Algorithmique — 2. Techniques algorithmiques

9. Algorithmes d'approximation

Bruno Grenet

https://membres-ljk.imag.fr/Bruno.Grenet/Algorithmique.html

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Table des matières

Table des matières

Définition du problème

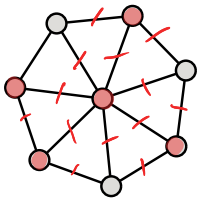
Couverture *Vertex Cover*

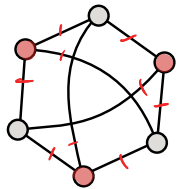
Entrée : Un graphe $G = (S, A)$

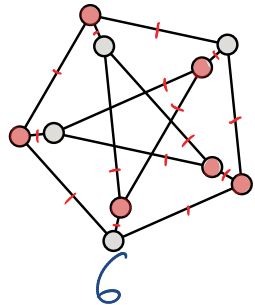
Sortie : Un sous-ensemble $C \subset S$ de sommets, qui *couvre* toutes les arêtes :

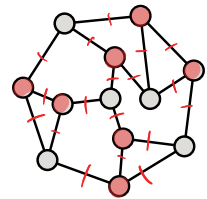
$$\forall \{u, v\} \in A, u \in C \text{ ou } v \in C$$

Objectif : Trouver C le plus petit possible









Solution exacte

Algorithme par recherche exhaustive

- Tester tous les sous-ensembles possibles, par taille croissante
- Complexité : $O(2^n n^2)$ où n est le nombre de sommets
  - $O(2^k n^2)$ si la couverture minimale est de taille k

A priori pas d'algorithme polynomial

- Couverture fait partie des problèmes NP-complets
- Meilleurs algorithmes connus en $O(2^k n)$, voire $O(1,2738^k + kn)$

en master *difficile !*

Que peut-on faire en *temps polynomial* ?

Un algorithme d'approximation

On ne cherche plus la couverture la plus petite possible mais *une couverture assez petite*

CouvApprox(G) :

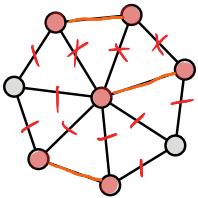
1. $C \leftarrow \emptyset$
2. Tant que G est non vide :
3. Choisir une arête $\{u, v\}$ dans G
4. Ajouter u **et** v dans C
5. Supprimer u et v (et les arêtes incidentes) de G
6. Renvoyer C

Complexité

L'algorithme CouvApprox a une complexité $O(n^2)$

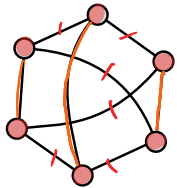
Boucle Tant que exécutée $\lfloor \frac{n}{2} \rfloor$ fois $\rightarrow \Theta(n)$
Corps de la boucle en $\Theta(n)$
$\} \Theta(n^2)$

Exemples



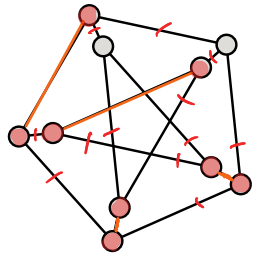
taille 6

optimal 5



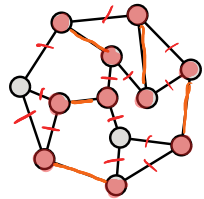
6

3



8

6



10

7

Garantie de l'algorithme d'approximation

Théorème

Soit OPT la taille d'une couverture de taille minimale de G, et $C \leftarrow \textsc{CouvApprox}(G)$. Alors

$$\#C \leq 2\,\text{OPT}$$

Preuve

- Soit B l'ensemble des arêtes choisies par CouvApprox
- Toutes les arêtes de B doivent être couvertes dans une solution optimale
  $\Rightarrow$ au moins 1 extrémité doit être dans la solution
- Comme les arêtes de B n'ont pas d'extrémité en commun, on a que

$$\text{OPT} \geq \#B = \frac{1}{2}\#C \qquad (\Leftrightarrow 2\,\text{OPT} \geq \#C)$$

Table des matières

Définition du problème

Somme Partielle — *Subset Sum*

Entrée : Un tableau T de n entiers strictement positifs, un entier cible c

Sortie : Un sous-ensemble $I \subset \{0, \dots, n-1\}$ tel que $\sum_{i \in I} T_{[i]} \leq c$

Objectif : Trouver I avec la somme la plus grande possible (la plus proche possible de c)

Notations

- I_{OPT} : meilleure solution possible
- $\text{OPT} = \sum_{i \in I_{\text{OPT}}} T_{[i]}$: valeur atteinte par $I_{\text{OPT}} \rightarrow$ *cible*

$T = [1, 7, 28, 3, 9, 41, 11, 8]$

$c = 16$: $I = \{1, 4\}$ car $T_{[1]} + T_{[4]} = 9 + 7 = 16$

$c = 30$: $7 + 3 + 9 + 11 = 30$

$c = 33$: $1 + 28 + 3 = 32$

Solution exacte

Recherche exhaustive et *backtrack*

- Parcours de tous les sous-ensembles $I \subset \{0, \dots, n-1\}$ — Livret TD2 Ex. 8
  - Complexité $O(n2^n)$
- *Backtrack* si entiers tous positifs
  - Complexité $O(2^n)$

A priori pas d'algorithme polynomial

- Somme Partielle fait partie des problèmes NP-complets
- Meilleur algorithme connu en $O(2^{n/2}) = O(1,414^n)$ — *difficile !*

Que peut-on faire en *temps polynomial* ?

Un algorithme d'approximation

Idée

- Prendre les éléments par valeur décroissante
- Sélectionner tous ceux qu'on peut

SommePartApprox(T, c) :

1. Trier T par ordre décroissant
2. $I \leftarrow \emptyset$
3. Pour $i = 0$ à $\#T - 1$:
4. Si $c \geq T_{[i]}$:
5. Ajouter i à I
6. $c \leftarrow c - T_{[i]}$
7. Renvoyer I

[41, 28, 11, 9, 8, 7, 3, 1]

opt

16 : 11 + 3 + 1 = 15 — 16

30 : 28 + 1 = 29 — 30

33 : 28 + 3 + 1 = 32 — 32

Complexité

L'algorithme SommePartApprox a une complexité $O(n \log n)$

Garantie de l'algorithme d'approximation

Théorème

Soit $I \leftarrow \textsc{SommePartApprox}(T, c)$ et OPT la valeur de la solution optimale. Alors

$$\sum_{i \in I} T_{[i]} \geq \frac{1}{2} \text{OPT}$$

Preuve

- On élimine de T les éléments $> c$.
- Si $I = \{0, \dots, \#T - 1\}$ $\left(\Leftrightarrow \sum_{i=0}^{\#T-1} T_{[i]} \leq c\right)$, alors $I = I_{OPT}$. On suppose que ce n'est pas le cas.
- Soit i le 1er indice non sélectionné.

$T_{[0]} \geq T_{[1]} \geq \dots \geq T_{[i-1]} \geq T_{[i]}$ et $\sum_{j=0}^{i-1} T_{[j]} + T_{[i]} > c$

$\Rightarrow T_{[i]} \leq \sum_{j=0}^{i-1} T_{[j]}$ donc $2 \sum_{j=0}^{i-1} T_{[j]} > c$. Or $\begin{cases} c \geq \text{OPT} \\ \sum_{j \in I} T_{[j]} \geqslant \sum_{j=0}^{i-1} T_{[j]} \end{cases} \Rightarrow \sum_{j \in I} T_{[j]} \geq \frac{\text{OPT}}{2}$

Table des matières

Problèmes d'optimisation

Cadre général : deux types d'optimisation

Max : sur une entrée, trouver une solution qui *maximise* une certaine fonction

Min : sur une entrée, trouver une solution qui *minimise* une certaine fonction

Exemples

MAX : SOMME PARTIELLE

MIN : COUVERTURE

Formalisation des algorithmes d'approximation

Ingrédients

- Ensemble I des instances — *entrées*
- Pour chaque $x \in I$, l'ensemble S des solutions *acceptables* — *sorties possibles*
- Une fonction de coût $c : S \to \mathbb{R}$ — *valeur d'une solution*

Objectifs

(max) Trouver $s \in S$ telle que $c(s)$ soit maximale — $\forall s' \in S, c(s') \leq c(s)$

(min) Trouver $s \in S$ telle que $c(s)$ soit minimale — $\forall s' \in S, c(s') \geq c(s)$

Valeur optimale

OPT : valeur de la solution optimale

(max) $\text{OPT} = \max_{s \in S} c(s)$

(min) $\text{OPT} = \min_{s \in S} c(s)$

COUVERTURE : $c(S) = \#$ sommets

SOUS-PARTITION : $c(S) =$ valeur de la somme

Résolution exacte

Recherche exhaustive et *backtrack* — Cours 6

- Parcours (intelligent) de toutes les solutions, en gardant la meilleure
- Fonctionne toujours ; complexité (en général) exponentielle

Programmation dynamique — Cours 8

- Décomposition du problème en sous-problèmes, et résolution par tailles croissantes
- Fonctionne souvent ; complexité (en général) exponentielle mais meilleure qu'en recherche exhaustive

Autres techniques

- Algorithmes gloutons, recherche locale
- Inclusion-exclusion
- Algorithmes paramétrés
- Compromis temps-mémoire
- …

Algorithmes d'approximation

Algorithmes de compromis

- Algorithmes efficaces → complexité polynomiale, voire linéaire
- Algorithmes non exacts → solution de valeur proche de l'optimal

Définition

- Un **algorithme d'α-approximation** est un algorithme qui *pour tout entrée x* renvoie une solution $s \in S$ telle que

  (max) $\alpha \cdot \text{OPT} \leq c(s) \leq \text{OPT}$ $\quad 0 < \alpha < 1$

  (min) $\text{OPT} \leq c(s) \leq \alpha \cdot \text{OPT}$ $\quad \alpha > 1$

- Le réel α est appelé **facteur d'approximation** de l'algorithme.

Exemples

CouvApprox : $\alpha = 2$

SommetApprox : $\alpha = \frac{1}{2}$

Comment concevoir des algorithmes d'approximation ?

Très vaste sujet, dépasse (très) largement le cadre de ce cours !

Une technique fructueuse : algorithmes gloutons

- Exemple : Somme Partielle
  - choix des entiers par ordre décroissant
  - on sélectionne chaque entier si c'est possible, sans retour en arrière
- Caractéristiques :
  - souvent rapide $\rightarrow$ efficacité
  - pas toujours optimal $\rightarrow$ non exact
  - souvent *pas trop mauvais* $\rightarrow$ compromis

Remarque

- On ne cherche pas une solution *optimale*, mais *pas trop mauvaise*
- Parfois intéressant de faire des choix *un peu bêtes* mais pas loin de l'optimal
  - Exemple de Couverture : ajouter les 2 extrémités de l'arête choisie

Comment analyser un algorithme d'approximation ?

Objectif

Montrer que pour tout entrée, l'algorithme renvoie une solution s vérifiant

(max) $c(s) \geq \alpha \cdot \textsc{opt}$

(min) $c(s) \leq \alpha \cdot \textsc{opt}$

Deux bornes à trouver

- Borne c_1 sur le résultat renvoyé
- Borne c_2 sur le résultat optimal

$\Rightarrow$ Borne sur le facteur d'approximation

(max)	(min)
$c(s) \geq c_1$	$c(s) \leq c_1$
$\textsc{opt} \leq c_2$	$\textsc{opt} \geq c_2$
$\alpha \geq c_1/c_2$	$\alpha \leq c_1/c_2$

Pour trouver le facteur d'approximation, il faut aussi une borne sur la valeur optimale !

Table des matières

Définition du problème

Informellement

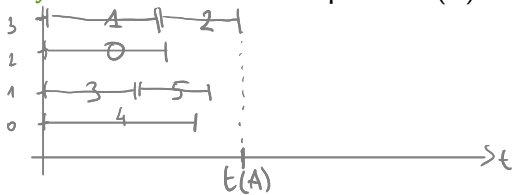
- Ensemble de *n tâches* à exécuter, chacune ayant une *durée*
- À disposition : *m processeurs*
- Objectif : répartir les tâches sur les processeurs, pour *minimiser* le temps total

Équilibrage — *Load Balancing*

Entrées : Tableau D de n entiers strictement positifs — *durées*
Entier m — *nombre de processeurs*

Sortie : Tableau A : affectation de chaque tâche à un processeur
$\rightarrow A_{[i]} = j$: « tâche i affectée au processeur j »

Objectif : Minimiser le temps total $t(A) = \max_{0 \leq j \leq m-1} \left(\sum_{i : A_{[i]} = j} D_{[i]} \right)$



Algorithme glouton *à la volée*

Scénario : les tâches arrivent les unes après les autres, on doit les traiter dans l'ordre

- Traduction : on ne peut pas trier le tableau D
- Idée de l'algo. : on affecte la prochaine tâche au processeur le moins occupé

ÉquilibrageGlouton(D, m) :

1. $T \leftarrow$ tableau de taille m, initialisé à 0
2. Pour $i = 0$ à $n - 1$:
3. $\quad j \leftarrow$ indice d'un minimum de T
4. $\quad A_{[i]} \leftarrow j$
5. $\quad T_{[j]} \leftarrow T_{[j]} + D_{[i]}$
6. Renvoyer A

$T_{[j]}$: *temps total du processeur* j

Complexité

L'algorithme ÉquilibrageGlouton a une complexité $O(nm)$
(ou $O(n \log m)$ en remplaçant T par une file de priorité)

Garantie de l'algorithme glouton

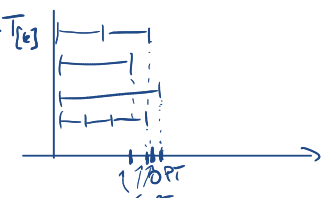
Théorème

L'algorithme ÉquilibrageGlouton est un algorithme de 2-approximation pour le problème Équilibrage

Preuve

- $OPT \geq \max_i D_{[i]}$ car la tâche la plus longue doit affecté à un processeur.
- $OPT \geq \frac{1}{m} \sum_{i=0}^{n-1} D_{[i]}$ car $\sum_{i=0}^{n-1} D_{[i]} = \sum_{j=0}^{m-1} T^{OPT}_{[j]} \leq m \cdot OPT$.

- Soit A l'affectation renvoyée par l'algo et j tq $T_{[j]} = \max_k T_{[k]}$
  Soit i la dernière tâche affectée au proc. j.
  Alors $T_{[j]} - D_{[i]} \leq T_{[k]}$ pour tout k



Donc $T_{[j]} - D_{[i]} \leq \frac{1}{m} \sum_{k=0}^{m-1} T_{[k]} = \frac{1}{m} \sum_{i=0}^{n-1} D_{[i]} \leq OPT$. et $D_{[i]} \leq OPT$.

Donc $T_{[j]} \leq 2OPT$.

Algorithme glouton avec tri

Nouveau scénario : on connaît toutes les tâches à l'avance → fait-on mieux ?

- Idée : affectation des tâches les plus longues en premier

Algorithme et complexité

- Même algorithme ÉquilibrageGlouton, avec tri de D initialement
- Complexité : $O(n \log n)$ pour le tri, puis pareil
  - $O(n(m + \log n))$ avec recherche *naïve* de minimum
  - $O(n(\log n + \log m))$ avec une file de priorité → $O(n \log n)$ car $n \geq m$

Garanties de l'algorithme glouton *avec tri*

Théorème

Si D est trié par ordre décroissant, ÉquilibrageGlouton a un facteur d'approximation $\leq \frac{3}{2}$

Preuve

- Soit j tq $T[j] = \max_k T[k]$.
- Si le proc. j n'a qu'une tâche, alors le résultat est optimal.
- Sinon, soit i la dernière tâche affectée au proc. j.

(1) $T[j] - D[i] \leq OPT$: cf preuve précédente

(2) $i \geq m$ car les m premières tâches vont à des proc. différents.

$\Rightarrow D[i] \leq D[m]$

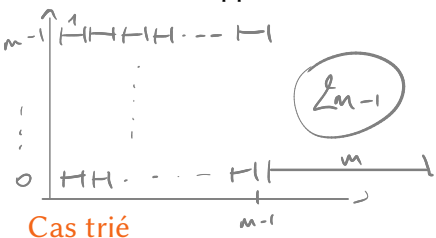
(3) $OPT \geq 2 D[m]$ car 1 proc. reçoit 2 tâches parmi les $(m+1)$ premières

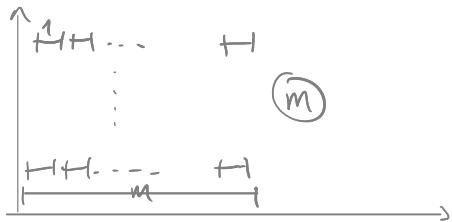
$\Rightarrow D[i] \leq \frac{1}{2} OPT$. $\Rightarrow$ $T[j] \leq OPT + D[i] \leq \frac{3}{2} OPT$

Bilan sur l'équilibrage de charge

Cas non trié

- L'algorithme glouton est une 2-approximation
- Un peu mieux : $(2 - 1/m)$-approximation
- Facteur d'approximation atteint





Cas trié

- L'algorithme glouton fournit une $\frac{3}{2}$-approximation
- On peut dire mieux : $(\frac{4}{3} - \frac{1}{m})$-approximation

meilleure borne sur OPT

Encore mieux ?

- Pour tout $\varepsilon > 0$, il existe un algorithme qui est une $(1 + \varepsilon)$-approximation