Approximationsalgorithmen

Wintersemester 2018/19

Annamaria Kovacs R.305, Mahyar Behdju R.311

Johannes Wolf Di. 12 Uhr SR 307
Conrad Schecker Di. 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

$$\begin{aligned} x_1 + x_2 + x_4 &\leq 2 \\ 3x_2 - x_3 &= 5 \\ x_3 + x_4 &\geq 3 \\ x_1 &\geq 0 \\ x_3 &\leq 0 \end{aligned}$$

(lineare Nebenbedingungen)

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

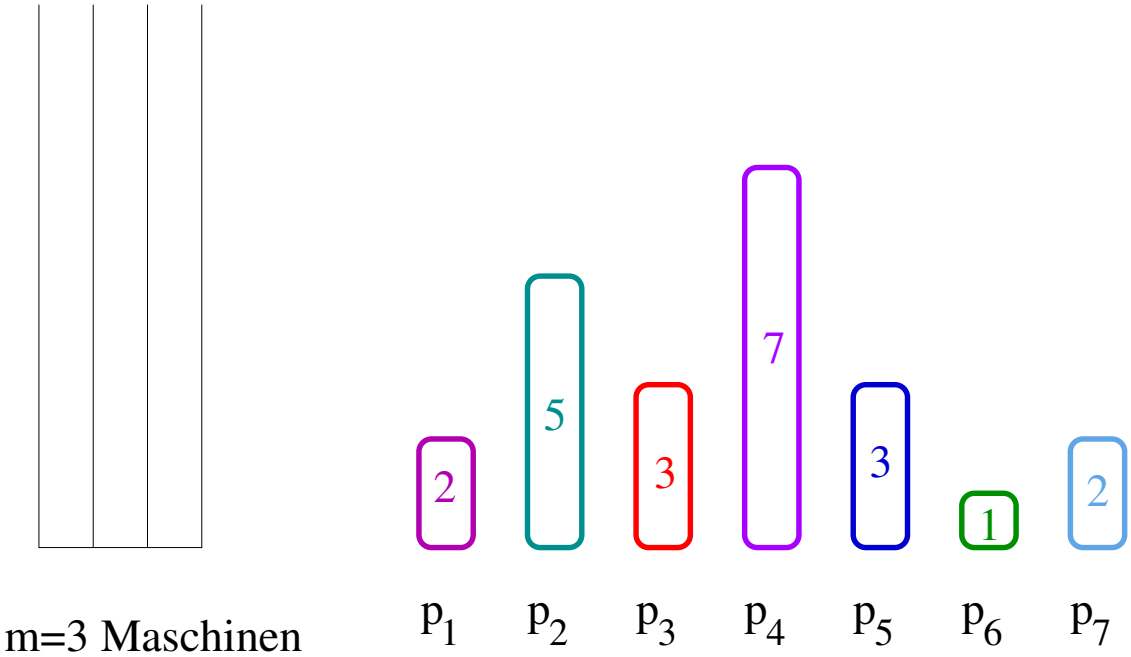
(Linear Program, LP)

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

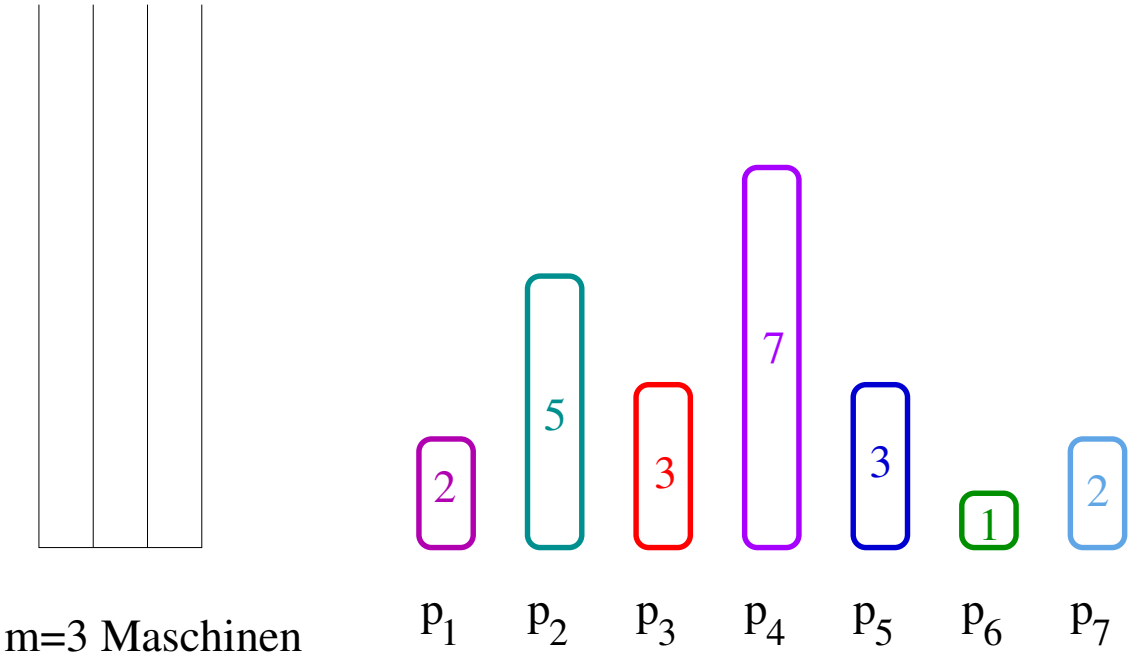
Beispiel: Scheduling auf identischen Maschinen



min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

Beispiel: Scheduling auf identischen Maschinen

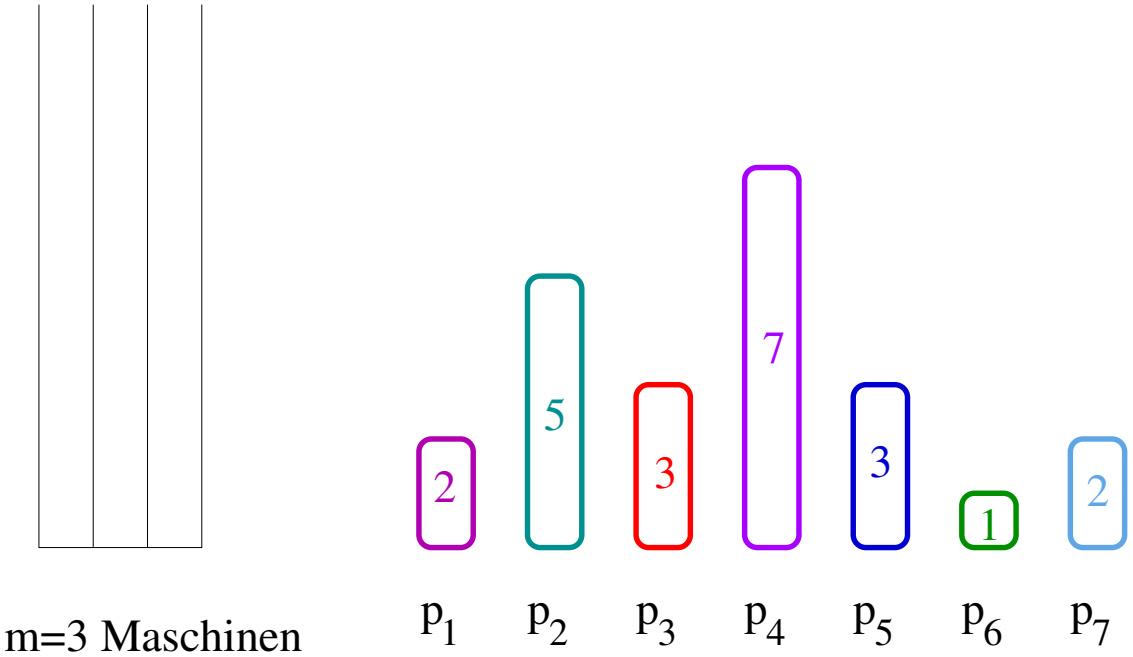


min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

Beispiel: Scheduling auf identischen Maschinen



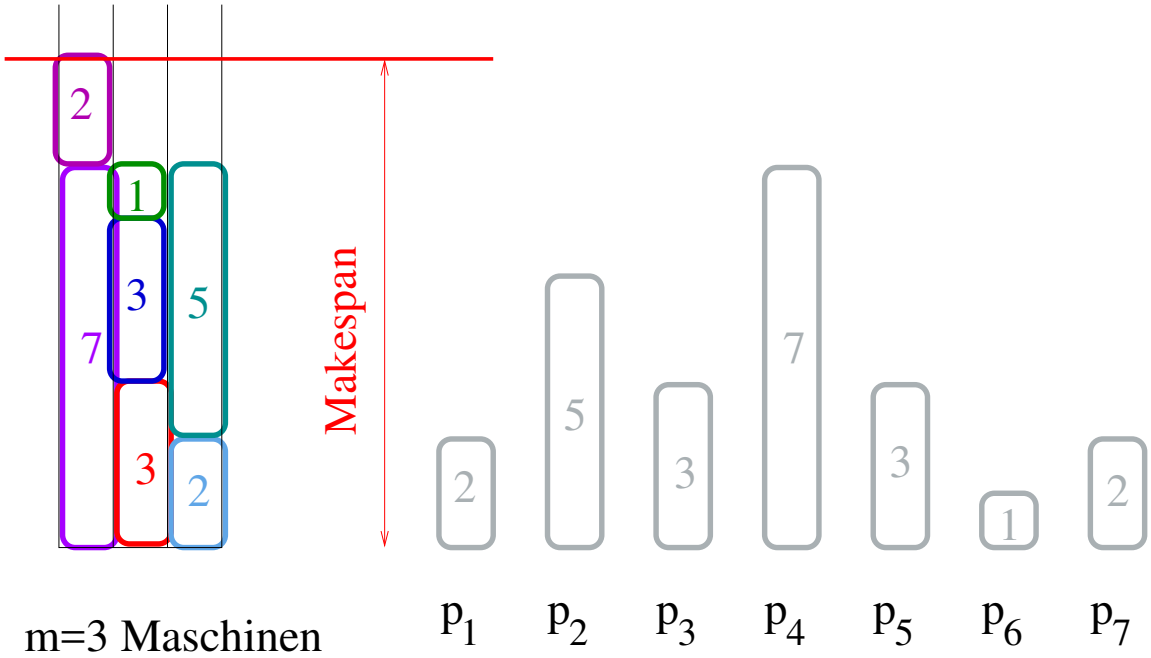
min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

(Makespan = maximale Fertigstellungszeit)

Beispiel: Scheduling auf identischen Maschinen



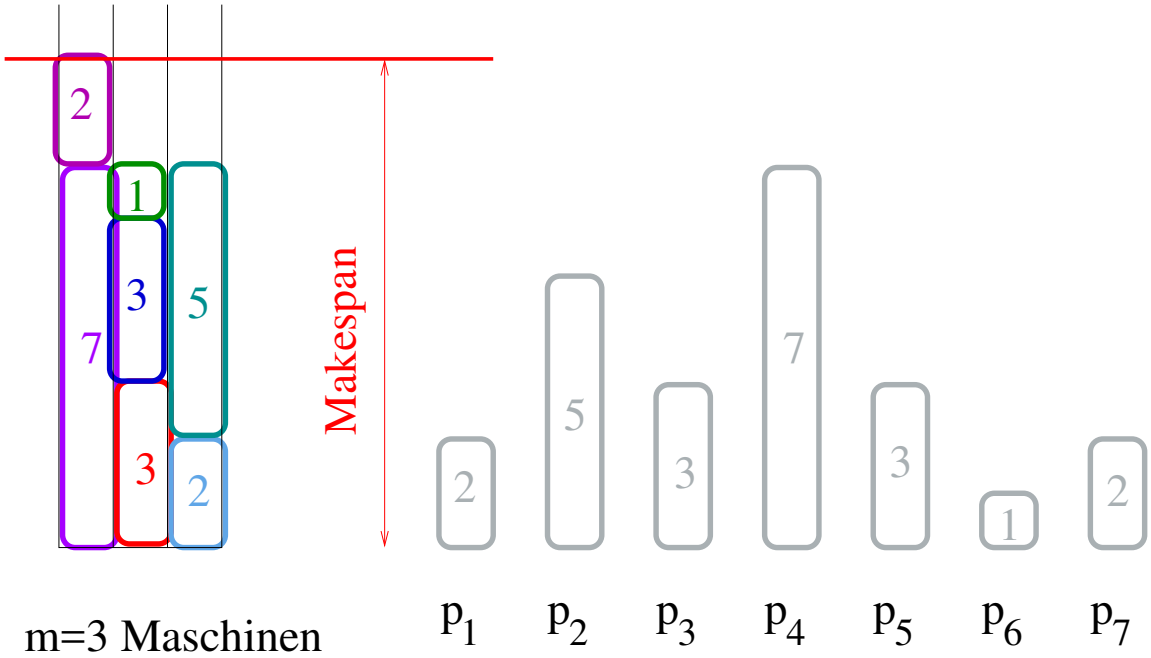
min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

(Makespan = maximale Fertigstellungszeit)

Beispiel: Scheduling auf identischen Maschinen



min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

(Makespan = maximale Fertigstellungszeit)

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

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

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

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

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

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

Clique: vollständiger (Teil-)Graph

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

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

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

Clique: vollständiger (Teil-)Graph

3 min-VERTEX COVER

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

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

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

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

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

Clique: vollständiger (Teil-)Graph

3 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

Beispiele für *Optimierungsprobleme*

1 min-SCHEDULING

2 max-CLIQUE

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

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

Clique: vollständiger (Teil-)Graph

3 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

4 max-MATCHING

Entscheidungsversion(en) von Optimierungsproblemen

Entscheidungsversion(en) von Optimierungsproblemen

MATCHING

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

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

Entscheidungsversion(en) von Optimierungsproblemen

MATCHING

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

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

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

MATCHING

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

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

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 $p_1, p_2, \ldots p_n$, Maschinenzahl m, und eine Zahl M

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

Weitere *Entscheidungsprobleme*

PARTITION

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

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

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

Weitere *Entscheidungsprobleme*

PARTITION

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

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

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

Weitere *Entscheidungsprobleme*

PARTITION

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

Ausgabe: Entscheide ob es eine Teilmenge der Zahlen gibt (also eine Indexmenge $S \subseteq \{1, 2, \ldots n\}$), 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 $z_1, z_2, \ldots z_n$ $(z_i \in \mathbb{N})$

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

$\Updownarrow$

für jede JA-Instanz des Problems kann uns ein Zauberer eine 'Lösung' polynomieller Länge ins Ohr flüstern

(Diese 'Lösung' heißt *Zeuge (witness)*).

Anhand eines Zeugen ist in Polynomialzeit nachweisbar dass die Instanz eine JA-Instanz ist!

$\Updownarrow$

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

ein Entscheidungsproblem gehört zur Klasse $\mathcal{NP}$

$\Updownarrow$

für jede JA-Instanz des Problems kann uns ein Zauberer eine 'Lösung' polynomieller Länge ins Ohr flüstern

(Diese 'Lösung' heißt *Zeuge (witness)*).

Anhand eines Zeugen ist in Polynomialzeit nachweisbar dass die Instanz eine JA-Instanz ist!

$\Updownarrow$

es gibt eine Nicht-deterministische Turingmaschine mit polynomieller Laufzeit die genau die JA-Instanzen von P akzeptiert

(Bemerkung: Die ND-Turingmaschine *rät* und dann verifiziert einen Zeugen in Polynomialzeit)

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*

$$\Updownarrow$$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

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

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

$\Updownarrow$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

$\Updownarrow$

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*

$\Updownarrow$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

$\Updownarrow$

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*

$\Updownarrow$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

$\Updownarrow$

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

$\Updownarrow$

Eine 'gute' Lösung, falls vorhanden, ist polynomiell *verifizierbar* (nachweisbar).

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

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

$\Updownarrow$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

$\Updownarrow$

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

$\Updownarrow$

Eine 'gute' Lösung, 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*

$\Updownarrow$

Die Entscheidungsversion des Problems liegt in $\mathcal{NP}$

$\Updownarrow$

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

$\Updownarrow$

Eine 'gute' Lösung, falls vorhanden, ist polynomiell *verifizierbar* (nachweisbar).

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

(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: $P = (\mathrm{opt}, \mathrm{f}, \mathrm{L})$

(Notation im Skript)

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

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

(Notation im Skript)

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

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

  $L()$ heißt Lösungsprädikat

(Notation im Skript)

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

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

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

  $L()$ heißt Lösungsprädikat
- eine *Zielfunktion* $f(x_0, x)$ muss minimiert oder maximiert werden:
- $\mathrm{opt} = \min$ oder $\mathrm{opt} = \max$;
- 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.

(Notation im Skript)

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

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

  $L()$ heißt Lösungsprädikat
- eine *Zielfunktion* $f(x_0, x)$ muss minimiert oder maximiert werden:
- $\mathrm{opt} = \min$ oder $\mathrm{opt} = \max$;
- 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.

(NP-Optimierungsproblem im Skript)

Ein Optimierungsproblem ist ein $\mathcal{NP}$-*Optimierungsproblem*, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin die Folgenden in polynomieller Laufzeit berechenbar sind:

(NP-Optimierungsproblem im Skript)

Ein Optimierungsproblem ist ein $\mathcal{NP}$-*Optimierungsproblem*, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin die Folgenden in polynomieller Laufzeit berechenbar sind:

- ob x_0 eine Instanz ist;

(NP-Optimierungsproblem im Skript)

Ein Optimierungsproblem ist ein $\mathcal{NP}$-*Optimierungsproblem*, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin die Folgenden in polynomieller Laufzeit berechenbar sind:

- ob x_0 eine Instanz ist;
- ob *gegebenes* x eine Lösung zu x_0 ist;

(NP-Optimierungsproblem im Skript)

Ein Optimierungsproblem ist ein $\mathcal{NP}$-*Optimierungsproblem*, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin die Folgenden in polynomieller Laufzeit berechenbar sind:

- ob x_0 eine Instanz ist;
- ob *gegebenes* x eine Lösung zu x_0 ist;
- berechnung des Zielwertes $f(x_0, x)$ für gegebene x und x_0.

(NP-Optimierungsproblem im Skript)

Ein Optimierungsproblem ist ein $\mathcal{NP}$-*Optimierungsproblem*, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin die Folgenden in polynomieller Laufzeit berechenbar sind:

- ob x_0 eine Instanz ist;
- ob *gegebenes* x eine Lösung zu x_0 ist;
- berechnung des Zielwertes $f(x_0, x)$ für gegebene x und x_0.

polynomiell: durch irgendein festgelegtes Polynom der Eingabelänge $|x_0|$ nach oben beschränkt zB. $\mathcal{O}(|x_0|^7)$ ist

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 $\rightarrow$ 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 $\rightarrow$ wir lassen nicht-effiziente Algorithmen zu

1+2 $\rightarrow$ 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 $\rightarrow$ wir lassen nicht-effiziente Algorithmen zu

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

2+3 $\rightarrow$ 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

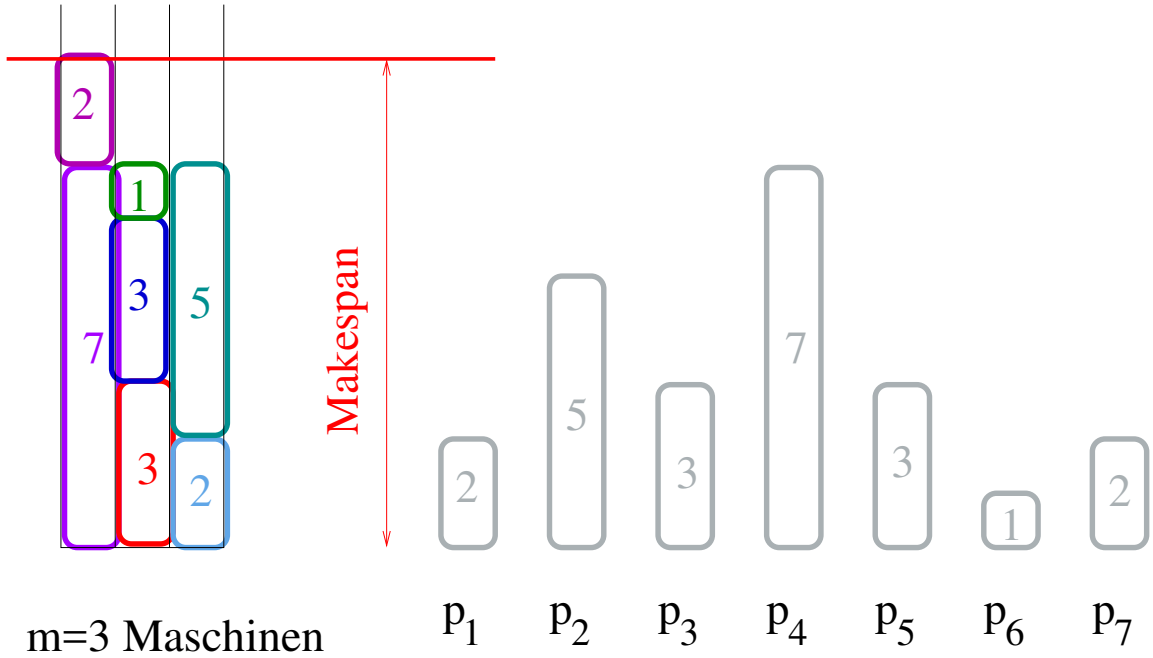
Auswege:

1+3 $\rightarrow$ wir lassen nicht-effiziente Algorithmen zu

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

2+3 $\rightarrow$ wir sind mit *approximativen Lösungen zufrieden*

Beispiel: Scheduling auf identischen Maschinen



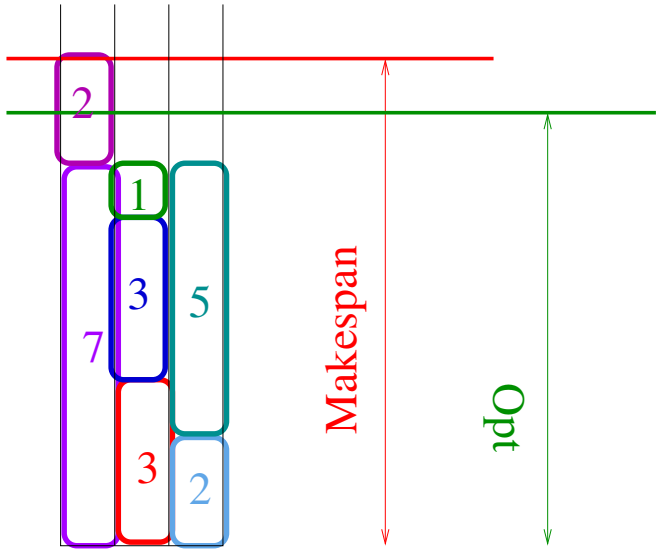
min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

(Makespan = maximale Fertigstellungszeit)

Beispiel: Scheduling auf identischen Maschinen



m=3 Maschinen

eine 9/8 −approximative

Lösung

min-SCHEDULING

Eingabe: n Job-Laufzeiten $p_1, p_2, \ldots p_n$, und Maschinenzahl m

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

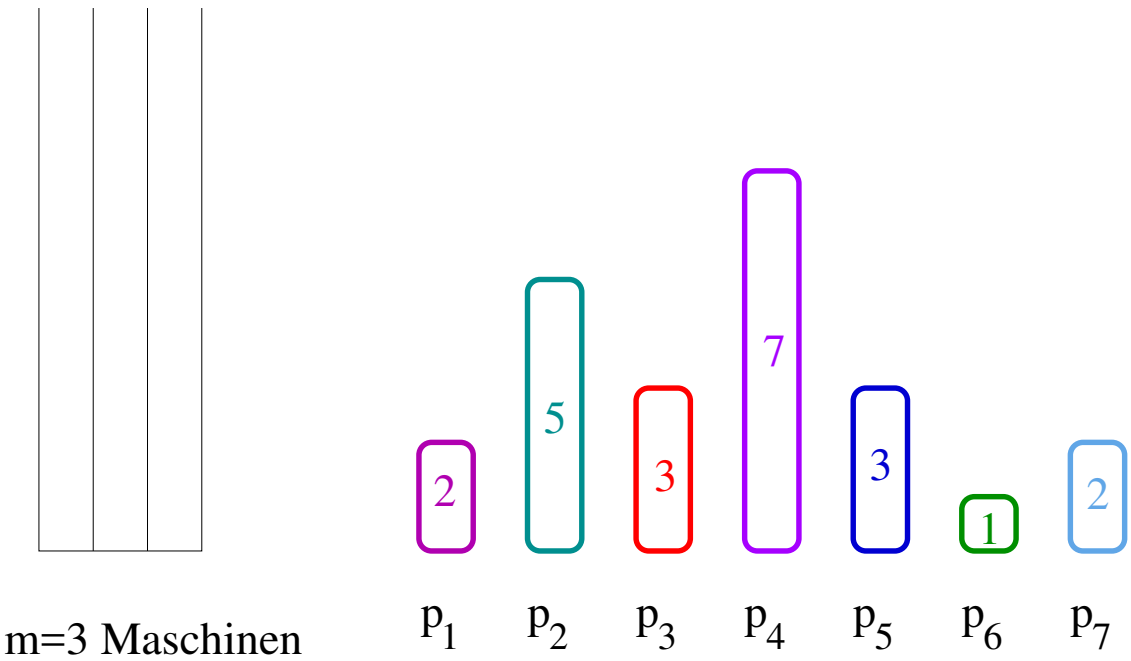
(Makespan = maximale Fertigstellungszeit)

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

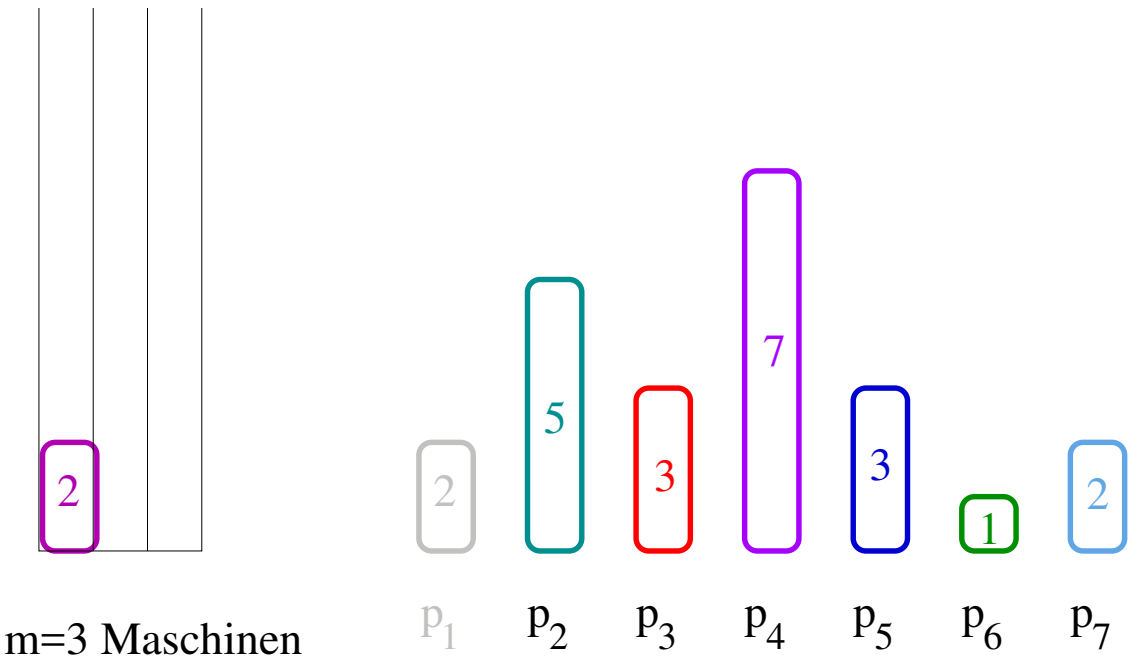
Der (greedy) LIST-Scheduling Algorithmus

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



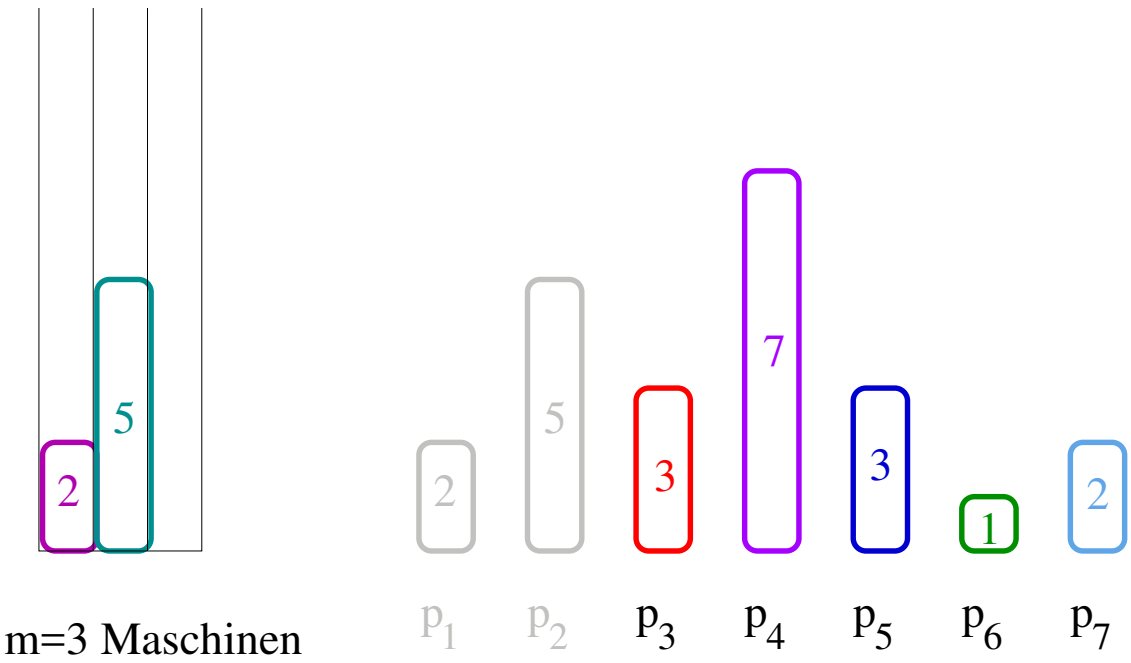
Der (greedy) LIST-Scheduling Algorithmus

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



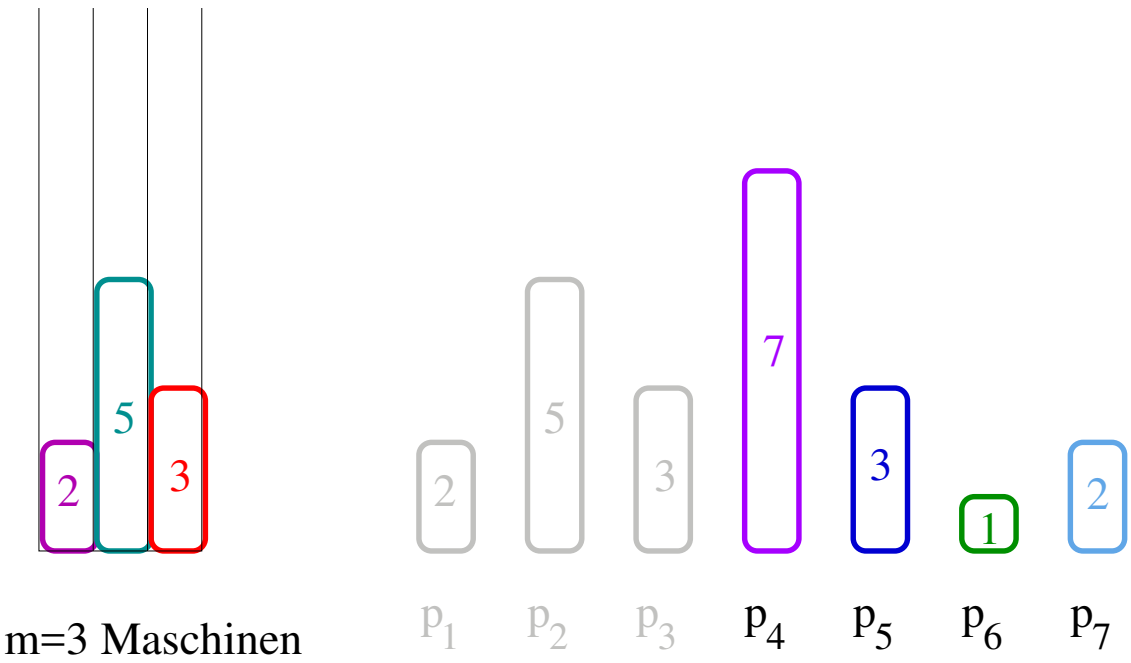
Der (greedy) LIST-Scheduling Algorithmus

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



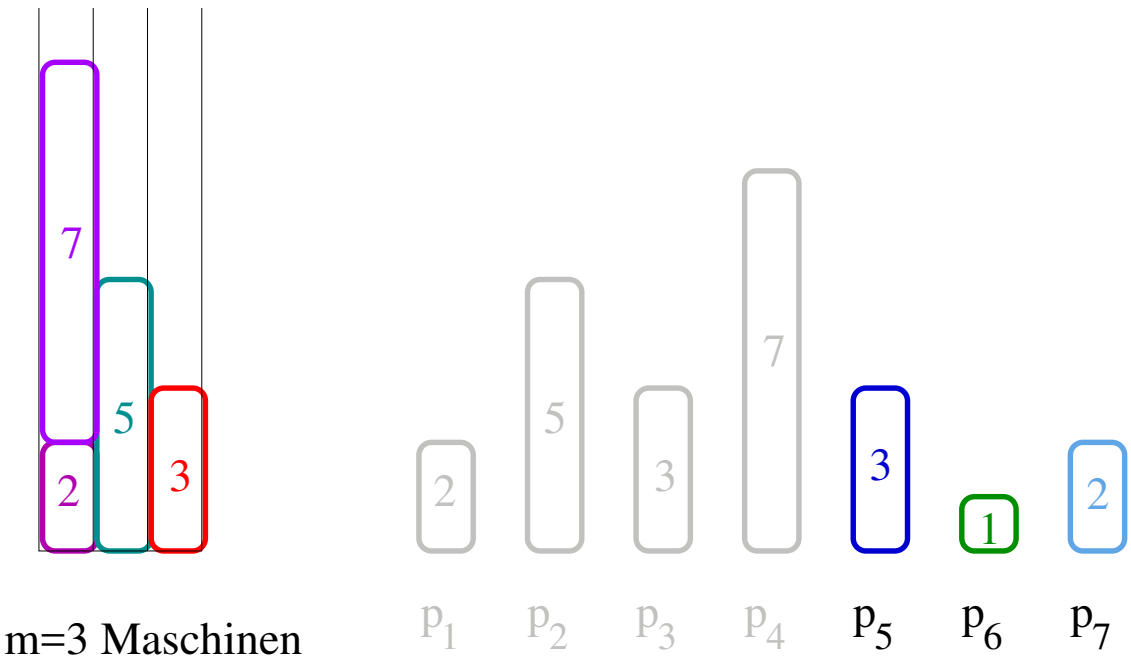
Der (greedy) LIST-Scheduling Algorithmus

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



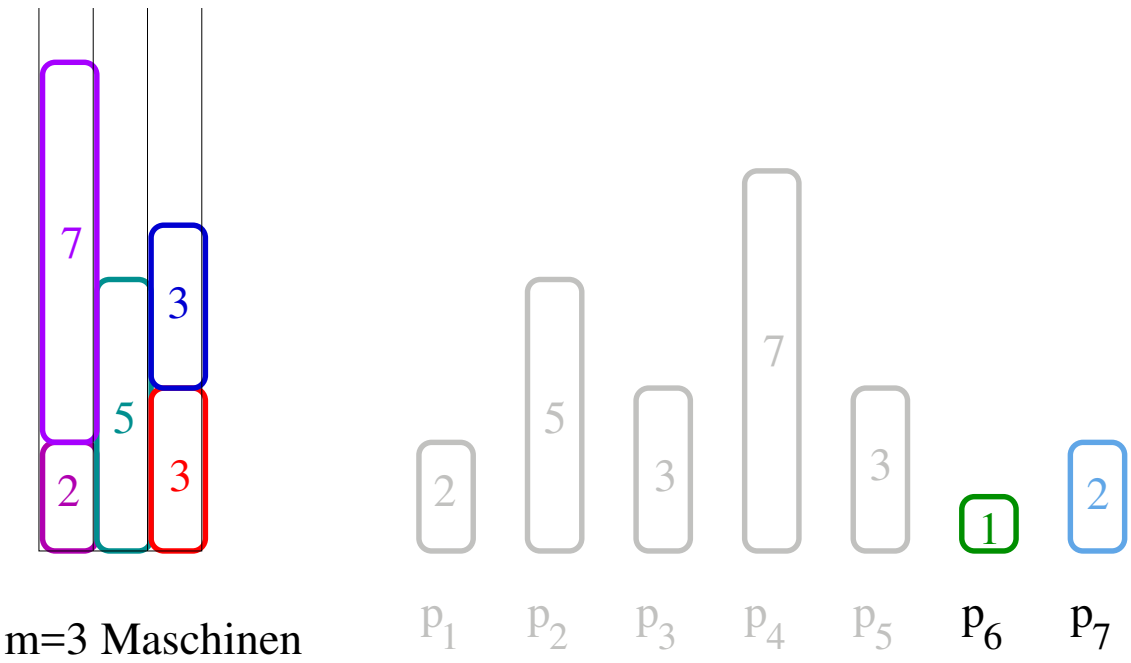
Der (greedy) LIST-Scheduling Algorithmus

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



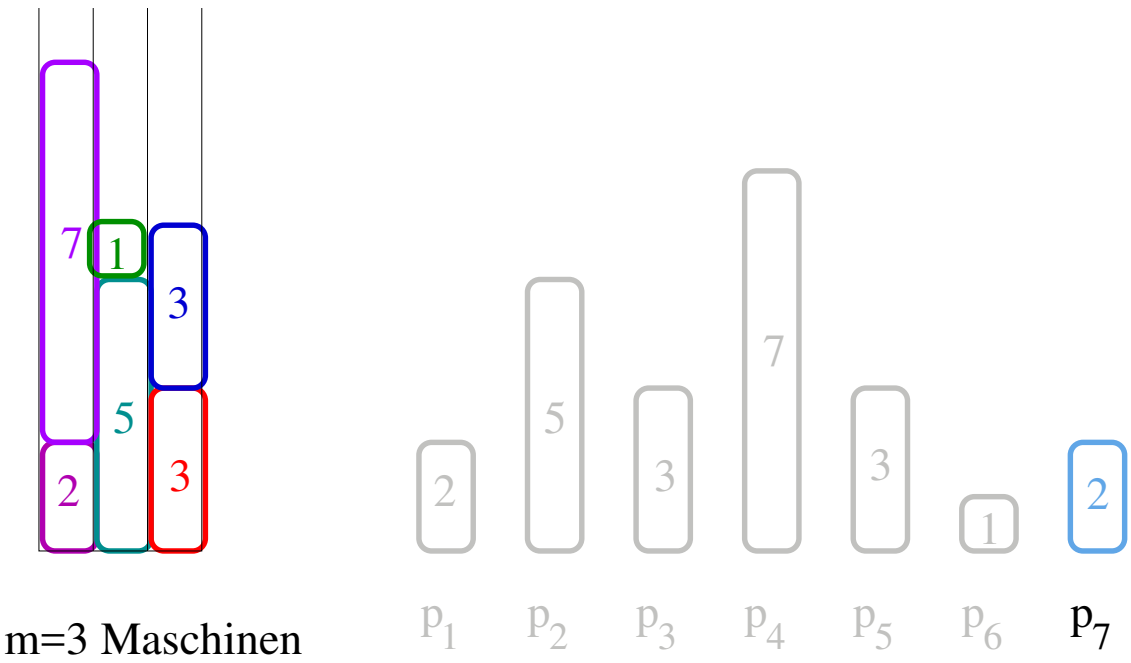
Der (greedy) LIST-Scheduling Algorithmus

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



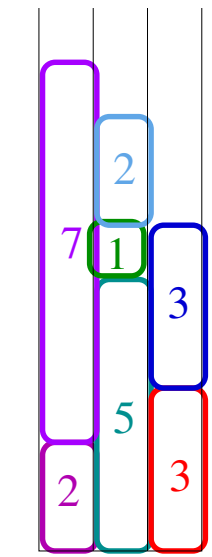
Der (greedy) LIST-Scheduling Algorithmus

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

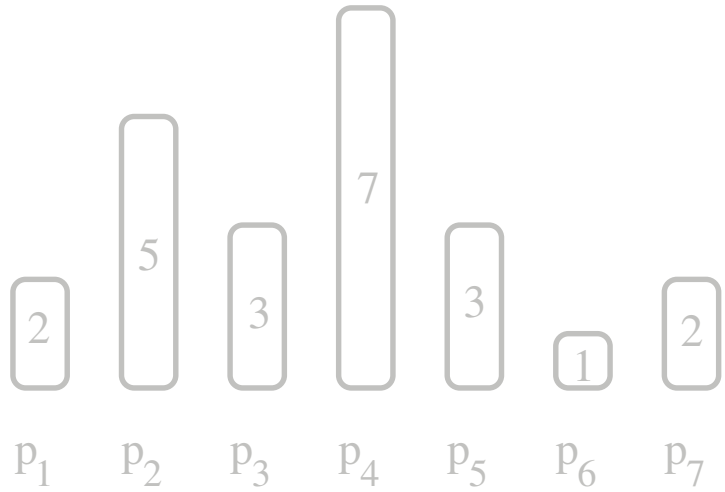


Der (greedy) LIST-Scheduling Algorithmus

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



m=3 Maschinen



(Wir denken an LIST als an einen *offline* Algorithmus.)

Approximationsfaktor von LIST

Theorem 1: Für beliebige Eingabe-Instanz I gilt dass

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

Der Approximatiosfaktor von LIST ist somit höchstens $2 - \frac{1}{m}$.

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

Approximationsfaktor von LIST

Theorem 1: Für beliebige Eingabe-Instanz I gilt dass

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

Der Approximatiosfaktor von LIST ist somit höchstens $2 - \frac{1}{m}$.

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

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

Konstante Approximation auch für min-VERTEX COVER

Konstante Approximation auch für min-VERTEX COVER

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Konstante Approximation auch für min-VERTEX COVER

Der Algorithmus Greedy Vertex Cover

Eingabe: $G(V, E)$

Setze $C = \emptyset$

Konstante Approximation auch für min-VERTEX COVER

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;

Konstante Approximation auch für min-VERTEX COVER

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;

Konstante Approximation auch für min-VERTEX COVER

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.

Konstante Approximation auch für min-VERTEX COVER

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.

Behauptung: Greedy-VC ist 2-approximativ.

Konstante Approximation auch für min-VERTEX COVER

Konstante Approximation auch für min-VERTEX COVER

Fakten:

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

Konstante Approximation auch für min-VERTEX COVER

Fakten:

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

Konstante Approximation auch für min-VERTEX COVER

Fakten:

- 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)

Konstante Approximation auch für min-VERTEX COVER

Fakten:

- 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)

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

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

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

sei $\alpha \geq 1$

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

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

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

sei $\alpha \geq 1$

- α-approximative Lösung:

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

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

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

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

sei $\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\leq \alpha \cdot \mathrm{OPT}(I)$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

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

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

sei $\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\leq \alpha \cdot \mathrm{OPT}(I)$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

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

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

Approximationsfaktor (Minimierungsproblem)

I : Eingabe-Instanz

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

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

sei $\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\leq \alpha \cdot \mathrm{OPT}(I)$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

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

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

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

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

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

$\alpha \geq 1$

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

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

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

$\alpha \geq 1$

- α-approximative Lösung:

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

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

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

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

$\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\geq \mathrm{OPT}(I)/\alpha$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

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

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

$\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\geq \mathrm{OPT}(I)/\alpha$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

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

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

Approximationsfaktor (Maximierungsproblem)

I : Eingabe-Instanz

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

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

$\alpha \geq 1$

- α-approximative Lösung:
  Lösung mit Zielwert $\geq \mathrm{OPT}(I)/\alpha$
- Approximationsfaktor von ALG ist höchstens α wenn
  für *jede* Eingabe I

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

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

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

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.

Keine konstante Approximation: max-CLIQUE:

Keine konstante Approximation: max-CLIQUE:

Fakten (ohne Beweis):

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

Keine konstante Approximation: max-CLIQUE:

Fakten (ohne Beweis):

- 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 $n = |V|$)

Definition: Polynomielles Approximationsschema (PTAS)

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,

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 $p_1, p_2, \ldots p_n$

min-SCHEDULING-m

Eingabe: n Jobs mit Laufzeiten $p_1, p_2, \ldots p_n$

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 $\rightarrow$ NICHTS

Was könnte besser sein als ein PTAS?

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

- Bzgl. der Approximation $\rightarrow$ NICHTS
- Bzgl. der Laufzeit?

Was könnte besser sein als ein PTAS?

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

- Bzgl. der Approximation $\rightarrow$ 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 $(A_\varepsilon)_{\varepsilon>0}$ so dass die Laufzeit jedes Algorithmus A_ε polynomiell ist in der Eingabelänge und in $1/\varepsilon$.

Definition: Volles Polynomielles Approximationsschema (FPTAS)

FPTAS: *Fully Polynomial Time Approximation Scheme*

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

$\mathcal{PTAS}$ ist die Klasse aller NP-Optimierungsprobleme mit einem PTAS.

$\mathcal{FPTAS}$ 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 von jeder 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 von jeder 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 von jeder 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 von jeder 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