

Übersicht

Einführung

Grundlagen

Methoden zum Entwurf von Approximationsalgorithmen

Greedy-Algorithmen

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

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.

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Das Spannbaum-Problem

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

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 $w : E \rightarrow \mathbb{N}$

Lösung: Spannbaum $T \subseteq E$

Maß: Gesamtgewicht $\sum_{e \in T} w(e)$

Ein klassischer Greedy-Algorithmus

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Satz

MINIMUM SPANNING TREE *ist in* **PO**.

Dies zeigt beispielsweise der Greedy-Algorithmus von **Kruskal**:

sortiere $E = \{e_1, \dots, e_m\}$ so dass $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$

$T := \emptyset$

for $i = 1$ to m 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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

MAXIMUM KNAPSACK

Instanz: Menge $X = \{x_1, \dots, x_n\}$,
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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Algorithmus *Greedy Knapsack*:

```
input  $X, \vec{p}, \vec{a}, b$ 
sortiere  $X$  so dass  $\frac{p_1}{a_1} \geq \frac{p_2}{a_2} \geq \dots \geq \frac{p_n}{a_n}$ 
 $Y := \emptyset$ 
for  $i := 1$  to  $n$  do
  if  $a_i \leq b$  then
     $Y := Y \cup \{x_i\}$ 
     $b := b - a_i$ 
return  $Y$ 
```

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: $m^* \geq k \cdot m_{\text{Greedy}}$.

Beweis: Betrachte die Instanz mit

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

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

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

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Algorithmus *Nearest Neighbor*:

Wähle c_{i_1} beliebig

Für $1 \leq k \leq n-1$:

wähle als $c_{i_{k+1}}$ ein $c_j \notin \{c_{i_1}, \dots, c_{i_k}\}$
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*

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Lemma: Sei x eine Instanz von MINIMUM METRIC TRAVELING SALESPERSON, und $\ell : \{c_1, \dots, c_n\} \rightarrow \mathbb{Q}$ mit

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

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

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

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Geeignet für Partitionierungsprobleme:

- ▶ Lösung ist Partition einer gegebenen Menge.

Sequentieller Algorithmus:

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

```
P := {{x1}}
for i := 2 to n
  falls xi zu einem p ∈ P zugefügt werden kann
    P := P \ {p} ∪ {p ∪ {xi}}
  sonst
    P := P ∪ {{xi}}
```

Ein Scheduling-Problem

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

MINIMUM SCHEDULING ON IDENTICAL MACHINES

Instanz: Menge $T = \{t_1, \dots, t_n\}$ von Aufträgen,
für jedes $t_i \in T$: Dauer $\ell_i \in \mathbb{N}$
Anzahl $p \in \mathbb{N}$ von Prozessoren

Lösung: **schedule**: Funktion $f : T \rightarrow \{1, \dots, p\}$

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

List Scheduling

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

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

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

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Verbesserter Algorithmus

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

Sortiere T so dass $\ell_1 \geq \ell_2 \geq \dots \geq \ell_n$,
dann List Scheduling.

Satz

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

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

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

Ein Packungsproblem

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

**Sequentielle
Algorithmen**

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

MINIMUM BIN PACKING

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

Lösung: Partition B_1, \dots, 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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

**Sequentielle
Algorithmen**

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Algorithmus *Next Fit*

```
k := 1 ; B1 := ∅  
for i := 1 to n do  
  if  $\sum B_k + a_i \leq 1$   
    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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

**Sequentielle
Algorithmen**

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Algorithmus *First Fit*

```
k := 1 ; B1 := ∅
for i := 1 to n do
  flag := true
  for j := 1 to k do
    if  $\sum B_j + a_i \leq 1$  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

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

Verbesserung durch Sortieren

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

**Sequentielle
Algorithmen**

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Algorithmus *First Fit Decreasing*

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

Satz

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

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

Nachteil: Algorithmus ist **offline**,
d.h. arbeitet erst, wenn gesamter Input gelesen.

Färbungen von Graphen

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

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

MINIMUM COLORING

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

Sequentielles Färben von Graphen

for $i := 1$ to n
 $\chi(v_i) := \min k \notin \{\chi(u); u \in N(v_i) \cap \{v_1, \dots, v_{i-1}\}\}$

Satz

Sei k die Anzahl der Farben, die der sequentielle Algorithmus bei Reihenfolge v_1, v_2, \dots, 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: $\deg v_1 \geq \deg v_2 \geq \dots \geq \deg v_n.$

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

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 $||V_0| - |V'_0|| = 1$

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 $Ax \geq b$
 $x \geq 0$

Satz

LINEAR PROGRAMMING *ist in* **PO**.

Algorithmen:

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

Äquivalente Varianten des Problems

- ▶ Gleichungen und unbeschränkte Variablen

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

- ▶ 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

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

INTEGER LINEAR PROGRAMMING ist **NP**-schwer

↔ viele Probleme in **NPO** darauf reduzierbar.

Idee:

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

Vertex Cover mittels LP

Approximationsalgorithmen

Einführung

Grundlagen

Methoden zum Entwurf

Greedy

Sequentielle Algorithmen

Lokale Suche

Lineare Programmierung

Dynamische Programmierung

Approximationsklassen

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} :

minimize $\sum_{i=1}^n c_i x_i$
subject to $x_i + x_j \geq 1$ für alle $\{v_i, v_j\} \in E$
 $x_i \geq 0$

Optimale Lösung $x_1^*, \dots, x_n^* \rightsquigarrow$ 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:

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

Koeffizientenvergleich zeigt: $m^* \geq 10$

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

Besser: $m^* \geq 16$

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

↪ duales Programm

Duales Programm

Programm P	Duales Programm 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)

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

Dual complementary slackness condition (CSC):

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

[Einführung](#)

[Grundlagen](#)

[Methoden zum Entwurf](#)

[Greedy](#)

[Sequentielle Algorithmen](#)

[Lokale Suche](#)

Lineare Programmierung

[Dynamische Programmierung](#)

[Approximationsklassen](#)

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

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

und eine abgeschwächte Dual CSC

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

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

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

[Einführung](#)

[Grundlagen](#)

[Methoden zum Entwurf](#)

[Greedy](#)

[Sequentielle Algorithmen](#)

[Lokale Suche](#)

Lineare Programmierung

[Dynamische Programmierung](#)

[Approximationsklassen](#)

Primal-Dual Algorithmus für VERTEX COVER

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

Duales Programm D_{VC} :

$$\begin{aligned} &\text{maximize} && \sum_{\{v_i, v_j\} \in E} y_{i,j} \\ &\text{subject to} && \sum_{j: \{v_i, v_j\} \in E} y_{i,j} \leq c_i \quad (C_i) \\ &&& y_{i,j} \geq 0 \end{aligned}$$

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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

MINIMUM MULTICUT

Instanz: ungerichteter Graph $G = (V, E)$,
Kapazität c_e für $e \in E$,
Menge $\{(s_1, t_1), \dots, (s_k, t_k)\} \subseteq V \times V$ 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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

**Lineare
Programmierung**

Dynamische
Programmierung

Approximationsklassen

Sei p_i der eindeutige Pfad von s_i nach t_i in G

Lineares Programm:

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

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

Primal-Dual Algorithmus für MINIMUM MULTICUT

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

**Lineare
Programmierung**

Dynamische
Programmierung

Approximationsklassen

Duales Programm:

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

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:

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

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 $V = \{v_1, \dots, v_n\}$ absteigend nach Tiefe

$y := \vec{0}$

$C := \emptyset$

für $i := 1, \dots, n$:

 für jedes j mit $\text{lca}(s_j, t_j) = v_i$

 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 $C = \{e_1, \dots, e_m\}$ geordnet wie zugefügt

für $i := m, \dots, 1$

 falls $C \setminus \{e_j\}$ Multicut ist

$C := C \setminus \{e_j\}$

Analyse des Algorithmus

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

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

- ▶ von s_j zu v
- ▶ von v zu t_j

höchstens eine Kante in C .

Satz

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

Dynamische Programmierung

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

Dynamische
Programmierung

Approximationsklassen

- ▶ **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**.
↪ 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, \dots, 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 $M_i \cdot M_{i+1} \cdot \dots \cdot M_j$,
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:

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

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: $M \subseteq \{x_1, \dots, x_k\}$ 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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

**Dynamische
Programmierung**

Approximationsklassen

```
for  $p := 0$  to  $P$        $S(1, p) := A + 1$ 
 $M(1, 0) := \emptyset$ ;     $S(1, 0) := 0$ ;
 $M(1, p_1) := \{x_1\}$ ;   $S(1, p_1) := a_1$ 

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

Approximations-
algorithmen

Einführung

Grundlagen

Methoden zum
Entwurf

Greedy

Sequentielle
Algorithmen

Lokale Suche

Lineare
Programmierung

**Dynamische
Programmierung**

Approximationsklassen

Satz

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

Approximations-Schema:

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

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

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

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 $m_{AS}(r) \geq m^/r$.*