Robin WIBAUX
Mica MÉNARD
M1 ATAL

Distanciel d'Algorithmes et Complexité

Le problème MIN-MAKESPAN

Question 1 : Indiquer ce que donne l'algorithme LPT sur l'exemple de l'Exercice 3.2 de la feuille TD3, sous la forme d'un dessin similaire à la Figure 1 de l'Exercice 3.2.

M_3	6	3	2	2
M_2	6	5		2
M_1	7	3	2	1

Question 2 : Quel est le ratio d'approximation obtenu par LPT sur cet exemple ?

On avait déduit que :

$$T_{opt}(I) \geq \frac{\sum_{i=1}^{n} d_i}{m}$$

Pour cette instance, nous obtenons $T_{opt}(I) = \frac{39}{3} = 13$, donc $T_{opt}(I) = T_{LPT}(I) = 13$.

On en déduit que $T_{LPT}(I) = r \times T_{opt}(I)$ avec $r = 1$.

En d'autres termes, nous trouvons un ratio d'approximation de 1 sur cet exemple.

Question 3 : Pour toute instance I de MIN-MAKESPAN, on note $T_{LTP}(I)$ le temps obtenu par l'algorithme LPT sur I et $T_{opt}(I)$ le temps optimal recherché pour I.

Montrer que pour toute instance I, $T_{LPT}(I) \leq \frac{\sum d'_k}{m} + d'_j$, ou j est le numéro de la tâche qui se termine en dernier.

La moyenne de l'ensemble des tâches correspond à une borne inférieure de l'optimal (voir Q2). Lorsqu'on exclut la tâche j de cette valeur, on obtient la situation suivante :

M_3	6	3	
M_2	6	d_j=5	
M_1	7	3	

$$T_{LPT}(I) = 6 + 5 = 11 \text{ et } \frac{\sum_{k \neq j} d'_k}{m} + d'_j = \frac{6+3+6+7+3}{3} + 5 = \frac{25}{3} + 5 = 8.33 + 5 = 13.33 > T_{LPT}(I)$$

M_3	25/3	
M_2	25/3	d_j=5
M_1	25/3	

Dans cette situation "idéale", la solution trouvée par LPT ne peut être plus que grande que la durée moyenne des tâches exceptée j, additionnée à j, puisque $T_{LPT}(I)$ correspond au temps auquel la dernière tâche se termine.

Chaque tâche est assignée à la machine la moins occupée (c'est-à-dire la machine dont la somme des tâches à traiter est minimale au moment de l'assignement d'une nouvelle tâche). Il en est inévitablement de même pour la tâche j, soit la dernière tâche à terminer. Toutes les autres machines termineront inévitablement après que j aie commencé.

Donc : soit moy la somme des durées des tâches - moins la tâche j - divisée par le nombre de machines, et soit deb le début de la tâche j. On aura forcément $moy \geq deb$.

Question 4 : Supposons pour commencer que $n \leq m$. En déduire dans ce cas que LPT est toujours optimal.

S'il y a, au pire des cas, autant de machines que de tâches, cela signifie que chaque machine exécute au plus une seule tâche. On en déduit que dans ce cas, le temps optimal correspond

au temps de terminaison de la dernière tâche. Cette valeur correspond dans tous les cas à $T_{LPT}(I)$. Par conséquent, lorsque $n \leq m$, on a $T_{opt}(I) = T_{LPT}(I)$.

Question 5 : Supposons maintenant que $n \geq m+1$.

M_m	d_m	
...	...	
M_{i+1}	d_{i+1}	
M_i	d_i	d_n

Montrer que $T_{opt}(I) \geq 2d'_{m+1}$.

Soit d_i une des m premières tâches affectées. On a alors $d_i \geq d_{m+1}$ car toutes les tâches sont triées dans l'ordre décroissant. Par conséquent, $d_i + d_{m+1} \geq 2d_{m+1}$.

D'où $T_{opt}(I) \geq 2d'_{m+1}$.

Question 6 : Appelons j le numéro de la tâche qui se termine en dernier quand LPT est appliqué. Montrer que l'on est nécessairement dans un des deux cas suivants :

$(a)\ T_{LPT}(I) = T_{opt}(I)$ ou $(b)\ j \geq m+1$

Plaçons nous dans le cas $\neg(b)$, c'est-à-dire où $j < m+1$.
j fait alors partie des m premières tâches à avoir été assignées - soit parce qu'il y avait m tâches ou moins, soit parce que la tâche j était tellement longue que les autres ne l'ont pas dépassé en longueur totale (une fois réparties dans les autres machines).
Dans tous les cas, nous sommes dans une situation où la solution obtenue est *forcément* la solution optimale, car elle correspond à la durée d'une tâche (placée en première dans sa file, et terminant en toute dernière), à savoir une durée irréductible.

On peut donc déduire l'implication suivante : $\neg(b) \Rightarrow (a)$

Il est possible de développer la formule pour arriver à la conclusion suivante : $\neg(a) \Rightarrow (b)$

C'est-à-dire que si $T_{LPT}(I) \neq T_{opt}(I)$, donc si l'algorithme LPT ne donne pas la solution optimale, alors on est inévitablement dans le cas $j \geq m + 1$.

De tout cela, on peut conclure l'impossibilité de se trouver dans la situation suivante :
$\neg(a)$ et $\neg(b)$
car chaque membre de la conjonction implique la négation de l'autre membre.

Question 7 : En vous appuyant sur les questions précédentes, montrer que pour toute instance I, $T_{LPT}(I) \leq r \times T_{opt}(I)$ *où r est une constante dont vous donnerez la valeur.*

Pour tout $j < m + 1$, on a conclus que $T_{LPT}(I) = r \times T_{opt}(I)$ avec $r = 1$.

Pour tout $j \geq m + 1$, on a :

$(a)\ T_{LPT}(I) - d_j \leq T_{opt}(I)$
$(b)\ d_{m+1} \leq \frac{1}{2}\ T_{opt}(I)$

Par ailleurs, la relation entre d_j et d_{m+1} est la suivante :

$d_{m+1} \geq d_j$

Car, les tâches étant triées dans l'ordre décroissant, d_j a potentiellement été placé après d_{m+1} .

Nous pouvons donc en déduire l'inégalité suivante :

$(a')\ T_{LPT}(I) - d_{m+1} \leq T_{opt}(I)$

Maintenant, posons cette égalité triviale :

$T_{LPT}(I) = T_{LPT}(I) - d_{m+1} + d_{m+1}$

Nous pouvons constater que le membre de droite de l'égalité est composé des membres de gauches des inégalités (a') et (b), ce qui nous amène à conclure :

$T_{LPT}(I) = (a') + (b) \leq T_{opt}(I) + \frac{1}{2}T_{opt}(I) = \frac{3}{2}T_{opt}(I)$

Soit :

$T_{LPT}(I) \leq \frac{3}{2}T_{opt}(I)$

Question 8 : Conclure quant à l'approximabilité de l'algorithme LPT.

En déterminant la dernière équation, nous avons conclus que $T_{LPT}(I) \leq \frac{3}{2} T_{opt}(I)$, avec $\frac{3}{2}$ correspondant au ratio d'approximabilité de l'algorithme LPT. En d'autres termes, nous déterminé que toute solution fournie par l'algorithme LPT est au pire des cas égale à $\frac{3}{2}$ fois la solution optimale.

Rapport de programmation

https://github.com/Dreyn/complexityWork

Ce projet de programmation consistait à implémenter trois algorithmes de résolution de problème Min Makespan, à savoir deux algorithmes d'approximations qui nous sont donnés et un algorithme de notre choix.

Le langage choisi pour ce projet est le C++.

Fonctionnalités du programme :

Notre programme s'installe en compilant le fichier principal main.cpp. Il suffit ensuite d'exécuter dans le terminal le programme créé pour accéder à ses fonctionnalités. Il est aussi possible d'ajouter un paramètre (à savoir un nom de fichier) à l'exécution du programme :

```
# ./program fichier_instance
```

Si le fichier ajouté en paramètre contient une instance de problème (correspondant au format demandé dans le projet), alors le programme traitera directement l'instance.

Le reste des fonctionnalités correspond à ce qui était demandé dans le projet.

Descriptif du programme :

Notre programme inclut à `main.cpp` deux fichiers annexes : `structures.cpp` et `algorithms.cpp`.

Le fichier `structures.cpp` contient les deux Structures utilisées dans ce programme : `Instance` et `Solution`. Le but à l'origine de ces Structures était simplement de pouvoir faire retourner plusieurs variables par une fonction. De ce fait, ces structures contiennent très peu de méthodes.

La structure `Instance` décrit une instance. Une `Instance` possède un nombre de machines, une liste de tâches, ainsi que l'indication de la plus grande tâche et de la somme des durées par machine. Ces deux derniers attributs sont calculés lors de la création d'une `Instance` et utilisés pour la borne inférieure de la solution optimale.

La structure `Solution` décrit une solution. Une `Solution` contient la liste des tâches (qu'on trouve aussi dans Instance), la liste contenant le numéro des machines pour chaque tâche, un tableau des tâches par machines (une ligne par machine, et dans

chaque ligne, une case par tâche assignée à la machine), ainsi que la liste des charges de chaque machine, c'est à dire la somme de toutes les tâches pour chaque machine (correspondant au résultat final, et non au calcul d'approximation qu'on trouve dans `Instance`). Un cinquième attribut est le temps final obtenue dans cette `Solution`.

Ces deux structures ont aussi une méthode en commun : la méthode `display()` qui, comme son nom l'indique, affiche dans le terminal le contenu des structures.

Les listes dans notre programme sont gérées par les `vector` de la librairie standard de C++.

Le fichier `algorithms.cpp` contient les trois algorithmes implémentés. Ce sont donc des fonctions qui prennent en paramètre une `Instance` et retournent une `Solution`.

Nous reviendrons plus tard sur `MyAlgo`.

Tout le reste se passe sur `main.cpp`.

On y trouve une fonction `instanceCreation`, prenant en paramètre une chaîne de caractères et retournant une `Instance`. La chaîne de caractères doit être correctement formatée, selon les instructions données dans l'énoncé du distanciel.

Puis on y trouve les cinq procédures restantes.

La première, `userInterface`, est la procédure appelée dans la fonction principale `main`. Elle prend les même paramètres que `main`, à savoir `argc` et `argv` qui gèrent les paramètres qu'on peut entrer en exécutant le programme en ligne de commande.

Si aucun paramètre n'était donné en ligne de commande, `userInterface` demandera à l'utilisateur comment générer l'instance à traiter. Selon le choix de l'utilisateur, `userInterface` appelle différentes procédures.

`generateFromInput` laisse l'utilisateur entrer une chaine de caractère du format voulu, puis transforme cette chaîne en `Instance` via `instanceCreation`. Elle appelle alors `singleInstance`, qui appliquera sur l'instance créée les trois algorithmes, puis affichera les résultats sur le terminal comme demandé dans l'énoncé.

`generateFromFile` prend en paramètre le nom du fichier contenant l'instance. Elle récupère la chaîne contenue dans le fichier et la transforme en instance via `instanceCreation`. Elle appelle ensuite `singleInstance`. Cette procédure est aussi appelée d'office quand un paramètre est entré en ligne de commande.

`generateFromRandom` demande à l'utilisateur cinq paramètres pour générer le bon nombre d'instances voulues. Cette procédure se fait en plusieurs étapes : la génération de

la liste d'`Instances`, celle des trois listes de `Solutions` (une liste par algorithme de résolution) et des moyennes des ratios d'approximations, puis l'étape de création du fichier. Le nom du fichier y est demandé à l'utilisateur, sans prendre en compte si un tel fichier existe déjà.

Présentation de **`MyAlgo`** :

L'idée de cet algorithme est de prendre une solution proposée par un autre algorithme et de voir s'il est possible d'inverser des tâches entre deux machines pour alléger la machine la plus chargée.

Notre algorithme prend la solution de l'algorithme LPT car les tâches y sont fournies déjà triées, ce qui permet certaines optimisations. De plus, LPT donne en moyenne de meilleurs résultat que LSA.

La machine la plus chargée sera systématiquement comparée aux autres, car abaisser la charge de cette machine reviendrait à améliorer le résultat. Nous comparerons donc ses tâches avec celles des autres machines, à partir de la moins chargée pour obtenir une différence plus importante, de manière à rapprocher les charges de ces deux machines à la charge moyenne par machine. Les machines sont donc préalablement triées par leurs charges.

Quand nos deux machines `m1` et `m2` sont sélectionnées, nous commençons par calculer leur `delta`, c'est à dire la moitié de la différence entre leurs charges. A partir de là, nous y cherchons deux tâches `t1` et `t2` (appartenant donc respectivement à `m1` et `m2`) où `t1` est supérieure à `t2`. Nous chercherons alors à avoir la plus grande différence possible, sans dépasser `delta`.

Plusieurs interruptions de boucles sont faites pour optimiser le code :

Lorsque nous passons à une machine `m2` (donc plus chargée que la machine `m2` précédente), si le nouveau `delta` calculé est inférieur à la différence entre les deux meilleurs tâches trouvées, il ne reste alors plus à rien de continuer car nous ne trouverons pas de meilleur paire de tâche.

Le parcour des tâches se fait, dans `m1`, de la plus grande à la plus petite et, dans `m2`, de la plus petite à la plus grande. Les différences calculées seront donc trouvées dans un ordre décroissant. Dés qu'on trouve une paire où la différence est inférieure ou égale à `delta`, nous pouvons interrompre l'une des deux boucles de parcours des tâches.

Lorsque notre meilleur paire de tâches est trouvée, nous inversons leurs propriétaires.

Dans notre programme, cela se traduit par les supprimer de leurs listes pour leurs machines respectives, et les insérer dans leurs nouvelles listes en respectant l'ordre décroissant des tâches. Puis mettre à jour la liste des charges par machine et le temps final de la solution, abaissé en conséquence. Une autre conséquence de toute cela est qu'il sera nécessaire de trier à nouveau la liste des charges des machines pour recommencer toutes ces opérations.

Tout cela est répété jusqu'à ce que nous ne trouvions plus de paire de tâches à échanger.

```
Répéter
   Soit m0 la machine la plus chargée
   Pour chaque machine m (excepté m0)
          delta = [charge(m0) - charge(m)] / 2
          Pour chaque tâche t1 de m0
                 Pour chaque tâche t2 de m
                        prendre la paire (t1, t2) avec le plus gros écart,
                        tel que (t1 - t2) >= delta
                 finPour
          finPour
   finPour
   Échanger t1 et t2 dans m0 et m_trouvé
Jusqu'à( aucune paire de tâches à échanger )
```

Complexité :

Notre algorithme est divisible en deux parties : l'exécution d'un algorithme d'approximation (ici LPT, algorithme polynomial), puis tout ce qui a été décrit précédemment.

Le trie de la liste des charges n'a pas un coût important, à savoir au pire le carré du nombre de machines.

Les trois boucles pour imbriquées, pour naviguer parmis les machines et les tâches, ont un coût se rapprochant du carré du nombre total de tâches.

Le meilleurs des cas serait nombre de tâches très homogène parmis les machines. Donc soient m le nombre de machines et n le nombre de tâches, ces trois boucles correspondraient (à peu près) à :

$m*(n/m * n/m) = n^2/m$

Le pire des cas serait où deux machines monopoliseraient quasiment toutes les tâches :

$m*((n-m)*(n-m)) = m*(n^2 - 2nm + m^2)$

Tout cela reste polynomial.

Maintenant, si nous prenons en compte la boucle while, cela devient très compliqué à prédire.

Cette boucle s'exécute tant que l'algorithme est capable d'améliorer la solution. Chaque tour de cette boucle implique que l'écart entre la machine la plus chargée et la moins chargée diminue (en prenant en compte que les machines les plus et moins chargées changent).

Résultat exact ou approché ?

Nous n'avons aucune preuve que notre algorithme soit exact, et nos tests tendent à infirmer l'hypothèse de l'exact. Il donne cependant un résultat quasi-systématiquement meilleur que LPT, même si encore une fois, nous ne sommes pas capable de donner son ratio d'approximation.

Notre conclusion est finalement très flou, puisque nous ne connaissons ni la complexité précise de MyAlgo (due à la boucle while), ni son ratio d'approximation.

Mais du fait de l'utilisation préalable de LPT, le nombre de tours de boucle while ne nous a pas semblé d'un ordre supérieur au nombre de tâches.

Quant à son approximation, il s'agit finalement d'un algorithme d'amélioration de résultat.