

Vertex Cover

Kapitel 3:

Approximationsalgorithmen

Hilfreiche Literatur:

- Vazirani: Approximation Algorithms, Springer Verlag, 2001.
- Hochbaum: Approximation Algorithms for NP-Hard Problems, Thomson Publishing, 1996.
- Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela, Protasi: Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties, Springer Verlag, 1999.
- Garey, Johnson: Computers and Intractability, Freeman and Company, 1979.
- Papadimitriou: Computational Complexity, Addison Wesley, 1994.

Für viele Entscheidungsproblem gibt es ein korrespondierendes Optimierungsproblem und umgekehrt. Wir betrachten ein Beispiel.

Sei $G = (V, E)$ ein Graph. Eine Teilmenge der Knoten $U \subseteq V$ wird als Vertex-Cover (Knotenüberdeckung) bezeichnet, falls jede Kante aus E inzident zu einem Knoten aus U ist.

VERTEX-COVER (VC):

Gegeben sei ein Graph G und eine Zahl $k \in \mathbb{N}$. Es ist zu entscheiden, ob es ein Vertex-Cover der Kardinalität k gibt.

MIN-VERTEX-COVER (MIN-VC):

Gegeben sei ein Graph G . Gesucht ist ein Vertex-Cover kleinster Kardinalität.

NP-harte Optimierungsprobleme:

- Ein Entscheidungs- bzw. Optimierungsproblem Π ist polynomialzeit-reduzierbar auf ein Entscheidungs- bzw. Optimierungsproblem Π' , falls ein Polynomialzeit-Algorithmus für Π' einen Polynomialzeit-Algorithmus für Π liefert. Offensichtlich ist VC polynomialzeit-reduzierbar auf MIN-VC.
- Ein Entscheidungs- bzw. Optimierungsproblem Π wird als NP-hart bezeichnet, falls alle (Entscheidungs)probleme aus NP polynomialzeit-reduzierbar auf Π sind. Wir wissen, VC ist NP-hart und polynomialzeit-reduzierbar auf MIN-VC. Wegen der Transitivität von Polynomialzeit-Reduktionen folgt somit, dass auch MIN-VC ein NP-hartes Problem ist.
- Unter der Annahme $P \neq NP$ gibt es keinen Polynomialzeit-Algorithmus für NP-harte Optimierungsprobleme. Deshalb interessieren wir uns für Approximationsalgorithmen, die in polynomieller Zeit eine Näherungslösung mit garantierter Güte berechnen.

Approximationsalgorithmen:

Approximationsalgorithmen berechnen typischerweise nicht die optimale Lösung, sondern eine Lösung „nah am Optimum“. Die Güte der Lösung wird durch einen Approximationsfaktor beschrieben.

Sei \mathcal{A} ein Approximationsalgorithmus für ein Optimierungsproblem Π . Sei \mathcal{I} die Menge der möglichen Eingabeinstanzen für Π . Für $I \in \mathcal{I}$ bezeichne $w(I)$ den Wert, der von \mathcal{A} berechneten Lösung, und $opt(I)$ den optimalen Lösungswert. Dann definieren wir den Approximationsfaktor von \mathcal{A} auf I durch

$$r(I) = \frac{w(I)}{opt(I)} .$$

Falls Π ein Minimierungsproblem ist, so gilt offensichtlich $r(I) \geq 1$, bei einem Maximierungsproblem gilt grundsätzlich $r(I) \leq 1$.

Im Allgemeinen wird der Approximationsfaktor als Funktion $R : \mathbb{N} \rightarrow \mathbb{R}$ in einem geeigneten Parameter $n \in \mathbb{N}$ beschrieben. Beispielsweise könnte n die Anzahl der Knoten eines Eingabegraphen bezeichnen.

Für $n \in \mathbb{N}$ bezeichne \mathcal{I}_n die Menge der Eingaben mit Parameter n . Ein Algorithmus für ein Minimierungsproblem garantiert einen Approximationsfaktor $R(n)$ (z.B. $R(n) = 2$ oder $R(n) = \ln n$), falls gilt

$$\forall n \in \mathbb{N} : \forall I \in \mathcal{I}_n : r(I) \leq R(n) .$$

Ein Algorithmus für ein Maximierungsproblem garantiert einen Approximationsfaktor $R(n)$ (z.B. $R(n) = \frac{1}{2}$ oder $R(n) = \frac{1}{\ln n}$), falls gilt

$$\forall n \in \mathbb{N} : \forall I \in \mathcal{I}_n : r(I) \geq R(n) .$$

Satz 1 MIN-VC hat einen polynomiellen Approximationsalgorithmus mit konstantem Approximationsfaktor 2.

Beweis: Der folgende Algorithmus liefert den Satz.

Algorithmus Approx-VC:

Sei $G = (V, E)$ der Eingabegraph. Berechne ein inklusions-maximales Matching M . Gib $V(M)$, die Menge aller Endpunkte der Kanten in M , aus.

Laufzeit: Ein inklusions-maximales Matching kann offensichtlich durch einen trivialen Greedy-Algorithmus in Zeit $O(|E|)$, also Linearzeit, berechnet werden.

Korrektheit: Zu zeigen ist, $V(M)$ deckt alle Kanten ab.

Widerspruchsbeweis: Sei e eine nicht abgedeckte Kante. Dann ist $e \in E \setminus M$ und $M \cup \{e\}$ ein Matching. Also ist M nicht inklusions-maximal. Ein Widerspruch!

Approximationsfaktor: Sei opt die Kardinalität eines Vertex-Covers kleinster Kardinalität. Es gilt $opt \geq |M|$, weil jedes Vertex-Cover mindestens einen Endpunkt jeder Kante in M abdecken muß. Somit gilt $|V(M)| = 2|M| \leq 2opt$.

□

Set Cover

Gegeben sei eine Menge von Fähigkeiten X sowie eine Menge von Personen \mathcal{S} , von denen jede über einige der Fähigkeiten in X verfügt. Verschiedene Personen verlangen möglicherweise eine unterschiedliche Bezahlung. Wir möchten ein möglichst günstiges Arbeitsteam zusammenstellen, so dass alle Fähigkeiten abgedeckt sind. Konkret ergibt sich das folgende Problem.

SET-COVER:

Gegeben sei eine Grundmenge X mit n Elementen sowie eine Kollektion von m Mengen $\mathcal{S} = \{S_1, \dots, S_m\}$, $S_i \subseteq X$, $\bigcup_{S \in \mathcal{S}} S = X$. Menge S_i hat Kosten $cost(S_i) = c_i \in \mathbb{N}$.

Gesucht ist eine Teilkollektion $\mathcal{C} \subseteq \mathcal{S}$ mit minimalen Kosten $cost(\mathcal{C}) = \sum_{S \in \mathcal{C}} cost(S)$, so dass alle Elemente aus X abgedeckt sind, d.h. $\bigcup_{S \in \mathcal{C}} S = X$.

Auch das Set-Cover-Problem ist NP-hart, da es eine Verallgemeinerung des Vertex-Cover Problems ist. Warum?

- Ein Graph, in dem Kanten nicht nur aus zwei Knoten bestehen, sondern aus Teilmengen von beliebig vielen Knoten heißt Hypergraph. Diese Teilmengen der Knoten heißen Hyperkanten.
- Beim Vertex-Cover-Problem für einen Hypergraphen $H = (V, E)$ muss eine Knotenmenge $U \subseteq V$ minimaler Kardinalität gewählt werden, so dass jede der Hyperkanten zu mindestens einem Knoten in U inzident ist, d.h. jede Hyperkante muss mindestens einen Knoten aus U enthalten.
- Beim gewichteten Vertex-Cover-Problem haben die Knoten Gewichte und die Summe der Gewichte in U soll minimiert werden.

Das Vertex-Cover-Problem für Hypergraphen mit Knotengewichten ist auch unter dem Namen Hitting-Set-Problem bekannt. Das Set-Cover-Problem mit Grundmenge X und Mengenkollektion \mathcal{S} entspricht diesem Problem, wenn wir $X = E$ und $\mathcal{S} = V$ setzen und die Knotengewichte als Kosten interpretieren.

Die folgende Heuristik sieht vielversprechend aus.

Algorithmus Greedy-Set-Cover:

Startend mit $\mathcal{C} = \emptyset$, solange \mathcal{C} nicht alle Elemente aus X abdeckt: füge jeweils die Menge $S \in \mathcal{S}$ zu \mathcal{C} hinzu, die die niedrigsten relativen Kosten hat.

Die relativen Kosten einer Menge $S \in \mathcal{S}$ sind dabei definiert als

$$\alpha(S) = \frac{\text{cost}(S)}{\text{uncovered}(S)},$$

wobei $\text{uncovered}(S)$ die Anzahl derjenigen Elemente aus S bezeichnet, die noch nicht durch \mathcal{C} abgedeckt sind.

Wie gut ist diese Heuristik?

Sei $x_i \in X$ das i -te Element, das durch den Algorithmus abgedeckt wird, wobei wir in derselben Iteration abgedeckte Elemente beliebig anordnen.

Die Kosten, die eine zu \mathcal{C} hinzugefügte Menge S verursacht, verteilen wir gleichmäßig auf die zusätzlich durch diese Menge abgedeckten Elemente. Jedes dieser Elemente verursacht somit Kosten in Höhe von $\frac{\text{cost}(S)}{\text{uncovered}(S)} = \alpha(S)$.

Bezeichne opt die Kosten eines optimalen Set-Covers.

Lemma 2 Für $i \in \{1, \dots, n\}$ gilt $\text{cost}(x_i) \leq \frac{opt}{n-i+1}$.

Beweis:

Sei S diejenige Menge durch deren Hinzunahme der Algorithmus das Element x_i erstmalig abdeckt. Betrachte den Zeitpunkt vor der Hinzunahme von S zu \mathcal{C} .

Wir behaupten es gilt $\alpha(S) \leq \frac{opt}{\text{uncovered}(X)}$, denn

- sonst hätten alle Mengen aus $\mathcal{S} \setminus \mathcal{C}$ relative Kosten größer als $\frac{opt}{\text{uncovered}(X)}$, so dass
- noch nicht einmal das Teilproblem, das nur aus den nicht durch \mathcal{C} abgedeckten Elementen besteht, zu Kosten opt gelöst werden könnte.

Die Menge \mathcal{C} deckt höchstens die Elemente x_i, \dots, x_{i-1} ab.

Damit ist $\text{uncovered}(X) \geq n - i - 1$ und somit folgt

$$\alpha(S) \leq \frac{opt}{n-i-1}. \quad \square$$

Satz 3 Algorithmus Greedy-Set-Cover hat einen Approximationsfaktor von höchstens $H(n) = \sum_{i=1}^n \frac{1}{i} \leq \ln n + 1$.

Beweis: Die Summe der Kosten über alle Elemente ergibt die Gesamtkosten des Algorithmus. Diese Kosten sind höchstens

$$\sum_{i=1}^n \frac{opt}{n - i + 1} = \sum_{i=1}^n \frac{opt}{i} = opt H(n) .$$

□

Es gibt eine einfache Set-Cover-Instanz, für die die Greedy-Heuristik den Faktor $(1 - \epsilon)H(n)$ für beliebig kleines $\epsilon > 0$ erreicht (siehe Tafelbild). Dieses Ergebnis gilt sogar, wenn alle Mengen dieselben Kosten haben.

Feige hat 1995 gezeigt, dass es keinen Polynomialzeit-Algorithmus mit Approximationsfaktor $(1 - \epsilon)H(n)$ gibt, es sei denn $NP = TIME(n^{O(\log \log n)})$. Auch dieses Ergebnis gilt, wenn alle Mengen dieselben Kosten haben.

Unter den üblichen komplexitätstheoretischen Annahmen bedeutet das, dass der einfache Greedy-Algorithmus den bestmöglichen Approximationsfaktor für Set-Cover liefert. Ein wirklich erstaunliches Ergebnis.

TSP

Wir haben jetzt Probleme mit konstantem bzw. logarithmischem Approximationsfaktor gesehen. Es stellt sich die Frage, ob jedes Problem derartig approximiert werden kann? – Wir werden sehen, das folgende Problem kann nicht vernünftig approximiert werden.

TRAVELING-SALESPERSON (TSP):

Gegeben sei ein vollständiger Graph $G = (V, E)$ mit Kantenlängen aus \mathbb{N} . Gesucht ist ein Hamilton-Kreis (auch TSP-Tour genannt) minimaler Länge.

Ein Hamilton-Kreis in einem Graphen G ist ein Kreis in G , der jeden Knoten genau einmal besucht. Der Beweis der Nicht-Approximierbarkeit von TSP basiert auf der bekannten NP-Vollständigkeit des Hamilton-Kreis-Problems.

HAMILTONIAN CYCLE:

Gegeben sei ein Graph $G = (V, E)$. Es ist zu entscheiden, ob G einen Hamilton-Kreis enthält.

Auf den Beweis der NP-Vollständigkeit dieses Problems verzichten wir. Im Folgenden bezeichne n die Anzahl der Knoten im betrachteten Graphen.

Satz 4 Sei $\alpha(n)$ eine beliebige polynomialzeit-berechenbare Funktion (z.B. $\alpha(n) = 2^n$). Unter der Annahme $P \neq NP$ gilt, TSP hat keinen Polynomialzeitalgorithmus mit Approximationsfaktor $\alpha(n)$.

Beweis: Zum Zwecke des Widerspruchs nehmen wir an, es gibt einen Polynomialzeitalgorithmus \mathcal{A} mit Approximationsfaktor $\alpha(n)$. Der folgende Algorithmus \mathcal{A}' verwendet \mathcal{A} als Unterprogramm und kann dadurch für einen beliebigen Graphen G' mit $n \geq 2$ Knoten in Polynomialzeit entscheiden, ob G' einen Hamilton-Kreis hat. Daraus folgt $P=NP$, also ein Widerspruch zu unserer Annahme!

- Algorithmus \mathcal{A}' transformiert G' in einen gewichteten, vollständigen Graphen $G = (V, E)$, in dem nur die Kanten aus E' die Länge 1 haben, alle anderen Kanten in E erhalten die Länge $n \cdot \alpha(n)$.
- Dann ruft \mathcal{A}' den Algorithmus \mathcal{A} auf G auf.
- Falls \mathcal{A} eine TSP-Tour der Länge höchstens $n \cdot \alpha(n)$ findet, so gibt \mathcal{A}' die Tour als Hamilton-Kreis von G aus.
- Ansonsten meldet \mathcal{A}' , dass G' keinen Hamilton-Kreis enthält.

Korrektheit von \mathcal{A}' :

- Zuerst nehmen wir an, G' hat mindestens einen Hamilton-Kreis. Jeder Hamilton Kreis in G' entspricht einer TSP-Tour der Länge n in G . Alle anderen TSP-Touren in G enthalten mindestens eine Kante der Länge $n \cdot \alpha(n)$ und haben somit eine Länge von mindestens $n \cdot \alpha(n) + n - 1 > n \cdot \alpha(n)$. Diese Touren sind somit keine $\alpha(n)$ -Approximationen der kürzesten Rundreise. Also muss \mathcal{A} eine derjenigen TSP-Touren finden, die einem Hamilton-Kreis entsprechen.
- Jetzt nehmen wir an, G' enthält keinen Hamilton-Kreis. Dann haben alle TSP-Touren in G die Länge mindestens $n \cdot \alpha(n) + n - 1 > n \cdot \alpha(n)$ und \mathcal{A}' meldet, dass G' keinen Hamilton-Kreis enthält.

Laufzeit von \mathcal{A}' :

Unter der Annahme, dass \mathcal{A} Polynomialzeit hat, hat auch \mathcal{A}' Polynomialzeit. Dies ist jedoch nicht möglich unter der Annahme $P \neq NP$. Also gibt es keinen Polynomialzeitalgorithmus \mathcal{A} mit Approximationsfaktor $\alpha(n)$ für TSP. \square

Der obige Beweis basierte darauf, dass die TSP-Tour jeden Knoten nur einmal besuchen darf. Das ist keine besonders praktische Annahme, insbesondere wenn die direkte Verbindung von einem Knoten u zu einem Knoten v länger ist als die Verbindung von u über einen weiteren Knoten w zu v .

Zu jedem Graphen mit nicht-negativen Kantenlängen können wir eine sogenannte „Metrik“ angeben, die jeweils die kürzesten Verbindungen zwischen den Knoten widerspiegelt. Eine Metrik entspricht einem vollständigen Graphen mit Kantenlängen, die die Dreiecksungleichung erfüllen, d.h. für jeweils drei Knoten u, v, w ist die Länge der Kante $\{u, v\}$ nicht länger als die Summe der Längen der Kanten $\{u, w\}$ und $\{w, v\}$.

METRIC-TSP:

Gegeben sei eine Metrik $G = (V, E)$ mit n Knoten und Kantenlängen aus \mathbb{N} . Gesucht ist eine TSP-Tour minimaler Länge.

METRIC-TSP ist ebenfalls NP-hart, aber wir werden sehen, METRIC-TSP erlaubt einen konstanten Approximationsfaktor von $\frac{3}{2}$.

Als Warm-Up starten wir mit einer 2-Approximation.

Algorithmus METRIC-TSP-via-MST

- 1 Finde einen MST T von G , verdopple die Kanten von T und erhalte einen Euler-Graphen T' ;
- 2 Berechne eine Euler-Tour auf T' ;
- 3 Gib die Knoten aus V in der Reihenfolge aus, wie sie in der Euler-Tour erscheinen.

MST steht für minimum spanning tree (minimaler Spannbaum).

Ein Euler-Graph ist ein Graph, in dem jeder Knoten einen geraden Grad hat. Eine Euler-Tour durch einen Graphen ist ein Kreis, der jede Kante genau einmal enthält. Eine Euler-Tour existiert genau dann, wenn der Graph ein Euler-Graph ist.

Satz 5 Algorithmus METRIC-TSP-via-MST berechnet eine 2-Approximation für METRIC-TSP.

Beweis:

- Aus einer TSP-Tour können wir einen Spannbaum erzeugen, indem wir eine Kante löschen.
- Also ist ein minimaler Spannbaum nicht teurer als die Länge einer minimalen TSP-Tour.
- Die Länge der berechneten Euler-Tour entspricht den doppelten Kosten des minimalen Spannbaums, ist also höchstens zweimal so lang wie die minimale TSP-Tour.
- Das Überspringen von mehrfach besuchten Knoten in Schritt 3 macht wegen der Dreiecksungleichung die Tour nicht teurer.

□

Der folgende Algorithmus ist ein echter Klassiker unter den Approximationsalgorithmen und wurde von Christofides im Jahr 1976 vorgestellt.

Der Algorithmus von Cristofides

- 1 Berechne einen MST T von G ;
- 2 $V' := \{v \in V \mid v \text{ hat ungeraden Grad in } T\}$;
- 3 Finde ein min-cost Matching M auf V' ;
- 4 Finde eine Euler-Tour auf den Kanten aus T und M ;
- 5 Gib die Knoten aus V in der Reihenfolge aus,
wie sie in der Euler-Tour erscheinen.

Ein min-cost Matching auf V' ist eine Kantenmenge $M \subseteq E$, die jeden Knoten aus V' *genau einmal* abdeckt und dabei die kleinstmöglichen Kosten hat, d.h. die Summe der Kantenlängen in M ist so klein wie möglich. Als Matchingkanten sind nicht nur die Baumkanten von T , sondern alle Kanten aus E zwischen den Knoten in V' erlaubt. Beachte, jedes Knotenpaar in V' ist durch eine Kante aus E miteinander verbunden, weil E eine Metrik, also einen vollständigen Graphen, definiert. Die Laufzeit des Algorithmus wird durch die Berechnung dieses Matchings dominiert und ist somit $O(n^3)$.

Die Existenz eines perfekten Matchings ist gesichert, weil die Knoten in V' vollständig miteinander verbunden sind und V' eine gerade Anzahl von Knoten enthält. Letztere Eigenschaft folgt aus dem folgenden Lemma.

Lemma 6 Gegeben sei ein beliebiger Graph $H = (V, E)$. Sei $V' \subseteq V$ die Teilmenge der Knoten, die einen ungeraden Grad haben. Dann ist $|V'|$ eine gerade Zahl.

Beweis: Zum Zwecke des Widerspruchs nehmen wir an, dass $|V'|$ ungerade ist. Jede Kante in E ist inzident zu zwei Knoten, hat also zwei Endpunkte. Bezeichne q die Anzahl dieser Kantenendpunkte.

- Einerseits ist $q = 2|E|$, und somit ist q eine gerade Zahl.
- Andererseits entspricht q der Summe der Knotengrade aller Knoten in V . Da wir annehmen, dass $|V'|$ ungerade ist, ist auch die Summe der Knotengrade in V' ungerade. Alle anderen Knoten haben geraden Grad. Somit ist auch die Summe der Knotengrade aller Knoten, also q , eine ungerade Zahl.

Ein Widerspruch. Es folgt, $|V'|$ ist eine gerade Zahl. \square

Für eine Menge von Kanten $X \subseteq E$ (z.B. beschrieben in Form eines Matchings oder einer TSP-Tour) bezeichne $cost(X)$ die Kosten von X , also die Summe der Kantenlängen in X .

Bezeichne opt die minimalen Kosten einer TSP-Tour.

Lemma 7 Es gilt $cost(M) \leq \frac{1}{2}opt$.

Beweis:

- Sei τ eine optimale TSP-Tour, also $cost(\tau) = opt$.
- Aus τ erhalten wir einen Kreis τ' , der die Knoten in V' verbindet, wenn wir alle nicht in V' enthaltenen Knoten streichen. Wegen der Dreiecksungleichung gilt $cost(\tau') \leq cost(\tau)$.
- τ' ist die Summe von zwei perfekten Matchings, die jeweils aus jeder zweiten Kanten auf dem Kreis τ' gebildet werden. Das günstigere der beiden Matchings hat höchstens die Kosten $\frac{1}{2}cost(\tau') \leq \frac{1}{2}cost(\tau) = \frac{1}{2}opt$.
- Also hat das günstigste perfekte Matching auf V' höchstens die Kosten $\frac{1}{2}opt$.

□

Satz 8 Der Algorithmus von Christofides berechnet eine $\frac{3}{2}$ -Approximation für METRIC-TSP.

Beweis: Die in Schritt 3 berechnete Euler-Tour hat die Kosten

$$cost(T) + cost(M) \leq opt + \frac{1}{2}opt = \frac{3}{2}opt .$$

In Schritt 4 erhöhen sich die Kosten aufgrund der Dreiecksungleichung nicht. □

Ein FPTAS für das Rucksackproblem

Ein Algorithmus \mathcal{A} für ein Optimierungsproblem Π wird als FPTAS (fully polynomial time approximation scheme) bezeichnet, falls \mathcal{A} bei Eingabe einer Instanz I sowie einer Zahl $\epsilon > 0$ eine zulässige Lösung mit Approximationsfaktor $1 + \epsilon$ bzw. $1 - \epsilon$ in Zeit polynomiell in der Länge der Eingabe und polynomiell in $\frac{1}{\epsilon}$ berechnet.

RUCKSACK (KNAPSACK):

Gegeben sei eine Menge von Objekten $[n] = \{1, \dots, n\}$ mit Gewichten $w_1, \dots, w_n \in \mathbb{N}$ und Nutzenwerten $p_1, \dots, p_n \in \mathbb{N}$ sowie eine Gewichtsschranke $b \in \mathbb{N}$.

Gesucht ist eine Teilmenge der Objekte $K \subseteq [n]$, die $p(K) = \sum_{i \in K} p_i$ maximiert unter der Nebenbedingung $w(K) = \sum_{i \in K} w_i \leq b$.

O.B.d.A. nehmen wir an, dass $w_i \leq b$ für jedes $i \in [n]$.

Wir werden zeigen, dass das Rucksackproblem ein FPTAS mit Laufzeit $O(n^3/\epsilon)$ hat. Die Basis für dieses FPTAS ist ein pseudopolynomieller Algorithmus.

Lemma 9 Das Rucksackproblem hat einen pseudopolynomiellen Algorithmus mit Laufzeit $O(n^2P)$ wobei $P = \max_i p_i$.

Beweis: Für $i \in [n]$ und $p \in \{0, \dots, nP\}$, sei $A_{i,p}$ das kleinstmögliche Gewicht, mit dem man den Nutzen p exakt erreichen kann, wenn man nur Objekte aus $[i] = \{1, \dots, i\}$ verwenden darf. Wir setzen $A_{i,p} = \infty$, falls der Nutzen p nicht durch eine Teilmenge von $[i]$ erreicht werden kann. Ansonsten gilt

$$A_{i+1,p} = \min(A_{i,p}, A_{i,p-p_{i+1}} + w_{i+1}) .$$

Wir verwenden den Ansatz der dynamischen Programmierung und berechnen Zeile für Zeile alle Einträge in der Tabelle bzw. Matrix $(A_{i,p})_{i \in \{1, \dots, n\}, p \in \{0, \dots, nP\}}$. Die Laufzeit entspricht der Größe der Tabelle und ist somit $O(n^2P)$.

Der gesuchte, maximal erreichbare Nutzenwert ist

$$\max\{p \mid A_{n,p} \leq b\} .$$

Die zugehörige Rucksackbelegung lässt sich durch Abspeichern geeigneter Zusatzinformationen zu den einzelnen Tabelleneinträgen rekonstruieren, so dass die optimale Rucksackbelegung in Zeit $O(n^2P)$ berechnet werden kann. \square

Satz 10 Das Rucksackproblem hat ein FPTAS mit Laufzeit $O(n^3/\epsilon)$.

Beweis:

Algorithmus RUCKSACK-FPTAS:

1. skaliere und runde die Nutzenwerte, so dass der maximale Nutzenwert nicht größer als $P' = \frac{n}{\epsilon}$ ist, d.h.
 - setze $\alpha = \frac{n}{\epsilon P}$;
 - für jedes Objekt $i \in [n]$, setze $p'_i = \lfloor \alpha p_i \rfloor$;
2. berechne eine optimale Rucksackbepackung K für die Nutzenwerte p'_1, \dots, p'_n mit dem pseudopolynomiellen Algorithmus aus Lemma 9, und gib K aus.

Die Laufzeit dieses Algorithmus ist $O(n^2 P') = O(n^3/\epsilon)$.

Wir müssen noch zeigen, dass der Algorithmus einen Approximationsfaktor von $1 - \epsilon$ garantiert. Sei K^* eine optimale Rucksackbepackung.

Zu zeigen: $p(K) \geq (1 - \epsilon)p(K^*)$.

- Wir skalieren die gerundeten Nutzenwerte wieder rauf und setzen $p''_i = p'_i/\alpha$. Die Rucksackbepackung K ist optimal für die Nutzenwerte p'_i und somit auch optimal für die Nutzenwerte p''_i .
- Für jedes Objekt macht der Algorithmus einen Rundungsfehler. Objekte sehen weniger profitabel aus als sie sind. Der Rundungsfehler ist

$$p_i - p''_i = p_i - \frac{\lfloor \alpha p_i \rfloor}{\alpha} \leq p_i - \frac{\alpha p_i - 1}{\alpha} = \frac{1}{\alpha} .$$

- Der Rundungsfehler für die optimale Lösung K^* ist somit höchstens

$$p(K^*) - p''(K^*) \leq \sum_{i \in K^*} \frac{1}{\alpha} \leq \frac{n}{\alpha} = \epsilon P \leq \epsilon p(K^*) .$$

Die Ungleichung $p(K^*) \geq P$ gilt, weil der optimale Nutzen $p(K^*)$ mindestens so groß ist wie der Nutzen des Objektes mit dem maximalen Nutzenwert P .

- Also gilt $p''(K^*) \geq (1 - \epsilon)p(K^*)$ und somit

$$p(K) \geq p''(K) \geq p''(K^*) \geq (1 - \epsilon)p(K^*) .$$

□

Stark und schwach NP-harte Probleme

Wenn wir alle Zahlen in der Eingabe des Rucksackproblems unär kodieren, so wächst die Eingabelänge. Bezeichne N die Eingabelänge des Rucksackproblems bei unärer Kodierung. Es gilt $N \geq n + P$. Der pseudopolynomielle Algorithmus aus Lemma 9 hat Laufzeit $O(n^2 P) = O(N^3)$. Das Rucksackproblem mit unärer Kodierung ist also in Polynomialzeit lösbar.

Das Rucksackproblem ist also nur dann NP-hart, wenn wir vereinbaren die Eingabe binär zu kodieren. Das Rucksackproblem mit unärer Kodierung ist hingegen in Polynomialzeit lösbar.

Definition 11 Ein NP-hartes Problem, das einen pseudopolynomiellen Algorithmus hat, wird als schwach NP-hart bezeichnet. Ein Problem, das auch bei unärer Kodierung der Eingabezahlen NP-hart bleibt, wird als stark NP-hart bezeichnet.

Falls $P \neq NP$, so haben stark NP-harte Probleme keinen pseudopolynomiellen Algorithmus.

Das Rucksackproblem ist schwach NP-hart, weil es einen pseudopolynomiellen Algorithmus hat. Min-Vertex-Cover ist hingegen stark NP-hart, weil keine Zahlen in der Eingabe vorkommen. Auch Set-Cover ist stark NP-hart, da dieses Problem bereits NP-hart ist, wenn alle Mengen Kosten 1 haben. Der folgende Satz liefert, dass Min-Vertex-Cover und Set-Cover kein FPTAS haben.

Satz 12 Sei Π ein Optimierungsproblem mit ganzzahliger Zielfunktion. Für eine Eingabe I , bezeichne $opt(I) \in \mathbb{N}$ den Wert einer optimalen Lösung für I . Für jede Eingabe I gelte $opt(I) < p(N)$, wobei N die unäre Eingabelänge für I und $p(\cdot)$ ein beliebiges Polynom bezeichnet. Falls Π stark NP-hart ist und $P \neq NP$, so hat Π kein FPTAS.

Der obige Satz macht einige kompliziert klingende technische Annahmen, die aber für typische Optimierungsprobleme ohne weiteres erfüllt sind. Als Faustformel können wir uns merken, stark NP-harte Probleme haben kein FPTAS oder $P = NP$.

Beweis von Satz 12:

Zum Zwecke des Widerspruchs nehmen wir an, Π ist stark NP-hart und hat ein FPTAS. Aus dem FPTAS werden wir einen pseudopolynomiellen Algorithmus konstruieren. Das würde $P = NP$ bedeuten. Also hat Π unter der Hypothese $P \neq NP$ kein FPTAS.

Der Einfachheit halber nehmen wir an, Π ist ein Minimierungsproblem. Sei \mathcal{A} ein FPTAS für Π . Bei Eingabe I mit unärer Länge N setzen wir $\epsilon = 1/p(N)$. Die von \mathcal{A} berechnete Lösung hat damit den Wert höchstens

$$(1 + \epsilon) \text{opt}(I) < \text{opt}(I) + \epsilon p(N) = \text{opt}(I) + 1 .$$

Also ist \mathcal{A} gezwungen die optimale Lösung zu berechnen, weil ja die Zielfunktion ganzzahlig ist. Die Laufzeit ist polynomiell in $\frac{1}{\epsilon} = p(N)$, also polynomiell in N , und somit ist \mathcal{A} pseudopolynomiell. Das liefert den gewünschten Widerspruch.

□

Die Umkehrung von Satz 12 gilt übrigens nicht. Es gibt Probleme mit pseudopolynomiellen Algorithmen, für die kein FPTAS bekannt ist, und sogar Probleme mit pseudopolynomiellen Algorithmen, für die es beweisbar kein FPTAS gibt, es sei denn $P = NP$.

Eine wichtige Grundlage für das FPTAS des Rucksackproblems ist der pseudopolynomielle Algorithmus mit Laufzeit $O(n^2 P)$. Ein anderes dynamische Programm hat Laufzeit $O(n^2 W)$, wobei W das maximale Gewicht ist. Dieser Algorithmus ist also pseudopolynomiell in den Gewichten statt in den Nutzenwerten. Um aus diesem Algorithmus ein FPTAS zu gewinnen, müsste man die Gewichte runden. Das ist aber problematisch, denn

- rundet man die Gewichte ab, so berechnet der Algorithmus möglicherweise eine unzulässige Lösung;
- rundet man die Gewichte jedoch auf, so ignoriert der Algorithmus möglicherweise einige zulässige Lösungen.

Um ein FPTAS zu erhalten, sollte die Laufzeit also pseudopolynomiell in der Zielfunktion und nicht in den Nebenbedingungen sein. Ist dies gegeben, so ist im Einzelfall zu überprüfen, ob und wie man durch geschicktes Runden zu einem FPTAS gelangen kann.

Bin Packing ist stark NP-vollständig

Wir zeigen das folgende Zahlenproblem ist stark NP-vollständig. Das Problem ist also auch dann NP-hart, wenn die Eingabezahlen unär kodiert sind.

BIN PACKING:

Gegeben sei eine Menge von Objekten $[n] = \{1, \dots, n\}$ mit Gewichten $w_1, \dots, w_n \in \mathbb{N}$ sowie zwei natürliche Zahlen m und b . Gesucht ist eine Einteilung der Objekte in m Teilmengen, so dass jede Teilmenge ein Gewicht von höchstens b hat.

Intuitiv suchen wir also nach einer Verteilung von n gewichteten Objekten auf m Kisten, die jeweils eine Gewichtskapazität von b haben.

Satz 13 BIN PACKING ist stark NP-vollständig.

Beweis: Offensichtlich liegt BIN PACKING in NP, denn eine richtig geratene Lösung kann ohne Weiteres in Polynomialzeit verifiziert werden.

Wir müssen also nur noch die starke NP-Härte nachweisen. Dies tun wir durch eine Reduktion von einem NP-harten Problem in dem keine Zahlen in der Eingabe vorkommen. Es handelt sich um ein ungewichtetes Matchingproblem auf tripartiten Hypergraphen.

TRIPARTITES MATCHING:

Gegeben sei eine Menge von Jungen $B = \{b_1, \dots, b_n\}$, eine Menge von Mädchen $G = \{g_1, \dots, g_n\}$, eine Menge von Häusern $H = \{h_1, \dots, h_n\}$ sowie eine Menge von Tripeln $T = \{t_1, \dots, t_m\} \subseteq B \times G \times H$. Gesucht ist eine Teilmenge der Größe n von T , so dass jeder Junge, jedes Mädchen und jedes Haus in einem dieser n Tripel enthalten ist.

Im Gegensatz zu den bekannten Matchingproblemen auf Graphen, die ja bekanntermaßen in Polynomialzeit gelöst werden können, ist das oben beschriebene Matchingproblem auf Hypergraphen NP-hart. Die NP-Härte von TRIPARTITE MATCHING beruht auf einer Reduktion von 3-SAT und kann z.B. im Buch von Garey und Johnson nachgelesen werden.

O.B.d.A. nehmen wir an jeder Knoten (Junge, Mädchen, Haus) wird durch mind. eine Hyperkante (Tripel) abgedeckt.

Wir präsentieren nun eine Polynomialzeitreduktion des Tripartite-Matching-Problems auf das unäre Bin-Packing-Problem. Dazu übersetzen wir die Struktur des Tripartite-Matching-Problems in die Gewichte des Bin-Packing-Problems. Bei dieser Transformation dürfen nur polynomiell große Zahlen vorkommen, da die Eingabezahlen ja in polynomieller Zeit unär kodiert werden müssen.^a

Die Bin-Packing-Instanz, die wir konstruieren hat m Kisten, jeweils eine Kiste für jedes Tripel. Insgesamt gibt es $n = 4m$ Objekte. Zunächst jeweils ein Objekt für jedes Vorkommen eines Jungen, Mädchens oder Hauses in einer Hyperkante. Die Objekte für das Vorkommen eines Jungen b_i bezeichnen wir mit $b_i[1], b_i[2], \dots, b_i[N(b_i)]$, wobei $N(b_i)$ der Anzahl der Vorkommen von b_i entspricht. Für Mädchen und Häuser verwenden wir analoge Bezeichnungen. Zudem gibt es m Objekte für die Tripel, die wir mit t_1, \dots, t_m bezeichnen.

^aWir erinnern uns: Bei der Reduktion von 3SAT auf das nur schwach NP-harte Rucksackproblem tauchen exponentiell große Zahlen auf.

Die Gewichte der Objekte werden in der folgenden Tabelle beschrieben. Dabei ist M eine genügend große Zahl; z.B. $M = 100n$. Beachte, dass das Gewicht eines der Vorkommen eines jeden Jungen, Mädchens bzw. Hauses einer anderen Größenformel folgt als die anderen. Beliebigerweise handelt es sich dabei jeweils um das jeweils erste Vorkommen.

Objekt	Gewicht
Junge $b_i[1]$	$10M^4 + iM + 1$
Junge $b_i[q], q > 1$	$11M^4 + iM + 1$
Mädchen $g_j[1]$	$10M^4 + jM^2 + 2$
Mädchen $g_j[q], q > 1$	$11M^4 + jM^2 + 2$
Haus $h_l[1]$	$10M^4 + lM^3 + 4$
Haus $h_l[q], q > 1$	$8M^4 + lM^3 + 4$
Tripel $(b_i, g_j, h_l) \in T$	$10M^4 + 8 - iM - jM^2 - lM^3$

Die Kapazität jeder Kiste definieren wir als $b = 40M^4 + 15$, also gerade genug um jeweils ein Tripel, einen Jungen, ein Mädchen und ein Haus aufzunehmen, zumindest wenn es sich um die ersten Vorkommen handelt.

Wir nehmen an, es gibt eine Möglichkeit diese Objekte in m Kisten unterzubringen. Dann gilt

- Alle Kisten sind exakt gefüllt, denn sowohl die Summe aller Gewichte als auch die Summe aller Kapazitäten ist $m(40M^4 + 15)$.
- Jede Kiste beinhaltet genau vier Objekte, denn alle Gewichte liegen zwischen $\frac{1}{3}$ und $\frac{1}{5}$ der Kistenkapazität.
- Jede Kiste enthält einen Jungen, ein Mädchen, ein Haus und ein Tripel, da dieses die einzige mögliche Zusammenstellung von vier Objekten ist, welche die Gewichtssumme $15 \pmod{M}$ ergibt.
- Die Kiste mit dem Tripel (b_i, g_j, h_l) enthält auch jeweils ein Objekt für den Jungen b_i , das Mädchen g_j und das Haus h_l , denn wäre beispielsweise kein Objekt für das Mädchen g_j enthalten, so wäre die Gewichtssumme nicht $15 \pmod{M^3}$.
- Entweder entsprechen alle diese drei Objekte dem ersten Vorkommen des jeweiligen Jungen, Mädchen oder Hauses oder keines dieser Objekte, denn sonst könnte das Gewicht $40M^4 + 15$ nicht erreicht werden.

Wir fassen zusammen: Falls es also eine Möglichkeit gibt, die Objekte in m Kisten unterzubringen, so gibt es n Kisten die die jeweils ersten Vorkommen enthalten. Die Tripel in diesen Kisten sind die gewünschten Matchingkanten, weil jedes Objekt in nur einer dieser Kanten vorkommt.

Umgekehrt gilt, falls es ein tripartites Matching gibt, so können wir die n Matchingtripel jeweils zusammen mit den Objekten für das erste Vorkommen der entsprechenden Jungen, Mädchen und Häuser in eine Kiste legen. Die $m - n$ verbleibenden Kisten können wir mit den $m - n$ verbleibenden Tripeln sowie den zugehörigen Objekten füllen. Für jede Kiste ergibt sich dabei eine Gewichtssumme von $40M^4 + 15 = b$.

Somit liefert ein polynomieller Algorithmus für das unäre Bin-Packing-Problem auch einen polynomiellen Algorithmus für das NP-harte Tripartite-Matching-Problem.

□

Makespan-Scheduling auf identischen Maschinen

Wir untersuchen ein fundamentales Problem aus der Schedulingtheorie.

MAKESPAN SCHEDULING:

Gegeben sei eine Menge von Jobs $[n] = \{1, \dots, n\}$ mit Größen $p_1, \dots, p_n \in \mathbb{N}$ und eine natürliche Zahl m .

Gesucht ist eine Zuteilung $f : [n] \rightarrow [m]$ der n Jobs auf m identische Maschinen, so dass der Makespan, also

$$\max_{j \in [m]} \sum_{i \in [n]: f(i)=j} p_i$$

minimiert wird.

Diese Zuteilung wird als Schedule (Ablaufplan) bezeichnet. Zu einem Schedule gehört normalerweise auch eine Beschreibung, in welcher Reihenfolge die Jobs auf den einzelnen Maschinen abgearbeitet werden. Diese Reihenfolge spielt jedoch bei der Makespan-Minimierung offensichtlich keine Rolle.

Algorithmus Least-Loaded (LL):

Für $i = 1$ bis n , weise Job i derjenigen Maschine zu, die bisher die geringste Last hat.

Wie gut ist diese Heuristik?

ein Beispiel:

- Sei $n = m^2 + 1$.
- Jobs 1 bis m^2 haben Größe 1.
- Job $m^2 + 1$ hat Größe m .
- Die LL-Heuristik erreicht den Makespan $2m$.
- Der optimale Makespan ist $m + 1$.

Damit ist der Approximationsfaktor bestenfalls 2. Der folgende Satz zeigt, dass dieses Beispiel tatsächlich den schlimmsten Fall beschreibt.

Satz 14 LL garantiert eine 2-Approximation.

Beweis: Es gelten die folgenden zwei trivialen unteren Schranken für ein optimales Schedule:

$$opt \geq \max \left\{ \max_{i \in [n]} (p_i), \frac{1}{m} \sum_{i \in [n]} p_i \right\} .$$

Wir gehen davon aus, jede Maschine arbeitet ihre Jobs nacheinander in der Reihenfolge ihrer Zuweisung ab. Sei i' der Index desjenigen Jobs, der als letztes fertig wird. Sei $j' = f(i')$, d.h. Maschine j' wird als letztes fertig und bestimmt damit den Makespan.

Zum Zeitpunkt als Job i' Maschine j' zugewiesen wurde, hatte diese Maschine die geringste Last. Die Last von Maschine j' zu diesem Zeitpunkt war also höchstens $\frac{1}{m} \sum_{i < i'} p_i$. Damit ist die Last von Maschine j' höchstens

$$\left(\frac{1}{m} \sum_{i' < i} p_i \right) + p_{i'} \leq 2opt .$$

□

Algorithmus Longest-Processing-Time (LPT):

1. Sortiere die Jobs, so dass $p_1 \geq p_2 \geq \dots \geq p_n$;
2. Für $i = 1$ bis n , weise Job i derjenigen Maschine zu, die bisher die geringste Last hat.

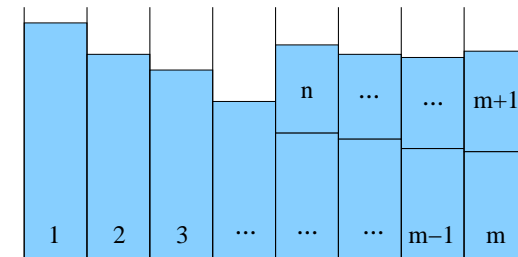
Graham hat 1969 gezeigt, dass LPT einen Approximationsfaktor von höchstens $\frac{4}{3}$ hat. Auch diese Schranke ist scharf.

Satz 15 LPT garantiert eine $\frac{4}{3}$ -Approximation.

Beweis:

- Zum Zwecke des Widerspruchs nehmen wir an, es gibt eine Eingabeinstanz, für die LPT einen Makespan von $\tau > \frac{4}{3}opt$ auf m Maschinen erzeugt.
- Sei p_1, p_2, \dots, p_n eine Eingabeinstanz minimaler Länge mit $\tau > \frac{4}{3}opt$. Es gelte $p_1 \geq p_2 \geq \dots \geq p_n$.
- Sei i' der Index desjenigen Jobs, der als letztes fertig wird. Es gilt $i' = n$, sonst wäre ja $p_1, \dots, p_{i'}, i' < n$, eine kürzere Eingabesequenz mit $\tau > \frac{4}{3}opt$, aber wir haben angenommen p_1, p_2, \dots, p_n ist die kürzeste Eingabe mit dieser Eigenschaft.
- Job n wird auf der am wenigsten belasteten Maschine platziert. Zum Zeitpunkt der Zuweisung von Job n hat diese Maschine höchstens Last $\frac{1}{m} \sum_{i=1}^{n-1} p_i \leq opt$. Damit $\tau > \frac{4}{3}opt$ gilt, muss also gelten $p_n > \frac{1}{3}opt$.
- Aus $p_n > \frac{1}{3}opt$ folgt, jeder Job ist größer als $\frac{1}{3}opt$, weil $p_1 \geq p_2 \geq \dots \geq p_n$.

- Falls jeder Job größer als $\frac{1}{3}opt$, so kann ein optimaler Schedule nicht mehr als zwei Jobs an eine Maschine zuweisen.
- Jeder Schedule mit höchstens zwei Jobs pro Maschine kann in den folgenden schematisch dargestellten Schedule überführt werden, ohne den Makespan zu erhöhen (vgl. Übung).



- Dieser Schedule entspricht jedoch genau dem LPT-Schedule. Also berechnet LPT einen optimalen Schedule.
- Dies ist ein Widerspruch zu unserer Annahme $\tau > \frac{4}{3}opt$. Somit folgt $\tau \leq \frac{4}{3}opt$, und der Approximationsfaktor ist höchstens $\frac{4}{3}$.

□

Eine einfache Reduktion vom Bin-Packing-Problem zeigt, dass das Makespan-Scheduling-Problem stark NP-hart ist. Es gibt also kein FPTAS für dieses Problem. Trotzdem werden wir zeigen, dass das Problem in Polynomialzeit beliebig gut approximiert werden kann.

Wir sagen, ein Optimierungsproblem Π hat ein PTAS (polynomial time approximation scheme), falls für jede Konstante $\epsilon > 0$ eine $(1 \pm \epsilon)$ -Approximation in polynomieller Zeit berechnet werden kann. Der Unterschied zum FPTAS ist, dass ϵ hierbei als konstant angesehen wird.

Wir werden zeigen, dass das Makespan-Scheduling-Problem ein PTAS mit Laufzeit ungefähr n^{1/ϵ^2} hat. Für ein FPTAS wäre eine derartige Laufzeitschranke nicht zulässig, weil sie nicht polynomiell in $\frac{1}{\epsilon}$ ist.

Für kleines ϵ ist die obige Laufzeitschranke offensichtlich nicht praktikabel. Wir wollen uns trotzdem einmal ansehen, wie ein derartiges Approximationsschema aussieht. Dieses PTAS ist aber eher unter komplexitätstheoretischen als unter praktischen Gesichtspunkten interessant.

Ein PTAS für MAKESPAN-SCHEDULING:

1. Ein *Orakel* verrät uns den Wert des optimalen Makespans, den wir Z nennen.
2. Wir kümmern uns zunächst um die großen Jobs, d.h. um die Jobs $\{i \in [n] \mid p_i > \epsilon Z\}$.

- a) Wir skalieren und runden die Größen dieser Jobs d.h. wir setzen

$$p'_i = \left\lceil \frac{p_i}{\epsilon^2 Z} \right\rceil .$$

- b) Wir berechnen einen Schedule für die aufgerundeten Jobgrößen mit Makespan höchstens

$$Z' = \left\lceil (1 + \epsilon) \frac{1}{\epsilon^2} \right\rceil .$$

3. Jetzt kümmern wir uns um die kleinen Jobs, d.h. um die Jobs $\{i \in [n] \mid p_i \leq \epsilon Z\}$. Wir verteilen diese Jobs mittels der LL-Heuristik auf das durch die großen Jobs entstandene Lastgebirge.

Bemerkungen zur Skalierung und Rundung in Schritt 2:

Das Skalieren in Schritt 2a) läßt sich am Besten durch ein Beispiel illustrieren. Sei $Z = 1000$ und $\epsilon = 10\%$. Die großen Jobs haben dann mindestens Größe $\epsilon Z = 100$. Wir gehen schrittweise vor: skalieren zunächst ohne zu runden, d.h. wir setzen

$$p_i^* = \frac{p_i}{\epsilon^2 Z} = \frac{p_i}{10} .$$

Nach dem Runden setzen wir dann $p_i' = \lceil p_i^* \rceil$. Aus $p_i = 101$ ergibt sich also beispielsweise $p_i^* = 10.1$ und $p_i' = 11$. Der relative Rundungsfehler in diesem Beispiel ist somit

$$\frac{p_i' - p_i^*}{p_i^*} = \frac{11 - 10.1}{10.1} \leq 10\% = \epsilon .$$

Dies gilt auch im Allgemeinen.

Lemma 16 Der relative Rundungsfehler $(p_i' - p_i^*)/p_i^*$ ist höchstens ϵ .

Beweis: Für jeden großen Job $i \in [n]$ gilt $p_i > \epsilon Z$ und somit $p_i^* \geq \epsilon Z / (\epsilon^2 Z) = 1/\epsilon$. Es folgt

$$\frac{p_i' - p_i^*}{p_i^*} \leq \frac{1}{1/\epsilon} = \epsilon .$$

□

Das Aufrunden der skalierten Größen der großen Jobs verzehrt diese Größen also höchstens um den Faktor $1 + \epsilon$.

Für die eigentlichen Größen der großen Jobs gibt es einen Schedule mit Makespan höchstens Z . Für die skalierten (ungerundeten) Jobgrößen gibt es also einen Schedule mit Makespan $\frac{Z}{\epsilon^2 Z} = \frac{1}{\epsilon^2}$. Durch die Rundung erhöht sich dieser Wert maximal um den Faktor $1 + \epsilon$. Also gibt es für die skalierten und gerundeten Jobgrößen einen Schedule mit Makespan höchstens

$$(1 + \epsilon) \frac{1}{\epsilon^2} .$$

Da der Makespan ganzzahlig ist, kann er also höchstens den Wert

$$\left\lceil (1 + \epsilon) \frac{1}{\epsilon^2} \right\rceil = Z'$$

annehmen. Somit existiert der in Schritt 2b) beschriebene Schedule. Wie aber können wir diesen Schedule berechnen? – Bevor wir diese Frage klären, kümmern wir uns zunächst um den Approximationsfaktor.

Lemma 17 Der skizzierte Algorithmus berechnet eine $(1 + \epsilon)$ -Approximation für den minimalen Makespan.

Beweis: Zunächst nehmen wir an, es gibt nur große Jobs. Dann berechnet der Algorithmus einen Schedule mit Makespan höchstens

$$Z'(\epsilon^2 Z) = \left\lceil (1 + \epsilon) \frac{1}{\epsilon^2} \right\rceil (\epsilon^2 Z) \leq (1 + \epsilon)Z ,$$

also eine $(1 + \epsilon)$ -Approximation. Die kleinen Jobs behandeln wir in einer Fallunterscheidung.

- *Fall 1:* Die LL-Heuristik erhöht den Makespan nicht. Dann erhalten wir eine $(1 + \epsilon)$ -Approximation aufgrund obiger Überlegungen für die großen Jobs.
- *Fall 2:* Die LL-Heuristik erhöht den Makespan. In diesem Fall garantiert die Heuristik, dass der Lastunterschied zwischen der am stärksten und der am schwächsten belasteten Maschine nicht größer als der größte der kleinen Jobs ist. Damit ist der Lastunterschied zwischen unterschiedlichen Maschinen höchstens ϵZ , und auch in diesem Fall ist eine $(1 + \epsilon)$ -Approximation sichergestellt.

□

Laufzeitanalyse von Schritt 2: Zur Vereinfachung der Notation nehmen wir an, dass wir n große Jobs haben. In Schritt 2 müssen wir eine Variante des Bin-Packing-Problems lösen.

BIN PACKING mit eingeschränkten Gewichten:

Gegeben sei eine Menge von Objekten $[n] = \{1, \dots, n\}$ mit Gewichten $w_1, \dots, w_n \in [k]$, $k \geq 1$, sowie zwei natürliche Zahlen $m \geq 1$ und $b \geq 1$.

Gesucht ist eine Verteilung der Objekte auf m Kisten (Bins), die jeweils ein Gewicht von höchstens b tragen können.

Die Objekte des Bin-Packing-Problems repräsentieren dabei die großen Jobs des Schedulingproblems. Die Gewichte entsprechen den Jobgrößen p'_1, \dots, p'_n , und die Gewichtsschranke b entspricht der oberen Schranke für den Makespan Z' . Eine geeignete Abschätzung für k werden wir später bestimmen.

Die Lösung für das Bin-Packing-Problem spezifiziert eine Einteilung in m Teilmengen mit Gewicht höchstens b aus der wir leicht einen Schedule mit Makespan höchstens Z' berechnen können.

Lemma 18 Das Bin-Packing-Problem mit eingeschränkten Gewichten kann in Zeit $O((bn)^k)$ gelöst werden.

Beweis: Wir verwenden dynamische Programmierung und lösen dabei die folgenden Teilprobleme

- Sei $f(n_1, n_2, \dots, n_k)$ die minimale Anzahl von Kisten mit Gewichtsschranke b , in die wir eine Menge von Objekten bestehend aus n_i ($i \in [k]$) vielen Objekten mit Gewicht i packen können.
- Sei $Q = \{(q_1, \dots, q_k) \mid f(q_1, q_2, \dots, q_k) = 1\}$, d.h. Q beschreibt alle Gewichtskombinationen, die in eine Kiste passen. Beachte, $(q_1, \dots, q_k) \in Q$ impliziert $q_i \in \{0, \dots, b\}$ für jedes $i \in [k]$. Es folgt $|Q| \leq (b+1)^k$.

Die Funktion f erfüllt die folgende Rekursionsgleichung.

$$f(n_1, n_2, \dots, n_k) = 1 + \min_{q \in Q} f(n_1 - q_1, n_2 - q_2, \dots, n_k - q_k)$$

Wir berechnen die Lösung dieser Gleichung für alle k -Tupel aus $\{0, \dots, n\}^k$ in einer Tabelle der Größe $(n+1)^k$. Falls diese Lösung den Wert mindestens k hat, existiert eine Verteilung der Objekte auf k Kisten, und diese Verteilung kann leicht aus der Tabelle extrahiert werden.

Die Berechnung eines Tabelleneintrages kostet Zeit $|Q| \leq (b+1)^k$. Damit ist die Laufzeit $O(b^k n^k)$. \square

Was sind die Werte von b und k bezogen auf unser Schedulingproblem? – Die maximale Größe eines Jobs vor der Skalierung ist durch Z beschränkt, weil es ja einen Schedule mit Makespan Z gibt. Nach der Skalierung und Rundung ist die maximale Größe also höchstens $\lceil Z/(\epsilon^2 Z) \rceil = \lceil \frac{1}{\epsilon^2} \rceil$. Also können wir

$$k = \left\lceil \frac{1}{\epsilon^2} \right\rceil$$

setzen. Bei der Entwicklung eines PTAS fassen wir ϵ als positive Konstante auf. Somit gilt

$$b = Z' = \left\lceil (1 + \epsilon) \frac{1}{\epsilon^2} \right\rceil = O(1) .$$

Damit können wir die Laufzeit von Schritt 2 durch

$$O((bn)^k) = O\left(n^{\lceil 1/\epsilon^2 \rceil}\right)$$

abschätzen. Beachte, für konstantes $\epsilon > 0$ ist dies eine polynomielle Laufzeitschranke.

Zum Schluss müssen wir uns noch um das *Orakel* in Schritt 1 kümmern. Wir beobachten

- falls $Z \geq opt$, so ist der Algorithmus erfolgreich, d.h. er findet einen Schedule mit Makespan höchstens $(1 + \epsilon)Z$;
- falls $Z < opt$, so ist der Algorithmus möglicherweise nicht erfolgreich.

Wir suchen einen Wert für $Z \leq opt$, für den der Algorithmus erfolgreich ist. Diese Suche führen wir in Form einer Binärsuche durch.

Der Wert $S = \sum_i p_i$ ist eine obere Schranke für den Makespan. Wir können also einen geeigneten Wert für Z durch eine Binärsuche auf den Zahlen $\{1, \dots, S\}$ mit $O(\log S)$ vielen Aufrufen unseres Algorithmus finden.

Sei N die Länge der Eingabe in Bits. Dann gilt $\log S \leq N$. Die Laufzeit des Algorithmus ist somit $O(Nn^{\lceil 1/\epsilon^2 \rceil})$. Für konstantes $\epsilon > 0$ ist die Laufzeit also polynomiell in der Eingabelänge.

Wenn wir alle Ergebnisse zusammenfassen erhalten wir den folgenden Satz.

Satz 19 Es gibt ein PTAS für das Makespan-Scheduling-Problem auf identischen Maschinen. \square

Wegen des dramatischen Einflusses von ϵ auf die Laufzeit kann dieses PTAS nicht als praktisch angesehen werden. Wenn wir beispielsweise mit LPT konkurrieren wollen, so müssen wir $\epsilon = \frac{1}{3}$ setzen und erhalten eine Laufzeitschranke von $O(Nn^9)$. Das PTAS ist somit zwar nicht praktikabel, aber dennoch von großem theoretischen Interesse, weil es zeigt, dass es keine untere Schranke für den besten in Polynomialzeit erreichbaren Approximationsfaktor geben kann.

Makespan-Scheduling auf allgemeinen Maschinen

Es gibt verschiedene Varianten des Makespan-Scheduling-Problems.

- Scheduling auf identischen Maschinen: Job $i \in [n]$ hat Laufzeit p_i auf jeder Maschine.
- Scheduling auf Maschinen mit Geschwindigkeiten s_1, \dots, s_m : Job $i \in [n]$ hat Laufzeit $\frac{p_i}{s_j}$ auf Maschine $j \in [m]$.
- Scheduling auf allgemeinen Maschinen: Die Eingabe besteht aus einer Matrix $(p_{ij})_{i \in [n], j \in [m]}$. Dabei bezeichnet p_{ij} die Laufzeit von Job $i \in [n]$ auf Maschine $j \in [m]$.

Die ersten beiden Probleme haben ein PAS. Das Approximationsschema für das erste Problem haben wir vorgestellt. Das Schema für das zweite Problem ist ähnlich.

Für das dritte Problem, Makespan-Scheduling auf allgemeinen Maschinen, ist kein PAS bekannt. Der beste bekannte Algorithmus hat den Approximationsfaktor 2. Diesen Algorithmus von Lenstra, Shmoys und Tardos (1990) werden wir uns im Folgenden näher anschauen.

ILP-Formulierung des allgemeinen Schedulingproblems

Wir verwenden Indikatorvariablen

$$x_{ij} \in \{0, 1\}, i \in [n], j \in [m]$$

sowie eine Variable t , die dem Makespan entspricht.

Die Zielfunktion lautet

minimiere t

Die Nebenbedingungen sind

$$\forall i \in [n] : \sum_{j \in [m]} x_{ij} \geq 1$$

$$\forall j \in [m] : \sum_{i \in [n]} x_{ij} p_{ij} \leq t$$

$$\forall i \in [n], j \in [m] \quad x_{ij} \in \{0, 1\}$$

Wir können dieses ILP relaxieren, indem wir die Ganzzahligkeit aufgeben. Das so erhaltene LP kann in Polynomialzeit gelöst werden. Anschließend könnte man versuchen durch LP-Rundung eine ganzzahlige Lösung zu erhalten, die nah an der Lösung des LPs ist. Das folgende Beispiel zeigt jedoch, dass diese Idee so nicht aufgehen kann.

Beispiel: Wir nehmen an, es gibt nur einen Job und dieser Job hat Laufzeit 1 auf jeder Maschine. Dann hat die optimale ILP-Lösung den Wert 1. Die optimale Lösung des durch Relaxierung entstandenen LPs hingegen hat den Wert $\frac{1}{m}$. Damit ist der Faktor zwischen ILP- und LP-Optimum, das sogenannte Integrality-Gap, gleich m .

Ein derartig großes Integrality-Gap bedeutet, dass das Ergebnis des randomisierten Rundens sehr weit vom optimalen Wert des LPs entfernt sein muss. In diesem Fall liefert eine LP-Rundung kein brauchbares Ergebnis.

Um ein kleines Integrality-Gap zu erhalten, entwickeln wir eine alternative ILP-Formulierung.

Wir nehmen an, ein Orakel verrät uns den optimalen Makespan T . Das Orakel können wir, wie schon im Fall der identischen Maschinen gesehen, durch eine Binärsuche in Polynomialzeit simulieren. Die alternative ILP-Formulierung mit vorgegebenem Makespan T bezeichnen wir mit $ILP(T)$.

Wenn der Makespan bekannt ist, haben wir die folgende Zusatzinformation, die wir in unser ILP einfließen lassen: Ein Job i kann nur dann auf einer Maschine j platziert werden, falls gilt $p_{ij} < T$, denn sonst würde die Laufzeit dieses einzelnen Jobs bereits den vorgegebenen Makespan überschreiten.

Alternative ILP-Formulierung – ILP(T)

Definiere $S_T := \{(i, j) \in [n] \times [m] \mid p_{ij} \leq T\}$.

ILP(T) hat die Variablen x_{ij} nur für Paare $(i, j) \in S_T$.

Eine Zuteilung von Job i auf Maschine j für $(i, j) \notin S_T$ ist damit, wie erwünscht, nicht möglich.

Der Lösungsraum von ILP(T) wird beschrieben durch die Nebenbedingungen:

$$\begin{aligned} \forall i \in [n] : \quad & \sum_{j:(i,j) \in S_T} x_{ij} \geq 1 \\ \forall j \in [m] : \quad & \sum_{i:(i,j) \in S_T} x_{ij} p_{ij} \leq T \\ \forall (i, j) \in S_T \quad & x_{ij} \geq 0 \end{aligned}$$

Wir spezifizieren keine Zielfunktion. Es ist ausreichend eine beliebige zulässige Lösung zu berechnen, weil jede zulässige Lösung den Makespan (höchstens) T hat.

Relaxierung und Berechnung einer LP-Lösung

- Wir lassen die Ganzzahligkeitsbedingung fallen und erhalten aus ILP(T) das lineare Programm LP(T).
- Mit der Ellipsoidmethode berechnen wir eine zulässige Lösung für LP(T) in Polynomialzeit.

Diskussion verschiedener LP-Rundungsverfahren

Die nicht-ganzzahlige Lösung von $LP(T)$ kann durch randomisiertes Runden in eine ganzzahlige Lösung transformiert werden. Dabei geht man analog zum randomisierten Runden für das Routingproblem vor. Man zeigt, dass mit hoher Wahrscheinlichkeit auf jeder Maschine die zugewiesene Last nur um den Faktor $O(\log m)$ vom Wert der LP-Lösung abweicht. Daraus ergibt sich ein logarithmischer Approximationsfaktor. Eine etwas genauere Rechnung zeigt, dass randomisiertes Runden den Approximationsfaktor $\Theta\left(\frac{\log m}{\log \log m}\right)$ liefert. Unser Ziel in diesem Kapitel ist es, einen wesentlich besseren Faktor zu erreichen.

Wir werden im Folgenden ein deterministisches LP-Rundungsverfahren mit Approximationsfaktor 2 beschreiben. Unsere Analyse beruht auf interessanten, kombinatorischen Eigenschaften des Lösungspolyhedrons des linearen Programms $LP(T)$. Dazu müssen wir unser Wissen über lineare Programme ein wenig auffrischen und ergänzen.

Exkursion: Eigenschaften von Basislösungen

Wie die Simplexmethode berechnet auch die Ellipsoidmethode eine sogenannte „Basislösung“, die einem Knoten des Lösungspolyhedrons entspricht. Wir gehen von einem zulässigen und beschränkten LP mit endlicher Lösung aus. Bezeichne D die Dimension des LPs, also die Anzahl der Variablen, und $C \geq D$ die Anzahl der Nebenbedingungen (Constraints). Wir erinnern uns:

Eine Basislösung des LPs entspricht einem Knoten des Lösungspolyhedrons, also dem Schnittpunkt von D linear unabhängigen Nebenbedingungen bzw. der zugehörigen Hyperebenen im \mathbb{R}^D .

Somit ist eine Basislösung also in mindestens D der zu den Nebenbedingungen gehörenden Hyperebenen enthalten. Es folgt, dass in einer Basislösung mindestens D der C Nebenbedingungen des LPs exakt (also mit Gleichheit) erfüllt sind. Entsprechend sind höchstens $C - D$ der Nebenbedingungen nicht exakt erfüllt (d.h. diese Bedingungen sind übererfüllt).

Basislösungen werden häufig auch als Extrempunktlösungen bezeichnet. Dies hat seine Ursache in der geometrischen Interpretation des Lösungsraums als Polyhedron.

Ein Punkt x eines Polyhedrons \mathcal{P} ist genau dann ein Extrempunkt oder auch Eckpunkt von \mathcal{P} , wenn es *keine* Punkte $x_1, x_2 \in \mathcal{P} \setminus \{x\}$ mit der Eigenschaft gibt, dass x auf der Verbindungslinie zwischen x_1 und x_2 liegt.

Wir erinnern uns, die Verbindungslinie zwischen zwei Punkten x_1 und x_2 besteht aus allen Linearkombinationen $\lambda x_1 + (1 - \lambda)x_2$, $\lambda \in [0, 1]$.

Aus der Konvexität des Polyhedrons \mathcal{P} folgt, dass Extrempunkte nichts anderes sind als die Knoten von \mathcal{P} . Somit sind Basislösungen also Extrempunkte, und wir erhalten die folgende alternative Charakterisierung von Basislösungen.

Eine zulässige Lösung eines LPs ist genau dann eine Basis- oder Extrempunktlösung, wenn sie sich *nicht* als Linearkombination aus anderen zulässigen Lösungen bilden lässt.

Lemma 20 In einer Basislösung für $LP(T)$ haben alle bis auf höchstens $n + m$ der Variablen den Wert 0 haben.

Beweis:

- $LP(T)$ hat $D \leq nm$ Variablen und $C = D + n + m$ Nebenbedingungen.
- In der berechneten Basislösung sind höchstens $C - D = n + m$ viele Nebenbedingungen nicht exakt bzw. nicht mit Gleichheit erfüllt.
- Somit sind auch höchstens $n + m$ der Nichtnegativitätsbedingungen (also der Bedingungen $x_{ij} \geq 0$) nicht mit Gleichheit erfüllt.
- Es folgt, alle bis auf höchstens $n + m$ Variablen haben den Wert 0.

□

Der Allokationsgraph G

Zu einer Basislösung x von $LP(T)$ definieren den Allokationsgraphen $G = ([n] \cup [m], E)$. G ist ein bipartiter Graph, dessen Knoten den Jobs und Maschinen entsprechen. Job $i \in [n]$ ist mit Maschine $j \in [m]$ genau dann durch eine Kante verbunden, wenn $x_{ij} > 0$.

Ein Pseudobaum mit Knotenmenge V ist ein zusammenhängender Graph mit höchstens $|V|$ Kanten. Jeder Spannbaum enthält $|V| - 1$ Kanten. Ein Pseudobaum ist also entweder ein Spannbaum oder ein Spannbaum mit Zusatzkante, d.h. ein Graph, der höchstens *einen* Kreis enthält.

Lemma 21 Falls G zusammenhängend ist, so ist G ein Pseudobaum.

Beweis: Gemäß Annahme ist G ein zusammenhängender Graph mit $n + m$ Knoten. Aus Lemma 20 folgt, G hat höchstens $n + m$ Kanten. Damit ist G ein Pseudobaum. \square

Ein Graph ist ein Pseudowald, wenn jede Zusammenhangskomponente jeweils einem Pseudobaum entspricht.

Lemma 22 Der Allokationsgraph G ist ein Pseudowald.

Beweis: Betrachte eine beliebige Zusammenhangskomponente $G' = (V', E')$. Die Knotenmenge V' besteht aus Teilmengen der Jobs und Maschinen, d.h. $V' = J' \cup M'$ mit $J' \subseteq [n]$ und $M' \subseteq [m]$. Wir müssen zeigen, dass G' ein Pseudobaum ist.

Das Tupel (J', M') definiert ein eingeschränktes Schedulingproblem, bei dem die Jobs aus J' auf die Maschinen in M' verteilt werden sollen. Sei $LP'(T)$ die Relaxierung zum Schedulingproblem (J', M') mit vorgegebenem Makespan T . Wenn wir die Basislösung x für $LP(T)$ auf die Variablen x_{ij} mit $i \in J'$ und $j \in M'$ einschränken (d.h. alle anderen Variablen streichen), dann erhalten wir eine zulässige Lösung x' für $LP'(T)$.

Behauptung: x' ist eine Basislösung für $LP'(T)$.

Widerspruchsbeweis: Wir nehmen an x' ist keine Basislösung. Dann ist x' kein Extrempunkt des Lösungspolyhedrons von $LP'(T)$. Also gibt es zwei andere Lösungen x'_1 und x'_2 für $LP'(T)$, so dass gilt

$$x' = \lambda x'_1 + (1 - \lambda)x'_2$$

für ein geeignetes $\lambda \in [0, 1]$. Wir ergänzen x'_1 und x'_2 um die fehlenden Variablen aus x und erhalten die Lösungen $x_1 \neq x$ und $x_2 \neq x$ für $LP(T)$. Auf diese Art gilt

$$x = \lambda x_1 + (1 - \lambda)x_2 .$$

Dies ist aber ein Widerspruch zur Eigenschaft, dass x eine Extrempunktlösung für $LP(T)$ darstellt. Also ist x' auch eine Basislösung.

Damit ist G' also ein zusammenhängender Allokationsgraph zur Basislösung x' von $LP'(T)$. Jetzt folgt aus Lemma 21, dass G' ein Pseudobaum ist. Da diese Eigenschaft für jede Zusammenhangskomponente von G gilt, ist G somit ein Pseudowald.

□

LP-Rundung mit Hilfe des Allokationsgraphen

- **Ungeteilte Jobs:** Falls die Basislösung x für einen Job $i \in [n]$ eine ganzzahlige Zuteilung berechnet hat, d.h. es gibt eine Maschine $j \in [n]$ mit $x_{ij} = 1$, so wird diese Zuteilung direkt übernommen. Wir entfernen die entsprechenden Jobknoten und die inzidente Kanten aus dem Graphen G und erhalten dadurch einen Graphen, den wir H nennen.
- **Geteilte Jobs:** Wir berechnen ein einseitig perfektes Matching M für H , d.h. eine Teilmenge der Kanten, so dass jeder Jobknoten zu genau einer Kante in M inzident ist und jeder Maschinenknoten zu höchstens einer Kante. M ordnet jedem geteilten Job i also genau eine Maschine j zu, und wir setzen $x_{ij} = 1$ und $x_{ij'} = 0$ für $j' \neq j$. Beachte, jede Maschine erhält bei diesem Rundungsschritt höchstens einen zusätzlichen Job.

Wir müssen noch die Existenz des einseitig perfekten Matchings M auf H nachweisen und zeigen, wie dieses Matching effizient berechnet werden kann. Wir beobachten, dass der Graph H ein bipartiter Pseudowald ist, in dem alle Blätter Maschinenknoten sind, weil alle Jobknoten mit Grad 1 durch Streichung der ungeteilten Jobs entfernt wurden. Diese Eigenschaft werden wir ausnutzen.

Lemma 23 Der Allokationsgraph H hat ein einseitig perfektes Matching, und dieses Matching kann in Polynomialzeit berechnet werden.

Beweis: Wir beschreiben einen einfachen Algorithmus, der das Matching berechnet. Wir nutzen aus, dass alle Blätter im Pseudowald H Maschinenknoten sind. Zunächst entfernen wir alle isolierten Maschinenknoten, d.h. alle Zusammenhangskomponenten, die nur aus einem Maschinenknoten bestehen. Den folgenden Schritt wenden wir solange an, bis kein Blatt mehr verfügbar ist.

Wähle ein beliebiges Blatt $j \in [m]$, und füge die inzidente Kante $\{j, i\}$ ($i \in [n]$) zu M hinzu. Dann entferne die Knoten j und i mit allen inzidenten Kanten aus H und lösche die dadurch möglicherweise neu entstandenen isolierten Maschinenknoten.

Da der Graph H ein bipartiter Pseudowald ist, verbleiben nach dem iterierten Entfernen aller Blätter nur ein paar Kreise gerader Länge. Von diesen Kreisen nehmen wir jede zweite Kante zum Matching M hinzu. Auf diese Art wird jeder Jobknoten durch genau eine Kante aus M abgedeckt, und wir haben ein einseitig perfektes Matching konstruiert.

□

Satz 24 Das Makespan-Scheduling-Problem für allgemeine Maschinen hat einen Polynomialzeitalgorithmus mit Approximationsfaktor 2.

Beweis: Wir müssen uns nur noch um den Nachweis des Approximationsfaktors kümmern.

- Auf jeder Maschine verursachen die durch die Basislösung ungeteilt zugewiesenen Jobs eine Last von höchstens T .
- Beim Runden der geteilten Jobs wird jeder Maschine maximal ein Job zugeordnet, weil wir ein Matching verwenden. Eine Matchingkante $\{i, j\}$ existiert nur dann, wenn in der Basislösung $x_{ij} > 0$ gilt. Notwendige Bedingung dafür ist aber, dass $(i, j) \in S_T$ ist. Aus $(i, j) \in S_T$ folgt aber $p_{ij} \leq T$. Deshalb erzeugt die Zuteilung entlang der Matchingkanten einen Lastzuwachs von höchstens T je Maschine.

Somit ist der berechnete Makespan höchstens zweimal so groß wie der optimale Makespan T . □