UFR d'Informatique

Paris 7 – Paris Diderot

Année 2015–2016

Notes de cours d'algorithmique – M1

François Laroussinie

francois.laroussinie@liafa.univ-paris-diderot.fr

Page web du cours : http://www.liafa.univ-paris-diderot.fr/~francoisl/m1algo.html

1 Algorithmes Gloutons

Dans cette famille d'algorithmes, on est capable de faire un *bon* choix à chaque étape de l'algorithme qui nous permet de passer à la résolution d'un sous-problème : le choix « localement optimal » conduit à une solution optimale (le choix est donc aussi « globalement optimal »). Notons que la notion d'optimalité renvoie à une vision « problèmes d'optimisation » qui aide à voir la structure de ces algorithmes mais ces algorithmes ne s'appliquent pas seulement à des problèmes d'optimisation.

Ce type d'algorithme n'existe que pour des problèmes bien particuliers où on peut faire un choix localement optimal et où une solution optimale contient des solutions optimales pour des sous-problèmes. Pour prouver la correction de ces algorithmes, on pourra souvent se ramener à ces deux propriétés pour structurer la preuve.

A la différence de la programmation dynamique, c'est une méthode top-down.

1.1 La recherche d'arbres couvrant minimaux dans les graphes valués non-orientés

voir le poly de L3.

1.2 Le problème de l'allocation d'une ressource

On dispose d'une ressource (par exemple, un camion, un processeur,...), et on a un ensemble $\mathcal{E}$ de *requêtes* : chaque requête vise à utiliser cette ressource entre deux dates. **L'objectif est de satisfaire le plus grand nombre de requêtes possible.**

On formalise ce problème de la manière suivante. Une requête sera une paire $(d, f) \in \mathbb{N}\times\mathbb{N}$ avec $d < f$. Un sous-ensemble $\mathcal{F} \subseteq \mathcal{E}$ de requêtes est dit *compatible* lorsqu'une aucune requête n'en intersecte une autre, c'est-à-dire lorsque pour tout $(d_i, f_i), (d_j, f_j) \in \mathcal{F}$, on a soit $f_i \leq d_j$ (la requête i se termine avant le début de la requête j, on dit alors que la requête i précède j) ou $f_j \leq d_i$ (la requête j précède la requête i).

Le problème s'énonce alors comme suit :

Input : un ensemble $\mathcal{E}$ de n requêtes $(d_i, f_i)_{1\leq i\leq n}$ avec $d_i < f_i$ pour tout i.
Output : un sous-ensemble compatible $\mathcal{F} \subseteq \mathcal{E}$ de taille maximale.

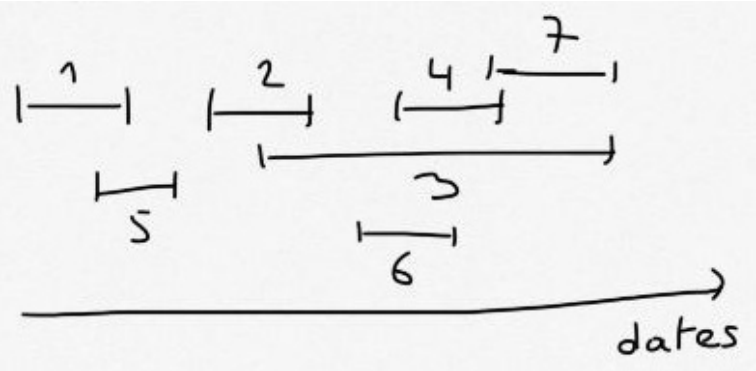

FIGURE 1 – Exemple pour l'allocation d'une ressource

La taille du problème est n.

Dans la suite, on parlera des « solutions optimales pour un ensemble de requêtes $\mathcal{E}$ » pour désigner les sous-ensembles compatibles de taille maximale.

Attention ! Ici l'objectif du problème est de maximiser le nombre de requêtes satisfaites (donc le plus grand nombre de clients si il s'agit d'un camion...). Le problème est très différent lorsqu'il s'agit de maximiser la *durée totale* de location, ou plus généralement d'utilisation de la ressource.

Exemple. Dans l'exemple décrit à la figure 1, l'ensemble $\mathcal{E}$ contient 7 requêtes numérotées de 1 à 7. On utilise la représentation graphique naturelle selon un axe horizontal des dates, chaque requête (*i.e.* ses deux valeurs d et f) sont représentées par un segment. Ici il est clair qu'une solution optimale est le sous-ensemble des requêtes $\{1, 2, 6, 7\}$.

Algorithme. On va utiliser l'algorithme suivant :

Trier l'ensemble $\mathcal{E}$ par ordre de date de fin croissante (donc tel que $f_i \leq f_{i+1}$)
$\mathcal{F} = \emptyset$
Pour $i = 1, \ldots, n$:
..... Ajouter la requête (d_i, f_i) à $\mathcal{F}$ si $\mathcal{F} \cup \{(d_i, f_i)\}$ est compatible.
Retourner $\mathcal{F}$

On verra ci-dessous pourquoi cet algorithme est correct mais avant on peut préciser un point : comment tester la compatibilité de $\mathcal{F} \cup \{(d_i, f_i)\}$? En fait, puisque l'ensemble $\mathcal{E}$ a été trié par f croissante, il suffit de comparer d_i avec la date de fin de la dernière requête ajoutée à $\mathcal{F}$: si d_i est supérieur, c'est que la requête (d_i, f_i) commence après la fin de la dernière requête ajoutée à $\mathcal{F}$, le nouvel ensemble est donc bien compatible. Cela donne donc l'algorithme suivant :

Trier l'ensemble $\mathcal{E}$ par ordre de date de fin croissante (donc tel que $f_i \leq f_{i+1}$)
$\mathcal{F} = \emptyset$
$\mathsf{aux} = 0$
Pour $i = 1, \ldots, n$:
..... Si $\mathsf{aux} \leq d_i$ Alors :
..... $\mathcal{F} += \{(d_i, f_i)\}$
..... $\mathsf{aux} = \mathsf{f_i}$
Retourner $\mathcal{F}$

La complexité de l'algorithme est donc $O(n \cdot \log n)$ pour la phase de tri et $O(n)$ pour le reste... C'est donc un algorithme très efficace.

Il reste à prouver sa correction ! Il est clair qu'il renvoie bien un sous-ensemble compatible (c'est un invariant direct de l'algorithme) mais il faut montrer son optimalité. On va la montrer de deux manières. D'abord avec une preuve *adhoc* et ensuite en repartant des deux propriétés des algorithmes gloutons décrites précédemment...

Preuve de correction 1. On va montrer que l'algorithme donne une solution optimale, c'est-à-dire un sous-ensemble $\mathcal{F}$ de taille maximale. Pour cela, on va supposer que ce n'est pas le cas et donc que la ou les solutions optimales contiennent plus de requêtes que le sous-ensemble renvoyé par l'algorithme. Dans la suite, on note $\mathcal{F} = \{x_1, \ldots, x_p\}$ le sous-ensemble compatible renvoyé par l'algorithme trié par date de fin croissante. Etant donnée la manière dont l'algorithme procède, on sait aussi que x_1 est la première requête choisie par l'algorithme pour $\mathcal{F}$, x_2 est la seconde, x_3 la troisième,...

On suppose à présent qu'une solution optimale comprend q requêtes avec $q > p$. Parmi toutes les solutions optimales, on *choisit celle qui partage le plus des premiers choix fait par l'algorithme.* Soit $\mathcal{G}$ cette solution optimale $\{y_1, y_2, \ldots, y_q\}$ aussi triée par date de fin croissante. On a donc $d(y_1) < f(y_1) \leq d(y_2 \leq f(y_2) < \ldots$ où $d(y_i)$ désigne la date de début de la requête y_i et $f(y_i)$ sa date de fin.

$\mathcal{G}$ peut aussi s'écrire $\{x_1, \ldots, x_k, y_{k+1}, \ldots, y_q\}$ et $x_{k+1} \neq y_{k+1}$: k est le nombre de premiers choix de l'algorithme partagés par $\mathcal{G}$. En effet, il ne peut pas y avoir une requête (d, f) dans $\mathcal{G}$ qui ne soit pas dans $\mathcal{F}$ avec une date de fin f strictement inférieure à celles de la requête x_k car cette requête (d, f) aurait été examinée par l'algorithme avant x_k et aurait donc été ajoutée à $\mathcal{F}$ (son appartenance à $\mathcal{G}$ montre qu'elle est bien compatible avec les premiers choix). La représentation de la solution optimale $\mathcal{F}$ par date de fin croissante contient donc bien les k premiers choix faits par l'algorithme en tête de liste, et le $k+1$-ème choix diffère : x_{k+1} n'est pas dans $\mathcal{F}$. Notons enfin que par hypothèse sur le choix de $\mathcal{F}$, il n'existe pas de solutions optimales qui contiennent les requêtes $x_1, \ldots, x_{k+1}$: $\mathcal{G}$ est la solution qui partage le plus de premiers choix de l'algorithme.

Il faut aussi ajouter que $k < p$ car si $k = p$ alors l'algorithme aurait aussi essayer d'ajouter les requêtes y_{k+1}, y_{k+2},... et que certaines d'entre elles auraient été ajoutées puisqu'elles sont compatibles avec les premières requêtes choisies... Quelques remarques :

- Il se peut qu'il y ait d'autres requêtes contenues dans $\mathcal{F}$ aussi présentes dans $\mathcal{G}$ mais il s'agit alors de choix faits après la $k+1$-ème étape de l'algorithme (donc après que les choix aient divergé).
- Les requêtes x_1, ..., x_k sont communes à $\mathcal{F}$ et $\mathcal{G}$ mais nous ne savons pas quand ils ont été choisies lors de la construction de $\mathcal{G}$: nous n'avons aucune hypothèse sur la construction de $\mathcal{G}$.

Maintenant on définit $\mathcal{G}'$ par $\mathcal{G}\backslash\{y_{k+1}\} \cup \{x_{k+1}\}$. On voit alors que $\mathcal{G}'$ est un sous-ensemble compatible puisque si l'algorithme choisit la requête x_{k+1}, c'est qu'elle est la requête ayant une date de fin minimale parmi celles compatibles avec $\{x_1, \ldots, x_k\}$, et donc la requête y_{k+1} a une date de fin supérieure ou égale à celle de x_{k+1}, et donc remplacer y_{k+1} par x_{k+1} ne pose pas de problème pour les autres requêtes de $\mathcal{G}$: on a bien $f(x_{k+1}) \leq d(y_{k+2})$ (car $f(x_{k+1}) \leq f(y_{k+1})$).
$\mathcal{G}'$ est donc un sous-ensemble compatible et de même taille que $\mathcal{G}$, c'est donc une solution optimale et il partage un premier choix de l'algorithme de plus que $\mathcal{G}$, on a donc une contradiction avec les hypothèses de départ.

Preuve de correction 2. Le premier choix de l'algorithme consiste à prendre une requête (d, f) ayant une date de fin minimale puis à passer à l'itération suivante où il va chercher la prochaine requête ayant une date de fin minimale compatible avec le premier choix, *etc.* On voit alors la forme du sous-problème qui va nous intéresser : étant donné un problème pour un ensemble E de requêtes, on peut voir l'algorithme comme choisissant une requête e et passant ensuite à la résolution du problème défini par l'ensemble E *restreint aux requêtes compatibles avec* e.

Pour cette seconde preuve, on procède en montrant d'abord que l'algorithme fait un « bon » premier choix, c'est à dire un choix qui permettra *in fine* d'obtenir un sous-ensemble optimal : autrement dit, un « bon choix » est un choix inclus dans une solution optimale.

Lemme 1 *Soit $\mathcal{E}$ un ensemble de requête et soit e une requête de $\mathcal{E}$ ayant une date de fin minimale. Il existe une solution optimale avec la requête e.*

Preuve : Soit $\mathcal{G} = \{y_1, \ldots, y_q\}$ une solution optimale, où les y_i sont triés par date de fin croissante, et donc telle que $d(y_i) < f(y_i) \leq d(y_{i+1}) \ldots$ Etant donné l'algorithme, on sait que $f(e) \leq f(y_1)$, d'où $f(e) \leq d(y_2)$ et donc le sous-ensemble $\{e, y_2, \ldots, y_q\}$ est compatible et constitue aussi une solution optimale qui contient le premier choix de l'algorithme. □

Lemme 2 *Soit $\mathcal{E}$ un ensemble de requêtes. Soit $\mathcal{G}$ une solution optimale pour $\mathcal{E}$ et soit $e \in \mathcal{G}$. Alors $\mathcal{G}' = \mathcal{G}\backslash\{e\}$ est une solution optimale pour les requêtes de $\mathcal{E}$ compatibles avec e, c'est-à-dire pour l'ensemble de requêtes $\mathcal{E}' = \{e' \in \mathcal{E} \mid d(e') \geq f(e) \vee d(e) \geq f(e')\}$.*

Preuve : Soit $|\mathcal{G}| = k$ et donc $|\mathcal{G}'| = k - 1$.
Supposons que le lemme soit faux. Alors il existe $\mathcal{G}''$ une solution optimale pour $\mathcal{E}'$ avec $|\mathcal{G}''| > |\mathcal{G}'| = k - 1$. Il suffit de prendre $\mathcal{G}'' \cup \{e\}$ pour obtenir un sous-ensemble compatible (car par hypothèse $\mathcal{G}''$ est compatible avec e) pour $\mathcal{E}$ et il est de cardinal strictement supérieur à k, donc $\mathcal{G}$ n'est pas optimal, et cela contredit les hypothèses de départ. □

Il reste à en tirer le théorème de correction... Pour cela il faut noter qu'une itération de l'algorithme consiste à choisir une requête ayant une date de fin minimale puis à passer à l'étape suivante où il va chercher une solution pour l'ensemble des requêtes compatibles avec son choix précédent, *etc.*

La preuve de correction se fait alors par induction sur la taille n de $\mathcal{E}$:

— $n = 1$: l'algorithme renvoie l'unique requête et c'est bien évidemment la solution optimale !

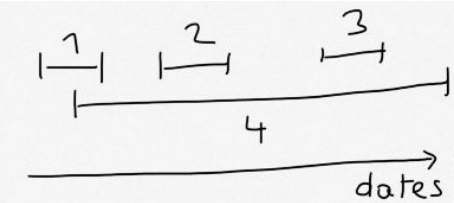

FIGURE 2 – Exemple pour l'allocation d'une ressource

— $n+1$: Soit $\mathcal{F}$ la solution renvoyée par l'algorithme, et soit e la première requête choisie par l'algorithme. On sait par le lemme 1, qu'il existe une solution optimale $\mathcal{G}$ contenant e. Soit $k = |\mathcal{G}|$. Par le lemme 2, on sait que $\mathcal{G}' = \mathcal{G}\backslash\{e\}$ est une solution optimale pour l'ensemble $\mathcal{E}'$ des requêtes compatibles avec e (donc avec $\mathcal{E}' = \{e' \in \mathcal{E} \mid d(e') \geq f(e) \vee d(e) \geq f(e')\}$). Et on a $|\mathcal{G}'| = k-1$.
A quoi correspond l'ensemble $\mathcal{F}$ renvoyé par l'algorithme ? A $\{e\} \cup \mathcal{F}'$ où $\mathcal{F}'$ est l'ensemble renvoyé par l'algorithme pour l'ensemble $\mathcal{E}'$. Mais par hypothèse d'induction, on sait que l'algorithme est correct pour $\mathcal{E}'$ (car $|\mathcal{E}'| \leq n$) et donc $|\mathcal{F}'| = |\mathcal{G}'| = k-1$ et donc $|\mathcal{F}| = k$ et c'est donc une solution optimale.

1.2.1 Variante : maximiser la durée totale d'allocation de la resource

Le problème est fort différent lorsque l'objectif n'est pas de maximiser le nombre de requêtes mais de **maximiser le temps total d'occupation de la ressource**. Dans l'exemple de la figure 2, une solution optimale est clairement l'unique requête 2 (et non les trois autres !).

Pour ce problème, il n'existe pas d'algorithme glouton comme dans le cas précédent. Si on voulait suivre le schéma précédent, on pourrait essayer de trier les requêtes par durée décroissante et choisir la plus longue, puis la deuxième plus longue qui est compatible, *etc.* Mais cet algorithme n'est pas correct : un exemple simple pour le montrer est l'ensemble de requête $\{(10,30),(5,20),(25,40)\}$, choisir la première requête (la plus longue) élimine les deux autres et donne une durée totale de 20, mais choisir les deux dernières (qui sont compatibles entre elles) permet d'obtenir une durée totale de 30 !

Ici un algorithme efficace est possible en réduisant ce problème à la recherche d'un plus long chemin dans un graphe orienté valué acyclique. Etant donné $\mathcal{E}$ un ensemble de n requêtes $\{e_1, \ldots, e_n\}$, on note $\mathbb{T}$ l'ensemble de toutes les dates apparaissant dans les requêtes de $\mathcal{E}$ avec 0 : $\mathbb{T} = \{t_0, t_1, \ldots, t_p\}$ tel que (1) $t_0 = 0$, (2) $t_i < t_{i+1}$ pour $0 \leq i < p$, (3) pour tout $i \in \{1, \ldots, p\}$, il existe $e = (d, f) \in \mathcal{E}$ telle que $t_i = d$ ou $t_i = f$, et (4) pour tout $e = (d, f) \in \mathcal{E}$ on a $d \in \mathbb{T}$ et $f \in \mathbb{T}$.

Maintenant on définit le graphe $G_{\mathcal{E}} = (S, A, w)$ avec :

— $S = \mathbb{T}$,

— $A = \{(d, f) \mid \exists (d, f) \in \mathcal{E}\} \cup \{(t_i, t_{i+1}) \mid 0 \leq i \leq p-1\}$

— $w(t, t') = \begin{cases} t' - t & \text{si } \exists (t, t') \in \mathcal{E} \\ 0 & \text{sinon} \end{cases}$

Exemple. On considère à présent l'exemple d'ensemble de requêtes de la figure 3 et son graphe $G_{\mathcal{E}}$ associé à la figure 4.

Quelle est l'idée de ce codage ? On représente les requêtes comme des arcs valués avec leur durée. Ensuite on autorise aussi les arcs consistant à laisser passer du temps et donc passer de

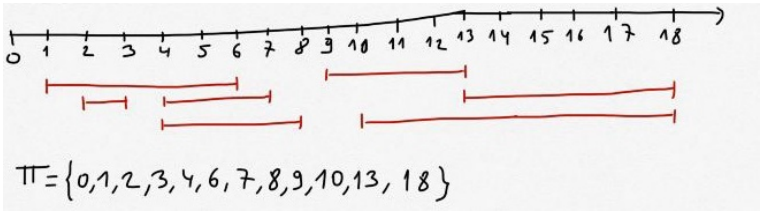

FIGURE 3 – Exemple pour le graphe $G_{\mathcal{E}}$

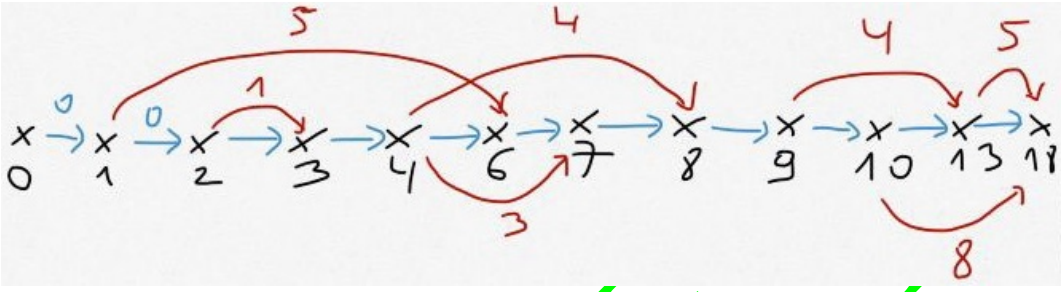

FIGURE 4 – Exemple de graphe $G_{\mathcal{E}}$

t_i à t_{i+1} mais dans ce cas, il n'y a pas d'utilisation de la ressource, et le coût est donc nul. Un chemin de durée maximale dans le graphe entre t_0 et t_p est donc un chemin où la durée totale d'utilisation de la ressource est maximale ! Pour que cela fournisse un algorithme efficace, il reste à remarquer que le graphe $G_{\mathcal{E}}$ est un graphe acyclique (condition pour que la recherche des plus longs chemins soit facile – d'un point de vue de complexité – à effectuer).

Dans l'algorithme ci-dessous, on va calculer la plus longue durée et aussi désigner un chemin correspondant à cette durée, de la même manière que les algorithmes de plus courts chemins dans les graphes.

Pour trouver la durée d'un plus long chemin entre t_0 et t_p dans $G_{\mathcal{E}}$, il suffit d'utiliser l'algorithme suivant :

$\mathsf{DMax}[t_p] = 0$
$\mathsf{Succ}[t_p] = nil$
Pour $t = t_{p-1}, t_{p-2}, \ldots, t_0$:
..... $\mathsf{DMax}[t] = -\infty$
..... Pour tout $(t, t') \in A$:
..... Si $\mathsf{DMax}[t] < \mathsf{DMax}[t'] + w(t, t')$ Alors :
..... $\mathsf{DMax}[t] = \mathsf{DMax}[t'] + w(t, t')$
..... $\mathsf{Succ}[t] = t'$
Retourner $\mathsf{DMax}, \mathsf{Succ}$

Remarque : Dans l'algorithme ci-dessus, les tableaux DMax et Succ sont indexés par les dates de $\mathbb{T}$ et non par la valeur max... Sinon la mémoire utilisée serait exponentielle dans l'entrée du problème (les valeurs numériques sont supposées être codées en binaire). De manière équivalente, on peut utiliser l'algorithme suivant :

$\mathsf{DMax}[p] = 0$
$\mathsf{Succ}[p] = nil$
Pour $i = p-1, p-2, \ldots, 0$:
..... $\mathsf{DMax}[i] = -\infty$
..... Pour tout $(t_i, t_j) \in A$:
..... Si $\mathsf{DMax}[i] < \mathsf{DMax}[j] + w(t_i, t_j)$ Alors :
..... $\mathsf{DMax}[i] = \mathsf{DMax}[j] + w(t_i, t_j)$
..... $\mathsf{Succ}[i] = j$
Retourner $\mathsf{DMax}, \mathsf{Succ}$

Quelle complexité ? La construction et le tri de $\mathbb{T}$ demande un temps en $O(n \cdot \log n)$, et la taille de $\mathbb{T}$ est inférieure à $2n+1$. Le graphe $G_{\mathcal{E}}$ est de taille en $O(n)$: son ensemble de sommet est $\mathbb{T}$ et son nombre d'arêtes est $n + (|\mathbb{T}| - 1)$, et sa construction se fait bien en $O(n)$. Quant à l'algorithme sur $G_{\mathcal{E}}$, il est clairement en $O(|S| + |A|)$ si le graphe est représenté sou forme d'une liste d'adjacence. Cela donne *in fine* une complexité globale en $O(n \cdot \log n)$.