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

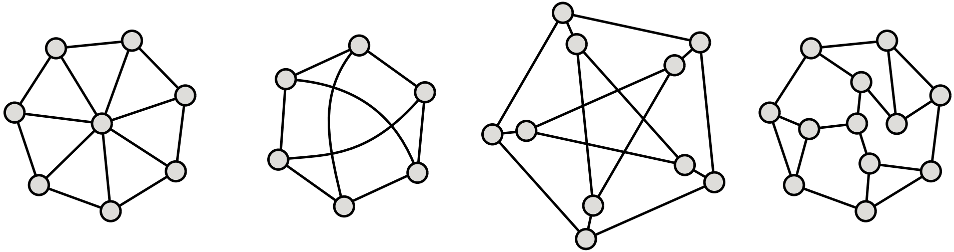
Couverture | *Vertex Cover*

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 — en master
- Meilleurs algorithmes connus en $O(2^k n)$, voire $O(1,2738^k + kn)$ — *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*

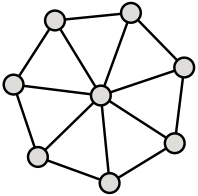
COUVAPPROX(G) :

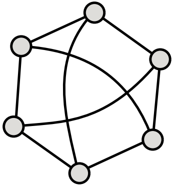
1. $C \leftarrow \emptyset$
2. Tant que G est non vide :
3.   Choisir une arête $\{u, v\}$ dans G
4.   Ajouter u **et** v dans C
5.   Supprimer u et v (et les arêtes incidentes) de G
6. Renvoyer C

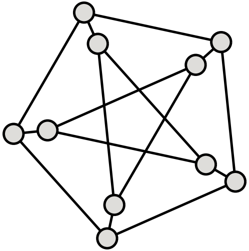
Complexité

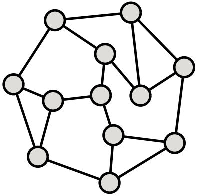
L'algorithme COUVAPPROX a une complexité $O(n^2)$

Exemples









Garantie de l'algorithme d'approximation

Théorème

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

$$\#C \leq 2\,\textsc{opt}$$

Preuve

Table des matières

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 $S_{\text{OPT}} \to$ *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

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

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

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

Preuve

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

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

$\textsc{opt}$: valeur de la solution optimale

(max) $\textsc{opt} = \max_{s \in S} c(s)$

(min) $\textsc{opt} = \min_{s \in S} c(s)$

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

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

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

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

Exemples

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 → efficacité
  - pas toujours optimal → non exact
  - souvent *pas trop mauvais* → 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

$$\begin{aligned} &(\max) \quad c(s) \geq \alpha \cdot \textsc{opt} \\ &(\min) \quad c(s) \leq \alpha \cdot \textsc{opt} \end{aligned}$$

Deux bornes à trouver

	(max)	(min)
▶ Borne c_1 sur le résultat renvoyé	$c(s) \geq c_1$	$c(s) \leq c_1$
▶ Borne c_2 sur le résultat optimal	$\textsc{opt} \leq c_2$	$\textsc{opt} \geq c_2$
$\Rightarrow$ Borne sur le facteur d'approximation	$\alpha \geq c_1/c_2$	$\alpha \leq c_1/c_2$

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

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 $\qquad T_{[j]}$: *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

Preuve

Algorithme glouton avec tri

Nouveau scénario : on connaît toutes les tâches à l'avance $\rightarrow$ 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 $\rightarrow O(n \log n)$ 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

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