

Approximationsalgorithmen

Wintersemester 2019/20

Annamaria Kovacs R.305, Mahyar Behdju R.311

Conrad Schecker Do. 14 Uhr SR 11

Webseite: <https://ae.cs.uni-frankfurt.de/>
→ teaching
→ Approximationsalgorithmen
(→ Skript)

die Themen

- Entwurfsmethoden I
 - Greedy Algorithmen I
 - Dynamische Programmierung I
 - Lokale suche
 - Lineare Programmierung (Grundlagen)
-

die Themen

- Entwurfsmethoden I
 - Greedy Algorithmen I
 - Dynamische Programmierung I
 - Lokale suche
 - Lineare Programmierung (Grundlagen)
-
- Lineare Programmierung (Dualität)
 - Entwurfsmethoden II
 - Greedy Algorithmen II: Matroide, Sequenzierung
 - Dynamische Programmierung II: Arora's PTAS
 - Branch & Bound

Lineare Programmierung Beispiel

Lineare Programmierung Beispiel

Minimiere $2x_1 - x_2 + 4x_3$ (*lineare Zielfunktion*)

so dass

$$x_1 + x_2 + x_4 \leq 2$$

$$3x_2 - x_3 = 5$$

$$x_3 + x_4 \geq 3 \quad (\textit{lineare Nebenbedingungen})$$

$$x_1 \geq 0$$

$$x_3 \leq 0$$

Gesucht wird eine Belegung der Variablen x_1, x_2, x_3, x_4 die die Nebenbedingungen erfüllt, und die Zielfunktion minimiert.

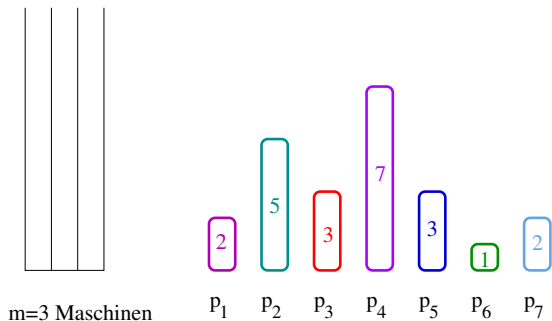
(*Linear Program, LP*)

Bücher:

1. Vazirani: Approximation algorithms
2. Shmoys, Williamson: The design of approximation algorithms
3. Bertsimas, Tsitsiklis: Introduction to linear optimization
4. Moore, Mertens: The nature of computation, Chapter 9.
(unbedingt reinschauen!)
5. Jansen, Margraf: Approximative Algorithmen und Nichtapproximierbarkeit
6. Wanka: Approximationsalgorithmen, Eine Einführung

EINFÜHRUNG

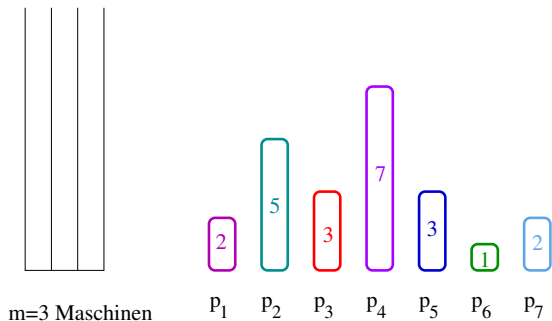
Beispiel 1: Scheduling auf identischen Maschinen



min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Beispiel 1: Scheduling auf identischen Maschinen

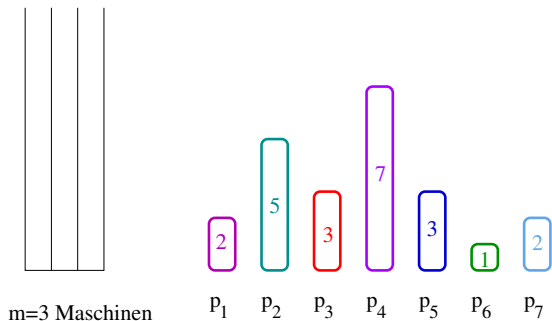


min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

Beispiel 1: Scheduling auf identischen Maschinen



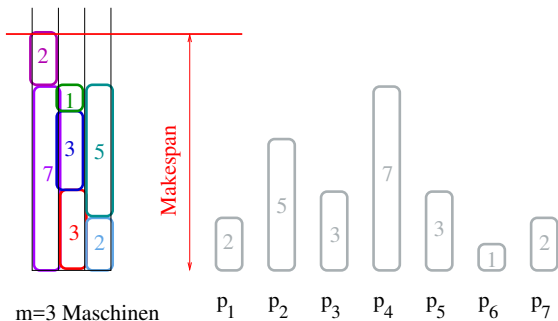
min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

(Makespan = maximale Fertigstellungszeit)

Beispiel 1: Scheduling auf identischen Maschinen



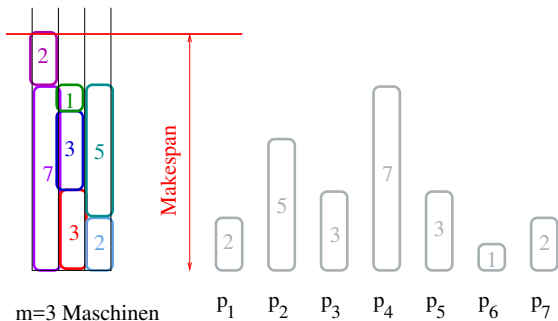
min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

(Makespan = maximale Fertigstellungszeit)

Beispiel 1: Scheduling auf identischen Maschinen



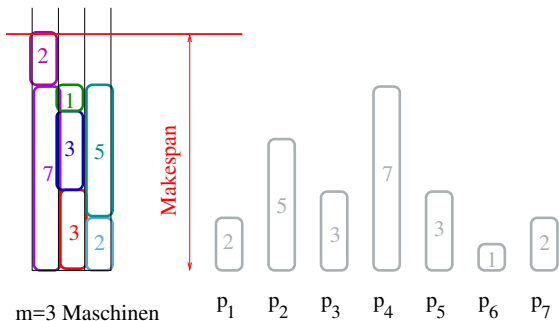
min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

(Makespan = maximale Fertigstellungszeit)

Beispiel 1: Scheduling auf identischen Maschinen



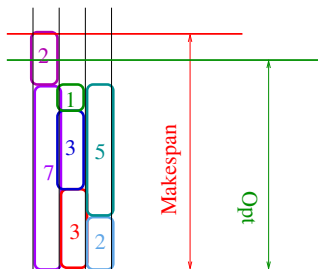
min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

(Makespan = maximale Fertigstellungszeit)

Beispiel 1: Scheduling auf identischen Maschinen



eine $9/8$ -approximative

Lösung

$m=3$ Maschinen

min-SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , und Maschinentzahl m

Ausgabe: Teile jeden Job genau einer Maschine zu so dass der Makespan minimiert wird

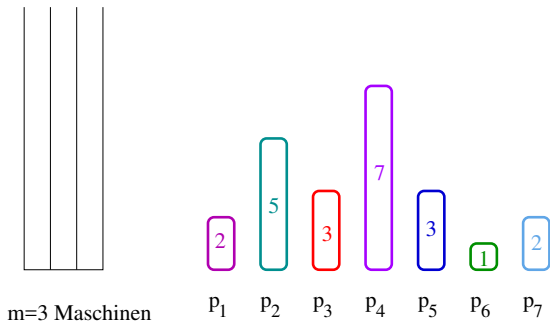
(Makespan = maximale Fertigstellungszeit)

Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat

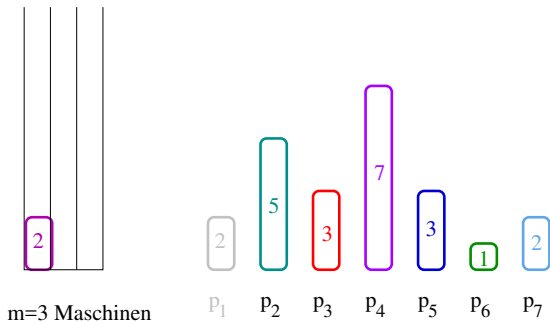
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



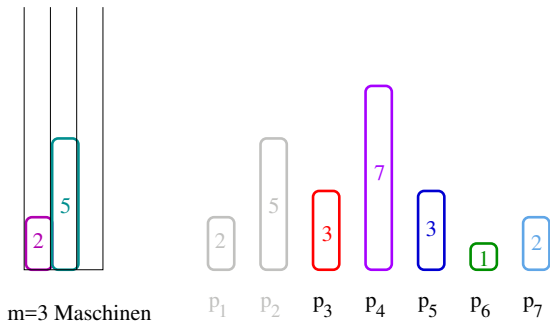
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



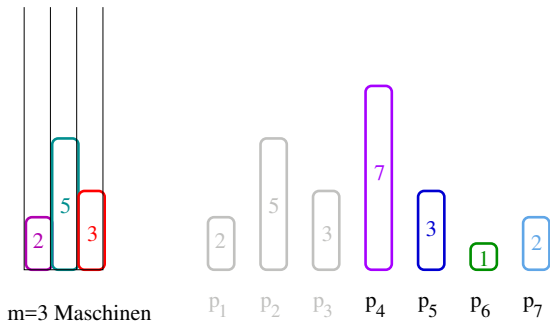
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



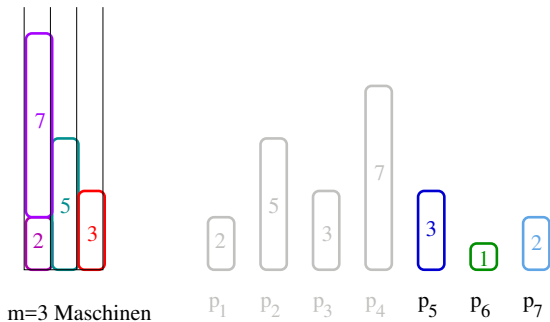
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



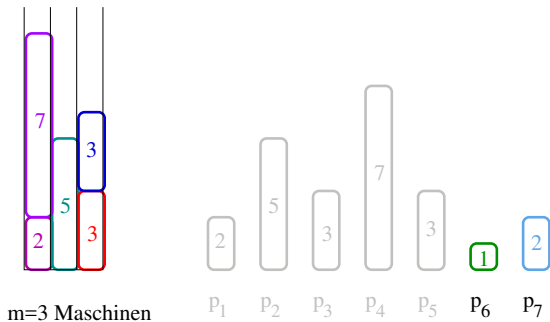
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



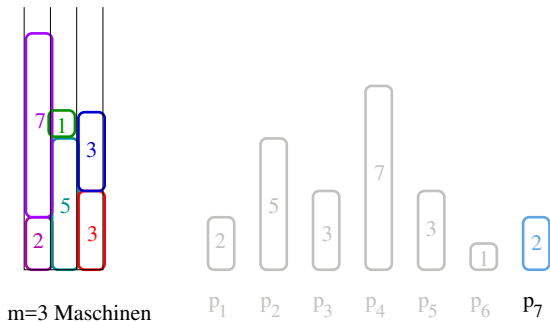
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



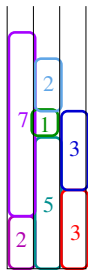
Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat

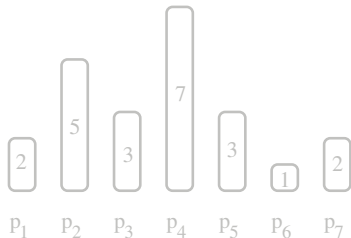


Der (greedy) LIST-Scheduling Algorithmus

- ▶ Weise die Jobs nacheinander der Maschinen zu:
- ▶ gib p_i der Maschine die bisher die kleinste Gesamtlaufzeit von Jobs hat



m=3 Maschinen



Approximationsfaktor von LIST

Theorem 1: Der Approximationsfaktor von LIST ist mindestens

$$2 - \frac{1}{m}.$$

Approximationsfaktor von LIST

Theorem 1: Der Approximationsfaktor von LIST ist mindestens

$$2 - \frac{1}{m}.$$

Theorem 2: Für beliebige Eingabe-Instanz I gilt dass

$$LIST(I) \leq \left(2 - \frac{1}{m}\right)$$

Approximationsfaktor von LIST

Theorem 1: Der Approximationsfaktor von LIST ist mindestens

$$2 - \frac{1}{m}.$$

Theorem 2: Für beliebige Eingabe-Instanz I gilt dass

$$LIST(I) \leq \left(2 - \frac{1}{m}\right)OPT(I).$$

Der Approximationsfaktor von LIST ist somit auch höchstens (also genau) $2 - \frac{1}{m}$.

$OPT(I)$: der minimale Makespan

$LIST(I)$: der Makespan ausgegeben von LIST

Approximationsfaktor von LIST

Theorem 1: Der Approximationsfaktor von LIST ist mindestens

$$2 - \frac{1}{m}.$$

Theorem 2: Für beliebige Eingabe-Instanz I gilt dass

$$LIST(I) \leq \left(2 - \frac{1}{m}\right)OPT(I).$$

Der Approximationsfaktor von LIST ist somit auch höchstens (also genau) $2 - \frac{1}{m}$.

$OPT(I)$: der minimale Makespan

$LIST(I)$: der Makespan ausgegeben von LIST

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: minimaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

sei $\alpha \geq 1$

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: minimaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

sei $\alpha \geq 1$

– α -approximative Lösung:

Lösung mit Zielwert $\leq \alpha \cdot \text{OPT}(I)$

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: minimaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

sei $\alpha \geq 1$

– α -approximative Lösung:

Lösung mit Zielwert $\leq \alpha \cdot \text{OPT}(I)$

– Approximationsfaktor von ALG ist höchstens α wenn
für *jede* Eingabe I

$$\text{ALG}(I) \leq \alpha \cdot \text{OPT}(I)$$

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: minimaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

sei $\alpha \geq 1$

- α -approximative Lösung:

Lösung mit Zielwert $\leq \alpha \cdot \text{OPT}(I)$

- Approximationsfaktor von ALG ist höchstens α wenn für *jede* Eingabe I

$$\text{ALG}(I) \leq \alpha \cdot \text{OPT}(I)$$

- Approximationsfaktor von ALG ist mindestens α wenn für jede $\epsilon > 0$ gibt es *mind. eine* Eingabe I so dass

$$\text{ALG}(I) \geq (\alpha - \epsilon) \cdot \text{OPT}(I)$$

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: minimaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

sei $\alpha \geq 1$

- α -approximative Lösung:

Lösung mit Zielwert $\leq \alpha \cdot \text{OPT}(I)$

- Approximationsfaktor von ALG ist höchstens α wenn für *jede* Eingabe I

$$\text{ALG}(I) \leq \alpha \cdot \text{OPT}(I)$$

- Approximationsfaktor von ALG ist mindestens α wenn für jede $\epsilon > 0$ gibt es *mind. eine* Eingabe I so dass

$$\text{ALG}(I) \geq (\alpha - \epsilon) \cdot \text{OPT}(I)$$

Beispiel 2: max-CLIQUE:

Beispiel 2: max-CLIQUE:

max-CLIQUE

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde in G eine Clique maximaler Größe als Teilgraph.

Beispiel 2: max-CLIQUE:

max-CLIQUE

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde in G eine Clique maximaler Größe als Teilgraph.

Clique: vollständiger (Teil-)Graph

Beispiel 2: max-CLIQUE:

max-CLIQUE

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde in G eine Clique maximaler Größe als Teilgraph.

Clique: vollständiger (Teil-)Graph

- Es gibt überhaupt keine Konstante c s.d. max-CLIQUE effizient c -approximierbar ist (falls $\mathcal{P} \neq \mathcal{NP}$).

Beispiel 2: max-CLIQUE:

max-CLIQUE

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde in G eine Clique maximaler Größe als Teilgraph.

Clique: vollständiger (Teil-)Graph

- Es gibt überhaupt keine Konstante c s.d. max-CLIQUE effizient c -approximierbar ist (falls $\mathcal{P} \neq \mathcal{NP}$).
- Es gibt sogar keinen effizienten Algorithmus mit Approx.-Faktor $n^{(1-\delta)}$ für $\delta > 0$ (wobei $n = |V|$)

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: maximaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

$\alpha \geq 1$

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: maximaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

$\alpha \geq 1$

– α -approximative Lösung:

Lösung mit Zielwert $\geq \text{OPT}(I)/\alpha$

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: maximaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

$\alpha \geq 1$

– α -approximative Lösung:

Lösung mit Zielwert $\geq \text{OPT}(I)/\alpha$

– Approximationsfaktor von ALG ist höchstens α wenn
für *jede* Eingabe I

$$\text{ALG}(I) \geq \frac{\text{OPT}(I)}{\alpha}$$

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: maximaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

$\alpha \geq 1$

- α -approximative Lösung:

Lösung mit Zielwert $\geq \text{OPT}(I)/\alpha$

- Approximationsfaktor von ALG ist höchstens α wenn für *jede* Eingabe I

$$\text{ALG}(I) \geq \frac{\text{OPT}(I)}{\alpha}$$

- Approximationsfaktor von ALG ist mindestens α wenn für jede $\epsilon > 0$ gibt es *mind. eine* Eingabe I so dass

$$\text{ALG}(I) \leq \frac{\text{OPT}(I)}{(\alpha - \epsilon)}$$

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

$\text{OPT}(I)$: maximaler Zielwert über Lösungen für I

$\text{ALG}(I)$: Zielwert der Lösung von ALG

$\alpha \geq 1$

- α -approximative Lösung:

Lösung mit Zielwert $\geq \text{OPT}(I)/\alpha$

- Approximationsfaktor von ALG ist höchstens α wenn für *jede* Eingabe I

$$\text{ALG}(I) \geq \frac{\text{OPT}(I)}{\alpha}$$

- Approximationsfaktor von ALG ist mindestens α wenn für jede $\epsilon > 0$ gibt es *mind. eine* Eingabe I so dass

$$\text{ALG}(I) \leq \frac{\text{OPT}(I)}{(\alpha - \epsilon)}$$

Beispiel 3: min-VERTEX COVER

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante mind. einen Endpunkt in C hat

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

REPEAT

- für eine beliebige Kante $\{u, v\} \in E$ füge u und v zu C hinzu;

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante
mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

REPEAT

- für eine beliebige Kante $\{u, v\} \in E$ füge u und v zu C hinzu;
- entferne $\{u, v\}$ und alle Kanten adjazent zu $\{u, v\}$ aus E ;

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante
mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

REPEAT

- für eine beliebige Kante $\{u, v\} \in E$ füge u und v zu C hinzu;
- entferne $\{u, v\}$ und alle Kanten adjazent zu $\{u, v\}$ aus E ;

UNTIL $E = \emptyset$.

return C als Vertex Cover.

Beispiel 3: min-VERTEX COVER

min-VERTEX COVER

Eingabe: ein Graph $G(V, E)$

Ausgabe: Finde eine Knotenüberdeckung $C \subseteq V$ minimaler Grösse $|C|$

Knotenüberdeckung: Teilmenge der Knoten so dass jede Kante mind. einen Endpunkt in C hat

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

REPEAT

- für eine beliebige Kante $\{u, v\} \in E$ füge u und v zu C hinzu;
- entferne $\{u, v\}$ und alle Kanten adjazent zu $\{u, v\}$ aus E ;

UNTIL $E = \emptyset$.

return C als Vertex Cover.

Beispiel 3: min-VERTEX COVER

Beispiel 3: min-VERTEX COVER

Behauptung: Greedy-VC ist 2-approximativ.

Beispiel 3: min-VERTEX COVER

Behauptung: Greedy-VC ist 2-approximativ.

Approximierbarkeit von min-VERTEX COVER allgemein:

- min-VERTEX COVER ist 2-approximierbar (wie gesehen)

Beispiel 3: min-VERTEX COVER

Behauptung: Greedy-VC ist 2-approximativ.

Approximierbarkeit von min-VERTEX COVER allgemein:

- min-VERTEX COVER ist 2-approximierbar (wie gesehen)
- Es ist nicht bekannt ob min-VC c -approximierbar ist für ein $c < 2$.

Beispiel 3: min-VERTEX COVER

Behauptung: Greedy-VC ist 2-approximativ.

Approximierbarkeit von min-VERTEX COVER allgemein:

- min-VERTEX COVER ist 2-approximierbar (wie gesehen)
- Es ist nicht bekannt ob min-VC c -approximierbar ist für ein $c < 2$.
- min-VC ist nicht c -approximierbar für $c < 10\sqrt{5} - 21 \approx 1.36$.
(ohne beweis)

Beispiel 3: min-VERTEX COVER

Behauptung: Greedy-VC ist 2-approximativ.

Approximierbarkeit von min-VERTEX COVER allgemein:

- min-VERTEX COVER ist 2-approximierbar (wie gesehen)
- Es ist nicht bekannt ob min-VC c -approximierbar ist für ein $c < 2$.
- min-VC ist nicht c -approximierbar für $c < 10\sqrt{5} - 21 \approx 1.36$.
(ohne beweis)

Definition: die Klasse \mathcal{APX}

Definition: die Klasse \mathcal{APX}

\mathcal{APX} ist die Klasse aller NP-Optimierungsprobleme die effiziente c -approximative Algorithmen für irgendeine Konstante c besitzen.

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

Beispiele für *Optimierungsprobleme*

- 1 min-SCHEDULING
- 2 max-CLIQUE
- 3 min-VERTEX COVER

Beispiele für *Optimierungsprobleme*

- 1 min-SCHEDULING
- 2 max-CLIQUE
- 3 min-VERTEX COVER
- 4 max-MATCHING

Entscheidungsversion(en) von Optimierungsproblemen

Entscheidungsversion(en) von Optimierungsproblemen

CLIQUE

Eingabe: ein Graph $G(V, E)$ und eine Zahl $q \in \mathbb{N}$

Ausgabe: Entscheide ob der Graph eine Clique der Grösse q als Teilgraphen besitzt

Entscheidungsversion(en) von Optimierungsproblemen

CLIQUE

Eingabe: ein Graph $G(V, E)$ und eine Zahl $q \in \mathbb{N}$

Ausgabe: Entscheide ob der Graph eine Clique der Grösse q als Teilgraphen besitzt

SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , Maschinenzahl m , und eine Zahl M

Ausgabe: Entscheide ob ein Schedule auf m Maschinen mit Makespan $\leq M$ für diese Jobs existiert

Entscheidungsversion(en) von Optimierungsproblemen

CLIQUE

Eingabe: ein Graph $G(V, E)$ und eine Zahl $q \in \mathbb{N}$

Ausgabe: Entscheide ob der Graph eine Clique der Grösse q als Teilgraphen besitzt

SCHEDULING

Eingabe: n Job-Laufzeiten p_1, p_2, \dots, p_n , Maschinenzahl m , und eine Zahl M

Ausgabe: Entscheide ob ein Schedule auf m Maschinen mit Makespan $\leq M$ für diese Jobs existiert

MATCHING

Eingabe: ein Graph $G(V, E)$

Ausgabe: Entscheide ob ein perfektes Matching existiert, eine unabhängige Kantenmenge die alle Knoten überdeckt

Weitere *Entscheidungsprobleme*

PARTITION

Eingabe: n Zahlen z_1, z_2, \dots, z_n ($z_i \in \mathbb{N}$)

Ausgabe: Entscheide ob es eine Teilmenge der Zahlen gibt (also eine Indexmenge $S \subseteq \{1, 2, \dots, n\}$), so dass

$$\sum_{i \in S} z_i = \frac{\sum_{i=1}^n z_i}{2}$$

Weitere *Entscheidungsprobleme*

PARTITION

Eingabe: n Zahlen z_1, z_2, \dots, z_n ($z_i \in \mathbb{N}$)

Ausgabe: Entscheide ob es eine Teilmenge der Zahlen gibt (also eine Indexmenge $S \subseteq \{1, 2, \dots, n\}$), so dass

$$\sum_{i \in S} z_i = \frac{\sum_{i=1}^n z_i}{2}$$

Weitere Entscheidungsprobleme

PARTITION

Eingabe: n Zahlen z_1, z_2, \dots, z_n ($z_i \in \mathbb{N}$)

Ausgabe: Entscheide ob es eine Teilmenge der Zahlen gibt (also eine Indexmenge $S \subseteq \{1, 2, \dots, n\}$), so dass

$$\sum_{i \in S} z_i = \frac{\sum_{i=1}^n z_i}{2}$$

HAMILTONSCHER KREIS

Eingabe: ein ungerichteter Graph $G(V, E)$

Ausgabe: Entscheide ob ein Kreis in G existiert, der jeden Knoten genau einmal durchläuft

Weitere Entscheidungsprobleme

PARTITION

Eingabe: n Zahlen z_1, z_2, \dots, z_n ($z_i \in \mathbb{N}$)

Ausgabe: Entscheide ob es eine Teilmenge der Zahlen gibt (also eine Indexmenge $S \subseteq \{1, 2, \dots, n\}$), so dass

$$\sum_{i \in S} z_i = \frac{\sum_{i=1}^n z_i}{2}$$

HAMILTONSCHER KREIS

Eingabe: ein ungerichteter Graph $G(V, E)$

Ausgabe: Entscheide ob ein Kreis in G existiert, der jeden Knoten genau einmal durchläuft

Zur Erinnerung: Die Problem-Klasse \mathcal{NP}

Zur Erinnerung: Die Problem-Klasse \mathcal{NP}

ein Entscheidungsproblem gehört zur Klasse \mathcal{NP}



Zur Erinnerung: Die Problem-Klasse \mathcal{NP}

ein Entscheidungsproblem gehört zur Klasse \mathcal{NP}



für jede **JA-Instanz** des Problems gibt es eine 'Lösung'
polynomieller Länge

Zur Erinnerung: Die Problem-Klasse \mathcal{NP}

ein Entscheidungsproblem gehört zur Klasse \mathcal{NP}



für jede **JA-Instanz** des Problems gibt es eine 'Lösung'
polynomieller Länge (diese 'Lösung' heißt **Zeuge (witness)** und ist
allgemein einfach ein polynomiell langer Beweis).

Anhand eines Zeugen (falls durch ein Wunder erraten) ist in
Polynomialzeit nachweisbar dass die Instanz eine JA-Instanz ist!



Zur Erinnerung: Die Problem-Klasse \mathcal{NP}

ein Entscheidungsproblem gehört zur Klasse \mathcal{NP}



für jede **JA-Instanz** des Problems gibt es eine 'Lösung'
polynomieller Länge (diese 'Lösung' heißt **Zeuge (witness)** und ist
allgemein einfach ein polynomiell langer Beweis).

Anhand eines Zeugen (falls durch ein Wunder erraten) ist in
Polynomialzeit nachweisbar dass die Instanz eine JA-Instanz ist!



es gibt eine Nicht-deterministische Turingmaschine mit
polynomieller Laufzeit die genau die JA-Instanzen von P akzeptiert
(Bemerkung: Eine erfolgreiche Berechnung der ND-Turingmaschine
entspricht einem Zeugen.)

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -*Optimierungsproblem*

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}



Die JA-Instanzen der Entscheidungsversion des Problems haben eine polynomiell *verifizierbare* Lösung (sog. Zeugen),

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}



Die JA-Instanzen der Entscheidungsversion des Problems haben eine polynomiell *verifizierbare* Lösung (sog. Zeugen),
die aber nicht effizient gefunden werden soll!

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}



Die JA-Instanzen der Entscheidungsversion des Problems haben eine polynomiell *verifizierbare* Lösung (sog. Zeugen),
die aber nicht effizient gefunden werden soll!



Eine Lösung mit gutem Zielwert, falls vorhanden, ist polynomiell *verifizierbar* (nachweisbar).

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}



Die JA-Instanzen der Entscheidungsversion des Problems haben eine polynomiell *verifizierbare* Lösung (sog. Zeugen),
die aber nicht effizient gefunden werden soll!



Eine Lösung mit gutem Zielwert, falls vorhanden, ist polynomiell *verifizierbar* (nachweisbar).

Die Klasse aller NP-Optimierungsprobleme heißt \mathcal{NPO} .

Wir betrachten \mathcal{NP} -Optimierungsprobleme (die Klasse \mathcal{NPO})

Ein Optimierungsproblem ist ein \mathcal{NP} -Optimierungsproblem



Die Entscheidungsversion des Problems liegt in \mathcal{NP}



Die JA-Instanzen der Entscheidungsversion des Problems haben eine polynomiell *verifizierbare* Lösung (sog. Zeugen),
die aber nicht effizient gefunden werden soll!



Eine Lösung mit gutem Zielwert, falls vorhanden, ist polynomiell *verifizierbar* (nachweisbar).

Die Klasse aller NP-Optimierungsprobleme heißt \mathcal{NPO} .

NP -schwere Optimierungsprobleme

Es ist nicht möglich NP -schwere Optimierungsprobleme *effizient, exakt* zu lösen.

NP -schwere Optimierungsprobleme

Es ist nicht möglich NP -schwere Optimierungsprobleme *effizient*, *exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen

NP -schwere Optimierungsprobleme

Es ist nicht möglich NP -schwere Optimierungsprobleme *effizient*, *exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen

NP -schwere Optimierungsprobleme

Es ist nicht möglich NP -schwere Optimierungsprobleme *effizient, exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen
3. dies für jede Instanz tun

NP -schwere Optimierungsprobleme

Es ist nicht möglich NP -schwere Optimierungsprobleme *effizient*, *exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen
3. dies für jede Instanz tun

Auswege:

1+3 → wir lassen nicht-effiziente Algorithmen zu

NP-schwere Optimierungsprobleme

Es ist nicht möglich NP-schwere Optimierungsprobleme *effizient*, *exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen
3. dies für jede Instanz tun

Auswege:

1+3 → wir lassen nicht-effiziente Algorithmen zu

1+2 → vielleicht entspricht unsere Instanz einem *Spezialfall*,
der in Polynomialzeit lösbar ist...

NP-schwere Optimierungsprobleme

Es ist nicht möglich *NP*-schwere Optimierungsprobleme *effizient, exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen
3. dies für jede Instanz tun

Auswege:

1+3 → wir lassen nicht-effiziente Algorithmen zu

1+2 → vielleicht entspricht unsere Instanz einem *Spezialfall*,
der in Polynomialzeit lösbar ist...

2+3 → wir sind mit *approximativen Lösungen zufrieden*

NP-schwere Optimierungsprobleme

Es ist nicht möglich *NP*-schwere Optimierungsprobleme *effizient*, *exakt* zu lösen.

Ein Algorithmus kann nicht gleichzeitig

1. optimale Lösungen bestimmen
2. in polynomieller Zeit laufen
3. dies für jede Instanz tun

Auswege:

1+3 → wir lassen nicht-effiziente Algorithmen zu

1+2 → vielleicht entspricht unsere Instanz einem *Spezialfall*,
der in Polynomialzeit lösbar ist...

2+3 → wir sind mit *approximativen Lösungen zufrieden*

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$
- (Eingabe-)Instanz: x_0

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$
- (Eingabe-)Instanz: x_0
- x ist eine *Lösung* zu x_0 , falls $L(x_0, x) = \text{YES}$;
 $L()$ heißt Lösungsprädikat

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$
- (Eingabe-)Instanz: x_0
- x ist eine *Lösung* zu x_0 , falls $L(x_0, x) = \text{YES}$;
 $L()$ heißt Lösungsprädikat
- eine *Zielfunktion* $f(x_0, x)$ muss minimiert oder maximiert werden:

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$
- (Eingabe-)Instanz: x_0
- x ist eine *Lösung* zu x_0 , falls $L(x_0, x) = \text{YES}$;
 $L()$ heißt Lösungsprädikat
- eine *Zielfunktion* $f(x_0, x)$ muss minimiert oder maximiert werden:
- $\text{opt} = \text{min}$ oder $\text{opt} = \text{max}$;
- x^* heißt eine *optimale Lösung* zur Instanz x_0 , falls der Zielwert $f(x_0, x^*)$ unter allen Lösungen x optimal (maximal bzw. minimal) ist.

(Notation im Skript)

Die folgende Notation wird im Skript allgemein für beliebiges (unspezifiziertes) Optimierungsproblem verwendet:

- Optimierungsproblem: $P = (\text{opt}, f, L)$
- (Eingabe-)Instanz: x_0
- x ist eine *Lösung* zu x_0 , falls $L(x_0, x) = \text{YES}$;
 $L()$ heißt Lösungsprädikat
- eine *Zielfunktion* $f(x_0, x)$ muss minimiert oder maximiert werden:
- $\text{opt} = \min$ oder $\text{opt} = \max$;
- x^* heißt eine *optimale Lösung* zur Instanz x_0 , falls der Zielwert $f(x_0, x^*)$ unter allen Lösungen x optimal (maximal bzw. minimal) ist.

Definition: Polynomielles Approximationschema (PTAS)

Definition: Polynomielles Approximationsschema (PTAS)

PTAS – Polynomial Time Approximation Scheme

Ein *Polynomielles Approximationsschema (PTAS)* für ein Optimierungsproblem P ist eine Familie $(A_\epsilon)_{\epsilon>0}$ von Approximationsalgorithmen für P ,

Definition: Polynomielles Approximationsschema (PTAS)

PTAS – Polynomial Time Approximation Scheme

Ein *Polynomielles Approximationsschema (PTAS)* für ein Optimierungsproblem P ist eine Familie $(A_\varepsilon)_{\varepsilon>0}$ von Approximationsalgorithmen für P , so dass für jedes ε der zugehörige Algorithmus A_ε $(1 + \varepsilon)$ -approximativ ist.

Definition: Polynomielles Approximationsschema (PTAS)

PTAS – Polynomial Time Approximation Scheme

Ein *Polynomielles Approximationsschema (PTAS)* für ein Optimierungsproblem P ist eine Familie $(A_\varepsilon)_{\varepsilon>0}$ von Approximationsalgorithmen für P , so dass für jedes ε der zugehörige Algorithmus A_ε $(1 + \varepsilon)$ -approximativ ist.

Die Laufzeit von jedem A_ε muss in der Eingabelänge (aber nicht in $1/\varepsilon$) polynomiell sein.

min-SCHEDULING-m

min-SCHEDULING-m

Eingabe: n Jobs mit Laufzeiten p_1, p_2, \dots, p_n

min-SCHEDULING- m

Eingabe: n Jobs mit Laufzeiten p_1, p_2, \dots, p_n

Ausgabe: Ein Schedule der Jobs auf m Maschinen so dass der Makespan minimal ist.

ein PTAS für min-SCHEDULING-m

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

- a. $(1 + \varepsilon)$ -Approximation?

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

- a. $(1 + \varepsilon)$ -Approximation?
- b. polynomielle Laufzeit?

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

- a. $(1 + \varepsilon)$ -Approximation?
- b. polynomielle Laufzeit?

für m Maschinen $k = m/\varepsilon$ funktioniert!

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

- a. $(1 + \varepsilon)$ -Approximation?
- b. polynomielle Laufzeit?

für m Maschinen $k = m/\varepsilon$ funktioniert!

Laufzeit: $\mathcal{O}(n \log n + m^{\frac{m}{\varepsilon}})$

ein PTAS für min-SCHEDULING-m

Sei ε gegeben, wir definieren den Algorithmus A_ε

Eingabe: n Job-Laufzeiten

1. Sortiere die Jobs nach absteigenden Laufzeiten;
2. bestimme einen optimalen Schedule *der größten k Jobs*;
3. ergänze diesen Schedule mit LIST für die übrigen $n - k$ jobs!

Für welches k erhalten wir gleichzeitig

- a. $(1 + \varepsilon)$ -Approximation?
- b. polynomielle Laufzeit?

für m Maschinen $k = m/\varepsilon$ funktioniert!

Laufzeit: $\mathcal{O}(n \log n + m^{\frac{m}{\varepsilon}})$

(sogar $k = (m - 1)/\varepsilon$ ist gut;

Laufzeit: $\mathcal{O}(n \log n + m^{\frac{m-1}{\varepsilon}})$)

Was könnte besser sein als ein PTAS?

(für **NP-schwere** Optimierungsprobleme, wenn $\mathcal{P} \neq \mathcal{NP}$)

Was könnte besser sein als ein PTAS?

(für **NP-schwere** Optimierungsprobleme, wenn $\mathcal{P} \neq \mathcal{NP}$)

- Bzgl. der **Approximation**

Was könnte besser sein als ein PTAS?

(für NP-schwere Optimierungsprobleme, wenn $\mathcal{P} \neq \mathcal{NP}$)

- Bzgl. der Approximation → NICHTS

Was könnte besser sein als ein PTAS?

(für **NP-schwere** Optimierungsprobleme, wenn $\mathcal{P} \neq \mathcal{NP}$)

- Bzgl. der **Approximation** → NICHTS
- Bzgl. der **Laufzeit**?

Was könnte besser sein als ein PTAS?

(für **NP-schwere** Optimierungsprobleme, wenn $\mathcal{P} \neq \mathcal{NP}$)

- Bzgl. der **Approximation** → NICHTS
- Bzgl. der **Laufzeit**?

Manche PTAS sind besser als andere...

Definition: Volles Polynomielles Approximationsschema (FPTAS)

Definition: Volles Polynomielles Approximationsschema (FPTAS)

FPTAS: *Fully Polynomial Time Approximation Scheme*

Definition: Volles Polynomielles Approximationsschema (FPTAS)

FPTAS: *Fully Polynomial Time Approximation Scheme*

Ein *Volles Polynomielles Approximationsschema (FPTAS)* für ein Optimierungsproblem P ist ein polynomielles Approximationsschema $(A_\epsilon)_{\epsilon>0}$ so dass die Laufzeit jedes Algorithmus A_ϵ polynomiell ist in der Eingabelänge und in $1/\epsilon$.

Definition: Volles Polynomielles Approximationsschema (FPTAS)

FPTAS: *Fully Polynomial Time Approximation Scheme*

Ein *Volles Polynomielles Approximationsschema (FPTAS)* für ein Optimierungsproblem P ist ein polynomielles Approximationsschema $(A_\epsilon)_{\epsilon>0}$ so dass die Laufzeit jedes Algorithmus A_ϵ polynomiell ist in der Eingabelänge und in $1/\epsilon$.

PTAS ist die Klasse aller NP-Optimierungsprobleme mit einem PTAS.

FPTAS ist die Klasse aller NP-Optimierungsprobleme mit einem FPTAS.

kein FPTAS

kein FPTAS

1. SCHEDULING-m besitzt sogar ein FPTAS;

kein FPTAS

1. SCHEDULING-m besitzt sogar ein FPTAS;
SCHEDULING besitzt ein PTAS, aber kein FPTAS

kein FPTAS

1. SCHEDULING-m besitzt sogar ein FPTAS;
SCHEDULING besitzt ein PTAS, aber kein FPTAS
2. Oft ist die Suche nach einem FPTAS von vornherein hoffnungslos!

kein FPTAS

1. SCHEDULING-m besitzt sogar ein FPTAS;
SCHEDULING besitzt ein PTAS, aber kein FPTAS
2. Oft ist die Suche nach einem FPTAS von vornherein hoffnungslos!
min-VERTEX COVER oder max-CLIQUE sind triviale Beispiele ohne FPTAS

Polynomiell beschränkte Probleme

Theorem:

Ein NP-vollständiges Optimierungsproblem besitzt *kein* FPTAS,

Polynomiell beschränkte Probleme

Theorem:

Ein NP-vollständiges Optimierungsproblem besitzt *kein* FPTAS, wenn ein Polynom $q()$ existiert so dass für jede Instanz I und für jede Lösung von I

Polynomiell beschränkte Probleme

Theorem:

Ein NP-vollständiges Optimierungsproblem besitzt *kein* FPTAS, wenn ein Polynom $q()$ existiert so dass für jede Instanz I und für jede Lösung von I der Zielwert *ganzzahlig und höchstens* $q(|I|)$ ist.

Polynomiell beschränkte Probleme

Theorem:

Ein NP-vollständiges Optimierungsproblem besitzt *kein* FPTAS, wenn ein Polynom $q()$ existiert so dass für jede Instanz I und für jede Lösung von I der Zielwert *ganzzahlig und höchstens* $q(|I|)$ ist.

(d.h. Der Zielwert ist polynomiell in der Eingabelänge)

Polynomiell beschränkte Probleme

Theorem:

Ein NP-vollständiges Optimierungsproblem besitzt *kein* FPTAS, wenn ein Polynom $q()$ existiert so dass für jede Instanz I und für jede Lösung von I der Zielwert *ganzzahlig und höchstens* $q(|I|)$ ist.

(d.h. Der Zielwert ist polynomiell in der Eingabelänge)

Warum?

Sonst gäbe es bei $\varepsilon = 1/q(|I|)$ optimalen effizienten Algorithmus!

Literatur

Zu den Scheduling-Algorithmen siehe auch

Jansen– Margraf: S. 106, 151

Vazirani: Chapter 10.1

Shmoys– Williamson: Kapitel 2.3, und 3.2

Zur Klasse NP

Moore-Mertens: Chapter 4.1, 4.3

Siehe außerdem

Shmoys– Williamson: Kapitel 1.1

Moore-Mertens: Chapter 9.2