Ivan Dromigny--Chevreuil
Romain Bernard

Rapport

Projet de Complexité et Algorithmes

Le problème Min Makespan

Table des matières

I. Exercice

1. Résultat de LPT sur l'instance 3.2

M 1	d2 = 7	d8 = 3	d5 = 2	d3
M 2	d6 = 6	d11 = 5		d7 = 2
M 3	d9 = 6	d4 = 3	d1 = 2	d10 = 2

$T_{LPT} = 13$

2. Ratio d'approximation de LPT (I)

$$R_a = Max(\frac{T_{opt}(I)}{T_{LPT}(I)}, \frac{T_{LPT}(I)}{T_{opt}(I)})$$

$$= Max(\frac{13}{16}, \frac{16}{13}) = 1.23$$

3. Preuve de $T_{LPT}(I) \leq \frac{\sum_{k \neq j} d'_k}{m} + d'_j$

Dans l'algorithme LPT, le cas qui maximise le résultat obtenu est si l'ensemble des tâches hormis la dernière *j* donnent un temps d'exécution équivalent sur chaque machine (ce qui correspond à la fraction de l'inéquation à prouver). On ajoute alors dans ce cas la dernière tâche *j* au résultat, donnant le temps d'exécution final. Ce type de résultat d'instance étant le pire par conception, on peut en déduire que le temps obtenu par l'algorithme LPT sera supérieur ou égal au temps donné par l'équation représentant ce type d'instances.

4. Optimalité de LPT si $n \leq m$

Dans le cas où le nombre *m* de machines est supérieur au nombre *n* de tâches, l'algorithme LPT va placer une tâche par machine jusqu'à ne plus avoir de tâche, ce qui correspond à une solution optimale, car on obtient alors comme résultat le temps mis par la plus grande tâche. Nos tâches n'étant pas sécables, c'est une des meilleurs solutions (des équivalences sont possibles, mais rien de mieux).

5. Montrer que $T_{opt}(I) \geq 2d'_{m+1}$ dans le cas où $n \geq m+1$

Sous condition que *n* soit plus grand que le nombre *m* de machines, sachant que $d'_1 \geq \ldots \geq d'_{m+1} \geq \ldots \geq d'_n$, les tâches étant en ordre décroissant, on sait que $d'_m + d'_{m+1}$ donnera au mieux $2d'_{m+1}$, comme on se trouve dans le cas où le nombre de machines est inférieur au nombre de tâches, on devra ajouter (au moins) une tâche à une machine en contenant déjà une, et les valeurs des tâches étant décroissantes, dans le meilleur cas, toute les tâches avant d'_{m+1} vaudront la même valeur que ce dernier, et au moment de l'ajout à la machine, on est alors dans le cas $d'_m + d'_{m+1}$. Comme ce cas est le meilleur et qu'on a vu précédemment que $d'_m + d'_{m+1}$ donne au mieux $2d'_{m+1}$, T_{opt} vaudra toujours au moins $2d'_{m+1}$.

6. L'exécution de LPT se résume aux cas (a) $T_{LPT}(I) = T_{opt}(I)$ et (b) $j \geq m+1$

Les deux cas où $T_{LPT}(I) = T_{opt}(I)$ (à coup sûr) sont soit dans le cas où le nombre de tâches est inférieur au nombre de machines, comme on l'a vu dans la question 4, T_{LPT} est alors optimal, soit si une tâche est assez grande pour être plus grande que le remplissage des autres machines par les tâches restantes, le résultat optimal et obtenu par LPT sont alors les mêmes. Ces deux cas de figure, sont ceux possibles quand on a $j < m+1$, ne laissant alors que les cas où $j \geq m+1$, soit le cas (b), où l'on ne dispose pas de propriété permettant de définir d'autre sous cas. Une exécution de LPT répond donc soit au cas (a), soit au cas (b).

7. Montrer que $T_{LPT}(I) \leq r \cdot T_{opt}(I)$ où *r* est une constante

Voici la suite d'inéquation représentant le chemin de pensée utilisé pour cette preuve et qui va être expliqué par étapes :

$$T_{LPT}(I) \leq \frac{\sum_{k \neq j} d'_k}{m} + d'_j \leq \frac{\sum_{k \neq m+1} d'_k}{m} + d'_{m+1} \leq T_{opt}(I) + d'_{m+1} \leq \frac{3}{2} T_{opt}(I)$$

En utilisant la preuve en question 6, on va réduire l'intervalle des possibles de notre problème. On sait que soit $T_{LPT}(I) = T_{opt}(I)$, soit $j \geq m+1$. Le ratio dans le premier cas est évidemment de 1, mais c'est la seconde partie qui va nous intéresser pour démontrer le ratio d'approximation, on se place donc dans ce cadre pour la suite de la preuve.

Dans un premier temps, on part de la partie gauche de l'équation à prouver, et on applique l'inéquation prouvée à la question 3 pour avancer dans notre raisonnement vers T_{opt}.

Ensuite, j étant le dernier numéro de tâche affectée, et la liste des tâches étant décroissante et comme on est dans le cas où $j \geq m+1$, on peut en déduire que d'_j sera inférieur ou égal à d'_{m+1}, permettant la deuxième inéquation.

Maintenant qu'on a posé d'_{m+1}, on modifie la fraction et la remplace par T_{opt}, effectivement, ce dernier valant au mieux $\frac{\sum_i d'_i}{m}$ comme vu dans la question 2 de l'exercice 3.3 de TD, la partie gauche de cette troisième inéquation est donc inférieure à $T_{opt}(I) + d'_{m+1}$.

Enfin, d'après la question 5, on a $T_{opt}(I) \geq 2d'_{m+1}$, on peut donc adapter cette inéquation et rassembler les valeurs de $T_{opt}(I)$ ainsi obtenues, permettant ainsi d'obtenir un ratio r de 1,5

8. Conclure quant à l'approximabilité de LPT

D'après le ratio trouvé en question 7, LPT est 1,5-approximable, son ratio étant constant, on peut en conclure que le problème de Min Makespan fait partie de la classe de problèmes APX

II. Projet de programmation

A. Présentation du programme

Le programme créé a été implémenté en C++. Connaissant bien ce langage de programmation, du fait de sont apprentissage lors de nos années à l'université, il nous a semblé évident de le choisir comme langage d'implémentation.

Ce programme est composé d'un unique fichier .cpp qu'il faut compilé et exécuter au moyen de la ligne de commande suivante (en étant placé dans le dossier contenant tous le projet) :

```
g++ -std=c++0x -Ofast -W -Wall -Wextra -pedantic -Wno-sign-compare -Wunused-result -Wno-unused-parameter Programme/MinMakespan.cpp -o Programme/MinMakespan && ./Programme/MinMakespan
```

Ce programme étant *user friendly* il est donc possible d'interagir avec celui-ci au moyen des chiffres du clavier. Nous ne voulions pas passer trop de temps à la création d'une interface visuelle « développée ». Il nous a semblé plus important d'avoir un programme ayant un affichage simple mais clair, concis et avec une gestion des erreurs (si l'utilisateur essaye de rentrer des informations erronées par exemple).

Par ailleurs, ce programme est composé de :

- Trois fonctions permettant de choisir/définir les instances à fournir aux algorithmes d'approximation ;
- Six fonctions utilitaires (récupérer la borne inférieur moyenne par exemple) ;
- Les trois fonctions qui implémentent les trois algorithmes demandé ;
- La fonction principale permettant de faire fonctionner le programme de manière *user friendly.* Dans cette fonction principale nous avons ajouté plusieurs variables utilisées par les algorithmes :
  - Deux entiers : l'un pour le nombre de machines de l'instance et l'autre pour le nombre de taches de l'instance (nbMachines et nbTaches) ;
  - Un vector d'entier étant le le temps de chaque taches (tempsTaches) ;
  - Un vector de vector d'entier utilisé pour la génération aléatoire des instances (tempsTachesRand).

B. Présentation des algorithmes implémentés

Chacune des fonctions implémentant les algorithmes prennent en paramètre : le nombre de machines, le nombre de taches et la liste de valeur des taches (*nbM*, *nbT* et *tempsT* respectivement).

Chacune des fonction retourne le total maximum de temps attribué aux machines.

a. LSA (List Scheduling Algorithm)

Afin d'implémenter cet algorithme vue en cours, nous avons utilisé un vector de pair, ces pair étant constituées d'un entier et d'un vector d'entier (*taches_par_machines*). Le premier entier correspond au temps total des taches attribuées à la machine et le vector d'entier est la liste des taches qui sont attribuées à la machine.

En récupérant les tâches présentes dans *tempsT*, on les attribues aux machines (*taches_par_machines*) selon l'algorithme vu en cours.

b. LPT (Largest Processing Time)

On part sur le même principe du vector de pair que dans l'algorithme précédant (*taches_par_machines*), mais on rajoute un vector de pair constitué de deux entier, le premier étant l'index de la tache et le second son temps de réalisation.

On tri ensuite ce vector par ordre décroissant, et on l'utilise (au lieu de *tempsT* comme dans LSA) afin d'affecter les différentes taches aux machines.

c. MyAlgo

On utilise également le vector de pair (*taches_par_machines*) et on ajoute cette fois là deux vector de pair constitués de deux entier (comme celui de LPT). L'un va être trié par ordre décroissant et l'autre par ordre croissant. On met en place également un booléen (*same*).

Ensuite, tant que toutes les taches n'ont pas été attribuées, vérification au moyen de deux compteur que l'on additionne (un pour chaque vector de pair) :

- si *same* est à faux : on récupère la machine ayant le temps total le plus faible et on lui attribue la valeur suivante dans le tableau décroissant (en fonction du premier compteur que l'on incrémente de un par la suite) et on passe *same* à vrai ;
- si *same* est à vrai : on attribue à la même machine récupéré précédemment la valeur suivante dans le tableau croissant (en fonction du second compteur que l'on incrémente de un par la suite) et on passe *same* à faux.

Cela permet donc d'attribuer à chaque machine une tache ayant un temps de réalisation important et une tache ayant un temps de réalisation faible, et permet donc d'équilibrer les temps affectés à chaque machine.

Cela induit que pour que l'algorithme soit efficace, celui-ci doit être applique sur des instances de type : pour *nbM*, le nombre de machines et *nbT*, le nombre de taches il faut que nous ayons :

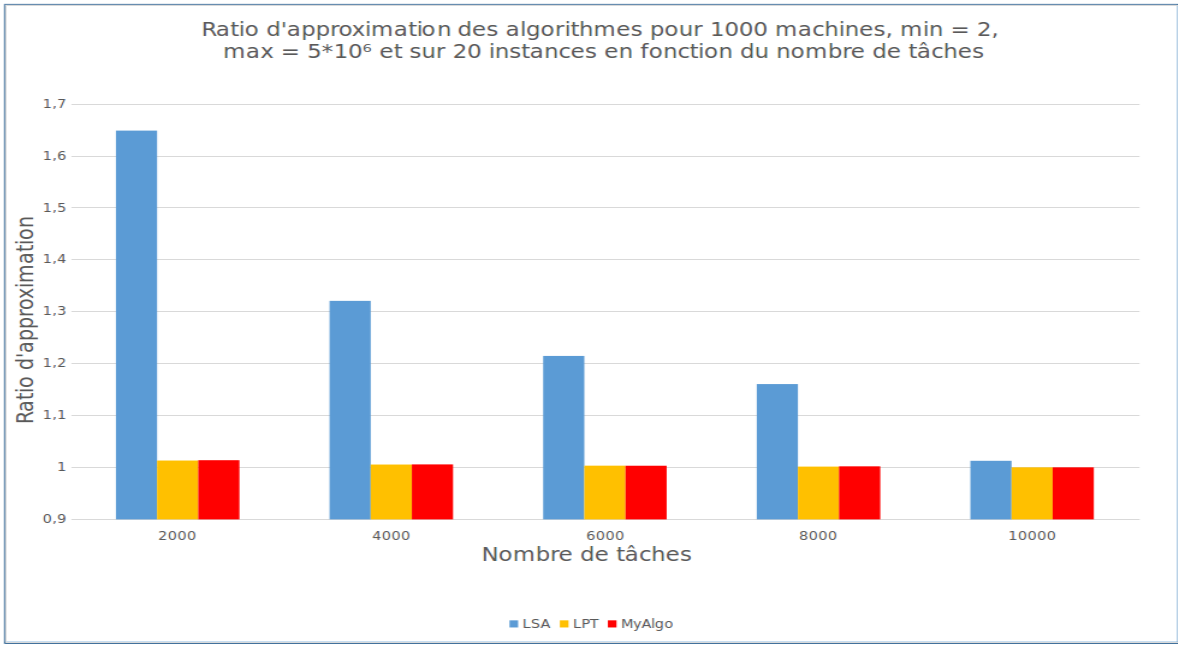
nbT = nbM*2n, avec n ≥ 1

En ce qui concerne la complexité de l'algorithme, on met tous les éléments du vector *tempsT* dans les deux vector de pair, ce qui correspond à une complexité en O(2n) donc O(n). On tri ensuite ces deux vector, ce qui correspond à une complexité en O(2(n*log(n)) donc O(n*log(n)). Pour finir, on ajoute les éléments des vector dans le vector *taches_par_machines,* ce qui correspond à une complexité en O(n).

On a donc une complexité totale qui est ici en O(n). Cet algorithme est donc bien logarithmique comme attendu.

C. Analyse détaillée des algorithmes

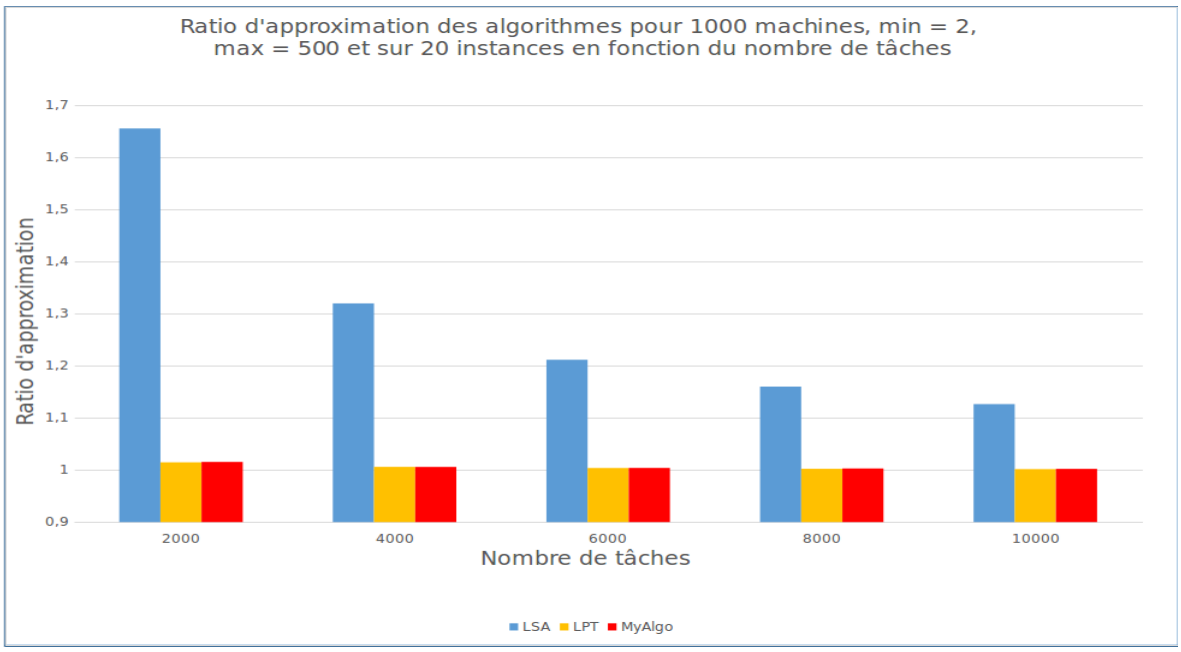
Dans la partie suivante nous allons au moyen de diagrammes montrer l'efficacité des différents algorithmes en fonction des instances.



Pour les jeux de test effectué sur ces instances, nous avons bien $nbT = nbM*2n$, où $nbM = 1000$, $nbT = \{2000,4000,6000,8000,10\ 000\}$ et $n = \{1,2,3,4,5\}$.

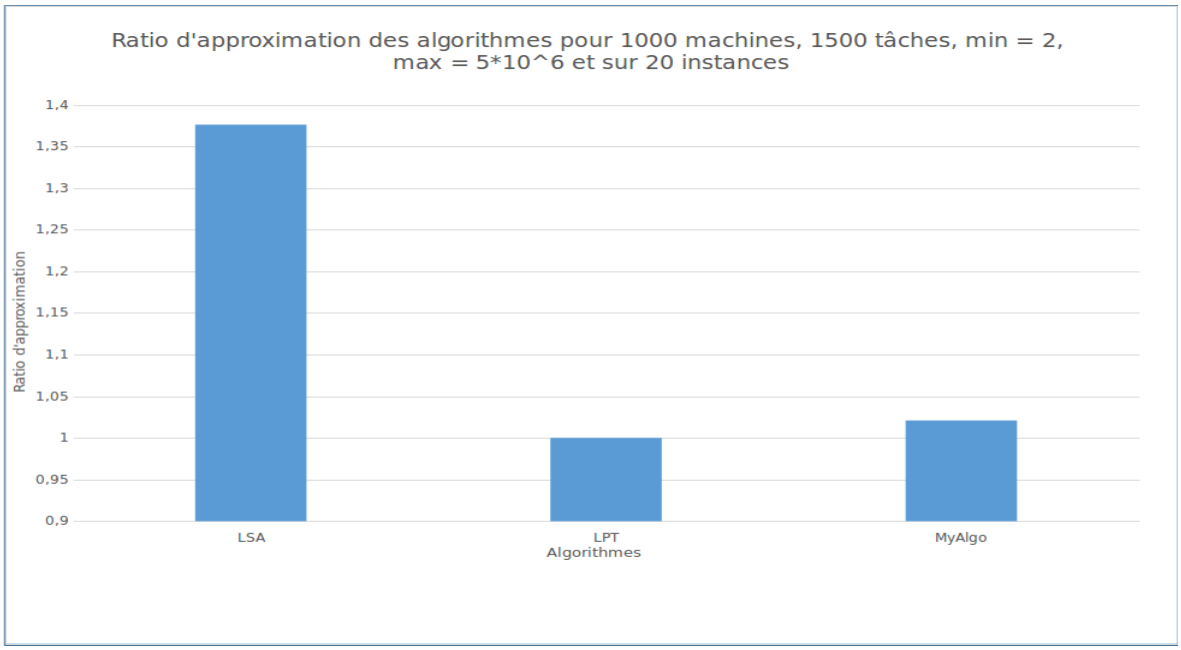
LSA est, comme l'on peut s'y attendre, peu efficace quand le nombre de machines est faible vis à vis du nombre de tâches. Plus la différence entre le nombre de tâches et le nombre de machines est grand est plus LSA est efficace.

On remarque toutefois que MyAlgo est aussi efficace que LPT quelque soit le nombre de machines et le nombre de tâches.



Pour ces instances, la différence entre le minimum et le maximum est beaucoup plus petit que pour les instances précédentes. On remarquera que cela n'affecte en rien l'efficacité de MyAlgo vis à vis de LPT.

Des tests ont été également effectués avec *min* = 2 et *max* = 10 et les résultats sont presque identiques, nous n'avons donc pas voulu les ajouter car apportant peu d'intérêt.



Dans le cas présent, on remarque que même si *nbT* = *nbM**2n est incorrect (car *n* doit être un entier), le résultat obtenu reste très appréciable pour MyAlgo. Cela vient du fait que la fonction mettant en pair une tâche avec un grand poids avec une ayant un poids très faible, et du fait que *nbT* <

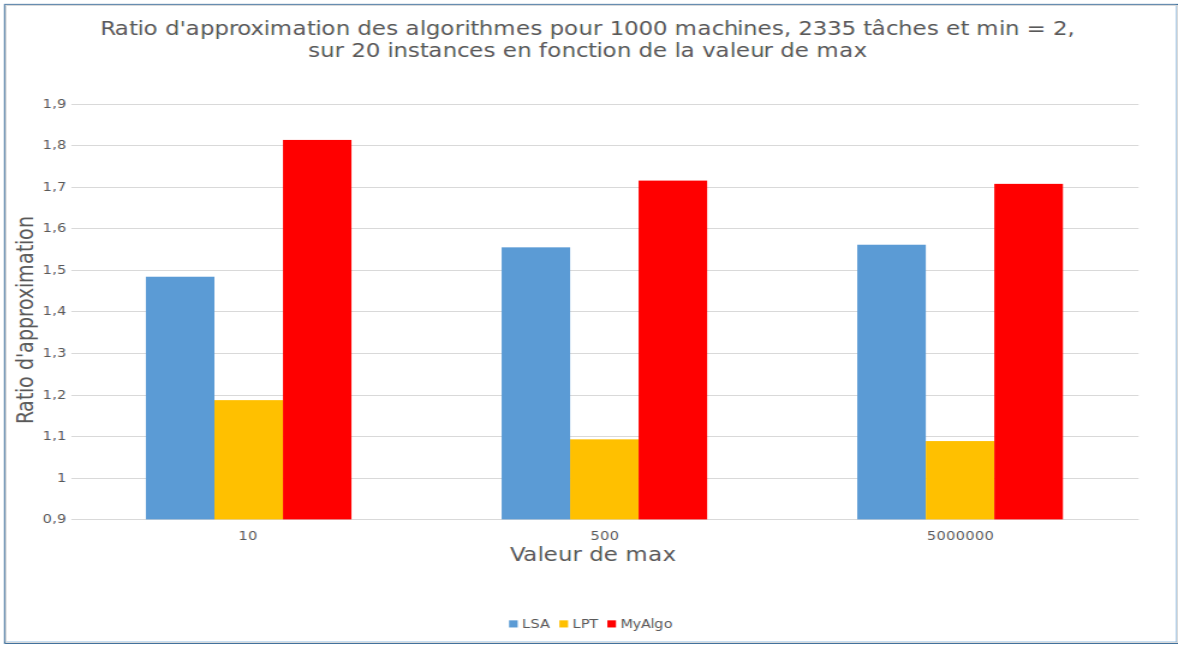
*nbM**2. En effet, nous avons ici des machines ne possédant aucune taches qui leur sont affecté, mais l'ensemble des tâches affecté aux machines reste dans la moyenne (car affecté par pair).

On remarquera toutefois, que moins l'écart entre min et max est important et moins cela est valable.

Exemple : machines = 7, tâches = 10, min = 1, max = 10 (2,3,3,4,5,6,7,8,10,10)

M 1	d10 = 10	d1 = 2
M 2	d9 = 10	d2 = 3
M 3	d8 = 8	d3 = 3
M 4	d7 = 7	d4 = 4
M 5	d6 = 6	d5 = 5
M 6		
M 7		

Ratio : T_{MyAlgo}/T_{OPT} = 12/10 = 1,20, ce qui est grandement supérieur au 1,02 trouvé dans l'exemple du diagramme.



Ce dernier exemple nous montre bien que si *nbT* = *nbM**2n n'est pas respecté (on rappel, *n* doit être un entier) et que *nbT* >= *nbM**2*n,* l'algorithme devient très peu efficace. On introduit maintenant la formule suivante pour plus de simplicité *nbT* = *nbM**2n + *nbTâchesRestantes*.

Cela s'explique simplement par le fait qu'une fois chaque machine lui ayant été attribué une pair de tâche (ou plusieurs si $n > 1$), on attribue les *nbTâchesRestantes* au *nbTâchesRestantes/2* machines restantes, ce qui avoir tendance à presque multiplier par deux le ratio d'approximation.

On remarque que quelque soit l'écart entre *min* et *max* le ratio d'approximation reste élevé pour MyAlgo.

Exemple : machines = 5, tâches = 12, min = 1, max = 10 (2, 3, 3, 4, 4, 6, 7, 8, 8, 9, 10, 10)

On a bien *nbT = nbM*2n* qui n'est pas respecté et *nbT >= nbM*2*

M 1	d12 = 10	d1 = 2	d7 = 7	d6 = 6
M 2	d11 = 10	d2 = 3		
M 3	d10 = 9	d3 = 3		
M 4	d9 = 8	d4 = 4		
M 5	d8 = 8	d5 = 5		

Ratio : $T_{MyAlgo}/T_{OPT} = 25/13 = 1,92$, ce qui correspond au résultat attendu

En conclusion, l'algorithme trouvé est donc particulièrement efficace pour les instances respectant les conditions posées : *nbT = nbM*2n* et *nbT >= nbM*2*, ou pour les instances respectant la condition *nbT < nbM*2* et ayant un grand écart entre *min* et *max*.

Dans le reste des cas, l'algorithme va être de très peu efficace à moyennement efficace.