École Nationale Supérieure des Sciences Appliquées et de Technologie
Logiciel et Systèmes Informatiques - Deuxième année

Algorithmique avancée

Ordonnancement simple

Kévin Vythelingum

Jean-Michel Nokaya

5 février 2014

Chargé de cours :
Olivier Pivert

Table des matières

1 Introduction

Ce projet consiste à examiner deux manières différentes pour résoudre le même problème afin de comparer leurs complexités. Ainsi, nous nous intéressons à un problème d'ordonnancement simple, où différentes tâches peuvent être exécutées avec des durées différentes, moyennant un coût plus important pour une durée plus courte. Il s'agit de déterminer le meilleur ordonnancement pour avoir un coût minimal en s'étant fixé une durée maximale d'exécution. Pour cela, nous réaliserons un premier algorithme suivant la méthode des essais successifs, puis nous élaborerons un algorithme de programmation dynamique.

2 Préliminaires

1. On cherche tout d'abord à montrer que la solution optimale recherchée est trouvée pour $\sum_{i=1}^{n} d_i = D$.

   Premièrement, il est important de remarquer qu'on a nécessairement $\sum_{i=1}^{n} d_i \leq D$ car D est la durée globale maximale. Or, si on considère une solution S telle que $\sum_{i=1}^{n} d_i < D$, alors il existe au moins un d_i qui ne soit pas maximal par rapport aux durées satisfaisantes pour la tâche T_i. Il existe donc une solution S' telle que $\sum_{i=1}^{n} d'_i \leq D$ et $\sum_{i=1}^{n} d'_i > \sum_{i=1}^{n} d_i$. Comme le coût diminue lorsque la durée augmente, S' est donc une meilleure solution que S. S n'est donc pas optimale.

   On en déduit que la solution optimale recherchée est trouvée pour $\sum_{i=1}^{n} d_i = D$.
2. Ensuite, on s'intéresse à la complexité au pire d'une solution de type essais successifs en termes de nombre de candidats à examiner.

   La complexité au pire correspond à considérer toutes les combinaisons différentes possibles. Il s'agit donc de dénombrer le nombre N de candidats.

   Construire une solution consiste à choisir une durée pour T_1 parmi les k durées possibles, puis une durée pour T_2, et ainsi de suite jusq'à T_n. On a donc :

$$\begin{aligned} N &= \underbrace{k \times k \times ... \times k}_{nfois} \\ N &= k^n \end{aligned}$$

   La complexité recherchée vaut donc k^n.

3 Méthodes des essais successifs

3.1 Analyse

On commence par définir les différents éléments qui interviennent dans le cadre de la méthode des essais successifs. En particulier, il s'agit de définir ce qu'est un vecteur représentant un candidat (solution) et quels sont les constituant de l'algorithme générique $(S_i, satisfaisant, enregistrer, soltrouvee, defaire)$.

Solution : un candidat est un vecteur de taille n où chaque coefficient est une durée choisie parmi l'ensemble {1,...,k} (à chaque tâche on associe une durée). On choisit d'enregistrer les choix réalisés dans un tableau T de taille n.

S_i : l'ensemble des durées possibles de 1 à k

$satisfaisant(x_i) = \sum_{j=1}^{i} x_j \leq D$
(la somme partielle des durées choisies est inférieure à la durée maximale autorisée)

$enregistrer(x_i) = T[i] \leftarrow x_i$

soltrouvée : $i = n$

$defaire(x_i) = T[i] \leftarrow 0$

Pour simplifier les vérifications au niveau de satisfaisant et des conditions d'élagage, on utilisera les variables entières *cout* et *duree* initialisée à 0, qui représenteront le coût courant et la durée courante duent à nos choix de durées. Elles seront mises à jour dans *enregistrer* et dans *défaire*. En effet, en notant *CD* le tableau à deux dimensions ayant les coûts en ligne et les durées en colonne, on effectura dans *enregistrer* :

$$\begin{aligned} \text{coût} &\leftarrow \text{coût} + CD[i][x_i] \\ \text{durée} &\leftarrow \text{durée} + x_i \end{aligned}$$

Ensuite on effectura dans *défaire* :

$$\begin{aligned} \text{coût} &\leftarrow \text{coût} - CD[i][x_i] \\ \text{durée} &\leftarrow \text{durée} - x_i \end{aligned}$$

3.2 Condition d'élagage

On a montré dans les préliminaires que la solution optimale recherchée est trouvée pour $\sum_{i=1}^{n} d_i = D$. Cela va nous servir de base pour notre condition d'élagage. En effet, on va élaguer si on ne peut pas atteindre D avec les durées restantes à choisir. Autrement dit, en notant S_1 la somme des durées choisies et S_2 la somme des durées maximales restantes, on élaguera si $S_1 + S_2 < D$. Cependant, lorsque D est grand (i.e. qu'il est impossible de l'atteindre, même en choisissant toutes les durées maximales), il ne faudra pas élaguer.

L'intérêt de cette condition d'élagage est d'arrêter plus tôt la recherche d'une solution en prévoyant qu'elle ne pourra pas être optimale. Nous vérifierons par des expérimentations que cette condition améliore effectivement la complexité temporelle en terme de nombre d'appels récursifs.

3.3 Algorithmes

3.3.1 Sans élagage

Cet algorithme repose sur le modèle de programmation de recherche d'une solution optimale par essais successifs. Le principe est d'essayer toutes les combinaisons possibles tant qu'elles sont satisfaisantes, c'est-à-dire dans notre cas que la somme partielle des durées choisies est inférieure à la durée maximale autorisée D. Pour obtenir la solution optimale, on vérifie que le coût d'une solution obtenue est strictement inférieur au coût de la meilleure solution trouvée, auquel cas on l'affiche. La dernière solution affichée sera donc la solution optimale.

L'appel de l'algorithme consiste à initialiser le coût optimal à une très grande valeur, puis à appeler la procédure au rang 1, c'est-à-dire commencer par choisir la durée de la première tâche T_1. On initialisera également les variables *cout* et *duree* à 0.

procédure *ordonnancement_simple* (ent i) ;
var ent x_i ;
début
    pour x_i **de** 1 **à** k **faire**
        si $duree + x_i \leq D$ **alors**
            $duree \leftarrow duree + x_i$;
            $cout \leftarrow cout + cd[i][x_i]$;
            $T[i] \leftarrow x_i$;
            si i = n **alors**
                si $cout < cout\_opt$ **alors**
                    $cout\_opt \leftarrow cout$;
                    $affiche(T)$;
                fsi ;
            sinon
                $ordonnancement\_simple(i + 1)$;
            fsi ;
            $T[i] \leftarrow 0$;
            $cout \leftarrow cout - cd[i][x_i]$;
            $duree \leftarrow duree - x_i$;
        fsi ;
    fait ;
fin ;

Appel :
$coutOpt \leftarrow \infty$;
$cout \leftarrow 0$;
$duree \leftarrow 0$;
$ordonnancement\_simple(1)$;

3.3.2 Avec élagage

Pour réaliser l'algorithme avec élagage, on dispose d'une fonction *encorepossible(di,ci)* qui prend en argument la durée et le coût sélectionnés et retourne un booléen : `vrai` si on peut encore espérer trouver une solution optimale, `faux` s'il est nécessaire d'élaguer. En partant de l'algorithme précédent, effectue l'appel à *ordonnancement_simple* pour le rang suivant à condition que *encorepossible* soit `vrai`. On obtient alors l'algorithme :

procédure *ordonnancement_simple* (ent i) ;
var ent x_i ;
début
    pour x_i **de** 1 **à** k **faire**
        si $duree + x_i \leq D$ **alors**
            $duree \leftarrow duree + x_i$;
            $cout \leftarrow cout + cd[i][x_i]$;
            $T[i] \leftarrow x_i$;

```
            si i = n alors
                si cout < cout_opt alors
                    cout_opt ← cout ;
                    affiche(T) ;
                fsi ;
            sinon
                si encorepossible(duree) alors
                    ordonnancement_simple(i + 1) ;
                fsi ;
            fsi ;
            T[i] ← 0;
            cout ← cout − cd[i][x_i];
            duree ← duree − x_i;
        fsi ;
    fait ;
fin ;
```

Appel :
$coutOpt \leftarrow \infty$;
$cout \leftarrow 0$;
$duree \leftarrow 0$;
$ordonnancement\_simple(1)$;

Le comportement de *encorepossible* reprend le principe d'élagage énoncé dans la section précédente. On suppose que l'on dispose d'un tableau *dmax* de taille n qui est initialisé à la durée maximale qu'il est possible d'allouer à chaque tâche, et d'une fonction *sommeTableau(tableau,taille)* qui permet de calculer la somme des éléments d'un tableau d'entiers.

```
fonction encorepossible (ent di, ent ci) : booléen ;
var ent S ;
début
    si (S < D) et (ci = 1) alors
        retourner vrai ;
    sinon
        si S − dmax[ci] + di < D alors
            retourner faux ;
        sinon
            dmax[ci] ← di;
            retourner vrai ;
fin
```

3.4 Expérimentations

Pour tester la validité de nos algorithmes et mesurer leurs performances, nous affichons la solution optimale proposée ainsi que le nombre d'appels effectués à la procédure *ordonnancement_simple*. Cela nous permettra de comparer les algorithmes avec et sans élagage.

Tout d'abord, nous avons testé l'exemple proposé par le sujet, c'est-à-dire avec $n = 4$, $k = 5$, $D = 10$, et le tableau suivant :

cd	1	2	3	4	5
c_1	110	90	65	55	–
c_2	120	90	70	50	40
c_3	90	70	65	60	–
c_4	65	60	55	–	–

Nous obtenons les résultats suivant pour les algorithmes sans et avec élagage :

Sans élagage

```
3 4 2 1
duree = 10 cout = 250

Nombre d'appels : 121
```

Avec élagage

```
3 4 2 1
duree = 10 cout = 250

Nombre d'appels : 99
```

Nous obtenons heureusement le même résultat avec les deux algorithmes, c'est-à-dire affecter respectivement les durées 3, 4, 2, 1 aux tâches c_1, c_2, c_3, c_4. De plus, nous arrivons à faire 1.2 fois moins d'appels avec la condition d'élagage, ce qui montre son intérêt.

Nous effectuons ensuite des tests avec des jeux de données différents en vérifiant à chaque fois la cohérence des résultats obtenus. Pour cela, nous avons conçu un programme *es-experimentations* qui génère des tableaux *cd* avec des valeurs aléatoires comprises entre 10 et 1000 ou $+\infty$ et les applique sur les deux algorithmes. L'objectif est de comparer les performances des deux algorithmes. Ainsi, nous avons recensé les nombres d'appels obtenus pour des valeurs de k et n différentes dans le tableau suivant.

Test	n	k	D	Nombre d'appels sans élagage	Nombre d'appels avec élagage
0	4	5	10	121	99
1	4	5	10	121	113
2	4	5	10	121	51
3	4	5	10	121	101
4	4	5	10	121	113
5	4	5	10	121	117
6	10	5	30	2 015 376	274 900
7	10	5	30	2 015 376	809 160
8	10	5	30	2 015 376	275 006
9	10	10	30	19 580 242	8 151 656
10	10	10	30	19 580 242	10 869 695
11	10	10	30	19 580 242	372 187
12	10	20	30	22 958 977	394 759
13	10	30	30	22 964 087	369 466
14	10	50	30	22 964 087	369 246
15	10	100	30	22 964 087	931 513
16	10	5000	30	22 964 087	368 336

On remarque deux choses. Tout d'abord, le nombre d'appels sans élagage ne dépend pas du jeu de données *cd* étudié. Ensuite, un seuil est atteint pour l'algorithme sans élagage avec un grand nombre de durées possibles. Ceci peut être explqué par le fait que fixer D impose une certaine plage de valeurs possibles pour les durées : cela ne sert à rien de choisir comme première valeur un di supérieur à D, la solution ne pourrait alors pas être satisfaisante. On obtient donc le même nombre d'appels pour les cas où $k \geq D$.

Il s'agit maintenant de fixer k à 10 et de faire varier n.

Test	n	k	D	Nombre d'appels sans élagage	Nombre d'appels avec élagage
0	5	10	30	10 396	11
1	10	10	30	19 580 242	2 721 266
2	15	10	30	449 171 192	182 779 801
3	20	10	30	1 010 258 502	581 312 322
4	50	10	30	1 063 267 584	1 063 267 584

On remarque qu'il n'est possible de monter que jusqu'à $n = 50$ dans un temps raisonnable. En effet, la complexité étant en k^n d'après les questions préliminaires, le nombre d'appels augmente de manière exponentielle.

En somme, la méthode des essais successifs est idéale pour se faire une idée du résultat avec des petites valeurs, puisqu'elle demande peu d'efforts de conception grâce à une trame fixe pour l'algorithme. En revanche, le nombre d'appels est exponentiel selon le nombre de tâches à ordonnancer. Il devient donc très long de faire tourner l'algorithme avec des grandes valeurs.

4 Programmation dynamique

On cherche le coût associé à l'affectation optimale de d unités de temps aux tâches consécutives T_1 à T_j. Soit *coût-opt(j, d)* la fonction réalisant cette association.

4.1 Formule de récurrence

On s'appuie sur le *principe d'optimalité de Bellman* : toute sous-solution d'une solution optimale est optimale. Ainsi, le coût de j tâches auxquelles on accorde une durée totale d de manière optimale est le minimum de la somme entre le coût de *j-1* tâches auxquelles on accorde *D-l* unités de temps, où l parcourt l'ensemble des durées possibles pour la tâche j, et le coût de la tâche j. En prenant en compte les cas particuliers, c'est-à-dire lorsque $j = 1$, $j = d$ et $j > d$, on obtient :

$$\textit{coût-opt(j,d)} = \begin{cases} cd[1,d] & \text{si } j = 1 \text{ et } d \leq d_{max}[j] \\ c_{min}[j] & \text{si } j = 1 \text{ et } d > d_{max}[j] \\ \textit{coût-opt(j-1,j-1)} + cd[j,1] & \text{si } j = d \\ \infty & \text{si } j > d \\ min_{l \in cd[j]}(\textit{coût-opt(j-1,d-l)} + cd[j,l]) & \text{sinon} \end{cases}$$

4.2 Structure tabulaire

Pour enregistrer les résultats, on décide de remplir un tableau tel que chaque ligne correspondra à un nombre de tâche à exécuter et chaque colonne représentera une durée totale. Ainsi, le résultat cherché, autrement dit *coût-opt(j, d)*, se trouvera à la case d'indice de ligne j et d'indice de colonne d. De plus, notre stratégie de remplissage du tableau sera de le remplir horizontalement, ligne par ligne. En effet, si on regarde notre formule de récurrence, on peut remarquer qu'on n'a besoin que de valeurs situées sur la ligne précédente pour continuer.

4.3 Algorithme

Une fois cette analyse terminée, il s'agit de réaliser un algorithme exclusivement itératif. La fonction *coût-opt(j, d)* remplira le tableau que nous avons défini précédemment puis retournera la valeur située à la case d'indice de ligne j et d'indice de colonne d.

4.4 Complexité temporelle

La complexité temporelle est liée au remplissage de la structure tabulaire. Tout d'abord, il s'agit de remplir $k \times n$ cases et chaque case correspond à une somme de k pour le calcul du mininum ($\sum k = k \times (k+1)/2$). On obtient donc une complexité égale à $k \times n \times k^2 = nk^3$.

4.5 Complexité spatiale

La complexité spatiale n'est pas négligeable comme dans le cas des algorithmes à essais successifs. Il convient donc de s'y intéresser. Tout d'abord, elle est liée au remplissage du tableau rempli dans le but de renvoyer la valeur recherchée. Il faut donc compter le nombre de cases à remplir avant de trouver le résultat : c'est $j \times d$ puisque on réalise autant de ligne qu'il y a de tâches à ordonnancer et autant de colonne que de durée possibles d'ordonnancement.

4.6 Expérimentations

Pour expérimenter cet algorithme, nous avons procédé en deux temps. Tout d'abord, nous avons essayé d'obtenir un premier résultat avec les valeurs du sujet. Ensuite, nous avons mesuré les performances de cet algorithme en testant le programme *prog-dyn-experimentations* avec de grandes valeurs (on génère le tableau des coûts et durées aléatoirement).

Premièrement, les résultats que nous obtenons avec les valeurs de l'énoncé sont les suivants :

```
Tableau des couts et durees :

110 90 65 55 4200
120 90 70 50 40
90 70 65 60 4200
65 60 55 4200 4200

n = 4
k = 5
D = 10

dmax : [4,5,4,3,]

cmin: [55,40,60,55,]

110, 90, 65, 55, 55, 55, 55, 55, 55, 55,
4200, 230, 200, 180, 155, 135, 115, 105, 95, 95,
4200, 4200, 320, 290, 270, 245, 225, 205, 185, 175,
4200, 4200, 4200, 385, 355, 335, 310, 290, 270, 250,

cout_opt : 250
```

On retrouve heureusement la valeur calculée précédemment.

Enfin, nous avons fait varier les valeurs k et n en fixant l'autre à 10. On remarque qu'on peut faire monter ces deux valeurs à plus de 10 000 sans que le temps d'exécution du programme dépasse les 1 seconde. De plus, la capacité du langage Pascal à représenter les grands nombres atteint ses limites avant les limites de notre programme.

On peut conclure que l'algorithme utilisant la méthode de programmation dynamique est beaucoup plus efficace que celui à essais successifs.

5 Compléments

1. Les algorithmes à essais successifs sont plus faciles à concevoir car ils suivent une trame fixe. Il suffit de déterminer les constituants de l'algorithme générique. En revanche, pour la méthode de programmation dynamique, il faut trouver une relation de récurrence qui n'est pas toujours évidente. Il est vrai qu'une fois qu'on l'a déterminé il devient aisé de produire l'algorithme mais cette étape est souvent dynamique. C'est pourquoi la méthode des essais successifs demande moins d'efforts de conception que la méthode de programmation dynamique qui est cependant plus efficace en terme de complexité temporelle.
2. D'après les résultats obtenus dans la phase d'expérimentation des algorithmes à essais successifs, on remarque qu'on peut aller en pratique jusqu'à une valeur $n = 50$

en fixant $k = 10$. On pouvait s'y attendre car la complexité au pire en termes de nombre de candidats à examiner est en k^n. Le nombre d'appels augmente donc de façon exponentielle, ce qui implique d'avoir une valeur de n faible pour pouvoir exécuter l'algorithme avec un temps raisonnable.

3. D'après les résultats obtenus dans la phase d'expérimentation des algorithmes à essais successifs, on remarque qu'on peut aller en pratique jusqu'à une valeur $k = 5000$ en fixant $n = 10$. Il est important de noter que la version sans élagage atteint une asymptote : cela est dû au fait que lorsque $k \geq D$, choisir une durée supérieure à D pour une tâche ne peut pas être satisfaisant. De plus, le nombre d'appels ne dépend pas des valeurs du tableau *cd* pour la version sans élagage, ce qui n'est pas le cas de la version avec élagage. Cela peut s'expliquer par le rang des durées max possible qui varie selon les tirages aléatoires.
4. Comme nous l'avons évoqué précédemment, il est possible de faire fonctionner le programme utilisant la méthode de programmation dynamique avec de très grande valeurs sans augmenter significativement le temps d'exécution : les limites du langage Pascal pour représenter les grands nombres sont atteintes plus rapidemment. Ainsi, jusqu'à $n = 50000$, le programme s'exécute en moins d'une seconde.
5. Les algorithmes de type "diviser pour régner" sont basés sur une équation de récurrence de même type que ceux utilisant la programmation dynamique. Il est donc possible de résoudre le problème posé à l'aide d'une solution de type "diviser pour régner" en théorie. Cependant, la complexité serait très importante à cause du très grand nombre d'appels récursifs.

6 Conclusion

En somme, nous avons pu expérimenter les différents aspects qui font d'un algorithme un bon algorithme. En effet, nos solutions ont des complexités très différentes, mais le meilleur algorithme nécessite un effort de conception plus important. Il est important de mettre en concurrence ces deux aspects car selon les données à traiter, il n'est pas forcément nécessaire d'avoir le meilleur algorithme qui soit. Nous avons en effet pu voir que sur de petites valeurs, les deux méthodes sont équivalentes en temps d'exécution.