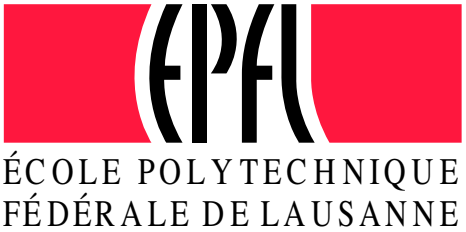


Projet de Semestre

Hiver 2005-2006

Ordonnancement online avec informations partielles supplémentaires

Sarah Emery (Math 4ème)

Responsable : Prof. Dominique de Werra

Chaire de recherche opérationnelle ROSE

Table des matières

Introduction

Le point de départ de ce projet est un article écrit par E. Angelleli, M. G. Speranza et Zs. Tuza (ANG05). Il traite de l'ordonnancement online de tâches sur des machines identiques, avec la somme des tailles des tâches connue.
Le but de ce projet est de se familiariser avec cette classe de problèmes, qui sont dits semi-online.
Pour cela, nous allons commencer par expliquer ce qu'est l'ordonnancement online en général. Ensuite nous parlerons des techniques d'évaluation des algorithmes qui résolvent ces problèmes, avant d'énoncer les principaux résultats dans le cas online. Nous continuerons notre travail sur la présentation des problèmes semi-online. Nous traiterons le même problème que dans le cas online, mais avec une information supplémentaire sur la donnée du problème. Après quoi, nous nous arrêterons un moment sur l'article cité ci-dessus, afin de mieux comprendre le mécanisme qui se cache derrière la recherche de bornes inférieures et supérieures pour les algorithmes (semi-)online. Finalement, nous terminerons par exposer les différents cas en semi-online qui ont été étudiés pour le problème d'ordonnancement de tâches, sans interruption, sur des machines identiques et nous proposerons une version du problème qui n'a pas encore été travaillée.

Chapitre 1

Les problèmes d'ordonnancement online

L'ordonnancement est une opération qui consiste à planifier l'exécution de tâches et l'allocation de ressources avec comme objectif l'optimisation d'une ou de plusieurs mesures de performance. Comme ressources, nous pouvons considérer les portes d'embarquement dans les aéroports, des conducteurs de bus, des machines dans une usine, ... Ainsi, les tâches correspondantes seraient, par exemple, les avions dans un aéroport, le travail des conducteurs des bus, l'assemblage de pièces dans une usine. Il y a aussi plusieurs fonctions que nous cherchons à minimiser, comme la durée totale d'exécution de tâches, le nombre de tâches réalisées en retard, le nombre de ressources utilisées ... .

Dans ce travail, nous nous intéressons au problème d'ordonnancer des tâches de tailles positives sur des machines identiques. La charge d'une machine est la somme des tailles de toutes les tâches affectées à cette machine. Et l'objectif est la minimisation de la charge maximale des machines, ce qui veut dire que nous cherchons à exécuter toutes les tâches en une durée totale minimale. Ce problème est connu comme NP-difficile. Il est assez important, car il est à la base de nombreux problèmes, dans lesquels, par exemple, une même tâche peut être coupée et exécutée sur deux machines différentes, il existe des contraintes de précédences entre les tâches, ou encore les machines exécutent les tâches à différentes vitesses.
L'ensemble des applications qui motivent la recherche dans ce domaine contient notamment les systèmes d'exploitation à plusieurs utilisateurs comme "Unix" et "Windows", les serveurs de bases de données, ...

En théorie, nous pouvons penser que la donnée du problème (nombre de tâches à effectuer, nombre de machines, grandeur des tâches, dates limites

d'exécution, ...) est connue à l'avance. Ainsi, sur la base des ces informations, nous pouvons essayer d'attribuer à chaque machine un certain nombre de tâches, de manière à minimiser la durée totale d'exécution. Dans ce cas-là, nous parlons de problèmes "offline".
Cependant, dans la réalité, nous ne connaissons bien souvent pas toutes les informations à l'avance sur les tâches à exécuter. Nous recevons des tâches une par une et il faut les ordonnancer immédiatement et irrévocablement sans aucune connaissance des tâches futures qui vont arriver. Nous appelons ce genre de problèmes les problèmes "online".

Le point important des algorithmes online est qu'ils n'ont pas accès à toute l'information sur le problème quand ils doivent prendre une décision. Ce manque de connaissance du futur empêche généralement de garantir un ordonnancement optimal. En fait, pour la plupart des problèmes, aucun algorithme online ne peut donner une solution optimale. Nous essayons alors de développer des algorithmes qui garantissent une solution proche de l'optimal. Nous voulons pouvoir mesurer la performance de ces algorithmes et l'outil qui est utilisé pour cela est l'analyse compétitive.

1.1 L'analyse de performance

On s'intéresse aux algorithmes online déjà depuis les années 1970, mais l'étude plus profonde et plus systématique de ces algorithmes a commencé quand Daniel Sleator et Robert Tarjan (SLE) proposèrent de comparer un algorithme online à un algorithme optimal offline et que Karlin et al. inventèrent le terme "competitiv analysis" (dans Competitive Snoopy Caching, Algorithmica, 3 : 79-119, 1988).

1.1.1 Analyse compétitive

Avant d'expliquer formellement ce qu'est l'analyse compétitive ("competitiv analysis"), illustrons-la par un petit exemple.
Supposons qu'un étudiant diplômé d'une certaine école reçoive une petite bourse pour faire un doctorat. Pendant ce temps, il reçoit régulièrement des offres d'emploi temporaire de différentes boîtes de consulting. Chaque travail est très bien payé. Mais ces projets demandent un tel investissement qu'il est impossible de faire le doctorat en même temps. Et après avoir accepté un emploi, il est impossible de reprendre le doctorat, en recevant à nouveau une bourse. Il n'y a aucun moyen d'estimer le flux de projets au cours du temps.

Si par contre le flux est continu, la valeur cumulée des salaires de la boîte de consulting est beaucoup plus élevée que la bourse (on ignore les bénéfices de l'enseignement lui-même). Ainsi, à chaque offre de projet de consulting, le doctorant est confronté à un réel problème d'ordonnancement online -faut-il oui ou non quitter l'école ?- alors qu'il n'a aucune connaissance des offres futures. Notons que le but est de maximiser l'argent gagné pendant la période où il recevrait la bourse sans aucun intérêt porté au contenu des projets eux-mêmes.

En général, nous n'avons aucun moyen d'obtenir la meilleure solution possible dans les problèmes online, mais nous aimerions éliminer la possibilité d'un désastre : une valeur terriblement mauvaise comparée à la meilleure solution possible, tel est le but de l'analyse compétitive.
Ainsi, cette dernière est l'étude de stratégies qui donnent une performance garantie dans n'importe quelles circonstances. Et c'est ce dont nous avons besoin pour les algorithmes online. En effet, le manque d'informations sur les données d'un problème ne permet pas aux algorithmes online qui le résolvent de produire une solution optimale dans tous les différents cas de figure. Nous aimerions pouvoir, pour chaque algorithme, connaître la performance qu'il garantit avec n'importe quelles données, et ainsi les comparer entre eux pour trouver le meilleur. Nous cherchons alors à calculer la pire performance des algorithmes.

Pour cela, nous utilisons une méthode assez intuitive qui consiste à comparer la plus mauvaise solution obtenue avec l'algorithme, dont nous voulons évaluer la performance, à la solution optimale du problème offline correspondant.
Notons $f(A, I)$ la valeur de la solution produite par l'algorithme A avec la liste des tâches I où f est la fonction que nous voulons minimiser, qui dans notre cas est la durée totale d'exécution des tâches.

Définition 1.1.1. *Un algorithme online A est dit c-compétitif si*

$$f(A, I) \leq c \cdot f(OPT, I) + b, \quad \forall \, \textit{liste de tâches} \, I,$$

où OPT représente la solution optimale de l'algorithme offline correspondant, dans lequel tout est connu à l'avance, et b une constante.

En général, nous laissons tomber la constante b, car nous nous intéressons aux cas asymptotiques.

Définition 1.1.2. *Le "ratio compétitif absolu" c_A est le nombre*

$$inf\{c \mid A \textit{ est } c\textit{-compétitif}\}$$

Nous cherchons alors des algorithmes dont le "ratio compétitif absolu" est le plus petit possible. Le mieux serait que c ne dépende d'aucun paramètre du problème, ce qui n'est pas toujours possible.

1.1.2 Analyse du cas le moins favorable

Nous pouvons mesurer la performance d'un algorithme sur la base d'une autre analyse, celle que nous appellerons analyse du pire cas ("worth case analysis"). Cela consiste à calculer une borne inférieure pour le "ratio compétitif absolu" de tous les algorithmes online pour un problème donné. En d'autres mots, nous ne pourrons trouver aucun algorithme online qui sera plus performant pour un problème donné que celui qui est b-compétitif où b est une borne inférieure. Et nous dirons de l'algorithme dont la performance correspond à la borne inférieure qu'il est optimal.

En résumé, dans la recherche d'algorithmes pour la résolution des problèmes online, nous sommes intéressés à calculer des bornes supérieures et des bornes inférieures. Nous cherchons à diminuer la borne supérieure, en trouvant des algorithmes dont la performance compétitive se rapproche le plus possible de 1. Cependant, nous arrivons à prouver que pour certaines données du problème considéré, aucun algorithme n'aura une meilleure performance qu'une certaine limite, qui est la borne inférieure. Nous essayons alors de trouver une borne supérieure toujours plus petite, tout en sachant qu'il n'existe pas d'algorithme avec une meilleure performance que la borne inférieure.

1.2 Résultats importants

Nous allons présenter dans cette section, les principaux résultats concernant les algorithmes online pour la résolution du problème d'ordonnancement de n tâches sur m machines identiques.

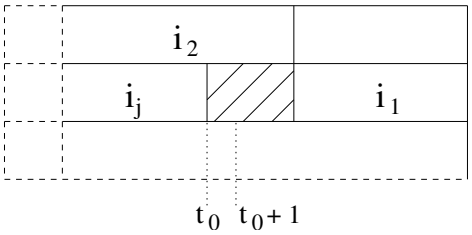
1.2.1 L'algorithme de liste de Graham

Le premier à donner une preuve de la compétitivité d'un algorithme online pour un problème d'ordonnancement fut Ronald L.Graham en 1966. Il travaillait sur un "simple" algorithme déterministe, que nous appelons communément l'Algorithme de Liste (LS). Nous nous arrêtons sur ce problème un instant, étant donné qu'il est très utilisé dans les problèmes online, de part sa puissance. En effet, comme cet algorithme est simple et que tous les ordonnancements optimaux peuvent être construits par un LS avec une liste

appropriée, LS est l'approche d'ordonnancement la plus utilisée. De plus, comme LS ne demande aucune connaissance des tâches imprévues aussi bien que de toutes les tâches en cours, il est très puissant pour les problèmes online.
Commençons par donner une idée de ce qu'est l'algorithme de liste. Les données sont constituées d'une liste de tâches à exécuter. Chaque fois qu'une machine se libère, la première tâche disponible de la liste est traitée et enlevée de la liste. Une tâche est disponible si elle se trouve en début de liste et si, possédant des contraintes de précédence, toutes les tâches qui doivent la précéder ont déjà été traitées.
Graham a considéré le problème de l'ordonnancement sur m machines de n tâches dont certaines sont dépendantes entre elles. Le but est de minimiser la durée totale d'exécution de ces n tâches. Nous pouvons montrer que tout LS construit une solution dont la distance en valeur relative à l'optimum est inférieure à $2 - \frac{1}{m}$
Mais pour pouvoir démontrer ce résultat, il nous faut le lemme suivant :

Lemme 1.2.1. *Soit T la durée cumulée des périodes d'inactivité des m machines dans un ordonnancement de liste. Alors, il existe un ensemble ordonné de tâches dont la durée d'exécution est inférieure à $\frac{T}{m-1}$.*

Démonstration. Soit i_1 une tâche commançant à la date t_{i_1} et se terminant à la date de fin d'ordonnancement. On pose t_0 la plus grande date strictement inférieure à t_{i_1} telle qu'une machine soit inactive dans l'intervalle $[t_0, t_0 + 1[$. Si à la date t_0 on n'a pas pu exécuter la tâche i_1, alors qu'une machine était libre, cela signifie que la tâche i_1 est contrainte à être exécutée après une tâche i_2 qui n'est pas encore terminée.



Par contre, d'après le choix de t_0, si à la fin C_{i_2} de la tâche i_2, la tâche i_1 n'est pas aussitôt exécutée, on peut dire qu'à ce moment-là toutes les machines sont occupées, donc actives.
Ainsi, si t_0 n'est pas défini, toutes les machines sont actives entre 0 et t_{i_1}. En

itérant ainsi, on trouve un ensemble ordonné de tâches i_1, ..., i_r telles que i_r précède i_{r-1}, ..., i_2 précède i_1.
De plus, on a que toutes les machines sont actives en dehors des périodes où s'exécutent ces tâches.
Soit p_{i_j} la durée de la tâche i_j, on a alors :

$$T \leq (p_{i_1} + p_{i_2} + ... + p_{i_r})(m-1)$$

Il existe donc un ensemble ordonné de tâches i_1, ..., i_r tel que

$$p_{i_1} + p_{i_2} + ... + p_{i_r} \leq \frac{T}{m-1}$$

□

Nous pouvons maintenant énoncer et démontrer le théorème suivant :

Théorème 1.2.2. *Soient D un algorithme de liste, $\hat{F}(D)$ sa valeur approchée et $F^*(D)$ la valeur optimale de l'algorithme offline. On a alors*

$$F^*(D) \leq \hat{F}(D)\left(2 - \frac{1}{m}\right)$$

Démonstration. Si on arrête toutes les machines seulement lorsque la dernière tâche a été effectuée, la durée totale d'un ordonnancement de liste est égale à la somme des périodes d'activité A et d'inactivité T de chacune des machines divisée par le nombre total de machines, qui ici vaut m. On peut écrire $A = \sum p_i$ et par le lemme précédent, $T \leq (p_{i_1} + p_{i_2} + ... + p_{i_r})(m-1)$.
On a alors

$$F^*(D) \leq \frac{\sum p_i + (p_{i_1} + p_{i_2} + ... + p_{i_r})(m-1)}{m}$$

Or, la valeur optimale de l'algorithme offline peut être évaluée par la somme des périodes d'activité divisée par le nombre de machines :

$$F^*(D) = \frac{\sum p_i}{m}$$

ou alors, par la somme de la suite ordonnée des tâches qui ont des contraintes de précédence :

$$F^*(D) = (p_{i_1} + p_{i_2} + ... + p_{i_r})$$

Ainsi, on obtient

$$\begin{aligned} F^*(D) &\leq \frac{1}{m}(F^*(D)m + F^*(D)(m-1)) \\ &\leq \frac{1}{m}(2F^*(D)m - F^*(D)) \\ &\leq \left(2 - \frac{1}{m}\right) F^*(D) \end{aligned}$$

□

1.2.2 Autres résultats importants

Nous pouvons résumer les autres résultats intéressants dans un tableau qui se trouve en page 13.

1.2.3 Les algorithmes online en pratique

Nous constatons que récemment, de nombreux algorithmes ont été développés. Mais comme l'analyse compétitive n'observe comment les algorithmes se comportent que dans les cas les moins favorables, nous pouvons nous demander si ces nouveaux algorithmes ne sont performants que dans ces situations pathologiques bien particulières ou alors s'ils donnent de meilleurs résultats en pratique aussi.

A ce sujet, une étude a été faite par Albers et Schröder (ALB02) qui confirme les résultats théoriques dans le sens qu'il existe bel et bien des problèmes dans la réalité qui sont mieux résolus par les nouveaux algorithmes.
Pour répondre à cette question, elles ont analysé les algorithmes d'abord sur des problèmes du monde réel et ensuite sur des problèmes générés par des distributions de probabilités. Il en résulte que la performance des algorithmes dépend fortement des caractéristiques des charges de travail. Quand les séquences de tâches sont données par des distributions de probabilités, la stratégie de Graham est clairement la meilleure. Par contre, dans le monde réel, les nouveaux algorithmes sont plus performants que ceux de Graham.
Il est donc intéressant et tout à fait utile d'étudier ces algorithmes et d'essayer de les rendre encore plus performants, car ce ne sont pas que des résultats qui sont vrais en théorie, mais que nous pouvons vraiment appliquer dans le monde qui nous entoure.

Auteur(s)	Année	Nombre de machines	Borne supérieure	Borne inférieure
Graham	1966	$\forall m$	$(2 - \frac{1}{m})$-compétitif	
Faigle, Kern & Turan	1989	$m = 2$ et $m = 3$	$(2 - \frac{1}{m})$-compétitif[1]	$(2 - \frac{1}{m})$-compétitif
Faigle, Kern & Turan	1989	$\forall m \geq 4$		1,707-compétitif
Galambos & Woeginger	1993	$\forall m$	$(2-\frac{1}{m}-\epsilon_m)$-compétitif, avec $(\epsilon_m > 0) \rightarrow 0$ quand $m \rightarrow \infty$	
Chen,van Vliet & Woeginger	1994	$m = 4$	1,733-compétitif	
Bartal, Karloff & Rabani	1994	$\forall m \geq 3454$	1,837-compétitif	
Bartal & al.	1995	$\forall m \geq 70$	1,986-compétitif	
Karger, Phillips & Torng	1996	$\forall m$	1,945-compétitif[2]	
Albers	1999	$\forall m \geq 80$		1,852-compétitif
Albers	1999	$\forall m \geq 2$	1,923-compétitif	
Fleischer & Wahl	2000	$m \rightarrow \infty$	1,9201-compétitif[3]	
Rudin II & Chandrasekaran	2003	$m = 4$		$\sqrt{3}$-compétitif

[1]Pour ces valeurs de m, l'algorithme de liste est optimal.
[2]Meilleure borne supérieure jusqu'en 1999 avec Albers.
[3]Meilleure borne supérieure jusqu'à présent.

Chapitre 2

Les problèmes d'ordonnancement semi-online

Travailler avec des algorithmes online purs n'est souvent pas réaliste. En effet, les problèmes de tous les jours contiennent généralement des informations partielles qui peuvent être exploitées par des algorithmes appropriés. De plus, la performance des algorithmes online n'est souvent pas très élévée, compte tenu de la très grande variance des différents paramètres concernant les tâches.
Ainsi, nous voulons donner aux algorithmes online un peu plus d'informations. Ces problèmes qui ont à disposition quelques connaissances en plus sur les tâches à traiter lorsqu'elles arrivent sont appelés des problèmes "semi-online".

Il y a différentes informations qu'un algorithme semi-online peut recevoir en avance : la taille de chacune des tâches, celle de la plus grande, la valeur de l'optimum, le nombre total de tâches, ...
Dans ce travail, nous allons d'abord nous intéresser au cas où la somme des tailles de chaque tâche est connue.

L'analyse de performance des algorithmes semi-online étant la même que pour les algorithmes online, nous n'allons pas en reparler ici.

2.1 Résultats importants

Avant d'aller plus loin, nous allons brièvement énoncer les quelques résultats connus sur l'ordonnancement online de n tâches sur m machines avec la somme des tailles des tâches connue :

Auteur(s)	Année	Nombre de machines	Borne supérieure	Borne inférieure
Keller & al.	1997	$m = 2$	$\frac{4}{3}$-compétitif	
Girlich, Kotov & Kovalev	1998	$\forall m$	$\frac{5}{3}$-compétitif[1]	
Angelelli & al.	2004	$\forall m$	$\frac{\sqrt{6}+1}{2} < 1,725$-compétitif	
Angelelli & al.	2004	$m \to \infty$		1,5656-compétitif
Angelelli & al.	2005	$m = 3$	1,5-compétitif	
Angelelli & al.	2005	$m = 3$		$\frac{1+\sqrt{129}-9}{6} > 1,3929$-compétitif
Angelelli & al.	2005	$m = 3$	1,4211-compétitif	

En observant ces résultats, nous pouvons nous rendre compte que l'ajout d'une simple information est suffisant pour améliorer considérablement la performance des algorithmes online. En effet, dans le cas de deux machines, nous passons d'une performance de $\frac{3}{2}$ pour l'algorithme online (avec Faigle & al., en 1989) à une performance de $\frac{4}{3}$ pour l'algorithme semi-online (avec Kellerer & al., en 1997). Et dans le cas d'un nombre quelconque de machines par exemple, le meilleur résultat online est donné par un algorithme 1,9201-compétitif (obtenu par Fleischer & Wahl, en 2000). Or en semi-online, on trouve un algorithme qui est $\frac{\sqrt{6}+1}{2} < 1,725$-compétitif (obtenu par Angelelli & al., en 2004).

2.2 Etude approfondie d'un article

Pour mieux comprendre le raisonnement qui se cache derrière le calcul de telles bornes et l'analyse de performance, nous allons consacrer cette section à l'étude un peu plus détaillée de l'article d'Angelelli, Speranza et Tuza (ANG05) qui est à la base de ce travail. En particulier, nous allons nous

[1] La preuve de ce résultat est incomplète dans la version originale. Et il semble qu'aucune version complète de la preuve n'existe jusqu'à aujourd'hui. Ce résultat est alors plutôt une conjecture.

intéresser à la démonstration de la borne inférieure qu'ils trouvent, à l'algorithme 1,5-compétitif qu'ils développent, ainsi qu'aux conditions sur les charges des machines qu'ils proposent pour améliorer encore un peu leur algorithme.
Il paraît peut-être inutile de s'attarder sur des preuves qui ont déjà été écrites, mais il nous semble intéressant de les faire une fois nous-même, premièrement pour vérifier qu'elles soient justes, et surtout pour les comprendre. Nous ne nous intéresserons pas réellement aux résultats, mais plutôt à la manière d'y parvenir. En effet, à la lecture d'autres articles, nous avons pu constater que le procédé de résolution de tels problèmes est souvent assez similaire.

Le sujet de cet article est la version semi-online du problème d'ordonnancement de tâches sur trois machines, dans le cas où la somme des tailles des tâches est connue.
Le cas de deux machines a déjà été étudié, et nous avons vu plus haut que le fait de connaître à l'avance la somme des grandeurs des tâches améliore la performance de $\frac{3}{2}$ à $\frac{4}{3}$.
Dans le cas de trois machines, ils trouvent une borne inférieure de $1+\frac{\sqrt{129}-9}{6} > 1.3929$ et ils proposent un algorithme H_x qui est 1,5-compétitif. Ils améliorent cette borne à $1 + \frac{8}{19} < 1,4211$ par une stratégie qui consiste à combiner l'algorithme H_x à un autre algorithme qui crée de bonnes conditions initiales pour H_x. Ils notent ce nouvel algorithme H_x^3. A nouveau, nous avons une meilleure performance que dans le cas online où le meilleur algorithme était l'algorithme de liste de Graham qui est $\frac{5}{3}$-compétitif.

2.2.1 Calcul de la borne inférieure pour la performance

Nous commençons par formuler le problème tel qu'ils l'utilisent dans le calcul de cette borne inférieure.
Soit trois machines P_1, P_2, P_3 disponibles pour exécuter un ensemble de n tâches dont les temps d'exécution de chacune d'entre elles est $p_1, p_2, ..., p_n$. Nous identifions chaque tâche i par sa grandeur p_i. Nous ne connaissons pas le nombre de tâches ni leur taille p_i individuelle. Par contre, la somme des tailles des tâches ($\sum p_i$) est connue et est posée égale à 3 dans ce problème. Les tâches arrivent une à une et doivent être aussitôt attribuées à une des trois machines.
Le but du problème est de minimiser le temps d'exécution de toutes les tâches.

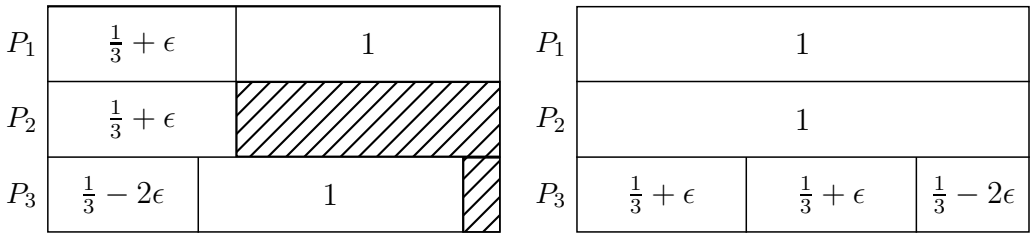
Théorème 2.2.1. *Aucun algorithme n'a un "ratio compétitif absolu" meilleur que* $1 + \frac{\sqrt{129}-9}{6} > 1,3929$.

Ce qui nous intéresse dans la preuve de ce théorème, c'est plutôt son déroule-§ment que son contenu arithmétique. C'est pourquoi, afin de mieux comprendre le mécanisme de celle-ci, nous pouvons l'écrire comme un jeu entre deux joueurs : l'algorithme H et son adversaire A.
A distribue une ou plusieurs tâches et H doit les attribuer aux machines. Pour gagner, A doit essayer de maximiser le temps total de réalisation de toutes les tâches, tandis que le but de H est de le minimiser.
Nous observons alors une stratégie pour le joueur A et nous montrons comment , et dans quelle mesure, elle empêche H d'ordonnancer les tâches de manière optimale.

Démonstration. A choisit $\epsilon \in \left(0, \frac{1}{6}\right)$.
Le jeu commence : le joueur A envoie deux tâches $p_1 = \frac{1}{3} + \epsilon$ et $p_2 = \frac{1}{3} + \epsilon$.

1. Soit le joueur H attribue p_1 et p_2 à deux différentes machines, alors le joueur A envoie deux tâches de taille 1 et une tâche de taille $\frac{1}{3} - \frac{2}{\epsilon}$ pour complèter le problème.

   Dans ce cas l'algorithme H doit charger en tout cas une machine à $\frac{4}{3} + \epsilon$, alors que l'optimum offline est égale à 1.

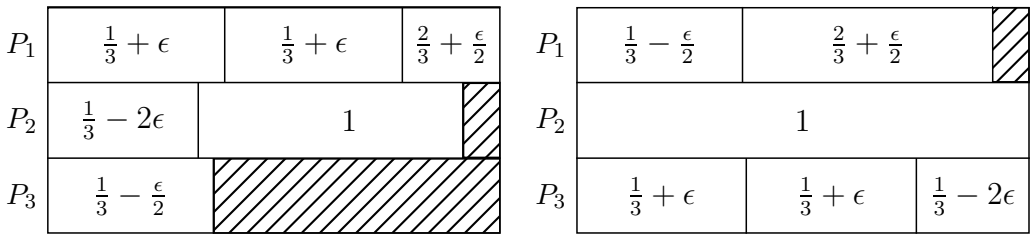


Algorithme H Algorithme offline

2. Soit le joueur H attribue p_1 et p_2 à la même machine, disons à P_1, alors A envoie une tâche $p_3 = \frac{2}{3} + \frac{\epsilon}{2}$.

   (a) Si H attribue p_3 à P_1, alors A envoie trois nouvelles tâches de taille $\frac{1}{3} - 2\epsilon, \frac{1}{3} - \frac{\epsilon}{2}, 1$.

   Ainsi l'algorithme sera forcé de charger une machine à plus de $\frac{4}{3} + \frac{5}{2}\epsilon$, alors que l'optimum offline est toujours à 1.

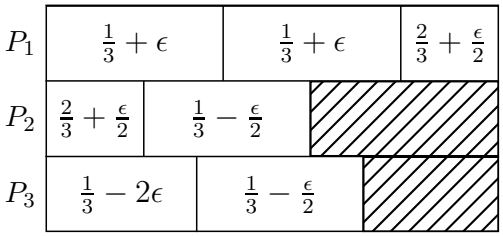


Algorithme H Algorithme offline

(b) Autrement, H attribue p_3 à P_2, et alors A envoie une quatrième tâche $p_4 = \frac{2}{3} + \frac{\epsilon}{2}$.

- H peut assigner p_4 à P_1 ou P_2. Dans ce cas A envoie une tâche de taille $\frac{1}{3} - 2\epsilon$ et deux autres de taille $\frac{1}{3} - \frac{\epsilon}{2}$.

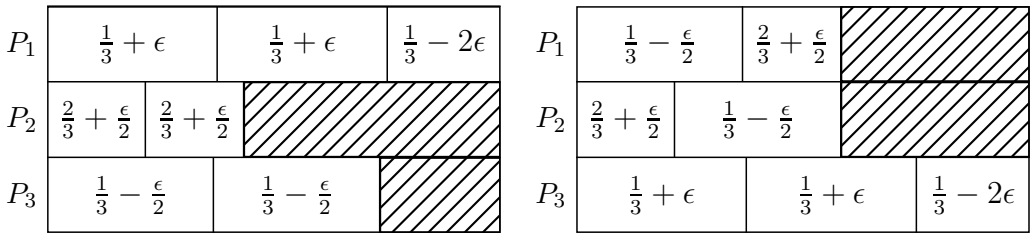
  - Dans le cas où p_4 est sur P_1, l'algorithme donne une solution qui est au minimum égale à $\frac{4}{3} + \frac{5\epsilon}{2}$.



Algorithme H

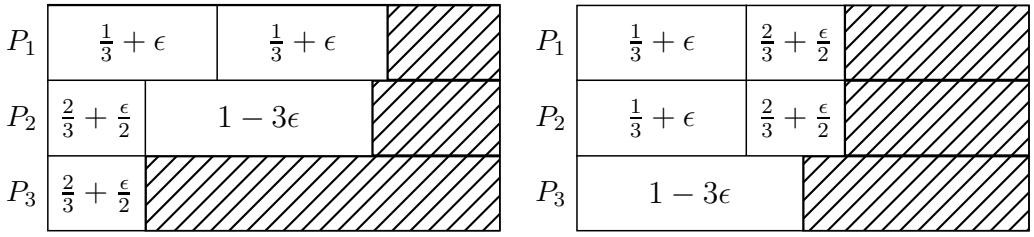
  - Et si H met p_4 sur P_2, alors la charge minimale est de $\frac{4}{3} + \epsilon$.

Alors que l'optimum offline vaut encore 1.



Algorithme H Algorithme offline

- Autrement, H attribue p_4 à la machine vide P_3. Et alors, le joueur A envoie une dernière tâche de taille $1 - 3\epsilon$. H est obligé de charger une machine à $\frac{5}{3} - \frac{5\epsilon}{2}$. Alors que l'optimum vaut dans ce cas $1 + \frac{3\epsilon}{2}$



Algorithme H Algorithme offline

En conclusion, l'algorithme H a essayé toutes les possibilités (tous les algorithmes possibles dans une telle situation ont été testés) pour gagner contre A, et le mieux qu'il puisse faire pour ce problème est :

$$\min\left(\frac{4}{3}+\epsilon, \frac{4}{3}+\frac{5\epsilon}{2}, \frac{\frac{5}{3}-\frac{5\epsilon}{2}}{1+\frac{3\epsilon}{2}}\right)$$

Ce minimum est atteint en $\epsilon = \frac{\sqrt{129}-11}{6}$ et vaut $1 + \frac{\sqrt{129}-9}{6} > 1,3929$.

□

Après cette preuve, nous pouvons peut-être mieux comprendre quel est le raisonnement à faire pour trouver une borne inférieure sur la performance de n'importe quel algorithme. L'idée est donc de trouver un seul exemple pour lequel aucun algorithme ne peut donner une solution plus proche de l'optimum qu'un certain nombre égal à $c \cdot OPT$ où c est la borne inférieure. Nous avons alors prouvé que le "ratio compétitif absolu" de n'importe quel algorithme ne pourra jamais être plus petit que cette borne inférieure c. Car sinon, il devrait l'être pour n'importe quelles données du problème et nous en avons trouvées pour lesquelles il ne l'était pas.

2.2.2 Analyse de performance

Nous allons maintenant nous attarder un peu sur l'analyse de performance d'un algorithme, en étudiant l'algorithme H_x qui est proposé dans cet article. L'algorithme H_x a le meilleur "ratio compétitif absolu" égal à 1,5 pour $x = \frac{1}{2}$. Cependant, il est améliorable, en posant quelques conditions, en un algorithme H_x^3 dont le "ratio compétitif absolu" est $(1 + x)$, $\forall x \geq \frac{8}{19}$.

Avant de commencer cette analyse, nous allons définir quelques notations qui nous faciliteront la compréhension pour la suite.

P_1, P_2, P_3 indiquent à tout moment aussi bien les machines que leur charge associée. Nous notons par T la somme du temps de réalisation des tâches restantes telle que l'équation $\sum P_i + T = 3$ soit toujours vraie. Nous utilisons la notation $\min P_i$ et $\max P_i$ comme abréviation de $\min(P_i|i = 1, 2, 3)$ et $\max(P_i|i = 1, 2, 3)$. Pour finir, nous écrirons H_x (resp. O) la valeur de l'algorithme évalué en une instance I (resp. la valeur de l'optimum offline évalué en I).

Algorithme H_x

Soient $x \in [0, 1]$ et p la tâche courante à assigner.

- $A = \{j|P_j + p \leq 1 + x\}$
- si $A \neq \emptyset$, alors mettre p sur la machine $\max(P_j|j \in A)$
- sinon mettre p sur la machine $\min(P_i)$

Théorème 2.2.2. *Le "ratio compétitif absolu" de H_x est*

$$R_{H_x} = \begin{cases} \frac{5}{3} & pour\, x \in \left[0, \frac{1}{3}\right) \\ \max(2 - x, 1 + x) & pour\, x \in \left[\frac{1}{3}, 1\right] \end{cases}$$

Démonstration. La preuve se fait en trois étapes.
Nous commençons par démontrer :

1) $\forall I$ et $\forall x \in [0, 1]$, $\frac{H_x}{O} \leq \max(2 - x, 1 + x)$

- Si $H_x \leq 1 + x$:
  on a
  $$\frac{H_x}{O} \leq H_x \quad (\text{car } O \leq H_x)$$
  et
  $$H_x \leq \max(2 - x, 1 + x)$$
  $$\Rightarrow \frac{H_x}{O} \leq \max(2 - x, 1 + x)$$

- Si $H_x > 1 + x$:
  Soit $\tilde{p}$ la tâche déterminant la valeur retournée par l'algorithme.
  On considère P_1, P_2, P_3 juste avant que $\tilde{p}$ y soit assignée.
  Comme $H_x > 1 + x$, $P_j + \tilde{p} > 1 + x, \forall j \Rightarrow A = \emptyset$
  $\Rightarrow \tilde{p}$ est attribuée à la machine $\min(P_i)$
  $\Rightarrow H_x = \min(P_i) + \tilde{p}$.

  * Si $\min(P_i) = 0$:
    La valeur optimale étant égale ou alors plus grande que la taille d'une seule tâche, on a $H_x = \tilde{p} \leq O$.
    Mais comme $O \leq H_x$, $H_x = O$
    $\Rightarrow \frac{H_x}{O} = 1 \leq \max(2 - x, 1 + x)$.

  * Si $\min(P_i) > 0$:

    a) $\sum P_i + \tilde{p} \leq 3 \Leftrightarrow \sum P_i \leq 3 - \tilde{p}$

    b) $\min(P_i) \leq \frac{\sum P_i}{3}$

    c) $\min(P_i) > 0 \Rightarrow$ toutes les machines ont au moins une tâche. Or une tâche p est affectée à une machine vide seulement si ajoutée aux autres machines elle engendre une charge plus grande que $1 + x$. Et donc quand on somme les charges de deux machines, forcément que l'une des deux contient la tâche p qui ajoutée à l'autre rend la charge plus grande que $1 + x$.
    $\Rightarrow P_i + P_j > 1 + x, \forall i, j = 1, 2, 3$ et $i \neq j$

    d) $P_1 + P_2 + P_1 + P_3 + P_2 + P_3 > 3(1 + x)$ (par le point c))

    $\Rightarrow 2(P_1 + P_2 + P_3) > 3(1 + x)$
    $\Rightarrow \sum_{i=1}^{3} P_i > \frac{3(1+x)}{2}$

    e) Par a), $\sum_{i=1}^{3} P_i < 3 - \tilde{p}$
    Par d), $\sum_{i=1}^{3} P_i > \frac{3(1+x)}{2}$

    $\Rightarrow \frac{3(1+x)}{2} < \sum_{i=1}^{3} P_i < 3 - \tilde{p}$
    $\Rightarrow \frac{3(1+x)}{2} < 3 - \tilde{p}$

$\Rightarrow \tilde{p} < 3 - \frac{3(1+x)}{2}$
$\Rightarrow \tilde{p} < \frac{3(1-x)}{2}$

On obtient alors :

$$\begin{aligned} \frac{H_x}{O} \leq H_x &= \min(P_i) + \tilde{p} \\ &< 1 - \frac{\tilde{p}}{3} + \tilde{p} \qquad (2.1) \\ &\leq 1 + \frac{2\tilde{p}}{3} \\ &< 1 + 1 - x \qquad (2.2) \\ &= 2 - x \\ &\leq \max(2 - x, 1 + x) \end{aligned}$$

L'inégalité (2.1) résulte d'une combinaison des points a) et b), tandis que l'inégalité (2.2) vient du point e).

Ensuite, nous raffinons l'intervalle de x , et nous démontrons :

2) $\forall I$ et $\forall x \in [0, \frac{1}{3})$, $\frac{H_x}{O} \leq \frac{5}{3}$

- Si $H_x \leq 1 + x$:
  $H_x \leq 1 + \frac{1}{3} = \frac{4}{3} < \frac{5}{3}$

- Si $H_x > 1 + x$:
  - $\min(P_i) = 0$:
    $\Rightarrow \frac{H_x}{O} \leq 1 < \frac{5}{3}$
  - $\min(P_i) > 0$:
    $H_x = \min(P_i) + \tilde{p} \leq 1 + \frac{2\tilde{p}}{3}$
    - Si $\tilde{p} \leq 1$:
      $\frac{H_x}{O} \leq H_x \leq 1 + \frac{2}{3} = \frac{5}{3}$
    - Si $\tilde{p} > 1$:
      on a aussi $O \geq \tilde{p}$
      $\Rightarrow \frac{H_x}{O} \leq \frac{1+\frac{2\tilde{p}}{3}}{\tilde{p}} < \frac{5}{3}$

3) Pour finir, il faut montrer que les bornes sont aussi serrées que possible.

Pour cela , nous devons trouver des exemples pour lesquels nous atteignons les bornes avec notre algorithme. Et il en existe effectivement, mais il n'est peut-être pas utile de les donner ici, car là n'est pas l'intérêt de ce travail. En effet, rappelons que ce ne sont pas les résultats eux-mêmes qui nous intéressent, mais plutôt le chemin pour y parvenir.

□

Nous pouvons déjà constater que, partant d'un algorithme assez simple, nous obtenons un bon résultat pour des $x \in [0, 1]$. A partir de là, nous essayons d'observer ce qui se passe lorsque x prend les différentes valeurs de l'intervalle. Nous remarquons alors que l'algorithme H_x est $(1 + x)$-compétitif seulement pour des $x \geq \frac{1}{2}$. Il faut donc travailler encore cet algorthime de manière à pouvoir garantir un "ratio compétitif absolu" pas plus grand que $1 + x$, pour des $x < \frac{1}{2}$. Et c'est en spécifiant quelques conditions sur les charges des machines que nous allons pouvoir garantir cette performance, même pour des $x < \frac{1}{2}$.
Nous laisserons le lecteur intéressé se référer à l'article d'Angelelli, Speranza et Tuza (ANG05) pour voir les énoncés et les démonstrations des six conditions qui améliorent la performance de l'algorithme.

A partir de ces résultats, nous construisons un algorithme qui crée ces bonnes conditions initiales et combiné avec l'algorithme H_x, cela donne l'algorithme H_x^3.

Algorithme H_x^3

- si $(P_1 + p \leq x)$, alors $p \rightarrow P_1$
- sinon si $\left(P_2 + p < \frac{3(1-x)}{2}\right)$, alors $p \rightarrow P_2$
- sinon si $(P_2 + p \leq 1 + x)$, alors $p \rightarrow P_2$ et appliquer H_x
- sinon si $(P_1 + P_2 + p \geq 2 - x)$, alors $p \rightarrow P_1$ et appliquer H_x
- sinon si $(P_3 = 0)$, alors $p \rightarrow P_3$
- sinon si $(P_3 + p \leq 1 + x)$, alors $p \rightarrow P_3$ et appliquer H_x
- sinon $p \rightarrow P_1$ et appliquer H_x

Cet algorithme est $(1 + x)$-compétitif , $\forall x \geq \frac{8}{19}$.
A nouveau, ceux que la preuve intéresse peuvent se référer à l'article d'Angelelli, Speranza et Tuza (ANG05).

Pour conclure cette section, nous pouvons noter que la recherche de bornes supérieures pour la performance d'algorithme se fait par construction. Il faut donc avoir déjà une bonne idée d'où nous voulons aller, pour parvenir à un quelconque résultat intéressant.

2.3 Différentes informations partielles

Nous pouvons trouver dans la littérature d'autres informations partielles sur notre problème qui améliorent la performance des algorithmes online.

Un des premiers cas étudié est celui où l'information supplémentaire concerne l'ordre d'arrivée des tâches. En effet, Liu , Sidney et Van Vliet (LIU) développent le problème où un nombre connu de tâches arrive dans un ordre non-croissant. Ce qui revient à dire, sans perte de généralité, que pour des tâches de tailles p_i, on a $p_1 \geq p_2 \geq ... \geq p_n$. Ils obtiennent une borne supérieure qui vaut $1 + (m-1)\frac{1}{m+\lceil \frac{m}{2} \rceil}, \forall m$. Dans le cas $m = 2$, $m = 3$ et $m = 4$, ils construisent trois algorithmes ordinaux dont la performance compétitive vaut $\frac{4}{3}$, $\frac{7}{5}$ et $\frac{101}{70}$ respectivement. Pour $m = 2$ et $m = 3$, c'est aussi une borne inférieure. Pour finir, ils prouvent une borne inférieure égale à $\frac{23}{16}$ pour $m = 4$.

S. Seiden, J. Sgall et G. Woeginger (SEI) présentent un problème assez similaire. Ils utilisent aussi l'information sur l'arrivée décroissante des tâches, mais ils ne connaissent par contre pas le nombre de tâches à ordonnancer. Ils trouvent une borne inférieure de $\frac{7}{6}$ pour $m = 2$ et de $\frac{1}{6}(1 + \sqrt{37})$ pour $m = 3$. Or, Graham a montré que pour des tâches de tailles décroissantes, l'algorithme de liste est $(\frac{4}{3} - \frac{1}{3m})$-compétitif. Par conséquent, pour $m = 2$, l'algorithme de liste est optimal. Pour $m \geq 3$, le problème reste ouvert.

Nous pouvons aussi aborder le sujet en passant par le problème de partition. Car en général le problème de partition d'objets en m ensembles est équivalent au problème classique d'ordonnancement de tâches sur m machines. Ainsi les résultats obtenus par Kellerer et al.(KEL) dans le développement d'algorithmes semi-online pour le problème de partition en deux ensembles sont aussi valables pour le problème d'ordonnancement sur deux machines. Ils étudient trois versions semi-online du problème. Dans la première, ils supposent avoir une "boîte" de longueur k pour stocker les tâches. Lorsqu'une tâche arrive, elle peut être mise en attente dans cette "boîte" tant que cette dernière n'est pas pleine. Et si elle est pleine, la tâche peut soit être assignée immédiatement à une machine, soit stockée dans la boîte, mais à la place d'une autre tâche qui doit aussitôt être placée sur une machine.

Dans la deuxième version, les tâches sont affectées à une machine lorsqu'elles arrivent, mais deux ensembles parallèles de machines sont disponibles pour la résolution du problème. Chaque tâche est dédoublée et envoyée sur une machine dans chacun des deux ensembles avant l'arrivée de la prochaine tâche. Finallement, la meilleure des deux solutions est gardée. La dernière version est celle où la somme des tâches est connue. Pour ces trois problèmes, ils prouvent que le "ratio compétitif absolu" est de $\frac{4}{3}$.

G. Zhang (ZHA) analyse aussi le problème semi-online où une "boîte" de longueur k capable de stocker k tâches est disponible. Il trouve un algorithme qui est $\frac{4}{3}$-compétitif, et cette performance est même atteinte avec une "boîte" de longueur 1. Et nous avons vu ci-dessus que Kellerer et al. ont montré qu'il n'existait pas d'algorithme pour ce problème avec une performance meilleure que $\frac{4}{3}$. Ainsi l'algorithme de Zhang est optimal pour $m = 2$.

B. Coleman et W. Mao (COL) développent un modèle qui différe de ce qui a déjà été fait par la connaissance de certaines tâches qui arriveront dans un futur proche. De plus, ils mettent à disposition une "boîte" qui peut stocker une certaine quantité de tâches, comme ce qui a été fait par Kellerer et al., ou encore par Zhang. L'algorithme qu'ils construisent est $\frac{2}{(k+1)(k+2)}\left(\frac{n}{m} + \frac{1}{2}k(k+1)\right)$-compétitif.

Une autre version du problème online sur deux machines identiques a été étudié par Z. Tan et Y. He (TAN). Il ne considère pas seulement une information, mais ils combinent deux informations en une. Ils donnent deux algorithmes optimaux pour le cas où la somme totale des tailles des tâches et la taille de la plus grande tâches sont connues à l'avance, ainsi que pour le cas où les deux informations combinées sont la somme totale des tailles des tâches et l'ordre décroissant de leur arrivée. La performance de ces algorithmes est de $\frac{6}{5}$, respectivement de $\frac{10}{9}$.

E. Angelleli, M. G. Speranza et Zs. Tuza (ANG03) ont aussi travaillé avec deux informations combinées. Ils traitent le problème de l'ordonnancement de tâches sur deux machines, avec la somme des tailles des tâches connue et une borne supérieure γ sur la taille des tâches. Ils prouvent qu'une performance de $\frac{4}{3}$ peut être obtenue pour n'importe quelle valeur de $\gamma < 1$. Ils présentent différentes bornes inférieures, pour différentes valeurs de γ. Ils obtiennent même des algortihmes optimaux dans les cas où $\frac{1}{2} \leq \gamma \leq \frac{3}{5}$, $\gamma = \frac{3}{4}$ et $\frac{16}{17} \leq \gamma < 1$, de performance compétitive égale à $\frac{6}{5}$, resp. $1+\frac{\gamma}{3}$ et resp. $\frac{2}{3}(1+\gamma)$.

Pour terminer, nous pouvons citer les travaux de Y. He et G. Dósa (HE)

sur l'ordonnancement sur trois machines identiques de tâches dont la taille se touve dans l'intervalle $[p, rp]$, où $p > 0$ et $r \geq 1$. Ils montrent que l'algorithme de liste de Graham est optimal pour les $r \in [1, 1.5], [\sqrt{3}, 2]$ et $[6, +\infty)$. Et ils présentent un algorithme optimal dont "le ratio compétitif" vaut $\frac{3}{2}$ pour des $r \in (2, 2.5]$.

Nous pouvons certainement trouver encore d'autres informations partielles qui pourraient améliorer la performance des algorithmes online, comme par exemple la connaissance de la fonction de distribution des tâches, ou la valeur de la solution du problème optimal offline, ou encore d'autres combinaisons d'informations telles que quand la dernière tâche arrive, nous savons que c'est la dernière et en plus la plus grande... . Cependant, nous avons déjà là un joli tableau de ce qui peut être fait en ordonnancement semi-online pour le problème d'assigner sur des machines identiques des tâches, qui doivent être exécutées sans interruption.

2.4 Le cas préemptif

Dans le problème étudié jusqu'à présent, nous nous intéressions au cas où les tâches devaient être exécutées sans interruption. Mais que se passe-t-il quand celles-ci peuvent être stoppées au cours de leur réalisation et terminées plus tard, peut-être sur une autre machine ? C'est la question à laquelle on essaye de répondre lorsqu'on étudie les problèmes préemptifs. Ce sont donc les problèmes dans lesquels les tâches peuvent être exécutées par morceaux, mais un morceau à la fois, c'est-à-dire, que deux morceaux d'une même tâche ne peuvent pas être exécutés sur deux différentes machines en même temps. Dans la littérature, nous n'avons rien trouvé sur le problème d'ordonnancer des tâches sur des machines identiques avec la somme des tailles des tâches connues et avec préemption. Le seul article trouvé sur le sujet de l'ordonnancement semi-online sur des machines identiques a été écrit par S. Seiden, J. Sgall et G. Woeginger (SEI) et traite le cas où les tâches arrivent dans un ordre décroissant. Ils développent un algorithme pour ce problème particulier dont la compétitivité tend vers $\frac{1}{2}(1 + \sqrt{3})$ quand m devient très grand. Peut-être bien qu'il existe d'autres études sur le sujet, mais nous pouvons quand même constater que c'est un domaine peu approfondi.
Par contre, la cas préemptif a été un peu plus étudié pour les problèmes où les machines ne sont pas identiques, c'est-à-dire avec des vitesses d'exécution différentes. Cependant le cas semi-online n'a eu guère de succès. En effet nous pouvons lire très peu d'articles sur le sujet.

Nous nous sommes alors demandés, étant donné que cela n'a pas encore été fait bien que la préemption soit connue, si vraiment il serait intéressant et utile de pourvoir couper les tâches dans notre problème. Nous pouvons faire plusieurs remarques à ce sujet.
Etant donné que toutes les machines sont les mêmes, il n'y a pas de réels avantages, si l'on considère la machine elle-même et non sa charge, à mettre une tâche sur une machine plutôt que sur une autre. Par contre, pour les problèmes où les machines n'ont pas la même vitesse d'exécution, il serait plus intéressant de mettre les tâches les plus grandes sur les machines qui tournent le plus vite. C'est un point intéressant de la préemption que nous n'avons pas dans notre problème.
De plus, les morceaux d'une même tâche doivent être réalisés sur les intervalles de temps disjoints. Nous ne pouvons donc pas, par exemple, couper la dernière tâche et la répartir sur les machines qui sont vides pour gagner du temps. Donc, en les coupant, tout ce que nous obtenons c'est une information sur des morceaux de tâches que nous allons devoir ordonnancer à un moment donné. Cela revient donc plus au moins, à notre avis, à considérer le cas semi-online où une "boîte" d'une certaine longueur est à disposition pour stocker les tâches. Nous coupons les tâches, nous les mettons de côté en attendant de voir ce qui va arriver, et ensuite par rapport à ce qui arrive, nous essayons d'ordonnancer le reste de manière optimale. Nous avons essayé d'appliquer manuellement la préemption sur l'exemple donné dans la preuve de la borne inférieure de l'article de E. Angelleli, M. G. Speranza et Zs. Tuza (ANG05) et le résultat n'est pas très concluant.
Il n'est donc pas très intéressant, à priori, d'étudier le cas de la préemption dans notre problème. Certainement, que le résultat serait un peu meilleur, mais les algorithmes pour y parvenir sont bien plus compliqués.

Conclusion

Ce travail nous a permis de faire un joli survol de l'ordonnancement online. Nous avons appris ce qu'est l'ordonnancement (semi-)online, quel genre de problèmes y sont traités. Nous avons essayé de mettre en avant les points importants des techniques d'évaluation de la performance des algorithmes (semi-)online et de les retrouver dans l'article de E. Angelleli, M. G. Speranza et Zs. Tuza (ANG05) en étudiant un peu plus en détails les preuves de bornes inférieures et supérieures qui y sont développées.
Le problème traité plus particulièrement dans ce travail est celui d'ordonnancer des tâches , sans préemption, sur des machines identiques avec la somme des tailles des tâches connue. Nous avons constaté, à l'aide d'une petite revue des résultats online et semi-online sur le sujet, que l'ajout d'informations améliore passablement la performance des algorithmes. Et cela est vrai même en pratique. Il est donc très utile de s'y intéresser et d'essayer d'améliorer encore la performance des algorithmes (semi-)online. Nous avons aussi vu les différentes informations qui ont été ajoutées jusqu'à présent dans le cas semi-online et celles qui reviennent le plus fréquemment sont les informations sur la grandeur des tâches et celles où est mise à disposition une "boîte" de stockage. Nous nous sommes interrogés sur la résolution du même problème, mais avec préemption. Et nous arrivons à la conclusion que, dans notre problème, permettre de couper les tâches n'est pas très utile.
Par contre, une situation qui n'a pas été encore étudiée pour notre problème est celle où une fois une tâche mise sur une machine, elle peut être retirée et relancée plus tard, sur la même machine ou une autre, mais depuis le début. Ainsi, pour qu'une tâche se termine, elle doit être assignée à la même machine pendant tout son temps de réalisation sans interruption. Cela pourrait donc être le sujet de futurs travaux.

Bibliographie

[ALB99] S. Albers (1999). *Better Bounds for Online Scheduling*. SIAM Journal of Computer, 29(2) : 459-473 .

[ALBS99] S. Albers et S. Leonardi (1999).*Online Algorithms*. ACM Computing Surveys (CSUR), 31(3), article no 4.

[ALB02] S. Albers et B. Schröder (2002). *An Experimental Study of Online Scheduling*. ACM Journal of Experimental Algorithms, 7(3).

[ANG03] E. Angelleli, M. G. Speranza et Zs. Tuza (2003). *Semi On-line Scheduling on two Parallel Processors with an Upper Bound on the Items*. Algorithmica ; 37 : 243-262.

[ANG04] E. Angelleli, A. B. Nagy, M. G. Speranza et Zs. Tuza (2004). *The On-line Multiprocessor Scheduling Problem with Known Sum of the Task*. Journal fo Scheduling, 7 : 421-428.

[ANG05] E. Angelleli, M. G. Speranza et Zs. Tuza (2005). *Semi On-line Scheduling on three Processors with Known Sum of the Tasks*. Preprint

[BAR] Y. Bartal, A. Fiat, H. Karloff et R. Vohra (1995). *New Algorithms for an Ancient Scheduling Problem*. Journal of Computer and System Sciences, 51 : 359-366.

[CAR] J. Carlier et P. Chrétienne (1988). *Problèmes d'Ordonnancemment : Modélisation, Complexité, Algorithmes*. Masson, Paris.

[COL] B. Coleman et W. Mao (2002). *Lookahead Scheduling in a Real-time Context*. Computer Science and Informatics, 205-209.

[HE] Y. He et G. Dósa (2005). *Semi-online Scheduling Jobs with Tightly-grouped Processing Times on three Identical Machines*. Discrete Applied Mathematics, 159 : 140-159.

[KEL] H. Kellerer, V. Kotov, M. G. Speranza et Z. Tuza (1997). *Semi On-line Algorithms for the Partition Problem*. Operations Research Letters, 21 : 235-242.

[LEU] J. Y.-T. Leung (2004). *Handbook of Scheduling : Algorithms, Models and Performance Analysis*. CRC Press LLC.

[LIU] W.-P. Liu, J. B. Sidney, A. van Vliet (1995). *Ordinal algorithms for parallel machine scheduling*. Operations Research Letters, 18 : 223-232.

[RUD] J. F. Rudin II et R. Chandrasekaran (2003). *Improved Bounds for the Online Scheduling Problem*. SIAM Journal of Computer, 32(3) : 717-735 .

[SEI] S. Seiden, J. Sgall et G. Woeginger (2000). *Semi-online Scheduling with Decreasing Job Sizes*. Operations Research Letters, 27(5) : 215-221.

[SLE] D. D. Sleator et R. E. Tarjan (1985). *Ammortized Efficiency of List Update and Paging Rules*. Communications of the ACM, 28(2).

[TAN] Z. Tan et Y. He (2000). *Semi-on-line Problems on two Identical Machines with Combined Partial Information* . Operations Research Letters, 30 : 408-414.

[ZHA] G. Zhang (1997). *A Simple Semi On-line Algorithm for* $P2//C_{\max}$ *with a Buffer*. Information Processing Letters, 61 : 145-148.