Skript zur Vorlesung

Algorithmen und Berechnungskomplexität III

Prof. Dr. Heiko Röglin
Institut für Informatik I

universitätbonn

Sommersemester 2011

13. Juli 2011

Vorwort

Das vorliegende Skript ist als Begleitmaterial für die Vorlesung „Algorithmen und Berechnungskomplexität III“ im Sommersemester 2011 an der Universität Bonn konzipiert. Nachdem wir uns in diesem und dem letzten Semester bereits mit einfachen Datenstrukturen, grundlegenden Algorithmen und den Grenzen der (effizienten) Berechenbarkeit beschäftigt haben, werden wir in diesem letzten Teil der Vorlesung wichtige Techniken zum Entwurf von Algorithmen kennenlernen. Wir werden uns insbesondere mit Greedy-Algorithmen, dynamischer Programmierung und linearer Programmierung beschäftigen.

Wir werden diese Techniken unter anderem zum Entwurf von Approximationsalgorithmen für NP-harte Optimierungsprobleme einsetzen. Dabei handelt es sich um Algorithmen, die nicht unbedingt optimale Lösungen berechnen, aber Lösungen, die nicht beliebig weit vom Optimum entfernt sind. Wir werden uns auch mit der Frage beschäftigen, wie gut optimale Lösungen für NP-harte Probleme überhaupt approximiert werden können.

Bisher sind wir bei algorithmischen Problemen davon ausgegangen, dass die konkrete Eingabe dem Algorithmus von Anfang an komplett bekannt ist. In manchen Anwendungen ist das aber nicht der Fall und die Eingabe wird erst Schritt für Schritt aufgedeckt. Wir werden uns auch mit solchen sogenannten Online-Problemen beschäftigen.

Zum Schluss lernen wir mit linearer Programmierung ein sehr allgemeines Konzept zur Lösung von Optimierungsproblemen kennen.

Für Hinweise auf Fehler im Skript und Verbesserungsvorschläge bin ich dankbar. Bitte senden Sie diese an die E-Mail-Adresse `roeglin@cs.uni-bonn.de` oder sprechen Sie mich in der Vorlesung an.

Heiko Röglin

Inhaltsverzeichnis

Kapitel 1

Approximationsalgorithmen

Wenn wir für ein Optimierungsproblem gezeigt haben, dass es NP-hart ist, dann bedeutet das zunächst nur, dass es unter der Annahme P≠NP keinen effizienten Algorithmus gibt, der für jede Instanz die optimale Lösung berechnet. Das schließt nicht aus, dass es einen effizienten Algorithmus gibt, der für jede Instanz eine Lösung berechnet, die fast optimal ist. Um das formal zu fassen, betrachten wir zunächst genauer, was ein *Optimierungsproblem* eigentlich ist, und definieren anschließend die Begriffe *Approximationsalgorithmus* und *Approximationsgüte*.

Ein *Optimierungsproblem* Π besteht aus einer Menge $\mathcal{I}_\Pi$ von *Instanzen* oder *Eingaben*. Zu jeder Instanz $I \in \mathcal{I}_\Pi$ gehört eine Menge $\mathcal{S}_I$ von *Lösungen* und eine *Zielfunktion* $f_I : \mathcal{S}_I \to \mathbb{R}_{\geq 0}$, die jeder Lösung einen reellen Wert zuweist. Zusätzlich ist bei einem Optimierungsproblem vorgegeben, ob wir eine Lösung x mit *minimalem* oder mit *maximalem Wert* $f_I(x)$ suchen. Wir bezeichnen in jedem Falle mit $\mathrm{OPT}(I)$ den Wert einer optimalen Lösung. Wenn die betrachtete Instanz I klar aus dem Kontext hervorgeht, so schreiben wir oft einfach OPT statt $\mathrm{OPT}(I)$.

Beispiel: Spannbaumproblem

Eine Instanz I des *Spannbaumproblems* wird durch einen ungerichteten Graphen $G = (V, E)$ und Kantengewichte $c : E \to \mathbb{N}$ beschrieben. Die Menge $\mathcal{S}_I$ der Lösungen für eine solche Instanz ist die Menge aller Spannbäume des Graphen G. Die Funktion f_I weist jedem Spannbaum $T \in \mathcal{S}_I$ sein Gewicht zu, also $f_I(T) = \sum_{e \in T} c(e)$, und wir möchten f_I minimieren. Demzufolge gilt $\mathrm{OPT}(I) = \min_{T \in \mathcal{S}_I} f_I(T)$.

Ein *Approximationsalgorithmus* A für ein Optimierungsproblem Π ist zunächst lediglich ein Polynomialzeitalgorithmus, der zu jeder Instanz I eine Lösung aus $\mathcal{S}_I$ ausgibt. Wir bezeichnen mit $A(I)$ die Lösung, die Algorithmus A bei Eingabe I ausgibt, und wir bezeichnen mit $w_A(I)$ ihren Wert, also $w_A(I) = f_I(A(I))$. Je näher der Wert $w_A(I)$ dem optimalen Wert $\mathrm{OPT}(I)$ ist, desto besser ist der Approximationsalgorithmus.

Definition 1.1 (Approximationsfaktor/Approximationsgüte)**.** *Ein Approximationsalgorithmus A für ein Minimierungs- bzw. Maximierungsproblem Π erreicht einen* Approximationsfaktor *oder eine* Approximationsgüte *von $r \geq 1$ bzw. $r \leq 1$, wenn*

$$w_A(I) \leq r \cdot \mathrm{OPT}(I) \quad \textit{bzw.} \quad w_A(I) \geq r \cdot \mathrm{OPT}(I)$$

*für alle Instanzen $I \in \mathcal{I}_\Pi$ gilt. Wir sagen dann, dass A ein r-*Approximationsalgorithmus *ist.*

Haben wir beispielsweise für ein Minimierungsproblem einen 2-Approximationsalgorithmus, so bedeutet das, dass der Algorithmus für jede Instanz eine Lösung berechnet, deren Wert höchstens doppelt so groß ist wie der optimale Wert. Haben wir einen $\frac{1}{2}$-Approximationsalgorithmus für ein Maximierungsproblem, so berechnet der Algorithmus für jede Instanz eine Lösung, deren Wert mindestens halb so groß ist wie der optimale Wert.

Ein Polynomialzeitalgorithmus für ein Problem, der stets eine optimale Lösung berechnet, ist ein 1-Approximationsalgorithmus. Ist ein Problem NP-hart, so schließt dies unter der Voraussetzung P≠NP lediglich die Existenz eines solchen 1-Approximationsalgorithmus aus. Es gibt aber durchaus NP-harte Probleme, für die es zum Beispiel 1.01-Approximationsalgorithmen gibt, also effiziente Algorithmen, die stets eine Lösung berechnen, deren Wert um höchstens ein Prozent vom optimalen Wert abweicht.

Die in diesem Skript vorgestellten Approximationsalgorithmen können auch im Buch von Vijay Vazirani [5] und in den Skripten von Berthold Vöcking [6, 9] nachgelesen werden.

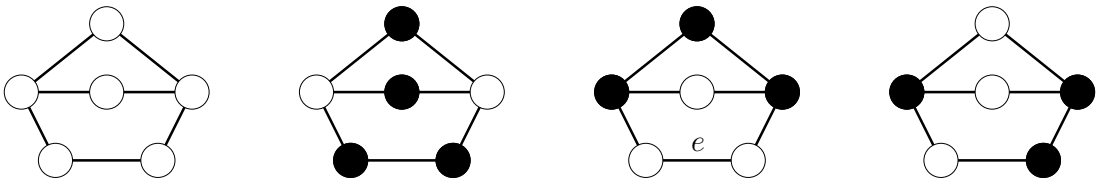
1.1 Vertex Cover und Set Cover

Wir betrachten zunächst die beiden NP-harten Probleme *Vertex Cover* und *Set Cover* und werden für diese Probleme einfache Approximationsalgorithmen entwerfen. Bei beiden Algorithmen handelt es sich um sogenannte *Greedy-Algorithmen*, also Algorithmen, die zu jedem Zeitpunkt eine möglichst gute ad hoc Entscheidung treffen, ohne die Konsequenzen für die weiteren Entscheidungen zu berücksichtigen.

1.1.1 Vertex Cover

Vertex Cover

Eingabe:	ungerichteter Graph $G = (V, E)$
Lösungen:	alle $V' \subseteq V$ mit $\forall e = (u, v) \in E : u \in V'$ oder $v \in V'$ (oder beides) ein V' mit dieser Eigenschaft nennen wir ein *Vertex Cover von G*
Zielfunktion:	minimiere $\lvert V' \rvert$



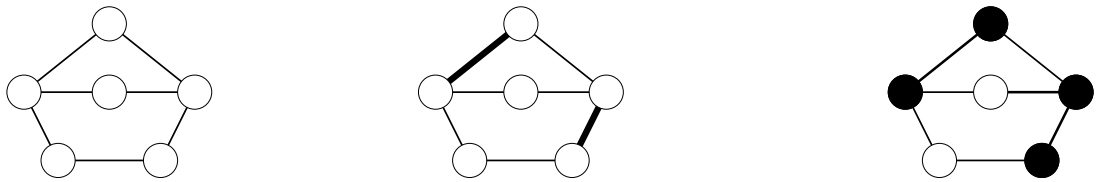
(a) $G = (V, E)$ (b) schwarze Knoten bilden ein Vertex Cover (c) kein Vertex Cover, da e nicht abgedeckt (d) minimales Vertex Cover der Größe 3

Abbildung 1.1: Beispiel für das Vertex Cover Problem

Wir wissen bereits aus dem zweiten Teil der Vorlesung, dass das Problem Vertex Cover NP-hart ist. Wir präsentieren nun einen 2-Approximationsalgorithmus. Der Algorithmus benutzt den Begriff eines *Matchings*. Ein Matching $M \subseteq E$ eines Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, sodass es keinen Knoten gibt, der zu mehr als einer Kante in M inzident ist. Ein Matching $M \subseteq E$ heißt *inklusions-maximal*, wenn für alle $e \in E \setminus M$ gilt, dass $M \cup \{e\}$ kein Matching ist.

Algorithmus: Approx-VC

1. Berechne ein inklusions-maximales Matching $M \subseteq E$.
   Setze dazu zunächst $M = \emptyset$ und gehe dann die Kantenmenge einmal durch. Füge dabei Kante $e = (u, v) \in E$ zur Menge M hinzu, wenn weder u noch v bereits inzident zu einer Kante in M sind.
2. Gib $V(M)$, die Menge aller Knoten, die zu einer Kante $e \in M$ inzident sind, aus.



(a) $G = (V, E)$ (b) inklusions-maximales Matching $M \subseteq E$ (c) Vertex Cover

Abbildung 1.2: Beispiel für den Algorithmus Approx-VC

Theorem 1.2 (Approx-VC)**.** *Der Algorithmus* Approx-VC *ist ein 2-Approximationsalgorithmus für Vertex Cover. Seine Laufzeit beträgt $O(|V|+|E|)$ für Graphen $G = (V, E)$, die in Adjazenzlistendarstellung gegeben sind.*

Beweis. Wir betrachten zunächst die Laufzeit. Für Schritt 1 müssen wir nur in einem Array speichern, welche Knoten bereits inzidente Kanten in M haben, und dann die Liste der Kanten einmal durchlaufen. Die Laufzeit für Schritt 1 ist somit $O(|V| + |E|)$. Mit Hilfe des Arrays kann die Menge $V(M)$ direkt ausgegeben werden. Die Laufzeit für Schritt 2 ist also $O(|V|)$.

Als nächstes zeigen wir, dass der Algorithmus korrekt in dem Sinne ist, dass er stets ein Vertex Cover ausgibt. Nehmen wir an, dass $V(M)$ nicht alle Kanten aus E abdeckt. Sei dann $e = (u, v) \in E$ eine beliebige nicht abgedeckte Kante. Da e von $V(M)$ nicht abgedeckt wird, gilt weder $u \in V(M)$ noch $v \in V(M)$. Damit ist aber auch $M \cup \{e\}$ ein Matching im Widerspruch zu der Konstruktion von M als inklusions-maximales Matching. Somit liefert der Algorithmus stets ein Vertex Cover.

Als letztes betrachten wir den Approximationsfaktor. Wir müssen zeigen, dass $V(M)$ für jeden Graphen $G = (V, E)$ höchstens doppelt so viele Knoten enthält wie ein minimales Vertex Cover V^*. Wir wissen, dass V^* insbesondere jede Kante $e \in M$ abdecken muss. Das bedeutet, für jede solche Kante muss mindestens einer der beiden Endknoten in V^* sein. Da die Kanten in M ein Matching bilden und keine gemeinsamen Endknoten besitzen, kann kein Knoten mehr als eine Kante aus M abdecken. Zusammen bedeutet das, dass $\mathrm{OPT}(G) = |V^*| \geq |M|$ gelten muss. Für unsere Lösung $V(M)$ gilt $|V(M)| = 2|M|$. Somit folgt insgesamt

$$\frac{|V(M)|}{\mathrm{OPT}(G)} \leq \frac{2|M|}{|M|} = 2. \qquad \square$$

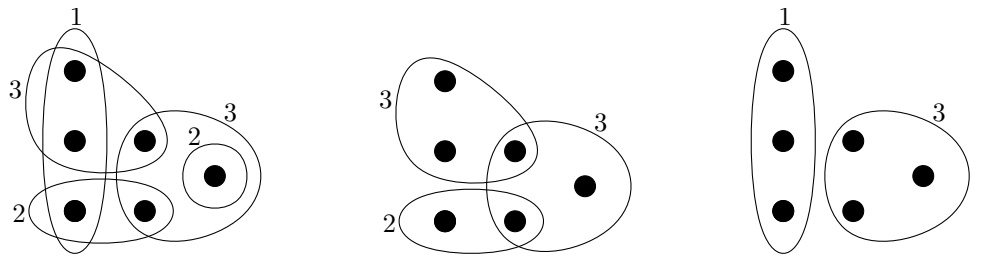
Der obige Beweis war nicht schwer. Dennoch sollten wir uns etwas Zeit nehmen, die wesentlichen Komponenten noch einmal anzuschauen. Um zu beweisen, dass ein Algorithmus eine bestimmte Approximationsgüte bei einem Minimierungsproblem erreicht, sind zwei Dinge notwendig: Erstens müssen wir eine obere Schranke für den Wert der Lösung angeben, die der Algorithmus berechnet (in obigem Beweis $2|M|$). Zweitens müssen wir eine untere

Schranke für den Wert der optimalen Lösung angeben (in obigem Beweis $|M|$). Der Quotient dieser Schranken ist dann eine Abschätzung für die Approximationsgüte. Bei Maximierungsproblemen ist es genau andersherum: man braucht eine untere Schranke für den Wert der vom Algorithmus berechneten Lösung und eine obere Schranke für den Wert der optimalen Lösung. Oftmals liegt die Schwierigkeit bei der Analyse von Approximationsalgorithmen genau darin, eine nützliche Schranke für den Wert der optimalen Lösung zu finden.

1.1.2 Set Cover

Set Cover

Eingabe:	Grundmenge S mit n Elementen m Teilmengen $S_1, \ldots, S_m \subseteq S$ mit $\bigcup_{i=1}^{m} S_i = S$ Kosten $c_1, \ldots, c_m \in \mathbb{N}$
Lösungen:	alle Teilmengen $A \subseteq \{1, \ldots, m\}$ mit $\bigcup_{i \in A} S_i = S$ ein A mit dieser Eigenschaft nennen wir ein *Set Cover von* S
Zielfunktion:	minimiere $c(A) = \sum_{i \in A} c_i$



(a) Instanz (b) Set Cover mit Wert 8 (c) Set Cover mit Wert 4

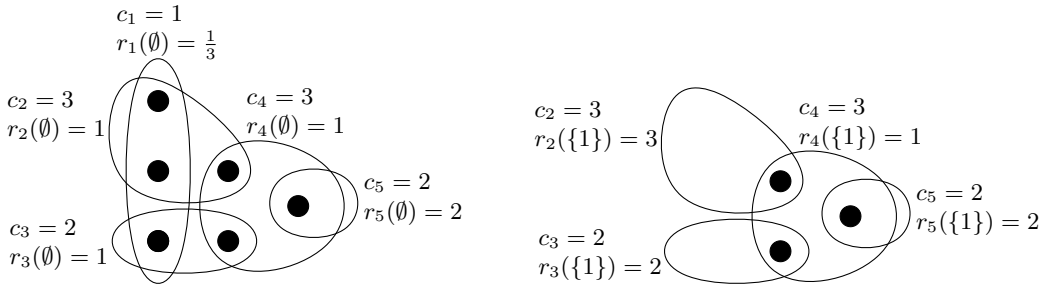
Abbildung 1.3: Beispiel für das Problem Set Cover

Das Problem Vertex Cover aus dem letzten Abschnitt kann als Spezialfall von Set Cover aufgefasst werden, indem man die Kanten des Graphen als Grundmenge auffasst und die Knoten als die Mengen, die jeweils ihre inzidenten Kanten abdecken. Vertex Cover ist somit der Spezialfall, bei dem jedes Element in genau zwei Mengen vorkommt. Da bereits dieser Spezialfall NP-hart ist, trifft dies auch auf das allgemeinere Problem Set Cover zu. Wir betrachten nun den folgenden Approximationsalgorithmus für Set Cover.

Algorithmus: Greedy-SC

1. $A := \emptyset$; $C := \emptyset$;
2. **while** $C \neq S$ **do**
3. Wähle eine Menge S_i, für die die relativen Kosten $r_i(A) = \frac{c_i}{|S_i \setminus C|}$ minimal sind.
4. Setze price$(x) := \frac{c_i}{|S_i \setminus C|}$ für alle $x \in S_i \setminus C$.
5. $A := A \cup \{i\}$; $C := C \cup S_i$;
6. Ausgabe: A

Theorem 1.3 (Greedy-SC)**.** *Der Algorithmus* Greedy-SC *ist ein H_n-Approximationsalgorithmus für Set Cover, wobei $n = |V|$ und $H_n = \sum_{i=1}^{n} \frac{1}{i}$ die n-te harmonische Zahl ist.*



(a) S_1 hat minimale relative Kosten (b) S_4 hat minimale relative Kosten

Abbildung 1.4: Beispiel für den Algorithmus Greedy-SC

An dieser Stelle sollte der aufmerksame Leser festgestellt haben, dass das obige Theorem nicht ganz zu Definition 1.1 passt. In dieser Definition sind wir nämlich von einer konstanten Approximationsgüte r ausgegangen. Nun hängt die Güte, die wir erreichen, aber von der Eingabelänge ab. Deshalb lassen wir im Allgemeinen auch zu, dass $r : \mathbb{N} \to [0, \infty)$ eine Funktion ist. Es muss dann bei einem Minimierungs- bzw. Maximierungsproblem

$$w_A(I) \leq r(|I|) \cdot \mathrm{OPT}(I) \quad \text{bzw.} \quad w_A(I) \geq r(|I|) \cdot \mathrm{OPT}(I)$$

für alle Instanzen I mit Länge $|I|$ gelten.

Beweis von Theorem 1.3. Wir müssen es schaffen, die Kosten $C(A)$ der vom Algorithmus berechneten Lösung A mit den Kosten der optimalen Lösung zu vergleichen. Dazu ist es hilfreich, die Kosten $c(A)$ auf die Elemente der Grundmenge S zu verteilen. Genau aus diesem Grund haben wir im Algorithmus die Funktion price : $S \to \mathbb{R}$ definiert, die für die reine Beschreibung gar nicht benötigt wird, sondern uns lediglich die Analyse erleichtert. Diese Funktion ist so gewählt, dass wir die Kosten $c(A)$ der Lösung A als

$$c(A) = \sum_{x \in S} \mathrm{price}(x)$$

schreiben können. Was können wir nun über die Kosten der optimalen Lösung aussagen? Wir ordnen die Elemente von S in der Reihenfolge, in der sie zu C hinzugefügt werden. Werden in einer Iteration mehrere Elemente gleichzeitig zu C hinzugefügt, so können wir diese in einer beliebigen Reihenfolge anordnen. Sei $x_1, \ldots, x_n$ die Reihenfolge, die sich ergibt.

Lemma 1.4. *Für jedes $k \in \{1, \ldots, n\}$ gilt* $\mathrm{price}(x_k) \leq \mathrm{OPT}/(n - k + 1)$.

Beweis. Wir betrachten ein beliebiges $k \in \{1, \ldots, n\}$ und bezeichnen mit $i \in \{1, \ldots, m\}$ den Index der Menge S_i, durch deren Hinzunahme das Element x_k erstmalig abgedeckt wird. Bezeichne nun A die Auswahl unmittelbar bevor i zur Menge A hinzugefügt wird. Wir werden im Folgenden OPT und $\mathrm{price}(x_k)$ zueinander in Beziehung setzen.

Uns interessieren die Elemente, die noch nicht abgedeckt sind, also $\overline{C} = S \setminus C$ mit $C = \cup_{j \in A} S_j$. Direkt vor der Hinzunahme von Menge S_i gibt es davon noch mindestens $n - k + 1$, da höchstens die Elemente $x_1, \ldots, x_{k-1}$ abgedeckt sind. Wir betrachten jetzt die Instanz I' von Set Cover eingeschränkt auf die Elemente aus $\overline{C}$ (d. h. die bereits abgedeckten Elemente werden aus S und aus jeder Menge S_j entfernt). Sei dann A^* die optimale Lösung für die Instanz I' und seien $\mathrm{OPT}' = c(A^*)$ ihre Kosten. Wir können OPT' schreiben als

$$\mathrm{OPT}' = \sum_{j \in A^*} c_j = \sum_{j \in A^*} \left(|S_j \setminus C| \cdot \frac{c_j}{|S_j \setminus C|} \right) = \sum_{j \in A^*} \left(\sum_{x \in S_j \setminus C} r_j(A) \right).$$

Aus der Wahl von S_i als nächste Menge folgt, dass es in der aktuellen Situation keine Menge, gibt, deren relative Kosten kleiner als $r_i(A)$ sind. Damit folgt insgesamt

$$\begin{aligned}\mathrm{OPT} \geq \mathrm{OPT}' &= \sum_{j \in A^*} \left(\sum_{x \in S_j \setminus C} r_j(A) \right) \\ &\geq \sum_{j \in A^*} \left(\sum_{x \in S_j \setminus C} r_i(A) \right) \geq |S \setminus C| \cdot r_i(A) \\ &\geq (n - k + 1) \cdot r_i(A).\end{aligned}$$

Mit der Beobachtung $r_i(A) = \mathrm{price}(x_k)$ folgt nun das Lemma. □

Aus diesem Lemma folgt nun direkt der Beweis des Theorems, denn es gilt

$$\begin{aligned}c(A) &= \sum_{x \in S} \mathrm{price}(x) = \sum_{k=1}^{n} \mathrm{price}(x_k) \\ &\leq \sum_{k=1}^{n} \frac{\mathrm{OPT}}{n - k + 1} = \mathrm{OPT} \cdot \sum_{k=1}^{n} \frac{1}{k} = \mathrm{OPT} \cdot H_n.\end{aligned}$$

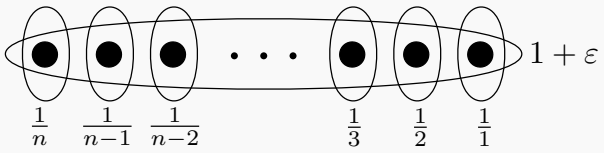
Da die Laufzeit des Algorithmus offensichtlich polynomiell ist, ist damit gezeigt, dass es sich um einen H_n-Approximationsalgorithmus handelt. □

Wir wissen bereits aus dem ersten Teil der Vorlesung, dass sich die harmonische Zahl asymptotisch wie der natürliche Logarithmus verhält. Für jedes $n \in \mathbb{N}$ gilt $\ln(n+1) \leq H_n \leq \ln(n) + 1$ und somit können wir auch sagen, dass der Algorithmus Greedy-SC einen Approximationsfaktor von $O(\log(n))$ liefert.

Als letztes stellen wir noch die Frage, ob wir den Algorithmus gut analysiert haben. Es könnte ja durchaus sein, dass er eigentlich eine 2-Approximation berechnet und unsere Analyse nicht genau genug war, um das zu zeigen. Dass dies nicht der Fall ist, zeigt das folgende Beispiel.

Untere Schranke für Greedy-SC

Die folgende Instanz von Set Cover besitzt eine optimale Lösung mit Kosten $1 + \varepsilon$, für ein beliebig kleines $\varepsilon > 0$, der Algorithmus Greedy-SC berechnet aber eine Lösung mit Kosten H_n. Damit ist gezeigt, dass sich die Analyse nicht wesentlich verbessern lässt und es Instanzen gibt, auf denen der Approximationsfaktor $\Omega(\log(n))$ ist.



1.1.3 Greedy-Algorithmen

Die beiden Approximationsalgorithmen, die wir in diesem Kapitel kennengelernt haben, sind sogenannte *Greedy-Algorithmen*. Dabei handelt es sich um Algorithmen, die zu jedem Zeitpunkt eine möglichst gute ad hoc Entscheidung treffen, ohne die Konsequenzen für die weiteren Entscheidungen zu berücksichtigen. Der Algorithmus Approx-VC berechnet beispielsweise ein inklusions-maximales Matching, indem er die Kanten in einer beliebigen Reihenfolge durchgeht und jede erlaubte Kante direkt einfügt. Der Algorithmus Greedy-SC wählt stets eine Menge mit den momentan kleinsten relativen Kosten. Wie wir gesehen haben, führt dies nicht unbedingt dazu, dass wir am Ende eine optimale Lösung erhalten, dennoch konnten wir den Approximationsfaktor beider Algorithmen beschränken. Tatsächlich ist es in beiden Fällen sogar so, dass die vorgestellten Greedy-Algorithmen die bisher besten bekannten Approximationsalgorithmen sind. Unter bestimmten Annahmen der Komplexitätstheorie (diese Annahmen sind etwas stärker als P≠NP) kann man sogar beweisen, dass es keine besseren Algorithmen gibt.

Für viele Probleme liefern Greedy-Algorithmen sogar optimale Lösungen. Ein Beispiel, das dem Leser vertraut sein sollte, ist der Algorithmus von Kruskal zur Berechnung eines minimalen Spannbaumes. Dieser fügt stets die günstigste noch erlaubte Kante ein. Auch den Algorithmus von Dijkstra kann man als eine Art Greedy-Algorithmus auffassen, weil er die Menge der bereits besuchten Knoten stets um den Knoten erweitert, dessen Distanz zur Quelle von allen noch nicht besuchten Knoten am geringsten ist.

1.2 Rucksackproblem

Das Rucksackproblem haben wir bereits im zweiten Teil der Vorlesung als NP-hartes Optimierungsproblem kennengelernt. Der Vollständigkeit halber definieren wir es hier noch einmal.

Rucksackproblem (Knapsack Problem, KP)

Eingabe:	Kapazität $b \in \mathbb{N}$
	Nutzen $p_1, \ldots, p_n \in \mathbb{N}$
	Gewichte $w_1, \ldots, w_n \in \{1, \ldots, b\}$
Lösungen:	alle Teilmengen $A \subseteq \{1, \ldots, n\}$ mit $w(A) = \sum_{i \in A} w_i \leq b$
Zielfunktion:	maximiere $p(A) = \sum_{i \in A} p_i$

1.2.1 Greedy-Algorithmus

Wir entwerfen einen Greedy-Algorithmus, der eine $\frac{1}{2}$-Approximation für das Rucksackproblem liefert. Dieser Algorithmus sortiert die Objekte zunächst absteigend gemäß ihrer Effizienz (dem Quotienten aus Nutzen und Gewicht) und packt dann greedy die effizientesten Objekte ein, bis der Rucksack voll ist.

Algorithmus: GREEDY-KP

1. Sortiere die Objekte gemäß ihrer Effizienz. Danach gelte
$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}.$$
2. $B := \emptyset$;
3. **for** $i = 1$ **to** n **do**
4. **if** $w(B) + w_i \leq b$ **then**
5. $B := B \cup \{i\}$;
6. **else**
7. break;
8. Sei i^* ein Objekt mit dem größten Nutzen, d.h. $p_{i^*} \geq p_i$ für alle $i \in \{1, \ldots, n\}$.
9. **if** $p(B) < p_{i^*}$ **then** $A := \{i^*\}$ **else** $A := B$;
10. Ausgabe: A

Theorem 1.5 (GREEDY-KP)**.** *Der Algorithmus* GREEDY-KP *ist ein* $\frac{1}{2}$*-Approximationsalgorithmus für das Rucksackproblem.*

Beweis. Offensichtlich hat der Algorithmus polynomielle Laufzeit. Um seine Approximationsgüte abzuschätzen, benötigen wir eine obere Schranke für den Wert OPT der optimalen Lösung. Sei $B = \{1, \ldots, i\}$ die im Algorithmus berechnete Menge. Gilt $i = n$, so passen alle Objekte in den Rucksack und die Lösung B ist trivialerweise optimal. Ansonsten ist $i < n$ und die Objekte $\{1, \ldots, i+1\}$ haben zusammen ein Gesamtgewicht echt größer als b. Sei dann $r = p_{i+1}/w_{i+1}$. Wir behaupten, dass

$$\text{OPT} \leq p(B) + p_{i+1}$$

gilt. Nehmen wir an, es sei nicht so und es gäbe eine Lösung A' mit $w(A') \leq b < w(B) + w_{i+1}$ und $p(A') > p(B) + p_{i+1}$. Für $J = \{1, \ldots, i+1\} \setminus A'$ und $J' = A' \setminus \{1, \ldots, i+1\}$ gilt

$$p(B) + p_{i+1} < p(A') = (p(B) + p_{i+1}) + p(J') - p(J)$$

und

$$w(B) + w_{i+1} > w(A') = (w(B) + w_{i+1}) + w(J') - w(J).$$

Dementsprechend muss

$$p(J) < p(J') \qquad \text{und} \qquad w(J) > w(J')$$

gelten. Weil das Objekt $i+1$ eine Effizienz von r hat und die Objekte gemäß ihrer Effizienz sortiert sind, gilt

$$p(J) \geq r \cdot w(J) \qquad \text{und} \qquad p(J') \leq r \cdot w(J').$$

Somit erhalten wir insgesamt

$$p(J) \geq r \cdot w(J) > r \cdot w(J') \geq p(J')$$

im Widerspruch zu obiger Feststellung $p(J) < p(J')$. Damit ist die Behauptung bewiesen und $p(B) + p_{i+1}$ ist wirklich eine obere Schranke für OPT.

Nun können wir den Nutzen der vom Algorithmus berechneten Lösung A und den Wert OPT der optimalen Lösung zueinander in Beziehung setzen. Im letzten Schritt des Algorithmus wird verglichen, ob die Menge B oder das einzelne Objekt i^* einen größeren Nutzen hat. Dementsprechend wird dann A definiert, wobei wir ausnutzen, dass jedes Objekt, also insbesondere i^*, ein Gewicht von höchstens b hat, und deshalb alleine in den Rucksack passt. Das bedeutet, der Nutzen von A ist gleich dem Maximum aus $p(B)$ und p_{i^*}. Nutzen wir dann noch aus, dass $p_{i+1} \leq p_{i^*}$ gilt, so erhalten wir

$$p(A) = \max\{p(B), p_{i^*}\} \geq \max\{p(B), p_{i+1}\} \geq \frac{1}{2}(p(B) + p_{i+1}) \geq \frac{\text{OPT}}{2},$$

womit das Theorem bewiesen ist. □

Es stellt sich nun die Frage, ob eine $\frac{1}{2}$-Approximation das Beste ist, was man für das Rucksackproblem in polynomieller Zeit erreichen kann. Wir werden im Folgenden zeigen, dass das nicht der Fall ist, und dass man das Rucksackproblem sogar beliebig gut approximieren kann.

1.2.2 Pseudopolynomieller Algorithmus

Eine weitere wichtige Technik zum Entwurf von Algorithmen ist *dynamische Programmierung*. Das bedeutet, dass wir ein gegebenes Problem in Teilprobleme zerlegen, diese lösen und aus den optimalen Lösungen der Teilprobleme eine optimale Lösung für das Gesamtproblem berechnen. Lösungen für Teilprobleme, die bereits berechnet sind, werden in einer Tabelle abgespeichert, damit sie nicht mehrfach berechnet werden müssen. Das unterscheidet dynamische Programmierung von Divide-and-Conquer. Dynamische Programmierung haben wir bereits beim Floyd-Warshall Algorithmus zur Berechnung kürzester Wege und beim CYK-Algorithmus für das Wortproblem kontextfreier Sprachen kennengelernt.

Es gibt viele Arten, eine gegebene Instanz des Rucksackproblems in kleinere Teilprobleme zu zerlegen. Zunächst ist es naheliegend Teilprobleme zu betrachten, die nur eine Teilmenge der Objekte enthalten. Das alleine reicht für unsere Zwecke aber noch nicht aus und wir gehen einen Schritt weiter.

Für eine gegebene Instanz mit Nutzen $p_1, \ldots, p_n$ definieren wir zunächst $P = \max_{i \in \{1,\ldots,n\}} p_i$. Dann ist nP offensichtlich eine obere Schranke für den Nutzen, den eine optimale Lösung erreichen kann. Wir definieren für jede Kombination aus $i \in \{1, \ldots, n\}$ und $p \in \{0, \ldots, nP\}$ folgendes Teilproblem: finde unter allen Teilmengen der Objekte $1, \ldots, i$ mit Gesamtnutzen mindestens p die mit dem kleinsten Gewicht. Es sei $W(i, p)$ das Gewicht dieser Teilmenge. Existiert keine solche Teilmenge, so setzen wir $W(i, p) = \infty$.

Mit anderen Worten können wir die Bedeutung von $W(i, p)$ auch wie folgt zusammenfassen: für jede Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{i \in I} p_i \geq p$ gilt $\sum_{i \in I} w_i \geq W(i, p)$. Ist $W(i, p) < \infty$, so gibt es eine Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{i \in I} p_i \geq p$ und $\sum_{i \in I} w_i = W(i, p)$.

Wir werden nun das Rucksackproblem lösen, indem wir eine Tabelle konstruieren, die für jede Kombination aus $i \in \{1, \ldots, n\}$ und $p \in \{0, \ldots, nP\}$ den Wert $W(i, p)$ enthält. Wir konstruieren die Tabelle Schritt für Schritt und fangen zunächst mit den einfachen Randfällen der Form $W(1, p)$ an. Wir stellen uns hier also die Frage, ob wir mit einer Teilmenge von $\{1\}$, also mit dem ersten Objekt, Nutzen mindestens p erreichen können, und wenn ja, mit welchem Gewicht. Ist $p > p_1$, so geht das nicht und wir setzen $W(1, p) = \infty$. Ist hingegen $1 \leq p \leq p_1$, so können wir mindestens Nutzen p erreichen, indem wir das erste Objekt wählen. Das Gewicht

ist dann entsprechend w_1. Dementsprechend setzen wir $W(1,p) = w_1$ für $1 \leq p \leq p_1$. Wir nutzen im Folgenden noch die Konvention $W(i,p) = 0$ für alle $i \in \{1,\ldots,n\}$ und $p \leq 0$.

Sei nun bereits für ein i und für alle $p \in \{0,\ldots,nP\}$ der Wert $W(i-1,p)$ bekannt. Wie können wir dann für $p \in \{0,\ldots,nP\}$ den Wert $W(i,p)$ aus den bekannten Werten berechnen? Gesucht ist eine Teilmenge $I \subseteq \{1,\ldots,i\}$ mit $\sum_{i\in I} p_i \geq p$ und kleinstmöglichen Gewicht. Wir unterscheiden zwei Fälle: entweder diese Teilmenge I enthält Objekt i oder nicht.

- Falls $i \notin I$, so ist $I \subseteq \{1,\ldots,i-1\}$ mit $\sum_{i\in I} p_i \geq p$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt I minimales Gewicht. Damit gilt $W(i,p) = W(i-1,p)$.
- Falls $i \in I$, so ist $I \setminus \{i\}$ eine Teilmenge von $\{1,\ldots,i-1\}$ mit Nutzen mindestens $p-p_i$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt $I\setminus\{i\}$ minimales Gewicht. Wäre das nicht so und gäbe es eine Teilmenge $I' \subseteq \{1,\ldots,i-1\}$ mit Nutzen mindestens $p-p_i$ und $W(I') < W(I)$, so wäre auch $p(I'\cup\{i\}) \geq p$ und $w(I'\cup\{i\}) < w(I)$ im Widerspruch zur Wahl von I. Es gilt also $w(I \setminus \{i\}) = W(i-1,p-p_i)$ und somit insgesamt $W(i,p) = W(i-1,p-p_i) + w_i$.

Wenn wir vor der Aufgabe stehen $W(i,p)$ zu berechnen, dann wissen wir a priori nicht, welcher der beiden Fälle eintritt. Wir testen deshalb einfach, welche der beiden Alternativen eine Menge I mit kleinerem Gewicht liefert, d. h. wir setzen $W(i,p) = \min\{W(i-1,p), W(i-1,p-p_i) + w_i\}$.

Ist $W(i,p)$ für alle Kombinationen von i und p bekannt, so können wir die Entscheidungsvariante des Rucksackproblems leicht lösen. Werden wir gefragt, ob es eine Teilmenge der Objekte mit Nutzen mindestens p gibt, deren Gewicht höchstens b ist, so testen wir einfach, ob $W(n,p) \leq b$ gilt.

Wir können nicht nur die Entscheidungsvariante anhand der Tabelle lösen, sondern auch die Zwischenvariante des Rucksackproblems, bei der wir nach dem maximal erreichbaren Nutzen (nicht aber nach der dazugehörigen Teilmenge der Objekte) gefragt werden, wenn die Kapazität des Rucksacks b ist. Dazu müssen wir lediglich in der Tabelle das größte p finden, für das $W(n,p) \leq b$ gilt.

Der folgende Algorithmus fasst zusammen, wie wir Schritt für Schritt die Tabelle füllen und damit die Zwischenvariante des Rucksackproblems lösen.

Algorithmus: DynProg-KP

1. Definiere $W(i,p) = 0$ für alle $i \in \{1,\ldots,n\}$ und $p \leq 0$.
2. $P := \max_{i\in\{1,\ldots,n\}} p_i$;
3. **for** $p = 1$ **to** p_1 **do** $W(1,p) := w_1$;
4. **for** $p = p_1 + 1$ **to** nP **do** $W(1,p) := \infty$;
5. **for** $i = 2$ **to** n **do**
6. **for** $p = 1$ **to** nP **do**
7. $W(i,p) = \min\{W(i-1,p), W(i-1,p-p_i) + w_i\}$;
8. Ausgabe: maximales $p \in \{1,\ldots,nP\}$ mit $W(n,p) \leq b$

Theorem 1.6. *Der Algorithmus* DynProg-KP *bestimmt in Zeit* $\Theta(n^2P)$ *den maximal erreichbaren Nutzen einer gegebenen Instanz des Rucksackproblems.*

Beweis. Die Korrektheit des Algorithmus folgt direkt aus unseren Vorüberlegungen. Formal kann man per Induktion über i zeigen, dass die Werte $W(i,p)$ genau die Bedeutung haben, die wir ihnen zugedacht haben, nämlich

$$W(i,p) = \min\left\{ w(I) \mid I \subseteq \{1,\dots,i\}, \sum_{i\in I} p_i \geq p \right\}.$$

Das ist aber mit den Vorüberlegungen nur eine leichte Übung, auf die wir hier verzichten.

Die Laufzeit des Algorithmus ist auch nicht schwer zu analysieren. Die Tabelle, die berechnet wird, hat einen Eintrag für jede Kombination von $i \in \{1,\dots,n\}$ und $p \in \{0,\dots,nP\}$. Das sind insgesamt $\Theta(n^2P)$ Einträge und jeder davon kann in konstanter Zeit durch die Bildung eines Minimums berechnet werden. □

Beispiel für DynProg-KP

Wir führen den Algorithmus DynProg-KP auf der folgenden Instanz mit drei Objekten aus.

	i		
	1	2	3
p_i	2	1	3
w_i	3	2	4

Es gilt $P = 3$ und der Algorithmus DynProg-KP berechnet die folgenden Werte $W(i,p)$.

	i		
p	1	2	3
1	3	2	2
2	3	3	3
3	∞	5	4
4	∞	∞	6
5	∞	∞	7
6	∞	∞	9
7	∞	∞	∞

Zur Beantwortung der Frage, ob es eine Teilmenge $I \subseteq \{1,\dots,n\}$ mit $w(I) \leq 4$ und $p(I) \geq 5$ gibt, betrachten wir den Eintrag $W(3,5)$. Dort steht eine 7 und somit benötigt man mindestens Gewicht 7, um Nutzen 5 zu erreichen. Wir können die Frage also mit „nein“ beantworten.

Möchten wir wissen, welchen Nutzen wir maximal mit Gewicht 5 erreichen können, so laufen wir die rechte Spalte von oben nach unten bis zum letzten Eintrag kleiner oder gleich 5 durch. Das ist der Eintrag $W(3,3)$, also kann man mit Gewicht 5 maximal Nutzen 3 erreichen.

Wir halten noch fest, dass der Algorithmus leicht so modifiziert werden kann, dass er die Optimierungsvariante des Rucksackproblems löst, dass er also nicht nur den maximal erreichbaren Nutzen, sondern auch die dazugehörige Teilmenge der Objekte berechnet. Dazu müssen wir nur zusätzlich zu jedem Eintrag $W(i,p)$ speichern, welche Teilmenge $I \subseteq \{1,\dots,i\}$ mit $\sum_{i\in I} p_i \geq p$ Gewicht genau $W(i,p)$ hat. Sei $I(i,p)$ diese Teilmenge. An der Stelle, an der wir

$W(i,p) = \min\{W(i-1,p), W(i-1,p-p_i)+w_i\}$ berechnen, können wir auch $I(i,p)$ berechnen: wird das Minimum vom ersten Term angenommen, so setzen wir $I(i,p) = I(i-1,p)$, ansonsten setzen wir $I(i,p) = I(i-1,p-1) \cup \{i\}$. Der Leser möge sich selbst Gedanken machen, wie eine genaue Implementierung dieses Algorithmus aussehen könnte.

Als nächstes betrachten wir die Laufzeit $\Theta(n^2P)$ des Algorithmus genauer. Ist dies eine polynomielle Laufzeit? Das wäre verwunderlich, denn wir wissen bereits, dass das Rucksackproblem NP-hart ist. Tatsächlich ist die Laufzeit nicht polynomiell. Der Grund ist, dass wir die Laufzeit in der Länge der Eingabe ausdrücken müssen, die Zahlen in der Eingabe aber binär kodiert werden (davon sind wir stillschweigend ausgegangen, weil es die natürliche Art und Weise ist, wie man Zahlen in einem Rechner darstellt). Wählen wir also z. B. $p_1 = \ldots = p_n = 2^n$, $w_1 = \ldots = w_n = 1$ und $b = 1$, so ist die Länge der Eingabe in Bits $\Theta(n^2)$, die Laufzeit ist jedoch $\Theta(n^2P) = \Theta(n^2 2^n)$ und damit exponentiell in der Eingabelänge. Die Laufzeit des Algorithmus ist aber zumindest polynomiell in der Anzahl der Objekte n und dem größten vorkommenden Nutzen P beschränkt.

Definition 1.7 (Pseudopolynomieller Algorithmus)**.** *Ein Algorithmus heißt* pseudopolynomieller Algorithmus*, wenn seine Laufzeit polynomiell in der Eingabelänge und* der größten in der Eingabe vorkommenden Zahl *beschränkt ist.*

Die Existenz eines pseudopolynomiellen Algorithmus für ein Problem bedeutet, dass das Problem unter der Annahme P≠NP nur für Eingaben schwer zu lösen ist, die große Zahlen enthalten. Sind alle Zahlen polynomiell in der Eingabelänge beschränkt, so ist das Problem effizient lösbar. Instanzen für das Rucksackproblem, in denen $p_i \leq n^2$ für alle Nutzenwerte p_i gilt, können mit dem Algorithmus DynProg-KP beispielsweise in Zeit $O(n^4)$ gelöst werden.

1.2.3 Polynomielles Approximationsschema

Wir werden nun den pseudopolynomiellen Algorithmus für das Rucksackproblem nutzen, um einen Approximationsalgorithmus zu entwerfen. Es handelt sich dabei um einen besonderen Approximationsalgorithmus, der als zusätzliche Eingabe einen Parameter erhält, der bestimmt, wie gut die Approximation sein soll.

Definition 1.8 (Approximationsschema)**.** *Ein Approximationsschema A für ein Optimierungsproblem Π ist ein Algorithmus, der als Eingabe ein Paar (I,ε) erhält, wobei $I \in \mathcal{I}_\Pi$ eine Instanz für das Problem Π und $\varepsilon > 0$ eine rationale Zahl ist. Als Ausgabe liefert der Algorithmus für ein Minimierungs- bzw. Maximierungsproblem eine Lösung $x \in \mathcal{S}_I$ mit*

$$w_A(I) \leq (1+\varepsilon)\cdot \mathrm{OPT}(I) \quad \textit{bzw.} \quad w_A(I) \geq (1-\varepsilon)\cdot \mathrm{OPT}(I).$$

- *Ein Approximationsschema heißt* polynomielles Approximationsschema (polynomial time approximation scheme, PTAS)*, wenn seine Laufzeit für jedes feste $\varepsilon > 0$ polynomiell in der Eingabelänge $|I|$ beschränkt ist.*
- *Ein Approximationsschema heißt* voll polynomielles Approximationsschema (fully polynomial time approximation scheme, FPTAS)*, wenn seine Laufzeit polynomiell in der Eingabelänge $|I|$ und in $1/\varepsilon$ beschränkt ist.*

Jedes FPTAS ist auch ein PTAS. Umgekehrt gilt dies jedoch nicht: Ein PTAS könnte z. B. eine Laufzeit von $\Theta(|I|^{1/\varepsilon})$ haben. Diese ist für jedes feste $\varepsilon > 0$ polynomiell in $|I|$ (wobei

der Grad des Polynoms von der Wahl von ε abhängt). Diese Laufzeit ist bei einem FPTAS jedoch nicht erlaubt, weil sie nicht polynomiell in $1/\varepsilon$ beschränkt ist. Eine erlaubte Laufzeit eines FPTAS ist z. B. $\Theta(|I|^3/\varepsilon^2)$.

Existiert ein PTAS für ein Optimierungsproblem, so bedeutet das, dass es für jede Konstante $\varepsilon > 0$ einen $(1+\varepsilon)$-Approximationsalgorithmus für das Problem gibt. Der Grad des Polynoms hängt aber im Allgemeinen vom gewählten ε ab. Ist die Laufzeit zum Beispiel $\Theta(|I|^{1/\varepsilon})$, so ergibt sich für $\varepsilon = 0.01$ eine Laufzeit von $\Theta(|I|^{100})$. Eine solche Laufzeit ist aber nur von theoretischem Interesse und für praktische Anwendungen vollkommen ungeeignet. Bei einem FPTAS ist es hingegen oft so, dass man auch für kleine ε eine passable Laufzeit erhält.

Wir entwerfen nun ein FPTAS für das Rucksackproblem. Die Idee, die diesem zu Grunde liegt, ist die folgende: Mit Hilfe von DynProg-KP können wir effizient Instanzen des Rucksackproblems lösen, in denen alle Nutzenwerte klein (d. h. polynomiell in der Eingabelänge beschränkt) sind. Erhalten wir nun als Eingabe eine Instanz mit großen Nutzenwerten, so skalieren wir diese zunächst, d. h. wir teilen alle durch dieselbe Zahl K. Diese Skalierung führt dazu, dass der maximal erreichbare Nutzen ebenfalls um den Faktor K kleiner wird. Die optimale Lösung ändert sich jedoch nicht. In der skalierten Instanz ist die gleiche Teilmenge $I \subseteq \{1, \ldots, n\}$ optimal wie in der ursprünglichen Instanz.

Nach dem Skalieren sind die Nutzenwerte rationale Zahlen. Der Trick besteht nun darin, die Nachkommastellen der skalierten Nutzenwerte abzuschneiden. Wir können dann Algorithmus DynProg-KP anwenden, um die optimale Lösung für die skalierten und gerundeten Nutzenwerte zu berechnen. Wählen wir den Skalierungsfaktor K richtig, so sind diese Zahlen so klein, dass DynProg-KP polynomielle Laufzeit hat. Allerdings verlieren wir durch das Abschneiden der Nachkommastellen an Präzision. Das bedeutet, die optimale Lösung für die Instanz mit den skalierten und gerundeten Nutzenwerten ist nicht unbedingt optimal für die ursprüngliche Instanz. Ist jedoch K richtig gewählt, so stellt sie eine gute Approximation dar. Wir beschreiben den Algorithmus nun formal.

Algorithmus: FPTAS-KP

Die Eingabe sei $(\mathcal{I}, \varepsilon)$ mit $\mathcal{I} = (p_1, \ldots, p_n, w_1, \ldots, w_n, b)$.

1. $P := \max_{i \in \{1,\ldots,n\}} p_i$; $K := \frac{\varepsilon P}{n}$; // wähle Skalierungsfaktor
2. **for** $i = 1$ **to** n **do** $p'_i = \lfloor p_i / K \rfloor$; // skaliere alle Nutzenwerte
3. Benutze Algorithmus DynProg-KP, um die optimale Lösung für die Instanz $p'_1, \ldots, p'_n, w_1, \ldots, w_n, b$ des Rucksackproblems zu bestimmen.

Theorem 1.9. *Der Algorithmus* FPTAS-KP *ist ein FPTAS für das Rucksackproblem mit Laufzeit* $O(n^3/\varepsilon)$.

Beweis. Wir betrachten zunächst die Laufzeit des Algorithmus. Diese wird durch den Aufruf von DynProg-KP dominiert, da die Skalierung der Nutzenwerte in den ersten beiden Schritten in Linearzeit erfolgt. Sei $P' = \max_{i \in \{1,\ldots,n\}} p'_i$ der größte der skalierten Nutzenwerte. Dann beträgt die Laufzeit von DynProg-KP $\Theta(n^2 P')$ gemäß Theorem 1.6. Es gilt

$$P' = \max_{i \in \{1,\ldots,n\}} \left\lfloor \frac{p_i}{K} \right\rfloor = \left\lfloor \frac{P}{K} \right\rfloor = \left\lfloor \frac{n}{\varepsilon} \right\rfloor \leq \frac{n}{\varepsilon}$$

und somit beträgt die Laufzeit von DynProg-KP $\Theta(n^2 P') = \Theta(n^3/\varepsilon)$.

Nun müssen wir noch zeigen, dass der Algorithmus eine $(1-\varepsilon)$-Approximation liefert. Sei dazu $I' \subseteq \{1,\dots,n\}$ eine optimale Lösung für die Instanz mit den Nutzenwerten $p'_1,\dots,p'_n$ und sei $I \subseteq \{1,\dots,n\}$ eine optimale Lösung für die eigentlich zu lösende Instanz mit den Nutzenwerte $p_1,\dots,p_n$. Da das Skalieren der Nutzenwerte die optimale Lösung nicht ändert, ist I' auch eine optimale Lösung bezüglich der Nutzenwerte $p^*_1,\dots,p^*_n$ mit $p^*_i = K \cdot p'_i$.

Beispiel

Sei $n=4$, $P=50$ und $\varepsilon = \frac{4}{5}$. Dann ist $K = \frac{\varepsilon P}{n} = 10$.
Die verschiedenen Nutzenwerte könnten zum Beispiel wie folgt aussehen:

$p_1 = 33$	$p'_1 = 3$	$p^*_1 = 30$
$p_2 = 25$	$p'_2 = 2$	$p^*_2 = 20$
$p_3 = 50$	$p'_3 = 5$	$p^*_3 = 50$
$p_4 = 27$	$p'_4 = 2$	$p^*_4 = 20$

Man sieht an diesem Beispiel, dass p_i und p^*_i in der gleichen Größenordnung liegen und sich nicht stark unterscheiden. Die Nutzenwerte p^*_i kann man als eine Version der Nutzenwerte p_i mit geringerer Präzision auffassen.

Für eine Teilmenge $J \subseteq \{1,\dots,n\}$ bezeichnen wir mit $p(J)$ bzw. $p^*(J)$ ihren Gesamtnutzen bezüglich der ursprünglichen Nutzenwerte und ihren Gesamtnutzen bezüglich der Nutzenwerte $p^*_1,\dots,p^*_n$:

$$p(J) = \sum_{i \in J} p_i \qquad \text{und} \qquad p^*(J) = \sum_{i \in J} p^*_i.$$

Dann ist $p(I')$ der Wert der Lösung, die der Algorithmus FPTAS-KP ausgibt und $p(I)$ ist der Wert OPT der optimalen Lösung.

Zunächst halten wir fest, dass die Nutzenwerte p_i und p^*_i nicht stark voneinander abweichen. Es gilt

$$p^*_i = K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor \geq K \left(\frac{p_i}{K} - 1 \right) = p_i - K$$

und

$$p^*_i = K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor \leq p_i.$$

Dementsprechend gilt für jede Teilmenge $J \subseteq \{1,\dots,n\}$

$$p^*(J) \in [p(J) - nK, p(J)]\,.$$

Wir wissen, dass I' auch die optimale Lösung bezüglich der Nutzenwerte $p^*_1,\dots,p^*_n$ ist, da diese durch Skalieren aus den Nutzenwerten $p'_1,\dots,p'_n$ hervorgegangen sind. Es gilt also insbesondere $p^*(I') \geq p^*(I)$ und somit

$$p(I') \geq p^*(I') \geq p^*(I) \geq p(I) - nK.$$

Da jedes Objekt alleine in den Rucksack passt, gilt $p(I) \geq P$ und damit auch

$$\frac{p(I')}{p(I)} \geq 1 - \frac{nK}{p(I)} = 1 - \frac{\varepsilon P}{p(I)} \geq 1 - \varepsilon.$$

Wegen $\mathrm{OPT} = p(I)$ ist der Beweis damit abgeschlossen. □

Das FPTAS für das Rucksackproblem rundet die Nutzenwerte so, dass der pseudopolynomielle Algorithmus die gerundete Instanz in polynomieller Zeit lösen kann. Wurde durch das Runden die Präzision nicht zu stark verringert, so ist die optimale Lösung bezüglich der gerundeten Nutzenwerte eine gute Approximation der optimalen Lösung bezüglich der eigentlichen Nutzenwerte. Diese Technik zum Entwurf eines FPTAS lässt sich auch auf viele andere Probleme anwenden, für die es einen pseudopolynomiellen Algorithmus gibt. Ob sich die Technik anwenden lässt, hängt aber vom konkreten Problem und dem pseudopolynomiellen Algorithmus ab.

Für das Rucksackproblem können wir beispielsweise auch einen pseudopolynomiellen Algorithmus angeben, dessen Laufzeit $\Theta(n^2W)$ ist, wobei $W = \max_{i\in\{1,\dots,n\}} w_i$ das größte Gewicht ist. Bei einem solchen Algorithmus wäre es naheliegend nicht die Nutzenwerte, sondern die Gewichte zu runden und dann mit Hilfe des pseudopolynomiellen Algorithmus die Instanz mit gerundeten Gewichten zu lösen. Das Ergebnis ist dann aber im Allgemeinen keine gute Approximation. Runden wir die Gewichte nämlich ab, so kann es passieren, dass dadurch eine Lösung gültig wird, die bezüglich der eigentlichen Gewichte zu schwer ist, die aber einen höheren Nutzen hat als das eigentliche Optimum. Der Algorithmus würde dann diese Lösung berechnen, die für die eigentliche Instanz gar nicht gültig ist. Rundet man andererseits alle Gewichte auf, so kann es passieren, dass die optimale Lösung ein zu hohes Gewicht bezüglich der gerundeten Gewichte hat und nicht mehr gültig ist. Es besteht dann die Möglichkeit, dass alle Lösungen, die bezüglich der aufgerundeten Gewichte gültig sind, einen viel kleineren Nutzen haben als die optimale Lösung. In diesem Fall würde der Algorithmus ebenfalls keine gute Approximation berechnen.

Als Faustregel kann man festhalten, dass die Chancen einen pseudopolynomiellen Algorithmus in ein FPTAS zu transformieren nur dann gut stehen, wenn der Algorithmus pseudopolynomiell bezogen auf die Koeffizienten in der Zielfunktion ist. Ist er pseudopolynomiell bezogen auf die Koeffizienten in den Nebenbedingungen, so ist hingegen besondere Vorsicht geboten.

1.3 Traveling Salesperson Problem

Das Traveling Salesperson Problem (TSP) haben wir bereits im zweiten Teil der Vorlesung kennengelernt. In diesem Abschnitt werden wir zunächst unser erstes negatives Ergebnis über Approximationsalgorithmen zeigen. Bisher haben wir für verschiedene Probleme gezeigt, dass Approximationsalgorithmen mit gewissen Güten existieren. Jetzt zeigen wir, dass für das allgemeine TSP kein Approximationsalgorithmus existiert, der eine sinnvolle Approximationsgüte liefert.

Traveling Salesperson Problem (TSP)

Eingabe:	Menge $V = \{v_1, \dots, v_n\}$ von Knoten symmetrische Distanzfunktion $d : V \times V \to \mathbb{R}_{>0}$ (d. h. $\forall u, v \in V, u \neq v : d(u, v) = d(v, u) > 0$)
Lösungen:	alle Permutationen $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$ ein solche Permutation nennen wir auch *Tour*
Zielfunktion:	minimiere $\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})$

1.3.1 Nichtapproximierbarkeit und starke NP-Härte

Was ist die beste Approximationsgüte, die für TSP in polynomieller Zeit erreicht werden kann? Wir zeigen, dass es nicht mal einen 2^n-Approximationsalgorithmus für das TSP gibt. Das bedeutet insbesondere, dass es für keine Konstante a einen a-Approximationsalgorithmus gibt.

Theorem 1.10. *Falls P≠NP, so existiert kein 2^n-Approximationsalgorithmus für das TSP.*

Beweis. Wir haben uns im zweiten Teil der Vorlesung bereits kurz mit dem Hamiltonian Cycle (HC) Problem beschäftigt. Dies ist das Problem für einen ungerichteten Graphen zu entscheiden, ob es einen Kreis gibt, der jeden Knoten genau einmal enthält. Ein solcher Kreis wird auch Hamiltonkreis genannt. Wir haben angesprochen, dass dieses Problem NP-vollständig ist und nutzen das als Ausgangspunkt für das TSP. Dazu geben wir eine spezielle polynomielle Reduktion von HC auf TSP an, aus der wir folgenden Schluss ziehen können: Falls ein 2^n-Approximationsalgorithmus für das TSP existiert, so kann HC in polynomieller Zeit gelöst werden.

Wir müssen uns also fragen, wie wir einen 2^n-Approximationsalgorithmus für das TSP nutzen können, um HC zu lösen. Sei $G = (V, E)$ der Graph, für den wir entscheiden wollen, ob er einen Hamiltonkreis besitzt. Dazu erzeugen wir eine TSP Instanz mit der Knotenmenge V und der folgenden Distanzfunktion $d : V \times V \to \mathbb{R}_{>0}$:

$$\forall u, v \in V, u \neq v : d(u, v) = d(v, u) = \begin{cases} 1 & \text{falls } \{u, v\} \in E, \\ n2^{n+1} & \text{falls } \{u, v\} \notin E. \end{cases}$$

Wir haben diese Distanzfunktion so gewählt, dass die konstruierte Instanz für das TSP genau dann eine Tour der Länge n besitzt, falls der Graph G einen Hamiltonkreis besitzt. Besitzt der Graph G hingegen keinen Hamiltonkreis, so hat jede Tour der TSP-Instanz mindestens Länge $n - 1 + n2^{n+1}$. Die Distanzfunktion d kann in polynomieller Zeit berechnet werden, da die Kodierungslänge von $n2^{n+1}$ polynomiell beschränkt ist, nämlich $\Theta(n)$.

Nehmen wir an, wir haben einen 2^n-Approximationsalgorithmus für das TSP. Wenn wir diesen Algorithmus auf einen Graphen G anwenden, der einen Hamiltonkreis enthält, so berechnet er eine Tour mit Länge höchstens $n2^n$, da die Länge der optimalen Tour n ist. Diese Tour kann offenbar keine Kante mit Gewicht $n2^{n+1}$ enthalten. Somit besteht die Tour nur aus Kanten aus der Menge E und ist somit ein Hamiltonkreis von G.

Damit ist gezeigt, dass ein 2^n-Approximationsalgorithmus für das TSP genutzt werden kann, um in polynomieller Zeit einen Hamiltonkreis in einem Graphen zu berechnen, falls ein solcher existiert. □

Die Wahl von 2^n in Theorem 1.10 ist relativ willkürlich. Tatsächlich kann man mit obigem Beweis sogar für jede in Polynomialzeit berechenbare Funktion $r : \mathbb{N} \to \mathbb{R}_{\geq 1}$ zeigen, dass es keinen r-Approximationsalgorithmus für das TSP gibt, falls P≠NP.

Die Technik, die wir im letzten Beweis benutzt haben, lässt sich auch für viele andere Probleme einsetzen. Wir fassen sie noch einmal zusammen: Um zu zeigen, dass es unter der Annahme P≠NP für ein Problem keinen r-Approximationsalgorithmus gibt, zeigt man, dass man mit Hilfe eines solchen Algorithmus ein NP-hartes Problem in polynomieller Zeit lösen kann.

Wäre es uns nur darum gegangen, zu zeigen, dass die Entscheidungsvariante des TSP NP-hart ist, so hätten wir in obiger Reduktion das Gewicht der Kanten $\{u, v\} \notin E$ auch auf 2 statt auf $n2^{n+1}$ setzen können. Dann wäre die Aussage, dass der gegebene Graph genau dann einen Hamiltonkreis besitzt, wenn die konstruierte TSP-Instanz eine Tour der Länge n enthält, immer noch richtig gewesen. Wir hätten so zwar kein so starkes Ergebnis über die Nichtapproximierbarkeit des TSP erhalten, aber wir hätten etwas anderes gezeigt: Das TSP ist bereits für den Spezialfall NP-hart, dass alle Gewichte entweder 1 oder 2 sind. Das ist deshalb interessant, weil es unter der Annahme P≠NP die Existenz eines pseudopolynomiellen Algorithmus ausschließt.

Definition 1.11. *Ein Problem heißt* stark NP-hart *wenn es bereits für den Spezialfall NP-hart ist, dass alle in der Eingabe vorkommenden Zahlen polynomiell in der Eingabelänge beschränkt sind.*

Lemma 1.12. *Ist ein Problem* Π *stark NP-hart, so gibt es unter der Annahme P≠NP keinen pseudopolynomiellen Algorithmus für das Problem.*

Beweis. Ein Algorithmus heißt pseudopolynomieller Algorithmus, wenn seine Laufzeit polynomiell in der Eingabelänge n und der größten in der Eingabe vorkommenden Zahl Z beschränkt ist. Sei A ein pseudopolynomieller Algorithmus für Π mit Laufzeit $O(p(n, Z))$, wobei p ein Polynom in zwei Variablen ist.

Ist Π stark NP-hart, so gibt es ein Polynom q, sodass Π bereits für den Spezialfall NP-hart ist, dass alle Zahlen bei Eingaben der Länge n durch $q(n)$ beschränkt sind. Der Algorithmus A würde diese Eingaben aber in Zeit $O(p(n, q(n))) = O(r(n))$, für ein geeignetes Polynom r, lösen. Somit wäre A ein Polynomialzeitalgorithmus für ein NP-hartes Problem und damit wäre P=NP. □

1.3.2 Metrisches TSP

Es gibt viele Möglichkeiten, mit Optimierungsproblemen zu verfahren, für die es nicht mal gute Approximationsalgorithmen gibt. Man sollte sich die Frage stellen, ob man das Problem zu allgemein formuliert hat. Oftmals erfüllen Eingaben, die in der Praxis auftreten nämlich Zusatzeigenschaften, die das Problem einfacher machen. Eine Eigenschaft, die in vielen Anwendungen gegeben ist, ist die Dreiecksungleichung. Das bedeutet, dass

$$d(u, w) \leq d(u, v) + d(v, w)$$

für alle Knoten $u, v, w \in V$ gilt. Möchte man von einem Knoten zu einem anderen, so ist also der direkte Weg nie länger als der zusammengesetzte Weg über einen Zwischenknoten. Sind die Knoten zum Beispiel Punkte im euklidischen Raum, so ist diese Eigenschaft erfüllt.

Definition 1.13. *Sei* X *eine Menge und* $d : X \times X \to \mathbb{R}_{\geq 0}$ *eine Funktion. Die Funktion* d *heißt* Metrik auf X*, wenn die folgenden drei Eigenschaften erfüllt sind.*

- $\forall x, y \in X : d(x, y) = 0 \iff x = y$ (positive Definitheit)
- $\forall x, y \in X : d(x, y) = d(y, x)$ (Symmetrie)
- $\forall x, y, z \in X : d(x, z) \leq d(x, y) + d(y, z)$ (Dreiecksungleichung)

Das Paar (X, d) *heißt* metrischer Raum.

Wir interessieren uns nun für Instanzen des TSP, bei denen d eine Metrik auf V ist. Beim metrischen TSP erfüllen also alle Instanzen die Dreiecksungleichung. Diesen Spezialfall des allgemeinen TSP nennen wir *metrisches TSP*. Wir zeigen, dass sich dieser Spezialfall deutlich besser approximieren lässt als das allgemeine Problem. Zunächst halten wir aber fest, dass das metrische TSP noch NP-hart ist. Dies folgt direkt aus der Diskussion nach dem Beweis von Theorem 1.10. Dort haben wir nämlich diskutiert, dass das TSP auch dann noch NP-hart ist, wenn alle Distanzen entweder 1 oder 2 sind. Der Leser möge sich überlegen, dass die Dreiecksungleichung automatisch erfüllt ist, wenn nur 1 und 2 als Distanzen erlaubt sind. Somit ist gezeigt, dass auch das metrische TSP noch NP-hart ist.

In folgendem Algorithmus benutzen wir den Begriff eines *Eulerkreises*. Dabei handelt es sich um einen Kreis in einem Graphen, der jede Kante genau einmal benutzt, der Knoten aber mehrfach benutzen darf. Ein Graph mit einem solchen Kreis wird auch *eulerscher Graph* genannt.

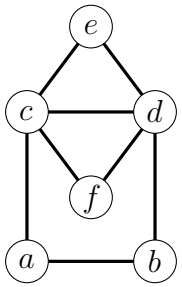


Abbildung 1.5: Dieser Graph enthält den Eulerkreis $(a, c, e, d, f, c, d, b, a)$.

Das Problem, einen Eulerkreis zu finden, sieht auf den ersten Blick so ähnlich aus wie das NP-harte Problem, einen Hamiltonkreis zu finden, also einen Kreis bei dem jeder Knoten genau einmal besucht wird. Tatsächlich kann man aber effizient entscheiden, ob ein gegebener Graph einen Eulerkreis besitzt und, wenn ja, einen solchen finden.

Wir erweitern unsere Betrachtungen auf Multigraphen. Das sind Graphen, in denen zwischen zwei Knoten eine beliebige Anzahl an Kanten verlaufen kann (statt maximal einer wie in einem normalen Graphen). Man kann zeigen, dass ein zusammenhängender Multigraph genau dann einen Eulerkreis enthält, wenn jeder Knoten geraden Grad hat. Außerdem kann man in einem solchen Graphen auch in polynomieller Zeit einen Eulerkreis berechnen. Diese Behauptungen zu beweisen, überlassen wir hier dem Leser als Übung.

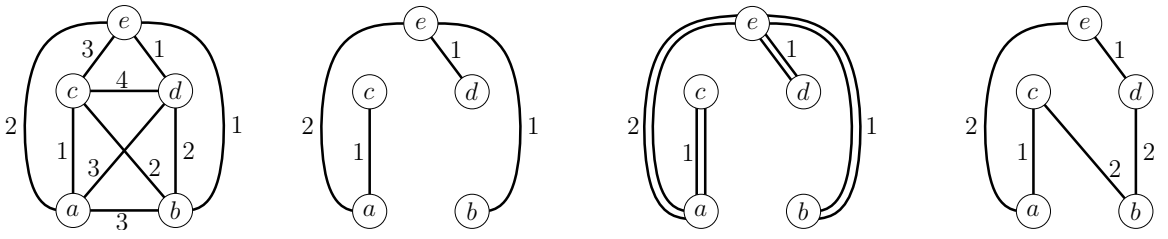
Nun können wir den ersten Approximationsalgorithmus für das metrische TSP vorstellen.

Algorithmus: METRIC-TSP

Die Eingabe sei eine Knotenmenge V und eine Metrik d auf V.

1. Sei $G = (V, E)$ ein vollständiger ungerichteter Graph mit Knotenmenge V. Berechne einen minimalen Spannbaum T von G bezüglich der Distanzen d.
2. Verdopple alle Kanten in T und erzeuge so einen Multigraphen G', in dem jeder Knoten geraden Grad hat.
3. Finde einen Eulerkreis A in G'.
4. Gib die Knoten in der Reihenfolge ihres ersten Auftretens in A aus. Das Ergebnis sei der Hamiltonkreis C.

In Schritt 4 wird also der gefundene Eulerkreis A in G' (der alle Knoten aus V mindestens einmal enthält) in einen Hamiltonkreis umgebaut. Dazu werden Knoten, die bereits besucht wurden, übersprungen. So wird aus einem Eulerkreis (a, b, c, b, a) zum Beispiel der Hamiltonkreis (a, b, c, a). Bilden die Distanzen d eine Metrik auf V, so ist garantiert, dass der Kreis durch das Überspringen von bereits besuchten Knoten nicht länger wird.



(a) Eingabe (b) Spannbaum (c) $A = (c, a, e, d, e, b, e, a, c)$ (d) Lösung C

Abbildung 1.6: Beispiel für den Algorithmus Metric-TSP

Theorem 1.14. *Der Algorithmus* Metric-TSP *ist ein 2-Approximationsalgorithmus für das metrische TSP.*

Beweis. Für eine Menge von Kanten $X \subseteq E$ bezeichnen wir im Folgenden mit $d(X)$ die Summe $\sum_{\{u,v\}\in X} d(u, v)$. Wir fassen in diesem Beweis T, A und C als ungeordnete Teilmengen der Kanten auf und benutzen entsprechend auch die Bezeichnungen $d(T)$, $d(A)$ und $d(C)$.

Zunächst zeigen wir, dass $d(T)$ eine untere Schranke für die Länge OPT der optimalen Tour ist. Sei $C^* \subseteq E$ ein kürzester Hamiltonkreis in G bezüglich der Distanzen d. Entfernen wir aus diesem Kreis eine beliebige Kante e, so erhalten wir einen Weg P, der jeden Knoten genau einmal enthält. Ein solcher Weg ist insbesondere ein Spannbaum des Graphen G, also gilt $d(P) \geq d(T)$, da T ein minimaler Spannbaum ist. Insgesamt erhalten wir damit

$$\mathrm{OPT} = d(C^*) \geq d(P) + d(e) \geq d(P) \geq d(T).$$

Wir schließen den Beweis ab, indem wir zeigen, dass die Länge des ausgegebenen Hamiltonkreises C durch $2d(T)$ nach oben beschränkt ist. Der Eulerkreis A in G' benutzt jede Kante aus dem Spannbaum T genau zweimal. Damit gilt $d(A) = 2d(T)$. Um von dem Eulerkreis A zu dem Hamiltonkreis C zu gelangen, werden Knoten, die der Eulerkreis A mehrfach besucht, übersprungen. Da die Distanzen die Dreiecksungleichung erfüllen, wird durch das Überspringen von Knoten die Tour nicht verlängert. Es gilt also $d(C) \leq d(A)$. Insgesamt erhalten wir

$$d(C) \leq d(A) \leq 2d(T).$$

Zusammen mit obiger Aussage über OPT beweist das, dass der Algorithmus stets eine 2-Approximation liefert. □

Es stellt sich wieder die Frage, ob wir den Algorithmus gut analysiert haben, oder ob er in Wirklichkeit eine bessere Approximation liefert. Das folgende Beispiel zeigt, dass sich unsere Analyse nicht signifikant verbessern lässt.

Untere Schranke für METRIC-TSP

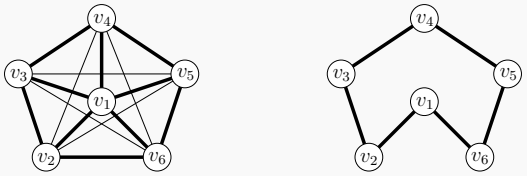
Wir betrachten eine Instanz mit Knotenmenge $V = \{v_1, \ldots, v_n\}$, in der jede Distanz entweder 1 oder 2 ist. Wie wir bereits oben erwähnt haben, ist die Dreiecksungleichung automatisch erfüllt ist, wenn nur 1 und 2 als Distanzen erlaubt sind. Wir gehen davon aus, dass n gerade ist. Es gelte

$$\forall i \in \{2, \ldots, n\} : d(v_1, v_i) = 1$$

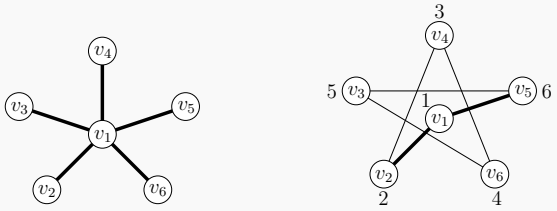
und

$$d(v_2, v_3) = d(v_3, v_4) = \ldots = d(v_{n-1}, v_n) = d(v_n, v_2) = 1.$$

Alle anderen Distanzen seien 2. Dann bilden die Kanten mit Distanz 1 die Vereinigung eines Sterns mit Mittelpunkt v_1 und des Kreises $(v_2, v_3, \ldots, v_n, v_2)$. Die folgende Abbildung zeigt links die Instanz für $n = 6$, wobei dicke Kanten Distanz 1 haben und dünne Kanten Distanz 2.



Die optimale Tour für diese Instanz hat Länge n. Eine mögliche Wahl, die auch in der obigen Abbildung rechts dargestellt ist, ist $(v_1, v_2, \ldots, v_n, v_1)$. Nun kann es sein, dass der Algorithmus METRIC-TSP genau den Stern als minimalen Spannbaum wählt. Dies ist in der Abbildung unten links dargestellt. Basierend auf diesem Spannbaum könnte er den Hamiltonkreis $(v_1, v_2, v_4, v_6, \ldots, v_n, v_3, v_5, \ldots, v_{n-1}, v_1)$ berechnen. Dies ist in der Abbildung unten rechts dargestellt.



Der vom Algorithmus berechnete Hamiltonkreis besteht aus zwei Kanten mit Distanz 1 und $n-2$ Kanten mit Distanz 2. Somit sind seine Kosten insgesamt $2+2(n-2) = 2n-2$. Für große n kommt der Approximationsfaktor $(2n-2)/n$ der Zahl 2 beliebig nahe.

1.3.3 Christofides-Algorithmus

Wir können den Algorithmus, den wir im letzten Abschnitt präsentiert haben, deutlich verbessern. Wir waren im zweiten Schritt des Algorithmus verschwenderisch. Dort haben wir alle Kanten des Spannbaums verdoppelt, um einen Graphen zu erhalten, in dem jeder Knotengrad gerade ist. Es ist aber gut möglich, dass in dem Spannbaum ohnehin bereits viele Knoten geraden Grad haben. Für diese ist eine Verdoppelung der Kanten gar nicht notwendig. Diese Beobachtung führt zu folgendem Algorithmus, den Nicos Christofides 1976 vorgeschlagen hat. Man nennt den Algorithmus deshalb auch Christofides-Algorithmus, und es handelt sich auch heute noch um den besten bekannten Approximationsalgorithmus für das metrische TSP.

Der Algorithmus benutzt den Begriff eines *perfekten Matchings*. Wir haben bereits bei dem

Entwurf eines Approximationsalgorithmus für das Vertex Cover Problem eingeführt, was ein Matching M eines Graphen $G = (V, E)$ ist, nämlich eine Teilmenge der Kanten, so dass kein Knoten zu mehr als einer Kante aus M inzident ist. Ein perfektes Matching M ist ein Matching mit $|M| = \frac{|V|}{2}$. Es muss also jeder Knoten zu genau einer Kante aus M inzident sein. Es ist bekannt, dass in einem Graphen mit einer Kantengewichtung ein perfektes Matching mit minimalem Gewicht in polynomieller Zeit berechnet werden kann. Wie der Algorithmus dafür aussieht, wollen wir hier aber nicht diskutieren.

Algorithmus: CHRISTOFIDES

Die Eingabe sei eine Knotenmenge V und eine Metrik d auf V.

1. Sei $G = (V, E)$ ein vollständiger ungerichteter Graph mit Knotenmenge V. Berechne einen minimalen Spannbaum T von G bezüglich der Distanzen d.
2. Sei $V' = \{v \in V \mid v \text{ hat ungeraden Grad in } T\}$. Berechne auf der Menge V' ein perfektes Matching M mit minimalem Gewicht bezüglich d.
3. Finde einen Eulerkreis A in dem Multigraphen $\tilde{G} = (V, T \cup M)$. Wir fassen $\tilde{G}$ als Multigraphen auf, d. h. jede Kante $e \in T \cap M$ ist in $\tilde{G}$ zweimal enthalten.
4. Gib die Knoten in der Reihenfolge ihres ersten Auftretens in A aus. Das Ergebnis sei der Hamiltonkreis C.

Theorem 1.15. *Der Christofides-Algorithmus ist ein $\frac{3}{2}$-Approximationsalgorithmus für das metrische TSP.*

Beweis. Zunächst beweisen wir, dass der Algorithmus, so wie er oben beschrieben ist, überhaupt durchgeführt werden kann. Dazu sind zwei Dinge zu klären:

1. Warum existiert auf der Menge V' ein perfektes Matching?
2. Warum existiert in dem Graphen $\tilde{G} = (V, T \cup M)$ ein Eulerkreis?

Da der Graph vollständig ist, müssen wir zur Beantwortung der ersten Frage nur zeigen, dass V' eine gerade Anzahl an Knoten enthält. Dazu betrachten wir den Graphen (V, T). Für $v \in V$ bezeichne $\delta(v)$ den Grad des Knoten v in diesem Graphen. Dann ist

$$\sum_{v \in V} \delta(v) = 2|T|$$

eine gerade Zahl, da jede der $|T|$ Kanten inzident zu genau zwei Knoten ist. Bezeichne q die Anzahl an Knoten mit ungeradem Grad. Ist q ungerade, so ist auch die Summe der Knotengrade ungerade, im Widerspruch zu der gerade gemachten Beobachtung.

Zur Beantwortung der zweiten Frage müssen wir lediglich zeigen, dass in dem Graphen $\tilde{G} = (V, T \cup M)$ jeder Knoten geraden Grad besitzt. Das folgt aber direkt aus der Konstruktion: ein Knoten hat entweder bereits geraden Grad im Spannbaum T, oder er hat ungeraden Grad im Spannbaum und erhält eine zusätzliche Kante aus dem Matching.

Nun wissen wir, dass der Algorithmus immer eine gültige Lösung berechnet. Um seinen Approximationsfaktor abzuschätzen, benötigen wir eine untere Schranke für den Wert der optimalen Lösung. Wir haben im vorangegangenen Abschnitt bereits eine solche untere Schranke gezeigt, nämlich das Gewicht des minimalen Spannbaumes. Wir zeigen nun noch eine weitere Schranke.

Lemma 1.16. *Es sei $V' \subseteq V$ beliebig, sodass $|V'|$ gerade ist. Außerdem sei M ein perfektes Matching auf V' mit minimalem $d(M)$. Dann gilt $d(M) \leq \mathrm{OPT}/2$.*

Beweis. Es sei C^* eine optimale TSP Tour auf der Knotenmenge V, und es sei C' eine Tour auf der Knotenmenge V', die entsteht, wenn wir die Knoten in V' in der Reihenfolge besuchen, in der sie auch in C^* besucht werden. Das heißt, wir nehmen in C^* Abkürzungen und überspringen die Knoten, die nicht zu V' gehören. Wegen der Dreiecksungleichung kann sich die Länge der Tour nicht vergrößern und es gilt $\mathrm{OPT} = d(C^*) \geq d(C')$.

Es sei $V' = \{v_1, \ldots, v_k\}$ und ohne Beschränkung der Allgemeinheit besuche die Tour C' die Knoten in der Reihenfolge $(v_1, \ldots, v_k, v_1)$. Dann können wir die Tour C' in zwei disjunkte perfekte Matchings M_1 und M_2 wie folgt zerlegen:

$$M_1 = \{\{v_1, v_2\}, \{v_3, v_4\}, \ldots, \{v_{k-1}, v_k\}\}$$

und

$$M_2 = \{\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_k, v_1\}\}.$$

Wir haben bei der Definition von M_1 und M_2 ausgenutzt, dass $k = |V'|$ gerade ist. Wegen $C' = M_1 \cup M_2$ und $M_1 \cap M_2 = \emptyset$ gilt

$$d(M_1) + d(M_2) = d(C') \leq \mathrm{OPT}.$$

Somit hat entweder M_1 oder M_2 ein Gewicht kleiner oder gleich $\mathrm{OPT}/2$. Dies gilt dann natürlich insbesondere für das perfekte Matching M mit minimalem $d(M)$. □

Mit Hilfe dieses Lemmas folgt nun direkt die Approximationsgüte. Mit den gleichen Argumenten wie schon beim Algorithmus Metric-TSP berechnet der Christofides-Algorithmus eine Tour C deren Länge durch $d(T) + d(M)$ nach oben beschränkt ist. Wir wissen bereits, dass $d(T) \leq \mathrm{OPT}$ gilt und das obige Lemma sagt nun noch, dass $d(M) \leq \mathrm{OPT}/2$ gilt. Zusammen bedeutet das

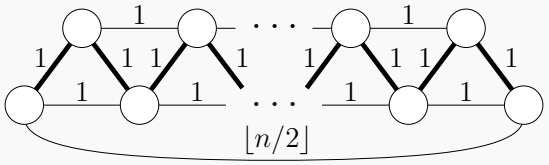
$$d(C) \leq d(T) + d(M) \leq \mathrm{OPT} + \frac{1}{2} \cdot \mathrm{OPT} \leq \frac{3}{2} \cdot \mathrm{OPT}.$$

Damit ist das Theorem bewiesen. □

Auch für den Christofides-Algorithmus können wir wieder zeigen, dass sich unsere Analyse nicht signifikant verbessern lässt.

Untere Schranke für den Christofides-Algorithmus

Sei $n \in \mathbb{N}$ ungerade. Wir betrachten die folgende Instanz des metrischen TSP.



Alle Kanten, die nicht eingezeichnet sind, haben die größtmögliche Länge, die die Dreiecksungleichung zulässt. Die optimale Tour für diese Instanz hat Länge n. Berechnet der Christofides-Algorithmus jedoch im ersten Schritt den Spannbaum, der durch die dicken Kanten angedeutet ist, so ist der Spannbaum ein Pfad. Das heißt, nur die beiden Endknoten haben ungeraden Grad und werden im Matching M durch die Kante mit Gewicht $\lfloor n/2 \rfloor$ verbunden. Somit berechnet der Algorithmus eine Lösung mit Länge $(n-1)+\lfloor n/2 \rfloor$. Für große n entspricht dies in etwa $3n/2$ und der Approximationsfaktor kommt $3/2$ beliebig nahe.

Ist $\frac{3}{2}$ der bestmögliche Approximationsfaktor, der für das metrische TSP unter der Annahme P≠NP erreicht werden kann? Diese Frage ist bis heute ungeklärt. Man kennt bisher keinen besseren Approximationsalgorithmus als den Christofides-Algorithmus. Auf der anderen Seite haben Christos Papadimitriou und Santosh Vempala [4] gezeigt, dass es keinen $\frac{220}{219}$-Approximationsalgorithmus für das metrische TSP geben kann, falls P≠NP. Die Lücke zwischen $\frac{220}{219} \approx 1,004$ und $\frac{3}{2}$ zu schließen, ist ein interessantes offenes Problem der theoretischen Informatik.

Die untere Schranke von $\frac{220}{219}$ impliziert natürlich insbesondere, dass es unter der Annahme P≠NP kein PTAS für das metrische TSP geben kann. Es stellt sich die Frage, ob das metrische TSP immer noch zu allgemein ist. In manchen praktischen Anwendungen, in denen das TSP auftritt, erfüllen die Eingaben noch weitere Eigenschaften zusätzlich zur Dreiecksungleichung. Oft sind die Knoten Punkte in einem Raum $\mathbb{R}^d$ und der Abstand zweier Knoten $x \in \mathbb{R}^d$ und $y \in \mathbb{R}^d$ ist durch den euklidischen Abstand $d(x,y) = \sqrt{(x_1 - y_1)^2 + \ldots + (x_d - y_d)^2}$ definiert. Das TSP eingeschränkt auf solche Eingaben nennen wir *euklidisches TSP*. Es handelt sich dabei um einen Spezialfall des metrischen TSP. Für diesen Spezialfall gibt es ein PTAS, obwohl er immer noch stark NP-hart ist.

Kapitel 2

Online-Algorithmen

Bisher sind wir bei Optimierungsproblemen davon ausgegangen, dass die konkrete Eingabe dem Algorithmus von Anfang an komplett bekannt ist. In manchen Anwendungen ist das aber nicht der Fall und die Eingabe wird erst Schritt für Schritt aufgedeckt. Probleme mit dieser Eigenschaft nennt man *Online-Probleme*. Die Speicherverwaltung eines Rechners muss beispielsweise bei dem Ausführen eines Programms, ohne dessen zukünftiges Verhalten zu kennen, entscheiden, welche Seiten im CPU-Cache gehalten und welche in den Hauptspeicher ausgelagert werden. Auch im Wirtschaftsleben sind Online-Probleme keine Seltenheit. Autofahrer stehen beispielsweise oft vor der Entscheidung, wann sie eine Tankstelle anfahren, ohne die Entwicklung der Spritpreise in den nächsten Tagen zu kennen.

Man könnte meinen, dass man bei solchen Online-Problemen algorithmisch wenig ausrichten kann. Weiß man nicht, auf welche Seiten ein Programm als nächstes zugreifen wird, so kann man scheinbar keine sinnvolle Entscheidung treffen, welche Seiten in den Hauptspeicher ausgelagert werden sollten. Tatsächlich ist dies aber nicht der Fall und man kann durch geschicktes Verhalten bei Online-Problemen besser abschneiden als bei beliebigem Verhalten.

Um eine bessere Intuition für Online-Probleme und Algorithmen zu entwickeln, schauen wir uns zunächst ein Problem an, das in der Literatur als *Ski Rental* bekannt ist. Es lässt sich einfach beschreiben: Wir gehen zum ersten Mal in unserem Leben Ski fahren und wissen noch nicht, ob es uns gefallen wird und wie lange wir Spaß daran haben werden. Es gibt die Möglichkeiten, sich für 800 € eine neue Ski-Ausrüstung zu kaufen oder für 50 € pro Tag eine zu leihen. Solange man noch keine Ausrüstung gekauft hat, steht man jeden morgen erneut vor der Entscheidung, eine neue Ausrüstung zu kaufen oder für den Tag wieder eine zu leihen.

Kaufen wir direkt am ersten Tag die Ausrüstung, stellen dann aber fest, dass uns Skifahren keinen Spaß macht, so haben wir Kosten in Höhe von 800 €, obwohl 50 € ausgereicht hätten. Wir haben dann also 16 mal so viel ausgegeben wie notwendig gewesen wäre. Die andere extreme Strategie, nie eine Ausrüstung zu kaufen, ist nicht gut, wenn wir oft Skifahren. Sei x die Anzahl an Tagen, die wir Skifahren gehen. Für $x \geq 16$ betragen die optimalen Kosten 800 €, die Strategie, stets eine Ausrüstung zu leihen, verursacht jedoch Kosten in Höhe von $50x$ €. Der Quotient von $50x$ und 800 kann beliebig groß werden.

Betrachten wir die folgende Strategie: wir leihen die ersten 16 Tage eine Ausrüstung und kaufen eine am 17. Tag, wenn uns Skifahren dann immer noch Spaß macht. Ist $x \leq 16$, so ist diese Strategie optimal. Ist $x \geq 17$, so wäre es optimal, am ersten Tag direkt eine Ausrüstung zu kaufen, wofür Kosten in Höhe von 800 € angefallen wären. Unsere Strategie verursacht

für die ersten 16 Tage Kosten in Höhe von 800 € für das Ausleihen und an Tag 17 weitere Kosten in Höhe von 800 € für den Kauf der Ausrüstung. Somit sind die Gesamtkosten unserer Strategie 1600 € und damit genau doppelt so groß wie die Kosten der optimalen Lösung. Diese Strategie garantiert also, dass wir niemals mehr als doppelt so viel ausgeben wie die optimale Strategie, die x von Anfang an kennt.

Natürlich sind die Annahmen in diesem Beispiel, dass man auch nach den ersten Tagen keine Abschätzung treffen kann, ob und wie lange Skifahren einem noch gefallen wird, nicht besonders realistisch. Dieses Problem sollte den Leser deshalb auch nur mit der Denkweise bei Online-Problemen vertraut machen. Wir betrachten im Rest dieses Kapitels realistischere Probleme.

Formal betrachten wir wieder Optimierungsprobleme Π, so wie wir sie am Anfang von Kapitel 1 definiert haben. Für unsere Zwecke genügt es, sich auf Minimierungsprobleme zu beschränken, man kann aber auch für Maximierungsprobleme eine analoge Theorie entwickeln. Der Unterschied zu einem normalen Minimierungsproblem liegt darin, dass jede Eingabe $\sigma \in \mathcal{I}_\Pi$ aus einer endlichen Folge von Ereignissen besteht. Jede Eingabe $\sigma \in \mathcal{I}_\Pi$ hat also die Form $\sigma = (\sigma_1, \ldots, \sigma_p)$, wobei p nicht fest ist, sondern von Eingabe zu Eingabe variieren kann. Ein *Online-Algorithmus* A muss auf jedes Ereignis σ_i reagieren, wobei er weder die zukünftigen Ereignisse $\sigma_{i+1}, \sigma_{i+2}, \ldots$ noch die Anzahl der zukünftigen Ereignisse kennt. Aus den Entscheidungen, die der Online-Algorithmus A trifft, muss sich am Ende eine Lösung $A(\sigma) \in \mathcal{S}_\sigma$ ergeben. Wie sich diese Lösung genau ergibt und was es überhaupt bedeutet, dass der Algorithmus auf ein Ereignis reagieren muss, hängt vom konkreten Problem ab.

Definition 2.1 (Kompetitiver Faktor)**.** *Ein Online-Algorithmus A für ein Minimierungsproblem Π erreicht einen* kompetitiven Faktor (competitive ratio) *von $r \geq 1$, wenn es eine Konstante $\tau \in \mathbb{R}$ gibt, sodass*

$$w_A(\sigma) = f_\sigma(A(\sigma)) \leq r \cdot \mathrm{OPT}(\sigma) + \tau$$

für alle Instanzen $\sigma \in \mathcal{I}_\Pi$ gilt. Wir sagen dann, dass A ein r-kompetitiver Online-Algorithmus *ist. Gilt sogar*

$$w_A(\sigma) \leq r \cdot \mathrm{OPT}(\sigma)$$

für alle Instanzen $\sigma \in \mathcal{I}_\Pi$, so ist A ein strikt r-kompetitiver Online-Algorithmus*.*

Ein Online-Algorithmus ist also dann strikt r-kompetitiv, wenn er auf jeder Eingabe höchstens r mal so viele Kosten verursacht wie ein *optimaler Offline-Algorithmus*, der die Eingabe bereits von Anfang an kennt und beliebig viel Zeit hat, sie optimal zu lösen. Die Konstante τ erlaubt man bei r-kompetitiven Algorithmen, damit Eingaben σ mit kleinem Optimum $\mathrm{OPT}(\sigma)$ keinen zu großen Einfluss auf den kompetitiven Faktor haben. Oft führen Online-Algorithmen vor dem ersten Ereignis einen Vorverarbeitungsschritt durch, der gewisse Kosten verursacht, die sich erst bei Eingaben mit vielen Ereignissen amortisieren. Um eine solche Vorgehensweise zu erlauben und den Fokus nicht auf Eingaben zu legen, die nur aus wenigen Ereignissen bestehen, haben wir die Konstante τ eingeführt, die die Kosten des Vorverarbeitungsschrittes absorbieren kann.

Die Definition eines strikt r-kompetitiven Algorithmus erinnert stark an die Definition der Approximationsgüte. Ein Unterschied ist, dass wir nicht gefordert haben, dass die Laufzeit eines Online-Algorithmus polynomiell in der Länge der Eingabe beschränkt ist. Prinzipiell sind auch exponentielle oder noch größere endliche Laufzeiten erlaubt. Die Online-Algorithmen,

die wir besprechen werden, sind jedoch alle effizient. Ein effizienter strikt r-kompetitiver Online-Algorithmus für ein Problem Π ist insbesondere ein r-Approximationsalgorithmus für Π. Umgekehrt gilt dies jedoch nicht notwendigerweise, da Approximationsalgorithmen normalerweise nicht online arbeiten, sondern zu Beginn Kenntnis der kompletten Eingabe benötigen.

Die in diesem Skript vorgestellten Online-Probleme und Algorithmen können auch im Buch von Borodin und El-Yaniv [2] und in dem Skript von Berthold Vöcking [8] nachgelesen werden.

2.1 Paging

In modernen Rechnern gibt es eine sogenannte *Speicherhierarchie*, die aus den verschiedenen Speicherarten wie zum Beispiel CPU-Cache, Hauptspeicher und Festplattenspeicher besteht. Je schneller der Speicher ist, desto weniger steht zur Verfügung, weshalb ein sinnvolles Speichermanagement benötigt wird. Wir wollen unsere Betrachtungen hier auf zwei Ebenen beschränken, also beispielsweise den CPU-Cache als schnellen und den Hauptspeicher als langsamen Speicher. Der Speicher ist in *Seiten* eingeteilt, die in der x86-Architektur eine Größe von 4kB haben. Moderne Prozessoren haben einen CPU-Cache von mehreren Megabytes, in dem sie Seiten speichern können. Zugriffe auf diesen CPU-Cache sind deutlich schneller als Zugriffe auf den Hauptspeicher. Greift ein Programm auf eine Seite zu, die nicht im CPU-Cache verfügbar ist, so spricht man von einem *Seitenfehler*. Durch ein geschicktes Speichermanagement versucht man, die Zahl der Seitenfehler so klein wie möglich zu halten.

Wir formulieren Paging wie folgt als Online-Problem. Gegeben sei eine Zahl $k \in \mathbb{N}$, die die Anzahl Seiten bezeichnet, die im Cache gespeichert werden können. Eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_p)$ besteht dann einfach aus einer endlichen Folge von Seitenzugriffen. Das heißt, jedes Ereignis σ_i ist ein Element aus $\mathbb{N}$ und entspricht der Nummer der angefragten Seite. Bei einem Ereignis σ_i entstehen dem Algorithmus keine Kosten, falls die Seite σ_i bereits im Cache gespeichert ist. Ansonsten entstehen ihm Kosten 1 und er muss die Seite σ_i in den Cache laden. Ist dieser bereits voll, so muss er eine Seite aus dem Cache auswählen, die verdrängt (d. h. aus dem Cache entfernt) wird. Da es sich um ein Online-Problem handelt, muss die Entscheidung, welche Seite aus dem Cache verdrängt wird, unabhängig von den folgenden Seitenzugriffen erfolgen.

Wir präsentieren nun einige mehr oder weniger natürliche Algorithmen für dieses Problem.

Algorithmen für das Paging-Problem

- LRU (least-recently-used): Verdränge die Seite, deren letzter Zugriff am längsten zurückliegt.
- LFU (least-frequently-used): Verdränge die Seite, auf die am seltensten zugegriffen wurde.
- FIFO (first-in-first-out): Verdränge die Seite, die sich am längsten im Cache befindet.
- LIFO (last-in-first-out): Verdränge die Seite, die als letztes in den Cache geladen wurde.
- FWF (flush-when-full): Leere den Cache komplett, sobald ein Seitenfehler auftritt.
- LFD (longest-forward-distance): Verdränge die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt.

Bei den ersten fünf dieser Algorithmen handelt es sich um Online-Algorithmen. Die Strategie LFD kann jedoch nur mit Wissen über die Zukunft ausgeführt werden und ist deshalb kein Online-Algorithmus. Tatsächlich handelt es sich bei LFD um den optimalen Offline-Algorithmus.

2.1.1 Optimaler Offline-Algorithmus

Um ein besseres Verständnis für die Offline-Variante des Paging-Problems zu erlangen, beweisen wir, dass LFD ein optimaler Offline-Algorithmus ist. Das bedeutet insbesondere, dass die Offline-Variante des Paging-Problems effizient gelöst werden kann. Diese Eigenschaft wird nicht von allen Online-Problemen geteilt, da die Offline-Varianten vieler Online-Probleme NP-hart sind.

Theorem 2.2. *LFD ist ein optimaler Offline-Algorithmus für das Paging-Problem.*

Beweis. Um das Theorem zu beweisen, betrachten wir einen beliebigen optimalen Offline-Algorithmus OPT. Wir werden das Verhalten von OPT Schritt für Schritt so modifizieren, dass sich seine Kosten durch die Modifikation nicht vergrößern und dass sich der modifizierte Algorithmus am Ende wie LFD verhält. Gelingt uns eine solche Modifikation, dann haben wir gezeigt, dass die Kosten von LFD höchstens so groß sind wie die eines optimalen Offline-Algorithmus. Das bedeutet, dass LFD ein optimaler Offline-Algorithmus ist.

Die Modifikation von OPT beruht auf folgendem Lemma.

Lemma 2.3. *Sei A ein beliebiger Offline-Algorithmus für das Paging-Problem, sei σ eine beliebige Eingabe und sei $i \in \mathbb{N}$ beliebig mit der Eigenschaft, dass Algorithmus A auf Eingabe σ mindestens i Seitenfehler verursacht. Dann existiert ein Offline-Algorithmus A_i der die folgenden Eigenschaften erfüllt:*

1. *Sei σ_t das Ereignis, das bei Algorithmus A zum i-ten Seitenfehler führt. Dann verhält sich Algorithmus A_i auf der Teilsequenz $\sigma_1, \ldots, \sigma_{t-1}$ genauso wie Algorithmus A.*
2. *Bei Ereignis σ_t verdrängt Algorithmus A_i diejenige Seite aus dem Cache, deren nächster Zugriff am weitesten in der Zukunft liegt.*
3. *Auf Eingabe σ verursacht Algorithmus A_i höchstens so viele Seitenfehler wie Algorithmus A.*

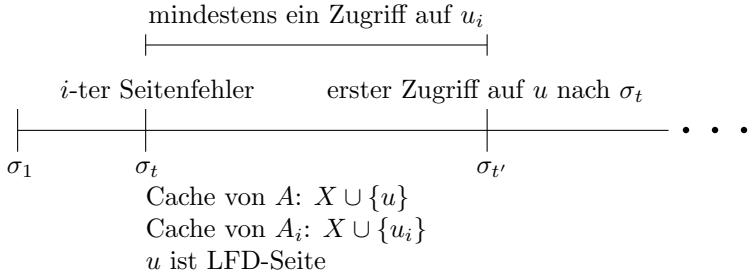
Aus diesem Lemma folgt, dass LFD ein optimaler Offline-Algorithmus ist. Um das einzusehen, betrachten wir einen optimalen Offline-Algorithmus OPT und eine beliebige Eingabe σ. Wir bezeichnen mit n die Anzahl an Seitenfehlern, die OPT auf Eingabe σ verursacht, das heißt, $\mathrm{OPT}(\sigma) = n$. Wie viele Seitenfehler verursacht LFD auf Eingabe σ? Wir wenden Lemma 2.3 nacheinander für $i = 1, \ldots, n$ an. Dadurch erhalten wir eine Sequenz $\mathrm{OPT}_1, \mathrm{OPT}_2, \ldots, \mathrm{OPT}_n$ von Algorithmen. Das Lemma besagt, dass

$$w_{\mathrm{OPT}_n}(\sigma) \leq w_{\mathrm{OPT}_{n-1}}(\sigma) \leq \ldots \leq w_{\mathrm{OPT}_1}(\sigma) \leq w_{\mathrm{OPT}}(\sigma) = n$$

gilt. Somit verursacht Algorithmus OPT_n auf der Eingabe σ höchstens so viele Seitenfehler wie der optimale Offline-Algorithmus OPT und verhält sich somit auf Eingabe σ selbst optimal. Auf der anderen Seite haben wir durch die schrittweise Transformation von OPT zu OPT_n erreicht, dass OPT_n sich auf Eingabe σ genauso wie LFD verhält. Damit ist gezeigt, dass LFD auf beliebigen Eingaben σ die gleiche Anzahl an Seitenfehlern verursacht wie der optimale Offline-Algorithmus.

Beweis von Lemma 2.3. Wir konstruieren nun einen Algorithmus A_i mit den geforderten Eigenschaften. Auf der Teilsequenz $\sigma_1, \ldots, \sigma_{t-1}$ verhalten sich die Algorithmen A und A_i identisch. Bei Ereignis σ_t verursachen beide einen Seitenfehler, verdrängen jedoch unter Umständen unterschiedliche Seiten aus dem Cache. Es sei $X \cup \{u\}$ der Cache von Algorithmus A nach Ereignis σ_t und es sei $X \cup \{u_i\}$ der Cache von Algorithmus A_i, wobei X mit $|X| = k-1$ die gemeinsamen Seiten im Cache bezeichnet. Gilt $u = u_i$, so verdrängt Algorithmus A bei Ereignis σ_t bereits die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt, und A_i kann sich auf der kompletten Eingabe σ genauso wie A verhalten.

Gilt $u \neq u_i$, so verdrängt Algorithmus A_i bei Ereignis σ_t die Seite u aus dem Cache. Somit ist u die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt. Sei $\sigma_{t'}$ mit $t' > t$ der erste Zugriff auf Seite u nach Ereignis σ_t, falls ein solcher Zugriff existiert. Diese Notationen sind in der folgenden Abbildung noch einmal dargestellt.



Verdrängt Algorithmus A auf der Teilsequenz $\sigma_{t+1}, \ldots, \sigma_{t'-1}$ Seite u nie aus dem Cache, so verhält sich Algorithmus A_i genauso wie Algorithmus A mit dem einzigen Unterschied, dass Algorithmus A_i auf dieser Teilsequenz niemals Seite u_i aus dem Cache verdrängt. Da in der Teilsequenz kein Zugriff auf Seite u erfolgt, verursacht Algorithmus A_i nicht mehr Seitenfehler als Algorithmus A. Da u die Seite ist, deren nächster Zugriff am weitesten in der Zukunft liegt, muss in dieser Teilsequenz mindestens einmal auf Seite u_i zugegriffen werden. Damit verursacht Algorithmus A_i auf dieser Teilsequenz sogar mindestens einen Seitenfehler weniger als Algorithmus A. Bei Ereignis $\sigma_{t'}$ verursacht Algorithmus A_i einen Seitenfehler und lädt Seite u in den Cache. Algorithmus A verursacht keinen Seitenfehler, da er die Seite u bereits im Cache hat. Algorithmus A_i verdrängt bei diesem Seitenfehler genau die Seite, die er zusätzlich zu Algorithmus A im Cache hat. Das heißt, nach Ereignis $\sigma_{t'}$ stimmen die Cacheinhalte wieder überein und Algorithmus A_i kann sich auf dem Rest der Sequenz wieder genauso verhalten wie Algorithmus A. Da Algorithmus A_i auf der Teilsequenz $\sigma_{t+1}, \ldots, \sigma_{t'-1}$ mindestens einen Seitenfehler gespart hat, verursacht er insgesamt nicht mehr Seitenfehler als Algorithmus A.

Gibt es in der Teilsequenz $\sigma_{t+1}, \ldots, \sigma_{t'-1}$ ein Ereignis $\sigma_{t''}$, bei dem Algorithmus A Seite u aus dem Cache verdrängt, so verhält sich Algorithmus A_i auf der Teilsequenz $\sigma_{t+1}, \ldots, \sigma_{t''-1}$ genauso wie Algorithmus A mit dem einzigen Unterschied, dass Algorithmus A_i auf dieser Teilsequenz niemals Seite u_i aus dem Cache verdrängt. Es gilt dann entweder $\sigma_{t''} = u_i$ oder Algorithmus A_i entsteht bei Ereignis $\sigma_{t''}$ ebenfalls ein Seitenfehler. In jedem Fall kann Algorithmus A_i sich so verhalten, dass nach Ereignis $\sigma_{t''}$ die Cacheinhalte von A und A_i übereinstimmen und dass die Anzahl Seitenfehler von Algorithmus A_i höchstens so groß ist wie die von Algorithmus A. Danach kann sich Algorithmus A_i auf dem Rest der Sequenz genauso verhalten wie Algorithmus A. □

Damit ist auch der Beweis des Theorems abgeschlossen. □

2.1.2 Markierungsalgorithmen

Wir betrachten nun eine spezielle Klasse von Online-Algorithmen für das Paging-Problem, sogenannte *Markierungsalgorithmen*. Um zu definieren, welche Algorithmen zu dieser Klasse gehören, zerlegen wir zunächst jede Eingabe $\sigma = (\sigma_1, \ldots, \sigma_p)$ in Phasen.

- Phase 1 ist die maximale Teilsequenz von σ, die mit der ersten Anfrage beginnt und in der auf höchstens k viele verschiedene Seiten zugegriffen wird.
- Phase $i \geq 2$ ist die maximale Teilsequenz von σ, die direkt im Anschluss an Phase $i-1$ startet und in der auf höchstens k viele verschiedene Seiten zugegriffen wird.

Beispiel: Phaseneinteilung

Für $k = 3$ ergibt sich die folgende beispielhafte Phaseneinteilung:

$$\sigma = (\overbrace{1, 2, 4, 2, 1}^{\text{Phase 1}}, \overbrace{3, 5, 2, 3, 5}^{\text{Phase 2}}, \overbrace{1, 2, 3}^{\text{Phase 3}}, \overbrace{4}^{\text{Phase 4}}).$$

Ein Markierungsalgorithmus ist ein Algorithmus, der für jede Seite implizit oder explizit verwaltet, ob sie markiert ist oder nicht. Zu Beginn jeder Phase werden alle Markierungen gelöscht und alle Seiten sind unmarkiert. Wird im Laufe einer Phase auf eine Seite zugegriffen, so wird diese Seite markiert. Ein Algorithmus ist ein Markierungsalgorithmus, wenn er niemals eine markierte Seite aus dem Cache verdrängt.

Wir zeigen zunächst, dass LRU und FWF Markierungsalgorithmen sind, und anschließend, dass jeder Markierungsalgorithmus k-kompetitiv ist.

Theorem 2.4. *LRU und FWF sind Markierungsalgorithmen.*

Beweis. Nehmen wir an, dass LRU kein Markierungsalgorithmus ist. Dann gibt es eine Eingabe σ, bei der er eine markierte Seite x in einer Phase i verdrängt. Sei σ_t der Seitenzugriff in Phase i, bei dem Seite x verdrängt wird. Da Seite x zu dem Zeitpunkt, zu dem sie verdrängt wird, markiert ist, muss es in Phase i bereits einen Zugriff auf Seite x gegeben haben. Der erste Seitenzugriff auf x in Phase i entspreche dem Ereignis $\sigma_{t'}$ mit $t' < t$. Direkt nach Seitenzugriff $\sigma_{t'}$ ist Seite x diejenige, die zuletzt angefragt wurde. Sie wird von LRU erst verdrängt, wenn anschließend auf k viele andere Seiten zugegriffen wird. Läge Ereignis σ_t in der gleichen Phase wie Ereignis $\sigma_{t'}$, so wäre in der Phase also auf $k+1$ verschiedene Seiten zugegriffen worden, was der Definition einer Phase widerspricht.

Betrachtet man die Arbeitsweise von FWF, so stellt man fest, dass dieser Algorithmus immer genau am Beginn einer neuen Phase einen Seitenfehler verursacht und den Cache komplett leert. Da innerhalb einer Phase auf maximal k verschiedene Seiten zugegriffen wird, verursacht FWF innerhalb einer Phase niemals einen Seitenfehler. Insgesamt können wir festhalten, dass FWF zu jedem Zeitpunkt exakt die markierten Seiten im Cache hält und somit ein Markierungsalgorithmus ist. □

Theorem 2.5. *Jeder Markierungsalgorithmus ist k-kompetitiv.*

Beweis. Wir gehen genau wie bei der Analyse von Approximationsalgorithmen vor. Das heißt, wir schätzen die Kosten des Markierungsalgorithmus nach oben und die Kosten des

optimalen Offline-Algorithmus nach unten ab. Sei σ eine beliebige Eingabe, und bezeichne ℓ die Anzahl Phasen, in die σ gemäß obiger Definition zerlegt wird. Wir können die Kosten eines beliebigen Markierungsalgorithmus auf Eingabe σ durch ℓk nach oben abschätzen. Der Grund ist, dass in jeder Phase i auf höchstens k verschiedene Seiten zugegriffen wird. Da eine Seite beim ersten Zugriff markiert wird und der Algorithmus markierte Seiten niemals verdrängt, verursacht er für jede von diesen Seiten maximal einen Seitenfehler in Phase i.

Wir behaupten, dass auch der optimale Offline-Algorithmus, der die Eingabe σ zu Beginn komplett kennt, mindestens $\ell - 1$ Seitenfehler auf σ verursacht, nämlich mindestens einen für jede Phase bis auf die letzte. Dazu zerlegen wir die Eingabe σ in $\ell - 1$ Teilsequenzen. Teilsequenz $i \in \{1, \ldots, \ell - 1\}$ fängt mit dem zweiten Zugriff von Phase i an und hört mit dem ersten Zugriff von Phase $i + 1$ auf (enthält diesen aber noch).

Beispiel: Einteilung in Teilsequenzen

Für obiges Beispiel mit $k = 3$ ergibt sich die folgende Einteilung in Teilsequenzen:

$$\sigma = (\overbrace{1, \underbrace{2, 4, 2, 1}_{\text{Teilseq. 1}}}^{\text{Phase 1}}, \overbrace{3, \underbrace{5, 2, 3, 5}_{\text{Teilseq. 2}}}^{\text{Phase 2}}, \overbrace{1, \underbrace{2, 3}_{\text{Teilseq. 3}}}^{\text{Phase 3}}, \overbrace{4}^{\text{Phase 4}}).$$

Sei x die erste Seite, auf die in Phase i zugegriffen wird. Dann sind zu Beginn der Teilsequenz i die Seite x und höchstens $k - 1$ weitere Seiten im Cache. In Teilsequenz i wird aber auf k verschiedene Seiten zugegriffen, die von x verschieden sind. Also gibt es für jede Teilsequenz mindestens einen Seitenfehler.

Damit haben wir gezeigt, dass

$$w_A(\sigma) \leq \ell k = k((\ell - 1) + 1) \leq k \cdot (\mathrm{OPT}(\sigma) + 1) = k \cdot \mathrm{OPT}(\sigma) + k$$

für alle Eingaben σ gilt. Da k eine Konstante ist, die unabhängig von der Eingabe σ ist, können wir $\tau = k$ setzen. Es ergibt sich damit, dass jeder Markierungsalgorithmus k-kompetitiv ist. □

Die beiden vorangegangenen Theoreme ergeben direkt das folgende Korollar.

Korollar 2.6. *LRU und FWF sind k-kompetitiv.*

Es ist eine Übungsaufgabe für den Leser, zu beweisen, dass FIFO zwar kein Markierungsalgorithmus aber dennoch k-kompetitiv ist.

2.1.3 Untere Schranken

Gibt es für einen Online-Algorithmus A keine Konstante r, für die er r-kompetitiv ist, so sagen wir, dass der Algorithmus *nicht kompetitiv* ist.

Theorem 2.7. *LFU und LIFO sind nicht kompetitiv.*

Beweis. Sei $\ell \geq 2$ eine beliebige Konstante und $k \geq 2$. Wir betrachten das Verhalten von LFU und LIFO auf der folgenden Sequenz:

$$\sigma = (1^\ell, 2^\ell, \ldots, (k-1)^\ell, (k, k+1)^{\ell-1}).$$

Es erfolgen also zunächst ℓ Zugriffe auf Seite 1, dann ℓ Zugriffe auf Seite 2 usw. Am Ende erfolgen dann jeweils $\ell - 1$ abwechselnde Zugriffe auf Seite k und Seite $k + 1$.

Es ist nicht schwierig eine Strategie anzugeben, die bei Eingabe σ genau $k + 1$ Seitenfehler verursacht. LFU und LIFO füllen bei Eingabe σ zunächst den Cache mit den Seiten 1 bis k. Wird das erste Mal auf Seite $k + 1$ zugegriffen, so wird von beiden Algorithmen Seite k aus dem Cache verdrängt. Danach wird auf Seite k zugegriffen und Seite $k + 1$ wird aus dem Cache verdrängt usw. Das bedeutet, dass bei jeder Anfrage in der Teilsequenz $(k, k+1)^\ell$ ein Seitenfehler erfolgt. Damit erzeugen LFU und LIFO mindestens $2(\ell - 1)$ Seitenfehler.

Wir müssen zeigen, dass für jede beliebige Konstante $\tau \in \mathbb{R}$ und jede Konstante r eine Eingabe σ existiert, für die

$$2(\ell - 1) > r \cdot (k + 1) + \tau$$

gilt. Dazu müssen wir ℓ für gegebene r, k und τ nur hinreichend groß wählen. Somit gibt es keine Konstante r, für die LFU oder LIFO r-kompetitiv sind. □

Wir haben gesehen, dass LRU k-kompetitiv ist. Eine natürliche Frage ist nun, ob wir einen besseren Algorithmus finden können. Das folgende Theorem beantwortet diese Frage negativ.

Theorem 2.8. *Es gibt für kein $r < k$ einen deterministischen r-kompetitiven Online-Algorithmus für das Paging-Problem.*

Beweis. Für $k = 1$ ist die Aussage trivial, da es keinen Algorithmus mit einem kompetitiven Faktor echt kleiner als 1 geben kann. Also nehmen wir im Folgenden $k \geq 2$ an. Sei A ein beliebiger deterministischer Online-Algorithmus für das Paging-Problem. Wir wollen zeigen, dass für jede beliebige Konstante $\tau \in \mathbb{R}$ und jede Konstante $r < k$ eine Eingabe σ existiert, für die

$$w_A(\sigma) > r \cdot \mathrm{OPT}(\sigma) + \tau$$

gilt. Wir konstruieren dazu eine Eingabe σ mit $k + \ell$ Ereignissen für ein beliebiges $\ell \in \mathbb{N}$. Dazu benötigen wir lediglich $k+1$ verschiedene Seiten. In den ersten k Ereignissen $\sigma_1, \ldots, \sigma_k$ wird auf k verschiedene Seiten zugegriffen. Das heißt, nach der Teilsequenz $\sigma_1, \ldots, \sigma_k$ haben sowohl Algorithmus A als auch der optimale Offline-Algorithmus genau diese k Seiten im Cache. Bei den folgenden Ereignissen $\sigma_{k+1}, \ldots, \sigma_{k+\ell}$ wird stets auf die Seite zugegriffen, die Algorithmus A nicht im Cache hat. Das bedeutet, Algorithmus A verursacht auf der so konstruierten Sequenz σ genau $k + \ell$ Seitenfehler. Der optimale Offline-Algorithmus LFD verdrängt stets die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt. Somit verursacht LFD bei Ereignis σ_{k+1} einen Seitenfehler und dann frühestens wieder bei Ereignis σ_{2k+1}. Dann verdrängt LFD wieder die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt, und damit erfolgt der nächsten Seitenfehler frühestens bei Ereignis σ_{3k+1} usw.

Insgesamt können wir die Anzahl Seitenfehler, die LFD auf der Eingabe σ verursacht, durch

$$k + \left\lceil \frac{\ell}{k} \right\rceil \leq k + 1 + \frac{\ell}{k}$$

nach oben abschätzen. Für jedes $k \geq 2$, für jedes $r < k$ und jede Konstante $\tau \in \mathbb{R}$ können wir ℓ so groß wählen, dass

$$w_A(\sigma) = k + \ell > r \cdot \left(k + 1 + \frac{\ell}{k}\right) + \tau = \frac{r}{k} \cdot \ell + r \cdot (k + 1) + \tau.$$

gilt. Also ist Algorithmus A nicht r-kompetitiv. □

Die unteren Schranken, die wir bisher für die Approximierbarkeit von Optimierungsproblemen gesehen haben, beruhten auf der Annahme P≠NP oder auf noch stärkeren Annahmen. Bei Online-Problemen ist das nicht so. Theorem 2.8 beruht auf keiner unbewiesenen Annahme.

Wir haben in Theorem 2.8 explizit von *deterministischen* Algorithmen gesprochen. Tatsächlich kann man für Paging und viele andere Online-Probleme deutlich bessere Algorithmen finden, wenn man *Randomisierung* erlaubt. Das heißt, man erlaubt dem Algorithmus, zufällige Entscheidungen zu treffen. Zwei einfache randomisierte Online-Algorithmen für Paging sind RANDOM und MARK.

Randomisierte Algorithmen für das Paging-Problem

- RANDOM: Verdränge eine uniform zufällig gewählte Seite aus dem Cache.
- MARK: Verdränge eine uniform zufällig gewählte nicht markierte Seite aus dem Cache.

Während RANDOM nur k-kompetitiv ist, ist MARK $(2H_k)$-kompetitiv, wobei H_k die k-te harmonische Zahl ist. Das bedeutet, MARK schlägt jeden deterministischen Algorithmus deutlich: statt k erreicht er einen kompetitiven Faktor von $\Theta(\log(k))$. Warum das so ist und wie genau der kompetitive Faktor für randomisierte Algorithmen überhaupt definiert ist, werden wir hier nicht besprechen. Dies ist Stoff für weiterführende Vorlesungen.

2.2 Scheduling

Scheduling ist ein wichtiges Problem in der Informatik, bei dem es darum geht, Prozessen Ressourcen zuzuteilen. In dem Modell, das wir betrachten, ist eine Menge $J = \{1, \dots, n\}$ von *Jobs* oder *Prozessen* gegeben, die auf eine Menge $M = \{1, \dots, m\}$ von *Maschinen* oder *Prozessoren* verteilt werden muss. Jeder Job $j \in J$ besitzt dabei eine *Größe* $p_j \in \mathbb{R}_{>0}$ und jede Maschine $i \in M$ hat eine *Geschwindigkeit* $s_i \in \mathbb{R}_{>0}$. Wird ein Job $j \in J$ auf einer Maschine $i \in M$ ausgeführt, so werden dafür p_j/s_i Zeiteinheiten benötigt. Ein *Schedule* $\pi : J \to M$ ist eine Abbildung, die jedem Job eine Maschine zuweist, auf der er ausgeführt wird. Wir bezeichnen mit $L_i(\pi)$ die *Ausführungszeit* von Maschine $i \in M$ in Schedule π, d. h.

$$L_i(\pi) = \frac{\sum_{j \in J : \pi(j) = i} p_j}{s_i}.$$

Der *Makespan* $C(\pi)$ eines Schedules entspricht der Ausführungszeit derjenigen Maschine, die am längsten benötigt, die ihr zugewiesenen Jobs abzuarbeiten, d. h.

$$C(\pi) = \max_{i \in M} L_i(\pi).$$

Wir betrachten den Makespan als Zielfunktion, die es zu minimieren gilt.

In der Online-Variante dieses Problems werden die Jobs nacheinander präsentiert, und sobald ein neuer Job präsentiert wird, muss unwiderruflich entschieden werden, auf welcher Maschine er ausgeführt wird. Ein Online-Algorithmus kennt also von Anfang an die Menge M der Maschinen und die zugehörigen Geschwindigkeiten. Die Größen $p_1, p_2, \dots$ werden nach und nach präsentiert und er muss Job j auf einer Maschine platzieren, sobald er die Größe p_j erfährt, ohne zu wissen wie die Größen $p_{j+1}, p_{j+2}, \dots$ aussehen und wie viele Jobs noch

kommen. Ein einmal platzierter Job kann nachträglich nicht mehr einer anderen Maschine zugewiesen werden.

Wir betrachten im Folgenden zunächst den Spezialfall *Online-Scheduling mit identischen Maschinen*. In diesem Spezialfall haben alle Maschinen $i \in M$ die Geschwindigkeit $s_i = 1$. Danach betrachten wir den Fall mit allgemeinen Geschwindigkeiten, den wir einfach als *Online-Scheduling* bezeichnen werden.

2.2.1 Identische Maschinen

Haben alle Maschinen die gleiche Geschwindigkeit, so erscheint es natürlich, jeden neuen Job der Maschine zuzuweisen, die momentan die kleinste Ausführungszeit hat. Dies ist ein Greedy-Algorithmus, den wir auch LEAST-LOADED-Algorithmus nennen werden.

Theorem 2.9. *Der* LEAST-LOADED*-Algorithmus ist für Online-Scheduling mit identischen Maschinen strikt* $(2 - 1/m)$*-kompetitiv.*

Beweis. Sei eine beliebige Eingabe für Online-Scheduling mit identischen Maschinen gegeben. Wie immer bei Approximations- und Online-Algorithmen beruht der Beweis auf einer unteren Schranke für den Makespan des optimalen Schedules π^*. In diesem Fall haben wir sogar zwei untere Schranken:

$$C(\pi^*) \geq \frac{1}{m} \sum_{j \in J} p_j \qquad \text{und} \qquad C(\pi^*) \geq \max_{j \in J} p_j.$$

Die erste Schranke basiert auf der Beobachtung, dass die durchschnittliche Ausführungszeit der Maschinen in jedem Schedule gleich $\frac{1}{m} \sum_{j \in J} p_j$ ist. Deshalb muss es in jedem Schedule (also auch in π^*) eine Maschine geben, deren Ausführungszeit mindestens diesem Durchschnitt entspricht. Die zweite Schranke basiert auf der Beobachtung, dass die Maschine, der der größte Job zugewiesen ist, eine Ausführungszeit von mindestens $\max_{j \in J} p_j$ besitzt.

Wir betrachten nun den Schedule π, den der LEAST-LOADED-Algorithmus berechnet. In diesem Schedule sei $i \in M$ eine Maschine mit größter Ausführungszeit. Es gilt also $C(\pi) = L_i(\pi)$. Es sei $j \in J$ der Job, der als letztes Maschine i hinzugefügt wurde. Zu dem Zeitpunkt, als diese Zuweisung erfolgt ist, war Maschine i die Maschine mit der kleinsten Ausführungszeit. Es waren genau die Jobs $1, \ldots, j-1$ bereits verteilt und dementsprechend muss es mindestens eine Maschine gegeben haben, deren Ausführungszeit höchstens dem Durchschnitt $\frac{1}{m} \sum_{k=1}^{j-1} p_k$ entsprach. Dementsprechend können wir den Makespan von π wie folgt abschätzen:

$$\begin{aligned}
C(\pi) = L_i(\pi) &\leq \frac{1}{m} \left(\sum_{k=1}^{j-1} p_k \right) + p_j \leq \frac{1}{m} \left(\sum_{k \in J \setminus \{j\}} p_k \right) + p_j \\
&= \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) p_j \\
&\leq \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) \cdot \max_{k \in J} p_k \\
&\leq C(\pi^*) + \left(1 - \frac{1}{m} \right) C(\pi^*) = \left(2 - \frac{1}{m} \right) \cdot C(\pi^*),
\end{aligned}$$

wobei wir die beiden oben angegebenen Schranken für $C(\pi^*)$ benutzt haben. □

Das folgende Beispiel zeigt, dass der Least-Loaded-Algorithmus im Worst-Case nicht besser als $(2 - 1/m)$-kompetitiv ist.

Untere Schranke für den Least-Loaded-Algorithmus

Sei eine Anzahl m an Maschinen vorgegeben. Wir betrachten eine Instanz mit $n = m(m-1)+1$ vielen Jobs. Die ersten $m(m-1)$ Jobs haben jeweils eine Größe von 1 und der letzte Job hat eine Größe von m. Der optimale Offline-Algorithmus verteilt die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m-1$ und platziert den letzten Job auf Maschine m. Dann haben alle Maschinen eine Ausführungszeit von genau m. Der Least-Loaded-Algorithmus hingegen verteilt die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m$ und platziert den letzten Job auf einer beliebigen Maschine i. Diese Maschine hat dann eine Ausführungszeit von $(m-1)+m = 2m-1$. Es gilt

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

und damit ist gezeigt, dass der Least-Loaded-Algorithmus auf dieser Eingabe nicht besser als $(2 - 1/m)$-kompetitiv ist.

Es ist bekannt, dass der Least-Loaded-Algorithmus für $m = 2$ und $m = 3$ der optimale Online-Algorithmus ist. Für mehr als drei Maschinen ist das aber nicht mehr der Fall. So gibt es beispielsweise für den Fall $m = 4$ einen Algorithmus, der einen kompetitiven Faktor von $1.7333 < 2 - 1/4$ erreicht. Außerdem gibt es einen Algorithmus, der für jedes m einen kompetitiven Faktor von 1.9201 erreicht, und man weiß, dass kein deterministischer Online-Algorithmus existiert, der für alle m einen Faktor besser als 1.88 erreicht. Auch dieses einfache Scheduling-Problem ist also noch nicht vollständig gelöst, und es ist nicht klar, ob es für allgemeine m einen besseren Online-Algorithmus als den o. g. 1.9201-kompetitiven gibt.

Da der Least-Loaded-Algorithmus und der o. g. 1.9201-kompetitive Algorithmus effizient ausgeführt werden können, handelt es sich insbesondere um Approximationsalgorithmen für die Offline-Variante des Scheduling-Problems mit identischen Maschinen. Diese Offline-Variante ist NP-hart, was durch eine einfache Reduktion von Partition gezeigt werden kann. Somit liegt die Frage nahe, ob es Offline-Algorithmen gibt, die einen besseren Approximationsfaktor als 1.9201 erreichen. Wir beantworten diese Frage positiv und betrachten einen einfachen Offline-Algorithmus, der eine $\frac{4}{3}$-Approximation erreicht.

Algorithmus: Longest-Processing-Time (LPT)

1. Sortiere die Jobs so, dass $p_1 \geq p_2 \geq \ldots \geq p_n$ gilt.
2. Führe den Least-Loaded-Algorithmus auf den so sortierten Jobs aus.

Es ist klar, dass der LPT-Algorithmus kein Online-Algorithmus ist, da er zunächst alle Jobs kennen muss, um sie bezüglich ihrer Größe zu sortieren. Diese Sortierung verbessert den Approximationsfaktor allerdings deutlich.

Theorem 2.10. *Der* Longest-Processing-Time*-Algorithmus ist ein $\frac{4}{3}$-Approximationsalgorithmus für Scheduling mit identischen Maschinen.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, es gäbe eine Eingabe mit m Maschinen und n Jobs mit Größen $p_1 \geq \ldots \geq p_n$, auf der der LPT-Algorithmus einen Schedule π berechnet, dessen Makespan mehr als $\frac{4}{3}$-mal so groß ist wie der Makespan des

optimalen Schedules π^*. Außerdem gehen wir davon aus, dass wir eine Instanz mit der kleinstmöglichen Anzahl an Jobs gewählt haben, auf der der LPT-Algorithmus keine $\frac{4}{3}$-Approximation liefert.

Es sei nun $i \in M$ eine Maschine, die in Schedule π die größte Ausführungszeit besitzt, und es sei $j \in J$ der letzte Job, der vom LPT-Algorithmus Maschine i zugewiesen wird. Es gilt $j = n$, da ansonsten $p_1, \ldots, p_j$ eine Eingabe mit weniger Jobs ist, auf der der LPT-Algorithmus keine $\frac{4}{3}$-Approximation liefert. Genauso wie im Beweis von Theorem 2.9 können wir argumentieren, dass es zum Zeitpunkt der Zuweisung von Job n eine Maschine mit Ausführungszeit höchstens $\frac{1}{m}\left(\sum_{j=1}^{n-1} p_j\right) \leq \mathrm{OPT}(\sigma)$ gibt. Da wir Job n der Maschine i mit der bisher kleinsten Ausführungszeit zuweisen, gilt

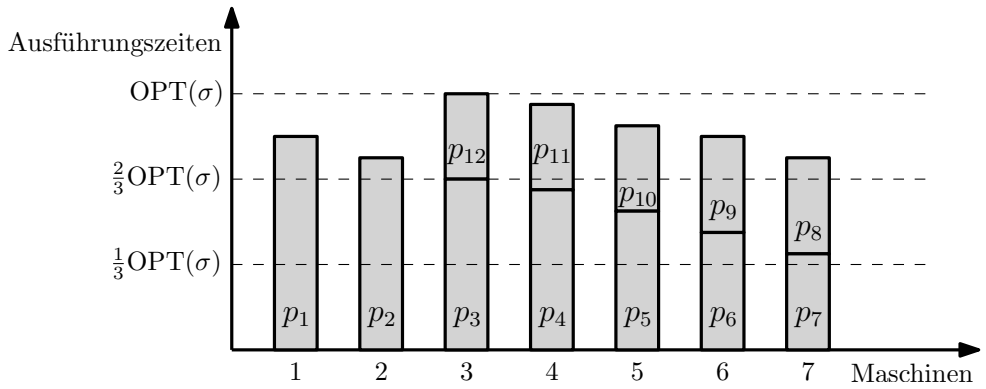
$$C(\pi) = L_i(\pi) \leq \frac{1}{m}\left(\sum_{j=1}^{n-1} p_j\right) + p_n \leq \mathrm{OPT}(\sigma) + p_n.$$

Der Makespan von π kann also nur dann größer als $\frac{4}{3}\mathrm{OPT}(\sigma)$ sein, wenn $p_n > \frac{1}{3}\mathrm{OPT}(\sigma)$ gilt. Wegen der Sortierung der Jobs gilt $p_1 \geq \ldots \geq p_n$ und somit gilt $p_j > \frac{1}{3}\mathrm{OPT}(\sigma)$ dann für alle Jobs $j \in J$.

Das bedeutet, dass in jedem optimalen Schedule jeder Maschine höchstens zwei Jobs zugewiesen sein können. Für Eingaben, die diese Bedingung erfüllen, gilt insbesondere $n \leq 2m$ und man kann einen optimalen Schedule explizit angeben:

- Jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ wird Maschine j zugewiesen.
- Jeder Job $j \in \{m+1, \ldots, n\}$ wird Maschine $2m - j + 1$ zugewiesen.

Dieser Schedule ist in folgender Abbildung dargestellt. Den einfachen Beweis, dass dies wirk-



lich ein optimaler Schedule ist, falls alle Jobs echt größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, überlassen wir dem Leser.

Wir zeigen nun, dass der optimale Schedule, den wir gerade beschrieben haben, dem Schedule entspricht, den der LPT-Algorithmus berechnet: Zunächst ist klar, dass die ersten $\min\{n, m\}$ Jobs jeweils einer eigenen Maschine zugewiesen werden, und wir deshalb ohne Beschränkung der Allgemeinheit davon ausgehen können, dass jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ Maschine j zugewiesen wird. Wir betrachten nun die Zuweisung eines Jobs $j \in \{m+1, \ldots, n\}$ und gehen induktiv davon aus, dass die Jobs aus $\{1, \ldots, j-1\}$ bereits so zugewiesen wurden wie im oben beschriebenen optimalen Schedule. Für eine Maschine $i \in M$ bezeichnen wir mit L_i die Ausführungszeit, die durch die bisher zugewiesenen Jobs verursacht wird. Im optimalen

Schedule wird Job j Maschine $2m-j+1$ zugewiesen. Deshalb gilt $L_{2m-j+1}+p_j \leq \mathrm{OPT}(\sigma)$. Ferner gilt $L_1 \geq L_2 \geq \ldots \geq L_{2m-j+1}$. Alle Maschinen $i \in \{2m-j+2, \ldots, m\}$ haben bereits zwei Jobs zugewiesen. Da alle Jobs größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, folgt für diese Maschinen $L_i + p_j > \mathrm{OPT}(\sigma)$ und damit $L_i > L_{2m-j+1}$. Damit ist gezeigt, dass Maschine $2m-j+1$ bei der Zuweisung von Job j tatsächlich eine Maschine mit kleinster Ausführungszeit ist.

Damit haben wir einen Widerspruch zu der Annahme, dass der LPT-Algorithmus auf der gegebenen Eingabe keine $\frac{4}{3}$-Approximation berechnet. Wir haben also gezeigt, dass der LPT-Algorithmus auf Eingaben, in denen alle Jobs echt größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, den optimalen Schedule berechnet. Für alle anderen Eingaben haben wir bewiesen, dass der LPT-Algorithmus eine $\frac{4}{3}$-Approximation berechnet. □

Ist der LPT-Algorithmus der bestmögliche Approximationsalgorithmus für das Scheduling-Problem mit identischen Maschinen? Die Antwort auf diese Frage lautet „nein“, denn es gibt ein PTAS für dieses Problem. Dieses Approximationsschema wird erst in weiterführenden Vorlesungen besprochen.

2.2.2 Maschinen mit Geschwindigkeiten

Wir betrachten nun den Fall, dass Maschinen verschiedene Geschwindigkeiten haben können. Als erstes betrachten wir wieder einen Greedy-Algorithmus à la Least-Loaded. Es gibt zwei Varianten dieses Algorithmus. Wird ein neuer Job präsentiert, so weisen wir ihn entweder der Maschine zu, die *derzeit* die kleinste Ausführungszeit hat, oder der Maschine, die *nach der Zuweisung des neuen Jobs* die kleinste Ausführungszeit hat.

Bei identischen Maschinen sind diese beiden Alternativen äquivalent, bei Maschinen mit verschiedenen Geschwindigkeiten sind sie es nicht mehr, wie das folgende einfache Beispiel zeigt. Sei $s_1 = 3$, $s_2 = 1$, $p_1 = p_2 = 3$ und sei der erste Jobs bereits Maschine 1 zugewiesen. Dann hat Maschine 2 derzeit die kleinste Ausführungszeit (0 statt 1), Maschine 1 hat jedoch nach der Zuweisung des neuen Jobs die kleinste Ausführungszeit (2 statt 3). Dieses Beispiel zeigt auch direkt, dass die erste Alternative, nur die derzeitige Ausführungszeit zu berücksichtigen, wenig sinnvoll ist. Wir hätten die erste Maschine in diesem Beispiel beliebig viel schneller machen können als die zweite und somit hätte der Algorithmus eine beliebig schlechte Lösung geliefert.

Es besteht also nur die Hoffnung, dass der Greedy-Algorithmus, der die Ausführungszeit nach der Zuweisung des neuen Jobs betrachtet, einen endlichen kompetitiven Faktor erreicht. Man kann jedoch zeigen, dass der kompetitive Faktor dieses Algorithmus nur $\Theta(\log m)$ beträgt.

Wir lernen jetzt einen Algorithmus kennen, der einen konstanten kompetitiven Faktor erreicht. Um diesen zu beschreiben, betrachten wir zunächst einen mit $\alpha \in \mathbb{R}_{>0}$ parametrisierten Online-Algorithmus SlowFit(α), der auf allen Eingaben σ mit $\mathrm{OPT}(\sigma) \leq \alpha$ einen Schedule π mit $C(\pi) \leq 2\alpha$ berechnet. Wäre uns bei einer Eingabe σ also bereits von Anfang an der Makespan $\mathrm{OPT}(\sigma)$ des optimalen Schedules bekannt, so könnten wir entsprechend den Algorithmus SlowFit($\mathrm{OPT}(\sigma)$) ausführen und würden so einen 2-kompetitiven Online-Algorithmus erhalten. Das Problem ist aber natürlich, dass wir zu Beginn nicht wissen, wie groß $\mathrm{OPT}(\sigma)$ tatsächlich ist. Bevor wir dieses Problem lösen, stellen wir zunächst den Algorithmus SlowFit(α) vor.

Algorithmus: SlowFit(α)

Wird ein neuer Job $j \in J$ der Größe p_j präsentiert, so weise ihn der langsamsten Maschine $i \in M$ zu, die nach der Zuweisung von Job j eine Ausführungszeit von höchsten 2α besitzt. Falls keine solche Maschine existiert, gib eine Fehlermeldung aus.

Um diesen Algorithmus formaler zu beschreiben, gehen wir davon aus, dass die Maschinen aufsteigend gemäß ihrer Geschwindigkeit sortiert sind, d. h. $s_1 \leq s_2 \leq \ldots \leq s_m$. Für $j \in J$ bezeichnen wir den (partiellen) Schedule, den SlowFit(α) für die Jobs $1, \ldots, j$ berechnet, mit π_j. Für $i \in M$ und $j \in J$ bezeichnet $L_i(\pi_j)$ dann entsprechend die Ausführungszeit von Maschine i zu dem Zeitpunkt, zu dem die Jobs $1, \ldots, j$ zugewiesen sind. Der Algorithmus SlowFit(α) weist einen Job $j \in J$ der Maschine

$$\min\{i \in M \mid L_i(\pi_{j-1}) + p_j/s_i \leq 2\alpha\}$$

zu.

Lemma 2.11. *Es sei $\alpha \in \mathbb{R}_{>0}$ beliebig, und σ sei eine beliebige Eingabe für Online-Scheduling mit $\mathrm{OPT}(\sigma) \leq \alpha$. Dann gibt der Algorithmus* SlowFit(α) *auf der Eingabe σ keine Fehlermeldung aus und berechnet einen Schedule π mit $C(\pi) \leq 2\alpha$.*

Beweis. Falls SlowFit(α) keine Fehlermeldung ausgibt, so berechnet er per Definition einen Schedule π mit $C(\pi) \leq 2\alpha$. Somit müssen wir nur zeigen, dass er auf Eingabe σ keine Fehlermeldung ausgibt. Wir führen einen Widerspruchsbeweis und gehen davon aus, dass die Eingabe σ aus n Jobs mit Größen $p_1, \ldots, p_n$ besteht und dass die Fehlermeldung erst beim letzten Job p_n ausgegeben wird. Dies können wir ohne Beschränkung der Allgemeinheit annehmen, denn wird die Fehlermeldung bereits früher ausgegeben, so können wir die nachfolgenden Jobs einfach aus der Eingabe σ entfernen.

Zunächst halten wir fest, dass $L_i(\pi_{n-1}) > \mathrm{OPT}(\sigma)$ nicht für alle Maschinen $i \in M$ gelten kann, denn ansonsten wäre

$$\sum_{j=1}^{n-1} p_j = \sum_{i \in M} (s_i \cdot L_i(\pi_{n-1})) > \sum_{i \in M} (s_i \cdot \mathrm{OPT}(\sigma)) \geq \sum_{i \in M} (s_i \cdot L_i(\pi^*)) \geq \sum_{j=1}^{n} p_j,$$

wobei π^* einen optimalen Schedule bezeichnet. Bei dieser Rechnung haben wir ausgenutzt, dass sich die Gesamtgröße der Jobs, die Maschine i in Schedule π_{n-1} zugewiesen sind, als $s_i \cdot L_i(\pi_{n-1})$ schreiben lässt. Außerdem haben wir ausgenutzt, dass die Gesamtgröße der Jobs, die Maschine i in einem optimalen Schedule π^* zugewiesen sind, nicht größer als $s_i \cdot \mathrm{OPT}(\sigma)$ sein kann.

Wir betrachten nun die schnellste Maschine $f \in M$, für die $L_f(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)$ gilt, d. h.

$$f = \max\{i \in M \mid L_i(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)\}.$$

Da m die schnellste Maschine ist, muss $f < m$ gelten. Denn wäre $L_m(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)$, dann wäre

$$L_m(\pi_{n-1}) + p_n/s_m \leq 2 \cdot \mathrm{OPT}(\sigma) \leq 2\alpha$$

und somit hätte SlowFit(α) keine Fehlermeldung ausgegeben. Wir haben bei dieser Ungleichung ausgenutzt, dass p_n/s_m eine untere Schranke für $\mathrm{OPT}(\sigma)$ ist, da m die schnellste Maschine ist.

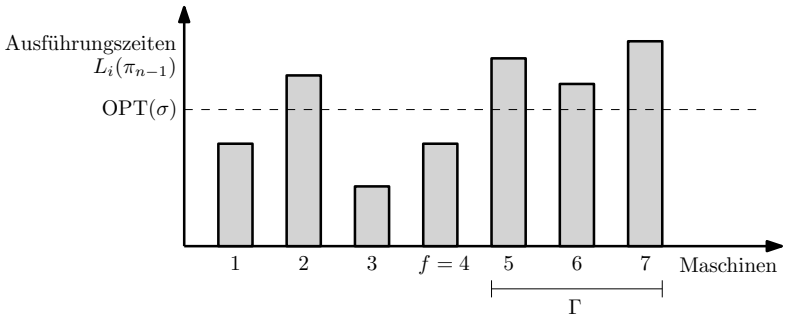


Abbildung 2.1: Illustration der Definitionen im Beweis von Lemma 2.11.

Wir setzen $\Gamma = \{i \in M \mid i > f\}$. Dann haben alle Maschinen in Γ im Schedule π_{n-1} eine Ausführungszeit größer als OPT(σ). Wegen $f < m$ gilt außerdem $\Gamma \neq \emptyset$. Diese Notationen und Definitionen sind in Abbildung 2.1 noch einmal dargestellt.

Die Gesamtgröße aller Jobs, die in Schedule π_{n-1} Maschinen aus Γ zugewiesen sind, ist

$$\sum_{i \in \Gamma} (s_i \cdot L_i(\pi_{n-1})) > \sum_{i \in \Gamma} (s_i \cdot \mathrm{OPT}(\sigma)).$$

Dies ist eine obere Schranke für die Gesamtgröße der Jobs, die in einem optimalen Schedule Maschinen aus Γ zugewiesen sind. Somit muss es einen Job $j \in J \setminus \{n\}$ mit $\pi_{n-1}(j) \in \Gamma$ geben, der in einem optimalen Schedule einer Maschine $i \notin \Gamma$, also $i \leq f$, zugewiesen wird. Für diesen Job j und diese Maschine i muss demnach $p_j/s_i \leq \mathrm{OPT}(\sigma)$ gelten. Wegen $i \leq f$ und der Sortierung der Maschinen gemäß ihrer Geschwindigkeit muss somit auch $p_j/s_f \leq \mathrm{OPT}(\sigma)$ gelten. Betrachten wir nun den Zeitpunkt, zu dem Job j eingefügt wurde. Zu diesem Zeitpunkt waren die Jobs $1, \ldots, j-1$ gemäß des partiellen Schedules π_{j-1} verteilt. Es gilt

$$L_f(\pi_{j-1}) + p_j/s_f \leq L_f(\pi_{n-1}) + p_j/s_f \leq \mathrm{OPT}(\sigma) + \mathrm{OPT}(\sigma) \leq 2\alpha.$$

Somit hätte zu diesem Zeitpunkt Job j Maschine f zugewiesen werden dürfen. Stattdessen wurde er aber einer Maschine $i' \in \Gamma$ zugewiesen. Wegen $f < i'$ ist dies ein Widerspruch zur Definition des Algorithmus. Damit ist gezeigt, dass der Algorithmus auf der Eingabe σ keine Fehlermeldung ausgibt. □

Wir müssen nun die Frage klären, wie wir den Algorithmus SlowFit(α) nutzen können, ohne das richtige α zu kennen. Wir werden dazu einfach den Makespan der optimalen Lösung schätzen und unsere Schätzung im Laufe des Algorithmus gegebenenfalls anpassen. Wir fangen mit einer Schätzung α an und jedes Mal, wenn SlowFit(α) eine Fehlermeldung ausgibt, verdoppeln wir unsere Schätzung für α. Formal ist der Algorithmus SlowFit wie folgt definiert.

Algorithmus: SlowFit

1. Setze $\alpha_0 = p_1/s_m$.
2. Beginne mit Phase $k = 0$.
3. Für jeden Job $j = 1, \ldots, n$ führe folgende Anweisungen durch:
4. Versuche, Job j mit SlowFit(α_k) einer Maschine zuzuweisen.
   Ignoriere dabei alle Jobs, die in früheren Phasen zugewiesen wurden.
5. Falls SlowFit(α_k) keine Fehlermeldung ausgibt, so übernimm die Zuweisung.
6. Ansonsten erhöhe k um eins, setze $\alpha_k = 2^k \alpha_0$ und gehe zurück zu Schritt 4.

Theorem 2.12. *Der Algorithmus* SlowFit *ist ein strikt* 8*-kompetitiver Algorithmus für Online-Scheduling.*

Beweis. Es sei σ eine beliebige Eingabe, und der Algorithmus SlowFit durchlaufe die Phasen $0, 1, \ldots, h$. Wir bezeichnen dann mit σ_k die Teilsequenz der Jobs, die in Phase k zugewiesen werden. Wir können mit Lemma 2.11 einen Rückschluss über den Makespan des optimalen Schedules ziehen. Gilt $h = 0$, so beträgt der Makespan des Schedules π, den der Algorithmus berechnet, höchstens $2\alpha_0$. Da α_0 eine untere Schranke für $\mathrm{OPT}(\sigma)$ ist, liefert der Algorithmus in diesem Falle sogar eine 2-Approximation. Wir betrachten nun den Fall $h > 0$. Dazu betrachten wir Phase $h-1$ und den ersten Job $j \in J$, der zu Phase h gehört. Da in Phase $h-1$ alle Jobs ignoriert werden, die bereits in früheren Phasen zugewiesen wurden, erzeugt der Algorithmus SlowFit(α_{h-1}) gemäß Lemma 2.11 nur dann bei der Einfügung von Job j eine Fehlermeldung, wenn für die Teilsequenz $\sigma_{h-1}j$ (alle Jobs aus Phase $h-1$ gefolgt von Job j) gilt

$$\mathrm{OPT}(\sigma_{h-1}j) > \alpha_{h-1} = 2^{h-1}\alpha_0.$$

Nun schätzen wir den Makespan des Schedules π ab, den der Algorithmus SlowFit berechnet. Dazu betrachten wir alle Phasen $k = 0, \ldots, h$ und addieren jeweils den Makespan auf, der durch die Jobs verursacht wird, die in Phase k zugewiesen werden. Aus Lemma 2.11 folgt, dass der Makespan, der durch die Jobs in Phase k verursacht wird, höchstens $2\alpha_k$ ist. Damit ergibt sich insgesamt

$$C(\pi) \leq \sum_{k=0}^{h} 2\alpha_k = 2\sum_{k=0}^{h} 2^k\alpha_0 = 2(2^{h+1}-1)\alpha_0 \leq 2^{h+2}\alpha_0.$$

Damit ist gezeigt, dass

$$C(\pi) \leq 2^{h+2}\alpha_0 = 8 \cdot 2^{h-1}\alpha_0 < 8 \cdot \mathrm{OPT}(\sigma_{h-1}j) \leq 8 \cdot \mathrm{OPT}(\sigma)$$

gilt. □

Der Algorithmus SlowFit ist nicht der beste bekannte Online-Algorithmus für das betrachtete Scheduling-Problem. Es ist ein Algorithmus bekannt, der einen kompetitiven Faktor von 5.828 erreicht, und es ist bekannt, dass kein deterministischer Online-Algorithmus existieren kann, der einen Faktor kleiner als 2.438 erreicht. Auch dieses grundlegende Scheduling-Problem ist also noch nicht vollständig gelöst.

Kapitel 3

Lineare Programmierung

Wir haben bereits einige allgemeine Techniken zum Entwurf von Algorithmen wie zum Beispiel dynamische Programmierung und Greedy-Algorithmen kennengelernt. Dennoch mussten wir bei jedem neuen Problem separat prüfen, ob und wie genau sich diese Methoden anwenden lassen. Wir werden nun *lineare Programmierung* kennenlernen. Dabei handelt es sich um eine sehr allgemeine Technik zur Lösung von Optimierungsproblemen, die noch standardisierter ist als die Techniken, die wir bisher gesehen haben.

Ein *lineares Programm (LP)* ist ein Optimierungsproblem, bei dem es darum geht, die optimalen Werte für d reelle *Variablen* $x_1, \ldots, x_d \in \mathbb{R}$ zu bestimmen. Dabei gilt es, eine lineare *Zielfunktion*

$$c_1 x_1 + \ldots + c_d x_d \tag{3.1}$$

für gegebene Koeffizienten $c_1, \ldots, c_d \in \mathbb{R}$ zu minimieren oder zu maximieren. Es müssen dabei m lineare *Nebenbedingungen* eingehalten werden. Für jedes $i \in \{1, \ldots, m\}$ sind Koeffizienten $a_{i1}, \ldots, a_{id} \in \mathbb{R}$ und $b_i \in \mathbb{R}$ gegeben, und eine Belegung der Variablen ist nur dann gültig, wenn sie die Nebenbedingungen

$$\begin{aligned} a_{11} x_1 + \ldots + a_{1d} x_d &\leq b_1 \\ &\vdots \\ a_{m1} x_1 + \ldots + a_{md} x_d &\leq b_m \end{aligned} \tag{3.2}$$

einhält. Statt dem Relationszeichen $\leq$ erlauben wir prinzipiell auch Nebenbedingungen mit dem Relationszeichen $\geq$. Da man das Relationszeichen einer Nebenbedingung aber einfach dadurch umdrehen kann, dass man beide Seiten mit -1 multipliziert, gehen wir im Folgenden ohne Beschränkung der Allgemeinheit davon aus, dass alle Nebenbedingungen das Relationszeichen $\leq$ enthalten.

Um die Notation zu vereinfachen, definieren wir $x^{\mathrm{T}} = (x_1, \ldots, x_d)$ und $c^{\mathrm{T}} = (c_1, \ldots, c_d)$. Damit sind $x \in \mathbb{R}^d$ und $c \in \mathbb{R}^d$ Spaltenvektoren und wir können die Zielfunktion (3.1) als Skalarprodukt $c \cdot x$ schreiben. Außerdem definieren wir $A \in \mathbb{R}^{m \times d}$ als die Matrix mit den Einträgen a_{ij} und wir definieren $b^{\mathrm{T}} = (b_1, \ldots, b_m) \in \mathbb{R}^m$. Dann entspricht jede Zeile der Matrix einer Nebenbedingung und wir können die Nebenbedingungen (3.2) als $Ax \leq b$ schreiben. Zusammengefasst können wir ein lineares Programm also wie folgt beschreiben.

Lineares Programm

Eingabe:	$c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times d}$
Lösungen:	alle $x \in \mathbb{R}^d$ mit $Ax \leq b$
Zielfunktion:	minimiere/maximiere $c \cdot x$

Wir präsentieren nun einige Beispiele, die den Leser davon überzeugen sollen, dass man viele Probleme in Form eines linearen Programms ausdrücken kann.

Hobbygärtner

Ein Hobbygärtner besitzt 100 m^2 Land, auf dem er Blumen und Gemüse anbauen möchte.

- Nur 60 m^2 des Landes sind für den Anbau von Blumen und Gemüse geeignet. Auf den restlichen 40 m^2 kann nur Gemüse angebaut werden.
- Er hat ein Budget in Höhe von 720 €, das er investieren kann. Das Pflanzen und Düngen der Blumen kostet ihn 9 € pro Quadratmeter; bei Gemüse sind es nur 6 € pro Quadratmeter.
- Der Erlös, den er pro Quadratmeter für Blumen erzielt, beträgt 20 €; bei Gemüse sind es nur 10 € pro Quadratmeter.

Auf wie vielen Quadratmetern sollte der Hobbygärtner Blumen und auf wie vielen sollte er Gemüse anbauen, wenn er seinen Gewinn maximieren möchte?

Um dieses Problem als LP zu formulieren, führen wir zwei Variablen $x_B \in \mathbb{R}$ und $x_G \in \mathbb{R}$ ein, die angeben, auf wie vielen Quadratmetern Blumen bzw. Gemüse angebaut werden. Dann können wir die Nebenbedingungen wie folgt formulieren:

$x_B \geq 0, x_G \geq 0$	(keine negative Anbaufläche)
$x_B + x_G \leq 100$	(maximal 100 m^2)
$x_B \leq 60$	(maximal 60 m^2 Blumen)
$9x_B + 6x_G \leq 720$	(Budget von 720 €)

Der Gewinn ist die Differenz von Erlös und Ausgaben. Somit soll die folgende Zielfunktion maximiert werden:

$$(20 - 9)x_B + (10 - 6)x_G = 11x_B + 4x_G.$$

Wir können auch einige Probleme, die wir bereits kennen, als lineares Programm formulieren.

Maximaler Fluss

Gegeben sei ein Flussnetzwerk $G = (V, E)$ mit Quelle $s \in V$ und Senke $t \in V$ und einer *Kapazitätsfunktion* $c : E \to \mathbb{N}_0$. Das Problem, einen maximalen Fluss von s nach t zu berechnen, können wir als LP darstellen, indem wir für jede Kante $e \in E$ eine Variable $x_e \in \mathbb{R}$ einführen, die dem Fluss auf Kante e entspricht. Wir wollen dann die Zielfunktion

$$\sum_{e=(v,t)} x_e - \sum_{e=(t,v)} x_e$$

unter den folgenden Bedingungen maximieren.

$\forall e \in E : x_e \geq 0$	(Fluss nicht negativ)
$\forall e \in E : x_e \leq c(e)$	(Fluss nicht größer als Kapazität)
$\forall v \in V \setminus \{s, t\} : \sum_{e=(u,v)} x_e - \sum_{e=(v,u)} x_e = 0$	(Flusserhaltung)

Einführungen in lineare Programmierung mit weiteren Details finden sich unter anderem im Skript von Berthold Vöcking [7] und in den Büchern von Norbert Blum [1] und Christos Papadimitriou und Kenneth Steiglitz [3].

3.1 Grundlagen

3.1.1 Kanonische Form und Gleichungsform

Um Algorithmen zur Lösung von linearen Programmen zu beschreiben, ist es hilfreich, die folgenden beiden Formen zu betrachten. Dabei sei $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times d}$, und $x \in \mathbb{R}^d$ sei ein Vektor aus d Variablen.

kanonische Form	*Gleichungsform*
$\min c \cdot x$	$\min c \cdot x$
$Ax \leq b$	$Ax = b$
$x \geq 0$	$x \geq 0$

Wir können uns ohne Beschränkung der Allgemeinheit auf eine dieser beiden Formen konzentrieren, denn jedes lineare Programm kann in ein äquivalentes lineares Programm in kanonischer Form oder in Gleichungsform transformiert werden. Dazu müssen nur die folgenden Operationen in der richtigen Reihenfolge angewendet werden. Wir bezeichnen bei der Beschreibung dieser Operationen mit $a_i = (a_{i1}, \ldots, a_{id})^{\mathrm{T}} \in \mathbb{R}^d$ den Vektor, der der i-ten Zeile der Matrix A entspricht. Dann können wir die i-te Nebenbedingung in kanonischer Form beispielsweise kurz als $a_i \cdot x \leq b_i$ schreiben.

- Haben wir ein lineares Programm, in dem eine Zielfunktion $c \cdot x$ maximiert werden soll, so können wir dies äquivalent dadurch ausdrücken, dass die Zielfunktion $-c \cdot x$, minimiert werden soll.
- Für jede Variable x_i, für die es keine Nebenbedingung $x_i \geq 0$ gibt, können wir zwei Variablen x_i' und x_i'' mit Nebenbedingungen $x_i' \geq 0$ und $x_i'' \geq 0$ einführen und jedes Vorkommen von x_i in Zielfunktion und Nebenbedingungen durch $x_i' - x_i''$ substituieren.
- Bei Nebenbedingungen der Form $a_i \cdot x \geq b_i$ können wir das Relationszeichen umdrehen, indem wir beide Seiten der Ungleichung mit -1 multiplizieren.
- Eine Gleichung $a_i \cdot x = b_i$ kann durch die beiden Ungleichungen $a_i \cdot x \leq b_i$ und $a_i \cdot x \geq b_i$ ersetzt werden.
- Eine Ungleichung $a_i \cdot x \leq b_i$ können wir in eine Gleichung umwandeln, indem wir eine *Schlupfvariable* s_i mit Nebenbedingung $s_i \geq 0$ einführen und die Ungleichung durch die Gleichung $s_i + a_i \cdot x = b_i$ ersetzen.

3.1.2 Geometrische Interpretation

Bevor wir einen Algorithmus beschreiben, der für ein gegebenes lineares Programm eine optimale Belegung der Variablen berechnet, ist es zunächst hilfreich, eine geometrische Interpretation von linearen Programmen zu erarbeiten. Dazu gehen wir davon aus, dass ein lineares Programm in kanonischer Form gegeben ist.

Nebenbedingungen

Jede Variablenbelegung $x \in \mathbb{R}^d$ kann als ein Punkt in dem d-dimensionalen Raum $\mathbb{R}^d$ aufgefasst werden. Die Nebenbedingungen bestimmen dann, welche Punkte zulässige Lösungen des linearen Programms sind. Eine Gleichung der Form $a_i \cdot x = b_i$ definiert eine *affine Hyperebene* $\{x \in \mathbb{R}^d \mid a_i \cdot x = b_i\}$. Dabei handelt es sich um die Verallgemeinerung einer normalen Ebene auf beliebige Dimensionen $d \in \mathbb{N}$. Für $d = 3$ entsprechen Hyperebenen normalen Ebenen; für $d = 2$ sind es beispielsweise Geraden. Jede solche affine Hyperebene definiert den *abgeschlossenen Halbraum*

$$\mathcal{H}_i = \{x \in \mathbb{R}^d \mid a_i \cdot x \leq b_i\}.$$

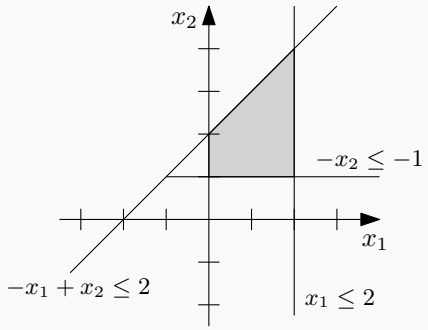
Eine Variablenbelegung $x \in \mathbb{R}^d$ erfüllt genau dann Nebenbedingung i, wenn $x \in \mathcal{H}_i$ gilt. Insgesamt ist eine Variablenbelegung $x \in \mathbb{R}^d$ in einem linearen Programm in kanonischer Form genau dann gültig, wenn $x \in \mathcal{P} := \mathcal{H}_1 \cap \ldots \cap \mathcal{H}_m \cap \mathbb{R}^d_{\geq 0}$ gilt, wobei durch den Schnitt mit $\mathbb{R}^d_{\geq 0}$ der Bedingung $x \geq 0$ Rechnung getragen wird.

Beispiel

In der folgenden Abbildung ist ein lineares Programm mit den Nebenbedingungen

$$\begin{aligned} x_1 &\geq 0, \quad & x_2 &\geq 0, \\ x_1 &\leq 2, \quad & -x_2 &\leq -1, \\ -x_1 + x_2 &\leq 2 \end{aligned}$$

dargestellt. Der gültige Bereich $\mathcal{P}$ ist in der Abbildung grau hervorgehoben.



Wir können die Menge $\mathbb{R}^d_{\geq 0}$ selbst als einen Schnitt von d Halbräumen darstellen:

$$\mathbb{R}^d_{\geq 0} = \{x \in \mathbb{R}^d \mid -x_1 \leq 0\} \cap \ldots \cap \{x \in \mathbb{R}^d \mid -x_d \leq 0\}.$$

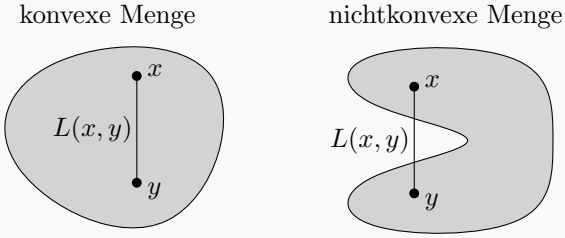
Somit können wir insgesamt festhalten, dass $\mathcal{P}$ der Schnitt von endlich vielen Halbräumen ist. Einen solchen Schnitt nennt man *konvexes Polyeder*, und wir werden das zu einem linearen Programm gehörige konvexe Polyeder auch als sein *Lösungspolyeder* bezeichnen. Wir sagen, dass ein lineares Programm *zulässig* ist, wenn sein Lösungspolyeder nicht leer ist.

Wie der Name schon sagt, ist jedes konvexe Polyeder $\mathcal{P}$ *konvex*. Das bedeutet, dass für zwei beliebige Punkte $x, y \in \mathcal{P}$ auch alle Punkte auf der direkten Verbindungslinie zwischen x und y zum konvexen Polyeder $\mathcal{P}$ gehören. Formal ausgedrückt ist eine Menge $X \subseteq \mathbb{R}^d$ konvex, wenn für jedes Paar $x, y \in X$ von Punkten

$$L(x, y) := \{\lambda x + (1 - \lambda)y \mid \lambda \in [0, 1]\} \subseteq X.$$

gilt. Hierbei ist $L(x, y)$ die Verbindungslinie zwischen x und y.

Beispiel



Lemma 3.1. *Jedes konvexe Polyeder $\mathcal{P}$ ist konvex.*

Beweis. Sei $\mathcal{P}$ der Durchschnitt der abgeschlossenen Halbräume $\mathcal{H}_1, \ldots, \mathcal{H}_n$. Zunächst beweisen wir, dass jeder abgeschlossene Halbraum konvex ist. Seien $u \in \mathbb{R}^d$ und $w \in \mathbb{R}$ beliebig und seien x und y beliebige Punkte aus dem abgeschlossenen Halbraum $\mathcal{H} = \{x \mid u \cdot x \leq w\}$. Für jedes $\lambda \in [0, 1]$ gehört dann auch der Punkt $\lambda x + (1 - \lambda)y$ zu $\mathcal{H}$, denn

$$u \cdot (\lambda x + (1 - \lambda)y) = \lambda(u \cdot x) + (1 - \lambda)(u \cdot y) \leq \lambda w + (1 - \lambda)w = w.$$

Somit gehören alle Punkte aus $L(x, y)$ zum abgeschlossenen Halbraum $\mathcal{H}$, und damit ist bewiesen dass jeder beliebige abgeschlossene Halbraum konvex ist.

Nun müssen wir nur noch zeigen, dass der Schnitt zweier konvexer Mengen wieder konvex ist. Durch mehrfache Anwendung dieser Beobachtung folgt dann direkt das Lemma. Seien also $X \subseteq \mathbb{R}^d$ und $Y \subseteq \mathbb{R}^d$ konvexe Mengen, und seien $x, y \in X \cap Y$ beliebig. Da die Mengen X und Y konvex sind, gilt $L(x, y) \subseteq X$ und $L(x, y) \subseteq Y$. Dementsprechend gilt auch $L(x, y) \subseteq X \cap Y$ und somit ist $X \cap Y$ konvex. □

Sei ein lineares Programm in kanonischer Form mit Lösungspolyeder $\mathcal{P}$ gegeben. Die Tatsache, dass bei einem linearen Programm die erlaubten Variablenbelegungen eine konvexe Menge bilden, hat eine wichtige Konsequenz für die optimale Variablenbelegung. Wir sagen, eine Variablenbelegung $x \in \mathcal{P}$ ist *lokal optimal*, wenn es ein $\varepsilon > 0$ gibt, für das es kein $y \in \mathcal{P}$ mit $\|x - y\| \leq \varepsilon$ und $c \cdot y < c \cdot x$ gibt. Hierbei bezeichnet $\|x - y\|$ den euklidischen Abstand zwischen x und y, also $\|x - y\| = \sqrt{(x_1 - y_1)^2 + \ldots + (x_d - y_d)^2}$. Eine Variablenbelegung ist also lokal optimal, wenn es eine Umgebung um sie herum gibt, in der es keine echt bessere zulässige Variablenbelegung gibt.

Theorem 3.2. *Sei ein lineares Programm in kanonischer Form mit Lösungspolyeder $\mathcal{P}$ gegeben und sei $x \in \mathcal{P}$ eine lokal optimale Variablenbelegung. Dann ist x auch global optimal, d. h. es gibt kein $y \in \mathcal{P}$ mit $c \cdot y < c \cdot x$.*

Beweis. Angenommen, es gäbe ein $y \in \mathcal{P}$ mit $c \cdot y < c \cdot x$. Da das Lösungspolyeder $\mathcal{P}$ konvex ist, liegt auch jeder Punkt auf der Verbindungslinie $L(x, y)$ in $\mathcal{P}$. Sei $z = \lambda x + (1 - \lambda)y$ mit $\lambda \in [0, 1)$ ein solcher Punkt. Dann können wir den Wert der Zielfunktion an der Stelle z schreiben als

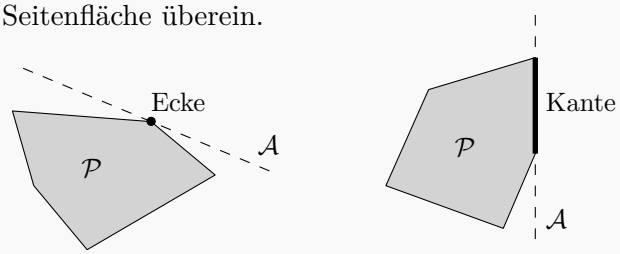
$$c \cdot z = c \cdot (\lambda x + (1 - \lambda)y) = \lambda(c \cdot x) + (1 - \lambda)(c \cdot y) < c \cdot x,$$

wobei wir bei der letzten Ungleichung $\lambda < 1$ und $c \cdot y < c \cdot x$ ausgenutzt haben. Das bedeutet, jeder Punkt $z \in L(x, y)$ mit $z \neq x$ ist eine echt bessere Variablenbelegung als x. Da jeder solche Punkt zu $\mathcal{P}$ gehört und damit gültig ist, gibt es für jedes $\varepsilon > 0$ einen Punkt $z \in \mathcal{P}$ mit $c \cdot z < c \cdot x$ und $\|x - z\| \leq \varepsilon$. Damit ist x im Widerspruch zur Annahme nicht lokal optimal. □

Wir betrachten nun den Rand von konvexen Polyedern genauer. Sei $\mathcal{P} \subseteq \mathbb{R}^d$ ein beliebiges konvexes Polyeder und sei eine Hyperebene $\mathcal{A} = \{x \in \mathbb{R}^d \mid u \cdot x = w\}$ mit beliebigem $u \in \mathbb{R}^d$ und $w \in \mathbb{R}$ gegeben. Ist $\mathcal{P} \cap \mathcal{A} \neq \emptyset$ und gilt $\mathcal{P} \subseteq \mathcal{H}$, wobei $\mathcal{H} = \{x \in \mathbb{R}^d \mid u \cdot x \leq w\}$, den zu $\mathcal{A}$ gehörigen Halbraum bezeichnet, so sagen wir, dass $\mathcal{A}$ eine *stützende Hyperebene* von $\mathcal{P}$ ist. Die Menge $\mathcal{P} \cap \mathcal{A}$ nennen wir *Fläche von* $\mathcal{P}$. Eine nulldimensionale Fläche heißt auch *Ecke von* $\mathcal{P}$, eine eindimensionale Fläche heißt auch *Kante von* $\mathcal{P}$ und eine Fläche der Dimension $d-1$ heißt auch *Seitenfläche von* $\mathcal{P}$.

Beispiel

Die folgende Abbildung zeigt ein zweidimensionales Beispiel. Für $d = 2$ stimmen die Begriffe Kante und Seitenfläche überein.



Zielfunktion

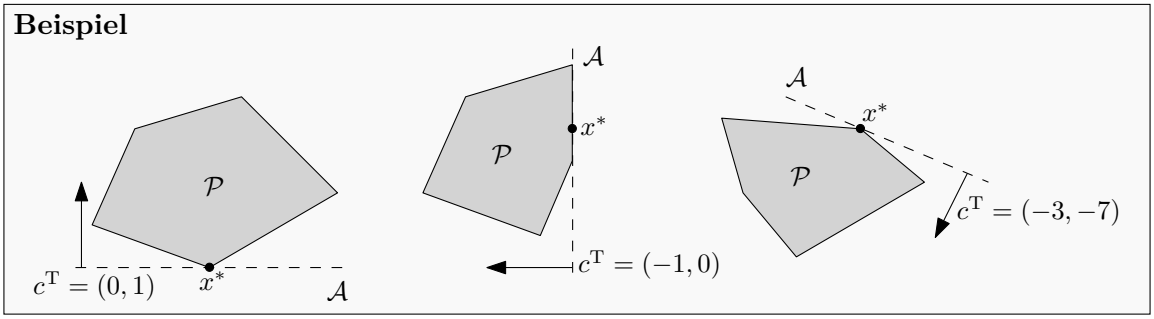
Wir bezeichnen ein lineares Programm als *unbeschränkt*, wenn der zu minimierende Zielfunktionswert innerhalb des Lösungspolyeders $\mathcal{P}$ beliebig klein werden kann, d. h. wenn für alle $z \in \mathbb{R}$ eine Variablenbelegung $x \in \mathcal{P}$ mit $c \cdot x \leq z$ existiert. Ansonsten heißt das lineare Programm *beschränkt*.

Bisher haben wir lediglich eine geometrische Interpretation der Nebenbedingungen erarbeitet. Wir können auch die Zielfunktion geometrisch interpretieren. Dazu überlegen wir uns zunächst, welche Teilmengen von $\mathbb{R}^d$ den gleichen Zielfunktionswert bezüglich einer linearen Zielfunktion haben. Sei $c \cdot x$ eine beliebige lineare Zielfunktion und sei $w \in \mathbb{R}$ beliebig. Die Menge

$$\{x \in \mathbb{R}^d \mid c \cdot x = w\}$$

bildet eine affine Hyperebene mit Normalenvektor c. Der Vektor c steht also senkrecht auf der Hyperebene, die die Punkte mit gleichem Zielfunktionswert enthält. Wir können nun wie folgt grafisch eine optimale Lösung eines linearen Programms bestimmen.

1. Finde ein $w \in \mathbb{R}$, für das der Schnitt der Hyperebene $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ mit dem Lösungspolyeder nicht leer ist.
2. Verschiebe die Hyperebene $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ solange parallel in Richtung $-c$ wie sie einen nichtleeren Schnitt mit dem Lösungspolyeder $\mathcal{P}$ besitzt. Das ist äquivalent dazu, den Wert w solange zu verringern wie $\{x \in \mathbb{R}^d \mid c \cdot x = w\} \cap \mathcal{P} \neq \emptyset$ gilt.
3. Terminiert der zweite Schritt nicht, da jede betrachtete Hyperebene einen nichtleeren Schnitt mit $\mathcal{P}$ besitzt, so ist das lineare Programm unbeschränkt. Ansonsten sei $\mathcal{A} = \{x \in \mathbb{R}^d \mid c \cdot x = w\}$ die letzte Hyperebene, für die $\mathcal{A} \cap \mathcal{P} \neq \emptyset$ gilt. Dann ist $\mathcal{A}$ eine stützende Hyperebene von $\mathcal{P}$. Jeder Punkt $x^* \in \mathcal{A} \cap \mathcal{P}$ stellt eine optimale Variablenbelegung des linearen Programms dar, da $\mathcal{P} \subseteq \{x \in \mathbb{R}^d \mid c \cdot x \geq w\}$.



An dieser Stelle weisen wir darauf hin, dass man mit dem oben beschriebenen grafischen Verfahren zwar jedes lineare Programm optimal lösen kann, dass es sich jedoch um keinen Algorithmus handelt, der so ohne Weiteres implementiert werden kann. In höheren Dimensionen ist es beispielsweise bereits nicht trivial im ersten Schritt ein geeignetes $w \in \mathbb{R}$ zu finden, für das der Schnitt von $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ und $\mathcal{P}$ nicht leer ist.

3.1.3 Algebraische Interpretation

Die kanonische Form eignet sich gut zur anschaulichen Darstellung von linearen Programmen. Bei dem Entwurf von Algorithmen ist es jedoch einfacher, lineare Programme in Gleichungsform zu betrachten. In diesen Abschnitt gehen wir deshalb davon aus, dass ein solches lineares Programm gegeben ist, in dem die Zielfunktion $c \cdot x$ unter den Nebenbedingungen $Ax = b$ und $x \geq 0$ minimiert werden soll. Dabei seien $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times d}$ und $x \in \mathbb{R}^d$ sei ein Vektor, der aus d Variablen besteht. Es bezeichne $\mathcal{P}$ das zugehörige Lösungspolyeder.

Wir haben bereits gesehen, dass eine Nebenbedingung der Form $a_i \cdot x = b_i$ auch durch die beiden Ungleichungen $a_i \cdot x \leq b_i$ und $a_i \cdot x \geq b_i$ ausgedrückt werden kann. Das bedeutet, dass auch die Nebenbedingungen in einem linearen Programm in Gleichungsform ein Lösungspolyeder bestimmen. Alle Begriffe, die wir im vorangegangenen Abschnitt für Lösungspolyeder definiert haben (stützende Hyperebene, Fläche, Ecke, ...), können wir also weiterhin benutzen.

Die erste Schwierigkeit beim Lösen von linearen Programmen besteht darin, dass das Lösungspolyeder im Allgemeinen eine unendliche Menge ist. Wir können also nicht einfach alle zulässigen Variablenbelegungen testen und die beste wählen. Nach der Diskussion des grafischen Lösungsverfahrens im vorangegangenen Abschnitt mag der Leser aber bereits die Intuition gewonnen haben, dass es stets eine Ecke des Lösungspolyeders gibt, die eine optimale Variablenbelegung beschreibt. Wir zeigen als erstes, dass dies tatsächlich so ist und dass es nur endlich viele Ecken gibt. Daraus ergibt sich ein (nicht effizienter) Algorithmus zur Lösung von beschränkten linearen Programmen.

Wir benötigen zunächst zwei Definitionen.

- Ein Punkt $x \in \mathcal{P}$ heißt *Extrempunkt*, wenn es kein $y \in \mathbb{R}^d$ mit $y \neq 0$ gibt, für das $x + y \in \mathcal{P}$ und $x - y \in \mathcal{P}$ gelten.
- Für $B \subseteq \{1, \ldots, d\}$ bezeichnen wir mit A_B die Teilmatrix von A, die aus genau den Spalten von A besteht, deren Index in B enthalten ist. Analog bezeichnen wir für $x \in \mathbb{R}^d$ mit x_B den $|B|$-dimensionalen Subvektor von x, der nur aus den Komponenten aus B besteht. Sei nun $x \in \mathbb{R}^d$ und $B(x) = \{i \in \{1, \ldots, d\} \mid x_i \neq 0\}$. Anstatt $A_{B(x)}$ schreiben wir im Folgenden kurz A_x. Sind die Spalten von A_x linear unabhängig, so heißt x *Basislösung*. Gilt zusätzlich $x \in \mathcal{P}$, so nennen wir x *zulässige Basislösung*. Für

jede Basislösung x gilt $|B(x)| \leq m$, da der Rang der Matrix A durch m nach oben beschränkt ist.

Lemma 3.3. *Sei $\mathcal{P}$ das Lösungspolyeder eines linearen Programms in Gleichungsform und sei $x \in \mathcal{P}$. Dann sind die folgenden drei Aussagen äquivalent.*

a) x ist eine Ecke von $\mathcal{P}$.

b) x ist ein Extrempunkt von $\mathcal{P}$.

c) x ist eine zulässige Basislösung.

Beweis. Wir beweisen das Lemma mit einem Ringschluss.

a) ⇒ b) Sei x eine Ecke und sei $\mathcal{A} = \{z \in \mathbb{R}^d \mid u \cdot z = w\}$ für $u \in \mathbb{R}^d$ und $w \in \mathbb{R}$ eine stützende Hyperebene mit $A \cap \mathcal{P} = \{x\}$ und $\mathcal{P} \subseteq \{z \in \mathbb{R}^d \mid u \cdot z \leq w\}$. Angenommen es gäbe ein $y \neq 0$ mit $x + y \in \mathcal{P}$ und $x - y \in \mathcal{P}$. Wegen $x + y \notin A \cap \mathcal{P}$ muss $u \cdot (x + y) < w$ und damit $u \cdot y < 0$ gelten. Dann gilt aber $u \cdot (x - y) > w$ im Widerspruch zur Annahme $x - y \in \mathcal{P}$.

b) ⇒ c) Angenommen die Spalten in A_x sind nicht linear unabhängig. Wir zeigen, dass x dann kein Extrempunkt sein kann. Da die Spalten von A_x nicht linear unabhängig sind, besitzt das Gleichungssystem $A_x y' = 0$ mit $y' \in \mathbb{R}^{|B(x)|}$ eine Lösung $y' \neq 0$. Wir erweitern diese zu einer Lösung $y \in \mathbb{R}^d$ des Gleichungssystem $Ay = 0$, indem wir $y_i = 0$ für alle $i \notin B(x)$ setzen. Für jedes $\lambda \in \mathbb{R}$ gilt dann $A(x + \lambda y) = b$ und $A(x - \lambda y) = b$. Außerdem gilt für hinreichend kleine $\lambda > 0$ auch $x + \lambda y \geq 0$ und $x - \lambda y \geq 0$. Damit ist gezeigt, dass x kein Extrempunkt ist.

c) ⇒ a) Sei $x \in \mathcal{P}$ eine Basislösung, und seien $B = B(x)$ und $N = \{1, \ldots, d\} \setminus B$. Dann ist $\mathcal{A} = \{z \in \mathbb{R}^d \mid \sum_{i \in N} z_i = 0\}$ eine stützende Hyperebene mit $\mathcal{P} \cap \mathcal{A} = \{x\}$. Zunächst folgt direkt aus der Definition, dass $x \in \mathcal{A}$ gilt, und wegen $\mathcal{P} \subseteq \mathbb{R}^d_{\geq 0}$ folgt auch $\mathcal{P} \subseteq \{z \in \mathbb{R}^d \mid \sum_{i \in N} z_i \geq 0\}$. Es bleibt also lediglich zu zeigen, dass es kein $y \in \mathcal{P}$ mit $y \neq x$ und $\sum_{i \in N} y_i = 0$ gibt. Nehmen wir an, es gäbe ein solches y. Es gilt dann $y_i = 0$ für alle $i \in N$ und somit besitzt das Gleichungssystem $A_B z = b$ zwei verschiedene Lösungen, nämlich x_B und y_B. Dies steht aber im Widerspruch dazu, dass die Spalten von A_B linear unabhängig sind. □

Mit Hilfe der gerade gezeigten Charakterisierung von Ecken können wir zeigen, dass der optimale Wert eines linearen Programms stets an einer Ecke angenommen wird.

Theorem 3.4. *Zu jedem zulässigen und beschränkten linearen Programm mit Lösungspolyeder $\mathcal{P}$ gibt es eine Ecke von $\mathcal{P}$, die einer optimalen Variablenbelegung entspricht.*

Beweis. Sei $x \in \mathcal{P}$ eine optimale Variablenbelegung. Ist x eine Ecke, so ist nichts zu tun. Ansonsten gibt es laut Lemma 3.3 ein $y \in \mathbb{R}^d$ mit $y \neq 0$, für das $x + y \in \mathcal{P}$ und $x - y \in \mathcal{P}$ gelten. Das bedeutet, dass für dieses y insbesondere $A(x + y) = b$ und $A(x - y) = b$ gelten, woraus direkt $Ay = 0$ folgt. Damit gilt $A(x + \lambda y) = Ax + \lambda Ay = b$ für jedes $\lambda \geq 0$.

Gilt $c \cdot y < 0$ oder $c \cdot y > 0$, so ist im Widerspruch zur Optimalität von x entweder $c \cdot (x + y) < c \cdot x$ oder $c \cdot (x - y) < c \cdot x$. Somit muss $c \cdot y = 0$ gelten. Dann gilt $c \cdot (x + \lambda y) = c \cdot x$ für jedes $\lambda \geq 0$. Wir können ohne Beschränkung der Allgemeinheit davon ausgehen, dass es einen Index j mit $y_j < 0$ gibt, da wir ansonsten y durch $-y$ ersetzen könnten.

Wir werden eine Lösung $z \in \mathcal{P}$ konstruieren, für die $c \cdot z = c \cdot x$ gilt und in der mindestens eine Variable mehr den Wert Null annimmt als in der Lösung x. Wir können dann x durch z ersetzen. Dies können wir höchstens d mal wiederholen, bis wir bei einer Ecke angelangt sind.

Wir wählen $\lambda \geq 0$ größtmöglich unter der Einschränkung $z := x + \lambda y \geq 0$. Dies entspricht der Wahl

$$\lambda := \min \left\{ -\frac{x_j}{y_j} \;\middle|\; j \in \{1, \ldots, d\}, y_j < 0 \right\}.$$

Wir bezeichnen mit k einen Index, für den dieses Minimum angenommen wird, d. h. $y_k < 0$ und $-x_k/y_k = \lambda$. Wegen $x \geq 0$ folgt $\lambda \geq 0$. Es gilt sogar $\lambda > 0$, denn ist $y_j < 0$, so muss $x_j > 0$ sein, denn ansonsten wäre $x_j + y_j < 0$ und damit $x + y \notin \mathcal{P}$.

Wegen $z \geq 0$ und $Az = Ax + \lambda Ay = b$ folgt insgesamt, dass z eine gültige Lösung für das gegebene lineare Programm ist. Ferner gilt $c \cdot z = c \cdot x$ wegen $c \cdot y = 0$. Die Lösung z hat im Gegensatz zur Lösung x an der Stelle k eine Null. Wir haben oben bereits argumentiert, dass $x_j = 0$ impliziert, dass auch $y_j = 0$ gilt, da ansonsten $x + y \geq 0$ oder $x - y \geq 0$ nicht gilt. Somit hat z an jeder Stelle, an der x eine Null hat, ebenfalls eine. Insgesamt ist die Anzahl an Nullen in z also um mindestens eins größer als in x. □

Wir gehen im Folgenden ohne Beschränkung der Allgemeinheit davon aus, dass die Matrix $A \in \mathbb{R}^{m \times d}$ Rang m hat, d. h. dass ihre Zeilen linear unabhängig sind. Ist das nicht der Fall, so enthält das lineare Programm entweder eine redundante Nebenbedingung, die entfernt werden kann, oder es ist nicht zulässig.

Theorem 3.4 besagt zusammen mit Lemma 3.3, dass wir zur Lösung eines beschränkten linearen Programms nur alle Basislösungen testen müssen. Sei $x \in \mathbb{R}^d$ eine Basislösung, und sei $B \supseteq B(x)$ so gewählt, dass $|B| = m$ gilt und die Spalten von A_B linear unabhängig sind. Die Existenz eines solchen B folgt daraus, dass Matrix A Rang m hat. Es gilt dann $A_B x_B = b$, und wegen der linearen Unabhängigkeit der Spalten ist x_B sogar die eindeutige Lösung dieses Gleichungssystems. Um nun alle Basislösungen zu testen, genügt es also alle m-elementigen Teilmengen $B \subseteq \{1, \ldots, d\}$ darauf zu testen, ob die Spalten von A_B linear unabhängig sind und ob das Gleichungssystem $A_B x_B = b$ eine Lösung $x_B \in \mathbb{R}^m_{\geq 0}$ besitzt.

Algorithmus: Teste-alle-Basislösungen

1. min := ∞; $x^* :=\perp$;
2. **for all** $B \subseteq \{1, \ldots, d\}$ mit $|B| = m$ **do**
3. **if** Spalten von A_B linear unabhängig **then**
4. Löse das Gleichungssystem $A_B x_B = b$ mit $x_B \in \mathbb{R}^m$.
5. Erweitere x_B zu $x \in \mathbb{R}^d$ durch Ergänzung von Nullen für alle $i \notin B$.
6. **if** $(x \geq 0)$ **and** $(c \cdot x < \min)$ **then**
7. min := $c \cdot x$; $x^* := x$;
8. Ausgabe: x^*

Dieser Algorithmus ist natürlich nicht effizient, da er alle $\binom{d}{m}$ vielen m-elementigen Teilmengen von $\{1, \ldots, d\}$ testet. Außerdem liefert er nur bei beschränkten linearen Programmen das richtige Ergebnis. Wir lernen im folgenden Kapitel einen besseren Algorithmus kennen.

Zum Schluss beantworten wir noch die Frage, wie man sich eine Basislösung anschaulich in einem linearen Programm in kanonischer Form vorstellen kann. Sei ein lineares Programm in

kanonischer Form mit d Variablen und m Nebenbedingungen gegeben, das wir in Gleichungsform transformieren, indem wir m Schlupfvariablen einfügen. Das entsprechende lineare Programm in Gleichungsform besteht dann aus $d+m$ Variablen und m Nebenbedingungen. In jeder Basislösung $x \in \mathbb{R}^{d+m}$ dieses linearen Programms sind höchstens m Variablen ungleich Null und damit mindestens $(d+m)-m = d$ Variablen gleich Null. Jede dieser Variablen ist entweder eine Variable aus dem ursprünglichen linearen Programm in kanonischer Form oder eine neu eingeführte Schlupfvariable. Ist eine Schlupfvariable gleich Null, so bedeutet das, dass die zugehörige Ungleichung mit Gleichheit erfüllt ist. Ist eine Variable aus dem ursprünglichen linearen Programm gleich Null, so bedeutet das, dass die Nichtnegativitätsbedingung mit Gleichheit erfüllt ist. Auf jeden Fall sind d Nebenbedingungen mit Gleichheit erfüllt und die Basislösung entspricht dem Schnittpunkt der zugehörigen Hyperebenen.

Beispiel

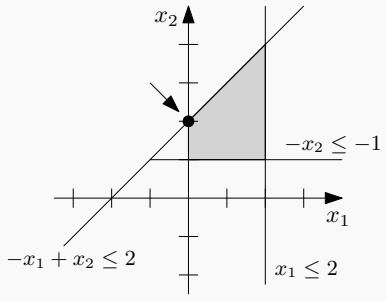
Wie betrachten die folgenden Nebenbedingungen in kanonischer Form:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq 0 \quad \text{und} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

In Gleichungsform können wir diese Nebenbedingungen wie folgt schreiben:

$$\begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} \geq 0 \quad \text{und} \quad \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

Das Lösungspolyeder $\mathcal{P}$ ist in der Abbildung grau hervorgehoben. Die markierte Ecke $(0,2)$ entspricht dem Schnittpunkt der Hyperebenen $x_1 = 0$ und $-x_1+x_2 = 2$. In dem linearen Programm in Gleichungsform entspricht diese Ecke der Basislösung $(0,2,2,1,0)$, wobei die letzten drei Komponenten s_1, s_2 und s_3 entsprechen.



Für $x = (0,2,2,1,0)$ gilt $B = B(x) = \{1,5\}$ und dementsprechend ist x_B die eindeutige Lösung des folgenden Gleichungssystems:

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_2 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

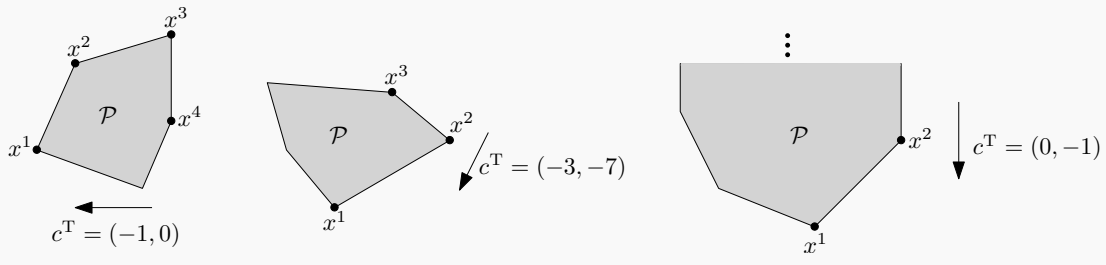
3.2 Simplex-Algorithmus

Der Simplex-Algorithmus, den wir in diesem Abschnitt beschreiben werden, wurde bereits 1947 von George Dantzig vorgeschlagen. Es handelt sich dabei auch heute noch um einen der erfolgreichsten Algorithmen zur Lösung von linearen Programmen in der Praxis. Wir beschreiben den Algorithmus zunächst wieder anschaulich, bevor wir eine formale Beschreibung angeben.

Sei ein lineares Programm mit Lösungspolyeder $\mathcal{P}$ gegeben. Der Simplex-Algorithmus führt eine lokale Suche auf den Ecken des Lösungspolyeders durch. Das bedeutet, er startet mit einer beliebigen Ecke $x^1 \in \mathcal{P}$ und testet, ob es eine benachbarte Ecke mit besserem Zielfunktionswert gibt. Dabei sind zwei Ecken benachbart, wenn sie auf der gleichen Kante liegen. Gibt es eine solche benachbarte Ecke $x^2 \in \mathcal{P}$, so macht der Algorithmus mit x^2 analog weiter und testet wieder, ob es eine bessere benachbarte Ecke gibt. Den Übergang von einer Ecke zu einer besseren in der Nachbarschaft nennen wir auch *Pivotschritt*. Da der Algorithmus den Zielfunktionswert mit jedem Schritt verbessert, kann er keine Ecke mehrfach besuchen. Da es nur endliche viele Ecken gibt, muss also nach einer endlichen Anzahl an Schritten eine Ecke $x^k \in \mathcal{P}$ erreicht sein, in deren Nachbarschaft es keine bessere Ecke gibt. Es gibt dann zwei Fälle, die eintreten können. Entweder die Ecke x^k ist in einer unbeschränkten Kante enthalten, entlang derer der Zielfunktionswert beliebig gut werden kann, oder, wenn das nicht der Fall ist, dann ist die Ecke x^k optimal. Wir werden später beweisen, dass dies wirklich so ist.

Beispiel

Bei den ersten beiden Beispielen wird das Optimum an den Knoten x^4 bzw. x^3 angenommen. Im dritten Beispiel ist x^2 der letzte besuchte Knoten, von dem aus es eine unendliche Kante gibt, entlang derer der Wert der Zielfunktion beliebig klein wird.



Genau wie bei dem geometrischen Algorithmus zur Lösung von linearen Programmen weisen wir auch hier darauf hin, dass sich aus der obigen anschaulichen Beschreibung des Simplex-Algorithmus noch keine direkte Implementierung ergibt. So ist zum Beispiel nicht klar, wie im ersten Schritt überhaupt eine beliebige Ecke von $\mathcal{P}$ gefunden werden kann oder wie man die Nachbarschaft einer Ecke durchsuchen kann.

3.2.1 Formale Beschreibung

Zur formalen Beschreibung betrachten wir wieder ein lineares Programm in Gleichungsform, in dem die Zielfunktion $c \cdot x$ unter den Nebenbedingungen $Ax = b$ und $x \geq 0$ minimiert werden soll. Dabei seien $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times d}$ und $x \in \mathbb{R}^d$ sei ein Vektor, der aus d Variablen besteht. Es bezeichne $\mathcal{P}$ das zugehörige Lösungspolyeder. Wie bereits im letzten Abschnitt gehen wir davon aus, dass alle Zeilen der Matrix A linear unabhängig sind.

Wir sagen, dass das lineare Programm *degeneriert* ist, wenn es eine zulässige Basislösung $x \in \mathcal{P}$ mit $|B(x)| < m$ gibt. Ist ein lineares Programm nicht degeneriert, so sind in jeder zulässigen Basislösung genau m Variablen ungleich Null. Wir werden in diesem Abschnitt davon ausgehen, dass das vorliegende lineare Programm nicht degeneriert ist. Diese Annahme erleichtert zwar die Beschreibung, sie ist aber nicht notwendig, denn an sich können mit dem Simplex-Algorithmus auch degenerierte lineare Programme gelöst werden. Wir haben gesehen, dass Basislösungen in linearen Programmen in kanonischer Form Punkten entsprechen, in denen sich mindestens d der durch die Nebenbedingungen beschriebenen Hyperebenen schneiden. Ein solches lineares Programm ist genau dann degeneriert, wenn es einen Punkt in $\mathcal{P}$ gibt, in dem sich mehr als d Hyperebenen schneiden.

Wir gehen zunächst davon aus, dass wir eine beliebige Ecke $x = x^1 \in \mathcal{P}$ kennen. Diese Ecke entspricht gemäß Lemma 3.3 einer zulässigen Basislösung. Für $B = B(x)$ und $N = \{1, \ldots, d\} \setminus B$ ergibt sich diese Basislösung als eindeutige Lösung des Gleichungssystems $A_x x_B = b$ und $x_N = 0$. Auf Grund der Annahme, dass das lineare Programm nicht degeneriert ist, besteht die Matrix A_x aus genau m linear unabhängigen Spalten. Somit ist A_x einer invertierbare $m \times m$-Matrix. Es gilt also insgesamt

$$x = \begin{pmatrix} x_B \\ x_N \end{pmatrix} = \begin{pmatrix} A_x^{-1} b \\ 0 \end{pmatrix}.$$

Wir nennen die Variablen in x_B *Basisvariablen* und die Variablen in x_N *Nichtbasisvariablen*. Um zu einer benachbarten Ecke zu gelangen, wollen wir jetzt den Wert einer Nichtbasisvariablen erhöhen. Dazu ist es zunächst hilfreich, die Werte der Basisvariablen in Abhängigkeit von den Werten der Nichtbasisvariablen auszudrücken. Dazu bezeichne $\bar{A}_x$ die Teilmatrix von A, die aus genau den Spalten besteht, die nicht in A_x enthalten sind. Es gilt dann

$$A_x x_B + \bar{A}_x x_N = b \iff x_B = A_x^{-1}(b - \bar{A}_x x_N).$$

Wählen wir $x_N \in \mathbb{R}_{\geq 0}^{d-m}$ beliebig, so ergibt sich daraus eine eindeutige Wahl von x_B. Wir können deshalb die Zielfunktion in Abhängigkeit von x_N ausdrücken:

$$\begin{aligned} c \cdot x &= c_B \cdot x_B + c_N \cdot x_N = c_B \cdot (A_x^{-1}(b - \bar{A}_x x_N)) + c_N \cdot x_N \\ &= c_B \cdot (A_x^{-1} b) + (c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_x^{-1} \bar{A}_x) x_N. \end{aligned} \tag{3.3}$$

Eine Wahl $x_N \in \mathbb{R}_{\geq 0}^{d-m}$ ist nur dann zulässig, wenn sich daraus nichtnegative Werte für die Basisvariablen ergeben.

Wir nennen zwei Basislösungen *benachbart*, wenn sich die Mengen der Basisvariablen nur in einem Element unterscheiden. Wir können von einer Basislösung x zu einer benachbarten Basislösung gelangen, indem wir eine Nichtbasisvariable $j \in N$ auswählen und den Wert von x_j erhöhen und die Werte aller anderen Nichtbasisvariablen auf Null lassen. Die zu j gehörige Komponente des Vektors $\bar{c} := c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_x^{-1} \bar{A}_x$ aus Gleichung (3.3) beschreibt dann, wie sich der Wert der Zielfunktion ändert. Wir nennen $\bar{c}$ auch den Vektor der *reduzierten Kosten*. Nur wenn die zu j gehörige Komponente des Vektors $\bar{c}$ negativ ist, verbessert sich die Lösung durch die Erhöhung von x_j. Das folgende Lemma besagt, dass wir anhand der reduzierten Kosten erkennen können, ob x eine optimale Lösung ist.

Lemma 3.5. *Falls der Vektor der reduzierten Kosten $\bar{c}$ zu einer Basislösung x keinen negativen Eintrag enthält, so ist x eine optimale Lösung für das lineare Programm.*

Beweis. Es sei $\bar{c} := c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_x^{-1} \bar{A}_x \geq 0$. Sei $x' \in \mathcal{P}$ eine beliebige zulässige Lösung. Dann ist $x'_N \geq 0$ und Gleichung (3.3) impliziert

$$c \cdot x' = c_B \cdot (A_x^{-1} b) + \bar{c}\, x'_N \geq c_B \cdot (A_x^{-1} b) = c \cdot x.$$

Also ist x' nicht besser als x, und somit muss x eine optimale Lösung sein. □

Gilt $\bar{c} \geq 0$ für die aktuelle Basislösung x, so terminiert der Simplex-Algorithmus und gibt aus, dass x optimal ist. Ansonsten betrachten wir eine beliebige Nichtbasisvariable x_j mit $\bar{c}_j < 0$ und erhöhen diese kontinuierlich von 0. Wenn wir die Basisvariablen gemäß $x_B = A_x^{-1}(b - \bar{A}_x x_N)$ entsprechend anpassen, dann verringert sich durch die Erhöhung von x_j gemäß Gleichung (3.3) der Zielfunktionswert. Es kann jedoch sein, dass die Erhöhung von x_j dazu führt, dass sich der Wert einiger Basisvariablen verringert und sogar negativ wird. Dann erhalten wir keine zulässige Lösung mehr. Da wir davon ausgehen, dass das lineare Programm nicht degeneriert ist, haben in x aber alle Basisvariablen echt positive Werte. Das bedeutet, wir können x_j zumindest von Null auf einen eventuell kleinen aber zumindest echt positiven Wert setzen, ohne dadurch eine unzulässige Lösung zu erhalten.

Die zu x_j gehörige Spalte der Matrix $A_x^{-1}\bar{A}_x$ nennen wir im Folgenden $u \in \mathbb{R}^m$. Ist keine Komponente von u echt positiv, so können wir x_j beliebig groß setzen, ohne dadurch eine unzulässige Lösung zu erhalten. Das bedeutet, dass das lineare Programm unbeschränkt ist, da wir beliebig gute zulässige Lösungen finden können. Ansonsten setzen wir x_j größtmöglich unter der Bedingung, dass alle Basisvariablen nichtnegativ bleiben. Das bedeutet

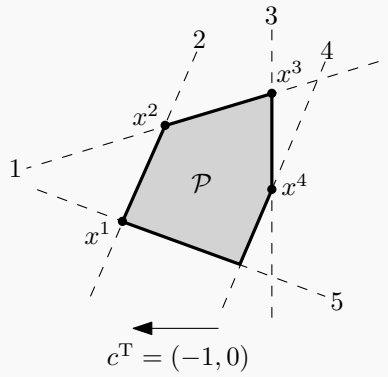
$$x_j := \min \left\{ \frac{x_\ell}{u_\ell} \;\middle|\; \ell \in B, u_\ell > 0 \right\}.$$

Sei x_k eine Basisvariable, für die das obige Minimum angenommen wird. Nachdem wir x_j erhöht haben, nimmt x_k den Wert Null an. Auf diese Weise erhalten wir eine neue Basislösung x' mit $B(x') = B(x) \cup \{j\} \setminus \{k\}$. Damit ist x' eine zu x benachbarte Basislösung mit besserem Zielfunktionswert. Hier haben wir ausgenutzt, dass das lineare Programm nicht degeneriert ist und dass deshalb $|B(x')| = m$ gilt.

Wir haben uns bereits im vorigen Abschnitt überlegt, dass Basislösungen in linearen Programmen in kanonischer Form Schnittpunkten von d durch die Nebenbedingungen beschriebenen Hyperebenen entsprechen. Diese Hyperebenen sind durch die Nichtbasisvariablen bestimmt. Nun haben wir gerade definiert, dass zwei Basislösungen benachbart sind, wenn genau eine Nichtbasisvariable durch eine andere ausgetauscht wird. Für lineare Programme in kanonischer Form können wir also sagen, dass zwei Basislösungen benachbart sind, wenn sie jeweils ein Schnittpunkt von d Hyperebenen sind und wenn sie auf $d-1$ dieser Hyperebenen gemeinsam liegen.

Beispiel

In dem folgenden Beispiel ist eine Ausführung des Simplex-Algorithmus zu sehen, in der die Folge x^1, x^2, x^3, x^4 von Basislösungen durchlaufen wird. Jede dieser Basislösungen können wir als Schnittpunkt zweier Hyperebenen darstellen. Die Basislösung x^1 entspricht zum Beispiel der Menge $\{2, 5\}$ von Hyperebenen. Insgesamt ergibt sich die Folge $\{2, 5\}, \{1, 2\}, \{1, 3\}, \{3, 4\}$. Dort ist zu erkennen, dass jeweils eine Hyperebene ausgetauscht wird.



3.2.2 Berechnung der initialen Basislösung

Wir haben noch nicht diskutiert, wie eine initiale Basislösung gefunden werden kann, mit der der Simplex-Algorithmus gestartet werden kann. Sei ein lineares Programm mit den Nebenbedingungen $Ax = b$ gegeben und sei ohne Beschränkung der Allgemeinheit $b \geq 0$. Ist das nicht der Fall, so können die entsprechenden Nebenbedingungen mit -1 multipliziert werden. Für die m Nebenbedingungen führen wir Hilfsvariablen $h_1 \geq 0, \ldots, h_m \geq 0$ ein. Die Nebenbedingung $a_i \cdot x = b$ ersetzen wir für jedes $i \in \{1, \ldots, m\}$ durch die Nebenbedingung $a_i \cdot x + h_i = b_i$. Wir ignorieren außerdem die gegebene Zielfunktion des linearen Programms und definieren als neue Zielfunktion $h_1 + \ldots + h_m$.

Das so entstandene lineare Programm hat die Eigenschaft, dass wir einfach eine zulässige Basislösung angeben können. Dazu setzen wir $h_i = b_i$ für jedes $i \in \{1, \ldots, m\}$ und $x_i = 0$ für alle $i \in \{1, \ldots, d\}$. Wir können somit den Simplex-Algorithmus benutzen, um für dieses lineare Programm eine optimale Basislösung (x^*, h^*) zu berechnen. Dieser Schritt wird als die *erste Phase* des Simplex-Algorithmus bezeichnet. Ist der Wert der Zielfunktion in dieser optimalen Lösung echt größer als Null, dann gibt es keine Lösung, die alle Gleichungen des ursprünglichen linearen Programms erfüllt, und somit ist dieses lineare Programm nicht zulässig. Ansonsten, wenn $h^* = 0$ gilt, so ist x^* eine zulässige Basislösung für das ursprüngliche lineare Programm. Mit dieser Lösung x^* kann dann der Simplex-Algorithmus für das ursprüngliche lineare Programm initialisiert werden. Die Optimierung der eigentlich Zielfunktion wird dann als *zweite Phase* des Simplex-Algorithmus bezeichnet.

3.2.3 Laufzeit

Die Laufzeit des Simplex-Algorithmus setzt sich aus zwei Komponenten zusammen. Zum einen muss geklärt werden, was die Komplexität eines einzelnen Pivotschrittes ist und zum anderen muss geklärt werden, wie viele Pivotschritte es gibt. Die Antwort auf die erste Frage gibt folgendes Theorem, das wir nicht beweisen werden.

Theorem 3.6. *Die Laufzeit eines einzelnen Pivotschrittes ist polynomiell in der Eingabelänge des linearen Programms beschränkt.*

Aus der Beschreibung des Simplex-Algorithmus geht bereits hervor, dass zur Durchführung eines Pivotschrittes nur einige elementare Rechenoperationen durchgeführt werden müssen. Trotzdem ist das obige Theorem nicht trivial, denn die Laufzeit dieser Rechenoperationen hängt von der Darstellungslänge der vorkommenden Zahlen ab. Diese muss man in Beziehung zu der Eingabelänge setzen. Man muss also zeigen, dass die Zahlen, mit denen man beim Simplex-Algorithmus rechnet, nicht zu groß werden können.

Die Antwort auf die zweite Frage ist leider negativ. Klee und Minty haben 1972 folgendes Theorem gezeigt.

Theorem 3.7. *Für jedes $n \in \mathbb{N}$ gibt es ein lineares Programm in Gleichungsform mit $3n$ Variablen und $2n$ Nebenbedingungen, in dem alle Koeffizienten ganzzahlig sind und Absolutwert höchstens 4 haben, und auf dem der Simplex-Algorithmus $2^n - 1$ Pivotschritte durchführen kann.*

In diesem Theorem haben wir davon gesprochen, dass der Simplex-Algorithmus $2^n - 1$ Pivotschritte „durchführen kann“. Diese Formulierung ist so gewählt, da es für eine Basislösung oft mehrere mögliche Pivotschritte gibt und wir bisher nicht spezifiziert haben, welcher davon durchgeführt wird. Das Theorem ist so zu lesen, dass eine exponentiell lange Folge von Pivotschritten existiert, die der Simplex-Algorithmus durchführen kann. Es ist aber zunächst nicht ausgeschlossen, dass man durch eine geschickte Wahl der Pivotschritte garantieren kann, dass der Simplex-Algorithmus stets polynomiell viele Schritte benötigt. Eine Regel, die die Wahl des nächsten Pivotschrittes bestimmt, wenn mehrere zur Auswahl stehen, heißt *Pivotregel*. Zahlreiche solche Regeln wurden vorgeschlagen, für fast alle von ihnen kennt man aber mittlerweile Instanzen, auf denen der Simplex-Algorithmus exponentiell viele Pivotschritte benötigt. Ob es eine Pivotregel gibt, die garantiert, dass der Simplex-Algorithmus nach polynomiell vielen Schritten terminiert, ist bis heute unklar.

3.3 Komplexität von linearer Programmierung

Lineare Programmierung ist ein sehr gut untersuchtes Thema der Optimierung, und für lange Zeit war es eine große offene Frage, ob es Algorithmen gibt, die lineare Programme in polynomieller Zeit lösen. Diese Frage wurde 1979 von Leonid Khachiyan positiv beantwortet. Er stellte die *Ellipsoidmethode* vor, die in polynomieller Zeit testet, ob ein gegebenes Lösungspolyeder $\mathcal{P}$ leer ist oder nicht. Die Ellipsoidmethode erzeugt zunächst ein Ellipsoid (die mehrdimensionale Entsprechung einer Ellipse), das $\mathcal{P}$ komplett enthält. In jeder Iteration wird dann getestet, ob das Zentrum des Ellipsoid in $\mathcal{P}$ liegt. Falls ja, so ist $\mathcal{P}$ nicht leer. Ansonsten wird ein neues Ellipsoid berechnet, das $\mathcal{P}$ immer noch enthält, dessen Volumen, aber um einen gewissen Faktor kleiner ist als das Volumen des alten Ellipsoid. Man kann nachweisen, dass nach polynomiell vielen Iterationen entweder ein Punkt in $\mathcal{P}$ gefunden wird, oder das aktuelle Ellipsoid so klein geworden ist, dass $\mathcal{P} = \emptyset$ gelten muss.

Eine beliebige Variablenbelegung $x \in \mathcal{P}$ zu berechnen oder zu entscheiden, dass keine solche existiert, ist ein scheinbar einfacheres Problem, als die optimale Lösung eines linearen Programms zu bestimmen. Tatsächlich gilt aber, dass sich ein polynomieller Algorithmus $\mathcal{A}$ für das erste Problem in einen polynomiellen Algorithmus für das zweite Problem transformieren

lässt. Aus einem linearen Programm, in dem die Zielfunktion $c \cdot x$ unter den Nebenbedingungen $Ax \leq b$ minimiert werden soll, erzeugen wir ein neues lineares Programm mit den Nebenbedingungen $Ax \leq b$ und zusätzlich $c \cdot x \leq z$ für ein $z \in \mathbb{R}$. Ist das zu diesem linearen Programm gehörige Lösungspolyeder nicht leer, so besitzt das ursprüngliche lineare Programm eine Lösung mit Wert höchstens z. Wir suchen jetzt mit Hilfe von Algorithmus $\mathcal{A}$ und einer binären Suche das kleinste z für das das oben genannte lineare Programm eine Lösung besitzt. Für dieses lineare Programm bestimmen wir dann mit Hilfe von A eine zulässige Lösung. Diese Lösung entspricht dann einer optimalen Lösung des ursprünglichen linearen Programms.

Obwohl die Ellipsoidmethode ein großer theoretischer Durchbruch war, ist ihr Einfluss auf die Praxis gering, denn in praktischen Anwendungen ist der Simplex-Algorithmus trotz seiner exponentiellen worst-case Laufzeit der polynomiellen Ellipsoidmethode deutlich überlegen. Im Jahre 1984 gelang es Narendra Karmarkar mit dem *Innere-Punkte-Verfahren* einen polynomiellen Algorithmus zur Lösung von linearen Programmen vorzustellen, der auch in einigen praktischen Anwendungen ähnlich gute Laufzeiten wie der Simplex-Algorithmus erzielt. Dennoch ist der Simplex-Algorithmus auch heute noch der am meisten benutzte Algorithmus zur Lösung von linearen Programmen.

Literaturverzeichnis

[1] Norbert Blum. *Algorithmen und Datenstrukturen - eine anwendungsorientierte Einführung*. Oldenbourg, 2004.

[2] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[3] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[4] Christos H. Papadimitriou and Santosh Vempala. On the approximability of the traveling salesman problem. *Combinatorica*, 26(1):101–120, 2006.

[5] Vijay V. Vazirani. *Approximation Algorithms*. Springer Verlag, 2001.

[6] Berthold Vöcking. Einführung in Approximationsalgorithmen, Sommersemester 2008. http://www-i1.informatik.rwth-aachen.de/Lehre/SS08/VEA/Skripte/approx.pdf.

[7] Berthold Vöcking. Einführung in die Lineare Programmierung, Sommersemester 2008. http://www-i1.informatik.rwth-aachen.de/Lehre/SS08/VEA/Skripte/lp.pdf.

[8] Berthold Vöcking. Einführung in Online-Algorithmen, Sommersemester 2008. http://www-i1.informatik.rwth-aachen.de/Lehre/SS08/VEA/Skripte/online.pdf.

[9] Berthold Vöcking. Berechenbarkeit und Komplexität, Wintersemester 2009/10. http://algo.rwth-aachen.de/Lehre/WS0910/VBuK/BuK.pdf.