Approximationsalgorithmen

Jan Johannsen

Vorlesung im Sommersemester 2007

Einordnung

Algorithmik

- Entwurf und
- Analyse von Algorithmen

Komplexitätstheorie

- Analyse der Komplexität von Problemen
- Einteilung in Klassen ähnlicher Komplexität
- Untersuchung der Struktur der Klassen

Analyse von Algorithmen

Analyse des Verbrauchs an Ressourcen

- Zeit
- Speicher
- ...

Abhängig von der Größe des Input

- Komplexität im *worst case*

Asymptotische Analyse – vernachlässigt

- konstante Faktoren
- endliche Anfangsstücke

Komplexität von Problemen

Obere Schranken

- durch Angabe eines Algorithmus
- und dessen Analyse

Untere Schranken

- durch Analyse des Problems
- Beweis, dass jeder effizientere Algorithmus versagt

Beispiel: Sortieren

- HeapSort, MergeSort brauchen $O(n \log n)$ Vergleiche
- untere Schranke: $\Omega(n \log n)$ Vergleiche
- $\rightsquigarrow$ Komplexität: $\Theta(n \log n)$

Problemtypen

Problem: Erreichbarkeit

Instanz: Graph G, Ecken s, t.
Lösung: Weg von s nach t.

Entscheidung: Gibt es einen Weg von s nach t ?

Suche: Finde einen Weg von s nach t.

Optimierung: Finde einen *kürzesten* Weg von s nach t.

Zählen: Bestimme die *Anzahl* der Wege von s nach t.

Optimierungsprobleme

Eigenschaften:

- mehrere potentielle Lösungen zu jedem Input
- mit Kosten- / Nutzenmaß versehen
- gesucht: Lösung mit minimalen Kosten / maximalem Nutzen

Beispiele:

- kürzeste Wege
- Travelling Salesperson
- Scheduling
- ...

Approximation

Viele Optimierungsprobleme sind NP-schwer

- effizienter Algorithmus nur, wenn P = NP
- nur exponentielle Algorithmen zur optimalen Lösung

Deshalb: Approximationsalgorithmen

- effizient
- liefern eine nicht notwendig optimale Lösung
- Abweichung vom Optimum a priori abzuschätzen

Approximierbarkeit

NP-schwere Optimierungsprobleme unterscheiden sich bzgl. ihrer Approximierbarkeit:

- es gibt beliebig gute Approximationsalgorithmen
- es gibt Approximationsalgorithmen einer gewissen Güte, aber keine besseren.
- es gibt überhaupt keine guten Approximationsalgorithmen

Ziele:

- Methoden zum Entwurf
- Grundlagen zur Untersuchung dieser Unterschiede

Übersicht

Übersicht

Komplexitätsklassen

Entscheidungsproblem ist gegeben durch:

- Menge von Instanzen I
- Teilmenge der positiven Instanzen $Y \subseteq I$

Beispiel: SAT

- I_{SAT} = Formeln in KNF
- Y_{SAT} = erfüllbare Formeln

Formel in KNF	$F = C_1 \wedge \ldots \wedge C_m$
Klausel	$C = a_1 \vee \ldots \vee a_k$
Literal	$a = x$ oder $a = \neg x$.

Komplexitätsklassen

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}$$

NP als Verifikation von Lösungen

Problem P ist in NP gdw.

- es gibt Menge W_P, Problem $Q \in \mathrm{P}$ und Polynom $q()$ mit
- I_Q sind Paare (x, w) mit $x \in I_P$, $w \in W_P$ und $|w| \leq q(|x|)$
- $\forall x \in I_P : \quad x \in Y_P \leftrightarrow \exists w \in W_P : (x, w) \in Y_Q$

Beispiel: SAT

- W_{SAT} = Bewertung der Variablen
- $(F, \alpha) \in Y_R$ gdw. α erfüllt F

Reduktion

Funktion f reduziert P auf Q, wenn

- $f : I_P \longrightarrow I_Q$ in polynomieller Zeit berechenbar
- für alle $x \in I_P$ ist $x \in Y_P \leftrightarrow f(x) \in Y_Q$

$P \leq_m Q$ falls es eine Reduktion f von P auf Q gibt.

Eigenschaft:

Ist $Q \in \mathrm{P}$ und $P \leq_m Q$, dann auch $P \in \mathrm{P}$.

Vollständigkeit

Definition:

- Problem P ist NP-schwer, falls $Q \leq_m P$ für alle $Q \in \text{NP}$.
- P ist NP-vollständig, falls $Q \in \text{NP}$ und NP-schwer.

Q NP-vollständig $\Rightarrow$ $Q \in \text{P}$ gdw. $\text{P} = \text{NP}$

Satz (Cook)

SAT ist NP-vollständig.

Problem 3-*SAT*: Jede Klausel $C = a_1 \vee a_2 \vee a_3$

Satz (Cook)

3-*SAT ist NP-vollständig.*

Lineare Programmierung

Problem *ILP* (Integer Linear Programming)

Instanz:	Menge I von linearen Ungleichungen über $\mathbb{Z}$ in Variablen $z_1, \ldots, z_n$
Frage:	Gibt es eine Lösung von I in $\mathbb{Z}$ d.h., Zuweisung von Werten $z_1, \ldots, z_n \in \mathbb{Z}$ so dass Ungleichungen I gelten.

Satz

ILP ist NP-vollständig.

Problem *LP*:

Wie ILP, aber gesucht sind Lösungen in $\mathbb{Q}$.

Im Gegensatz zum obigen Satz ist *LP* in P.

Graphenprobleme in NP

Problem *VC*

Instanz:	Graph $G = (V, E)$, $k \in \mathbb{N}$
Frage:	Gibt es vertex cover U mit $\lvert U\rvert \leq k$?

Vertex cover: $U \subseteq V$ mit $e \cap U \neq \emptyset$ für alle $e \in E$.

Problem CLIQUE

Instanz:	Graph $G = (V, E)$, $k \in \mathbb{N}$
Frage:	Gibt es Clique U mit $\lvert U\rvert \geq k$?

Clique: $U \subseteq V$ mit $\{x, y\} \in E$ für alle $x, y \in U$.

Problem *HC*

Instanz:	Graph $G = (V, E)$
Frage:	Gibt es einen Hamilton-Kreis in G ?

Hamilton-Kreis: Kreis, der jedes $v \in V$ genau einmal besucht.

NP-vollständige Graphenprobleme

Offensichtlich: *VC*, CLIQUE, *HC* sind in NP.

Reduktionen (Karp)

- $3\text{-}SAT \leq_m VC$
- $VC \leq_m \text{CLIQUE}$
- $VC \leq_m HC$

Also: *VC*, CLIQUE, *HC* sind NP-vollständig.

Optimierungsprobleme

Ein *Optimierungsproblem* ist gegeben durch:

- Menge I von Instanzen.
- Für jedes $x \in I$ eine Menge $S(x)$ von potentiellen Lösungen.
- Funktion m, die jedem (x, y) mit $x \in I$ und $y \in S(x)$ ein Maß $m(x, y)$ zuordnet.
- Ziel goal $\in \{\min, \max\}$.

Eine Lösung $y^* \in S(x)$ heißt optimal, wenn

$$m(x, y^*) = m^*(x) := \text{goal}\{\, m(x, y)\,;\; y \in S(x)\,\}$$

$$S^*(x) := \{\, y^* \in S(x)\,;\; m(x, y^*) = m^*(x)\,\}$$

Beispiele

Minimum Path

Instanz:	ungerichteter Graph $G = (V, E)$, Ecken s, t
Lösung:	Weg von s nach t
Maß:	Länge des Weges

Minimum Vertex Cover

Instanz:	ungerichteter Graph $G = (V, E)$
Lösung:	vertex cover $U \subseteq V$
Maß:	$\lvert U \rvert$

Maximum Clique

Instanz:	ungerichteter Graph $G = (V, E)$
Lösung:	Clique $U \subseteq V$
Maß:	$\lvert U \rvert$

Problemvarianten

Zu einem Optimierungsproblem P sind definiert:

konstruktives Problem P_C: Gegeben $x \in I$, finde $y \in S^*(x)$.

Auswertungsproblem P_E: Gegeben $x \in I$, berechne $m^*(x)$.

Entscheidungsproblem P_D: Gegeben $x \in I$ und $k \in \mathbb{N}$, ist $\text{goal}(m^*(x), k) = m^*(x)$?

Beispiele:

$P =$ Minimum Vertex Cover	$P_D =$ VC.
$P =$ Maximum Clique	$P_D =$ Clique

Die Klassen NPO und PO

Ein Optimierungsproblem ist in der Klasse NPO, falls

1. I ist in P.
2. für $x \in I$ und $y \in S(x)$ ist $|y| \leq q(|x|)$
3. für x, y mit $|y| \leq q(|x|)$ ist die Frage "$y \in S(x)$?" in P
4. m ist in polynomieller Zeit berechenbar.

Eigenschaft: P in NPO $\Longrightarrow$ P_D in NP

$P \in$ NPO ist in der Klasse PO, falls
das konstruktive Problem P_C in polynomieller Zeit lösbar ist.

Analog: P in PO $\Longrightarrow$ P_D in P

Turing-Reduktion

Eine Turing-Reduktion von P auf Q ist

- Algorithmus, der P löst
- benutzt Subroutine zur Lösung von Q
- Laufzeit des Algorithmus ohne Subroutine ist polynomiell

$P \leq_T Q$ falls eine Turing-Reduktion von P auf Q gibt.

Eigenschaft:

Ist $Q \in \mathrm{P}$ und $P \leq_T Q$, dann auch $P \in \mathrm{P}$.

Relationen zwischen Problemvarianten

Offensichtlich: $P_D \leq_T P_E \leq_T P_C$

Fakt

Für alle Optimierungsprobleme $P \in NPO$ ist $P_E \leq_T P_D$.

Beweis:

Da $|m(x,y)| \leq p(|x|)$ für alle $y \in S(x)$, ist $m^*(x) < 2^{p(|x|)}$

$\rightsquigarrow$ binäre Suche findet $m^*(x)$ in Zeit $O(p(|x|)$.

Relationen zwischen Problemvarianten

Beispiel

Für $P = \textsc{Maximum Clique}$ ist $P_C \leq_T P_E$.

Für $G = (V, E)$ definiere

- $N(v) := \{ u \in V ; \{u, v\} \in E \}$
- $G(v)$: induzierter Teilgraph auf $\{v\} \cup N(v)$
- $G^-(v)$: induzierter Teilgraph auf $N(v)$

Algorithmus:

MaxClique(G)
  $k := \textsc{Maximum Clique}_E(G)$
  `if` $k = 1$ `then return any` $v \in V$
  `finde` $v \in V$ `mit` $\textsc{Maximum Clique}_E(G(v)) = k$
  `return` $\{v\} \cup$ *MaxClique*$(G^-(v))$

Relationen zwischen Problemvarianten

Ob $P_C \leq_T P_D$ allgemein gilt, ist unbekannt.

Satz

$P_C \leq_T P_D$ gilt, falls P_D NP-vollständig ist.

NP-schwere Optimierungsprobleme

Definition

Problem P in NPO ist NP-schwer, falls $Q \leq_T P_C$ für alle $Q \in \text{NP}$.

Eigenschaft:

$$P_D \text{ NP-vollständig} \implies P \text{ ist NP-schwer.}$$

$$\text{P} \neq \text{NP} \implies \text{PO} \neq \text{NPO}$$

Ein NP-schweres Problem

MINIMUM TRAVELING SALESPERSON

Instanz: n Städte $c_1, \ldots, c_n$,
Distanzmatrix $D \in \mathbb{N}^{n \times n}$

Lösung: *Tour*: Permutation $c_{i_1}, \ldots, c_{i_n}$ der Städte

Maß: Kosten $\sum_{k=1}^{n-1} D(i_k, i_{k+1}) + D(i_n, i_1)$

Satz: MINIMUM TRAVELING SALESPERSON ist NP-schwer.

Übersicht

Greedy-Algorithmen

Geeignet, wenn Lösung Menge von Objekten

- Lösung wird schrittweise aufgebaut
- in jedem Schritt wird ein Element aufgenommen
- Greedy-Strategie: wähle in jedem Schritt so, dass Profit maximiert wird.

Liefert optimale Lösung, wenn Matroid-struktur

In anderen Fällen: approximative Lösung.

Das Spannbaum-Problem

Für $G = (V, E)$ ist $T \subseteq E$ ein Spannbaum, wenn (V, T) zusammenhängend und azyklisch ist.

Minimum Spanning Tree

Instanz:	zusammenhängender Graph $G = (V, E)$, Kantengewichte $w : E \to \mathbb{N}$
Lösung:	Spannbaum $T \subseteq E$
Maß:	Gesamtgewicht $\sum_{e \in T} w(e)$

Ein klassischer Greedy-Algorithmus

Satz

MINIMUM SPANNING TREE *ist in PO.*

Dies zeigt beispielsweise der Greedy-Algorithmus von Kruskal:

sortiere $E = \{e_1, \ldots, e_m\}$ so dass $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$
$T := \emptyset$
`for` $i = 1$ `to` n `do`
    `let` $e_i = \{x_i, y_i\}$
    `if` x_i in T nicht mit y_i verbunden
        `then` $T := T \cup \{e_i\}$

Das Rucksackproblem

Maximum Knapsack

Instanz: Menge $X = \{x_1, \ldots, x_n\}$,
für jedes $x_i \in X$: Gewicht $a_i \in \mathbb{N}$
Wert $p_i \in \mathbb{N}$
Kapazität $b \in \mathbb{N}$

Lösung: Teilmenge $Y \subseteq X$ mit $\sum_{x_i \in Y} a_i \leq b$

Maß: Gesamtwert $\sum_{x_i \in Y} p_i$

Greedy Algorithmus für Maximum Knapsack

Algorithmus *Greedy Knapsack*:

```
input X, p⃗, a⃗, b
sortiere X so dass p_1/a_1 ≥ p_2/a_2 ≥ ··· ≥ p_n/a_n
Y := ∅
for i := 1 to n do
    if a_i ≤ b then
        Y := Y ∪ {x_i}
        b := b − a_i
return Y
```

Analyse von *Greedy Knapsack*

Satz

Für jedes $k \in \mathbb{N}$ gibt es Instanzen von MAXIMUM KNAPSACK *mit:*

ist m_{Greedy} der Wert der Lösung, die Greedy Knapsack findet, und m^* der optimale Wert, so gilt: $m^* \geq k \cdot m_{Greedy}$.

Beweis: Betrachte die Instanz mit

- $X = \{x_1, \ldots, x_{n+1}\}$
- $p_i = a_i = 1$ für $i \leq n$
- $a_{n+1} = b = kn + 1$
- $p_{n+1} = b - 1$

Dann ist $m^* = b - 1 = kn$,
aber $m_{Greedy} = n$.

Verbesserter Algorithmus für Maximum Knapsack

Sei μ dasjenige i mit $p_i = \max\{p_j\,, j \leq n\}$.

Algorithmus:

Berechne Lösung Y mit *Greedy Knapsack*
`if` $m(Y) < p_\mu$
    `then` $Y := \{x_\mu\}$
`return` Y

Satz

Sei $m_A(x)$ der Wert der vom Algorithmus berechneten Lösung. Dann gilt: $m^*(x) < 2m_A(x)$.

Greedy Algorithmus für Handlungsreisende

Algorithmus *Nearest Neighbor*:

Wähle c_{i_1} *beliebig*
Für $1 \leq k \leq n-1$*:*
wähle als $c_{i_{k+1}}$ *ein* $c_j \notin \{c_{i_1}, \ldots, c_{i_k}\}$
mit $D(c_{i_k}, c_j)$ *minimal.*

Minimum Metric Traveling Salesperson:

Spezialfall von Minimum Traveling Salesperson, wo gilt:

Symmetrie: $D(i,j) = D(j,i)$

Dreiecksungleichung: $D(i,j) \leq D(i,k) + D(k,j)$

Performance von *Nearest Neighbor*

Lemma: Sei x eine Instanz von Minimum Metric Traveling Salesperson, und $\ell : \{c_1, \ldots, c_n\} \to \mathbb{Q}$ mit

$$D(i,j) \geq \min(\ell(c_i), \ell(c_j)) \qquad \text{für alle } c_i \neq c_j$$

$$\ell(c_i) \leq \frac{1}{2} m^*(x) \qquad \text{für alle } c_i$$

Dann gilt: $$\sum_{i=1}^{n} \ell(c_i) \leq \frac{1}{2}(\lceil \log n \rceil + 1) m^*(x) \ .$$

Satz

Sei $m_{NN}(x)$ die Länge der vom Algorithmus Nearest Neighbor gefundenen Tour. Dann gilt:

$$m_{NN}(x) \leq \frac{1}{2}(\lceil \log n \rceil + 1) \cdot m^*(x) \ .$$

Sequentielle Algorithmen

Geeignet für Partitionierungsprobleme:

- Lösung ist Partition einer gegebenen Menge.

Sequentieller Algorithmus:

- sortiert Menge $X = \{x_1, \ldots, x_n\}$
- baut Partition P sequentiell auf:

$P := \{\{x_1\}\}$
`for` $i := 2$ `to` n
    falls x_i zu einem $p \in P$ zugefügt werden kann
        $P := P \setminus \{p\} \cup \{p \cup \{x_i\}\}$
    sonst
        $P := P \cup \{\{x_i\}\}$

Ein Scheduling-Problem

Minimum Scheduling on Identical Machines

Instanz: Menge $T = \{t_1, \ldots, t_n\}$ von Aufträgen,

für jedes $t_i \in T$: Dauer $\ell_i \in \mathbb{N}$

Anzahl $p \in \mathbb{N}$ von Prozessoren

Lösung: schedule: Funktion $f : T \to \{1, \ldots, p\}$

Maß: makespan: $\max_{1 \le i \le p} \sum_{f(t_j)=i} \ell_j$

List Scheduling

Finish time: $A_i(j) := \sum_{\substack{k \le j \\ f(t_k)=i}} \ell_k$

Algorithmus *List Scheduling*:

```
for j := 1 to n do
    assign t_j to processor i with A_i(j-1) minimal
```

Satz

Für jede Instanz x findet List Scheduling eine Lösung mit makespan $m_{LS}(x)$, so dass

$$m_{LS}(x) \le (2 - \frac{1}{p}) \cdot m^*(x) .$$

Verbesserter Algorithmus

Regel *LPT* (*largest processing time*):

> *Sortiere T so dass $\ell_1 \geq \ell_2 \geq \ldots \geq \ell_n$,*
> *dann List Scheduling.*

Satz

Für jede Instanz x liefert LPT eine Lösung
mit makespan $m_{LPT}(x)$ so dass

$$m_{LPT}(x) \leq (\frac{4}{3} - \frac{1}{3p}) \cdot m^*(x)\,.$$

Ein Packungsproblem

Minimum Bin Packing

Instanz: Multimenge $\{a_1, \ldots, a_n\} \subset \mathbb{Q}$ mit $0 < a_i \leq 1$.

Lösung: Partition $B_1, \ldots, B_k$ von A
mit $\sum_{a_i \in B_j} a_i \leq 1$ für alle j.

Maß: Anzahl der Blöcke k

Einfacher sequentieller Algorithmus für Bin Packing

Algorithmus *Next Fit*

$k := 1\,;\ B_1 := \emptyset$
`for` $i := 1$ `to` n `do`
  `if` $\Sigma B_k + a_i \leq 1$
    `then` $B_k := B_k \cup \{a_i\}$
    `else` $k := k + 1\,;\ B_k := \{a_i\}$

Satz

Für alle Instanzen x ist

$$m_{\text{Next Fit}}(x) \leq 2m^*(x)$$

Etwas besserer Algorithmus

Algorithmus *First Fit*

$k := 1$; $B_1 := \emptyset$
`for` $i := 1$ `to` n `do`
    `flag :=` `true`
    `for` $j := 1$ `to` k `do`
        `if` $\Sigma B_j + a_i \leq 1$ `and flag`
            `then` $B_j := B_j \cup \{a_i\}$; `flag :=` `false`
    `if flag then` $k := k + 1$; $B_k := \{a_i\}$

Satz

Für alle Instanzen x ist

$$m_{\mathit{First\ Fit}}(x) \leq 1.7 m^*(x) + 2$$

Verbesserung durch Sortieren

Algorithmus *First Fit Decreasing*

> Sortiere A so dass $a_1 \geq a_2 \geq \ldots \geq a_n$,
> dann wende *First Fit* an.

Satz

Für alle Instanzen x ist $m_{FFD}(x) \leq \frac{3}{2} m^*(x) + 1$.

Optimale Schranke: $m_{FFD}(x) \leq \frac{11}{9} m^*(x) + 4$.

Nachteil: Algorithmus ist offline,

d.h. arbeitet erst, wenn gesamter Input gelesen.

Färbungen von Graphen

Definition: Eine Färbung eines Graphen $G = (V, E)$ ist $\chi : V \to \{1, \ldots, k\}$ mit

$$\{u, v\} \in E \quad \Rightarrow \quad \chi(u) \neq \chi(v) \; .$$

Minimum Coloring

Instanz:	ungerichteter Graph $G = (V, E)$
Lösung:	Färbung $\chi : V \to \{1, \ldots, k\}$
Maß:	k

Sequentielles Färben von Graphen


```
for i := 1 to n
    χ(v_i) := min k ∉ {χ(u); u ∈ N(v_i) ∩ {v_1, ..., v_{i-1}}}
```

Satz

Sei k die Anzahl der Farben, die der sequentielle Algorithmus bei Reihenfolge $v_1, v_2, \ldots, v_n$ verwendet. Dann ist

$$k \leq \max_{1 \leq i \leq n} \min(\deg v_i, i-1) \; .$$

Reihenfolge, die dies minimiert:
Decreasing Degree: $\deg v_1 \geq \deg v_2 \geq \ldots \geq \deg v_n$.

Bessere Reihenfolge

Smallest Last: v_i hat minimalen Grad im induzierten Subgraphen auf $\{v_1 \ldots v_i\}$.

Satz

Smallest Last färbt jeden planaren Graphen mit ≤ 6 Farben.

Beweis benutzt den Satz von Euler:

In planaren Graphen $G = (V, E)$ ist $|E| \leq 3|V| - 6$.

Lokale Suche

Anfangslösung	Für jedes $x \in I$:	$y_0(x) \in S(x)$
Nachbarschaftsstruktur	Für jedes $y \in S(x)$	$N(x, y) \subseteq S(x)$

Algorithmus *Local Search*

$y := y_0(x)$
solange es $y' \in N(x, y)$ mit $m(x, y')$ besser als $m(x, y)$ gibt
    setze $y := y'$

Probleme:

- Wie findet man bessere Nachbarlösungen effizient?
- Wie gut ist ein lokales Optimum?

Beispiel für Lokale Suche

Problem MAXIMUM CUT

Instanz:	Graph $G = (V, E)$
Lösung:	Partition $V = V_0 \cup V_1$ mit $V_0 \cap V_1 = \emptyset$
Maß:	Anzahl der Kanten $\{x, y\}$ mit $x \in V_0$ und $y \in V_1$.

Anfangslösung: $(\emptyset, V)$

Nachbarn von (V_0, V_1): (V_0', V_1') mit $\big||V_0| - |V_0'|\big| = 1$

Satz

Für ein lokales Maximum (V_0, V_1) in dieser Nachbarschaftsstruktur mit Maß m_N gilt $m_N \geq \frac{1}{2} m^*$.

Lineare Programmierung

Ein lineares Programm ist Instanz des Optimierungsproblems:

LINEAR PROGRAMMING

Instanz:	$A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m,\ c \in \mathbb{R}^n$
Lösung:	$x \in \mathbb{R}^n$ mit $Ax \geq b$ und $x \geq \vec{0}$
Maß:	$c^T x$

Notation:

minimize	$c^T x$
subject to	$Ax \geq b$
	$x \geq 0$

LP-Algorithmen

Satz

Linear Programming *ist in PO.*

Algorithmen:

- Simplex (Dantzig, 1947)
  in der Praxis gut, aber im worst case exponentiell
- Ellipsoid-Methoide (Khachiyan, 1979)
  polynomiell, aber nicht praktisch anwendbar
- Interior-Point-Methode (Karmarkar, 1984)
  polynomiell, auch praktisch effizient

Äquivalente Varianten des Problems

- Gleichungen und unbeschränkte Variablen

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & Ax = b \\ & x_1 \geq 0, \ldots, x_r \geq 0 \end{array}$$

- nur Gleichungen

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

- Maximierung

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array}$$

Runden der Lösung eines linearen Programms

Integer Linear Programming ist NP-schwer
$\rightsquigarrow$ viele Probleme in NPO darauf reduzierbar.

Idee:

- Lockern der Anforderung "ganzzahlig"
- lineares Programm — effizient lösbar
- optimale rationale Lösung
- Runden liefert ganzzahlige Lösung
- nicht otwendig optimal, aber oft gute Approximation

Vertex Cover mittels LP

Minimum Weighted Vertex Cover

Instanz:	ungerichteter Graph $G = (V, E)$, Gewicht c_i für $v_i \in V$.
Lösung:	vertex cover $U \subseteq V$
Maß:	$\sum_{v_i \in U} c_i$

Lineares Programm P_{VC}:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} & x_i + x_j \geq 1 \quad \text{für alle } \{v_i, v_j\} \in E \\ & x_i \geq 0 \end{array}$$

Optimale Lösung $x_1^*, \ldots, x_n^* \quad \rightsquigarrow \quad$ vertex cover $U := \{v_i \,;\, x_i^* \geq \frac{1}{2}\}$.

Satz

Für das Maß m_{LP} dieses vertex cover U gilt $m_{LP} \leq 2m^*$.

Dualität

$m^* \leq t$ wird durch Lösung x mit $c^T x \leq t$ bezeugt.

Frage: Wie kann $m^* \geq t$ bezeugt werden ?

Beispiel:

$$\begin{array}{ll} \text{minimize} & 7x_1 + x_2 + 5x_3 \\ \text{subject to} & x_1 - x_2 + 3x_3 \geq 10 \\ & 5x_1 + 2x_2 - x_3 \geq 6 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Koeffizientenvergleich zeigt: $m^* \geq 10$

$$7x_1 + x_2 + 5x_3 \geq x_1 - x_2 + 3x_3 \geq 10$$

Besser: $m^* \geq 16$

$$7x_1 + x_2 + 5x_3 \geq x_1 - x_2 + 3x_3 + 5x_1 + 2x_2 - x_3 \geq 16$$

$\rightsquigarrow$ duales Programm

Duales Programm

Programm P	Duales Pogramm D
Ungleichung $\sum_j a_{i,j} x_j \geq b_i$	Variable y_i mit $y_i \geq 0$
Gleichung $\sum_j a_{i,j} x_j = b_i$	Variable y_i
Variable x_j mit $x_j \geq 0$	Ungleichung $\sum_i a_{i,j} y_i \leq c_j$
Variable x_j	Gleichung $\sum_i a_{i,j} y_i \leq c_j$
`minimize` $c^T x$	`maximize` $b^T y$

Dualitätstheorem

Für Programm P und duales D gilt

- P hat optimale Lösung gdw. D hat optimale Lösung.
- Ist x^* optimal für P und y^* für D, so ist $c^T x^* = b^T y^*$.

Complementary Slackness Condition

Schwaches Dualitätstheorem

Für Lösungen x für P und y für D gilt:

$$c^T x \geq b^T y$$

Lösungen x für P und y für D sind optimal gdw. die folgenden Bedingungen gelten:

Primary complementary slackness condition (CSC)

$$x_j = 0 \quad \text{oder} \quad \sum_i a_{i,j} y_i = c_j \qquad \text{für alle } j$$

Dual complementary slackness condition (CSC):

$$y_i = 0 \quad \text{oder} \quad \sum_j a_{i,j} x_j = b_i \qquad \text{für alle } i$$

Primal-Dual-Schema

Gegeben I ganzzahliges LP, sei D das duale Programm.

Halte zwei Vektoren:

- x potentielle Lösung für I
- y Lösung für D, nicht notwendig optimal.

x, y werden iterativ verbessert, bis x Lösung von I ist.

Dabei gilt stets die Primal CSC

$$x_j = 0 \quad \text{oder} \quad \sum_i a_{i,j} y_i = c_j \qquad \text{für alle } j$$

und eine abgeschwächte Dual CSC

$$y_i = 0 \quad \text{oder} \quad \sum_j a_{i,j} x_j \leq \alpha \cdot b_i \qquad \text{für alle } i$$

Algorithmus terminiert, wen x Lösung von I ist. Dann gilt:

$$\sum_j c_j x_j = \sum_j \left(\sum_i a_{i,j} y_i\right) x_j = \sum_i \left(\sum_j a_{i,j} x_j\right) y_i \leq \alpha \cdot \sum_i b_i y_i \leq \alpha m^*$$

Primal-Dual Algorithmus für Vertex Cover

Duales Programm D_{VC}:

$$\text{maximize} \quad \sum_{\{v_i,v_j\}\in E} y_{i,j}$$

$$\text{subject to} \quad \sum_{j:\{v_i,v_j\}\in E} y_{i,j} \leq c_i \quad (C_i)$$

$$y_{i,j} \geq 0$$

Algorithmus:

$\vec{y} := 0;\ U := \emptyset$
while U not a vertex cover
    find $\{v_i, v_j\}$ not covered
    increase $y_{i,j}$
        until (C_i) or (C_j) tight
    if (C_i) is tight then
        $U := U \cup \{v_i\}$
    else
        $U := U \cup \{v_j\}$

Satz

Der obige Algorithmus findet ein vertex cover mit Maß m_{PD} derart dass $m_{PD} \leq 2m^*$.

Das Multicut-Problem

MINIMUM MULTICUT

Instanz:	ungerichteter Graph $G = (V, E)$, Kapazität c_e für $e \in E$, Menge $\{(s_1, t_1), \ldots, (s_k, t_k)\} \subseteq V \times V$ mit $s_i \neq t_i$.
Lösung:	Schnitt $C \subseteq E$ mit: s_i und t_i in $(V, E \setminus C)$ nicht verbunden
Maß:	$\sum_{e \in C} c_e$

Satz

MINIMUM MULTICUT *ist NP-schwer. für Bäume der Höhe 1.*

Beweis: Reduktion von VERTEX COVER.

Multicut in Bäumen

Sei p_i der eindeutige Pfad von s_i nach t_i in G

Lineares Programm:

$$\begin{array}{lll} \text{minimize} & \sum_{e \in E} c_e x_e & \\ \text{subject to} & \sum_{e \in p_i} x_e \geq 1 & \text{für } i = 1 \ldots k \\ & x_e \geq 0 & \end{array}$$

Bemerke: Es gibt Lösungen, die besser sind als jeder Multicut.

Primal-Dual Algorithmus für Minimum Multicut

Duales Programm:

$$\begin{array}{ll} \text{maximize} & \sum_{i \leq k} y_i \\ \text{subject to} & \sum_{i:p_i \ni e} y_i \leq c_i \quad \text{für } e \in E \\ & y_i \geq 0 \end{array}$$

Interpretation: y_i ist Fluss von s_i nach t_i.

Fall $k = 1$: Dualitätstheorem $\hat{=}$ Max Flow - Min Cut

Complementary slackness conditions

Primäre CSC:

$$x_e \neq 0 \;\Rightarrow\; \sum_{i:p_i \ni e} y_i = c_e \qquad \text{für alle } e \in E$$

d.h. nur Kanten ausgewählt, durch die maximaler Fluss geht.

Abgeschwächte duale CSC:

$$y_i \neq 0 \;\Rightarrow\; \sum_{e \in p_i} x_e \leq 2$$

d.h. auf Pfad mit Fluss > 0 höchstens zwei Kanten ausgewählt.

Der Algorithmus

Wähle ein $v \in V$ als Wurzel
sortiere $V = \{v_1, \ldots, v_n\}$ absteigend nach Tiefe
$y := \vec{0}$
$C := \emptyset$

für $i := 1, \ldots, n$:
    für jedes j mit $\mathrm{lca}(s_j, t_j) = v_i$
        finde $e \in p_j$ mit $c_e - \sum_\ell y_\ell$ minimal
        $y_j := c_e - \sum_{\ell<i} y_\ell$
        füge alle Kanten e zu C hinzu, für die jetzt $\sum_\ell y_\ell = c_e$

sei $C = \{e_1, \ldots, e_m\}$ geordnet wie zugefügt
für $i := m, \ldots, 1$
    falls $C \setminus \{e_j\}$ Multicut ist
        $C := C \setminus \{e_j\}$

Analyse des Algorithmus

Lemma: Sei (s_i, t_i) mit $y_i > 0$, und sei $\mathrm{lca}(s_j, t_j) = v$.
Dann ist auf jedem der Pfade

- von s_i zu v
- von v zu t_i

höchstens eine Kante in C.

Satz

Für den Wert m_{PD} des gefundenen Multicut gilt $m_{PD} \leq 2m^*$.

Dynamische Programmierung

- Zerlegung des Problems in kleinere, gleichartige Teilprobleme.
- Lösung ist zusammengesetzt aus den Lösungen der Teilprobleme.
- Aber: Teilprobleme sind nicht unabhängig, sondern haben ihrerseits gemeinsame Teilprobleme.

$\rightsquigarrow$ rekursives Divide-and-Conquer-Verfahren löst dieselben Teilprobleme immer wieder.

Lösung: Rekursion mit Memoisierung, oder besser:

Dynamische Programmierung:

Bottom-Up-Berechnung der Lösungen größerer Probleme aus denen kleinerer, die dabei in einer Tabelle gespeichert werden.

Beispiel: Matrizen-Kettenmultiplikation

Problem: n Matrizen $M_1, \ldots, M_n$ sollen multipliziert werden, wobei $M_i \in \mathbb{R}^{n_i \times n_{i+1}}$

$\rightsquigarrow$ Aufwand hängt von der Klammerung ab.

Bezeichne $M_{i..j}$ das Produkt $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$,
und $m[i,j]$ die minimale Zahl von Multiplikationen für $M_{i..j}$.

- Optimale Klammerung von $M_{1..n}$ ist $M_{1..k} \cdot M_{k+1..n}$, für optimale Klammerungen von $M_{1..k}$ und $M_{k+1..n}$.
- Deshalb gilt:

$$m[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1} & \text{sonst.} \end{cases}$$

Teilproblemstruktur von KNAPSACK

Sei $P := \sum_{i=1}^{n} p_i$ und $A := \sum_{i=1}^{n} a_i$

Teilproblem $\Pi(k, p)$ für $k \leq n$ und $p \leq P$:

Lösung: $M \subseteq \{x_1, \ldots x_k\}$ mit $\sum_{x_i \in M} a_i \leq b$
und $\sum_{x_i \in M} p_i = p$

Ziel: minimiere $\sum_{x_i \in M} a_i$

Definition

$M(k, p)$ optimale Lösung von $\Pi(k, p)$
$S(k, p) := \sum_{x_i \in M(k,p)} a_i$

Dynamisches Programm für KNAPSACK

for $p := 0$ to P $\quad S(1,p) := A+1$

$M(1,0) := \emptyset$; $\quad S(1,0) := 0$;
$M(1,p_1) := \{x_1\}$; $\quad S(1,p_1) := a_1$

for $k := 1$ to n
  for $p := 1$ to P

    if $p_k \leq p$ and $M(k-1, p-p_k)$ defined
      and $S(k-1,p-p_k) + a_k \leq b$
      and $S(k-1,p-p_k) + a_k \leq S(k-1,p)$

    then $M(k,p) := M(k-1,p-p_k) \cup \{x_k\}$
      $S(k,p) := S(k-1,p-p_k) + a_k$

    else $M(k,p) := M(k-1,p)$
      $S(k,p) := S(k-1,p)$

$p^* := \max\{\, p ;\ M(n,p) \text{ defined} \,\}$
return $M(n,p^*)$

Approximations-Schema für KNAPSACK

Satz

Das Dynamische Programm für MAXIMUM KNAPSACK *findet eine optimale Lösung in Zeit* $O(n \cdot P)$.

Approximations-Schema:

$p_{\max} := \max_i p_i$

$t := \lfloor \log(\frac{r-1}{r} \frac{p_{\max}}{n}) \rfloor$

Ersetze $p'_i := \lfloor \frac{p_i}{2^t} \rfloor$ für $i = 1, \ldots, n$

Wende das Dynamische Programm an

Satz

Das obige Schema findet bei Eingabe $1 < r \in \mathbb{Q}$
in Zeit $O(rn^3/(r-1))$ *eine Lösung mit Maß* $m_{AS}(r)$
so dass $m_{AS}(r) \geq m^*/r$.

Übersicht

Absolute Approximation

Sei P ein Optimierungsproblem , $x \in I_P$ und $y \in S_P(x)$.

Definition: Absoluter Fehler

$$D(x, y) := |m^*(x) - m(x, y)|$$

Ein Approximationsalgorithmus A heißt absolut,
wenn $D(x, A(x) \leq k$ ist, für alle x und eine Konstante k.

Beispiel: Minimum Planar Graph Coloring

Satz

Falls $P \neq NP$, gibt es keinen absoluten Approximationsalgorithmus für Maximum Knapsack.

Relative Approximation

Definition: Performanz:

$$R(x,y) := \max\left(\frac{m(x,y)}{m^*(x)}, \frac{m^*(x)}{m(x,y)}\right)$$

Relativer Fehler:

$$E(x,y) := \frac{|m^*(x) - m(x,y)|}{\max(m^*(x), m(x,y))} = 1 - \frac{1}{R(x,y)}$$

Für $r \geq 1$ ist ein r-Approximationsalgorithmus ein Algorithmus A, der für jedes $x \in I_P$ ein $A(x) \in S_P(x)$ mit $R(x, A(x)) \leq r$ liefert.

P heißt r-approximierbar, falls es einen r-Approximationsalgorithmus für P gibt.

Die Klasse APX

APX ist die Klasse aller Optimierungsprobleme $P \in \mathrm{NPO}$, die r-approximierbar für ein $r \geq 1$ sind.

$$\mathrm{PO} \subseteq \mathrm{APX} \subseteq \mathrm{NPO}$$

APX enthält NP-schwere Probleme, z.B. Maximum Knapsack, also

$$\mathrm{P} \neq \mathrm{NP} \quad \Longrightarrow \quad \mathrm{PO} \subsetneq \mathrm{APX}$$

Maximum SAT

Instanz:	CNF $D_1 \wedge \ldots \wedge D_m$ in Variablen $x_1, \ldots, x_n$
Lösung:	Bewertung $\alpha : \{x_1, \ldots, x_n\} \to \{0, 1\}$
Maß:	Anzahl der erfüllten Klauseln $\#\{i; \alpha \models D_i\}$

Maximum SAT $\in$ APX.

Traveling Salespersons revisited

Satz

MINIMUM TRAVELING SALESPERSON $\notin$ *APX*,
außer wenn $P = NP$.

Also gilt:

$$\mathsf{P} \neq \mathsf{NP} \quad \Longrightarrow \quad \mathsf{PO} \subsetneq \mathsf{APX} \subsetneq \mathsf{NPO}$$