COMPLEX : Ordonnancement de tâches

Benlarbi Samy et Gagnard François

Ce projet consiste en l'implémentation de plusieurs algorithmes d'ordonnancement, suivie de l'analyse des temps d'éxécution de ces derniers pour plusieurs types d'instances, ces dernières ayant été générées de trois manières différentes. Ce rapport contient la liste des réponses aux questions demandées dans le sujet, ainsi que les graphiques des temps d'éxécution pour chaque méthode, commentés et etudiés dans des parties spécifiques.

Nos algorithmes ont été programmé en Objective-CAML dans l'objectif de travailler un langage fonctionnel. En effet, ce paradigme de programmation nous a particulièrement intéressé dans la résolution des divers problêmes posés. La mise en place de certains biais de programmation impératives a cependant été utlisé, notamment pour la lecture de fichiers.

Sommaire

1 Algorithme 3-approché dans tous les cas

Le cas où les tâches sont effectuées le plus rapidement (en t_{min}) est celui où toutes les machines tournent en permanence (simultanément). De là, on a $\sum_{i=1}^{n}(d_i^A + d_i^B + d_i^C) \leq 3 \times t_{min}$, i.e $t_{min} \geq \frac{\sum_{i=1}^{n}(d_i^A + d_i^B + d_i^C)}{3}$. On a $OPT \geq t_{min}$, d'où $OPT \geq \frac{\sum_{i=1}^{n}(d_i^A + d_i^B + d_i^C)}{3}$.

NB : On pourrait généraliser le raisonnement à un problème à n machines pour tout n entier positif non nul :
Le meilleur cas possible reste celui où toutes les machines travaillent en même temps et en permanence. Si l'on note $\mathcal{M}$, le calcul devient donc $\sum_{i=1}^{n}\sum_{k\in\mathcal{M}} d_i^k \leq n \times t_{min}$, i.e. $t_{min} \geq \frac{\sum_{i=1}^{n}\sum_{k\in\mathcal{M}} d_i^k}{n}$ et comme $OPT \geq t_{min}$, on a $OPT \geq \frac{\sum_{i=1}^{n}\sum_{k\in\mathcal{M}} d_i^k}{n}$.

2 Algorithme 2-approché

2-approché

On note res le résultat de l'algorithme approché et t_{res} le temps associé, t_J le temps associé à la la solution renvoyée par l'algorithme de Johnson appliqué aux machines A et B, $\sigma_C = \sum_i^n d_C^i$. On veut montrer que $res \leq 2 \times OPT$.
Distingons deux cas :

- Si $t_J \geq \sigma_C$ alors
  - on ne peut avoir mieux que $t_{res} = t_J$ (la machine C fonctionne que quand A ou B fonctionne).
  - on ne peut avoir pire que $t_{res} = 2 \times t_J$ (car si l'on suppose que C ne fonctionne que quand A et B ont terminé, on a $t_{res} = t_J + \sigma_C \leq 2t_J$).

  or $t_J \leq t_{OPT}$, donc $t_{res} \leq 2t_{OPT}$.
- Si $t_J \leq \sigma_C$ alors
  - on ne peut avoir mieux que $t_{res} = \sigma_C$
  - on ne peut avoir pire que $t_{res} = t_J + \sigma_C \leq 2\sigma_C$

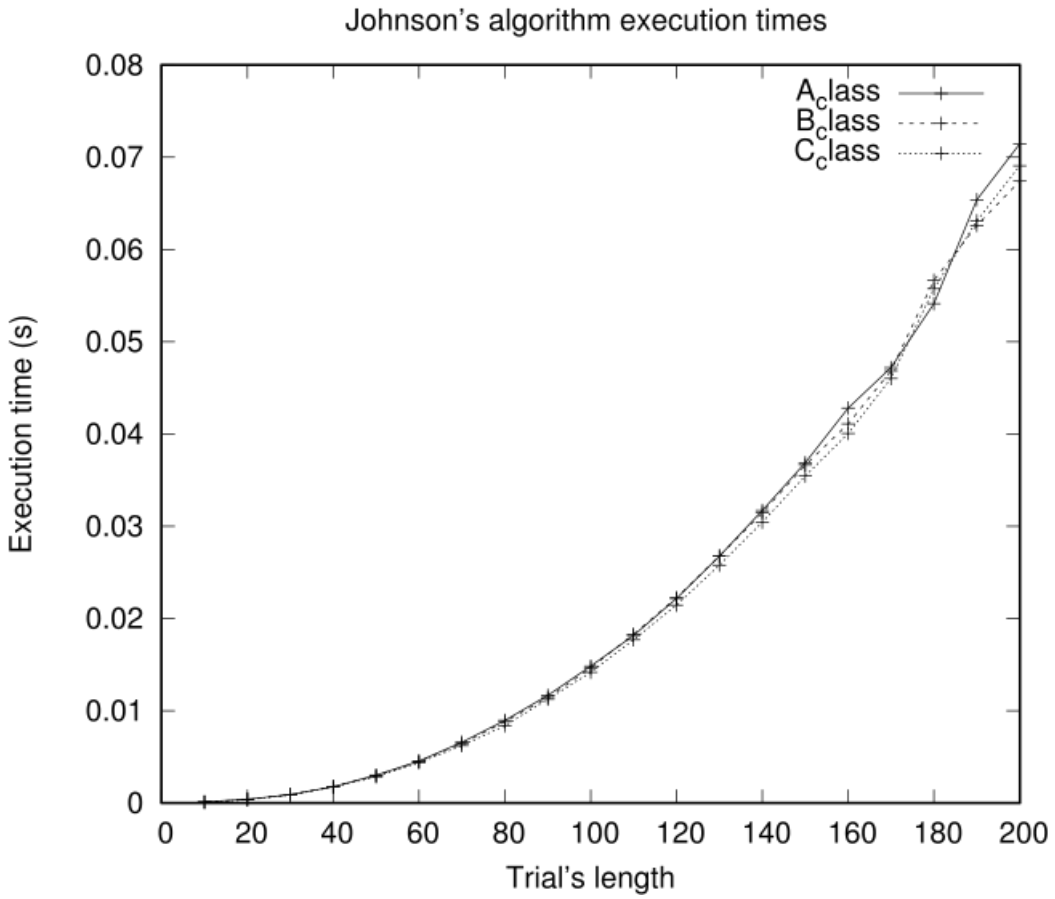
  or $\sigma_C \leq t_{OPT}$, donc $t_{res} \leq 2t_{OPT}$

Reformulation de l'algorithme et analyse de sa complexité

- Trier X par ordre de croissant des valeurs de $\min d_A, d_B$ -> $O(n \log n)$
- Tant que... -> s'exécute $O(n)$ fois.
- Trouver... -> prend un temps $O(log(n))$ (car on a trié X).

La complexité de l'algorithme de Johnson est donc de $O(nlog(n))$.

3 Implémentation de l'algorithme de Johson



On observe que la durée d'éxécution de l'algorithme est sensiblement la même, peu importe la classe d'instances utilisée. On voit néanmoins que les temps d'éxécution liés à la classe A (aucune corrélation) sont légérement plus élevés en général, notamment sur de plus grandes instances.

4 Amélioration de b_1

Dans b_B^π, on analyse ce qu'il se passe séquentiellement : tout d'abord les tâches de π finissent sur les machines A et B, puis les tâches de π' peuvent commencer à passer sur la B, enfin il faut que la dernière tâche s'exécute sur C.
Or avant que les tâches de π' puissent commencer à s'exécuter sur B, il faut aussi que les tâches de π aient finit de s'exécuter sur A ainsi que la première tâche de π' (que l'on suppose ici de temps minimal).
Ainsi, comme il faut aussi attendre au moins $t_A^\pi + \min_{i\in\pi'}(d_A^i)$, le maximum de ces deux valeurs constitue une borne inférieure pour le temps mis avant que les tâches de π' ne commencent à s'exécuter sur B. D'où le résultat.

De même, on doit attendre au moins $t_B^\pi + \min_{i\in\pi'}(d_B^i)$ avant que la première tâche de π' ne commence à être exécutée sur C.
De plus, avant que la première tâche de π' ne commence à s'exécuter sur la C, il faut attendre que toutes les tâches de π aient finit de s'exécuter sur A et que la première tâche de π' se soit exécutée sur A et B. D'où le temps d'attente d'au moins $t_A^\pi + \min_{i\in\pi'}(d_A^i + d_B^i)$.
Ainsi, on a trois conditions nécessaires au fait que la première tâche de π' commence à s'exécuter sur C, on doit donc attendre au minimum le plus grand de ces trois temps.

5 Une autre borne inférieure b_2

Séquentiellement : π s'exécute sur A, les tâches i telles que $i \in \pi'$, $i \neq k$ et $d_A^i \leq d_C^i$ s'exécutent sur A, k s'exécute sur A, B, C et enfin les tâches i telles que $i \in \pi'$, $i \neq k$ et $d_C^i < d_A^i$ s'exécutent sur C. Au mieux, le temps d'attente entre chaque étape sera nul, et on obtient bien la somme voulue.

Cela donne une borne inférieure pour chaque k choisi dans π', on en déduit donc la borne inférieure

$$b_2 = \max_{k \in \pi'} \left(t_A^\pi + d_A^k + d_B^k + d_C^k + \sum_{i \in \pi', i \neq k | d_A^i \leq d_C^i} d_A^i + \sum_{i \in \pi', i \neq k | d_A^i > d_C^i} d_C^i \right)$$

6 b_3, borne inférieure analogue à b_2

On tient le même raisonnement que dans la question (5) mais cette fois en tenant compte de la machine B et non de la A.
Séquentiellement : π s'exécute sur A puis sur B, les tâches i telles que $i \in \pi'$, $i \neq k$ et $d_B^i \leq d_C^i$ s'exécutent sur B, k s'exécute sur B, C et enfin les tâches i telles que $i \in \pi'$, $i \neq k$ et $d_C^i < d_B^i$ s'exécutent sur C. Au mieux, entre chaque étape sera nul, et on obtient la somme

$$t_B^\pi + d_B^k + d_C^k + \sum_{i \in \pi', i \neq k | d_B^i \leq d_C^i} d_B^i + \sum_{i \in \pi', i \neq k | d_B^i \leq d_C^i} d_C^i$$

Cela donne une borne inférieure pour chaque k choisi dans π', on en déduit donc la borne inférieure

$$b_3 = \max_{k \in \pi'} \left(t_B^\pi + d_B^k + d_C^k + \sum_{i \in \pi', i \neq k | d_B^i \leq d_C^i} d_B^i + \sum_{i \in \pi', i \neq k | d_B^i > d_C^i} d_C^i \right)$$

7 Implémentation, tests de performance et borne supérieure

L'algorithme exact a été réalisé grâce à deux fonctions récursives à appels imbriqués. L'une se chargeant de traiter le cas d'un noeud, l'autre de visiter les différents fils. L'arbre d'exécution de la résursion comporte dans le pire des cas $n!$ noeuds.

On remarque une augmentation drastique du temps d'éxécution dés que la taille des instances augmente.

8 Adaptation pour plus de machines

Si l'on rajoute des machines, les bornes inférieures calculées sont toujours des bornes inférieures, mais potentiellement beaucoup moins efficaces : les élagages seront plus rares. On peut donc appliquer l'algorithme tel quel en ne tenant pas compte des nouvelles machines, mais le temps d'exécution bien plus long qu'il ne serait avec des bornes plus efficaces.

9 Optimisations de la méthode exacte

Tout d'abord on peut appliquer les résultats de la question (4) en posant

$$b_1' = \max \left(t_B'^\pi + \sum_{i \in \pi'} d_B^i + \min_{i \in \pi'} d_B^i + d_C^i, t_B'^\pi + \sum_{i \in \pi'} d_B^i + \min_{i \in \pi'} d_C^i, t_C'^\pi + \sum_{i \in \pi'} d_C^i \right)$$

On peut aussi suivre le même raisonnement que dans les questions (5) et (6), ou s'inspirer de l'algorithme de Johnson et poser

$$b_4 = \max_{k \in \pi'} \left(t_A^\pi + d_A^k + d_B^k + \sum_{i \in \pi', i \neq k | d_A^i \leq d_B^i} d_A^i + \sum_{i \in \pi', i \neq k | d_A^i > d_B^i} d_B^i \right)$$

Par ailleurs, les deux premières questions nous donnent deux bornes inférieures :

1. On a vu en (1) que n'importe quelle permutation des tâches donne un résultat 3-approché, on peut donc se contenter de calculer le temps t_{alea} mis par n'importe quelle permutation et le diviser par 3. Ainsi, on obtient une borne inférieure

$$b_0 = \frac{t_{alea}}{3}$$

2. On peut aussi utiliser une solution donnée par l'algorithme 2-approché adapté de l'algorithme de Johnson s'exécutant en temps t_J et obtenir

$$b_0' = \frac{t_J}{2}$$

10 Méthode arborescente approchée

On propose de modifier l'algorithme pour qu'il soit $(1+\epsilon)$-approché pour $\epsilon \in \mathbb{R}_+^*$. On remplace simplement la condition $t_C^\pi = b_{inf}$ dans le cas où l'on se trouve dans une feuille de l'arbre des solutions par $t_C^\pi \leq b_{inf} \times (1+\epsilon)$ de sorte que l'algorithme renvoie une permutation des tâches dès que cette dernière s'exécute sur les machines en au plus $(1+\epsilon)$ fois plus de temps qu'une solution optimale.

11 Evaluation des solutions

Testons la qualité des solutions retournées par l'algorithme de Johnson vis à vis de l'algorithme exact programmé: chaque instance étudiée sera représentée de manière à ce que les lignes correspondent aux machines, et les colonnes aux tâches.

Notons i1, l'instance suivante, dont la solution optimale a un coût de 357, vérifié par l'algorithme exact:

35	12	25	6	76	56	54	25
26	15	41	15	65	68	42	32
13	19	21	25	51	42	28	31

Notons i2, l'instance suivante, dont la solution optimale a un coût de 282, vérifié par l'algorithme exact:

36	47	5	78
15	89	11	61
18	31	7	62

- Pour i1, on a

$$rapport = \frac{Solu_{Johnson}}{OPT_{exact}} = \frac{357}{357} = 1$$

- Pour i2, on a

$$rapport = \frac{Solu_{Johnson}}{OPT_{exact}} = \frac{300}{282} = 1.06382978723$$

On obtient des rapports proches voir égaux à 1. Johnson est ainsi quasi-optimal sur de petites instances.