Partie 2. Techniques algorithmiques

9. Algorithmes d'approximation

Bruno Grenet

Université Grenoble Alpes – IM²AG

L3 Mathématiques et Informatique

UE Algorithmique

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

Table des matières

1. Exemple 1. Couverture par sommets

2. Exemple 2. Somme partielle

3. Les algorithmes d'approximation

4. Borne sur OPT : exemple de l'équilibrage de charge

Table des matières

1. Exemple 1. Couverture par sommets

2. Exemple 2. Somme partielle

3. Les algorithmes d'approximation

4. Borne sur OPT : exemple de l'équilibrage de charge

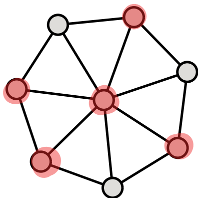
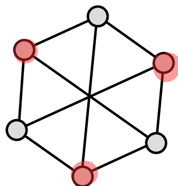
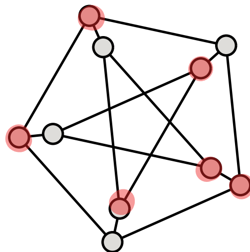
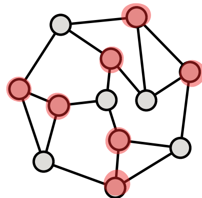
Définition du problème

Couverture — *Vertex Cover*

Entrée : Un graphe $G = (S, A)$

Sortie : Un sous-ensemble $C \subset S$ de sommets, qui *couvre* toutes les arêtes : pour tout $\{u, v\} \in A$, $u \in C$ 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*

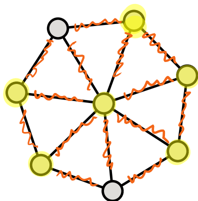
CouvApprox(G) :

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

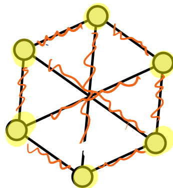
Complexité

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

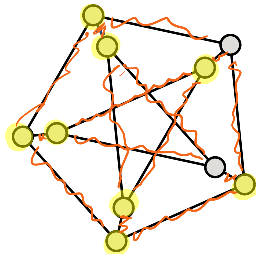
Exemples



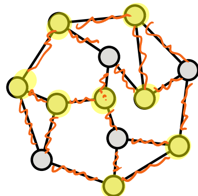
taille 6
au lieu de 5



taille 6
— 3



8
6



8
7

Garantie de l'algorithme d'approximation

Théorème

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

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

- Soit B l'ensemble des arêtes choisies par CouvApprox.
- Les arêtes de B n'ont aucun sommet en commun
- Elles doivent être couvertes dans une solution optimale

Donc $\text{OPT} \geq \#B = \frac{1}{2}\#C$.

Table des matières

1. Exemple 1. Couverture par sommets

2. Exemple 2. Somme partielle

3. Les algorithmes d'approximation

4. Borne sur OPT : exemple de l'équilibrage de charge

Définition du problème

SOMME PARTIELLE *SUBSET SUM*

Entrée : Un ensemble E d'entiers strictement positifs, un entier cible T

Sortie : Un sous-ensemble $S \subset E$ dont la somme est $\leq T$

Objectif : Trouver S de somme la plus grande possible (la plus proche possible de T)

Notations

- S_{OPT} : meilleure solution possible
- $\text{OPT} = \sum_{x \in S_{\text{OPT}}} x$: valeur atteinte par $S_{\text{OPT}} \rightarrow$ *cible*

Solution exacte

Recherche exhaustive et *backtrack*

- Parcours de tous les sous-ensembles $S \subset E$ TD6 Ex. 2
  - Complexité $O(n2^n)$ où $n = \#E$
- *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

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

SommePartApprox(E, T) :

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

T=30 → [28, 1] ⇝ 29 au lieu de [11, 9, 7, 3] ↳ 30

T=33 → [28, 3, 1] ⇝ 32 optimal

T=16 → [11, 3, 1] ⇝ 15 au lieu de [8, 7, 1] ↳ 16

Complexité

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

Garantie de l'algorithme d'approximation

Théorème

Soit $S \leftarrow \textsc{SommePartApprox}(E, T)$ et $\textsc{opt}$ la valeur de la solution optimale. Alors

$$\sum_{x \in S} x > \frac{1}{2}\,\textsc{opt}$$

- On élimine de E les éléments $> T$.
- Si $S = E$: optimal
- On suppose $S \neq E$. On note $E[i]$ le 1er élément non choisi (ordre décroissant)

$$E[0] \geq E[1] \geq \dots \geq E[i-1] \geq E[i]$$

$$\sum_{j<i} E[j] + E[i] > T$$

Donc $E[i] \leq \sum_{j<i} E[j]$ $\longrightarrow$ $2 \cdot \sum_{j<i} E[j] > T$

Donc $\sum_{x \in S} x \geq \sum_{j<i} E[j] > T/2$. Mais $\textsc{opt} \leq T$ donc $\sum_{x \in S} x > \frac{1}{2}\,\textsc{opt}$.

Table des matières

1. Exemple 1. Couverture par sommets

2. Exemple 2. Somme partielle

3. Les algorithmes d'approximation

4. Borne sur OPT : exemple de l'équilibrage de charge

Problèmes d'optimisation

Cadre général : deux types d'optimisation

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

Minimisation : 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

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

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

Valeur optimale

OPT : valeur de la solution optimale

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

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

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 $\rightarrow$ complexité polynomiale, voire linéaire
- Algorithmes non exacts $\rightarrow$ 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

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

(minimisation) $\text{OPT} \leq c(s) \leq \alpha \cdot \text{OPT}$ $\qquad \alpha > 1$

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

Exemples

COUVERTURE : $\alpha \leq 2 \rightsquigarrow \alpha = 2$

SOMME PARTIELLE : $\alpha \geq 1/2$

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

(maximisation) $c(s) \geq \alpha \cdot \text{OPT}$

(minimisation) $c(s) \leq \alpha \cdot \text{OPT}$

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

(maximisation)	(minimisation)
$c(s) \geq c_1$	$c(s) \leq c_1$
$\text{OPT} \leq c_2$	$\text{OPT} \geq c_2$
$\alpha \geq c_1/c_2$	$\alpha \leq c_1/c_2$

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

COUVERTURE : $\text{OPT} \geq 1 \rightarrow c_2$ trivial

SOMME PARTIELLE : $\text{OPT} \leq T$

Table des matières

1. Exemple 1. Couverture par sommets

2. Exemple 2. Somme partielle

3. Les algorithmes d'approximation

4. Borne sur OPT : exemple de l'équilibrage de charge

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

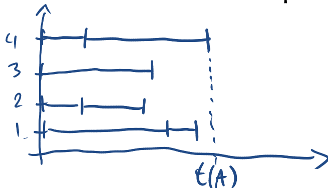
ÉQUILIBRAGE — *LOAD BALANCING*

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

Sortie : Tableau A : affectation de chaque tâche à un processeur
(tâche i affectée au processeur j : $A_{[i]} = j$)

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

4 proc.



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

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

- $OPT \geq \max_{0 \leq i < n} D[i]$ et $OPT \geq \frac{1}{m} \sum_{i=0}^{n-1} D[i]$ car ça serait une répartition parfaitement équilibrée.

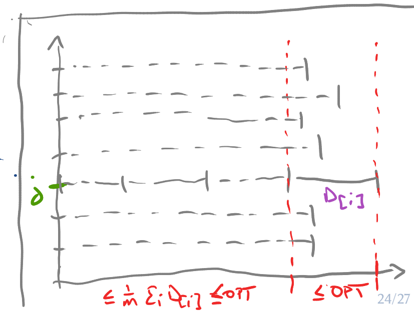
- Soit A l'affectation renvoyée et j tq $T[j] = \max_{0 \leq k < m} T[k]$ $(t(A) = T[j])$

Soit i la dernière tâche affectée au processeur j

- $T[j] - D[i] \leq T[k]$ pour tout k

Donc $T[j] - D[i] \leq \frac{1}{m} \sum_{j=0}^{m-1} T[j] = \frac{1}{m} \sum_{i=0}^{n-1} D[i] \leq OPT$.

$T[j] \leq OPT + D[i] \leq 2\,OPT$.



Algorithme glouton avec tri

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

- Idée : tri des tâches par durée décroissante et 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 un tas $\rightarrow O(n \log n)$ car $n \geq m$

Garanties de l'algorithme glouton *avec tri*

Théorème

Si D est trié par ordre décroissant, le facteur d'approximation α de ÉQUILIBRAGEGLOUTON est $\leq 3/2$

- Soit j le processeur le plus chargé. Si j n'a qu'une tâche, alors l'affectation est optimale.
- Sinon : soit i la dernière tâche affectée à j.
  (1) $T[j] - D[i] \leq OPT$ → preuve précédente
  (2) $i \geq m+1$ car les m premières tâches sont affectées aux m processeurs
  (3) $2D[m+1] \leq OPT$ car dans une solution optimale, 2 tâches $k, \ell \leq m+1$ sont affectées à un même proc. Or $D[k], D[\ell] \geq D[m+1]$ et $D[k] + D[\ell] \leq OPT$.

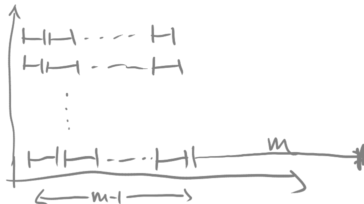
$$\Rightarrow D[i] \underset{(2)}{\leq} D[m+1] \underset{(3)}{\leq} \frac{1}{2} OPT$$

Donc $T[j] \underset{(1)}{\leq} OPT + 1/2\, OPT$.

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 $3/2$-approximation
- On peut dire mieux : $(4/3 - 1/m)$-approximation *meilleure borne sur OPT*

Encore mieux ?

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