Glouton et dynamique

Yann Strozecki
yann.strozecki@uvsq.fr

Septembre 2016

Objectif

Dans ce cours on va étudier les méthodes qui permettent de concevoir des algorithmes de bonne complexité.

Nous allons illustrer par des exemples les algorithmes gloutons, diviser pour régner et la programmation dynamique.

Ordonnancement d'intervalles

Imaginons que nous voulons faire l'emploi du temps d'une salle. On nous fournis des demandes de réservation $\{r_1, \ldots r_n\}$, de temps de début et de fin $\{(d(1), f(1)), \ldots (d(n), f(n))\}$.

Deux réservations sont compatibles si elles ne s'intersectent pas.

Le problème est de trouver un sous-ensemble de réservations compatibles deux à deux, qui soit de taille maximale.

Des algorithmes gloutons

Proposer une technique pour résoudre le problème précédent.

On peut imaginer plusieurs méthodes gloutonnes, c'est à dire qui font un choix "optimal" élément par élément pour construire la solution :

- prendre la réservation qui commence le plus tôt
- prendre la plus petite réservation
- prendre la réservation avec le moins de conflits
- prendre la réservation qui finit le plus tôt

Preuve de correction

Il est clair que les algorithmes gloutons donnent une solution correcte.
Mais est-elle optimale ?

On suppose que l'algorithme glouton construise la solution $r_1, \ldots, r_k$ et on considère une autre solution $s_1, \ldots, s_l$ (les intervalles sont ordonnés du plus petit au plus grand).

Lemma

Pour tout indice $i \leq k$, $f(r_i) \leq f(s_i)$.

Preuve par induction.

Preuve de correction

Il est clair que les algorithmes gloutons donnent une solution correcte.
Mais est-elle optimale ?

On suppose que l'algorithme glouton construise la solution $r_1, \ldots, r_k$ et on considère une autre solution $s_1, \ldots, s_l$ (les intervalles sont ordonnés du plus petit au plus grand).

Lemma

Pour tout indice $i \leq k$, $f(r_i) \leq f(s_i)$.

Preuve par induction.

Preuve de correction (2)

Par contradiction : si la solution $s_1, \ldots, s_l$ est meilleure que $r_1, \ldots, r_k$ alors $l > k$.
On applique le lemme : $f(r_k) \leq f(s_k)$. Donc si on peut prolonger la solution s, on peut prolonger de la même manière la solution $r \rightarrow$ contradiction.

Quelle est la complexité de notre algorithme ?
$O(n \log(n))$

Les gloutons sont partouts

On trouve des algorithmes gloutons dans de nombreux contextes :

1. Le rendu de monnaie.
2. La construction d'arbre couvrants minimaux.
3. L'algorithme de Huffman pour la compression.
4. Des algorithmes d'approximation.

Ordonnancement bis

On généralise le problème d'ordonnancemment précédent en affectant à chaque réservation une valeur, les données sont donc $\{(d(1), f(1), v(1)), \ldots (d(n), f(n), v(n))\}$.

Le problème est maintenant de trouver un sous-ensemble S de réservations de valeur maximale, c'est à dire $\sum_{i \in S} v(i)$.

Une idée de solution ?

Comment découper le problème

On suppose les réservations triées par ordre de fin croissant, $(r_1 \ldots, r_n)$.

On fait le constant suivant, soit :

- 1. la solution optimale ne contient pas r_n
  2. la solution optimale est une solution optimale de $(r_1, \ldots, r_{n-1})$
- 1. la solution optimale contient r_n

Comment découper le problème

On suppose les réservations triées par ordre de fin croissant, $(r_1 \ldots, r_n)$.

On fait le constant suivant, soit :

- 1. la solution optimale ne contient pas r_n
  2. la solution optimale est une solution optimale de $(r_1, \ldots, r_{n-1})$
- 1. la solution optimale contient r_n
  2. la solution optimale ne contient aucune réservation qui finit après $d(n)$

Comment découper le problème

On suppose les réservations triées par ordre de fin croissant, $(r_1 \ldots, r_n)$.

On fait le constant suivant, soit :

- 1. la solution optimale ne contient pas r_n
  2. la solution optimale est une solution optimale de $(r_1, \ldots, r_{n-1})$
- 1. la solution optimale contient r_n
  2. la solution optimale ne contient aucune réservation qui finit après $d(n)$
  3. la solution optimale moins r_n est une solution optimale pour $r_1, \ldots, r_{p(n)}$ où $r_{p(n)}$ est la plus grande réservation dont la fin est plus petite que d_n

Comment découper le problème

On suppose les réservations triées par ordre de fin croissant, $(r_1 \ldots, r_n)$.

On fait le constant suivant, soit :

- 1. la solution optimale ne contient pas r_n
  2. la solution optimale est une solution optimale de $(r_1, \ldots, r_{n-1})$
- 1. la solution optimale contient r_n
  2. la solution optimale ne contient aucune réservation qui finit après $d(n)$
  3. la solution optimale moins r_n est une solution optimale pour $r_1, \ldots, r_{p(n)}$ où $r_{p(n)}$ est la plus grande réservation dont la fin est plus petite que d_n

Établir une équation de récurrence

On note O_j une solution optimale pour $r_1, \ldots, r_j$ de valeur $OPT(j)$.

On a établit que :

Formule

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j-1))$$

Que vérifie la solution optimale ?

Déroulement du calcul

On suppose qu'on a déjà calculé $p(j)$ pour tous les j.

Algorithme 1 : CalculOPT

Data : V : Tableau d'entiers, P : Tableau d'entiers, j entier

if *j==0* **then**
  return 0
else
  return max(CalculOPT(P[j])+ V[j], CalculOPT(j-1))

Mémoïsation

Comment éviter de calculer un nombre exponentiel de sous-problèmes ?

On remarque qu'il n'y a que n sous-problèmes différents. On stocke la valeur d'un sous-problème quand on l'a calculé et quand on doit le recalculer on se contente de lire cette valeur.

Cette méthode, la mémoïsation, permet d'avoir des algorithmes de programmation dynamique polynomiaux.