Chap. 5 – Algorithmes d'approximation

HAI503I – Algorithmique 4

Bruno Grenet

Université de Montpellier – Faculté des Sciences

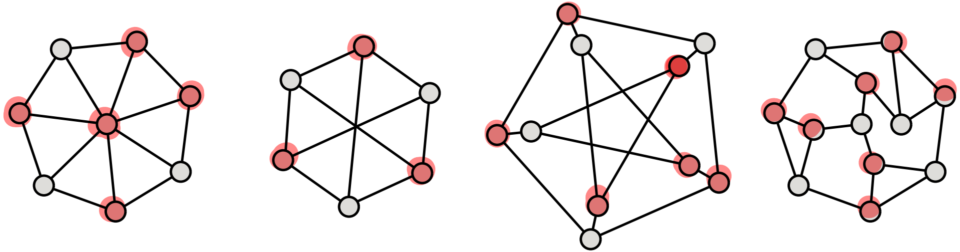
Définition du problème

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 *HAI602I*
- 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*

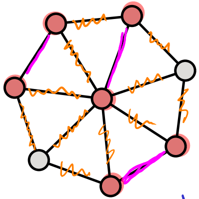
CouvApprox(G) :

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

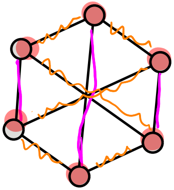
Complexité

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

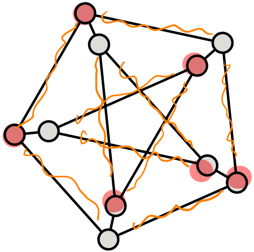
Exemples



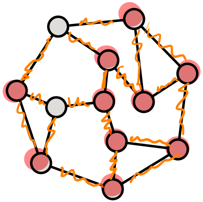
Solution de taille 6
optimal : 5



6
3



6
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

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

B : ensemble des arêtes choisies par l'algo pour créer C

Dans une solution optimale, toutes les arêtes de B doivent être couvertes.

Les arêtes de B n'ont pas de sommet en commun,

donc $\text{OPT} \geq |B|$, $|C| = 2|B| \Rightarrow |C| = 2|B| \leq 2\,\text{OPT}$ $\square$

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

- Pour $S \subset E$, $\Sigma S = \sum_{x \in S} x$
- Objectif : trouver S tel que $\Sigma S \leq T$ est maximale
- OPT : valeur de la solution maximale (la meilleure possible)

$E = \{1, 7, 28, 3, 9, 41, 11, 8\}$

$T = 16 \rightsquigarrow \{7, 9\}$
$T = 30 \rightsquigarrow \{7, 3, 9, 11\}$
$T = 33 \rightsquigarrow \{28, 3, 1\} \rightsquigarrow 32$

Solution exacte

Recherche exhaustive et *backtrack* — TD2 Ex. 1

- Parcours de tous les sous-ensembles $S \subset E$
  - 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 — *HAI602I*
- Meilleur algorithme connu en $O(2^{n/2}) = O(1,414^n)$ — *difficile !*

Que peut-on faire en *temps polynomial* ?

Un algorithme d'approximation

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

$E = \{1, 7, 28, 3, 9, 41, 11, 8\}$

↳ $E = \{41, 28, 11, 9, 8, 7, 3, 1\}$

$T = 30$: $\{28, 1\}$ 29 au lieu de 30

$T = 16$: $\{11, 3, 1\}$ 15 — 16

$T = 33$: $\{28, 3, 1\}$ 32 — 32

Complexité

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

Garantie de l'algorithme d'approximation

Théorème

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

$$\Sigma S \geq \frac{1}{2}\,\textsc{opt}$$

- On élimine de E tous les éléments $> T$.
- Si $S = E$, la solution est optimale.
- Sinon : soit $E[i]$ le 1er élément <u>non choisi</u> par l'algo
  - $E[0] \geq E[1] \geq \dots \geq E[i] \rightsquigarrow E[i] \leq \sum_{j=0}^{i-1} E[j]$
  - $E[i] + \sum_{j=0}^{i-1} E[j] > T$ $\rightsquigarrow$ sinon l'algo sélectionne $E[i]$

$$\Rightarrow \textsc{opt} \leq T < E[i] + \sum_{j=0}^{i-1} E[j] \leq 2 \sum_{j=0}^{i-1} E[j] \leq 2\,\Sigma S \Rightarrow \Sigma S \geq \frac{1}{2}\,\textsc{opt} \quad \boxtimes$$

Problèmes d'optimisation

Cadre général

- *Problème de maximisation* : sur une entrée, trouver une solution qui *maximise* une certaine fonction
- *Problème de minimisation* : sur une entrée, trouver une solution qui *minimise* une certaine fonction

Exemples

Max : SommePartielle , SAC-A-DOS , ChoixCours

Min : Couverture , Voyageur de commerce

Formalisation

Définition

- 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 $\quad \forall s' \in S, c(s') \leq c(s)$
  - minimisation : trouver $s \in S$ telle que $c(s)$ soit minimale $\quad \forall s' \in S, c(s') \geq c(s)$
- Valeur optimale : on note OPT la valeur de la solution optimale
  - maximisation : $\text{OPT} = \max_{s \in S} c(s)$
  - minimisation : $\text{OPT} = \min_{s \in S} c(s)$

COUV : $c(s)$ = #sommets

S.P. : $c(s)$ = somme des élts de s

Résolution exacte

Comment résoudre un problème d'optimisation de manière exacte ?

Recherche exhaustive et *backtrack* — Chap. 2

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

Algorithmes gloutons — Cours L2

- Construction d'une solution en optimisant localement à chaque étape
- Fonctionne parfois… ; complexité *souvent assez bonne*

Programmation dynamique — Cours L2

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

Algorithmes d'approximation

Algorithmes de compromis

- Algorithmes efficaces $\to$ complexité polynomiale, voire linéaire
- Algorithmes non exacts $\to$ 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 : $\alpha \cdot \textsc{opt} \leq c(s) \leq \textsc{opt}$ $\qquad 0 < \alpha < 1$
- minimisation : $\textsc{opt} \leq c(s) \leq \alpha \cdot \textsc{opt}$ $\qquad \alpha > 1$

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

Exemples

COUVERTUREAPPROX : 2-approximation

S.P. APPROX : 1/2-approximation.

Comment concevoir des algorithmes d'approximation ?

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

Une technique fructueuse : algorithmes glouton

- Approche gloutonne souvent rapide $\rightarrow$ efficacité
- Pas toujours la meilleure solution $\rightarrow$ non exact
- Solution souvent *pas trop mauvaise* $\rightarrow$ 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

Objectif du cours

- Concevoir et analyser des algorithmes d'approximation *simples*

Comment analyser un algorithme d'approximation ?

Objectif

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

- $c(s) \geq \alpha \cdot \text{OPT}$ (si maximisation)
- $c(s) \leq \alpha \cdot \text{OPT}$ (si minimisation)

Deux bornes à trouver (cas max.) (cas min.)

- Trouver une borne c_1 telle que $c(s) \geq c_1$ — $c(s) \leq c_1$
- Trouver une borne c_2 telle que $\text{OPT} \leq c_2$ — $\text{OPT} \geq c_2$

$\rightarrow$ On en déduit que $\alpha \geq c_1/c_2$ — $\alpha \leq c_1/c_2$

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

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 de calcul

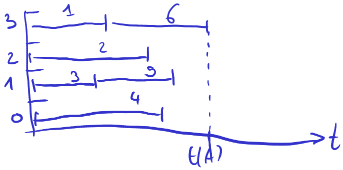
Équilibrage *Load Balancing*

Entrée : Tableau D d'entiers positifs (*durées*) et entier m

Sortie : Tableau A : affectation de chaque tâche à un processeur

- tâche i affectée au processeur j : $A_{[i]} = j$

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



NP-difficile

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, intialisé à 0 (*temps total par processeur*)
2. Pour $i = 0$ à $n-1$:
3. $\quad j \leftarrow$ indice du 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 $O(n \log m)$ avec un tas)

Garantie de l'algorithme glouton

Théorème

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

- $\mathrm{OPT} \geqslant \max_i D[i]$ car il faut affecter le max à un proc.
- $\mathrm{OPT} \geqslant \frac{1}{m} \sum_{i=0}^{n-1} D[i]$ car $\sum_{i=0}^{n-1} D[i] = \sum_{j=0}^{m-1} T^{\mathrm{OPT}}_{[j]} \leqslant m \cdot \mathrm{OPT}$

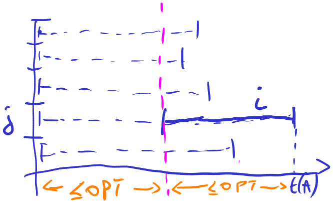
Soit A l'affectation calculée par l'algo et j tq $T[j] = \max_k T[k]$

Soit i la dernière tâche affectée à j.

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

$$T[j] - D[i] \leqslant \frac{1}{m} \sum_{k=0}^{m-1} T[k] = \frac{1}{m} \sum_{i=0}^{n-1} D[i] \leqslant \mathrm{OPT}$$

$\Rightarrow T[j] \leqslant \mathrm{OPT} + D[i] \leqslant 2\,\mathrm{OPT}$



Algorithme glouton avec tri

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

- On peut trier les tâches par durée décroissante et affecter les 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 $\leq 3/2$

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 $\rightarrow$

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