scheAddElement(tf.scheElementAt(i)) ;}
public void actionPerformed(ActionEvent event) {
   final String Tache =
    new String(((MenuItem)event.getSource()).getLabel()) ;
   if (Tache.equals(QUITTER)) System.exit(0) ;
   else if (Tache.equals(OUVRIRIMAGE)) {
      (new Anime(temoin,ligne)).start() ;}
   else System.err.println("|" + Tache + "|") ;}
public void run(){
   barre window = this ;
   window.setTitle(titre);
   window.pack();
   window.setVisible(true); }

/* ************************************************************************/


public class Anime extends Thread{
   private MyCanvas window = null ;
   private tempusFugit prog = null ;
   public Anime(MyCanvas truc, tempusFugit t){
      window = truc ;
      prog = t ;}
   private void traceRectE(int x, int y, int lig, int col,
       Color color,Color color2,String t1, String t2){

      traceRect(x,y,lig,col,color) ;
      window.write(t1,x+5,y+20) ;
      if (!(t2.equals(""))) window.write('(' + t2+ ')',x+5,y+40) ;   }
   private void traceRect(int x, int y, int lig, int col, Color color){
      for (int i = 1 ; i < (lig-1); i++) window.setH(x,y+i,color,col) ;
      window.setH(x,y,Color.white,col) ;
      window.setH(x,y+lig-1,Color.white,col) ;
      window.setV(x+col-1,y,Color.white,lig-1) ;
      window.setV(x,y,Color.white,lig-1) ;}
```

```
        private void traceRectBarreE(int x, int y, int lig, int col,
         Color color,Color color2,String t1, String t2){
           traceRect(x,y,lig,col,color) ;
           window.line(x,y,x+col-1,y+lig-1,Color.white) ;
           window.line(x,y+lig-1,x+col-1,y,Color.white) ;
           window.write(t1,x+1,y+20) ;
           window.write('(' + t2+ ')',x+1,y+40) ;    }

        public void run() {
           window.initImage() ;

           // determiner le nombre de trucs a afficher
           int nombre = prog.nbEven() ;
           int nombreTot = prog.nbSche() ;
           int resolCol = window.getMaxX() ;
           int resolLig = 50 ;
        // creer la table des couleurs ;
           Color[] couleurs = new Color [nombreTot] ;
           boolean legende[] = new boolean [nombreTot] ;
           window.write("Nous déplorons la présence de " +
            nombre + " évènements, pour " + nombreTot +
            " tâches",10,15,Color.white) ;
           for (int i = 0 ; i < nombreTot ; i++) {
              int tmp = (int)(255 / (nombreTot+2)) * (i+2);
              couleurs[i] = new Color(tmp,tmp,tmp) ;
              legende[i] = false ;}
        // la liste des noms
           long noms[] = prog.getNames() ;
        // affichage des bazars
           int maxLegByLine = (int) (window.getMaxX()/(200)) ;
           int k = 7 ;
           if (k > nombre) k = nombre ;
           for (int ligne = 0 ; ligne < (50 * k) ; ligne += 50){
              int nbLocal = (int)(nombre / k) ;
              if (ligne == ((k-1)*50)) nbLocal = nombre - (k-1) * nbLocal ;
              int carre = (int)(window.getMaxX() / nbLocal) ;
              for (int i = 0 ; i < nbLocal ; i++) {
                 tranche t = prog.getNext() ;
                 if (t.lID() == -1){
                 // affichage d'un temps libre
                    traceRectE(i*carre, ligne + 20, 50, carre,
                    Color.black,Color.white,"LIBRE",
                    Long.toString(t.duree())) ;}
                 else {
                 // trouver la couleur
                    int c = 0 ;
                    while ((noms[c]) != t.lID()) c++ ;
                    Color color = couleurs[c] ;
                 // affichage de la legende
                    if (!legende[c]) {
                       int x = (c % maxLegByLine) * 200 ;
                       int y = window.getMaxY() - 25 *
                                 ((c / maxLegByLine) + 1)
;                      traceRect(x,y, 20, 200, color) ;
                       window.write(t.aff(), x + 5, y + 15) ;
                       legende[c] =  true ;}

                    if ((t.name().indexOf("FIN")) == -1)
                       traceRectE(i*carre, ligne + 20, 50, carre,
                       color,Color.white,Long.toString(t.lID()),
                       Long.toString(t.duree())) ;
                    else
                       traceRectBarreE(i*carre, ligne + 20, 50,
                       carre, color,Color.white,Long.toString(t.lID()),
                       Long.toString(t.duree())) ;}}}

         // affichage dans la fenetre ; fin du processus
         window.setImage() ;}}
      private class MyCanvas extends Canvas {
```

```
private String title ;
private Image image ;
private Image fond ;
Image offscreen = null ;
Graphics offgraphics;

final int resX = 1600 ;
final int resY = 600 ;

public void initImage() {
   offscreen = createImage(resX,resY) ;
   offgraphics = offscreen.getGraphics();
   offgraphics.setFont(getFont());
   offgraphics.setColor(Color.black);
   offgraphics.fillRect(0, 0, resX-1, resY-1);
   FontMetrics fm = offgraphics.getFontMetrics();}
public void setH(int i, int j, Color c, int length){
   offgraphics.setColor(c) ;
   offgraphics.drawLine(i,j,i+length,j);}
public void setV(int i, int j, Color c, int length){
   offgraphics.setColor(c) ;
   offgraphics.drawLine(i,j,i,j+length);}
public void line(int x1,int x2, int y1,int y2, Color c){
   offgraphics.setColor(c) ;
   offgraphics.drawLine(x1,x2,y1,y2);}
public void write(String s, int lig, int col,Color c){
   offgraphics.setColor(c) ;
   setFont(new Font("Helvetica", Font.PLAIN, 5));
   offgraphics.drawString(s, lig, col);}
public void write(String s, int lig, int col){
   setFont(new Font("Helvetica", Font.PLAIN, 5));
   offgraphics.drawString(s, lig, col);}
public void fillpoly(int[] x, int[] y, int nb, Color c){
   offgraphics.setColor(c) ;
   offgraphics.fillPolygon(x,y,nb);}
public void setImage () {
   image = offscreen ;
   repaint() ;}
public Image getImage() {
   return image ;}
public void resetImage() {
   image = null ; }
public MyCanvas (String titre) {
   super() ;
   title = new String (titre) ;
   image = null ;}
public Dimension getMinimumSize() {
   return new Dimension(resX,resY);}
public Dimension getMaximumSize() {
   return getMinimumSize();}
public Dimension getPreferredSize() {
   return getMinimumSize();}
public int getMaxX(){
   return resX;}
public int getMaxY(){
   return resY;}
public void paint(Graphics g) {
   int w = getSize().width;
   int h = getSize().height;
   g.drawRect(0, 0, w - 1, h - 1);
   if (getImage() != null) g.drawImage(image, (w - resX)/ 2,
                              (h - resY) / 2, this);
   else {
      g.setFont(new Font("Helvetica", Font.PLAIN, 30));
      g.drawString("Pas D'image", 80, 100);}}}}
```

C.1.2 coords.java

Ce module est consécré au stockage des coordonnées d'une cible.

```
public class coords {

private int lig = 0 ;
private int col = 0 ;
private long dist = 0 ;

public  coords initCoords(coords c){
lig = c.getLig() ;
col = c.getCol() ;
dist = c.getDist() ;
return this ;}

public coords(int lig, int col, long dist){
this.lig = lig ;
this.col = col ;
this.dist = dist ;}

public coords(coords c){initCoords(c);}

public int getLig() {return lig ;}
public int getCol() {return col ;}
public long getDist() {return dist ;}
public String toString(){return "( coords :"+getLig()+","+getCol()+
" distance = " + getDist() +" )" ;}}
```

C.1.3 coordsTime.java

Ce module utilise coords.java et lui associe un temps.

```
public class coordsTime {


   private coords c = null ;
   private time t   = null ;

   public coordsTime(coords c, time t){
      init(c, t);}

   public coordsTime(coordsTime c){
      init(c.getCoords(), c.getTime());}

   public void init(coords c, time t){
      this.c = new coords(c) ;
      this.t = new time(t);}

   public coords getCoords() {
      return c ;}
   public time getTime() {
      return t ;}
   public String toString() {
      return "{"+getCoords()+" "+getTime()+"}" ;}}
```

C.1.4 FileReader.java

Ce module effectue la lecture de données constituées pendant l'exécution.

```
import java.util.Vector ;
import java.io.* ;
public class FileReader{

   private class entree{
      private String nom = null ;
      private String data = null ;
```

```
   public entree(String nom, String data) {
      this.nom = new String(nom) ;
      this.data = new String(data) ;}

   public boolean cherche(String c){
      return (c.equals(nom)) ;}
   public String get() {
      return data ;}
   public String toString(){
      return ("[nom : "+nom+ "  | Data : "+data+"]") ;}}


public class setOf{

   private Vector tas = null ;

   public setOf(String s) throws Exception{
      tas = new Vector() ;
      initFile(s) ;}

// TOUT LE PRIVE

   private entree get(int i){
      return ((entree)tas.elementAt(i)) ;}
   private void set(String nom,String data){
      tas.addElement(new entree(nom, data)) ;}
   private void remove(int i){
      tas.removeElementAt(i) ;}
   private int extraire(String nom){
      int i = 0 ;
      while((i < size())&&(!(get(i).cherche(nom)))) i++ ;
      if (i == size()) i = -1 ;
      return i ;}
   private String extraction(String nom){
      int i = extraire(nom) ;
      if (i==-1)
         throw new NumberFormatException(nom+
               " est inconnu !!!\r\n");
       else {String retour = get(i).get() ;
         remove(i) ;
         return retour ;}}
   public String toString() {
      String retour = "[" ;
      for (int i = 0 ; i < size() ; i++)
            retour += get(i).toString() ;
      retour +="]" ;
      return retour ;}

   public void initFile(String s) throws Exception{
      File sin = new File(s) ;
      FileInputStream in = null ;
      try{in  = new FileInputStream(sin);}
         catch(Exception e) {
            throw new Exception("PAS GRUMPF !") ;}
      String tampon = "" ;
      char c = ' ' ;
      try{
         while (in.available() != 0) tampon += ((char)in.read()) ;
         in.close() ;}
         catch(Exception e) {
            throw new Exception("PAS GRUMPF 2 !") ;}

   // conversion du String en elements comprehensibles.
      String n1 = "" ;
      String n2 = "" ;
      int fin = 0 ;
   // suppression des retours
      while ((fin = tampon.indexOf('\r')) != -1){
         n1 = tampon.substring(0,fin) ;
```

```
            n2 = tampon.substring(fin+1) ;
            tampon = n1+ n2 ;}
         while ((fin = tampon.indexOf('\n')) != -1){
            n1 = tampon.substring(0,fin) ;
            n2 = tampon.substring(fin+1) ;
            tampon = n1+ n2 ;}
         while (tampon.length() > 0){
            while ((tampon.charAt(0)) == ' ')
             tampon = tampon.substring(1) ;
            fin = tampon.indexOf(' ') ;
            n1 = tampon.substring(0,fin) ;
            tampon = tampon.substring(fin+1) ;
            while ((tampon.charAt(0)) == ' ')
             tampon = tampon.substring(1) ;
            fin = tampon.indexOf(' ') ;
            if (fin == -1) {
               n2 = tampon ;
               tampon = "" ;}
            else{n2 = tampon.substring(0,fin) ;
               tampon = tampon.substring(fin+1) ;}
            set(n1,n2) ;}}


   // TOUT LE PUBLIC
      public int size(){
         return tas.size() ;}
      public String getString(String nom){
         return extraction(nom) ;}
      public double getDouble(String nom){
         return Double.parseDouble(extraction(nom)) ;}
      public long getLong(String nom){
         return Long.parseLong(extraction(nom)) ;}
      public int getInt(String nom){
         return Integer.parseInt(extraction(nom)) ;}}

   private setOf datas = null ;
   public FileReader(String s) throws Exception{
      datas = new setOf(s) ;}
   public double getDouble(String nom){
      return datas.getDouble(nom) ;}
   public long getLong(String nom){
      return datas.getLong(nom) ;}
   public int getInt(String nom){
      return datas.getInt(nom) ; }
   public boolean getBoolean(String nom){
      return (datas.getString(nom)).equals("true") ;}
   public int reste(){
      return datas.size() ;}}
```

C.1.5 grilleTask.java

Ce module sert à la modélisation du ciel pour les tâches de surveillance.

```
public class grilleTask{

   private grillePT PT = null ;
   private grilleMarque M = null ;
   private int lig = 0 ;
   private int col = 0 ;

   public grilleTask(int lig , int col, int Max, double proba) {
      PT = new grillePT(lig,col, Max,proba) ;
      M = new grilleMarque(lig,col) ;
      this.lig = lig ;
      this.col = col ;}
```

```
// methodes pour gerer le marquage
   public boolean ligAvailable(int col){
      return (M.resteDesTrucs(col)) ;}

   public int getFirstFree(int col){
      return M.getFirstFree(col) ;}
   public void setAJouer(int lig, int col) {
      M.onJoue(lig,col);}
   public int reset(int col) {
      return M.update(col) ;}

// methodes pour les taches de probe

   public void inc(int lig, int col) {
      PT.inc(lig,col) ;}
   public void dec(int lig, int col){
      PT.dec(lig,col) ;}
   public boolean joue(int lig, int col, double sonde){
      return PT.joue(lig,col,sonde) ;}

   private class grillePT{

      private int maGrille [][] = null ;
      private int lig = 0 ;
      private int col = 0 ;
      private int max = 0 ;
      private double proba = 0.0 ;

      public grillePT(int lig, int col, int max, double proba){
         maGrille = new int [lig][col] ;
         for (int i = 0 ; i < lig ; i++)
            for (int j = 0 ; j < col ; j++)
               set(i,j,0) ;
         this.lig = lig ;
         this.col = col ;
         this.max = max ;
         this.proba = proba ;}

      private int get(int lig, int col){
         return maGrille[lig][col] ;}

      private void set(int lig, int col, int nb){
         maGrille[lig][col] = nb ;}

      public void inc(int lig, int col){
         int temp = get(lig, col) ;
         if (temp < max ) set(lig,col,temp+1) ;
         else  throw new outOfBoundsException("TROP D'OBJETS") ;}
      public void dec(int lig, int col){
         int temp = get(lig, col) ;
         if (temp > 0 ) set(lig,col,temp-1) ;
         else  throw new outOfBoundsException("PAS ASSEZ D'OBJETS") ;}

      public boolean joue(int lig, int col, double test){
         boolean retour = false ;
         if (get(lig,col) < max) retour = (test < proba) ;
         return retour ;}}


    private class grilleMarque{
       private class Marque{
          private int m  = 0 ;

          public Marque(int toto) {
             m = toto ;}
          public int get(){
             return m ;}
          public boolean equals(Marque m2){
             return (get() == m2.get()) ;}
```

```
    public String toString(){
       return Integer.toString(m) ;}}
 private class grille{
    private class ligne {
       private Object maLigne[] ;
       private int length = 0 ;
       public ligne(int l) {
          if ((l>=64000)||(l<0))
             throw new outOfBoundsException(l) ;
          else
          { length = l ;
             maLigne = new Object[getLength()] ;
             for (int i = 0 ; i < getLength() ; i++)
                set(null,i) ;}}
       private boolean inBounds(int i) {
          return ((i>=0)&&(i < getLength())) ;}
       public int getLength() {
          return length ;}
       public String toString(){
          String toto = new String() ;
          toto = "[" ;
          for (int i = 0 ; i < (getLength()-1) ; i++)
                   toto += get(i) + "," ;
          toto += get(getLength()) + "]";
          return toto ;}
       public void set(Object o, int i){
          if (!inBounds(i)) throw new outOfBoundsException(i) ;
          else if (o == null) maLigne[i] = null ;
          else maLigne[i] = o ;}
       public Object get(int i){
          if (inBounds(i))
             return (maLigne[i]) ;
          else
          throw new outOfBoundsException(i+">"+getLength()) ;}}
    private int lig = 0 ;
    private ligne maLigne =  null ;
    public grille (int lig, int col){
       maLigne = new ligne(lig) ;
       for (int i = 0 ; i < lig ; i++)
             maLigne.set(new ligne(col),i) ;
       this.lig = lig ;}
    public grille (int lig, int col, Object init){
       maLigne = new ligne(lig) ;
       for (int i = 0 ; i < lig ; i++)
             maLigne.set(new ligne(col),i) ;
       this.lig = lig ;
       for (int i = 0 ; i < lig ; i++)
          for (int j = 0 ; j < col ; j++)
             set(init, i,j) ;}
    public Object get(int lig,int col) {
       ligne toto = getLigne(lig) ;
       return (toto.get(col));}
    private ligne getLigne(int i) {
       return (ligne)maLigne.get(i) ;}
    public void set (Object o, int lig, int col) {
       ligne toto = getLigne(lig) ;
       toto.set(o,col) ;}
    public String toString(){
       String toto = new String() ;
       toto = "[" ;
       for (int i = 0 ; i < lig ; i++)
           toto += ((ligne)maLigne.get(i)).toString() + "," ;
       toto += ((ligne)maLigne.get(lig)).toString() + "]";
       return toto ;}}

private grille maGrille = null ;
private int lig = 0 ;
private int col = 0 ;

private final Marque Joue = new Marque(-1) ;
```

```
    private final Marque AJouer = new Marque(0) ;
    private final Marque enJoue = new Marque(1) ;

    public int tag [] = null ;

    public grilleMarque(int lig, int col){
       this.lig = lig ;
       this.col = col ;
       maGrille = new grille(lig, col,AJouer) ;
       tag = new int [col] ;
       for (int i = 0 ; i < col ; i++) tag [i] = lig;}

    private Marque get(int lig, int col){
       return ((Marque)maGrille.get(lig,col));}

    private void set(int lig, int col, Marque m){
       maGrille.set(m,lig,col);}

    private boolean test(int lig, int col, Marque m){
       return  ((get(lig,col)).equals(m));}

    private boolean dejaVisite(int lig, int col){
       return test(lig, col, Joue);}

    private boolean AJouer(int lig, int col){
       return test(lig, col, AJouer);}

    private boolean enJoue(int lig, int col){
       return test(lig, col, enJoue);}

    public boolean resteDesTrucs(int col){
       return (tag[col] > 0) ;}

    public int getFirstFree(int col){
       if (!resteDesTrucs(col))
          return -1 ;
       else {
          int i = 0 ;
          while (!AJouer(i,col)) i++ ;
          return i ;}}

    public void onJoue(int lig, int col){
       if (AJouer(lig,col)) {
          set(lig, col, enJoue) ;
          tag[col] -= 1 ;}
       else throw new outOfBoundsException("DEJA OCCUPE !") ; }

    public int update(int col){
       int retour = tag[col] ;
       for (int i = 0 ; i < lig ; i++)
          if (enJoue(i,col)) set(i,col, Joue) ;
          else if (dejaVisite(i,col)){
             set(i,col,AJouer) ;
             tag[col] += 1 ;}
       return retour ;}
}}
```

C.1.6 history.java

Ce module stocke un évènement (Découverte d'une nouvelle cible, perte d'une cible, ...).

```
public class history {

   private String nom = null ;
```

```
    private long debut = 0 ;
    private long duree = 0 ;
    private int lig = 0 ;
    private int col = 0;

    public history(String nom, long debut, long duree, int lig, int col){
       this.nom = new String(nom) ;
       this.debut = debut ;
       this.duree = duree ;
       this.lig = lig ;
       this.col = col ;}

    public String toString(){
       return ( "\r\n"+  debut + " : " + nom + " " + duree
              + " " + lig + " " + col) ;}

    public int getLig(){
       return lig ;}

    public long duree(){
       return duree;}

    public String nom(){
       return nom;}}
```

C.1.7 outOfBoundsException.java

Gestion de l'exception *outOfBounds.*

```
public class outOfBoundsException extends IndexOutOfBoundsException {

    public outOfBoundsException() {super();}

    public outOfBoundsException(int index) {
super("Array index out of range: " + index);}

    public outOfBoundsException(String s) {super(s);}
}
```

C.1.8 sche2.java

Module contenant l'algorithme d'ordonnancement.

```
import java.util.Vector ;

public class sche2 {

   private tempusFugit tf = null ;
   private tempusFugit bakeupe = null ;

   private double deperdition = 0.00000000000000000001 ;
   private double cross = 0.5 ;
   private double c = 300000000 ;
   private long as = 0 ;
   private final int updateCol = 10 ;
   private int cptCol = 0 ;
   private barre maBarre = null ;

   private boolean entrelace = true ;

   public sche2 (long t1, long t2, double dep, double cross,
            double c,long as, boolean entrelace){
      tf = new tempusFugit() ;
```

```
      tf.freeAddElement(new tranche(t1,t2*2,"",0,0,-1)) ;
      deperdition = dep ;
      this.cross = cross ;
      this.c = c ;
      this.as = as ;
 this.entrelace = entrelace ;}

   public void reInit(long tps){
      tf.tri() ;
      tranche f =  null ;
      long time = 0 ;
      if (tf.scheSize() != 0) {
         f = tf.scheElementAt(tf.scheSize()-1) ;
         time = f.fin() ;}
      f = tf.freeElementAt(tf.freeSize()-1) ;
      long t2 = f.fin() ;
      if (time > t2)
         tf.freeAddElement(new tranche(time, tps,"",0,0,-1)) ;
      else
         tf.freeAddElement(new tranche(t2, tps,"",0,0,-1)) ;}

   public void backup(){
      bakeupe = tf.backup() ;}

   public void restore(){
      tf = bakeupe.backup() ;}



// chercher un creneau de temps flottant ou pas
// i.e : 1ere occurence ou pas
// contenant la tache a jouer. As + Ls + Bs

// ca revient a chercher 2 blocs de temps libre,
// chacun contenant une longueur As pour la tache.

   private long max (long a, long b) {
      if (a>b)
         return a ;
      else
         return b;}
   private long min (long a, long b) {
      if (a<b)
         return a ;
      else
         return b;}


   private long intersection (tranche t1, tranche t2){
      if ((t1.debut()> t2.fin())
         ||(t2.debut() > t1.fin()))
         return 0 ;
      else
         return ((min(t1.fin(),t2.fin())-
                     (max (t1.debut(),t2.debut())))) ;}

// extrait de la liste le 1er libre dans la plage horaire
// qui soit suffisamment grand.

   private int extraire(long debut, long fin, long as, int num){
      tranche t = new tranche (debut,fin,"",0,0,-1) ;
      int retour = 0;
      num = -1 ;
      while ((retour < tf.freeSize()) &&
               (intersection(t,(tranche)tf.freeElementAt(retour))<as))
               retour++ ;
      if ((retour == tf.freeSize())
         ||((retour == num )&&
               ((intersection (t,(tranche)tf.freeElementAt(retour))<as))))
         return -1 ;
```

```
      else
         return retour ;   }


// insertion des taches de probe.
   public int inserableProbe(long as, long bs){
      int retour = -1 ;
      int t = 0 ;
      while ((t < tf.freeSize())&&(retour == -1)){
         tranche tr = tf.freeElementAt(t) ;
         if ((tr.duree() >= as)&&((tr.debut()+as) < bs)) retour = t ;
         else t++ ;}
      return retour ;}

   public long insererProbe(int i, long as, int lig,
       int col, temperature tantpis,
       long finale,long id){
      tranche t = tf.freeElementAt(i) ;
      tranche proby = new tranche(t.debut(), t.debut()+ as,
                     "Surveillance ",lig,col,id) ;
      backup() ;
      tf.freeRemoveElementAt(i) ;
      tf.upfree(t,proby) ;
      tf.scheAddElement(proby) ;
      tf.evalue(tantpis, finale, as)  ;
      if (tantpis.claque()) {restore() ;
         return (-1) ;}
      else
         return proby.debut() ;}
   public long insere(String nom, int lig, int col, long dist,
            long time,  boolean imp, temperature tantpis,
            long limite, long id){
      long as =
      (long)(deperdition * Math.pow((double) dist,(long)4) * cross) ;
      as++ ;
      long ls = ((long)(((double)(2*(double)dist)/(double)c)*1000000))-as ;

      tranche t = new tranche(time,time+as,"",0,0,-3) ;
      tranche t2 = null ;
      int tr = tf.freeInserable(t) ;
      if (tr != -1) t = tf.freeElementAt(tr) ;
      long retour = -1 ;
      if (tr != -1) {
         int tr2 = 0 ;
         if (imp) tr2 = extraire(time+as+ls,time+2*as+ls,as,-1) ;
         else tr2 = extraire(t.debut()+as+ls,as+ls+t.fin(),as,-1) ;
         if (tr2 != -   1){
            if (tr != tr2) {
               t2 = tf.freeElementAt(tr2) ;
               tf.freeRemoveElementAt(tr) ;
               if (tr2 > tr) tf.freeRemoveElementAt(tr2-1) ;
               else tf.freeRemoveElementAt(tr2) ;}
            else tf.freeRemoveElementAt(tr) ;

         // variables locales
            long a = 0 ;
            long b = 0 ;
            tranche n1 = null ;
            tranche n2 = null ;

         // on va chercher a commencer la tache au plus tot

            if (!imp) {
               if (tr == tr2){
               // une tranche de temps et debut non fixe.
                  a = t.debut();
                  b = a + as ;
                  n1 = new tranche(a,b,nom+ " DEBUT",lig,col,id) ;
                  a = b + ls ;
                  b = a + as ;
```

```
                n2 = new tranche(a,b,nom+ " FIN",lig,col,id) ;
                retour = b ;
                tf.scheAddElement(n1) ;
                tf.scheAddElement(n2) ;
                tf.oneFree(t,n1,n2) ;}
             else{
             // deux tranches de temps et debut non fixe.
                long k = t2.debut() -as -ls;
                a = 0 ;
                if (k > t.debut()) a = k ;
                else a = t.debut() ;
                b = a + as ;
                n1 = new tranche(a,b,nom+ " DEBUT",lig,col,id) ;
                a = b  + ls ;
                b = a + as ;
                n2 = new tranche(a,b,nom+ " FIN",lig,col,id) ;
                retour = b ;
                tf.scheAddElement(n1) ;
                tf.scheAddElement(n2) ;
                tf.twoFree(t,t2,n1,n2) ;}}
          else {
          // temps de depart fixe ;
             n1 = new tranche(time,time+as,nom
                              +" DEBUT",lig,col,id) ;
             n2 = new tranche(time+as+ls,time+2*as+ls,nom+
                              " FIN",lig,col,id) ;
             retour = time+2*as+ls ;
             tf.scheAddElement(n1) ;
             tf.scheAddElement(n2) ;
             if (tr == tr2)
             // une seule tranche
                tf.oneFree(t,n1,n2);
             else
             // deux tranches
                tf.twoFree(t,t2,n1,n2) ;}}}
      tf.evalue(tantpis,limite,as) ;
      if (tantpis.claque()){
         return (-1) ;}
      else
         return retour ;}

// nouvelle methode pour l'insertion des confirmtations

   private int logeable(long as, long tmin){
      tranche f = null ;
      int retour = -1 ;
      int index = 0 ;
      while ((index < tf.freeSize()) && (retour == -1)){
         f = tf.freeElementAt(index);
         if ((f.debut() <= tmin)&&(f.fin() >= tmin+as)) retour = index ;
         index++ ;}
      return retour ;}

   public long insereConf(String nom, int lig, int col,
                          long dist, long Tmin,
                          boolean imp, temperature tantpis,
                          long limite,long id){
      long as =
      (long)(deperdition * Math.pow((double) dist,(long)4) * cross) ;
      as++ ;
      long ls =
      ((long)(((double)(2*(double)dist)/(double)c)*1000000))-as ;

   // on se place dans le cas d'un depart tmin, retard possible.

      tranche A = null ;
      tranche B = null ;
      tranche Afinal = null ;
      tranche Bfinal = null ;
      int a = -1 ;
```

```
        int aprime = -1 ;
        int b = -1 ;
        while ((a = logeable(as,Tmin)) != -1){
           aprime = a;
        // on cherche B en iterant.
           A = tf.freeElementAt(a) ;
           long dep = Tmin + as +ls ;
           long fin = A.fin() - as + ls ;

           while ((dep <= fin)&&(B == null)) {
              tranche local =
              new tranche(dep, dep+2*as+ls,nom+" FIN",lig,col,id) ;
              tranche essai = null ;
              int index = 0 ;
              while ((index < tf.freeSize()) && (B == null)){
                 essai = tf.freeElementAt(index);
                 if (essai.inclus(local)) {
                    B = essai ;
                    b = index ;
                    Bfinal = local ;}
                 index++ ;}
              dep++ ;}
           Tmin = A.fin() ; }

        if ((A != null)&&(B != null)){
           Afinal =
           new tranche(Bfinal.debut() - as - ls,
               Bfinal.debut() - ls,nom+" DEBUT",
               lig,col,id) ;
           tf.freeRemoveElementAt(aprime) ;
           tf.scheAddElement(Afinal) ;
           tf.scheAddElement(Bfinal) ;
           if (A.debut() != B.debut()) {
              tf.freeRemoveElementAt(b) ;
              tf.twoFree(A,B,Afinal,Bfinal) ;}
           else tf.oneFree(A,Afinal,Bfinal) ;
           return Bfinal.fin() ;}
        else
           return -1 ;}

  // insertion des taches pulsees

     public boolean inserePulsedTask(long intersstask,
      long intersstaskdelta,
      long timeD, long timeF,
                      long as, long ls, String nom,
                      int lig, int col, long dist,
                      temperature tantpis,
                      long id,int pulse) {
        boolean retour = false ;
        if (entrelace) retour =
        (new insertPulsedTask(
              intersstask, intersstaskdelta,
              timeD, timeF,
              as, ls, tf, nom, lig,
              col, dist, tantpis, id,pulse)).go() ;
        else retour =
        (new insertPulsedTaskNONE(
         intersstask, intersstaskdelta,
         timeD, timeF,
               as, ls, tf, nom, lig, col,
               dist, tantpis, id,pulse)).go() ;
        return retour ;}

  // fonctions annexes
     public void makeHistory(Vector v) {
        tf.tri() ;
        int iFree = 0 ;
        int iSche = 0 ;
        tranche tFree = null ;
```

```
      if (tf.freeSize() != 0) tFree = tf.freeElementAt(iFree) ;
      tranche tSche = null ;
      if (tf.scheSize() != 0) {
         tSche = tf.scheElementAt(iSche) ;
         while ((iFree < tf.freeSize())||(iSche < tf.scheSize())){
            if ((iFree == tf.freeSize()))
               while ((iSche < tf.scheSize())){
                  v.addElement(tSche.toHist()) ;
                  iSche++ ;
                  if (iSche < tf.scheSize()) tSche =
                   tf.scheElementAt(iSche) ;}
            else
               if ((iSche == tf.scheSize()))
                  while ((iFree < tf.freeSize())){
                     v.addElement(tFree.toHist("TEMPS LIBRE")) ;
                     iFree++ ;
                     if (iFree < tf.freeSize()) tFree =
                      tf.freeElementAt(iFree) ;}
               else
                  if (tFree.debut() < tSche.debut()){
                     v.addElement(tFree.toHist("TEMPS LIBRE")) ;
                     iFree++ ;
                     if (iFree < tf.freeSize())
                     tFree = tf.freeElementAt(iFree) ;}
                  else{
                     v.addElement(tSche.toHist()) ;
                     iSche++ ;
                     if (iSche < tf.scheSize()) tSche =
                     tf.scheElementAt(iSche) ;}}}
      else{
      // il n'y a que des temps libres.
         while ((iFree < tf.freeSize())){
            v.addElement(tFree.toHist("TEMPS LIBRE")) ;
            iFree++ ;
            if (iFree < tf.freeSize()) tFree = tf.freeElementAt(iFree) ;}
      }}

   public void makeHistory2(long t2, Vector v) {
      tf.tri() ;
      tempusFugit tf2 = tf.over(t2) ;
      if (tf2.freeSize() > 0){
         tranche k = tf2.freeElementAt(0) ;
         if (k.debut() < t2){
            tf2.freeRemoveElementAt(0) ;
            tf.freeAddElement(new tranche(k.debut(),t2,"",-1,-1,-1)) ;
            tf2.freeAddElement(new tranche(t2,k.fin(),"",-1,-1,-1)) ;}}
      if (cptCol != -1){
         if (cptCol == 0) maBarre = new barre("1ere colonne",tf) ;
         else if (cptCol < updateCol){
            maBarre.update(tf) ;
            if (cptCol == (updateCol-1)) maBarre.run() ;}
         cptCol ++ ;
         if (cptCol == updateCol) cptCol = -1 ;}
      makeHistory(v) ;
      tf = tf2.backup() ;}

   public String toString() {
      return tf.toString() ;}

// ************************************************************************
   private class insertPulsedTask{
      private long epsilon = 0 ;
      private long epsilonPrime = 0 ;
      private int pulse = 0 ;
      private long courant = 0 ;
      private long Td = 0 ;
      private long Tf = 0 ;
      private long as = 0 ;
      private long ls = 0 ;
      private long B1 = 0 ;
```

```
private tempusFugit tf = null ;
private long delta = 0 ;
private long ep [] = null ;
private int epP = 0 ;
private String nom = null ;
private int lig = 0 ;
private int col = 0 ;
private long id = 0 ;
public insertPulsedTask(long intersstask,
     long intersstaskdelta, long timeD,
     long timeF,  long as, long ls,tempusFugit tf,
       String nom, int lig, int col, long dist,
       temperature tantpis, long id, int pulse) {
   epsilon = intersstask  ;
   epsilonPrime = intersstaskdelta ;
   Td = timeD ;
   Tf = timeF ;
   this.tf = tf ;
   this.pulse = pulse ;
   this.id = id ;
   this.nom = new String(nom) ;
   this.lig = lig ;
   this.col = col ;
   this.as = as ;
   this.ls = ls ;
   init() ;}

private void init(){
   courant = Td ;
   delta = 0 ;
   ep = new long[pulse+1] ;
   for (epP = 0 ; epP < (pulse+1) ; epP++) ep[epP] =
    epsilon - epsilonPrime ;
   epP = 0 ;}
private int phrase(long temps){
   if (temps >= as) {
      courant += as ;
      temps -= as ;
      int ret = suite(temps-as) ;
      if (ret == -1)
         return  1 ;
      else
         return ret ;}
   else
      return (-1) ;}

private int suite(long temps){
   if ( temps >= (ls+as)) {
      courant += ls + as ;
      temps -= ls+as ;
      if (epP == (pulse-1))
         return 4 ;
      else{
         epP++ ;
         int ret = phrase(temps - as - ls - epsilon + epsilonPrime) ;
         if (ret == -1){
            epP-- ;
            return 2 ;}
         else{
            courant += epsilon - epsilonPrime ;
            return ret ;}}}
   else
      return -1 ;}

private int suite2(long temps){
   if ( temps >= (as)) {
      courant += as ;
      temps-= as ;
      if (epP == (pulse-1))
         return 4 ;
```

```
      else{
         epP++ ;
         int ret = phrase(temps - as - epsilon + epsilonPrime) ;
         if (ret == -1){
            epP-- ;
            return 2 ;}
         else{
            courant += epsilon - epsilonPrime ;
            return ret ;}}}
   else
      return -1 ;}

private int gramBloc(boolean b){
// recherche d'un bloc contenant le temps "courant"
   long backup = courant ;
   int bloc = tf.enBloc(courant) ;
   if (bloc != -1) {
      tranche tr = tf.freeElementAt(bloc) ;
      int ret = 0 ;
      if (b) ret = phrase(tr.fin()-tr.debut()) ;
      else ret = suite2(tr.fin()-tr.debut()) ;
      if (ret != -1){
         long tmp = tr.fin() - courant ;
         if (tmp < delta) delta = tmp ;
         if (ret == 1) courant += ls ;
         else if (ret == 2) {
            courant += epsilon + epsilonPrime ;
            ep[epP] = epsilon + epsilonPrime ;
            epP++ ;}}

      return ret ;}
   else
      return bloc ;}

private void inclureFree(tranche t){
   int i = 0 ;
   int trouve  = -1 ;
   tranche contenant = null ;
   while ((i < tf.freeSize())&&(trouve == -1))
      if (!(contenant = tf.freeElementAt(i)).inclus(t)) i++ ;
      else trouve = i ;
   if (trouve != -1){
      tf.freeRemoveElementAt(i) ;
      tf.upfree(contenant,t) ;
      tf.scheAddElement(t) ;}}

public boolean go(){
   int i = 0 ;
   int local = 2 ;
   tf.tri() ;
   tranche t = null ;
   long Dmade = 0 ;
   while (local < 3){
      init() ;

      while((i < tf.freeSize())&& (t == null))
      {t = tf.freeElementAt(i) ;
         if (!(t.enBloc(Td))) {
            if (t.debut() < Td){
               Td=t.debut();}
            else{
               i++;
               t = null ;}}}

      if (t != null){
         delta = t.fin() - t.debut() ;
         Dmade = 0 ;
         int r = 0 ;
         while (local < 3) {
            r = local ;
```

```
                if (local == 1) local = gramBloc(false) ;
                else local = gramBloc(true) ;
                if (local == -1)
                   if (delta == 0) local = 5 ;
                   else {delta-- ;
                      Dmade++ ;
                      courant-- ;
                      local = r ;}
                if (local == 5)
                   if (i != (tf.freeSize()-1)) {
                      i++ ;
                      t = tf.freeElementAt(i) ;
                      Td = t.debut() ;}}}
          else local = 5 ;
            }

      // conclusion. soit tout est insere soit ce n'est pas inserable
      // si c'est tout insere dans les limites de temps alors
      // on effectue l'insertion physique a l'aide
      // de td et du tableau ep, on renvoie true
         tranche nouveau = null ;
         int epP2 = 0 ;
         if ((local == 4)&&(courant < Tf))
         {
            Td += Dmade ;
            epP2 = 0 ;
            for (int j = 0 ; j < (2*pulse); j++){
               if ((j % 2) != 0) {
                  nouveau = new tranche(Td,Td+as,nom+" FIN",lig,col,id) ;
                  inclureFree(nouveau) ;
                  Td += as + ep[epP2] ;
                  epP2 ++ ;}
               else {
                  nouveau = new tranche(Td,Td+as,nom+" DEBUT",lig,col,id) ;
                  inclureFree(nouveau) ;
                  Td += as+ls ;}}}
         return ((local == 4) && (courant < Tf)) ;}}

   private class insertPulsedTaskNONE{

      private long epsilon = 0 ;
      private long epsilonPrime = 0 ;
      private int pulse = 0 ;
      private long courant = 0 ;
      private long Td = 0 ;
      private long Tf = 0 ;
      private long as = 0 ;
      private long ls = 0 ;
      private long B1 = 0 ;
      private tempusFugit tf = null ;
      private long delta = 0 ;
      private long ep [] = null ;
      private int epP = 0 ;
      private String nom = null ;
      private int lig = 0 ;
      private int col = 0 ;
      private long id = 0 ;

      public insertPulsedTaskNONE(long intersstask,
           long intersstaskdelta, long timeD,
            long timeF,
                     long as, long ls,tempusFugit tf,
                     String nom, int lig, int col,
                     long dist, temperature tantpis,
                     long id, int pulse) {
         epsilon = intersstask  ;
         epsilonPrime = intersstaskdelta ;
         Td = timeD ;
         Tf = timeF ;
         this.tf = tf ;
```

```
   this.pulse = pulse ;
   this.id = id ;
   this.nom = new String(nom) ;
   this.lig = lig ;
   this.col = col ;
   this.as = as ;
   this.ls = ls ;
   init() ;}
private void init(){
   courant = Td ;
   delta = 0 ;
   ep = new long[pulse+1] ;
   for (epP = 0 ; epP < (pulse+1) ; epP++) ep[epP] =
    epsilon - epsilonPrime ; ;
   epP = 0 ;}
private void inclureFree(tranche t){
   int i = 0 ;
   int trouve  = -1 ;
   tranche contenant = null ;
   while ((i < tf.freeSize())&&(trouve == -1))
      if (!(contenant = tf.freeElementAt(i)).inclus(t)) i++ ;
      else trouve = i ;
   if (trouve != -1){
      tf.freeRemoveElementAt(i) ;
      tf.upfree(contenant,t) ;
      tf.scheAddElement(t) ;}}
public boolean go(){
   tf.tri() ;
   long longueurTask = as+ls+as ;
   long total = ((longueurTask+epsilon)* pulse-1) + longueurTask ;
 // tester les blocs libres :
 // si le bloc est assez grand pour la tache,
 //    alors regarder si ca correspond aux
 //    creneaux de temps
 // si oui OK sinon on passe au bloc suivant.
 //    s'il n'y en a plus ECHEC
   boolean trouve = false ;
   int i = 0 ;
   while ((i < tf.freeSize())&&(!trouve)){
      tranche t = tf.freeElementAt(i) ;
      long a ;
      long b ;
      if (t.debut() > Td) a = t.debut() ;
      else a = Td ;
      if (t.fin() < Tf) b = t.fin() ;
      else b = Tf ;
      if ((b - a) <= 0) i++ ;
      else if ((b-a) >= total){
      // OK on a trouve qqchose de valide, alors on place
      // et l'origine du temps c'est a
         trouve = true ;
         for (int j = 0 ; j < pulse ; j++){
            tranche ins = new tranche(a,a+as,nom+" DEBUT",lig,col,id) ;
            inclureFree(ins) ;
            a += as + ls ;
            ins = new tranche(a,a+as,nom+" FIN",lig,col,id) ;
            inclureFree(ins) ;
            a += as+epsilon ;}}
      else i++ ;
   }
   return trouve ;}}}
```

C.1.9 simulation.java

Module principal du programme.

```
import java.util.Vector ;
 import java.util.Random ;
 import java.io.* ;
 import java.awt.*;
 import java.awt.event.*;
 public class simulation {

    static long longueurTrain = 12 ;

    static long attenteMin ; // en micro-s
    static long dureeMax ;
    static long intersstask ;
    static long intersstaskdelta ;
    static long distMin ; // en metres
    static long distMax ; // en metres

    static long c ; //en m/s
    static double deperdition ;
    static double cross ;
    static int probeRefresh ;

    static int lig ;
    static int col ;
    static long tour ;// temps de rotation du radar
    static long fin ;// fin de la simulation
    static long angle ; // angle de visibilite du radar

    static boolean jaffiche ; // genere un journal sur la sortie standard
    static boolean killer ; // vrai si on tue le front.
    static boolean entrelace ; // vrai si on entrelace les taches.
    static int snapshotVal ;

    static double p ;
    static double q ;
    static double confirm ;
    static double confirmRep ;
    static int maxItems ;

    static double MaxTemps ; // valeur max de la temperature ;

    static long asProbe  ;
    static long bsProbe  ;
    static long lsProbe  ;
    static long ts       ; // tps d'une sous-tache de probe

    static Vector histoire  = new Vector() ;
    static Random randy = new Random() ;
    static Stat statistik = null ;
    static temperature limite = null ;
    static grilleTask radar = null ;
    static barre3 front = new barre3("SIMULATION DE RADAR 1.0") ;

    static long idJob = 0 ;

    public static long dist(double alea){
       return (long)(alea*(distMax-distMin) + distMin) ;}

    public static long terre(long a, long b){
       long  c = a % b ;
       return (long)((a-c)/b) ;}

    public static boolean jouable(int cc, int cg, int cd) {
       if (cd >= cg)
          return ((cc >= cg)&& (cc <=cd)) ;
```

```
      else
         return ((cc <= cd)||(cc >= cg)) ;}

   public static liste trier(liste ajouer) {
      liste ajouer2  = new liste() ;
      long min = 0 ;  long min2 = 0 ;
      int ind = 0 ;
      int act = 0 ;
      int i = 0 ;
      int j = 0;
      while ((i = ajouer.getSize()) != 0) {
         j = 0 ;
         min = ((tacheConf)ajouer.getTache(0)).getDebut();
         for (int k = 1 ; k < i ; k++)
            if ((min2 = ((tacheConf)ajouer.getTache(k)).getDebut()) < min)
            { j = k ;
               min = min2 ;}
         ajouer2.setTache(ajouer.getTache(j)) ;
         ajouer.removeTache(j) ;}
      return ajouer2 ;}

   public static int jouerTache(liste maListe, int cg, int cd,
                   String name,boolean probing,
                   long fenetre, sche2 s, long temps,
                   liste pistage){
   // y a t-il des taches a lancer dans le pinceau ?

   // variables globales

      int loupe = 0 ;
      tache courante = null ;
      simulation.liste ajouer = new liste() ;
      int cc = cd ;
      while (jouable(cc,cg,cd)){
         maListe.creneau(cc,ajouer) ;
         cc-- ;
         if (cc < 0) cc = col - 1 ;}

      int temp = 0 ;
      if (ajouer != null)
         while (temp < ajouer.getSize()){
            courante = (tacheConf)ajouer.getTache(temp) ;
            if (fenetre < ((tacheConf)courante).getDebut()) {
               maListe.setTache(courante)  ;
               ajouer.removeTache(temp) ;}
            else temp++ ;}

      ajouer = trier(ajouer) ;
      temp = 0 ;
      Vector h2 = new Vector() ;
      long last[] = new long [ajouer.getSize()] ;
      for (int i = 0; i < ajouer.getSize() ; i++)
         last[i] = ((tacheConf)ajouer.getTache(temp)).getDebut() ;

      while (ajouer.getSize() != 0) {
         while (temp < ajouer.getSize()) {
            courante = (tacheConf)ajouer.getTache(temp) ;
            long t1 = ((tacheConf)courante).getDistance() ;
            long t2 = (long)(deperdition *
                 Math.pow(t1,(long)4) * cross) ;
            long t3 = ((long)(((double)(2*(double)t1)/
                      (double)c)*1000000)-t2) ;
            temperature myT = new temperature(limite) ;
            last[temp] = s.insere(name +
 Integer.toString(((tacheConf)courante).getT()),
((tacheConf)courante).getLig(), courante.getCol(),t1,
last[temp],true,myT,fenetre,
((tacheConf)courante).getID()) ;

            if (last[temp]==-1) {
```

```
            loupe++;
            ajouer.removeTache(temp) ;}
         else
         {limite.setTemperature(myT.getTemperature()) ;
            ((tacheConf)courante).itere() ;  }
         if (((tacheConf)courante).over()) {
            if (!probing) statistik.pistageEffectue() ;
            ajouer.removeTache(temp) ;
            if (courante.play(randy.nextDouble(),0) != null){
               ((tacheConf)courante).respawn(tour) ;
               pistage.setTache(courante) ;
               if (probing) statistik.confirmed(); }
            else {
               radar.dec(((tacheConf)courante).getLig(),
               ((tacheConf)courante).getCol()) ;
               if (!probing) statistik.ciblesTuees() ;}
            for (int i = temp ; i < (ajouer.getSize()-1) ; i++)
             last[i] = last[i+1] ;}
         else temp++ ;}
      temp = 0 ;}
   return loupe ;}

public static int jouerTachePursuit(liste maListe, int cg,
 int cd, String name,
                  boolean probing,long fenetre,
                  sche2 s, long temps, liste pistage,
                  liste confirmation, int nb){


// y a t-il des taches a lancer dans le pinceau ?

// variables globales

   int loupe = 0 ;
   tacheConf courante = null ;
   liste ajouer = new liste() ;
   int cc = cd ;
   while (jouable(cc,cg,cd)){
      maListe.creneau(cc,ajouer) ;
      cc-- ;
      if (cc < 0) cc = col - 1 ;}

   int temp = 0 ;
   if (ajouer != null)
      while (temp < ajouer.getSize()){
         courante = (tacheConf)(ajouer.getTache(temp)) ;
         if (fenetre < courante.getDebut()) {
            maListe.setTache(courante)  ;
            ajouer.removeTache(temp) ;}
         else temp++ ;}

   ajouer = trier(ajouer) ;
   temp = 0 ;

   long last = 0 ;
   while  (ajouer.getSize() != 0){
      courante = (tacheConf)(ajouer.getTache(0)) ;
      ajouer.removeTache(0) ;
      long t1 = (courante).getDistance() ;
      long t2 = (long)(deperdition *
        Math.pow(t1,(long)4) * cross) ;
      t2++ ;
      long t3 = ((long)(((double)(2*(double)t1)/
       (double)c)*1000000)-t2) ;
      temperature myT = new temperature(limite) ;
      last = courante.getDebut() ;
      long tempsMax = last  +  dureeMax ;


      boolean l =
```

```
            s.inserePulsedTask(intersstask,
intersstaskdelta, last, tempsMax,
t2, t3, name, courante.getLig(),
courante.getCol(), t1,
                    limite, courante.getID(),nb) ;
          if (!1) {
             front.afficheLoupes(courante.toString() + " " +
                  last+ " "+t2 +" " +t3+" loupee") ;
             loupe++ ;
             radar.dec((courante).getLig(),(courante).getCol()) ;}
          else {
             statistik.pistageEffectue() ;
             if (courante.play(randy.nextDouble(),0) != null){
                (courante).respawn(tour+ (2*t2+t3)*nb) ;
                pistage.setTache(courante) ;}
             else
             {(courante).respawnRep(attenteMin + (2*t2+t3)*nb) ;
                confirmation.setTache(courante) ;
                statistik.ciblesTuees() ;}}}
       return loupe ;}


    public static int jouerTacheConfirm(liste maListe, int cg,
     int cd, String name,
                    boolean probing,long fenetre,
                    sche2 s, long temps,
                    liste pistage, int nb){
    // y a t-il des taches a lancer dans le pinceau ?
       int loupe = 0 ;
       tacheConf courante = null ;
       liste ajouer = new liste() ;
       int cc = cd ;
       while (jouable(cc,cg,cd)){
          maListe.creneau(cc,ajouer) ;
          cc-- ;
          if (cc < 0) cc = col - 1 ;}

       int temp = 0 ;
       if (ajouer != null)
          while (temp < ajouer.getSize()){
             courante = (tacheConf)(ajouer.getTache(temp)) ;
             if (fenetre < courante.getDebut()) {
                maListe.setTache(courante)  ;
                ajouer.removeTache(temp) ;}
             else temp++ ;}

       ajouer = trier(ajouer) ;
       temp = 0 ;

       long last = 0 ;
       while  (ajouer.getSize() != 0){
          courante = (tacheConf)(ajouer.getTache(0)) ;
          ajouer.removeTache(0) ;
          long t1 = (courante).getDistance() ;
          long t2 = (long)(deperdition * Math.pow(t1,(long)4) * cross) ;
          t2++ ;
          long t3 = ((long)(((double)(2*(double)t1)/(double)c)*1000000)-t2) ;
          temperature myT = new temperature(limite) ;
          last = courante.getDebut() ;
          long tempsMax = last  +  dureeMax ;

          int pulse = 0 ;
          boolean 1 =
           s.inserePulsedTask(intersstask,
intersstaskdelta, last, tempsMax,
t2, t3, name, courante.getLig(),
courante.getCol(), t1,
limite, courante.getID(),nb) ;
    if (!1) {
             front.afficheLoupes(courante.toString()+" " +
```

```
                last+ " "+t2 +" " +t3+" loupee") ;

            loupe++ ;
            radar.dec((courante).getLig(),(courante).getCol()) ;}
         else {
            if (courante.play(randy.nextDouble(),0) != null){
               (courante).respawn(tour+ (2*t2+t3)*nb) ;
               pistage.setTache(courante) ;
               statistik.confirmed(); }
            else
               radar.dec((courante).getLig(),(courante).getCol()) ;}}
      return loupe ;}


   public static void main(String argv[]){
      liste confirmation = new liste() ;
      liste pistage = new liste() ;
      FileReader fl = null ;
      try{
         fl = new FileReader("param.txt") ;}
         catch(Exception e){
            System.err.println(
            "le fichier de paramètres, param.txt, n'est"
            "pas localisable dans le répertoire courant!!!") ;
            System.exit(-1);}
      try{
         longueurTrain = fl.getLong("longueurTrain") ;
         attenteMin= fl.getLong("attenteMin");
         dureeMax= fl.getLong("dureeMax");
         intersstask = fl.getLong("intersstask");
         intersstaskdelta = fl.getLong("intersstaskdelta");
         distMin = fl.getLong("distMin");
         distMax = fl.getLong("distMax");
         c = fl.getLong("c");
         deperdition = fl.getDouble("deperdition") ;
         cross = fl.getDouble("cross") ;
         probeRefresh = fl.getInt("probeRefresh") ;
         killer = fl.getBoolean("killer") ;
         lig = fl.getInt("lig") ;
         col = fl.getInt("col") ;
         tour = fl.getLong("tour") ;
         fin = fl.getLong("fin") * tour ;
         angle = fl.getLong("angle") ;
         jaffiche = fl.getBoolean("jaffiche") ;
         snapshotVal = fl.getInt("snapshotVal") * col ;
         p = fl.getDouble("p") ;
         q = fl.getDouble("q") ;
         confirmRep = fl.getDouble("confirmRep") ;
         confirm = fl.getDouble("confirm") ;
         maxItems = fl.getInt("maxItems")  ;
         MaxTemps = fl.getDouble("MaxTemps") ;
         entrelace = fl.getBoolean("entrelace") ;
         front.afficheLoupes("Il reste "+fl.reste()+
                   " éléments non exploités !") ;

         asProbe = (long)(deperdition *
          Math.pow(distMin+distMax,(long)4) * cross) ;
         bsProbe = ((long)(((double)(2*(double)distMax)/
          (double)c)*1000000)) ;
         lsProbe = (long) ((((2*(double)distMin)/(double)c)
          *1000000))-asProbe ;
         ts = (asProbe + bsProbe + lsProbe) ;

         statistik =
          new Stat(System.currentTimeMillis(),
          asProbe,bsProbe,lsProbe) ;
         limite = new temperature(MaxTemps) ;
         radar = new grilleTask(lig,col,maxItems,p) ;}

         catch(Exception e){
```

```
      front.afficheLoupes(
       "Une erreur a ete rencontree : donc stop\r\n" + e) ;
      System.exit(-1) ;}

long temps = 0 ; // temps du systeme en micro-secondes.
long fenetre = 0 ; // temps de fin de visibilite de la colonne
int colonne = 0 ; // nom de la colonne la plus a droite.
int colonneFin = 0; // nom de la derniere colonne
int compteur = 0 ; // compte le nbre de probe effectues.
long base = 0 ; // temps de base.
tache courante = null ;
long tempsAff = 0 ;
front.run() ;
front.afficheLoupes("Debut de la simulation ") ;

liste ajouer = null ;
int cg = 0 ;
int cd = 0 ;
int cc = 0 ;
int cinit = 0 ;
long delta = 0 ;
int temp = 0 ;
boolean init = true ;
boolean init2 = true ;
int nbtour = 0 ;
sche2 s = new sche2(0,fin,deperdition,cross,c,asProbe,entrelace) ;
do {
   delta = (long)(tour / col) ;
   fenetre = temps + delta ;

// quels sont les numeros des colonnes qui nous preoccupent ?

   cd = (int)(((double)(temps%tour)/tour)*col)- 1 ;
   if (cd == -1) cd = col - 1 ;
   cg = cd - ((int)((2*angle*col) / 360))  ;
   if (cg < 0) cg = col + cg  ;
   if (init2) {
      cinit = cd ;
      init2 = false ;}
// y a t-il des taches de confirmation a lancer dans le pinceau ?
   int ret = jouerTacheConfirm(confirmation, cg, cd, "Confirmation ",
             true, fenetre,s,temps,pistage,1) ;
   statistik.loupeConfirm(ret) ;

// y a t-il des taches de confirmation a lancer dans le pinceau ?
   ret = jouerTachePursuit(pistage, cg, cd, "_Pistage ", false,
                     fenetre,s,temps,pistage,
                     confirmation,(int)longueurTrain) ;
   statistik.loupePistage(ret) ;

// taches de probe.
   cc = cd ;
   int j = 0 ;
   int place = -1 ;

   while (((place = s.inserableProbe(ts,temps+delta))!= -1) &&
         jouable(cc,cg,cd)){
      while (((place = s.inserableProbe(ts,temps+delta))!= -1) &&
            (radar.ligAvailable(cc))){
         j = radar.getFirstFree(cc) ;
         temperature myT = new temperature(limite) ;
         long delta2 = s.insererProbe(place,ts,j,cc,myT,fenetre,idJob);
         idJob++ ;
         if (delta2 != -1){
            statistik.probe() ;
            radar.setAJouer(j,cc) ;
            limite.setTemperature(myT.getTemperature()) ;
            if (radar.joue(j,cc,randy.nextDouble()))
            {confirmation.setTache((new tacheConf(
                     new coordsTime(
```

```
                                new coords(j,cc,dist(randy.nextDouble())),
                                new time(delta2+attenteMin+ts)),
                                q,confirm,confirmRep,idJob))) ;
                    idJob++ ;
                    radar.inc(j,cc) ;
                    statistik.alertes();}}}

            cc-- ;
            j = 0 ;
            if (cc < 0) cc = col - 1 ;}

         temp = radar.reset(cg) ;
         if (init)
            if (cg != cinit) temp = 0 ;
            else init = false ;
         if (temp != 0){
         front.afficheLoupes(temp+
         " Tache(s) de surveillance loupee(s) colonne :"+cg) ;}
         statistik.loupeProbe(temp) ;
         s.makeHistory2(temps + (long)(tour / col),histoire) ;
      // mise a jour de la temperature.

         history t = null ;
         double maxT = 0 ;
         int local = 0;

         while(local < histoire.size()){
            t = (history)(histoire.elementAt(local)) ;
            if (t.getLig() == -1) limite.relaxation(t.duree()) ;

            if (((t.nom()).substring(0,1)).equals("C"))
               if (((t.nom()).indexOf("DEBUT")) != -1)
                   limite.tension(t.duree());
               else limite.relaxation(t.duree()) ;

            if (((t.nom()).substring(0,1)).equals("S"))
               if (((t.nom()).indexOf("DEBUT")) != -1)
                  limite.tension(t.duree());
               else limite.relaxation(t.duree()) ;

            if (((t.nom()).substring(0,1)).equals("P")){
               limite.tension(asProbe);
               limite.relaxation(lsProbe+bsProbe) ;}
            if (maxT < limite.getTemperature())
               maxT = limite.getTemperature() ;
            local ++ ;}


         statistik.update(histoire,jaffiche) ;
         temps += (long)(tour / col) ;

      // eliminer les taches dont la date de validite est depassee.
         temp = 0 ;
         while (temp < confirmation.getSize()){
            courante = confirmation.getTache(temp) ;
            if (temps > ((tacheConf)courante).getDebut()) {
               histoire.addElement(
                    new history("remove -- conf", temps, 0,
                            ((tacheConf)courante).getCol(),
                            ((tacheConf)courante).getLig())) ;
               confirmation.removeTache(temp) ;
               radar.dec(((tacheConf)courante).getLig(),
                        ((tacheConf)courante).getCol()) ;
               statistik.loupeConfirm(1);}
            else temp++ ;}

         temp = 0 ;
         while (temp < pistage.getSize()){
            courante = pistage.getTache(temp) ;
            if (temps > ((tacheConf)courante).getDebut()) {
```

```
            histoire.addElement(
                    new history("remove -- pistage", temps, 0,
                        ((tacheConf)courante).getCol(),
                        ((tacheConf)courante).getLig())) ;
            radar.dec(((tacheConf)courante).getLig(),
               ((tacheConf)courante).getCol()) ;
            pistage.removeTache(temp) ;
            statistik.loupePistage(1) ;}
         else temp++ ;}
      if ((temps-tempsAff) > (tour/10)) {
         front.affichePercent(
         statistik.lasts(((((float)temps/(float)fin)*100.0)),
                        System.currentTimeMillis())) ;
         tempsAff += (tour/10) ;}
      nbtour++ ;
      if ((nbtour % snapshotVal) == 0)
         statistik.snapShoter((long)(nbtour/col),pistage.getSize()) ;
   } while(temps<fin);

   s.makeHistory2(fin,histoire) ;
   statistik.update(histoire,jaffiche) ;
   front.afficheLoupes("FIN DE LA SIMULATION    ") ;
   front.affichePercent("LA SIMULATION \nEST TERMINEE") ;
   front.afficheLoupes(statistik.snapshot((long)(nbtour/col),
                  pistage.getSize())) ;
   String lafin = statistik.toString() ;
   front.afficheLoupes(lafin) ;
   statistik.close(lafin) ;
   if (killer)front.cestFini() ;}


private static class Stat{
   private class fili{

      private File sin = null ;
      private FileOutputStream in = null ;
      public fili(String s){
         sin = new File(s) ;
         try{in  = new FileOutputStream(sin);}
            catch(Exception e) {
               System.err.println("PAS GRUMPF !") ;}}
      public void write(String tampon){
         int i = 0 ;
         try{
            while (i < tampon.length()) {
               in.write(tampon.charAt(i)) ;
               i++ ;}}
            catch (Exception e) {
               System.err.println("PAS GRUMPF !") ;}}
      public void close(){
         try{in.close() ;}
            catch (Exception e) {
               System.err.println("PAS GRUMPF !") ;}}}


   private long vide = 0 ;
   private long confirming = 0 ;
   private long probing = 0 ;
   private long pisting = 0 ;
   private long actif = 0 ;
   private long debutReel = 0 ;
   private long loupePistage = 0 ;
   private long loupeProbe = 0 ;
   private long loupeConfirm = 0 ;
   private long alerte = 0 ;
   private long confirmed = 0 ;
   private long madeProbe = 0 ;
   private long killedTargets = 0 ;
   private long nbpisteff = 0 ;
```

```
long machin = 0 ;
long m2 = 0 ;

private long as = 0 ;
private long bs = 0 ;
private long vs = 0 ;
private String ret = "\r\n" ;

private fili sauveSnapshot = null ;

public Stat (long debutReel,long as, long bs, long vs) {
   this.debutReel = debutReel ;
   this.as = as ;
   this.bs = bs ;
   this.vs = vs ;
   sauveSnapshot = new fili("snapshot") ;}

public String lasts(double pourcent, long tcourant){
   double temps = tcourant - debutReel ;
   if (pourcent > 1.0){
      temps = (temps * (100.0 - pourcent)) / pourcent ;
      int chrono = (int)(temps / 1000) ;
      int heure = (int) (chrono / 3600) ;
      chrono = chrono - 3600 * heure ;
      int minute = (int)(chrono / 60) ;
      int seconde = chrono - 60 * minute ;
      String retour = "Ecoulé : "+ ((int)(pourcent)) + "
               % du calcul \nTemps restant estime : "
               +heure+"H"+minute+"M"+seconde+"S\n";
      temps = tcourant - debutReel ;
      temps = (temps * (100.0)) / pourcent ;
      chrono = (int)(temps / 1000) ;
      heure = (int) (chrono / 3600) ;
      chrono = chrono - 3600 * heure ;
      minute = (int)(chrono / 60) ;
      seconde = chrono - 60 * minute ;
      return (retour+"Temps total estime     : "+
            heure+"H"+minute+"M"+seconde+"S");
   }
   else
      return ("Ecoulé : "+ pourcent + " % du calcul \nTemps restant non estimable.");}

public void update(Vector histoire, boolean affiche){
   history t = null ;
   while(histoire.size() != 0){
      t = (history)(histoire.elementAt(0)) ;
      machin +=  t.duree() ;
      if (t.getLig() == -1) vide += t.duree() ;
      if (((t.nom()).substring(0,1)).equals("C"))
           confirming += t.duree();
      if (((t.nom()).substring(0,1)).equals("P"))
           m2 += t.duree();
      if (((t.nom()).substring(0,1)).equals("_"))
           pisting += t.duree();
      if (((t.nom()).indexOf("DEBUT")) != -1)
           actif += t.duree() ;
      if (affiche) System.out.print(t) ;
      histoire.removeElementAt(0) ;}}

public void loupeProbe(int too) {
   loupeProbe += too ;}
public void loupePistage(int too) {
   loupePistage += too;}
public void loupeConfirm(int too) {
   loupeConfirm += too ;}
public void alertes(){
   alerte++;}
public void confirmed(){
   confirmed++;}
```

```
public void probe(){
   madeProbe++;}

public void ciblesTuees(){
   killedTargets++ ;}

public void pistageEffectue(){
   nbpisteff++ ;}

private String formate(String texte, long a, long b,String s){

   double temp = (((double)a/(double)b)*10000.0) ;
   int temp2   = (int) temp ;
   return (texte + ((float)(temp2)/100) + " %"+ s + ret) ;}


public String toString(){
   String retour = new String() ;
   long finReel = System.currentTimeMillis() ;

   probing = (as+bs) * madeProbe;
   long probingVide = (vs) * (madeProbe ) ;
   actif += as * (madeProbe);
   vide += vs * (madeProbe) ;

   retour +=
   "DEBUT      : " +debutReel+ ret ;
   retour +=
   "FIN        : " + finReel +ret ;
   retour +=
   "DIFFERENCE : " + (finReel - debutReel)+ret ;
   retour += ret ;
   retour +=
   "Surveillances        : " + madeProbe + "/" +
   (double) (m2/(as+bs+vs)) + ret ;
   retour += "Alertes confirmees   : " + confirmed+ret ;
   retour += "Alertes              : " + alerte+ret ;
   retour += ret ;
   retour += formate("LOUPE SURVEILLANCE  : ",
   loupeProbe, madeProbe+loupeProbe," ("+
   Long.toString(loupeProbe)+") des taches") ;
   retour += formate("LOUPE PISTAGE       : ",
   loupePistage, confirmed," ("+Long.toString(loupePistage)
   +") des taches") ;
   retour += formate("LOUPE CONFIRMATION  : ",
   loupeConfirm, alerte," ("+Long.toString(loupeConfirm)
   +") des taches") ;
   retour += "CIBLES TUEES        : " + killedTargets + ret ;

   retour += ret ;
   retour += formate("TEMPS MORT   : ",vide,fin," du temps") ;
   retour += formate("SURV. PUR    : ",probing,fin," du temps") ;
   retour += formate("PISTAGE      : ",pisting,fin," du temps") ;
   retour += formate("CONFIRMATION : ",confirming,fin," du temps") ;
   retour += formate("EMISSION     : ",actif,fin," du temps") ;
   retour += formate("SURV. MORT   : ",probingVide,fin," du temps") ;
   retour += "TOTAL        : " +
   (vide+probing+confirming+pisting) +
   "/"+fin +"/"+machin+ret ;
   return retour ;}

public String snapshot (long tour, long nbpiste){
   return  ("NUMERO DE L'ITERATION : " + tour + ret +
           "Nombre de pistages en cours          : " +
            nbpiste + ret +
           "Nombre de pistages effectues (cumul) : " +
           nbpisteff + ret +
           "Nombre d'alertes (cumul)             : " +
            alerte + ret +
           "Nombre de cibles tuees (cumul)       : " +
```

```
                killedTargets + ret +
                "Nombre d'alertes confirmees (cumul)  : " +
                confirmed + ret +
                "Nombre de pistages loupes (cumul)    : " +
                loupePistage) ;}
    public void snapShoter(long tour, long nbpiste){
       sauveSnapshot.write(snapshot(tour,nbpiste) + ret) ;}

    public void close(String fin){
       sauveSnapshot.write(fin + ret) ;
       sauveSnapshot.close() ;}}

 private static class liste{
    private Vector maListe = null ;
    private int maxLength = 0 ;
    public liste(int taille){
       init(taille);}
    public liste(){
       init(256*5);}
    public void init(int t){
       maListe = new Vector() ;
       maxLength = t ;}
    public int getSize(){
       return maListe.size();}
    public int getMaxSize(){
       return maxLength ;}
    public void setTache(tache c){
       if (getSize() < getMaxSize())
          maListe.addElement(c) ;}
    public tache getTache(int num){
       tache retour = null ;
       if ((num >= 0) && (num < getSize()))
          retour = (tache)maListe.elementAt(num) ;
       return retour ;}
    public void removeTache(int num){
       if ((num >= 0) && (num < getSize()))
          maListe.removeElementAt(num) ;}
    public void creneau(int col, liste retour){
       tache c2 = null ;
       int i = 0 ;
       while (i < getSize()){
          if ((c2 = getTache(i)).getCol()==col) {
             retour.setTache(c2) ;
             removeTache(i) ;}
          else i++ ;}}}
 private static class barre3 extends Frame implements ActionListener {
    private String titre = null ;
    private final String QUITTER      = "Quitter";
    private TextArea LOUPES   = new TextArea("",50,80,3) ;
    private TextArea POURCENT = new TextArea("",3,80,3) ;
    public barre3 (String titre) {
       this.titre = titre ;
       Panel RPanel = new Panel();
       setLayout(new BorderLayout());

       MenuBar mb = new MenuBar();
       Menu m = new Menu("Fichier");
       MenuItem fileMenuItem = new MenuItem(QUITTER);
       fileMenuItem.addActionListener(this);
       m.add(fileMenuItem);
       mb.add(m);
       setMenuBar(mb);

    //Création du menu de contrôle du programme
       Panel LPanel2 = new Panel() ;
       LPanel2.setLayout(new GridLayout(1,2));
       POURCENT.setEditable(false) ;
       LPanel2.add(POURCENT);
       add("North",LPanel2) ;
```

```
        LPanel2 = new Panel() ;
        LPanel2.setLayout(new GridLayout(1,2));
        LOUPES.setEditable(false) ;
        LPanel2.add(LOUPES);
        add("South",LPanel2) ;

        addWindowListener(
                            new WindowAdapter() {
                               public void windowClosing(WindowEvent e) {
                                  System.exit(0);}});}
     public void actionPerformed(ActionEvent event) {
        final String Tache =
        new String(((MenuItem)event.getSource()).getLabel()) ;
        if (Tache.equals(QUITTER)) System.exit(0) ;
        else System.err.println("|" + Tache + "|") ;}
     private String formate(String s){
        String retour = s+"\r\n" ;
        return retour ;}
     public void affichePercent(String s) {
        POURCENT.setText(s) ;}

     private int cpt = 0 ;
     public void afficheLoupes(String s) {
        cpt++ ;
        if (cpt == 51){
           String texte = LOUPES.getText() ;
           texte = texte.substring(texte.indexOf("\r\n")+2) + formate(s) ;
           LOUPES.setText(texte) ;
           cpt = 50 ;}
        else
           LOUPES.append(formate(s)) ;}
     public void cestFini(){
        System.exit(0) ;}
     public void run(){
        barre3 window = this ;
        window.setTitle(titre);
        window.pack();
        window.setVisible(true); }}}
```

C.1.10 tache.java

Définition d'une interface pour l'objet Tâche. Cette définition aura plusieurs implémentations (Tâche de confirmation, tâche de pistage,...).

```
public interface tache {
   public void setPriorite(int i) ;
   public int getPriorite() ;
   public void setFrequence(int i) ;
   public int getFrequence() ;
   public coordsTime play(double toto, int tim) ;
   public int getCol() ;
}
```

C.1.11 tacheConf.java

Implémentation de l'interface *tache* pour le cas d'une tâche de confirmation ou de pistage.

```
public class tacheConf extends task implements tache {

  private coordsTime argh = null ;
  private int train = 1 ;
  private boolean pursuit = false ;
  private boolean repeated = false ;
  private int oldness = 0 ;
  private double mort = 0.0 ;
  private double confirm = 0.0 ;
  private double confirmRep = 0.0 ;
  private long Id = 0 ;
  public long getDebut() {
     return argh.getTime().getTime();}
  public long getDistance() {
     return argh.getCoords().getDist() ;}
  public int getLig() {
     return argh.getCoords().getLig() ;}
  public int getCol() {
     return argh.getCoords().getCol() ;}
  public void itere() {
     train++;}
  public int getT() {
     return train;}
  public boolean over() {
     return (getT() == 13 );}

  public long getID(){
     return Id ;}

  public void respawn(long t) {
     argh.getTime().setTime(t+argh.getTime().getTime()) ;
     train = 1;
     setPursuit() ;
     repeated = false ;
     oldness++ ;}

  public void respawnRep(long t) {
     argh.getTime().setTime(t+argh.getTime().getTime()) ;
     train = 1;
     setPursuit() ;
     repeated = true ;
     oldness++ ;}

  public String toString() {
     if (pursuit)
        if (repeated)
           return ("CONFIRMATION DE POURSUITE : " +
           getLig() + " " + getCol() + " ("+oldness+")") ;
        else
           return ("POURSUITE : " +getLig() + " " +
            getCol() + " ("+oldness+")") ;
     else
        return  ("CONFIRMATION : " +getLig() + " " +
         getCol() + " ("+oldness+")") ;}

  public tacheConf(coordsTime c, double mort, double confirm,
                   double confirmRep,long Id){
     super(0,5,c.getCoords().getCol()) ;
     argh = new coordsTime(c);
     this.mort = mort ;
     this.confirm = confirm ;
     this.confirmRep = confirmRep ;
     this.Id = Id ;}

  private void setPursuit(){
     pursuit = true ;}

  public coordsTime play(double toto, int tim) {
     if (pursuit)
```

```
            if (repeated)
               if (toto <= confirmRep)
                  return new coordsTime(new coords(1,2,3), new time(4)) ;
               else
                  return null ;
            else if (toto >= mort)
               return new coordsTime(new coords(1,2,3), new time(4)) ;
            else
               return null ;
         else if (toto <= confirm)
            return new coordsTime(new coords(1,2,3), new time(4)) ;
         else
            return null ;}}
```

C.1.12 task.java

Implémentation de l'interface *tache* pour le cas d'une tâche de surveillance.

```
   public abstract class task implements tache {

private int priorite = 0 ;
      public void setPriorite(int i){
         priorite = 0 ;}
      public int getPriorite(){
         return priorite ;}

      private int frequence = 0 ;
      public void setFrequence(int i){
         frequence = 0;}
      public int getFrequence(){
         return frequence ;}

      public abstract coordsTime play(double toto, int tim) ;

      private int col = 0 ;
      public void setCol(int i) {
         col = i ;}
      public int getCol() {
         return col;}

      public task(int f, int p, int i)
      { setPriorite(p) ;
         setFrequence(f) ;
         setCol(i) ;}}
```

C.1.13 temperature.java

Module effectuant la simulation du composant d'émission du radar. (voir paragraphe 7.2).

```
   public class temperature{

      private double temp = 0.0 ;
      private double tempMax = 0 ;

      public temperature(double tempMax){
         setTemperature(0.0) ;
         this.tempMax = tempMax ;}
```

```
    public temperature(temperature t){
       setTemperature(t.getTemperature()) ;
       tempMax = t.getMax() ;}

    public double getTemperature() {
       return temp ;}

    public void setTemperature(double temp){
       if (temp < 0.0) temp = 0.0 ;
       this.temp = temp ;}

    public double getMax() {
       return tempMax ;}

    public void tension (long t){
       setTemperature(getTemperature() + (double)t * 10.0) ; }

    public void relaxation(long t){
       setTemperature(getTemperature() - (double)t) ;}

    public boolean claque() {
       return (temp >= tempMax) ;}

    public String toString() {
       return ("Temperature : " + getTemperature()) ;}}
```

C.1.14 tempusFugit.java

Module donnant les plages de temps libres ou occupés.

```
import java.util.Vector ;

public class tempusFugit{
   private intervalle free = null ;
   private intervalle sche = null ;
   private long [] noms = null ;
   private int pf = 0 ;
   private int ps = 0 ;
   public tempusFugit(intervalle free, intervalle sche){
      init() ;
      free().duplique(free) ;
      sche().duplique(sche) ;}
   public tempusFugit(){
      init() ;}
   private void init(){
      this.free = new intervalle() ;
      this.sche = new intervalle() ;
      long noms [] = null ;}

// les fonctions associees a free
   private intervalle free(){
      return free ;}
   public int freeSize() {
      return free().size() ;}
   public void freeAddElement(tranche t) {
      free().addElement(t);}
   public tranche freeElementAt(int i) {
      return ((tranche)free().elementAt(i)) ;}
   public void freeRemoveElementAt(int i) {
      free().removeElementAt(i) ;}
   public int freeInserable(tranche probe) {
      return (free().inserable(probe)) ;}

// les fonctions associees a Sche
   private intervalle sche(){
      return sche ;}
```

```
   public int scheSize() {
      return sche().size() ;}
   public void scheAddElement(tranche t) {
      sche().addElement(t);}
   public tranche scheElementAt(int i) {
      return ((tranche)sche().elementAt(i)) ;}
   public void scheRemoveElementAt(int i) {
      sche.removeElementAt(i) ;}

// fonctions d'extraction de temps libres.

   private void upfree (tranche oldfree, tranche insere, intervalle free) {
   // on met dans free les intervalles de temps non occupes par insere
      long a = oldfree.debut() ;
      long b = oldfree.fin() ;
      long c = insere.debut() ;
      long d = insere.fin() ;
      tranche t = null ;
      if (a != c) free.addElement(new tranche(a,c,"",-1,-1,-1)) ;
      if (b != d) free.addElement(new tranche(d,b,"",-1,-1,-1)) ;}

   public void upfree (tranche oldfree, tranche insere) {
      upfree(oldfree,insere,free());}
   public void oneFree(tranche oldfree, tranche n1, tranche n2){
   // extraction du temps restant libre dans oldfree
   // apres insertion de n1 et n2
      intervalle tampon = new intervalle() ;
      this.upfree(oldfree, n1, tampon) ;
      int h = (tampon.size()) ; h--;
      oldfree = tampon.elementAt(h) ;
      tampon.removeElementAt(h) ;
      this.upfree(oldfree, n2,tampon) ;
      for (int i = 0 ; i < tampon.size() ; i++)
           freeAddElement(tampon.elementAt(i)) ;}

   public void twoFree(tranche of1, tranche of2, tranche n1, tranche n2){
      this.upfree(of1,n1,free()) ;
      this.upfree(of2,n2,free()) ;}

// les communs

   public void evalue (temperature t, long timeMax,long as){

      intervalle pouic = new intervalle() ;
   // on commence par le free

      int i = 0 ;
      boolean over = (freeSize() == 0) ;
      while (!over){
         tranche tr = freeElementAt(i) ;
         if (timeMax > tr.fin())
         pouic.addElement(
         new tranche(tr.debut(),tr.fin(),"F",0,0,tr.lID()));
         else {
            if (timeMax > tr.debut())
               pouic.addElement(
               new tranche(tr.debut(),timeMax,"F",0,0,tr.lID()));
            over = true ;}
         i++ ;
         if (i == freeSize()) over = true ;}

      i = 0 ;
      over = (scheSize() == 0) ;
      while (!over){
         tranche tr = scheElementAt(i) ;
         if (timeMax > tr.fin()) {
            if (((tr.name()).substring(0,1)).equals("P")) {
               pouic.addElement(
               new tranche(tr.debut(),tr.debut()+as,"S",0,0,tr.lID()));
               pouic.addElement(
```

```
                new tranche(tr.debut()+as,tr.fin(),"F",0,0,tr.lID())); }
             else{
                if (((tr.name()).indexOf("DEBUT")) != -1)
                pouic.addElement(
                new tranche(tr.debut(),tr.fin(),"S",0,0,tr.lID()));
                else
                pouic.addElement(
                new tranche(tr.debut(),tr.fin(),"F",0,0,tr.lID()));}}
          else over = true ;
          i++ ;
          if (i == scheSize()) over = true ;}

       pouic.tri() ;
       while (pouic.size() > 0){
          tranche tr = pouic.elementAt(0) ;
          pouic.removeElementAt(0) ;
          if ((tr.name()).equals("F")) t.relaxation(tr.duree()) ;
          else t.tension(tr.duree()) ;
          if (t.claque()) pouic.vider() ;}}

    public int nbSche(){
       int nb = 0 ;
       noms = new long[nbEven()] ;
       for (int i = 0 ; i < scheSize() ; i++){
          tranche t = scheElementAt(i) ;
          boolean trouve = false ;
          for (int j = 0 ; j < nb ; j++)
          trouve = (trouve || (noms[j] == t.lID())) ;
          if (!trouve) {
             noms[nb] = t.lID() ;
             nb++;}}
       return (nb) ;}
    public int nbEven() {
       return (freeSize()+ scheSize()) ;}
    public long[] getNames(){
       return noms ;}
    public void initNext(){
       pf = 0 ;
       ps = 0 ;}
    public tranche getNext(){
       tranche tf = null ;
       if (pf < freeSize()) tf = freeElementAt(pf) ;
       tranche ts = null ;
       if (ps < scheSize()) ts = scheElementAt(ps) ;
       tranche retour = null ;
       if ((tf == null) && (ts != null)) {
          ps++;
          retour = ts ;}
       else if ((ts == null) && (tf != null)) {
          pf++;
          retour = tf ;}
       else
          if (tf.debut() < ts.debut()){
             pf++ ;
             retour = tf ;}
          else{
             ps++;
             retour =  ts ;}
       return retour ;}
    public void tri(){
       free().tri() ;
       sche().tri() ;}
    public tempusFugit backup(){
       return (new tempusFugit(free(),sche())) ;}
    public tempusFugit over(long t) {
       return (new tempusFugit(free().over(t),sche().over(t))) ;}
    public int enBloc(long t){
       return free().enBloc(t) ;}
    public String toString() {
       return ("\r\nSCHEDULED : "+sche() + "\r\nFREE : "+free()) ;}
```

```
private class intervalle {
   private Vector toto = null ;
   public intervalle () {
      toto = new Vector() ;}
   public int size() {
      return toto.size() ;}
   public void addElement(tranche t) {
      toto.addElement(t);}
   public tranche elementAt(int i) {
      return ((tranche)toto.elementAt(i)) ;}
   public void removeElementAt(int i) {
      toto.removeElementAt(i) ;}
   public int enBloc(long t){
      int ret = -1 ;
      int i = 0 ;
      while ((ret == -1)&&(i < size()))
         if (elementAt(i).enBloc(t)) ret = i ;
         else i++ ;
      return ret ; }
   public void vider(){
      while (size() > 0) removeElementAt(0) ;}
   public void duplique(intervalle inter){
      vider() ;
      for (int i = 0 ; i < inter.size() ; i++)
      addElement(new tranche(inter.elementAt(i))) ;}
   public String toString() {
      return toto.toString() ;}
   public intervalle over(long temps){
      tranche t = null ;
      int i = 0 ;
      intervalle retour = new intervalle() ;
      while (i < size()) {
         t = elementAt(i) ;
         if ((t.aEcouler(temps))) {
            retour.addElement(t) ;
            removeElementAt(i) ;}
         else i++ ;}
      return retour ;}
   public int inserable(tranche t){
      int retour = 0 ;
      while ((retour < size()) &&
      (!(elementAt(retour)).inclus(t))) retour ++ ;
      if (retour < size())
         return retour ;
      else
         return (-1) ;}
   public void tri(){
      intervalle toto = new intervalle() ;
      while(size() > 0) {
         toto.addElement(elementAt(0)) ;
         removeElementAt(0) ;}

      tranche t = null ;
      while (toto.size() > 0){
         int iMin = 0 ;
         long timeMin =  (toto.elementAt(0)).debut() ;
         for (int i = 1 ; i < toto.size() ; i++) {
            t = toto.elementAt(i) ;
            if (timeMin > t.debut()) {
               timeMin = t.debut() ;
               iMin = i ;}}
         addElement(toto.elementAt(iMin)) ;
         toto.removeElementAt(iMin) ;}}}
```

C.1.15 time.java

Module stockant un temps.

```
public class time {

   private long t = 0 ;

   public time(long t){
      this.t = t ;}
   public time(time t){
      this.t = t.getTime() ;}
   public long getTime() {
      return t ;}

   public void setTime(long t){
      this.t = t ;}
   public String toString(){
      return "(temps = "+getTime()+" )";}

}
```

C.1.16 tranche.java

Module stockant un évènement du radar : Son type (surveillance, confirmation, ...), son emplacement dans l'espace de surveillance du radar, et ses coordonnées temporelles.

```
public class tranche{


   private long debut = 0 ;
   private long fin = 0 ;
   private String nom =  null ;
   private int x = 0 ;
   private int y = 0 ;
   private long id = 0 ;

   public tranche(long debut, long fin, String n, int a, int b, long id){
      init(debut,fin,n,a,b,id) ;}
   public tranche(tranche t){
      init(t.debut(),t.fin(),t.name(),t.getLig(),t.getCol(),t.lID());}
   private void init(long debut, long fin, String n, int a, int b,long id){
      this.debut = debut ;
      this.fin = fin ;
      this.nom = new String(n);
      this.x = a ;
      this.y = b ;
      this.id = id  ;}
   public long debut () {
      return debut ;}
   public long fin () {
      return fin ;}
   public long duree() {
      return (fin - debut) ;}
   public int getLig() {
      return x;}
   public int getCol() {
      return y;}
   public String name() {
      return nom;}
   public long lID(){
```

```
    return id ;}
public boolean aEcouler(long t){
    return (t < fin()) ;}
public history toHist(String s){
    return (new history(s, debut(), fin()-debut(), x, y));}
public history toHist(){
    return (new history(nom, debut(), fin()-debut(), x, y));}
public boolean inclus(tranche t){
    return ((t.debut() >= debut())&&(t.fin() <= fin())) ;}
public String toString() {
    return (toHist().toString());}
public String aff(){
    String non = name().substring(0,name().indexOf(' ')+1) ;
    return lID() + " : " + non + " ["+getLig()+ ","+getCol()+"]" ;}
public boolean enBloc(long t){
    return ((debut() <= t)&&(fin() > t)) ;}}
```

C.2 Entrées/Sorties du programme.

C.2.1 Un écran d'exécution du programme

Cet écran représente ce qui est affiché lors d'une simulation. Trois types d'informations y sont fournies :

- Le pourcentage d'exécution de la simulation, c'est-à-dire le pourcentage de nombre de tours d'antenne déjà effectué par rapport au nombre total de tours d'antenne que l'on a défini pour la simulation.
- Le temps restant : c'est une estimation du temps restant avant que la simulation ne soit terminée. Cette estimation est basée sur le fait que l'on se place en régime permanent, que l'on connait le pourcentage de la simulation déjà effectué, et le temps que cela a pris.
- La dernière partie concerne les évènements de la simulation : un affichage est effectué à chaque fois qu'il est impossible d'effectuer des tâches. Cet affichage est réalisé au fur et à mesure de la simulation.

Une capture d'un de ces écrans est donnée dans la figure C.1.

Une option du fichier de paramètres de la simulation (voir paragraphe C.2.2) permet d'afficher tous les évènements (toutes les tâches effectuées et leur résultat) de la simulation et de les situer temporellement.

C.2.2 Un exemple de paramètres du programme

Les données nécessaires à la simulation sont fournies au programme via un fichier appelé *params.dat*. Ce fichier contient les données suivantes :

- $longueurTrain$: Nombre de bitâches pour une tâche de confirmation.

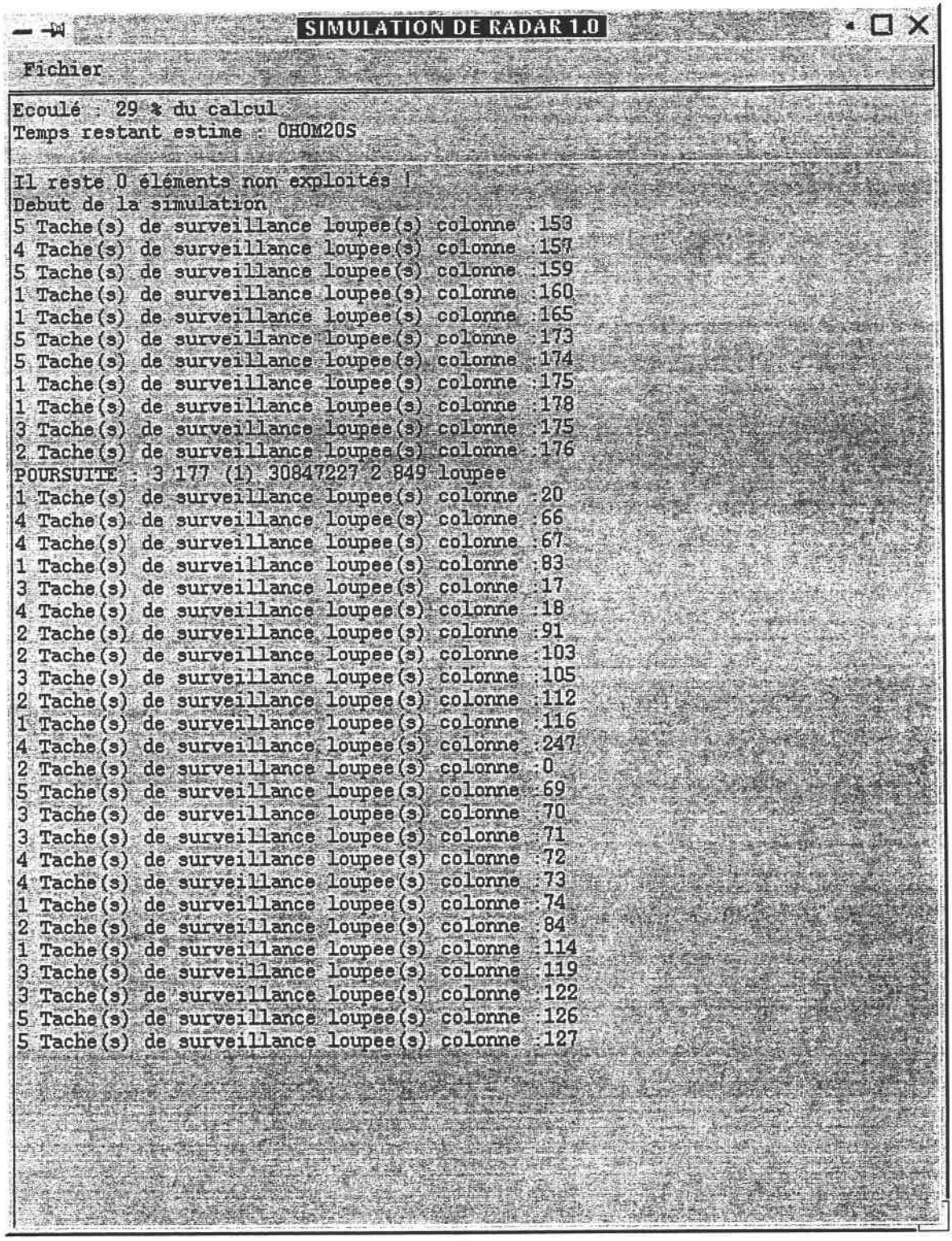


FIG. C.1 – Un écran d'exécution du programme.

- *attenteMin* : attente minimale entre la détection d'une cible et le début de la tâche de confirmation associée
- *dureeMax* : attenteMin+ dureeMax est égal au temps maximal devant s'écouler entre la découverte d'une nouvelle cible et la fin de l'exécution de la tâche associée.
- *intersstask* : temps devant s'écouler entre l'exécution de deux tâches.
- *intersstaskdelta* : flexibilité sur le temps devant s'écouler entre l'exécution de deux tâches.
- *distMin* : Distance minimale de compétence du radar.
- *distMax* : Distance maximale de compétence du radar.
- *c* : célérité de la lumière.
- *deperdition* : déperdition d'énergie du radar.
- *cross* : target cross section. (extrapolation)
- *probeRefresh* : nombre de tours d'antenne permis pour une exploration complète de l'espace à surveiller.
- *lig* : nombre de lignes pour modéliser le ciel observable par le radar.
- *col* : nombre de colonnes pour modéliser le ciel observable par le radar.
- *tour* : temps mis par le radar pour effectuer un tour d'antenne (en μs).
- *fin* : nombre de tours d'antenne à effectuer durant la simulation.
- *angle* : angle de déflection des composantes d'émission/réception de l'antenne.
- *jaffiche* : affichage de données de débugage si positionné à *true*.
- *killer* : lorsque *killer* est positionné à *true*, toutes les fenêtres du programme sont fermées lorsque la simulation est terminée.
- *entrelace* : entrelacement des tâches permis si positionné à *true*.
- *snapshotVal* : nombre de tours d'antenne entre les collectes des données.
- *p* : probabilité de détection d'un écho pour une tâche de surveillance.
- *q* : probabilité de disparition d'une cible.
- *confirm* : probabilité de confirmation d'un écho si la tâche de confirmation est constituée d'une seule bitâche
- *confirmRep* : probabilité de confirmation d'un écho si la tâche de confirmation est constituée de plusieurs bitâches.
- *maxItems* : Nombre maximal de cibles considérées en même temps.

```
longueurTrain 12
attenteMin 50000
dureeMax 50000
intersstask 5
intersstaskdelta 1
distMin 10000
distMax 300000
c 300000000
deperdition 0.00000000000000000001
cross 0.5
probeRefresh 2

lig 5
col 256
tour 4000000
fin 50
angle 40

jaffiche false
killer true
entrelace true
snapshotVal 10

p 0.3
q 0.22
confirm 0.1
confirmRep 0.1
maxItems 100

MaxTemps 3000.0
```

FIG. C.2 – Exemple de fichier de paramètres.

- $MaxTemps$: valeur maximale de la température.

Un exemple de fichier de paramètres est donné dans la figure C.2.

C.2.3 Un exemple d'écran de sortie du programme.

Cet écran affiche une petite partie des tâches qui sont exécutées lors d'une simulation. Cet écran se décompose en deux : le haut de la figure donne l'ordre d'exécution des tâches, ainsi que leurs caractéristiques. Le lecteur remarque que les tâches de réception sont signalées par une case barrée. Le bas de la

figure donne l'endroit du ciel où est exécutée la tâche. La figure C.3 donne une exemple de cet affichage dans le cas non entrelacé.

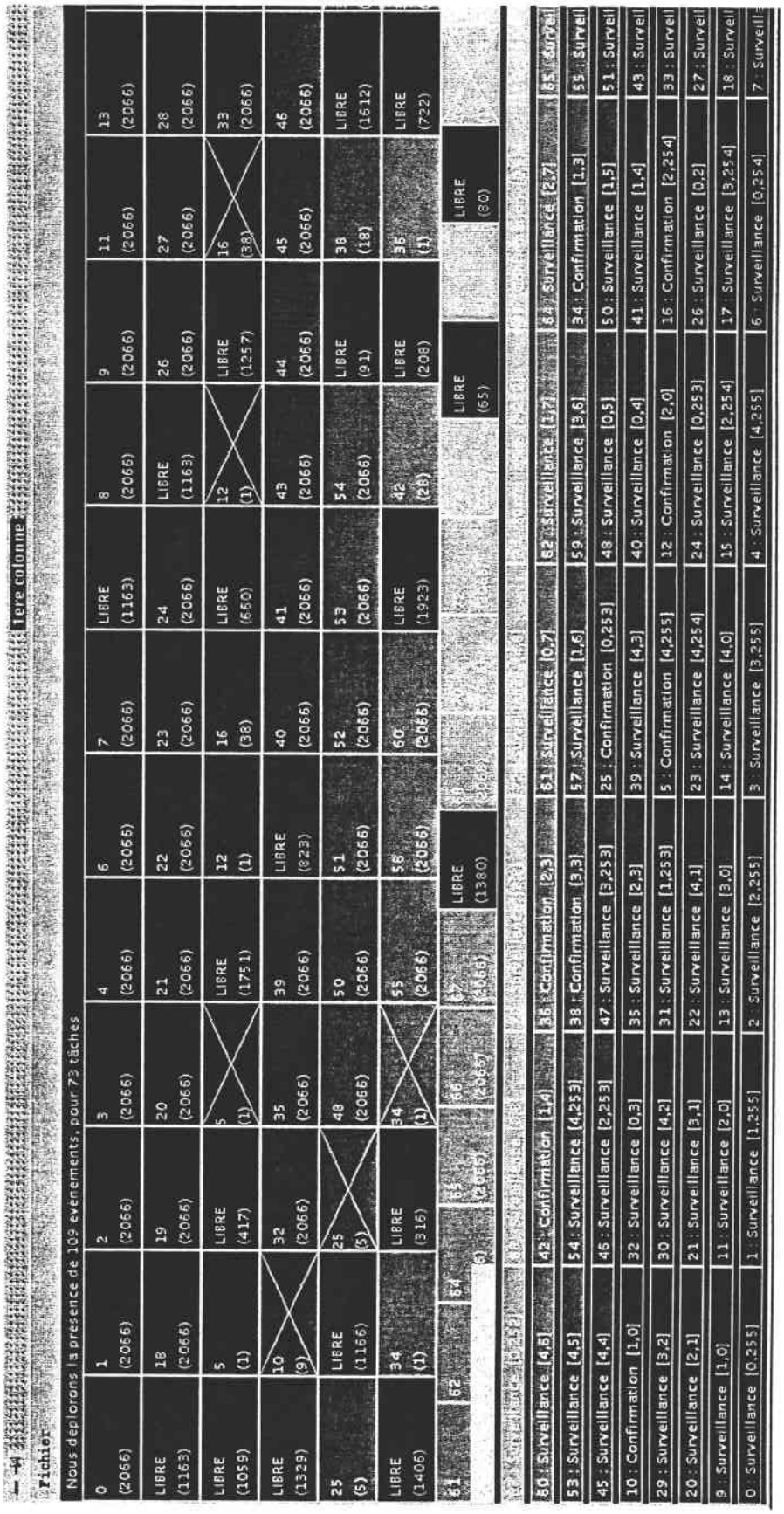


FIG. C.3 – Exemple d'écran de sortie du programme de simulation.

Références bibliographiques.

Bibliographie

[1] Miller L.W. Conway R.N. Maxwell W.L. *Theory of scheduling.* Addison-Wesley, Reading, Massachusetts, Etats-Unis, 1967.

[2] Baker K.R. *Introduction of sequencing and scheduling.* Wiley, New York, Etats-Unis.

[3] Coffman E.G.Jr. *Computer and job shop scheduling theory.* Wiley, New York, Etats-Unis, 1976.

[4] Rinnooy Kan A.H.V. *Machine scheduling problems : classification, complexity and computations.* Nijhoff, The Hague, 1976.

[5] Dempster M.A.H. Lenstra J.K. Rinnooy Kan A.H.G. *Deterministic and stochastic scheduling.* Reidel, Dordrecht, Pays-Bas, 1982.

[6] French S. *Sequencing and scheduling : an introduction to the mathematics of the job shop.* Ellis Horwood, Chichester, Grande-Bretagne, 1982.

[7] Carlier J. Chretienne P. *Les problèmes d'ordonnancement.* Masson, Paris, France, 1988.

[8] Blazewickz J. Ecker K. Schmidt G. Weglarz J. *Scheduling in computer and manufacturing systems.* Springer-Verlag, Berlin, Allemagne, 1993.

[9] Morton T. E. Pentico D. *Heuristics Scheduling Systems.* Wiley, New York, Etats-Unis, 1993.

[10] Feldmann A. Sgall J. Teng S.H. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, Vol. 130, No. 1, pp. 49-72., 1994.

[11] Tanaev V.S. Gordon V.S. Shafransky Y.M. *Scheduling theory : single-stage systems.* Kluwer, Dordrecht, Pays-Bas, 1994.

[12] Tanaev V.S. Sotskov Y.N. Strusevich V.A. *Scheduling theory : multi-stage systems.* Kluwer, Dordrecht, Pays-Bas, 1994.

[13] Zweben M. Fox M. *Intelligent scheduling.* Morgan and Kaufmann, San Mateo, Californie, Etats-Unis, 1994.

[14] Brucker P. *Scheduling algorithms.* Springer-Verlag, Berlin, Allemagne, 2001.

[15] Chretienne P. Coffmann E.G.Jr. Lenstra J.K. Liu Z. *Scheduling theory and its applications.* Wiley, New York, Etats-Unis, 1995.

[16] Pinedo M. *Scheduling : theory, algorithms, and systems.* Prentice-Hall, Englewood Cliffs, New Jersey, Etats-Unis, 1995.

[17] Chu C. Proth J.-M. *L'ordonnancement et ses applications.* Masson, Paris, France, 1996.

[18] Koole G.M. Stochastic scheduling and dynamic programming. *Journal of the Operational Research Society Vol. 47, No. 10, pp. 1313*, Vol. 47, No. 10, pp. 1313, 1996.

[19] Lai T.C. Sotskov Y.N. Sotskova N.YU. Werner F. Optimal makespan scheduling with given bounds of processing times. *Mathematical and Computer Modelling*, Vol. 26, No. 3, pp. 67-86, 1997.

[20] Shmelev V.V. Dynamic problems of scheduling. *Automation and Remote Control*, Vol. 58, No. 1-2, pp. 98-101, 1997.

[21] Esquirol P. Lopez P. *L'ordonnancement.* Economica, Paris, France, 1999.

[22] Albers S. Brucker P. The complexity of one-machine batching problems. *Discrete Applied Mathematics*, Vol. 47, pp. 87-107, 1993.

[23] Beauquier D. Bersyel J. Chretienne P. *Elément d'algorithmique.* Masson, 1992.

[24] Wigderson A. Ben-David S. Borodin A. Karp R.M. Tardos G. On the power of randomization in on-line algorithms. *Algorithmica*, Vol.11, pp. 2-14., 1994.

[25] Brucker P. Knust S. Complexity results for single-machine problems with positive finish-start time-lags. *Osnabruecker Schriften zur Mathematik*, Reihe P, No. 202, 1998.

[26] Chauvet F. Proth J.-M. Scheduling heuristics for minimizing the makespan. *9th SIAM-Discrete Mathematics'98, Toronto, Canada*, 1998.

[27] Chauvet F. Proth J.-M. Wardi Y. On-line scheduling with wip regulation. *Proceedings of the RensselaerÕs International Conference on Agile, Intelligent, and Computer-Integrated Manufacturing, Troy, New York*, 1998.

[28] Gondran M. Minoux M. *Graphes et algorithmes.* 1979.

[29] Nowicki E. An approximation algorithm for a single-machine scheduling problem with release times, delivery times and controllable processing times. *European Journal of Operational Research*, Vol. 72, No. 1, pp. 74-82., 1994.

[30] Panwalkar S.S. Rajagopalan R. Single-machine scheduling with controllable processing times. *European Journal of Operational Research*, Vol. 59, pp. 298-302, 1992.

[31] Chu C. Proth J.-M. *L'ordonnancement et ses applications.* Masson, January 1996.

[32] Faure R. Lemaire B. Picouleau C. *Précis de Recherche Opérationnelle.* Dunod, 2000.

[33] Glover F. Laguna M. *Taboo Search.* Kluwer Academic Publishers.

[34] Yagiura M. Ibaraki T. Genetic and local search algorithms as robust and simple optimization tools. *citeseer.nj.nec.com/yagiura96genetic.html*, pp 63-82., 1996.

[35] Bauer A. Bullnheimer B. Hartl R. F. Strauß C. Minimizing total tardiness on a single machine using ant colony optimization. *citeseer.nj.nec.com/234595.html*, 19??

[36] Yu-Chung Wang Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-linux real-time kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255, 1999.

[37] Mok A. *Fundamental Design Problems of Distributed Systems for the hard Real-Time Environment.* PhD thesis, MIT Laboratory for Computer Sciences, 1983.

[38] Hakan Aydin Rami G. Melhem Daniel Mosse Pedro Mejia-Alvarez. Optimal reward-based scheduling of periodic real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 79–89, 1999.

[39] Schwiegelshohn U. Preemptive weighted completion time scheduling of parallel jobs. In *European Symposium on Algorithms*, pages 39–51, 1996.

[40] Chauvet F. Levner E. Proth J.-M. On-line part scheduling in a surface traitment system. *Journal of Operational Research, IJOR,*, vol. 120/2, January 2000.

[41] Liu Z. Finta L. Single machine scheduling subject to precedence delays. *Rapport de recherches, INRIA*, n°2198, 1994.

[42] R.M. Russel and J.E. Holesnback. Evaluation of greedy, myopic and less greedy heuristics for the single machine, total tardiness problem. *Journal of the Operation Reasearch Society*, Vol.48 pp.640-646, 1997.

[43] Panwalkar S.S. Smith M.L. Russel R.M. A heuristic algorithm for sequencing on one machine to minimize total tardiness. *European Journal of Operation Research*, Vol. 70, pp304-310, 1992.

[44] C.N. Potts and L.N. Van Wassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operation Research Letters*, Vol. 1 N°5 pp.177-181, 1982.

[45] Schrage L. Baker K.R. Dynamic programming solution of sequencing problems with precedence constraints. *Operation Research*, Vol. 26 pp444-449, 1978.

[46] Orman A.J. Potts C.N. Shahani A.K. Moore A.R. Scheduling for a multifunction array radar system. *European Journal of operational research*, 90, pp 13-25, 1996.

[47] Orman A. J. Modelling for the control of a complex radar system. *Computers Ops Res.*, Vol. 25, n°3, pp 239-249, 1998.

[48] Orman A. J. *Models for scheduling a multifunction phased array radar system.* PhD thesis, University of Southampton, 1998.

[49] Fiat A. Woeginger G. Sgall J. *Lectures notes in computer science, chapter 9 : On-line algorithms-The state of the art.* Springer-Verlag, pp. 196-231, 1998.

[50] Knuth D.E. *The art of computer programming.* Vol. I-III, Addison-Wesley, Reading, Massachusetts, Etats-Unis, 1968-1973.

[51] Gaudel M.C. Soria M. Froidevaux C. *Types de données et algorithmes. Volume 2 : recherche, tri, algorithmes sur les graphes.* Collection Didactique, D-004, Inria, Rocquencourt, France, 1987.

[52] Garey M.R. Johnson D.S. *Computers and intractability. A guide to the theory of NP-Completeness.* W. H. Freeman And Company, New York, Etats-Unis, 1979.

[53] Mao W., Kincaid R. X., and Rifkin A. On-line algorithms for a single machine scheduling problem. *The Impact of Emerging Technologies on Computer Science and Operations Research*, pp. 157–173, 1995.

[54] Billeter D.R. *Multifunction Array Radar.* Artech House, 1989.

[55] Duron C. Proth J.-M. Linked task scheduling : Algorithms for the single machine case. *Submitted to the Journal of Combinatorial Optimization and INRIA Research Report 4602.*

[56] Duron C. Les radars multi-fonctions. *Internal publication for the French Arming Military Agency*, confidential, 2000.

[57] Duron C. Proth J.-M. Multifunction radar : Task scheduling. *Journal of Mathematical Modelling and Algorithms*, 1, num. 2, pp 105-116, ISSN 1570-1166, 2002.

[58] Duron C. Proth J.-M. Wardi Y. Insertion of a random task in a schedule : a real-time approach. *IEEE-ETFA Proceedings, Antibes-Juan les Pins, France*, Vol. 1 pp 559-565, ISBN 0-7803-7241-7, and INRIA Research Report 4337, 2001.

[59] Duron C. Proth J.-M. Wardi Y. Single resource scheduling : Insertion of a random task under real-time constraint. *IVth Brazilian Workshop on Continuous Optimization, Rio de Janeiro, Brazil*, July15-23, 2002.

[60] Duron C. Proth J.-M. Insertion of a random bitask in a schedule : a real-time approach (II), submitted to EJOR, and INRIA research report 4193. Nov. 2001.

[61] Duron C. Proth J.-M. Insertion of a random bitask in a schedule : a real-time approach (I). *Proceedings of the XV_{th} Conference of the European Chapter on Combinatorial Optimisation, Lugano, Switzerland, submitted to Computers and Operation Research*, May30-Junel, 2002.

[62] Duron C. Proth J.-M. Bitask scheduling : A heuristic approach under real-time constraints submitted to *journal of scheduling* . 2002.

[63] Cummings S. Behar K. Radar resource management for mechanically rotated, electronically scanned phased array radars. *procs. of IEEE National Radar Conference, Los Angeles*, pp.12-13, 1991.

[64] Shapiro R.D. Scheduling coupled tasks. *Naval. Res. Logist. Quart.*, 27, pp 477-481, 1980.

[65] Milevic D.J. Popovic B.M. Improved algorithms for the interleaving of radar pulses. *IEE Proceedings F 139, pp98-104*, 1992.

[66] Dijskstra E. A note on two problems in connexion with graphs. *Num. Maths.*, 1, pp 269-271, 1959.

[67] Chauvet F. Proth J.-M. On-line scheduling in assembly processes. *Information Systems and Operation Research*, vol.39 n°1, feb 2001.

[68] Dertouzos M. Control robotics : the procedural control of physical processors. *Proceedings of the IFIP Congress*, pp. 807-813, 1974.

[69] Lehoczky J.P. Sha L. and Ding Y. The rate monotonic scheduling algorith. exact characterization and average case behavior. *IEEE Real-time Systems Symposium*, 1989.

[70] Liu C.L. Layland J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, vol. 20, n°1, pp. 46-61, 1973.

[71] Proth J.-M. *On-line Scheduling in Supply Chain Environment*, chapter Optimal Control and Partial Differential Equations. Eds. Menaldi et al., IOS Press, 2001.

[72] Zhao W. Ramamritham K Stankovic A. Preemptive scheduling under time and resource constraints. *IEEE Transaction on Computers*, vol. 36, n°8, 1987.

[73] Lenstra J.K. Rinnooy Kan K. Graham R.L. Lawler E.L. Optimization and approximation in deterministic scheduling : A survey. *Ann. Disc. Math.*, 5. pp 287-326, 1979.

[74] Huizing A. Bossé E. A high-level multifunction radar simulation for studying the performance of multisensor data fusion systems. *Part of the*

SPIE conference on signal processing, Sensor fusion, and target recognition VII, Orlando, Florida, April 1998.

[75] Weinberg L. Scheduling multifunction radar systems. *RCA Government Systems, Division Missile and Surface Radar, Moorestown, N.J., 08057*, <10-4A-EASCON77 to 10-4J-EASCON77>,?

[76] Kohei Noshita. A theorem on the expected complexity of dijkstra's shortest path algorithm. *Journal of Algorithms*, 6(3) :400-408, September 1985.

[77] Chauvet F. Duron C. Proth J.-M. The PERT problem with alternatives : some basic results. *Proceedings of the International Workshop on Discrete Optimization Methods in Scheduling and Computer-Aided Design*, pp 10-14, ISBN 985-6453-49-6, Minsk 2000.

[78] Duron C. Parent M. Proth J.-M. Analysis of the balancing process in a pool of self-services cars. *INRIA*, Research Report 3949, 2000.

[79] Duron C. Proth J.-M. The 0-1 outcomes feature selection problem. *submitted to Data Knowledge and Engineering*, 2000.

[80] Duron C. Proth J.-M. Wardi Y. On-line scheduling in supply chain environment. *Proceeding of the IASTED International Conference Robotics and Manufacturing, Cancun,Mexico*, pp117-122, ISBN 0-88986-281-8 ISSN 1027-264X, May21-24, 2001.

[81] Duron C. Hafez N. Parent M. Proth J.-M. Wire guided vehicles in self-service : A design tool for simulation models. *Proceeding of the IASTED International Conference Robotics and Manufacturing, Cancun, Mexico*, pp249-254, ISBN 0-88986-281-8, ISSN 1027-264X, May21-24, 2001.

Rapport d'après soutenance
et
Autorisation de reproduction.

UFR MIM
Mathématique
Informatique
Mécanique
Automatique

DOCTORAT
DE L'UNIVERSITE DE METZ

Thèse soutenue le 20 décembre 2002
par Monsieur DURON Cyril

Signatures des membres du jury :

- M. DOLGUI Alexandre
- M. PROTH Jean-Marie
- M. MUTEL Bernard
- M. SAKHO Ibrahima

RAPPORT APRES SOUTENANCE

sur la thèse ayant pour sujet :

« Ordonnancement en temps réel des activités des radars. »

MENTION OBTENUE :

Mention très honorable avec félicitations.

Le problème auquel s'est attaché Monsieur Duron est un problème d'ordonnancement en temps réel nouveau et difficile, et d'une grande importance pratique. Peu de chercheurs ont fourni des algorithmes satisfaisants pour apporter une réponse à cette problématique.

L'approche proposée fait appel à la fois à des connaissances informatiques, parmi lesquelles le calcul parallèle, et aux techniques d'ordonnancement en temps réel. La contrainte "temps réel" représente la difficulté majeure attachée à cette approche.

L'exposé de Monsieur Duron a été clair, concis, et a montré une réelle maîtrise du sujet. Le jury a

UFR MIM Mathématique Informatique Mécanique Automatique

RAPPORT APRES-SOUTENANCE (suite)

Apprécié que Monsieur Jurson ait su prendre du recul par rapport au sujet pour présenter le contenu de sa thèse en évitant des exposés trop techniques qui auraient obscurci les tenants et les aboutissants du projet.

Les réponses aux questions du jury ont été claires et précises et ont montré les bases théoriques de l'approche ainsi que les débouchés pratiques attendus par la DGA (Direction Générale de l'Armement) : la gestion des radars de combat multi-fonctions.

Pour toutes ces raisons, le jury a décidé d'attribuer la mention "Très honorable avec félicitations du Jury"

Fait à Metz le 20 décembre 2000

UNIVERSITE DE METZ

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Nom et Prénom de l'auteur : Monsieur DURON Cyril

Titre de la thèse : « Ordonnancement en temps réel des activités des radars. »

Date de soutenance : le 20 décembre 2002

Président du jury : P. B. Merhet

Membres du jury : Messieurs DOLGUI, MUTEL, PROTH et SAKHO

<u>Reproduction de la thèse soutenue</u> :

☒ **thèse pouvant être reproduite en l'état**

☐ **thèse pouvant être reproduite après corrections suggérées au cours de la soutenance**

☐ **thèse interdite de reproduction**

Le Président du Jury :

B. Merhet

Le Directeur de thèse mandaté par le Président du Jury atteste que les corrections ont bien été effectuées le :
Signature :

Ordonnancement en temps-réel des activités des radars

Thèse de Doctorat soutenue par Cyril DURON

Résumé - L'objectif général de cette thèse, suggéré par le contrôle des radars de combat, consiste à intercaler en temps réel une tâche aléatoire dans un ordonnancement existant tout en limitant autant que possible l'augmentation de la valeur du critère. Dans notre cas, le critère que nous considérons est la somme des dépassements des délais des tâches déjà ordonnancées. Ces délais sont supposés quelconques : cette contrainte est plus dure que dans le cas des radars de combat où un certain nombre de tâches de surveillance doivent être effectuées de manière répétitive au cours d'une période donnée à l'intérieur de laquelle leur ordonnancement est libre, ce qui équivaut à un délai unique pour l'ensemble des tâches. La tâche à intercaler apparaît à un instant quelconque (c'est l'instant que nous considérons comme l'instant zéro). Sa durée n'est connue qu'au moment de son apparition. Il en est de même de son délai, qui est impératif. Nous considérons d'abord le cas d'une tâche aléatoire unique, puis le cas d'une tâche aléatoire composée de deux sous-tâches séparées par une période donnée. Enfin, nous proposons une amélioration de l'approche actuellement utilisée dans ce domaine.

Mots-clés - Ordonnancement, Temps-Réel, Ressource Unique.

Real-time scheduling of radar activities

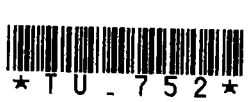
Ph. D. Thesis by Cyril DURON

Abstract - The goal of this thesis, inspired by the battle radar management, is to insert in real-time a random task in the current schedule while minimizing the criterion. In our case, the criterion is the minimization of the sum of the delays of the already scheduled tasks. There is no leading rule for these delays. It is a stronger constraint than in the battle radars case, because they have to perform an amount of repetitive tasks in a given period. This may be considered as a unique due-date for all the tasks. The task we have to insert may appear at any time (For the sake of simplicity, the time at which a random task appears is time zero). Its processing time and its delay, which cannot be violated, are known only at time zero. We firstly take into consideration the case of a single random task, then we consider the case of a task made of two subtasks separated by a given period.Finally, we propose an improvement of the approach currently used to manage battle radars.

Keywords - Scheduling, Real-time, Single resource.

INRIA-Lorraine
Université de Metz - Faculté des Sciences
UFR Mathématiques, informatique, mécanique
Ile du Saulcy - 57000 Metz

Thèse préparée à l'INRIA Lorraine dans le projet SAGEP



Imprimé à l'INRIA