EINFÜHRUNG, GRUNDBEGRIFFE

a) Beispiel: LIST-Scheduling auf identischen Maschinen

Das Problem:

min-SCHEDULING (das Problem bezeichnet man auch mit $P||C_{max}$)

Eingabe: m die Anzahl der Maschinen (Prozessoren)

n die Anzahl der jobs

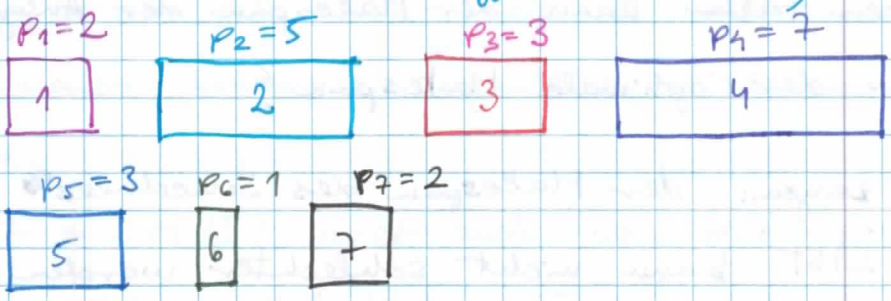
$p_1, p_2, \dots, p_n$ die Laufzeiten der n jobs ($p_j \in \mathbb{R}$ oder $p_j \in \mathbb{N}$)

Aufgabe: Teile jeden Job genau einer Maschine zu, so dass der Makespan (die maximale Fertigstellungszeit) minimiert wird.

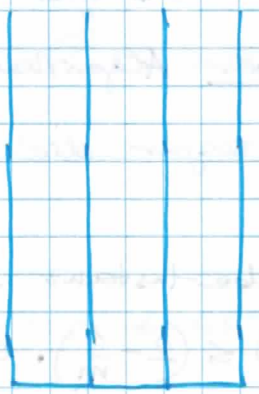
(Wir betrachten das Problem als offline Problem)

Illustration:

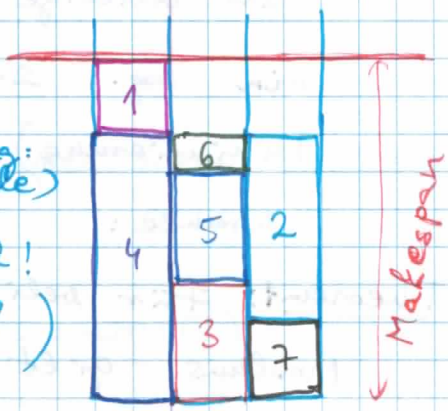
Jobs:



$m = 3$ Maschinen:



eine Zuteilung: (Schedule) (nicht optimal! Warum?)



E 2.

(Mit $p_1=2$, $p_2=5$, $p_3=3$, $p_4=7$, $p_5=3$, $p_6=1$, $p_7=2$
war diese eine $\frac{9}{8}$-approximative Lösung)
weil Opt = 8

Der Algorithmus LIST-Scheduling

nimmt die Jobs einen nach dem Anderen (in irgendeiner Reihenfolge), und weist jeden Job einer Maschine zu, die bis dahin die geringste Gesamtlaufzeit von Jobs hat.

(Beachte: LIST ist ein Greedy (gieriger) Algorithmus: die getroffenen Entscheidungen bzgl. eines Teils der Instanz werden später nicht revidiert.)

(LIST kann auch als online Algorithmus betrachtet/verwendet werden, aber uns interessiert dieser Aspekt hier nicht.)

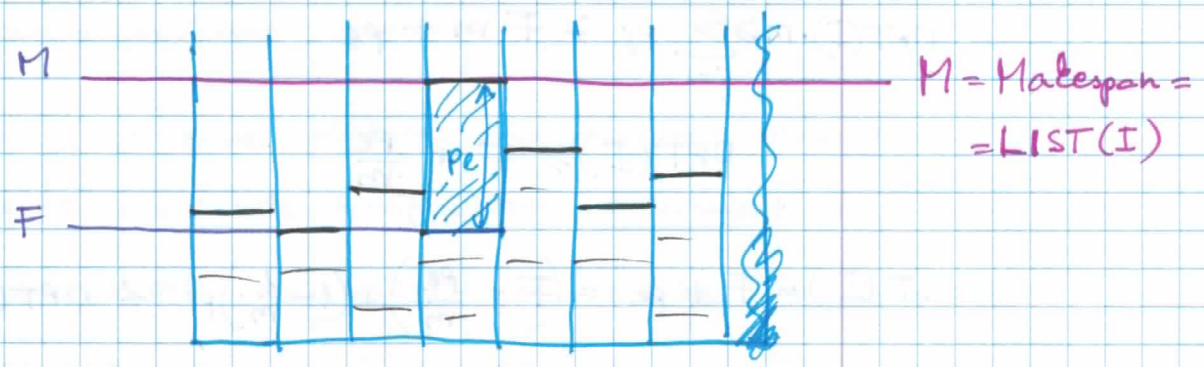
- LIST gibt im Allgemeinen keine optimale Lösung aus (z.B. auch für die obige Instanz (Eingabe) nicht), und ist ein Approximationsalgorithmus von LIST,
Um welchen Faktor kann der Makespan der Ausgabe höher sein, als der optimale Makespan?

→ Wir zeigen: der Makespan des Schedules ausgegeben von LIST kann nicht schlechter werden als $2 \cdot Opt(I)$ für beliebige Eingabe-Instanz I. LIST ist also ein sog. 2-approximativer Algorithmus (für Makespan-Minimierung). Es gilt sogar die folgende schärfere Schranke:

Theorem 1: Für beliebige Eingabe-Instanz I des Scheduling-Problems gilt $LIST(I) \leq \left(2-\frac{1}{m}\right) \cdot OPT(I)$,
wobei OPT(I) der optimale (minimale) Makespan, und

LIST(I) der Makespan des Schedules ausgegeben von LIST ist, für die Instanz I.
Der Approximationsfaktor von LIST ist also höchstens $2-\frac{1}{m}$

Beweis:



Sei I eine beliebige Instanz des SCHEDULING Problems:
Sei $I=(p_1, p_2, p_3 \dots, p_e, \dots, p_n)$ die Folge von Job-Laufzeiten für LIST, und pe sei ein Job mit der höchsten Fertigstellungszeit, also mit Fertigstellungszeit M, in der Ausgabe von LIST.
Sei F die Fertigstellungszeit der Maschine von pe ohne pe, so dass $M = F + p_e$ gilt.

Wir zeigen zuerst $M \leq 2 \cdot OPT(I)$.

$p_e \leq OPT(I)$ klar, weil pe auch in einem optimalen Schedule irgendeiner Maschine zugewiesen wird.

$F \leq OPT(I)$ gilt auch, weil die Gesamtlaufzeit aller Jobs mindestens $m \cdot F$ ist.

$$m \cdot OPT(I) \geq \sum_{j=1}^{n} p_j \geq m \cdot F$$

(die Maschinen sind bis Zeit F voll, siehe Abbildung)

Wir erhalten:

$$LIST(I) = M = F + p_e \leq OPT(I) + OPT(I) = 2 \cdot OPT(I).$$

Eh.

(Warum gilt sogar $LIST(I) \leq \left(2-\frac{1}{m}\right) \cdot OPT(I)$?

Da die Maschinen bis Zeit F voll sind ohne p_e,

$$OPT(I) \cdot m \geq \sum_{j=1}^{n} p_j \geq F \cdot m + p_e$$

$$OPT(I) \geq F + \frac{p_e}{m}$$

$$LIST(I) = F + p_e = \left(F + \frac{p_e}{m}\right) + \left(1-\frac{1}{m}\right) p_e \leq OPT(I) + \left(1-\frac{1}{m}\right) \cdot OPT(I) =$$

$$= \left(2 - \frac{1}{m}\right) OPT(I) \quad)$$

□

Könnte es sein, dass LIST sogar einen besseren Approximationsfaktor als $2-\frac{1}{m}$ hat, nur unsere Analyse nicht präzise genug war? NEIN!

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

Wie beweist man Theorem 2? Muss LIST für jede Instanz so schlechten Makespan erreichen $\left(2-\frac{1}{m}\right) \cdot OPT(I)$? Natürlich gibt es viele Instanzen, für die LIST viel besser abschneidet. Denken wir an eine Instanz mit Jobs der gleichen Größe. Oder an unsere Illustration mit 7 Jobs. Für den Beweis genügt eine einzige Instanz I mit $\frac{LIST(I)}{OPT(I)} = 2-\frac{1}{m}$

Beweis: Wir betrachten die Instanz I mit $m \cdot (m-1)$ Jobs der Größe $p_j = 1$, und einen letzten Job mit $p_n = m$.

Dann ist $OPT(I) = m$ und $LIST(I) = m-1+m = 2m-1$

Warum?

□

b.) Optimierungsprobleme und die Klasse NPO

- Bei welchem Typ von Problemen kann man überhaupt von einem Approximationsalgorithmus sprechen?

- Bei Problemen, wo eine Lösung als Ausgabe verlangt wird, die irgendeine bestimmte Zielfunktion minimiert oder maximiert, also mit einem Wort: optimiert. Ein solches Problem ist ein Optimierungsproblem (im Gegensatz zu einem Entscheidungsproblem, bei dem die Ausgabe entweder JA oder NEIN ist.)

Beispiele für Optimierungsprobleme sind

min-SCHEDULING siehe oben

max-CLIQUE

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

Aufgabe: Finde eine Clique als Teilgraphen in G mit maximaler Knotenzahl.

(eine Clique ist ein vollständiger (Teil-)Graph mit einer Kante zwischen je zwei Knoten

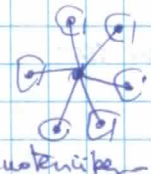
oder

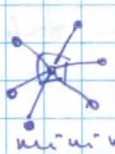
)

min-VERTEX COVER

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

Ausgabe: Eine Teilmenge $C \subseteq V$ der Knoten so dass jede Kante in E mindestens einen Endpunkt in C hat, also eine sog. Knotenüberdeckung minimaler Größe ($|C|$)





E 6.

Alle Optimierungsprobleme haben natürlicherweise eine entsprechende Entscheidungsversion (o. Sprachenversion), wobei in dem Eingabe ein Zielwert gesetzt wird:

CLIQUE:

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

Ausgabe: entscheide (JA/NEIN) ob der Graph G eine Clique der Größe q als Teilgraphen besitzt.

SCHEDULING:

Eingabe: m, $P_1 P_2 \ldots P_n$, K

Ausgabe: entscheide ob ein Schedule mit Makespan $\leq K$ für diese n Jobs existiert.

Weitere Entscheidungsprobleme sind z.B.

PARTITION

Eingabe: Zahlen $x_1, x_2, \ldots, x_n$ $(x_i \in \mathbb{N})$

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

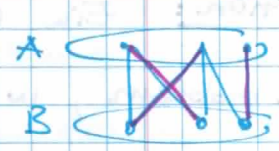
$$\sum_{i \in S} x_i = \frac{\sum_{i=1}^{n} x_i}{2}$$

die Summe dieser Zahlen die Hälfte der Summe aller n Zahlen ist

BIPARTITE MATCHING

Eingabe: ein bipartiter Graph $G(A, B, E)$ mit $|A| = |B|$

Ausgabe: entscheide ob ein perfektes Matching existiert (d.h. ohne gemeinsame Endpunkte → also unabhängige Menge von Kanten, die zusammen alle Knoten überdecken)



Notation Die folgende allgemeine Notation wird im Skript (selten) benutzt (d.h. allgemein für jedes Problem verwendbar):

In jedem Optimierungsproblem gibt es

- eine (Eingabe-)Instanz x_0 (auch I) (Graph, Jobgrößen)
- Lösungen X (die Clique, der Schedule) (irgendeine(r))

$L(x_0, X) = YES$ falls x eine Lösung für x_0 ist

- die Zielfunktion $f(x_0, x)$ (Größe der Clique, Makespan des Schedule)

die entweder zu minimieren oder zu maximieren ist

Ein unspezifiziertes allgemeines Optimierungsproblem wird dann (im Skript) durch $P = (opt, f, L)$ bezeichnet, wobei

$opt \in \{min, max\}$

(Es ist bekannt, dass Ein- und Ausgabe für Rechner als 0-1 sequenzen kodierbar sind, und dass wir von konstanten Faktoren absehen können Eingabegrößen $|x_0|$,

E 8. wie z.B. $\Theta(|E|+|V|)$ für Graphen, ~~[illegible]~~ $|x_0|$ $|x_0| = \Theta(m+n)$ bei SCHEDULING)

Wir wollen uns auf Optimierungsprobleme beschränken, die einigermaßen handhabbar sind. Grob gesagt, es muss alles in Polynomialzeit (in der Eingabegröße) berechenbar sein, was nötig ist um zu entscheiden ob eine Lösung richtig ist, (eine gegebene Lösung x) bzw. ihren Zielwert zu berechnen.

Definition: Ein Optimierungsproblem ist ein NP-Optimierungsproblem, wenn es nur polynomiell lange Lösungen hat in $|x_0|$, weiterhin es ist in $\text{Poly}(|x_0|)$ Laufzeit berechenbar

- ob x_0 eine Instanz des Problems ist
- ob x eine gegebene Lösung für x_0 ist, also $L(x_0, x) = \text{YES/NO}$
- $f(x_0, x)$ die Zielfunktion für x_0 und x

Beachte, dass x nicht in $\text{Poly}(|x_0|)$ Laufzeit berechnet werden soll!!!

NPO bezeichnet die Klasse aller NP-Optimierungsprobleme

Diese heißen NP-Optimierungsprobleme, weil die Entscheidungsversion eines solchen Problems in der Problemklasse NP liegt!

(Wir erinnern uns daran, dass in NP gehören die Probleme mit "kurzen Lösungen" bzw. mit "kurzen Beweisen/Zeugen" wobei "kurz" = "polynomiell in Eingabegröße". Diese Lösungen bzw. Beweise sind nicht immer effizient zu finden berechnen (wenn doch, dann liegt das Problem sogar in $P \subset NP$). Wenn für ein Problem (in NP) eine Lösung zu finden mindestens so schwierig ist, wie für alle andere Probleme in NP, dann ist das Problem NP-schwer (NP-vollständig), und ist sehr wahrscheinlich nicht polynomiell lösbar.)

Bemerkungen zur Klasse NP (von Entscheidungsproblemen)

- Bei Entscheidungsproblemen (zB HAMILTONSCHER KREIS) gibt es JA-Instanzen und NEIN-Instanzen; die Aufgabe ist es zu entscheiden, ob eine gegebene Instanz JA- oder NEIN-Instanz ist
  (Anders gesagt: die Sprache HAMILTONKREIS enthält alle Eingaben – eigentlich Worte aus $\{0,1\}^*$ – die einen Graphen beschreiben der einen Hamiltonkreis besitzt.) → die Sprache besteht aus den JA-Instanzen

- Entscheidungsprobleme haben somit keine Lösungen, aber für Entscheidungsprobleme in NP gibt es für jede JA-Instanz einen kurzen Beweis (oder Zeuge (witness) genannt); zB. einen Hamiltonschen Kreis im Eingabe-Graphen – und zwar so dass anhand eines Zeugen, sobald ihn uns ein Zauberer verraten hat, in Polynomialzeit verifizierbar ist, dass die Instanz eine JA-Instanz ist.
  (So ein Zeuge stimmt oft mit einer guten Lösung eines entsprechenden Optimierungsproblems überein.) bzw. die Lösung des Opt-Problems kann als Zeuge genommen werden

- Warum heißt die Klasse solcher Probleme NP?
  Sog. Nichtdeterministische Turingmaschinen (ein mathematisches Konzept) haben die 'Fähigkeit' einen Zeugen in polynomieller Laufzeit zu finden. Diese theoretischen Maschinen haben die Möglichkeit aus jedem Zustand für jedes gelesene Eingabe auf (insg. exponentiell vielen) mehreren verschiedenen Weisen weiterzurechnen. (die möglichen Berechnungswege verzweigen sich). Ein einziger Berechnungsweg einer solchen Maschine, der effizient einen Zeugen/Lösung findet und verifiziert, reicht, damit die Eingabe als JA-Instanz akzeptiert wird.
  Ein Berechnungsweg könnte einer Permutation der Knoten entsprechen die entweder ein HAM-KREIS ist oder nicht.

Wenn es eine Nichtdeterministische Turingmaschine in Polyzeit gibt, die genau die JA-Instanzen akzeptiert, dann ist das Problem, z.B. HAMILTONKREIS von einer (N)ichtdeterministischen Turingmaschine in (P)olynomieller Laufzeit "entscheidbar" und deshalb in der Klasse NP. (d.h. die Sprache HAMILTONKREIS erkennbar)

C.) Approximationsalgorithmen zu Optimierungsproblemen.

Warum brauchen wir Approximationsalgorithmen?

Wenn das Problem schwierig ist, d.h. seine Entscheidungsvariante NP-schwer (NP-vollständig) ist, dann haben wir wahrscheinlich keinen effizienten Algorithmus um eine optimale Lösung zu finden. ~~[illegible]~~ die Berechnung einer optimalen Lösung würde somit evtl länger als unser Leben dauern. Wie das Skript formuliert. wir können nicht verlangen, dass ein Algorithmus für ein NP-schweres Problem

① optimale Lösungen bestimmt

② in polynomieller Zeit läuft

③ dies für jede Instanz tut

(Wir müssen also mindestens eines von den drei Kriterien weglassen

NUR ② + ③ → wir sind mit approximativen Lösungen zufrieden und benutzen einen effizienten Approximationsalgorithmus

NUR ① + ③ → wir lassen nicht-effiziente Algorithmen zu (zB. exponentielle mit kleiner Basis

zB. der schnellste exakte Algorithmus für max-CLIQUE hat Laufzeit $O(1.2905^n)$)

NUR ① + ② → Vielleicht entspricht unsere Instanz einem Spezialfall der polynomiell lösbar ist?

(min-VERTEX-COVER ist in Zeit $O(2^k \cdot n)$ lösbar wenn es eine Knotenüberdeckung der Größe $\leq k$ gibt. Ähnliche Fragen werden im Gebiet von Problemen mit fixierten Parametern behandelt (Parametrisierte Optimierung))

Definition: Approximationsfaktor

Für beliebige Eingabe I eines Maxi-Minimierungsproblems sei $OPT(I)$ der max-minimale Zielwert unter allen Lösungen von I. Sei $\alpha \geq 1$ eine reelle Zahl.

→ Eine α-approximative Lösung für I ist eine Lösung mit Zielwert höchstens $\alpha \cdot OPT(I)$ / mindestens $\frac{OPT(I)}{\alpha}$

→ Sei noch $ALG(I)$ der Zielwert der Lösung für Eingabe I ausgegeben von einem Algorithmus ALG für das Problem.
ALG ist α-approximativ, oder hat Approximationsfaktor (höchstens) α, wenn für jede Eingabe I

$$ALG(I) \leq \alpha \cdot OPT(I) \quad / \; ALG(I) \geq \frac{OPT(I)}{\alpha}$$

Obere Schranke

→ Wenn es mindestens eine Eingabe I gibt so dass

$$ALG(I) \geq \alpha \cdot OPT(I) \quad / \; ALG(I) \leq \frac{OPT(I)}{\alpha}$$

Untere Schranke

dann hat ALG Approximationsfaktor mindestens α.

(dasselbe gilt, wenn es eine Folge von Eingaben (I_k) gibt so dass $\frac{ALG(I_k)}{OPT(I_k)} \to \alpha$ für $k \to \infty$)

[Der Approximationsfaktor ist also eigentlich $\sup_I \frac{ALG(I)}{OPT(I)}$ / $\sup_I \frac{OPT(I)}{ALG(I)}$]

Achtung! Für Maximierungsprobleme möchten wir auch Approximationsfaktoren $\alpha \geq 1$ erhalten deshalb werden sie wie oben im Grün definiert.
Viele Lehrbücher nehmen aber $\alpha \leq 1$ Werte für Maximierungsprobleme und behalten den Bruch $\frac{ALG(I)}{OPT(I)}$.

APPROXIMIERBARKEIT UND DIE APX-Klassen

— Was ist der bestmögliche Approximationsfaktor den es geben kann?

→ da $\alpha \geq 1$, $\alpha = 1$ wäre bestmöglich, und würde exakte Optimierung (einen optimalen Algorithmus) bedeuten.

— Was ist der bestmögliche Approximationsfaktor den ein effizienter (= mit polynomieller Laufzeit) Algorithmus für ein NP-schweres Optimierungsproblem haben kann? (angenommen $P \neq NP$)

Eigentlich können wir hoffen, dass wir beliebig gut effizient approximieren können. Wenn für ein Optimierungsproblem dies der Fall ist, dann hat man für jede $\alpha > 1$, anders gesagt, für jede $1+\varepsilon$ ($\varepsilon > 0$) einen effizienten $(1+\varepsilon)$-approximativen Algorithmus A_ε. Da wir in diesem Fall nicht einen, sondern unendlich viele Algorithmen haben (die Definition des Algorithmus hängt von ε ab! für jede ε einen Algorithmus:) nennen wir all diese Algorithmen A_ε für alle $\varepsilon > 0$ ein Polynomielles Approximationsschema (PTAS).

eigentlich ein Rezept wie man einen $(1+\varepsilon)$-approximativen Algorithmus für beliebige ε erstellt

a.) Approximationsschemata

Definition: Ein polynomielles Approximationsschema (PTAS – polynomial time approximation scheme) für ein Optimierungsproblem P ist eine Familie $(A_\varepsilon)_{\varepsilon>0}$ von Approximationsalgorithmen für P, so dass für jede $\varepsilon > 0$

E 12.

der zugehörige Algorithmus A_ε $(1+\varepsilon)$-approximativ ist. Die Laufzeit von jedem A_ε muss polynomiell in der Eingabelänge (aber nicht in $\frac{1}{\varepsilon}$) sein.

Ein PTAS für min-SCHEDULING bei fixierter Maschinenzahl.

min-SCHEDULING-m

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

Aufgabe: weise jedem Job eine von m Maschinen zu, so dass der Makespan minimiert wird

(Was ist der Unterschied zu min-SCHEDULING?
m ist hier festgelegt, und nicht Teil der Eingabe
Warum ist das überhaupt wichtig?
Die Eingabelänge ist n und [redacted] m ist hier "eine konkrete Zahl"
und die Laufzeit darf kann z.B. polynomiell in n aber exponentiell in m werden.)

PTAS für min-SCHEDULING-m.

Sei ε gegeben, wir definieren den $(1+\varepsilon)$-approximativen Algorithmus A_ε mit Laufzeit Poly(n).

Grobstruktur des Algorithmus A_ε

Eingabe I: n Job-Laufzeiten

1. sortiere die Jobs nach absteigender Laufzeit
   Seien $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n$
   die sortierten Laufzeiten (mit neuen Indizes)
2. bestimme einen optimalen Schedule der Jobs $p_1 p_2 \ldots p_k$ für eine geeignete k
   (wir teilen die "großen" Jobs optimal den m Maschinen zu, indem wir alle Möglichkeiten ausprobieren) (das braucht Zeit exponentiell in k, aber nicht in n; k wird nur von $\frac{1}{\varepsilon}$ und m abhängen, und wir bestimmen ihn später)
3. ergänze diesen Schedule mit LIST für die übrigen $n-k$ Jobs
   (das sind also die "kleinen" Jobs)

Worauf sollen wir bei der Bestimmung von k achten?

Was riskieren wir wenn k zu groß gewählt wird?

Was wenn k zu klein ist?

(Im ersten Fall haben wir zu viele Jobs als "groß" betrachtet, im zweiten Fall zu wenig Jobs als "groß" betrachtet. Warum ist das Eine und warum das Andere schlecht?)

E 14.

Der erste führt zu viel zu hoher Laufzeit; der zweite zu viel zu grober Approximation.

Im Folgenden bestimmen wir einen geeigneten k, und gleichzeitig analysieren den Approximationsfaktor und die Laufzeit: zur Illustration wählen wir o.B.d.A $m = 3$ Maschinen.

Analyse für $m = 3$ Maschinen

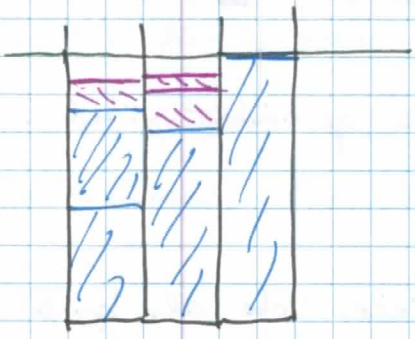
Der Algorithmus geht so vor:

OPT für große Jobs

$p_1 \geq p_2 \geq \dots \geq p_k \geq p_{k+1} \geq \dots \geq p_n$

große Jobs — kleine Jobs

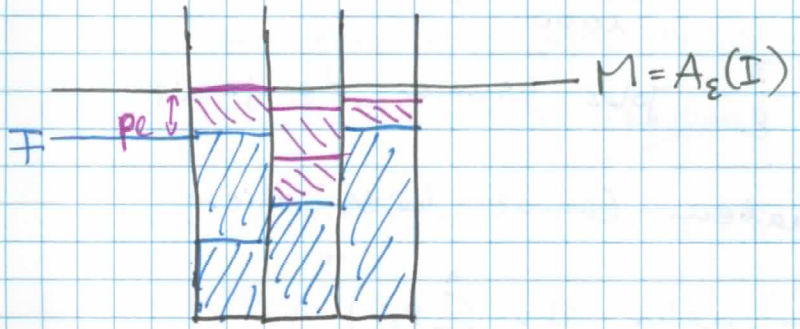
Fall 1. Der Makespan wird von einem großen Job bestimmt



da $Opt(\{p_1, \dots, p_k\}) \leq Opt(\{p_1, \dots, p_n\})$

und dieser Schedule hat Makespan $Opt(\{p_1, \dots, p_k\})$, wir haben einen optimalen Schedule im Fall 1.

Fall 2. der Makespan wird von einem kleinen Job bestimmt



Sei p_e die Laufzeit dieses kleinen Jobs, dann $l \geq k+1$ und $p_e \leq p_{k+1}$ (weil bis p_k die Jobs groß sind)

F kann die Fertigstellungszeit eines kleinen oder eines großen Jobs sein!

Sei $F = M - p_e$. Die Maschinen sind bis Zeit F alle voll, sonst hätte LIST eine andere Maschine für p_e gewählt. Deshalb gilt (wie bei LIST), dass $Opt(I) \geq F$

Für den Makespan von A_ε gilt

$$M = A_\varepsilon(I) = F + p_e \leq Opt(I) + p_e$$

Für die Approximationsgüte $1+\varepsilon$ brauchen wir also, dass $p_e < \varepsilon \cdot Opt(I)$ für jeden kleinen Job gilt.

$\Rightarrow$ k muss so gewählt werden, dass $p_{k+1} < \varepsilon \cdot Opt(I)$

— Für $Opt(I)$ haben wir allgemein nur eine Abschätzung (von der Gesamtlaufzeit aller Jobs)

$$Opt(I) \geq \frac{\sum_{j=1}^{n} p_j}{3} \qquad \left(\frac{\sum_{j=1}^{n} p_j}{m} \text{ allgemein}\right)$$

wir sind also richtig, wenn wir sicherstellen, dass

$$p_{k+1} < \frac{\varepsilon \cdot \sum_{j=1}^{n} p_j}{3} \quad \left(\leq \varepsilon \cdot Opt(I)\right)$$

Dies gilt mit Sicherheit, wenn $k \geq \frac{3}{\varepsilon}$ $\left(\frac{m}{\varepsilon} \text{ allgemein}\right)$

E 16.

(Warum? Nur maximal 1000 Jobs können Laufzeit mindestens $\frac{\sum p_j}{1000}$ haben.

Ebenso: höchstens $\frac{3}{\varepsilon}$ Jobs können Laufzeit mindestens $\frac{\sum p_j}{\frac{3}{\varepsilon}}$ haben (sonst hätten diese Jobs Gesamtlaufzeit mehr als $\sum_{j=1}^{n} p_j$))

Fazit: Wenn wir $k = \frac{3}{\varepsilon}$ (bzw. $\frac{m}{\varepsilon}$) setzen, dann hat A_ε Approximationsfaktor $\leq 1+\varepsilon$

Laufzeit:

Erreichen wir mit der Wahl $k = \frac{3}{\varepsilon}$ Laufzeit polynomiell in n?

Für die größten k Jobs müssen wir alle Zuteilungen zu den 3 Maschinen ausprobieren um den optimalen Schedule zu finden. (wir sehen jetzt von geringen Verbesserungen ab)
wie gleichgroße Jobs, Permutation der Maschinen usw.

Wieviele Möglichkeiten gibt es, die k Jobs zu den drei Maschinen zuzuteilen?

(Wenn die Maschinen nicht "gleich sind", zB. gibt es Maschine 1, Maschine 2 und Maschine 3, dann gibt es so viele Möglichkeiten wie (verschiedene) Vektoren mit k Einträgen aus $\{1,2,3\}$ — Job j wird Maschine $v(j)$ gegeben für einen solchen Vektor v

1 1 2 1 1 3 3 2 2 2 2 1 3 1 2 2 1

k lang

solche Verteilungen gibt es 3^k (allg. m^k)

[Wenn die Maschinen gleich sind, und nur die Aufteilung der Jobs in 3 Partitionen zählt, muss man 3^k durch 3! teilen $\frac{3^k}{6}$, weil so oft wird die Zuteilung zu Maschinen 1, 2, 3 dieselbe Job-Partition ergeben.] → stimmt nicht! Warum?

Die Laufzeit ist somit

$$O(n \cdot \log n) + O(3^k) + \overbrace{O(n)}^{O(n \cdot \log m)} =$$

$O(n \cdot \log n)$: sortieren; $O(3^k)$: Opt für große Jobs; $O(n)$: LIST für kleine Jobs

$$= O\left(n \cdot \log n + 3^{\frac{3}{\varepsilon}}\right)$$ → „Konstante"

(für m Maschinen $O(n \cdot \log n + m^{\frac{m}{\varepsilon}})$)

→ polynomialzeit in n □

Bemerkungen

1. mit besserer Analyse – wie bei LIST – findet man, dass $k = \frac{m-1}{\varepsilon}$ reicht; dies ergibt Laufzeit $O\left(m^{\frac{m-1}{\varepsilon}} + n \cdot \log n\right)$

2. Würde dieses PTAS für uns das Problem in der Praxis lösen? Würde man ihn verwenden wollen?
   – die Laufzeit für 2 Maschinen und $\varepsilon = 0.01$
   $m^{\frac{m-1}{\varepsilon}} = 2^{100}$ → nicht durchführbar!
   ebenso nicht für 10 Maschinen und $\varepsilon = 0.1$

Wozu sind solche PTAS mit oft enormer Laufzeit gut?

→ Orientierung für die Theorie: es hat Sinn nach praktischeren Algorithmen mit relativ guter Approximation weiterzusuchen. (das Problem ist effizient beliebig approximierbar)

3. Die Laufzeit dieses PTAS ist exponentiell in

$\frac{1}{\varepsilon}$ — das ist erlaubt, für den Algorithmus A_ε ist ε eine Konstante.

es ist auch exponentiell in

m — auch erlaubt, da m für das min-SCHEDULING-m Problem eine Konstante, kein Teil der Eingabe ist.

ABER: diese Algorithmen A_ε wären kein PTAS für min-SCHEDULING, weil nicht polynomiell in m.

b.) Volle Approximationsschemata

Was kann ein besserer Approximationsalgorithmus sein als ein PTAS? (für ein NP-schweres Optimierungsproblem)

- bezüglich der Approximation: ist ein PTAS bestmöglich
- bezüglich der Laufzeit: könnte etwas besser werden

wir könnten von einem PTAS träumen, dass auch in $\frac{1}{\varepsilon}$ polynomielle Laufzeit hat!

Für manche Probleme gibt es solche PTAS und sie (die PTAS) haben einen speziellen Namen:

Definition: Ein **volles** polynomielles Approximationsschema

(FPTAS – fully polynomial time approximation scheme)

ist ein PTAS $(A_\varepsilon)_{\varepsilon>0}$ so dass die Laufzeit jedes Algorithmus A_ε polynomiell ist in der Eingabelänge **und** in $\frac{1}{\varepsilon}$.

PTAS ist die Klasse aller NP-Optimierungsprobleme die ein PTAS besitzen.

FPTAS ist die Klasse aller NP-Optimierungsprobleme die ein FPTAS besitzen.

Beispiel: (wir haben für min-SCHEDULING-m ein PTAS gesehen, das kein FPTAS war)

Poly(n) ?

min-SCHEDULING-m besitzt sogar ein FPTAS

min-SCHEDULING besitzt ein PTAS aber kein FPTAS

– Was gilt also für die Laufzeit des FPTAS für min-SCHEDULING-m?

→ es ist poly $(n, \frac{1}{\varepsilon})$

Es kann nicht polynomiell sein in m, warum?

– Was gilt für die Laufzeit des PTAS für min-SCHEDULING?

→ ist poly (m, n) aber nicht in $\frac{1}{\varepsilon}$.

(sonst gäbe es FPTAS für min-SCHEDULING)

E 20.

C.) Nicht-Approximierbarkeits Resultate

Kein FPTAS:

Oft ist die Suche nach einem FPTAS von vornherein hoffnungslos! Ein triviales Beispiel dafür ist das CLIQUE Problem:

Beispiel: max-CLIQUE

CLIQUE ist ein NP-vollständiges Problem.

Angenommen, es gäbe ein FPTAS für max-CLIQUE. Dieses hätte worst-case Laufzeit $\text{Poly}(n, \frac{1}{\varepsilon})$ für Graphen mit $|V| = n$ Knoten.

Wie groß kann eine (maximale) Clique sein?

Also die Zielfunktion $A_\varepsilon(I)$ (Clique-Größe) kann ganzzahlige und nicht zu große Werte annehmen.

$\Longrightarrow$ Es ist möglich einen (nicht zu engen) ε für den Approximationsfaktor $(1+\varepsilon)$ vom (hypothetischen) FPTAS zu verlangen, dass der $(1+\varepsilon)$-approximative Algorithmus A_ε eine optimale Lösung (größtmögliche Clique) ausgeben muss, (weil es keine andere ganze Zahl zwischen $\frac{OPT(I)}{(1+\varepsilon)}$ und $OPT(I)$ gibt)!

(Ungefähr) Welches ε wäre in diesem Fall nah genug?

Gefühl: $\varepsilon = \frac{1}{n}$ würde reichen um optimales Ergebnis vom (hypothetischen) FPTAS zu bekommen in Laufzeit:

$$\text{Poly}(n, \tfrac{1}{\varepsilon}) = \text{Poly}(n, \tfrac{1}{\frac{1}{n}}) = \text{Poly}(n)$$

$A_\varepsilon = A_{\frac{1}{n}}$ vom FPTAS wäre somit ein optimaler Algorithmus der Poly(n) Laufzeit hat – WIDERSPRUCH! der FPTAS kann nicht existieren!

{ Wir prüfen dass $\varepsilon = \frac{1}{n}$ tatsächlich reicht:

$$\left(1+\frac{1}{n}\right)\left(1-\frac{1}{n}\right) < 1 \quad \text{(Warum?)}$$

Wenn

$$A_\varepsilon(I) \geq \frac{OPT(I)}{\left(1+\frac{1}{n}\right)} > \left(1-\frac{1}{n}\right)\cdot OPT(I) = OPT(I) - \frac{OPT(I)}{n} \geq$$

$$\geq OPT(I) - 1 \quad \text{weil } OPT(I) \leq n$$

da $A_\varepsilon(I)$ (Clique-Größe vom Algorithmus A_ε) ganzzahlig ist

$$A_\varepsilon(I) > OPT(I) - 1$$

impliziert

$$A_\varepsilon(I) = OPT(I)$$]

Beobachtung: Dasselbe Argument für die Nicht-Existenz eines FPTAS hätte funktioniert wenn die Zielfunktion $f()$ höchstens $c \cdot n^k$ (also polynomiell in n) ist. Diese Beobachtung formulieren wir allgemeiner, und solche Probleme nennen wir polynomiell beschränkt.

Theorem: Sei $P = (opt, f, L)$ ein NP-vollständiges Optimierungsproblem. Falls:

- $f(x_0, x)$ die Zielfunktion nur ganzzahlige Werte annimt, und
- es gibt ein Polynom $q()$ so dass $f(x_0, x) < q(|x_0|)$ für jede Instanz x_0 und Lösung x.

Dann besitzt P kein FPTAS

(Bei der Wahl $\varepsilon = \frac{1}{q(|x_0|)}$ hätte man effizienten und optimalen Algorithmus)

E 22.

(Überlegen wir noch: muss $f(x, x_0)$ nicht immer polynomiell in der Eingabegröße sein? NEIN!

Beispiel: min-SCHEDULING

der Einfachheit halber seien $p_1 p_2 .. p_j .. p_n$ ganze Zahlen.

Streng genommen ist die Länge der Instanz

$$O(m + n \cdot \log p_{max})$$

wobei p_{max} die maximale Laufzeit ist $p_{max} = \max_j p_j$

(die Maschinen werden als einzelne Eingabeelemente betrachtet, aber $m < n$ gilt meistens)

Da $\log p_{max}$ meistens relativ klein ist (im Vergleich mit n, m) wird normalerweise $\max(n, m)$ die Eingabelänge bestimmen.

ABER: $f(x, x_0)$ der Makespan hat Größenordnung

$$p_{max} \ldots n \cdot p_{max}$$

das ist exponentiell in $\log p_{max}$

(weil $2^{\log p_{max}} = p_{max}$) und (auch) nicht unbedingt polynomiell in $n + m$.

Ähnliche Überlegungen gelten für viele andere Probleme mit "Werten" in der Eingabe, wie Rucksack, [redacted] usw.)

kein PTAS: (Wie würde man mathematisch ausdrücken, dass ein bestimmtes Problem kein PTAS besitzt?)

Beispiel: min-VERTEX COVER

(für einen Eingabegraphen $G(V,E)$ wird eine Knotenüberdeckung – Knotenmenge die alle Kanten überdeckt – minimaler Größe gesucht)

Ein (2-approximativer) greedy Algorithmus für min-VERTEX COVER

Eingabe $G(V,E)$ $(E \neq \emptyset)$

- setze $C = \emptyset$

REPEAT

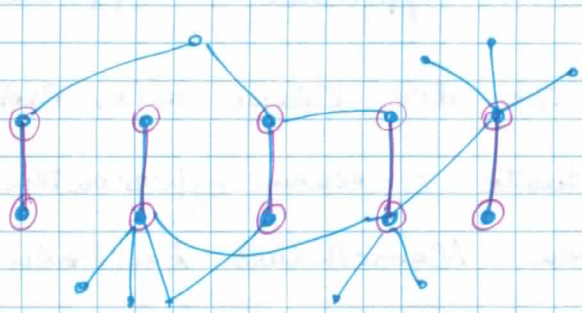
- nehme eine beliebige Kante $\{u,v\} \in E$
- nehme u und v in die Überdeckung C auf
- entferne alle zu u oder zu v inzidente Kanten aus E

UNTIL $E = \emptyset$

- return C

Warum ist dieser Algorithmus 2-approximativ?

In C haben wir alle Endknoten einer (maximalen nicht erweiterbaren) unabhängigen Kantenmenge



(alle andere Kanten haben mind. einen Endknoten in C)

E 24.

Sei $|C| = 2k$, d.h. der Alg. k Knoten (k Kanten) gehabt.

um diese k Kanten zu überdecken braucht eine beliebige Überdeckung (auch eine optimale) mindestens k Knoten, also $|C_{OPT}| \geq k$

Somit ist unser Algorithmus 2-approximativ

(Der Approximationsfaktor ist eigentlich genau 2; um zu zeigen, dass er ≥ 2 ist benutze einen vollständigen bipartiten Graphen) — oder einfach K_2

Fakten (ohne Beweis):

1. VERTEX-COVER ist 2-approximierbar (siehe oben) effizient
2. VERTEX-COVER ist nicht α-approximierbar (effizient) für $\alpha < 10\sqrt{5} - 21 \approx 1.36$ angenommen $P \neq NP$ (schwer zu Beweisen)
3. somit existiert auch kein PTAS für VERTEX COVER
4. für $10\sqrt{5} - 21 \leq \alpha < 2$ ist es nicht bekannt ob VC. effizient α-approximierbar ist.

Definition: APX ist die Klasse aller Probleme, die für irgend eine Konstante c einen effizienten c-approximativen Algorithmus besitzen.

keine konstante Approximation

Beispiel: max-CLIQUE

Fakten (ohne Beweis): Angenommen $P \neq NP$

Es gibt keine Konstante α, so dass max-CLIQUE in Polynomialzeit α-approximierbar ist.

Es gilt sogar das folgende viel stärkere Resultat:

Theorem: Für Beliebige $\delta > 0$ gibt es keinen effizienten $n^{1-\delta}$-approximativen Algorithmus für max-CLIQUE. (ohne Beweis)

Beachte, dass beliebiger vernünftiger Approximationsalgorithmus (der zB mindestens einen Knoten ausgibt) ist automatisch n-approximativ. Warum?

Ein greedy Algorithmus für max-CLIQUE ($\Theta(n)$-approximativ)

Eingabe: $G(V,E)$

$C := \emptyset$

REPEAT

- sei v ein Knoten mit maximalem Knotengrad $grad(v) (= deg(v))$

  $grad(v)$ = die Anzahl der zu v inzidenten Kanten

- $C := C \cup \{v\}$

- entferne v und alle Knoten nicht adjazent mit v aus V

UNTIL $V = \emptyset$

Def: $f(n)$-APX ist die Klasse von Problemen mit einem effizienten, $O(f(n))$-approximativen Algorithmus (für Eingabelänge n).

$max\text{-CLIQUE} \in n\text{-APX} \subseteq poly\text{-APX}$

E26.

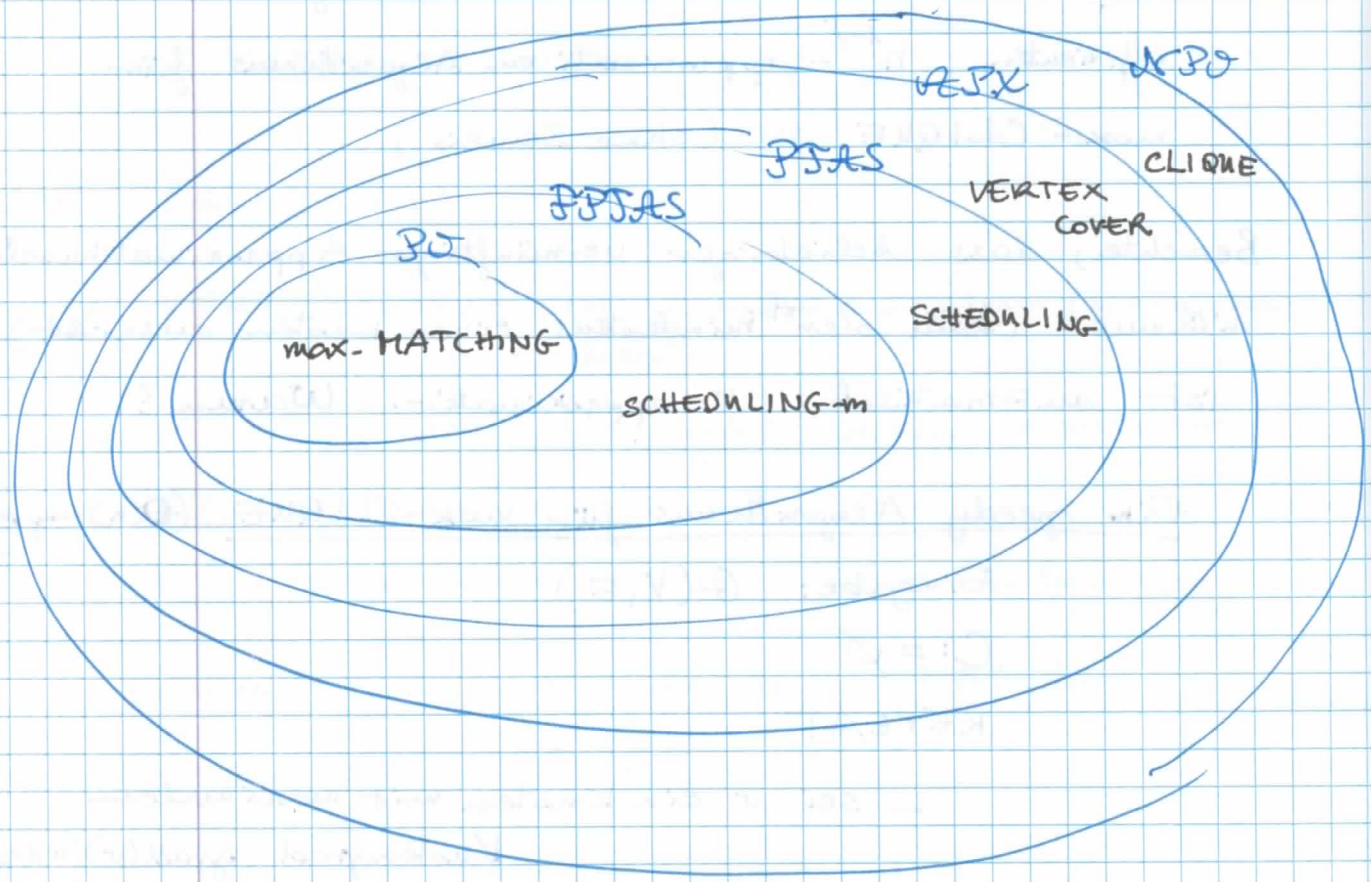
d.) Zusammenfassung

Wir haben bis jetzt die folgenden Problemklassen definiert:

$$NPO,\ PTAS,\ FPTAS,\ APX,\ g(n)\text{-}APX$$

Sei noch PO die Klasse der NP-Optimierungsprobleme, die in polynomieller (exakt) Zeit gelöst werden können.

Wie sieht die Hierarchie dieser Klassen aus?



Def: poly-APX ist die Klasse von Problemen in $q(n)$-APX für irgendein Polynom $q(n)$.

Analog werden die Klassen log-APX und poly(log)-APX definiert. Es gilt:

$$PO \subseteq FPTAS \subseteq PTAS \subseteq APX \subseteq \log\text{-}APX \subseteq \text{poly}(\log)\text{-}APX \subseteq \text{poly-}APX \subseteq NPO$$