Randomisierte und Approximative Algorithmen

Prof. Dr. Heiko Röglin
Institut für Informatik
Universität Bonn

20. November 2020

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `roeglin@cs.uni-bonn.de`.

Kapitel 1

Einleitung

Wenn wir für ein Optimierungsproblem gezeigt haben, dass es NP-schwer ist, dann bedeutet das zunächst nur, dass es unter der Annahme P $\neq$ NP keinen effizienten Algorithmus gibt, der für jede Instanz eine optimale Lösung berechnet. Das schließt nicht aus, dass es einen effizienten Algorithmus gibt, der für jede Instanz eine Lösung findet, 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 den Wert einer optimalen Lösung in jedem Falle mit $\mathrm{OPT}(I)$. 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. *Ein Approximationsalgorithmus A für ein Minimierungs- oder Maximierungsproblem Π erreicht einen* Approximationsfaktor *(auch* Approximationsgüte *genannt) 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.*

In vielen Fällen hängt der Approximationsfaktor von der Eingabelänge ab. Dann ist $r : \mathbb{N} \to [0, \infty)$ eine Funktion und es muss

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

für alle Instanzen I gelten, wobei $|I|$ die Länge der Eingabe I bezeichne.

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-schwer, so schließt dies unter der Voraussetzung $\mathrm{P} \neq \mathrm{NP}$ lediglich die Existenz eines solchen 1-Approximationsalgorithmus aus. Es gibt aber durchaus NP-schwere 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.

In dieser Vorlesung werden wir Approximationsalgorithmen für zahlreiche Probleme entwerfen. Wir werden die Approximationsgüten dieser Algorithmen untersuchen und allgemeine Methoden zum Entwurf von Approximationsalgorithmen kennenlernen. Ebenso werden wir uns mit den Grenzen der Approximierbarkeit beschäftigen und nachweisen, dass für bestimmte Probleme kleine Approximationsgüten nicht in polynomieller Zeit erreicht werden können, falls $\mathrm{P} \neq \mathrm{NP}$ gilt.

Dieses Skript enthält alle Inhalte der Vorlesung. Als weitere Literatur sind die Bücher über Approximationsalgorithmen von David Williamson und David Shmoys [4] sowie von Vijay Vazirani [3] sehr empfehlenswert. Ebenfalls eine schöne Einführung in die Thematik enthält das Buch von Bernhard Korte und Jens Vygen über kombinatorische Optimierung [2]. Viele der in diesem Skript dargestellten Themen basieren auf diesen Büchern.

Kapitel 2

Greedy-Algorithmen

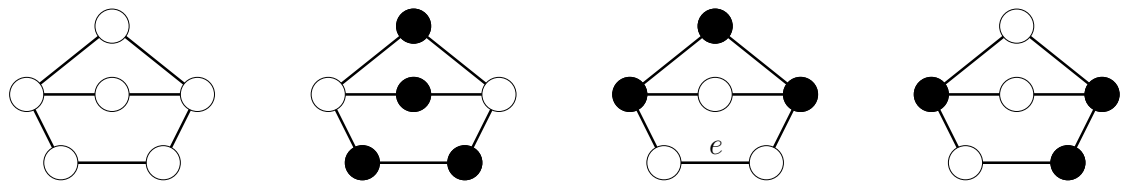
Wir haben Greedy-Algorithmen bereits in der Vorlesung „Algorithmen und Berechnungskomplexität I“ kennengelernt und nachgewiesen, dass sie für manche Probleme optimale Lösungen liefern. Die meisten Probleme können mithilfe von Greedy-Algorithmen allerdings nicht optimal gelöst werden. Als Einstieg in die Vorlesung zeigen wir in diesem Kapitel zunächst, dass Greedy-Algorithmen für eine Reihe von NP-schweren Problemen zumindest gute Approximationsalgorithmen sind.

2.1 Vertex Cover

Das folgende Problem *Vertex Cover* ist uns aus der Vorlesung „Algorithmen und Berechnungskomplexität II“ als NP-schweres Problem bekannt.

Vertex Cover

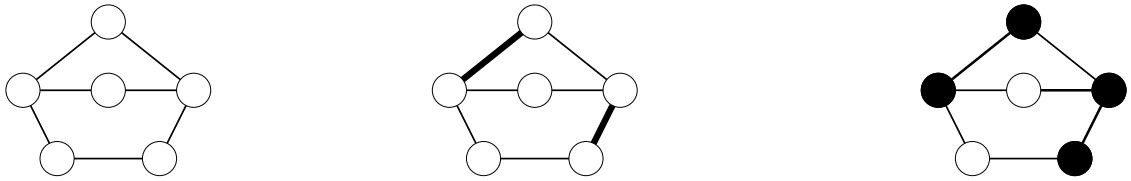
Eingabe:	ungerichteter Graph $G = (V, E)$
Lösungen:	alle $V' \subseteq V$ mit $\forall e = (u, v) \in E : u \in V'$ oder $v \in V'$ (oder beides) (Ein V' mit dieser Eigenschaft nennen wir *Vertex Cover von* G.)
Zielfunktion:	minimiere $\lvert V' \rvert$



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

Abbildung 2.1: Beispiel für das Vertex Cover Problem

Abbildung 2.1 illustriert die Definition des Problems. Wir präsentieren nun einen 2-Approximationsalgorithmus für Vertex Cover. Der Algorithmus benutzt den Begriff



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

Abbildung 2.2: Beispiel für den Algorithmus MATCHING-VC

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 aus 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.

Der folgende Algorithmus MATCHING-VC berechnet ein inklusions-maximales Matching und gibt die Endpunkte der Kanten aus diesem Matching aus. Er ist in Abbildung 2.2 illustriert.

Matching-VC

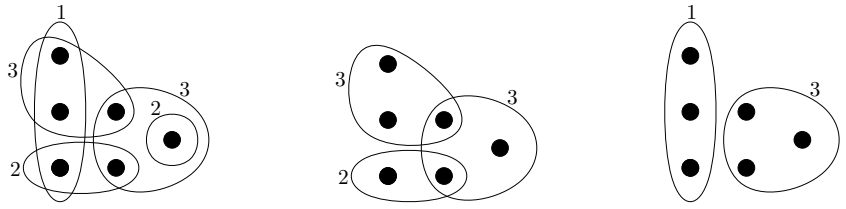
1. Berechne ein inklusions-maximales Matching $M \subseteq E$.
   Setze dazu zunächst $M = \emptyset$ und gehe dann die Kantenmenge einmal durch. Füge dabei eine Kante $e = (u, v) \in E$ zur Menge M hinzu, falls weder u noch v bereits inzident zu einer Kante aus M sind (falls also $M \cup \{e\}$ ein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

Theorem 2.1. *Der Algorithmus* MATCHING-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 Feld speichern, welche Knoten bereits inzidente Kanten in M haben, und dann die Liste der Kanten einmal durchlaufen. Die Laufzeit für Schritt 1 beträgt somit $O(|V| + |E|)$. Mithilfe des Feldes kann die Menge $V(M)$ direkt ausgegeben werden. Die Laufzeit für Schritt 2 beträgt 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 und der Algorithmus hätte e der Menge M hinzugefügt. 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



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

Abbildung 2.3: Beispiel für das Problem Set Cover

minimales Vertex Cover V^*. Wir wissen, dass V^* insbesondere jede Kante aus 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 somit keine gemeinsamen Endknoten besitzen, kann kein Knoten mehr als eine Kante aus M abdecken. Zusammen impliziert das $\mathrm{OPT}(G) = |V^*| \geq |M|$. 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.

2.2 Set Cover

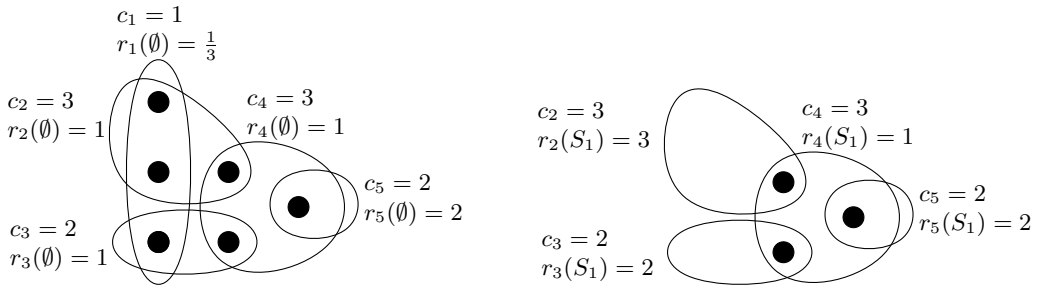
Als nächstes betrachten wir die folgende Verallgemeinerung von Vertex Cover.

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 *Set Cover von* S.)
Zielfunktion:	minimiere $c(A) = \sum_{i \in A} c_i$

Abbildung 2.3 illustriert die Definition des Problems. Das Problem Vertex Cover 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 und jede Menge dieselben Kosten besitzt. Da bereits dieser Spezialfall NP-schwer ist, trifft dies auch auf das allgemeinere Problem Set Cover zu. Wir betrachten nun den folgenden Approximationsalgorithmus Greedy-SC für Set Cover.

Greedy-SC

1. $A := \emptyset$; // Menge der ausgewählten Mengen
2. $C := \emptyset$; // Menge der abgedeckten Elemente
3. **while** $(C \neq S)$
4. Wähle eine Menge S_i mit minimalen *relativen Kosten* $r_i(C) = \frac{c_i}{|S_i \setminus C|}$.
5. Setze price$(x) := \frac{c_i}{|S_i \setminus C|}$ für alle $x \in S_i \setminus C$. // nur für Analyse
6. $A := A \cup \{i\}$; $C := C \cup S_i$;
7. **return** A



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

Abbildung 2.4: Beispiel für den Algorithmus Greedy-SC

Theorem 2.2. *Der Algorithmus* Greedy-SC *ist ein H_n-Approximationsalgorithmus für Set Cover, wobei $H_n = \sum_{i=1}^{n} \frac{1}{i}$ die n-te harmonische Zahl bezeichne.*

Beweis. Wir müssen die Kosten $c(A)$ der vom Algorithmus berechneten Lösung A mit den Kosten der optimalen Lösung 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

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

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

ordnen wir diese in einer beliebigen Reihenfolge. Sei $x_1, \ldots, x_n$ die Reihenfolge, die sich ergibt.

Lemma 2.3. *Für jedes $k \in \{1, \ldots, n\}$ gilt* $\text{price}(x_k) \leq \text{OPT}/(n-k+1)$.

Beweis. Wir betrachten ein beliebiges $k \in \{1, \ldots, n\}$. Sei $i \in \{1, \ldots, m\}$ der 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.

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 $\text{OPT}' = c(A^*)$ ihre Kosten. Wir können OPT' schreiben als

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

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

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

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

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

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

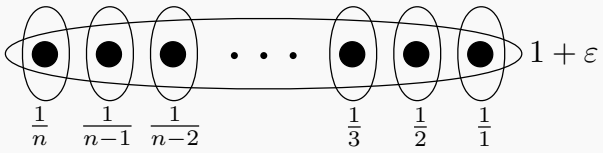
Da die Laufzeit des Algorithmus offensichtlich polynomiell ist, ist damit gezeigt, dass es sich um einen H_n-Approximationsalgorithmus handelt. □

Die harmonische Zahl verhält sich asymptotisch wie der natürliche Logarithmus. 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 $O(\log(n))$ erreicht.

Als letztes beantworten wir noch die Frage, ob wir den Algorithmus gut analysiert haben. Es könnte durchaus sein, dass er eigentlich sogar eine bessere Approximationsgüte als H_n erreicht und unsere Analyse nicht genau genug war, um dies 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 $\Omega(\log(n))$ beträgt.



2.3 Scheduling auf identischen Maschinen

Scheduling-Probleme haben eine große Bedeutung in der Informatik. Es geht dabei darum geht, Prozessen Ressourcen zuzuteilen. In dem Modell, das wir betrachten, ist eine Menge $J = \{1, \ldots, n\}$ von *Jobs* oder *Prozessen* gegeben, die auf eine Menge $M = \{1, \ldots, m\}$ von *Maschinen* oder *Prozessoren* verteilt werden muss. Jeder Job $j \in J$ besitzt dabei eine *Größe* $p_j \in \mathbb{R}_{>0}$, die angibt, wie viele Zeiteinheiten benötigt werden, um ihn vollständig auszuführen. 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* (oder *Last*) von Maschine $i \in M$ in Schedule π, d. h.

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

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. Wir nennen die hier beschriebene Scheduling-Variante *Scheduling auf identischen Maschinen*, da alle Maschinen dieselbe Geschwindigkeit haben. Dieses Problem ist NP-schwer, was durch eine einfache Reduktion von PARTITION gezeigt werden kann.

Wir betrachten als erstes den Greedy-Algorithmus LEAST-LOADED, der die Jobs in der Reihenfolge $1, 2, \ldots, n$ durchgeht und jeden Job einer Maschine zuweist, die die kleinste Ausführungszeit bezogen auf die bereits zugewiesenen Jobs besitzt.

Theorem 2.4. *Der* LEAST-LOADED*-Algorithmus ist ein* $(2 - 1/m)$*-Approximationsalgorithmus für das Problem Scheduling auf identischen Maschinen.*

Beweis. Sei eine beliebige Eingabe gegeben. Der Beweis beruht auf den folgenden beiden unteren Schranken für den Makespan des optimalen Schedules π^*:

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

Die erste Schranke folgt aus 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, zu dem diese Zuweisung erfolgt ist, war Maschine i eine Maschine mit der kleinsten Ausführungszeit. Nur die Jobs $1, \ldots, j-1$ waren bereits verteilt und dementsprechend muss es eine Maschine gegeben haben, deren Ausführungszeit höchstens dem Durchschnitt $\frac{1}{m} \sum_{k=1}^{j-1} p_k$ entsprach. Wir können den Makespan von π nun wie folgt abschätzen:

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

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

Das folgende Beispiel zeigt, dass der Least-Loaded-Algorithmus im Worst Case keinen besseren Approximationsfaktor als $(2 - 1/m)$ erreicht.

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. In einer optimalen Lösung werden die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m-1$ verteilt und der letzte 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

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

Damit ist gezeigt, dass der Least-Loaded-Algorithmus im Allgemeinen keinen besseren Approximationsfaktor als $2 - 1/m$ erreicht.

Wir zeigen nun, dass für das Problem Scheduling auf identischen Maschinen mit einem modifizierten Algorithmus ein Approximationsfaktor von $4/3$ erreicht werden kann.

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.

Der einzige Unterschied zwischen dem LPT- und dem Least-Loaded-Algorithmus besteht darin, dass die Jobs nach ihrer Größe sortiert in absteigender Reihenfolge betrachtet werden. Diese Sortierung verbessert den Approximationsfaktor deutlich.

Theorem 2.5. *Der* Longest-Processing-Time*-Algorithmus ist ein $\frac{4}{3}$-Approximationsalgorithmus für Scheduling auf 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 OPT des optimalen Schedules π^*. Außerdem gehen wir davon aus, dass wir eine Instanz mit der kleinstmöglichen Anzahl an Jobs gewählt haben, auf der dies der Fall ist.

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 2.4 können wir argumentieren, dass es zum Zeitpunkt der Zuweisung von Job n eine Maschine mit Ausführungszeit höchstens $\frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) \leq \mathrm{OPT}$ gibt. Da wir Job n der Maschine i mit der bisher kleinsten Ausführungszeit zuweisen, gilt

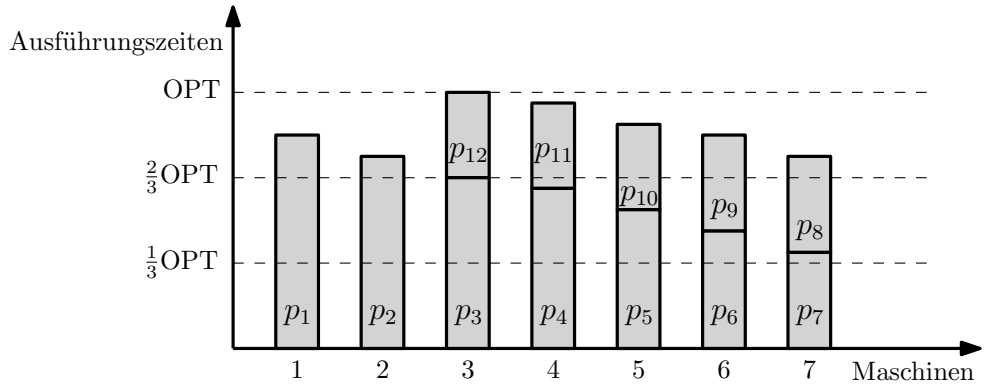
$$C(\pi) = L_i(\pi) \leq \frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) + p_n \leq \mathrm{OPT} + p_n.$$

Der Makespan von π kann also nur dann größer als $\frac{4}{3} \cdot \mathrm{OPT}$ sein, wenn $p_n > \mathrm{OPT}/3$ gilt. Wegen der Sortierung der Jobs gilt $p_j > \mathrm{OPT}/3$ dann sogar 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 wirklich ein optimaler Schedule ist, falls alle Jobs echt größer als OPT/3 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 wie im oben beschriebenen optimalen Schedule zugewiesen wurden. 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 $L_{2m-j+1} + p_j \leq \text{OPT}$. 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 $\frac{1}{3}$OPT sind, folgt für diese Maschinen $L_i + p_j > \text{OPT}$ 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 erreicht. Wir haben gezeigt, dass der LPT-Algorithmus auf Eingaben, in denen alle Jobs echt größer als $\frac{1}{3}$OPT sind, den optimalen Schedule berechnet. Für alle anderen Eingaben haben wir bewiesen, dass der LPT-Algorithmus eine $\frac{4}{3}$-Approximation berechnet. □

Wir werden in Kapitel 3 noch einmal auf das Problem Scheduling auf identischen Maschinen zurückkommen und zeigen, dass auch LPT nicht der beste Approximationsalgorithmus für dieses Problem ist. Wir werden sehen, dass der optimale Schedule in polynomieller Zeit sogar in einem gewissen Sinne beliebig gut approximiert werden kann.

2.4 Rucksackproblem

Das Rucksackproblem haben wir bereits in der Vorlesung „Algorithmen und Berechnungskomplexität II“ als NP-schweres Optimierungsproblem kennengelernt. Der Vollständigkeit halber definieren wir es hier noch einmal.

Rucksackproblem (Knapsack Problem, KP)

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

Wir entwerfen einen Greedy-Algorithmus mit einer Approximationsgüte von $\frac{1}{2}$ für das Rucksackproblem. 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.

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. $B := \emptyset$; $i := 1$;
3. **while** $((i \leq n)$ **and** $(w(B) + w_i \leq t))$
4. $\quad B := B \cup \{i\}$;
5. $\quad i := i + 1$;
6. Sei i^* ein Objekt mit dem größten Nutzen, d. h. $p_{i^*} \geq p_i$ für alle $i \in \{1, \ldots, n\}$.
7. **if** $(p(B) < p_{i^*})$ **then** $A := \{i^*\}$ **else** $A := B$;
8. **return** A;

Theorem 2.6. *Der Algorithmus* Greedy-KP *ist ein* $\frac{1}{2}$*-Approximationsalgorithmus für das Rucksackproblem.*

Beweis. Offensichtlich besitzt der Algorithmus eine 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 t. 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 $w(A') \leq t < w(B) + w_{i+1}$ und $p(A') > p(B) + p_{i+1}$. Dann führt das folgende Austauschargument

zu einem Widerspruch. 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 gezeigt, dass $p(B)+p_{i+1}$ eine obere Schranke für OPT ist.

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 t 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. □

Der Algorithmus Greedy-KP ist nicht der beste Approximationsalgorithmus für das Rucksackproblem. Genauso wie für das Problem Scheduling auf identischen Maschinen werden wir auch für das Rucksackproblem in Kapitel 3 zeigen, dass es in polynomieller Zeit beliebig gut approximiert werden kann.

Kapitel 3

Runden und dynamische Programmierung

Im vorangegangenen Kapitel haben wir bereits angedeutet, dass die Approximationsalgorithmen, die wir für das Rucksackproblem und das Problem Scheduling auf identischen Maschinen kennengelernt haben, nicht die bestmöglichen Approximationsalgorithmen für diese Probleme sind. Wir werden in diesem Kapitel zeigen, dass für beide Probleme sogenannte *Approximationsschemata* existieren. Dabei handelt es sich um Algorithmen, denen zusätzlich zu der eigentlichen Eingabe des Problems noch eine Zahl $\varepsilon > 0$ übergeben wird, und die eine $(1+\varepsilon)$-Approximation bzw. $(1-\varepsilon)$-Approximation berechnen. Es handelt sich also um Approximationsalgorithmen, denen man als zusätzliche Eingabe die gewünschte Approximationsgüte übergeben kann.

Definition 3.1. *Ein* Approximationsschema *A für ein Optimierungsproblem Π ist ein Algorithmus, der zu jeder Eingabe der Form (I, ε) mit $I \in \mathcal{I}_\Pi$ und $\varepsilon > 0$ eine Lösung $A(I,\varepsilon) \in \mathcal{S}_I$ berechnet. Dabei muss für den Wert $w_A(I,\varepsilon) = f_I(A(I,\varepsilon))$ dieser Lösung für jede Eingabe (I,ε) bei einem Minimierungs- oder Maximierungsproblem Π die Ungleichung*

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

gelten.

Ein Approximationsschema A heißt polynomielles Approximationsschema (PTAS[1])*, wenn die Laufzeit von A für jede feste Wahl von $\varepsilon > 0$ durch ein Polynom in $|I|$ nach oben beschränkt ist.*

Wir nennen ein Approximationsschema A voll-polynomielles Approximationsschema (FPTAS[2])*, wenn die Laufzeit von A durch ein bivariates Polynom in $|I|$ und $1/\varepsilon$ nach oben 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 $|I|$

[1] Die Abkürzung geht auf die englische Bezeichnung *polynomial-time approximation scheme* zurück.

[2] Die Abkürzung geht auf die englische Bezeichnung *fully polynomial-time approximation scheme* zurück.

(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. $\Theta(|I|^3/\varepsilon^2)$.

Ein PTAS impliziert, dass es für das entsprechende Problem für jede Konstante $\varepsilon > 0$ einen $(1+\varepsilon)$-Approximationsalgorithmus bzw. $(1-\varepsilon)$-Approximationsalgorithmus 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 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.

3.1 Rucksackproblem

3.1.1 Dynamische Programmierung

Das Approximationsschema, das wir für das Rucksackproblem entwerfen werden, nutzt als wesentlichen Baustein den pseudopolynomiellen Algorithmus zur Lösung des Rucksackproblems, den wir in der Vorlesung „Algorithmen und Berechnungskomplexität I“ kennengelernt haben. Wir werden diesen Algorithmus noch einmal wiederholen. Dazu überlegen wir zunächst, wie eine gegebene Instanz in kleinere Teilinstanzen zerlegt werden kann. Zunächst ist es naheliegend, Teilinstanzen zu betrachten, die nur eine Teilmenge der Objekte enthalten. Das alleine reicht aber noch nicht aus.

Sei eine Instanz mit n Objekten gegeben. Wir bezeichnen wieder mit $p_1, \dots, p_n \in \mathbb{N}$ die Nutzenwerte, mit $w_1, \dots, w_n \in \mathbb{N}$ die Gewichte und mit $t \in \mathbb{N}$ die Kapazität des Rucksacks. Zunächst definieren wir $P = \max_{i \in \{1,\dots,n\}} p_i$. Dann ist nP eine obere Schranke für den Nutzen einer optimalen Lösung. Wir definieren für jede Kombination aus $i \in \{1, \dots, n\}$ und $p \in \{0, \dots, nP\}$ folgendes Teilproblem: finde unter allen Teilmengen der Objekte $1, \dots, i$ mit Gesamtnutzen mindestens p eine mit dem kleinsten Gewicht. Es sei $W(i,p)$ das Gewicht dieser Teilmenge. Existiert keine solche Teilmenge, so setzen wir $W(i,p) = \infty$.

Wir können die Bedeutung von $W(i,p)$ auch wie folgt zusammenfassen: Für jede Teilmenge $I \subseteq \{1, \dots, i\}$ 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 außerdem eine Teilmenge $I \subseteq \{1, \dots, i\}$ 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 $i \in \{1, \dots, n\}$ und $p \in \{0, \dots, nP\}$ 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 beträgt dann 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 $i \in \{1, \dots, n\}$ 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 kleinstmöglichem 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 $w(I') < w(I \setminus \{i\})$, 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 t ist, so testen wir einfach, ob $W(n,p) \leq t$ gilt. Um den maximal erreichbaren Nutzen mit einem Rucksack der Kapazität t zu finden, genügt es, in der Tabelle das größte p zu finden, für das $W(n,p) \leq t$ gilt.

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

DynKP

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

Theorem 3.2. *Der Algorithmus* DynKP *bestimmt in Zeit* $\Theta(n^2 P)$ *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

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

Das ist mit den Vorüberlegungen 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 DynKP

Wir führen den Algorithmus DynKP 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 DynKP 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 Kapazität 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 Kapazität 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 $I \subseteq \{1,\ldots,i\}$ 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 $I(i,p) = I(i-1,p-p_i)\cup\{i\}$.

Kann der Algorithmus DynKP als effizient angesehen werden? Die Laufzeit $\Theta(n^2P)$ weist einen gravierenden Unterschied zu den Laufzeiten der Algorithmen auf, die wir bislang in der Vorlesung gesehen haben. Die Laufzeit hängt nämlich nicht nur von der Anzahl der Objekte in der Eingabe ab, sondern zusätzlich noch von dem maximalen Nutzen P. Im Gegensatz dazu hängt die Laufzeit von GreedyKP beispielsweise nur von der Anzahl der Objekte ab.

Dieser auf den ersten Blick kleine Unterschied hat große Auswirkungen. Bei den Algorithmen, die wir bisher gesehen haben, kann die Laufzeit polynomiell in der Eingabelänge beschränkt werden, wobei wir als Eingabelänge die Anzahl an Bits auffassen, die benötigt werden, um die Eingabe zu codieren. Betrachten wir hingegen beispielsweise eine Eingabe für das Rucksackproblem mit n Objekten, in der alle Gewichte, alle Nutzenwerte und die Kapazität binär codierte Zahlen mit jeweils n Bits sind, dann beträgt die Eingabelänge $\Theta(n^2)$. Da wir mit n Bits Zahlen bis 2^n-1 darstellen können, gilt im Worst Case $P = 2^n-1$. Die Laufzeit von DynKP beträgt dann $\Theta(n^2P) = \Theta(n^2 2^n)$, was exponentiell in der Eingabelänge ist.

Anders als bei den Algorithmen, die wir bisher gesehen haben, können bei DynKP also bereits Eingaben, die sich mit wenigen Bytes codieren lassen, zu einer sehr großen Laufzeit führen. Der Algorithmus DynKP löst aber zumindest diejenigen Instanzen effizient, in denen alle Nutzenwerte polynomiell in der Anzahl der Objekte n beschränkt sind. Instanzen, in denen $p_i \leq n^2$ für jedes i gilt, werden beispielsweise in Zeit $O(n^4)$ gelöst.

Der Algorithmus DynKP ist ein pseudopolynomieller Algorithmus gemäß der folgenden Definition.

Definition 3.3. *Es sei* Π *ein Optimierungsproblem, in dessen Instanzen ganze Zahlen enthalten sind. Ein Algorithmus* A *zur Lösung von* Π *heißt* pseudopolynomieller Algorithmus*, wenn seine Laufzeit durch ein Polynom in der Eingabelänge und dem größten Absolutwert der Zahlen der Eingabe nach oben beschränkt ist.*

Äquivalent zu der vorangegangenen Definition kann man pseudopolynomielle Algorithmen auch wie folgt beschreiben. Sei Π_u die Modifikation des Problems Π, in der alle Zahlen in der Eingabe unär statt binär codiert werden. Der Algorithmus A für Π besitzt genau dann eine pseudopolynomielle Laufzeit wenn er ein polynomieller Algorithmus für Π_u ist. Der Leser überlege sich, dass dies äquivalent zu Definition 3.3 ist.

3.1.2 Approximationsschema

Wir entwerfen nun ein FPTAS für das Rucksackproblem. Diesem liegt die folgende Idee zu Grunde: Mithilfe von DynKP 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 $I \subseteq \{1, \dots, n\}$ 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 den Algorithmus DynKP anwenden, um die optimale Lösung für die skalierten und gerundeten Nutzenwerte zu bestimmen. Wählen wir den Skalierungsfaktor K richtig, so sind diese Zahlen so klein, dass DynKP 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 geben den Algorithmus nun formal an.

FPTAS-KP

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

1. $P := \max_{i \in \{1,\dots,n\}} p_i$;
2. $K := \frac{\varepsilon P}{n}$; // Skalierungsfaktor
3. **for** $i = 1$ **to** n **do** $p'_i = \lfloor p_i / K \rfloor$; // skaliere und runde die Nutzenwerte
4. Benutze Algorithmus DynKP, um die optimale Lösung für die Instanz $p'_1, \dots, p'_n, w_1, \dots, w_n, t$ des Rucksackproblems zu bestimmen.

Theorem 3.4. *Der Algorithmus* FPTAS-KP *ist ein FPTAS für das Rucksackproblem mit einer Laufzeit von* $O(n^3/\varepsilon)$.

Beweis. Wir betrachten zunächst die Laufzeit des Algorithmus. Diese wird durch den Aufruf von DynKP dominiert, da die Skalierung der Nutzenwerte in den ersten drei Schritten in Linearzeit erfolgt. Sei $P' = \max_{i \in \{1,\dots,n\}} p'_i$ der größte der skalierten Nutzenwerte. Dann beträgt die Laufzeit von DynKP $\Theta(n^2 P')$ gemäß Theorem 3.2. Es gilt

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

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

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 Nutzenwerten $p_1, \dots, p_n$. Da das Skalieren der Nutzenwerte die

optimale Lösung nicht ändert, ist I' auch eine optimale Lösung für die Nutzenwerte $p_1^*, \ldots, p_n^*$ 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:

$$\begin{array}{lll} p_1 = 33 & p_1' = 3 & p_1^* = 30 \\ p_2 = 25 & p_2' = 2 & p_2^* = 20 \\ p_3 = 50 & p_3' = 5 & p_3^* = 50 \\ p_4 = 27 & p_4' = 2 & p_4^* = 20 \end{array}$$

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 $J \subseteq \{1, \ldots, n\}$ bezeichnen wir mit $p(J)$ bzw. $p^*(J)$ ihren Gesamtnutzen bezüglich der ursprünglichen Nutzenwerte und ihren Gesamtnutzen bezüglich der Nutzenwerte $p_1^*, \ldots, p_n^*$:

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

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 $J \subseteq \{1, \ldots, n\}$

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

Wir wissen, dass I' eine optimale Lösung für die Nutzenwerte $p_1^*, \ldots, p_n^*$ ist, da diese durch Skalieren aus den Nutzenwerten $p_1', \ldots, p_n'$ hervorgehen. 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

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

womit der Beweis abgeschlossen ist. □

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 $\Theta(n^2 W)$ beträgt, 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 mithilfe des pseudopolynomiellen Algorithmus die Instanz mit gerundeten Gewichten zu lösen. Das Ergebnis ist dann aber im Allgemeinen keine 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 die optimale Lösung. 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 Vorsicht geboten.

3.2 Scheduling auf identischen Maschinen

Wir haben in Abschnitt 2.3 mit LPT einen 4/3-Approximationsalgorithmus für das Problem Scheduling auf identischen Maschinen kennengelernt. In diesem Abschnitt werden wir für dieses Problem ein PTAS entwerfen. Wir haben in Abschnitt 2.3 gesehen, dass sich der Approximationsfaktor von Least-Loaded durch $1 + p_{\max}/\mathrm{OPT}$ beschränken lässt, wobei $p_{\max}$ die Größe des größten Jobs bezeichnet. Sind also alle Jobs kleiner als OPT/3, so liefert Least-Loaded bereits einen besseren Approximationsfaktor als 4/3.

Wir werden dem nun dadurch Rechnung tragen, dass wir die Menge der Jobs in zwei Klassen einteilen. Streben wir einen Approximationsfaktor von $1 + \varepsilon$ für ein konstantes $\varepsilon > 0$ an, so nennen wir einen Job j *klein*, falls $p_j \leq \frac{\varepsilon}{m} \sum_{k=1}^{n} p_k$ gilt, und ansonsten *groß*. Zunächst ignorieren wir alle kleinen Jobs und berechnen für die Instanz $I_{\text{groß}}$, die nur aus den großen Jobs besteht, eine optimale Lösung. Anschließend fügen wir

die kleinen Jobs mit dem LEAST-LOADED-Algorithmus dem optimalen Schedule der großen Jobs hinzu. Wir nennen diesen Algorithmus im Folgenden A_ε.

Um den optimalen Schedule der großen Jobs zu berechnen, nutzen wir eine Brute-Force-Suche, die alle möglichen Verteilungen der großen Jobs auf die Maschinen testet. Da die Gesamtgröße aller Jobs der Summe $\sum_{k=1}^{n} p_k$ entspricht, kann es höchstens m/ε viele große Jobs geben. Es gibt höchstens $m^{m/\varepsilon}$ Möglichkeiten, diese Jobs auf die m Maschinen zu verteilen. Ist die Anzahl der Maschinen konstant, so ist auch diese Anzahl eine Konstante und der Algorithmus A_ε benötigt insgesamt nur eine polynomielle Laufzeit.

Sei π der Schedule, den A_ε berechnet. Um den Approximationsfaktor dieses Schedules abzuschätzen, betrachten wir eine kritische Maschine i, also eine Maschine, die in π die maximale Ausführungszeit besitzt. Enthält die kritische Maschine i keinen kleinen Job, so stimmt der Makespan $C(\pi)$ von π mit dem optimalen Makespan $\mathrm{OPT}(I_{\text{groß}})$ der Instanz $I_{\text{groß}}$ überein, die nur die großen Jobs enthält. Dieser Makespan ist durch den optimalen Makespan OPT der Gesamtinstanz I nach oben beschränkt. Damit ist π sogar ein optimaler Schedule: $C(\pi) = \mathrm{OPT}(I_{\text{groß}}) \leq \mathrm{OPT}(I)$.

Es bleibt, den Fall zu betrachten, dass die kritische Maschine mindestens einen kleinen Job enthält. Sei p_j der letzte kleine Job, der Maschine i hinzugefügt wird. Dann erhalten wir analog zu den Überlegungen in Abschnitt 2.3

$$\begin{aligned} C(\pi) &\leq \frac{1}{m}\left(\sum_{k=1}^{n} p_k\right) + p_j \\ &\leq \frac{1}{m}\left(\sum_{k=1}^{n} p_k\right) + \frac{\varepsilon}{m}\left(\sum_{k=1}^{n} p_k\right) \\ &= (1+\varepsilon)\cdot\frac{1}{m}\left(\sum_{k=1}^{n} p_k\right) \\ &\leq (1+\varepsilon)\cdot\mathrm{OPT}(I), \end{aligned} \tag{3.1}$$

wobei wir in der zweiten Ungleichung ausgenutzt haben, dass j ein kleiner Job ist.

Insgesamt erhalten wir das folgende Resultat.

Theorem 3.5. *Für das Problem Scheduling auf identischen Maschinen existiert ein PTAS, wenn die Anzahl an Maschinen durch eine Konstante m nach oben beschränkt ist. Die Laufzeit dieses PTAS beträgt $O(m^{m/\varepsilon} \cdot \mathrm{poly}(n, m))$.*

Im Folgenden werden wir uns mit der Frage beschäftigen, wie der Algorithmus A_ε beschleunigt werden kann, sodass er auch dann zu einem PTAS führt, wenn die Anzahl an Maschinen nicht mehr konstant ist. Einen ersten Hinweis liefert die Beobachtung, dass es für den Approximationsfaktor nicht wichtig ist, den optimalen Schedule der großen Jobs zu finden. Tatsächlich würde es für das obige Argument bereits genügen, für diesen Schedule eine $(1+\varepsilon)$-Approximation zu berechnen. In den folgenden Abschnitten beschreiben wir, wie eine solche Approximation in polynomieller Zeit berechnet werden kann. Dazu ist ein Umweg über das Bin-Packing-Problem hilfreich.

3.2.1 Bin-Packing mit konstant vielen Objektgrößen

Beim *Bin-Packing-Problem* besteht die Aufgabe darin, eine gegebene Menge von Objekten mit potentiell unterschiedlichen Größen in möglichst wenige Klassen (Bins) einzuteilen, sodass keine Klasse eine vorgegebene Gesamtgröße überschreitet. Formal ist das Problem wie folgt definiert.

Bin Packing

Eingabe:	Objektgrößen $s_1, \ldots, s_n \in \mathbb{N}$, Kapazität $T \in \mathbb{N}$
Lösungen:	alle Partitionen $C_1, \ldots, C_\ell$ der Menge $\{1, \ldots, n\}$ mit $\sum_{i \in C_j} s_i \leq T$ für alle $j \in \{1, \ldots, \ell\}$
Zielfunktion:	minimiere ℓ

Das Bin-Packing-Problem ist im Allgemeinen NP-schwer. Wir betrachten hier aber nur den Spezialfall, dass die Anzahl an unterschiedlichen Objektgrößen durch eine Konstante beschränkt ist, und zeigen, dass dieser Spezialfall in polynomieller Zeit mittels dynamischer Programmierung gelöst werden kann.

Wir beschreiben im Folgenden Eingaben für das Bin-Packing-Problem durch zwei Vektoren $(s_1, \ldots, s_z) \in \mathbb{N}^z$ und $(n_1, \ldots, n_z) \in \mathbb{N}^z$ sowie die Kapazität $T \in \mathbb{N}$. Dabei seien $s_1, \ldots, s_z \in \mathbb{N}$ die verschiedenen Objektgrößen und n_i gebe an, wie viele Objekte mit Größe s_i in der Eingabe vorhanden sind. Wir setzen $n = \sum_{i=1}^{z} n_i$. Sei im Folgenden eine solche Eingabe fixiert.

Wir betrachten nun modifizierte Eingaben, in denen die Objektgrößen $(s_1, \ldots, s_z)$ und die Kapazität T unverändert bleiben, in denen aber die Anzahl der Objekte angepasst wird. Dazu bezeichne $\mathrm{BINS}(i_1, \ldots, i_z)$ für einen Vektor $(i_1, \ldots, i_z) \in \mathbb{N}_0^z$ den Wert der optimalen Lösung für die Eingabe bestehend aus den Objektgrößen $(s_1, \ldots, s_z)$, den Anzahlen $(i_1, \ldots, i_z)$ und der Kapazität T. Der Wert $\mathrm{BINS}(i_1, \ldots, i_z)$ gibt also an, wie viele Bins in der optimalen Partition benötigt werden, wenn es in der Eingabe für jedes $j \in \{1, \ldots, z\}$ genau i_j Objekte der Größe s_j gibt. Wir werden ein dynamisches Programm angeben, das für jeden Vektor $(i_1, \ldots, i_z) \in \mathbb{N}_0^z$ mit $(i_1, \ldots, i_z) \leq (n_1, \ldots, n_z)$ den Wert $\mathrm{BINS}(i_1, \ldots, i_z)$ berechnet. Diese Werte werden in einer z-dimensionalen Tabelle gespeichert. Die Anzahl der Zellen in dieser Tabelle kann durch $\prod_{i=1}^{z}(n_i + 1) \leq (n+1)^z = O(n^z)$ nach oben abgeschätzt werden (die Aussage $(n+1)^z = O(n^z)$ gilt nur, da z eine Konstante ist).

Als erstes betrachten wir die Menge $C \subseteq \mathbb{N}_0^z$ der sogenannten Binkonfigurationen. Dabei gelte $(i_1, \ldots, i_z) \in C$ genau dann, wenn $\sum_{j=1}^{z} i_j s_j \leq T$ und $i_1 + \ldots + i_z > 0$ gilt. Wir setzen $\mathrm{BINS}(0, 0, \ldots, 0) = 0$ und $\mathrm{BINS}(i_1, \ldots, i_z) = 1$ für alle $(i_1, \ldots, i_z) \in C$.

Sei $(i_1, \ldots, i_z) \notin C \cup \{(0, 0, \ldots, 0)\}$. Da $(i_1, \ldots, i_z)$ laut Voraussetzung keine Binkonfiguration und nicht der Nullvektor ist, gilt $\mathrm{BINS}(i_1, \ldots, i_z) \geq 2$. Wir betrachten eine optimale Partition der durch $(i_1, \ldots, i_z)$ beschriebenen Objekte und in dieser optimalen Partition einen beliebigen Bin $(c_1, \ldots, c_z)$ (der betrachtete Bin enthält für jedes j also c_j Objekte der Größe s_j). Bei $(c_1, \ldots, c_z)$ muss es sich um eine Binkonfiguration handeln. Da wir eine optimale Partition der Objekte betrachten, müssen die Objekte in den anderen Bins ebenfalls bestmöglich aufgeteilt sein. Es gilt also

$$\mathrm{BINS}(i_1, \ldots, i_z) = 1 + \mathrm{BINS}(i_1 - c_1, \ldots, i_z - c_z).$$

Da wir im Allgemeinen nicht wissen, wie die optimale Partition aussieht, testen wir alle Möglichkeiten für den ersten Bin und nehmen die beste. Dies führt zu der Formel

$$\mathrm{BINS}(i_1, \ldots, i_z) = 1 + \min_{(c_1,\ldots,c_z)\in C} \mathrm{BINS}(i_1 - c_1, \ldots, i_z - c_z). \tag{3.2}$$

Wir haben oben bereits diskutiert, dass die Tabelle $O(n^z)$ Zellen enthält, die gefüllt werden müssen. Mit demselben Argument ist auch die Anzahl an Binkonfigurationen durch $O(n^z)$ nach oben beschränkt. Somit benötigt das Auswerten des Minimums in (3.2) eine Laufzeit von $O(|C|) = O(n^z)$. Multiplizieren wir dies mit der Anzahl an Zellen, so erhalten wir das folgende Theorem.

Theorem 3.6. *Das Bin-Packing-Problem mit z verschiedenen Objektgrößen kann für Instanzen mit n Objekten in Zeit $O(n^{2z})$ gelöst werden (wobei die Konstante in der O-Notation von z abhängt).*

3.2.2 Approximationsschema

Sei nun wieder eine Instanz I für Scheduling auf identischen Maschinen mit n Jobs mit Größen $p_1, \ldots, p_n \in \mathbb{N}$ und m Maschinen gegeben. Wir bezeichnen mit $\mathrm{OPT} = \mathrm{OPT}(I)$ den optimalen Makespan dieser Instanz. Mithilfe des dynamischen Programms für das Bin-Packing-Problem aus dem letzten Abschnitt werden wir für jedes $\varepsilon > 0$ einen Algorithmus B_ε entwerfen, der zusätzlich zu der Eingabe für das Scheduling-Problem eine Zahl $T \in \mathbb{N}$ erhält. Dieser Algorithmus liefert entweder einen Beweis dafür, dass $\mathrm{OPT}(I) > T$ gilt, oder er berechnet einen Schedule mit Makespan höchstens $(1+\varepsilon)T$. Wir gehen im Folgenden davon aus, dass $T \geq p_{\max}$ und $T \geq \frac{1}{m}\sum_{j=1}^{n} p_j$ gilt, da ansonsten bereits klar ist, dass kein Schedule mit Makespan höchstens T existiert.

Wir nennen einen Job j groß, falls $p_j > \varepsilon T$ gilt. Ansonsten nennen wir den Job klein. Um mithilfe von Theorem 3.6 eine gute Verteilung der großen Jobs berechnen zu können, müssen wir zunächst erreichen, dass die großen Jobs nur wenige verschiedene Größen haben. Dazu runden wir jeden großen Job zunächst auf das nächste Vielfache von $\varepsilon^2 T$ ab. Da die Längen aller großen Jobs im Intervall $(\varepsilon T, T]$ liegen, besitzen die großen Jobs nach dem Runden nur noch maximal $1/\varepsilon^2$ viele verschiedene Größen. Sei I' die Instanz, die nur aus den gerundeten großen Jobs besteht, und sei $\mathrm{OPT}(I')$ der Makespan des optimalen Schedules dieser Instanz.

Wir nutzen Lemma 3.7, um in Zeit $O(n^{2/\varepsilon^2})$ zu testen, ob $\mathrm{OPT}(I') \leq T$ gilt. Dazu konstruieren wir eine Eingabe des Bin-Packing-Problems mit den Jobgrößen aus I' und der Kapazität $T \in \mathbb{N}$. Gibt das dynamische Programm aus Theorem 3.6 aus, dass die optimale Anzahl an Bins in dieser Instanz kleiner oder gleich m ist, so gilt $\mathrm{OPT}(I') \leq T$. In diesem Fall bezeichnen wir mit π' die Partition der großen Jobs, die von dem dynamischen Programm berechnet wird. Werden in der optimalen Partition mehr als m Bins mit Kapazität T benötigt, so gilt $\mathrm{OPT}(I') > T$.

Gilt $\mathrm{OPT}(I') > T$, so gilt insbesondere $\mathrm{OPT}(I) > T$, da die Instanz I' nur eine Teilmenge der Jobs von Instanz I mit abgerundeten Größen enthält. In diesem Fall

kann B_ε also mit der Ausgabe $\mathrm{OPT}(I) > T$ terminieren. Gehen wir nun davon aus, dass $\mathrm{OPT}(I') \leq T$ gilt. In diesem Fall berechnen wir wie folgt eine Lösung für die Instanz I: Für die großen Jobs übernehmen wir den Schedule π' der Instanz I' und weisen die kleinen Jobs anschließend mit dem Least-Loaded-Algorithmus zu. Es sei π der so berechnete Schedule. Dieser wird von B_ε ausgegeben. Der folgende Pseudocode stellt den Algorithmus B_ε noch einmal dar.

$B_\varepsilon(T)$

1. Sei $J' = \{j \in J \mid p_j > \varepsilon T\}$ die Menge der großen Jobs.
2. Für $j \in J'$ sei p'_j der Wert p_j auf das nächste Vielfache von $\varepsilon^2 T$ abgerundet. Sei I' die Instanz, die nur aus Jobs mit den Größen p'_j für $j \in J'$ besteht.
3. Konstruiere eine Instanz für Bin Packing mit Objektgrößen $\{p'_j \mid j \in J'\}$ und Kapazität T. Bezeichne a die minimale Anzahl an Bins, die für diese Instanz benötigt werden. Berechne a und die dazu gehörige Verteilung π' der Objekte mithilfe des dynamischen Programms aus Theorem 3.6.
4. Gilt $a > m$, so gib $\mathrm{OPT}(I) \geq \mathrm{OPT}(I') > T$ aus.
5. Ansonsten erweitere die Verteilung π' auf alle Jobs aus J durch Hinzufügen der Jobs aus $J \setminus J'$ mittels des Least-Loaded-Algorithmus. Gib den resultierenden Schedule π aus.

Um den Makespan von π zu bestimmen, betrachten wir wieder eine kritische Maschine i. Enthält i mindestens einen kleinen Job j, so folgt analog zu (3.1)

$$C(\pi) = L_i(\pi) \leq \frac{1}{m}\left(\sum_{k=1}^{n} p_k\right) + p_j \leq T + \varepsilon T = (1+\varepsilon)T.$$

Ist der kritischen Maschine i kein kleiner Job zugewiesen, so beträgt die Ausführungszeit von Maschine i bezüglich der abgerundeten Jobgrößen höchstens T. Da die großen Jobs eine Länge größer als εT besitzen, kann i höchstens $1/\varepsilon$ viele Jobs im Schedule π' enthalten. Sei S die Menge dieser Jobs und für einen Job $j \in S$ bezeichne p'_j den Wert, auf den p_j in I' abgerundet wird. Da wir auf das nächste Vielfache von $\varepsilon^2 T$ abrunden, gilt $p_j - p'_j \leq \varepsilon^2 T$. Insgesamt gilt in diesem Fall

$$C(\pi) = L_i(\pi) = \sum_{j \in S} p_j \leq \sum_{j \in S} (p'_j + \varepsilon^2 T) \leq \mathrm{OPT}(I') + |S|\varepsilon^2 T \leq T + \frac{1}{\varepsilon} \cdot \varepsilon^2 T = (1+\varepsilon)T.$$

Somit ergibt sich das folgende Zwischenergebnis.

Lemma 3.7. *Der Algorithmus B_ε liefert in Zeit $O(n^{2/\varepsilon^2})$ entweder einen Beweis dafür, dass $\mathrm{OPT} > T$ gilt, oder er berechnet einen Schedule mit Makespan höchstens $(1+\varepsilon)T$.*

Basierend auf dem vorangegangenen Lemma können wir mithilfe einer binären Suche eine $(1+\varepsilon)$-Approximation des optimalen Schedules berechnen. Aus den Überlegungen in Abschnitt 2.3 wissen wir, dass $\mathrm{OPT} \in [L, U]$ für

$$L = \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} p_j \right\rceil, p_{\max}\right\} \quad \text{und} \quad U = \left\lceil \frac{1}{m}\sum_{j=1}^{n} p_j \right\rceil + p_{\max}$$

gilt. Die untere Schranke L haben wir in Abschnitt 2.3 bewiesen (da der optimale Makespan ganzzahlig ist, können wir den ersten Term im Maximum aufrunden) und die obere Schranke U wird durch den Least-Loaded-Algorithmus erreicht. Es gilt $U - L \leq p_{\max}$. Wir werden nun das Intervall $[L, U]$ sukzessive halbieren und dabei die folgende Invariante aufrecht erhalten: es gilt $L \leq \text{OPT}$ und wir können einen Schedule mit Makespan höchstens $(1+\varepsilon)U$ in Zeit $O(n^{2/\varepsilon^2})$ berechnen. Diese Invariante ist für die initialen Schranken L und U per Konstruktion erfüllt (der zweite Teil folgt daraus, dass der Least-Loaded-Algorithmus einen Schedule mit Makespan höchstens U berechnet).

Wir wenden nun Lemma 3.7 mit $T = \lfloor \frac{L+U}{2} \rfloor$ an. Erhalten wir als Ausgabe, dass der optimale Makespan größer als T ist, so setzen wir $L = T + 1$ und belassen U bei seinem Wert. Erhalten wir als Ausgabe einen Schedule mit Makespan höchstens $(1 + \varepsilon)T$, so setzen wir $U = T$ und belassen L bei seinem Wert. In beiden Fällen bleibt die Invariante erhalten und die Länge des Intervalls $[L, U]$ halbiert sich. Nach i Halbierungen beträgt die Intervalllänge höchstens $p_{\max}/2^i$. Nach $\log_2(p_{\max}) + 1$ Schritten ist die Differenz zwischen der oberen und unteren Schranke kleiner als 1. Wegen der Ganzzahligkeit folgt dann $L = U$. Dann folgt aus der Invariante, dass wir in Zeit $O(n^{2/\varepsilon^2})$ einen Schedule mit Makespan höchstens $(1+\varepsilon)U = (1+\varepsilon)L \leq (1+\varepsilon)\text{OPT}$ berechnen können.

Theorem 3.8. *Für das Problem Scheduling auf identischen Maschinen existiert ein PTAS mit Laufzeit* $O(n^{2/\varepsilon^2} \cdot \log(p_{\max}))$.

3.2.3 Untere Schranke für Approximierbarkeit

Die Laufzeit des Approximationsschemas aus Theorem 3.8 hängt exponentiell von $1/\varepsilon$ ab. Somit handelt es sich nicht um ein voll-polynomielles Approximationsschema, was direkt die Frage aufwirft, ob ein solches für das Problem Scheduling auf identischen Maschinen existiert. Diese Frage beantworten wir in diesem Abschnitt negativ. Zunächst führen wir dafür den Begriff *stark NP-schwer* ein.

Definition 3.9. *Es sei* Π *ein Optimierungsproblem, in dessen Instanzen ganze Zahlen enthalten sind. Das Problem* Π *heißt* stark NP-schwer*, wenn es ein Polynom* q *gibt, sodass* Π *bereits eingeschränkt auf solche Instanzen* I *NP-schwer ist, in denen die Absolutwerte aller Zahlen durch* $q(|I|)$ *nach oben beschränkt sind.*

Es ist leicht zu zeigen, dass ein Problem genau dann stark NP-schwer ist, wenn es bereits dann NP-schwer ist, wenn alle Zahlen in der Eingabe unär codiert werden. Aus der Definition folgt direkt, dass die Existenz eines pseudopolynomiellen Algorithmus für ein stark NP-schweres Problem P = NP implizieren würde.

Bei dem Problem Scheduling auf identischen Maschinen handelt es sich um ein stark NP-schweres Problem, was aus einer polynomiellen Reduktion von dem stark NP-schweren Problem 3-Partition folgt [1].

Theorem 3.10. *Unter der Annahme* $P \neq NP$ *existiert für das Problem Scheduling auf identischen Maschinen kein FPTAS.*

Beweis. Wir gehen davon aus, dass ein FPTAS A für das Problem Scheduling auf identischen Maschinen existiert. Da das Problem stark NP-schwer ist, existiert ein Polynom q, sodass es bereits NP-schwer ist, Eingaben mit Jobgrößen $p_1, \ldots, p_n \in \{1, 2, \ldots, q(n)\}$ zu lösen. Wir zeigen nun, dass dieser Spezialfall mithilfe von A in polynomieller Zeit gelöst werden kann, was P = NP impliziert.

Sei eine Instanz I gegeben, in der alle Jobgrößen durch $q(n)$ nach oben beschränkt sind. Dann gilt $\mathrm{OPT}(I) \leq nq(n)$. Wir berechnen für die Instanz I mithilfe von A eine $(1+\varepsilon)$-Approximation mit $\varepsilon = 1/(2nq(n))$. Da die Laufzeit von A nur polynomiell von $1/\varepsilon$ abhängt, ist die Laufzeit dafür polynomiell. Es gilt dann

$$w_A(I, \varepsilon) \leq (1+\varepsilon) \cdot \mathrm{OPT}(I) \leq \mathrm{OPT}(I) + \varepsilon nq(n) = \mathrm{OPT}(I) + \frac{1}{2}.$$

Wegen der Ganzzahligkeit folgt $w_A(I, \varepsilon) = \mathrm{OPT}(I)$ und damit haben wir mithilfe von A in polynomieller Zeit eine optimale Lösung gefunden. □

Theorem 3.10 lässt sich auf eine große Klasse von stark NP-schweren Problemen erweitern. Allgemein kann man festhalten, dass stark NP-schwere Optimierungsprobleme mit ganzzahligen Zielfunktionswerten kein FPTAS besitzen, falls P $\neq$ NP gilt.

Kapitel 4

Randomisierte Approximationsalgorithmen

Wir werden in diesem Kapitel *randomisierte Approximationsalgorithmen* kennenlernen, also Approximationsalgorithmen, die zufällige Entscheidungen treffen. Diese Algorithmen können auf derselben Eingabe im Allgemeinen verschiedene Ausgaben produzieren, je nachdem, welche Entscheidungen sie treffen. In den meisten Fällen kann man nicht garantieren, dass sie stets einen bestimmten Approximationsfaktor erreichen. Deshalb interessieren wir uns für den erwarteten Wert der berechneten Lösung. Ist dieser für jede Eingabe höchstens um einen Faktor r von dem Wert einer optimalen Lösung entfernt, so sagen wir, dass der Algorithmus einen Approximationsfaktor von r erreicht.

Randomisierung spielt beim Entwurf von Approximationsalgorithmen eine große Rolle, da randomisierte Approximationsalgorithmen für viele Probleme einfacher zu entwerfen und zu analysieren sind als deterministische. In diesem Abschnitt führen wir zunächst einige Grundbegriffe der Wahrscheinlichkeitsrechnung ein, die im weiteren Verlauf der Vorlesung benötigt werden. Danach definieren wir den Approximationsfaktor eines randomisierten Algorithmus formal und lernen erste Beispiele für randomisierte Approximationsalgorithmen kennen.

4.1 Grundlagen der Wahrscheinlichkeitsrechnung

4.1.1 Diskrete Wahrscheinlichkeitsräume

Als erstes müssen wir klären, was genau wir unter Zufall verstehen. Dazu geben wir zunächst eine formale Definition, bevor wir uns über die Interpretation Gedanken machen.

Definition 4.1. *Sei Ω eine endliche oder abzählbare Menge, die wir im Folgenden die* Ergebnismenge *nennen werden. Die Elemente von Ω bezeichnen wir als* Elementarereignisse *und jede Teilmenge $A \subseteq \Omega$ bezeichnen wir als* Ereignis*. Für ein Ereignis A bezeichnen wir mit $\overline{A} = \Omega \setminus A$ sein* Gegenereignis*.*

Eine Wahrscheinlichkeit *(oder ein* Wahrscheinlichkeitsmaß*) auf* Ω *ist eine Funktion* $\mathbf{Pr} : 2^\Omega \to [0,1]$*, die jedem Ereignis einen Wert aus dem Intervall* $[0,1]$ *zuweist. Es muss dabei* $\mathbf{Pr}[\Omega] = 1$ *gelten und* $\mathbf{Pr}$ *muss* σ-additiv *sein, d. h. für jede abzählbare Folge* $A_1, A_2, \ldots$ *von paarweise disjunkten Ereignissen muss*

$$\mathbf{Pr}\left[\bigcup_{i=1}^{\infty} A_i\right] = \sum_{i=1}^{\infty} \mathbf{Pr}[A_i]$$

gelten. Das Paar $(\Omega, \mathbf{Pr})$ *nennen wir einen* diskreten Wahrscheinlichkeitsraum.

Eine direkte Konsequenz aus dieser Definition ist, dass $\mathbf{Pr}[\emptyset] = 0$ gelten muss. Es ist äquivalent zu Definition 4.1, die Wahrscheinlichkeit $\mathbf{Pr} : \Omega \to [0,1]$ zunächst nur auf den Elementarereignissen zu definieren und durch $\mathbf{Pr}[A] = \sum_{a \in A} \mathbf{Pr}[a]$ auf beliebige Ereignisse $A \subseteq \Omega$ zu erweitern. Wir müssen dann lediglich $\mathbf{Pr}[\Omega] = 1$ fordern, um einen diskreten Wahrscheinlichkeitsraum im Sinne der obigen Definition zu erhalten.

Bevor wir uns Beispiele für Wahrscheinlichkeitsräume anschauen, überlegen wir uns kurz, wie Wahrscheinlichkeiten zu interpretieren sind. Tatsächlich gibt es verschiedene Möglichkeiten, wir wollen hier aber nur kurz den *frequentistischen Wahrscheinlichkeitsbegriff* ansprechen. Bei diesem interpretiert man die Wahrscheinlichkeit für ein Ereignis einfach als die relative Häufigkeit, mit der es bei einer großen Anzahl unabhängig wiederholter Zufallsexperimente auftritt. Möchten wir also beispielsweise den Wurf eines fairen Würfels modellieren, so setzen wir $\Omega = \{1,2,3,4,5,6\}$ als die Ergebnismenge und $\mathbf{Pr}[i] = 1/6$ für jedes $i \in \Omega$. Die Wahrscheinlichkeit $\mathbf{Pr}[1] = 1/6$ ist dann so zu interpretieren, dass bei einer großen Anzahl unabhängiger Würfe des Würfels in ungefähr einem Sechstel der Fälle die Zahl 1 geworfen wird.

Kommen wir nun zu weiteren Beispielen für diskrete Wahrscheinlichkeitsräume.

- Bei dem obigen Beispiel eines Würfelwurfes entspricht das Ereignis, eine gerade Zahl zu würfeln, der Menge $A = \{2,4,6\}$. Die Wahrscheinlichkeit dafür ergibt sich mittels Additivität zu $\mathbf{Pr}[A] = \mathbf{Pr}[2] + \mathbf{Pr}[4] + \mathbf{Pr}[6] = 1/2$, was unserer Intuition entspricht, dass bei einer großen Anzahl an Würfen in ungefähr der Hälfte der Fälle eine gerade Zahl herauskommt.

- Werfen wir zwei faire Münzen nacheinander, so wählen wir als Ergebnismenge $\Omega = \{(K,K),(K,Z),(Z,K),(Z,Z)\}$, wobei (Z,K) beispielsweise dafür steht, dass die erste Münze „Zahl“ und die zweite „Kopf“ zeigt. Intuitiv ist klar, dass bei fairen Münzen alle Elementarereignisse die gleiche Wahrscheinlichkeit haben. Deshalb setzen wir $\mathbf{Pr}[(K,K)] = \mathbf{Pr}[(K,Z)] = \mathbf{Pr}[(Z,K)] = \mathbf{Pr}[(Z,Z)] = 1/4$. Das Ereignis, dass die erste Münze „Kopf“ ergibt, entspricht dann der Menge $A = \{(K,K),(K,Z)\}$. Gemäß der Additivität ergibt sich $\mathbf{Pr}[A] = \mathbf{Pr}[(K,K)] + \mathbf{Pr}[(K,Z)] = 1/2$, was unserer Intuition entspricht, dass die erste Münze bei einer großen Anzahl Wiederholungen in ungefähr der Hälfte der Fälle „Kopf“ zeigt.

- Wir werfen zwei faire Münzen; diesmal aber nicht nacheinander, sondern gleichzeitig. Gehen wir davon aus, dass wir die Münzen nicht unterscheiden können, dann enthält die Ergebnismenge nur noch drei Elementarereignisse: $\Omega =$

$\{\{K,K\},\{Z,Z\},\{K,Z\}\}$. Entweder beide Münzen zeigen „Kopf“, beide Münzen zeigen „Zahl“ oder eine Münze zeigt „Kopf“ und die andere „Zahl“. Diese drei Elementarereignisse sind aber nicht mehr gleichwahrscheinlich. Man überlegt sich leicht, dass $\mathbf{Pr}[\{K,K\}] = \mathbf{Pr}[\{Z,Z\}] = 1/4$ und $\mathbf{Pr}[\{K,Z\}] = 1/2$ gilt.

Wir betrachten nun einen einfachen randomisierten Algorithmus für das folgende Problem, einen maximalen Schnitt in einem Graphen zu berechnen.

Maximum-Cut Problem (MaxCut)

Eingabe:	ungerichteter Graph $G = (V, E)$
	Kantengewichte $w : E \to \mathbb{R}_{>0}$
Lösungen:	alle Mengen $U \subseteq V$
	wir nennen $(U, V \setminus U)$ einen *Schnitt* des Graphen G
Zielfunktion:	maximiere $\sum_{x \in U} \sum_{y \in V \setminus U} w(x,y)$,
	wobei $w(x,y) = 0$ für $(x,y) \notin E$ gilt

Der randomisierte Approximationsalgorithmus RandMaxCut für das Maximum-Cut-Problem fügt jeden Knoten des Graphen mit einer Wahrscheinlichkeit von $1/2$ in die Menge U ein. Sei $V = \{v_1, \ldots, v_n\}$. Wir modellieren dies durch einen Wahrscheinlichkeitsraum $(\Omega, \mathbf{Pr})$ mit $\Omega = \{0,1\}^n$. Ein Elementarereignis $\omega \in \Omega$ entspricht der Menge $U = \{v_i \in V \mid \omega_i = 1\}$.

Sei nun $e = (v_i, v_j) \in E$ eine beliebige Kante des Graphen. Wir sagen, dass die Kante e *über den Schnitt* $(U, V \setminus U)$ *läuft*, wenn genau einer der beiden Knoten v_i und v_j zu der Menge U gehört. Es gilt

$$\mathbf{Pr}[e \text{ läuft über den Schnitt } (U, V \setminus U)] = \mathbf{Pr}[\{\omega \in \Omega \mid \omega_i \neq \omega_j\}] = \frac{2^{n-1}}{2^n} = \frac{1}{2}.$$

Die zweite Gleichung ergibt sich aus der folgenden Überlegung: Es gilt $|\Omega| = 2^n$ und jedes Elementarereignis aus Ω besitzt die gleiche Wahrscheinlichkeit $1/2^n$ realisiert zu werden. Außerdem gibt es 2^{n-1} Elementarereignisse, die die Bedingung $\omega_i \neq \omega_j$ erfüllen.

Intuitiv hätten wir uns zur Bestimmung der obigen Wahrscheinlichkeit auch auf die beiden Knoten v_i und v_j beschränken können: es gibt vier Möglichkeiten für $\{v_i, v_j\} \cap U$ und davon führen zwei dazu, dass e über den Schnitt läuft. Dies liefert ebenfalls direkt die Wahrscheinlichkeit $1/2$. Diese Argumentation ist zulässig, da v_i und v_j *unabhängig* von den anderen Knoten mit Wahrscheinlichkeit $1/2$ zur Menge U hinzugefügt werden. Dies werden wir im folgenden Abschnitt formalisieren.

4.1.2 Unabhängigkeit und bedingte Wahrscheinlichkeit

Wir definieren nun allgemein, was es bedeutet, dass zwei Ereignisse in einem Wahrscheinlichkeitsraum unabhängig sind.

Definition 4.2. *Sei* $(\Omega, \mathbf{Pr})$ *ein diskreter Wahrscheinlichkeitsraum. Wir nennen zwei Ereignisse* $A \subseteq \Omega$ *und* $B \subseteq \Omega$ unabhängig*, wenn* $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$ *gilt. Wir nennen eine Folge* $A_1, \ldots, A_k$ *von Ereignissen* unabhängig*, wenn für jede Teilmenge* $I \subseteq \{1, \ldots, k\}$

$$\mathbf{Pr}\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \mathbf{Pr}[A_i]$$

gilt.

Intuitiv bedeutet die Unabhängigkeit zweier Ereignisse A und B, dass Informationen über das Eintreten oder Nichteintreten von A keinerlei Konsequenzen für B haben und umgekehrt. Zum besseren Verständnis geben wir einige Beispiele an.

- Sei $\Omega = \{(i,j) \mid i,j \in \{1,\ldots,6\}\}$ und $\mathbf{Pr}[(i,j)] = 1/36$ für alle $(i,j) \in \Omega$ die Modellierung zweier unabhängiger Würfe eines Würfels. Intuitiv hat das Ergebnis des ersten Wurfes keinerlei Konsequenzen für das Ergebnis des zweiten Wurfes. Bezeichne A beispielsweise das Ereignis, dass der erste Wurf ein gerades Ergebnis zurückliefert, und bezeichne B das Ereignis, dass der zweite Wurf mindestens den Wert drei zurückliefert. Dann rechnet man leicht nach, dass $\mathbf{Pr}[A] = 1/2$, $\mathbf{Pr}[B] = 2/3$ und $\mathbf{Pr}[A \cap B] = 1/3 = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$ gilt.
- Seien Ω und $\mathbf{Pr}$ wie in obigem Beispiel gewählt. Sei A das Ereignis, dass die Summe der gewürfelten Zahlen gleich 6 ist, und sei B das Ereignis, dass der erste Würfel die Zahl 3 zeigt. Intuitiv sind diese Ereignisse nicht unabhängig (als Summe kann nur 6 herauskommen, wenn im ersten Wurf nicht bereits eine 6 geworfen wurde). Tatsächlich gilt $\mathbf{Pr}[A] = \frac{5}{36}$, $\mathbf{Pr}[B] = \frac{1}{6}$ und

  $$\mathbf{Pr}[A \cap B] = \frac{1}{36} > \mathbf{Pr}[A] \cdot \mathbf{Pr}[B] = \frac{5}{216},$$

  was auch formal die Abhängigkeit der Ereignisse zeigt.
- Für eine endliche Folge von Wahrscheinlichkeitsräumen $(\Omega_1, \mathbf{Pr}_1), \ldots, (\Omega_n, \mathbf{Pr}_n)$ können wir durch $\Omega = \Omega_1 \times \ldots \times \Omega_n$ und $\mathbf{Pr}[x_1, \ldots, x_n] = \mathbf{Pr}_1[x_1] \cdot \ldots \cdot \mathbf{Pr}_n[x_n]$ den *Produktraum* definieren. Ereignisse, die sich auf unterschiedliche Komponenten des Produktraumes beziehen, sind stets unabhängig. Sei beispielsweise $n = 2$ und seien $A \subseteq \Omega_1$ und $B \subseteq \Omega_2$ beliebig, dann sind die Ereignisse $A \times \Omega_2$ und $\Omega_1 \times B$ unabhängig.

Betrachten wir noch einmal den obigen Algorithmus RandMaxCut. Wir können den diesem Algorithmus zugrundeliegenden Wahrscheinlichkeitsraum auch als Produktraum darstellen. Dazu führen wir für jeden Knoten $v_i \in V$ einen Wahrscheinlichkeitsraum $(\{0,1\}, \mathbf{Pr})$ mit $\mathbf{Pr}[0] = \mathbf{Pr}[1] = 1/2$ ein. Der Wahrscheinlichkeitsraum des Algorithmus ergibt sich dann durch das n-fache Produkt dieses einfachen Wahrscheinlichkeitsraums. In dieser Darstellung wird auch formal klar, dass Ereignisse, die sich auf disjunkte Mengen von Knoten beziehen, unabhängig sind. Betrachten wir also zum Beispiel zwei Kanten e und e', die keine gemeinsamen Endknoten haben, so sind die Ereignisse, dass e bzw. e' über den Schnitt läuft, unabhängig. Die Wahrscheinlichkeit, dass beide über den Schnitt laufen, beträgt also $(1/2)^2 = 1/4$.

Anhand des Algorithmus RandMaxCut führen wir nun auch das Konzept von *bedingten Wahrscheinlichkeiten* ein.

Definition 4.3. *Sei* $(\Omega, \mathbf{Pr})$ *ein diskreter Wahrscheinlichkeitsraum. Ferner seien* $A \subseteq \Omega$ *und* $B \subseteq \Omega$ *zwei Ereignisse mit* $\mathbf{Pr}[B] > 0$*. Wir bezeichnen mit*

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}$$

die bedingte Wahrscheinlichkeit von A gegeben B*.*

Auch dieses Konzept illustrieren wir an einigen Beispielen.

- Wir betrachten zwei unabhängige Würfe eines Würfels. Sei A das Ereignis, dass der erste Wurf 6 ergibt, und sei B das Ereignis, dass die Summe der gewürfelten Zahlen 10 ist. Es gilt $\mathbf{Pr}[A] = 1/6$ und intuitiv sollte Ereignis A wahrscheinlicher werden, wenn wir schon wissen, dass B eintritt. Da das Eintreten von B ausschließt, dass der erste Wurf eine Zahl kleiner oder gleich 3 liefert, sind nur noch die Zahlen 4, 5 und 6 übrig. Somit sollte intuitiv die Wahrscheinlichkeit von A ein Drittel sein, wenn wir schon wissen, dass B eintritt. Tatsächlich gilt

  $$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]} = \frac{1/36}{3/36} = \frac{1}{3}.$$

- Seien A und B zwei unabhängige Ereignisse mit $\mathbf{Pr}[B] > 0$. Dann gilt

  $$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]} = \frac{\mathbf{Pr}[A] \cdot \mathbf{Pr}[B]}{\mathbf{Pr}[B]} = \mathbf{Pr}[A].$$

  Dies entspricht unserer Intuition, dass bei zwei unabhängigen Ereignissen das Eintreten des einen Ereignisses keine Konsequenzen für das Eintreten des anderen besitzt.

- Nun betrachten wir wieder den Algorithmus RandMaxCut. Seien drei Kanten e_1, e_2 und e_3 gegeben, die ein Dreieck bilden, also $e_1 = (x, y)$, $e_2 = (y, z)$ und $e_3 = (z, x)$. Wir bezeichnen mit A_i das Ereignis, dass die Kante e_i über den Schnitt läuft. Außerdem sei $B = A_1 \cap A_2$ das Ereignis, dass die Kanten e_1 und e_2 beide über den Schnitt laufen. Man überlegt sich leicht, dass nicht alle drei Kanten gleichzeitig über den Schnitt laufen können. Es gilt also

  $$\mathbf{Pr}[A_3 \mid B] = \frac{\mathbf{Pr}[A_3 \cap B]}{\mathbf{Pr}[B]} = \frac{\mathbf{Pr}[A_1 \cap A_2 \cap A_3]}{\mathbf{Pr}[B]} = \frac{0}{\mathbf{Pr}[B]} = 0.$$

  Als nächstes bestimmen wir die bedingte Wahrscheinlichkeit von A_3 gegeben $\overline{B}$. Dazu betrachten wir in der folgenden Tabelle alle möglichen Elementarereignisse. Dort tragen wir ebenfalls für die Ereignisse B, $\overline{B}$, A_3 und $A_3 \cap B$ ein, für welche Elementarereignisse sie eintreten (mit 1 gekennzeichnet). Außerdem tragen wir in der letzten Spalte ein, wie viele Kanten bei dem entsprechenden Elementarereignis über den Schnitt laufen, da wir diese Information später benötigen werden.

U	B	$\overline{B}$	A_3	$A_3 \cap \overline{B}$	# Kanten im Schnitt
$\emptyset$	0	1	0	0	0
$\{x\}$	0	1	1	1	2
$\{y\}$	1	0	0	0	2
$\{x, y\}$	0	1	1	1	2
$\{z\}$	0	1	1	1	2
$\{x, z\}$	1	0	0	0	2
$\{y, z\}$	0	1	1	1	2
$\{x, y, z\}$	0	1	0	0	0

Anhand der Tabelle können wir direkt ablesen, dass gilt

$$\mathbf{Pr}\left[A_3 \mid \overline{B}\right] = \frac{\mathbf{Pr}\left[A_3 \cap \overline{B}\right]}{\mathbf{Pr}\left[\overline{B}\right]} = \frac{4}{6} = \frac{2}{3}.$$

4.2 Zufallsvariablen und Erwartungswerte

Oft sind wir bei einem Zufallsexperiment gar nicht an seinem exakten Ausgang interessiert, sondern nur an einer Kenngröße, die die für uns wesentliche Information enthält. Werfen wir beispielsweise zwei Würfel, so sind wir oft nur an der Summe der Augenzahlen interessiert. Ebenso sind wir bei dem Algorithmus RandMaxCut in der Regel nicht an der genauen Realisierung von U interessiert, sondern nur an der Anzahl an Kanten, die über den Schnitt laufen. Um dies zu modellieren, führen wir das Konzept von *Zufallsvariablen* ein.

Definition 4.4. *Sei $(\Omega, \mathbf{Pr})$ ein diskreter Wahrscheinlichkeitsraum. Wir nennen eine Abbildung $X : \Omega \to \mathbb{R}$, die die Ergebnismenge auf die reellen Zahlen abbildet, eine* diskrete reelle Zufallsvariable.

Bei einer Zufallsvariablen X interessiert uns die Wahrscheinlichkeit, mit der sie bestimmte Werte annimmt. Für eine reelle Zahl $a \in \mathbb{R}$ interpretieren wir $X = a$ als das Ereignis $\{\omega \in \Omega \mid X(\omega) = a\}$ im Wahrscheinlichkeitsraum $(\Omega, \mathbf{Pr})$. Mit dieser Interpretation ergibt sich

$$\mathbf{Pr}[X = a] = \mathbf{Pr}[\{\omega \in \Omega \mid X(\omega) = a\}].$$

Für eine Teilmenge $A \subseteq \mathbb{R}$ der reellen Zahlen ergibt sich analog

$$\mathbf{Pr}[X \in A] = \mathbf{Pr}[\{\omega \in \Omega \mid X(\omega) \in A\}].$$

Wir betrachten zunächst zwei Beispiele für Zufallsvariablen.

- Sei $(\Omega, \mathbf{Pr})$ mit $\Omega = \{1, \ldots, 6\}^2$ und $\mathbf{Pr}[\omega] = 1/36$ für $\omega \in \Omega$ der Wahrscheinlichkeitsraum, der zwei unabhängige Würfelwürfe beschreibt, so können wir als

Zufallsvariable $X : \Omega \to \{1, \ldots, 12\}$ die Summe der gewürfelten Zahlen definieren. Formal setzen wir dafür $X(\omega) = \omega_1 + \omega_2$ für $\omega = (\omega_1, \omega_2) \in \Omega$. Für diese Zufallsvariable gilt beispielsweise

$$\mathbf{Pr}[X = 1] = 0,\ \mathbf{Pr}[X = 2] = \frac{1}{36} \text{ und } \mathbf{Pr}[X = 7] = \frac{1}{6}.$$

- Sei $(\Omega, \mathbf{Pr})$ mit $\Omega = \{Z, K\}^{10}$ und $\mathbf{Pr}[\omega] = 1/1024$ für $\omega \in \Omega$ der Wahrscheinlichkeitsraum, der zehn unabhängige Münzwürfe beschreibt. Wir definieren als Zufallsvariable $X : \Omega \to \{0, 1, 2, \ldots, 10\}$, wie oft „Zahl“ geworfen wurde. Als Übung sollte der Leser sich überlegen, dass für diese Zufallsvariable

  $$\mathbf{Pr}[X = 0] = \frac{1}{1024},\ \mathbf{Pr}[X = 1] = \frac{10}{1024} \text{ und } \mathbf{Pr}[X = 2] = \frac{45}{1024}$$

  gilt.

- Wir betrachten noch einmal den Algorithmus RandMaxCut auf dem obigen Beispiel, in dem die drei Kanten e_1, e_2 und e_3 ein Dreieck bilden. Sei X die Zufallsvariable, die die Anzahl an Kanten beschreibt, die über den Schnitt laufen. Anhand der Tabelle, in der wir alle möglichen Realisierungen von U aufgelistet haben, können wir direkt ablesen, dass $\mathbf{Pr}[X = 0] = 2/8 = 1/4$ und $\mathbf{Pr}[X = 2] = 6/8 = 3/4$ gilt.

Genauso wie bei Ereignissen können wir auch bei mehreren Zufallsvariablen wieder unterscheiden, ob sie unabhängig oder abhängig sind.

Definition 4.5. *Sei $(\Omega, \mathbf{Pr})$ ein diskreter Wahrscheinlichkeitsraum und seien $X : \Omega \to \mathbb{R}$ und $Y : \Omega \to \mathbb{R}$ diskrete reelle Zufallsvariablen. Wir nennen X und Y* unabhängig*, wenn für alle $x \in \mathbb{R}$ und $y \in \mathbb{R}$*

$$\mathbf{Pr}[(X = x) \cap (Y = y)] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y]$$

gilt. Wir nennen eine Familie $X_1, \ldots, X_n$ von diskreten reellen Zufallsvariablen unabhängig*, wenn für jede Teilmenge $I \subseteq \{1, \ldots, n\}$ und für alle $x_i \in \mathbb{R}$, $i \in I$,*

$$\mathbf{Pr}\left[\bigcap_{i \in I} (X_i = x_i)\right] = \prod_{i \in I} \mathbf{Pr}[X_i = x_i]$$

gilt.

Diese Definition ist an Definition 4.2 angelehnt. Tatsächlich folgt unmittelbar, dass zwei Zufallsvariablen X und Y genau dann unabhängig sind, wenn die Ereignisse $\{\omega \in \Omega \mid X(\omega) = x\}$ und $\{\omega \in \Omega \mid Y(\omega) = y\}$ für alle $x, y \in \mathbb{R}$ unabhängig sind.

Oft möchte man den „durchschnittlichen Wert“ einer Zufallsvariablen angeben. Dazu summiert man alle Werte auf, die die Zufallsvariable annehmen kann, wobei die einzelnen Werte jeweils mit der Wahrscheinlichkeit gewichtet werden, mit der sie von der Zufallsvariablen angenommen werden.

Definition 4.6. *Sei* $(\Omega, \mathbf{Pr})$ *ein diskreter Wahrscheinlichkeitsraum und sei* $X : \Omega \to \mathbb{R}$ *eine diskrete reelle Zufallsvariable. Falls die Reihe* $\sum_{\omega\in\Omega} |X(\omega)| \cdot \mathbf{Pr}[\{\omega\}]$ *konvergiert, so definieren wir*

$$\mathbf{E}[X] = \sum_{\omega\in\Omega} X(\omega) \cdot \mathbf{Pr}[\{\omega\}]$$

als den Erwartungswert *von* X*.*

Sei X eine Zufallsvariable, für die der Erwartungswert gemäß obiger Definition existiert. Bezeichnen wir mit $R = X(\Omega)$ die abzählbare Teilmenge der reellen Zahlen, die von X angenommen werden können, dann können wir den Erwartungswert von X auch wie folgt berechnen:

$$\mathbf{E}[X] = \sum_{x\in R} x \cdot \mathbf{Pr}[X = x]. \tag{4.1}$$

- Für die Zufallsvariable X, die die Summe zweier unabhängiger Würfelwürfe beschreibt, gilt

$$\mathbf{E}[X] = 2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + 4 \cdot \frac{3}{36} + \ldots + 11 \cdot \frac{2}{36} + 12 \cdot \frac{1}{36} = 7.$$

- Für die Zufallsvariable X, die beschreibt, wie oft in zehn Münzwürfen „Zahl“ geworfen wird, ist es schon schwieriger, den Erwartungswert anhand der Definition nachzurechnen. Nur mit viel Mühe rechnet man nach, dass

$$\mathbf{E}[X] = 0 \cdot \frac{1}{1024} + 1 \cdot \frac{10}{1024} + 2 \cdot \frac{45}{1024} + \ldots + 10 \cdot \frac{1}{1024} = 5$$

  gilt.

- Wieder betrachten wir den Algorithmus RandMaxCut auf dem obigen Beispiel, in dem die drei Kanten e_1, e_2 und e_3 ein Dreieck bilden. Sei wieder X die Zufallsvariable, die die Anzahl an Kanten beschreibt, die über den Schnitt laufen. Da X nur die Werte 0 oder 2 annehmen kann, gilt

$$\mathbf{E}[X] = \mathbf{Pr}[X = 0] \cdot 0 + \mathbf{Pr}[X = 2] \cdot 2 = \frac{3}{4} \cdot 2 = \frac{3}{2}.$$

In dem obigen Beispiel für den Algorithmus RandMaxCut haben wir die erwartete Anzahl an Kanten, die über den Schnitt laufen, ausgerechnet. Für Graphen mit vielen Knoten und Kanten wäre eine ähnliche Rechnung sehr aufwändig, da wir in der Tabelle auf Seite 36 alle möglichen Realisierungen für U einzeln betrachtet haben. Um das Ausrechnen von Erwartungswerten zu vereinfachen, lernen wir jetzt einige wichtige Rechenregeln kennen.

Theorem 4.7. *Sei* $(\Omega, \mathbf{Pr})$ *ein diskreter Wahrscheinlichkeitsraum und seien* $X : \Omega \to \mathbb{R}$ *und* $Y : \Omega \to \mathbb{R}$ *diskrete reelle Zufallsvariablen, deren Erwartungswerte existieren. Dann gilt:*

a) Für $c, d \in \mathbb{R}$ *existiert der Erwartungswert der Zufallsvariablen* $(cX + d) : \Omega \to \mathbb{R}$ *mit* $(cX + d)(\omega) = cX(\omega) + d$ *für* $\omega \in \Omega$*. Es gilt* $\mathbf{E}[cX + d] = c\mathbf{E}[X] + d$*.*

b) Der Erwartungswert der Zufallsvariablen $(X+Y):\Omega\to\mathbb{R}$ *mit* $(X+Y)(\omega)=X(\omega)+Y(\omega)$ *für* $\omega\in\Omega$ *existiert und es gilt* $\mathbf{E}[X+Y]=\mathbf{E}[X]+\mathbf{E}[Y]$.

c) Sind X *und* Y *unabhängig, so existiert der Erwartungswert der Zufallsvariablen* $(X\cdot Y):\Omega\to\mathbb{R}$ *mit* $(X\cdot Y)(\omega)=X(\omega)\cdot Y(\omega)$ *für* $\omega\in\Omega$ *und es gilt* $\mathbf{E}[X\cdot Y]=\mathbf{E}[X]\cdot\mathbf{E}[Y]$.

Beweis. a) Aus Definition 4.6 folgt

$$\begin{aligned}\mathbf{E}[cX+d]&=\sum_{\omega\in\Omega}(cX+d)(\omega)\cdot\mathbf{Pr}[\{\omega\}]\\&=\sum_{\omega\in\Omega}(cX(\omega)+d)\cdot\mathbf{Pr}[\{\omega\}]\\&=c\sum_{\omega\in\Omega}X(\omega)\cdot\mathbf{Pr}[\{\omega\}]+d\sum_{\omega\in\Omega}\mathbf{Pr}[\{\omega\}]\\&=c\mathbf{E}[X]+d.\end{aligned}$$

Die Existenz des Erwartungswertes von $cX+d$ rechnet man ebenfalls leicht nach.

b) Auch diese Aussage folgt direkt aus Definition 4.6:

$$\begin{aligned}\mathbf{E}[X+Y]&=\sum_{\omega\in\Omega}(X+Y)(\omega)\cdot\mathbf{Pr}[\{\omega\}]\\&=\sum_{\omega\in\Omega}(X(\omega)+Y(\omega))\cdot\mathbf{Pr}[\{\omega\}]\\&=\sum_{\omega\in\Omega}X(\omega)\cdot\mathbf{Pr}[\{\omega\}]+\sum_{\omega\in\Omega}Y(\omega)\cdot\mathbf{Pr}[\{\omega\}]\\&=\mathbf{E}[X]+\mathbf{E}[Y].\end{aligned}$$

c) Seien $R=X(\Omega)$ und $S=Y(\Omega)$ die abzählbaren Mengen von Werten, die X bzw. Y annehmen können. Es gilt dann gemäß Definition 4.6

$$\begin{aligned}\mathbf{E}[X\cdot Y]&=\sum_{\omega\in\Omega}(X\cdot Y)(\omega)\cdot\mathbf{Pr}[\{\omega\}]\\&=\sum_{\omega\in\Omega}(X(\omega)\cdot Y(\omega))\cdot\mathbf{Pr}[\{\omega\}]\\&=\sum_{x\in R}\sum_{y\in S}xy\cdot\mathbf{Pr}[(X=x)\cap(Y=y)]\\&=\sum_{x\in R}\sum_{y\in S}xy\cdot\mathbf{Pr}[X=x]\cdot\mathbf{Pr}[Y=y]\\&=\sum_{x\in R}x\mathbf{Pr}[X=x]\sum_{y\in S}y\cdot\mathbf{Pr}[Y=y]\\&=\sum_{x\in R}x\mathbf{Pr}[X=x]\cdot\mathbf{E}[Y]\\&=\mathbf{E}[Y]\cdot\sum_{x\in R}x\mathbf{Pr}[X=x]\\&=\mathbf{E}[X]\cdot\mathbf{E}[Y],\end{aligned}$$

wobei wir in der vierten Zeile die Unabhängigkeit der Zufallsvariablen ausgenutzt haben. Die Existenz des Erwartungswertes rechnet man wieder analog nach, indem man in allen obigen Summen x durch $|x|$ und y durch $|y|$ ersetzt. □

Aussage b) des vorangegangenen Theorems werden wir auch als *Linearität des Erwartungswertes* bezeichnen. Es handelt sich dabei um eine äußerst nützliche Rechenregel, die sogar für abhängige Zufallsvariablen X und Y gültig ist.

Nutzen wir die Linearität des Erwartungswertes, so ist es viel einfacher, die Erwartungswerte in den beiden obigen Beispielen zu berechnen. Betrachten wir dazu noch einmal den Wahrscheinlichkeitsraum $(\Omega, \mathbf{Pr})$ mit $\Omega = \{Z, K\}^{10}$ und $\mathbf{Pr}[\omega] = 1/1024$ für $\omega \in \Omega$ und die Zufallsvariable $X : \Omega \to \{0, 1, 2, \ldots, 10\}$, die angibt, wie oft „Zahl“ geworfen wird. Diese können wir darstellen als die Summe der Zufallsvariablen $X_1, \ldots, X_{10}$, wobei $X_i : \Omega \to \{0, 1\}$ durch

$$X_i = \begin{cases} 1 & \text{falls } i\text{-ter Wurf „Zahl“ ergibt,} \\ 0 & \text{sonst,} \end{cases}$$

definiert ist. Eigentlich hätten wir in dieser Definition formal korrekt $X_i(\omega)$ statt einfach nur X_i schreiben müssen. Um die Notation einfach zu halten, haben wir dies nicht getan und werden auch im Folgenden in ähnlicher Weise Zufallsvariablen beschreiben. Für jede Zufallsvariable X_i gilt

$$\mathbf{E}[X_i] = 1 \cdot \mathbf{Pr}[i\text{-ter Wurf ergibt „Zahl“}] = \frac{1}{2}.$$

Somit ergibt sich

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{10} X_i\right] = \sum_{i=1}^{10} \mathbf{E}[X_i] = \sum_{i=1}^{10} \frac{1}{2} = 5.$$

Oft sind wir nicht nur daran interessiert, den Erwartungswert einer Zufallsvariablen abzuschätzen, sondern wir möchten zusätzlich zeigen, dass es unwahrscheinlich ist, dass die Zufallsvariable stark von ihrem Erwartungswert abweicht. Wir möchten also zeigen, dass die Zufallsvariable um ihren Erwartungswert herum konzentriert ist. Die einfachste Möglichkeit, ein solches Konzentrationsergebnis für nicht-negative Zufallsvariablen zu erhalten, ist die folgende *Markow-Ungleichung*.

Theorem 4.8. *Es sei $X : \Omega \to \mathbb{R}_{\geq 0}$ eine Zufallsvariable, die nur nicht-negative Werte annimmt und für die der Erwartungswert $\mathbf{E}[X]$ existiert. Dann gilt für alle $a > 0$*

$$\mathbf{Pr}[X \geq a] \leq \frac{\mathbf{E}[X]}{a}.$$

Wir können die Markow-Ungleichung umformulieren, um eine noch einprägsamere Form zu erhalten. Sei $X : \Omega \to \mathbb{R}_{\geq 0}$ eine Zufallsvariable, die nur nicht-negative Werte annimmt. Dann gilt für alle $a \geq 1$

$$\mathbf{Pr}[X \geq a \cdot \mathbf{E}[X]] \leq \frac{1}{a}.$$

Die Wahrscheinlichkeit, dass X a-mal so groß ist wie sein Erwartungswert, beträgt also höchstens $1/a$.

Beweis von Theorem 4.8. Für den Beweis definieren wir uns einfach eine Indikatorvariable I mit

$$I = \begin{cases} 1 & \text{falls } X \geq a, \\ 0 & \text{sonst.} \end{cases}$$

Für diese Indikatorvariable gilt $\mathbf{E}[I] = \mathbf{Pr}[I = 1] = \mathbf{Pr}[X \geq a]$. Außerdem gilt wegen $X \geq 0$ stets $I \leq X/a$. Damit folgt insgesamt, wie gewünscht,

$$\mathbf{Pr}[X \geq a] = \mathbf{E}[I] \leq \mathbf{E}\left[\frac{X}{a}\right] = \frac{\mathbf{E}[X]}{a}. \qquad \square$$

Mit der Markow-Ungleichung können wir beispielsweise die Wahrscheinlichkeit abschätzen, dass wir in einer Folge von n unabhängigen fairen Münzwürfen mindestens $(3n/4)$-mal „Zahl“ sehen. Sei X die Zufallsvariable, die angibt, wie oft „Zahl“ geworfen wird. Dann gilt gemäß der Markow-Ungleichung

$$\mathbf{Pr}\left[X \geq \frac{3n}{4}\right] \leq \frac{\mathbf{E}[X]}{3n/4} = \frac{n/2}{3n/4} = \frac{2}{3}.$$

Tatsächlich fällt die obige Wahrscheinlichkeit sogar exponentiell mit n. Dies werden wir in dieser Vorlesung jedoch nicht zeigen.

4.2.1 Analyse von RandMaxCut

Mithilfe des Erwartungswertes können wir nun auch den Approximationsfaktor eines randomisierten Algorithmus definieren.

Definition 4.9. *Ein randomisierter Approximationsalgorithmus A für ein Minimierungs- oder Maximierungsproblem Π erreicht einen* Approximationsfaktor *von $r \geq 1$ bzw. $r \leq 1$, wenn*

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

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

In vielen Fällen hängt der Approximationsfaktor von der Eingabelänge ab. Dann ist $r : \mathbb{N} \to [0, \infty)$ eine Funktion und es muss

$$\mathbf{E}[w_A(I)] \leq r(|I|) \cdot \mathrm{OPT}(I) \quad \textit{bzw.} \quad \mathbf{E}[w_A(I)] \geq r(|I|) \cdot \mathrm{OPT}(I)$$

für alle Instanzen I gelten, wobei $|I|$ die Länge der Eingabe I bezeichne.

Theorem 4.10. *Der Algorithmus* RandMaxCut *ist ein randomisierter $\frac{1}{2}$-Approximationsalgorithmus für das Maximum-Cut Problem.*

Beweis. Sei $G = (V, E)$ ein ungerichteter Graph mit Kantengewichten w. Sei $U \subseteq V$ die zufällige Teilmenge von V, die der Algorithmus RandMaxCut auswählt. Es sei X

die Zufallsvariable, die das Gesamtgewicht der Kanten angibt, die über den Schnitt laufen. Außerdem definieren wir für jede Kante $e \in E$ eine Zufallsvariable X_e durch

$$X_e = \begin{cases} w(e) & \text{falls } e \text{ über den Schnitt läuft,} \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt per Definition $X = \sum_{e \in E} X_e$ und dementsprechend

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbf{E}[X_e],$$

wobei die letzte Gleichung aus der Linearität des Erwartungswertes folgt. Da X_e nur die Werte 0 und $w(e)$ annehmen kann, gilt

$$\begin{aligned} \mathbf{E}[X_e] &= 0 \cdot \mathbf{Pr}[X_e = 0] + w(e) \cdot \mathbf{Pr}[X_e = w(e)] \\ &= w(e) \cdot \mathbf{Pr}[e \text{ läuft über den Schnitt}] \\ &= \frac{w(e)}{2}. \end{aligned}$$

Für die letzte Gleichung haben wir ausgenutzt, dass jede Kante e mit einer Wahrscheinlichkeit von $1/2$ über den Schnitt läuft (dies haben wir oben schon gezeigt). Insgesamt ergibt sich

$$\mathbf{E}[X] = \sum_{e \in E} \mathbf{E}[X_e] = \sum_{e \in E} \frac{w(e)}{2} = \frac{W}{2}$$

für $W = \sum_{e \in E} w(e)$. Außerdem gilt $\mathrm{OPT} \leq W$, da bestenfalls alle Kanten über den Schnitt laufen können. Insgesamt folgt damit, wie gewünscht,

$$\mathbf{E}[X] = \frac{W}{2} \geq \frac{\mathrm{OPT}}{2}. \qquad \square$$

4.2.2 Binomialverteilung und geometrische Verteilung

Wir führen in diesem Abschnitt als weitere Hilfsmittel für die Analyse von randomisierten Algorithmen *binomialverteilte Zufallsvariablen* und *geometrische Zufallsvariablen* ein. Betrachten wir zunächst ein Zufallsexperiment, das mit Wahrscheinlichkeit $p \in [0, 1]$ erfolgreich ist und mit Wahrscheinlichkeit $1 - p$ erfolglos. Dann nennen wir Y mit

$$Y = \begin{cases} 1 & \text{falls Zufallsexperiment erfolgreich,} \\ 0 & \text{falls Zufallsexperiment erfolglos,} \end{cases}$$

die *Indikatorvariable* für das Ereignis, dass das Zufallsexperiment erfolgreich ist. Eine solche Zufallsvariable, die nur die Werte 0 und 1 annehmen kann und die Wert 1 mit Wahrscheinlichkeit p annimmt, nennen wir auch *Bernoulli-Zufallsvariable mit Parameter p*. Es gilt

$$\mathbf{E}[Y] = 0 \cdot \mathbf{Pr}[Y = 0] + 1 \cdot \mathbf{Pr}[Y = 1] = \mathbf{Pr}[Y = 1] = p.$$

Seien $Y_1, \ldots, Y_n$ unabhängige Bernoulli-Zufallsvariablen mit demselben Parameter $p \in [0, 1]$. Die Zufallsvariable $X = Y_1 + \ldots + Y_n$ nennen wir *binomialverteilte Zufallsvariable mit Parametern n und p*.

Theorem 4.11. *Sei X eine binomialverteilte Zufallsvariable mit Parametern n und p. Es gilt $\mathbf{E}[X] = pn$ und für $j \in \{0, \ldots, n\}$ gilt*

$$\mathbf{Pr}[X = j] = \binom{n}{j} \cdot p^j \cdot (1-p)^{n-j}.$$

Beweis. Als erstes berechnen wir die Wahrscheinlichkeit für das Ereignis $X = j$. Dazu definieren wir für jede Möglichkeit, wie die j Erfolge sich auf die n Versuche aufteilen können, ein Ereignis. Seien dies die Ereignisse $A_1, \ldots, A_m$. Es gilt dann $m = \binom{n}{j}$, da dies der Anzahl von Möglichkeiten entspricht, eine Teilmenge mit j Elementen aus einer Menge mit n Elementen auszuwählen. Ein Ereignis A_i legt fest, welche j Versuche erfolgreich und welche $n - j$ erfolglos sind. Die Wahrscheinlichkeit, dass A_i eintritt, beträgt somit für jedes i genau $p^j(1-p)^{n-j}$. Die Ereignisse $A_1, \ldots, A_m$ sind paarweise disjunkt. Damit gilt

$$\mathbf{Pr}[X = j] = \mathbf{Pr}\left[\bigcup_{i=1}^{m} A_i\right] = \sum_{i=1}^{m} \mathbf{Pr}[A_i] = m \cdot p^j \cdot (1-p)^{n-j} = \binom{n}{j} \cdot p^j \cdot (1-p)^{n-j}.$$

Mithilfe dieser Wahrscheinlichkeit kann man mit ein wenig Rechnen den Erwartungswert bestimmen:

$$\mathbf{E}[X] = \sum_{j=0}^{n} \left(j \cdot \binom{n}{j} \cdot p^j \cdot (1-p)^{n-j} \right) = \ldots = np.$$

Viel einfacher geht es aber, wenn man die Linearität des Erwartungswertes ausnutzt. Damit folgt nämlich direkt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} Y_i\right] = \sum_{i=1}^{n} \mathbf{E}[Y_i] = np,$$

wobei wir $X = Y_1 + \ldots + Y_n$ für Bernoulli-Zufallsvariablen $Y_1, \ldots, Y_n$ mit Parameter p ausgenutzt haben. □

Sei nun $Y_1, Y_2, Y_3, \ldots$ eine Folge von unabhängigen Bernoulli-Zufallsvariablen mit Parameter p. Wir betrachten die Frage, wie lange wir auf den ersten Erfolg warten müssen. Dies wird durch eine *geometrische Zufallsvariable mit Parameter p* beschrieben. Eine solche Zufallsvariable nimmt den Wert $n \in \mathbb{N}$ an, wenn $Y_1 = \ldots = Y_{n-1} = 0$ und $Y_n = 1$ gilt. Die Wahrscheinlichkeit hierfür beträgt $(1-p)^{n-1} \cdot p$. Wir definieren für eine geometrische Zufallsvariable X mit Parameter p dementsprechend $\mathbf{Pr}[X = n] = (1-p)^{n-1} \cdot p$ für jedes $n \in \mathbb{N}$. Um den Erwartungswert einer geometrischen Zufallsvariablen zu bestimmen, leiten wir zunächst eine nützliche alternative Darstellung des Erwartungswertes her, die für alle Zufallsvariablen gilt, die nur Werte aus $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$ annehmen können.

Lemma 4.12. *Sei $(\Omega, \mathbf{Pr})$ ein diskreter Wahrscheinlichkeitsraum und sei $X : \Omega \to \mathbb{N}_0$ eine diskrete Zufallsvariable, deren Erwartungswert existiert. Dann gilt*

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i].$$

Beweis. Es gilt

$$\begin{aligned}\sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] &= \sum_{i=1}^{\infty} \sum_{j=i}^{\infty} \mathbf{Pr}[X = j] \\ &= \sum_{j=1}^{\infty} \sum_{i=1}^{j} \mathbf{Pr}[X = j] \\ &= \sum_{j=1}^{\infty} j \cdot \mathbf{Pr}[X = j] = \mathbf{E}[X].\end{aligned}$$

In obiger Rechnung haben wir ausgenutzt, dass wir bei absolut konvergenten Reihen die Reihenfolge der Summanden beliebig vertauschen können. □

Theorem 4.13. *Für eine geometrische Zufallsvariable X mit Parameter p gilt* $\mathbf{E}[X] = 1/p$.

Beweis. Für die geometrische Zufallsvariable X und $j \in \mathbb{N}$ gilt

$$\mathbf{Pr}[X \geq j] = \sum_{i=j}^{\infty} \mathbf{Pr}[X = i] = \sum_{i=j}^{\infty} (1-p)^{i-1} \cdot p = (1-p)^{j-1}.$$

Die letzte Gleichung in obiger Formel rechnet man entweder nach oder man überlegt sich einfach, was passieren muss, damit $X \geq j$ gilt. Dies ist genau dann der Fall, wenn die ersten $j-1$ Versuche erfolglos sind. Die Wahrscheinlichkeit dafür beträgt genau $(1-p)^{j-1}$.

Nun folgt das Theorem leicht aus Lemma 4.12:

$$\mathbf{E}[X] = \sum_{j=1}^{\infty} \mathbf{Pr}[X \geq j] = \sum_{j=1}^{\infty} (1-p)^{j-1} = \frac{1}{1-(1-p)} = \frac{1}{p}.$$ □

Eine wichtige Eigenschaft der geometrischen Verteilung, die man oft implizit oder explizit ausnutzt, ist ihre *Gedächtnislosigkeit*. Das bedeutet, dass die Wahrscheinlichkeit eines Erfolges unabhängig davon ist, wie viele Misserfolge es bis jetzt gegeben hat. Das ist wie beim Roulette: Die Information, dass in den letzten zehn Spielen immer rot geworfen wurde, ändert nichts an der Wahrscheinlichkeit, dass im nächsten Spiel rot geworfen wird. Formal drückt das folgende Lemma die Gedächtnislosigkeit von geometrischen Zufallsvariablen aus.

Lemma 4.14. *Es sei X eine geometrische Zufallsvariable mit Parameter p. Dann gilt für alle $n \in \mathbb{N}$ und $k \in \mathbb{N}_0$*

$$\mathbf{Pr}[X = n + k \mid X > k] = \mathbf{Pr}[X = n].$$

Beweis. Es gilt

$$\begin{aligned}\mathbf{Pr}[X = n+k \mid X > k] &= \frac{\mathbf{Pr}[(X = n+k) \cap (X > k)]}{\mathbf{Pr}[X > k]} = \frac{\mathbf{Pr}[X = n+k]}{\mathbf{Pr}[X > k]} \\ &= \frac{(1-p)^{n+k-1} \cdot p}{(1-p)^k} = (1-p)^{n-1} \cdot p = \mathbf{Pr}[X = n].\end{aligned}$$ □

Wir betrachten nun noch ein einfaches Zufallsexperiment, das in der Analyse von randomisierten Algorithmen oft auftritt. Das *Coupon Collector's Problem* lässt sich anhand von Sammelbildern beschreiben, wie sie zum Beispiel zu jeder Fußball-Weltmeisterschaft verkauft werden. Wir gehen davon aus, dass es insgesamt n verschiedene solcher Bilder gibt. Diese kann man nicht gezielt erwerben, sondern man kann nur verschlossene Tüten kaufen, in denen sich jeweils unabhängig ein uniform zufällig ausgewähltes Sammelbild befindet. Wir bezeichnen mit X die Zufallsvariable, die angibt, wie viele Tüten wir kaufen müssen, bis wir jedes Bild mindestens einmal in unserer Sammlung haben.

Theorem 4.15. *Für das Coupon Collector's Problem gilt* $\mathbf{E}[X] = n \ln n + \Theta(n)$.

Beweis. Zum Beweis des Theorems zerlegen wir den Zufallsprozess in n Phasen. Die erste Phase beginnt mit dem ersten Sammelbild, das wir kaufen. Phase i für $i \in \{2, \ldots, n\}$ beginnt direkt, nachdem wir das $(i-1)$-te verschiedene Sammelbild gekauft haben, und endet mit dem i-ten verschiedenen Sammelbild. Bezeichnen wir mit X_i die Anzahl an Sammelbildern, die wir in Phase i kaufen, dann gilt $X = X_1 + \ldots + X_n$. Außerdem ist jedes X_i eine geometrisch verteilte Zufallsvariable mit Parameter

$$p_i = \frac{n - i + 1}{n},$$

denn die Wahrscheinlichkeit, eines der $n - i + 1$ Sammelbilder zu kaufen, die nicht zu den $(i-1)$ bereits erworbenen gehören, beträgt in Phase i in jedem Versuch $(n-i+1)/n$. Mit der Linearität des Erwartungswertes und Theorem 4.13 folgt nun direkt

$$\begin{aligned}\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} \frac{1}{p_i} = \sum_{i=1}^{n} \frac{n}{n-i+1} \\ = n \sum_{i=1}^{n} \frac{1}{i} = n \ln n + \Theta(n).\end{aligned}$$

Im letzten Schritt haben wir ausgenutzt, dass für die n-te harmonische Zahl $\sum_{i=1}^{n} \frac{1}{i} = \ln n + \Theta(1)$ gilt. □

4.2.3 Vertex Cover

Wir betrachten nun einen simplen Algorithmus RandVertexCover für das Vertex Cover Problem. Für eine Eingabe $G = (V, E)$ startet dieser Algorithmus mit einer leeren Menge U und geht alle Kanten aus E in einer beliebigen Reihenfolge durch. Für jede Kante $e \in E$ überprüft der Algorithmus, ob bereits mindestens ein Endknoten von e in der Menge U enthalten ist. Ist dies der Fall, so macht der Algorithmus mit der nächsten Kante aus E weiter. Ansonsten wählt er uniform zufällig einen der beiden Endknoten von e aus und fügt diesen der Menge U hinzu. Anschließend macht er mit der nächsten Kante aus E weiter.

Um für diesen Algorithmus ein besseres Verständnis zu entwickeln, betrachten wir zunächst sein Verhalten auf einem Sterngraphen $G = (V, E)$. Für einen solchen Graphen

gilt $V = \{v, u_1, \ldots, u_{n-1}\}$ und die Menge E besteht aus den Kanten (v, u_i) für alle i. Das optimale Vertex Cover besteht nur aus dem Knoten v, der alle Kanten abdeckt. Bei jeder Kante fügt der Algorithmus RandVertexCover mit einer Wahrscheinlichkeit von $1/2$ den Knoten v zur Menge U hinzu. Sobald dies einmal passiert, sind alle Kanten des Graphen abgedeckt und es werden keine weiteren Knoten der Menge U hinzugefügt. Sei nun X eine geometrisch verteilte Zufallsvariable mit Parameter $1/2$. Die Anzahl an Knoten, die RandVertexCover zu Menge U hinzufügt, ist verteilt wie die Zufallsvariable $\min\{X, n-1\}$, da der Algorithmus maximal $|E| = n-1$ viele Knoten zu U hinzufügen kann. Es gilt

$$\mathbf{E}[\min\{X, n-1\}] \leq \mathbf{E}[X] = 2 = 2 \cdot \mathrm{OPT}$$

und somit handelt es sich bei RandVertexCover zumindest für dieses Beispiel um einen 2-Approximationsalgorithmus. Wir zeigen nun, dass dies auch allgemein der Fall ist, indem wir beliebige Graphen nicht-disjunkt in Sterngraphen zerlegen.

Theorem 4.16. *Der Algorithmus* RandVertexCover *ist ein randomisierter* 2*-Approximationsalgorithmus für das Vertex Cover Problem.*

Beweis. Aus der Beschreibung des Algorithmus folgt direkt, dass er stets ein Vertex Cover berechnet. Zu zeigen ist also nur, dass dieses im Erwartungswert höchstens doppelt so viele Knoten enthält wie ein minimales Vertex Cover. Sei $G = (V, E)$ eine beliebige Eingabe und sei $Z = \{z_1, \ldots, z_k\} \subseteq V$ ein minimales Vertex Cover von G. Wir bezeichnen mit $V_i \subseteq V$ die Menge der zu z_i adjazenten Knoten. Die Mengen V_i sind im Allgemeinen nicht paarweise disjunkt. Ferner bezeichne $E_i \subseteq E$ die Menge der zu z_i inzidenten Kanten. Da es sich bei Z um ein Vertex Cover handelt, gilt $E_1 \cup \ldots \cup E_k = E$.

Es bezeichne X_i die Anzahl an Knoten aus V_i, die RandVertexCover bei der Betrachtung der Kanten aus E_i der Menge U hinzufügt. Es seien $e_1, \ldots, e_\ell$ die Kanten aus E_i und wir nehmen an, dass RandVertexCover sie in dieser Reihenfolge betrachtet. Zwischendurch betrachtet RandVertexCover im Allgemeinen andere Kanten aus $E \setminus E_i$. Wenn der Algorithmus den Knoten z_i bei der Betrachtung einer Kante $e_j \in E_i$ der Menge U hinzufügt, so wird bei allen weiteren Kanten aus E_i kein weiterer Knoten der Menge U hinzugefügt. Die Situation ist also ähnlich zu dem oben betrachteten Fall des Sterngraphen, allerdings stimmt es nicht, dass X_i identisch verteilt ist wie die Zufallsvariable $\min\{X, |E_i|\}$, wobei X eine geometrisch verteilte Zufallsvariable mit Parameter $1/2$ ist. Der Grund hierfür ist, dass manche Kanten $e_j = (z_i, v)$ zu dem Zeitpunkt, zu dem sie von RandVertexCover betrachtet werden, bereits abgedeckt sind, da v vorher der Menge U hinzugefügt wurde. Dies kann den Wert von X_i aber nur weiter reduzieren. Deshalb gilt

$$\mathbf{E}[X_i] \leq \mathbf{E}[\min\{X, |E_i|\}] \leq \mathbf{E}[X] = 2.$$

Wegen $E_1 \cup \ldots \cup E_k = E$ gilt $|U| = X_1 + \ldots + X_k$. Somit folgt

$$\mathbf{E}[|U|] = \mathbf{E}[X_1 + \ldots + X_k] = \sum_{i=1}^{k} \mathbf{E}[X_i] \leq 2k = 2 \cdot \mathrm{OPT},$$

wobei wir die Linearität des Erwartungswertes ausgenutzt haben. Somit ist bewiesen, dass RandVertexCover eine 2-Approximation berechnet. □

4.3 Max-SAT

Wir beschäftigen uns nun mit dem Problem Max-SAT. Bei diesem Problem besteht die Eingabe aus einer aussagenlogischen Formel in konjunktiver Normalform und das Ziel ist es, eine Variablenbelegung zu finden, die möglichst viele Klauseln erfüllt. In der gewichteten Variante besitzt jede Klausel ein Gewicht und das Ziel besteht darin, das Gesamtgewicht der erfüllten Klauseln zu maximieren. Aus der NP-Schwere des Entscheidungsproblems SAT folgt direkt, dass auch das Problem Max-SAT NP-schwer ist.

Wir nennen eine Formel der Form x oder $\neg x$ für eine Variable x ein *Literal*. Ein Literal der Form x nennen wir *positives Literal* und ein Literal der Form $\neg x$ nennen wir *negatives Literal*. Wir nennen die Variablen, die in der Eingabe vorkommen, im Folgenden $x_1, \ldots, x_n$. Eine *Klausel* C ist eine Disjunktion (Oder-Verknüpfung) von Literalen über diesen Variablen und eine aussagenlogische Formel φ ist in *konjunktiver Normalform (KNF)*, wenn sie eine Konjunktion (Und-Verknüpfung) von Klauseln ist. Oft fassen wir Klauseln auch einfach als Menge von Literalen auf. Eine Formel in konjunktiver Normalform ist dann von der Gestalt

$$\varphi = \bigwedge_{i=1}^{m} \Big(\bigvee_{\ell \in C_i} \ell \Big),$$

wobei $C_1, \ldots, C_m$ die Klauseln von φ sind.

Max-SAT	
Eingabe:	aussagenlogische Formel φ in konjunktiver Normalform mit Variablen $x_1, \ldots, x_n$ und Klauseln $C_1, \ldots, C_m$ Gewichte $w_1, \ldots, w_m \in \mathbb{R}_{>0}$
Lösungen:	alle Belegungen der Variablen $x_1, \ldots, x_n$
Zielfunktion:	maximiere das Gesamtgewicht der durch die Belegung erfüllten Klauseln

Genau wie schon für das Maximum-Cut Problem und das Vertex Cover Problem geben wir für das Problem Max-SAT zunächst einen sehr einfachen Approximationsalgorithmus RandMaxSAT an. Dieser wählt einfach eine uniform zufällige Belegung α der Variablen $x_1, \ldots, x_n$. Das bedeutet, er setzt jede Variable unabhängig von den anderen mit einer Wahrscheinlichkeit von $1/2$ auf 0 und mit einer Wahrscheinlichkeit von $1/2$ auf 1.

Theorem 4.17. *Der Algorithmus* RandMaxSAT *ist ein randomisierter* $\frac{1}{2}$*-Approximationsalgorithmus für* Max-SAT*.*

Beweis. Wir definieren zunächst für jede Klausel C_j eine Zufallsvariable

$$Y_j = \begin{cases} w_j & \text{falls } \alpha \text{ Klausel } C_j \text{ erfüllt,} \\ 0 & \text{sonst.} \end{cases}$$

Dann können wir das Gesamtgewicht Y der durch α erfüllten Klauseln als die Summe $Y_1 + \ldots + Y_m$ schreiben und es gilt

$$\mathbf{E}[Y] = \mathbf{E}\left[\sum_{j=1}^{m} Y_j\right] = \sum_{j=1}^{m} \mathbf{E}[Y_j] = \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}[\alpha \text{ erfüllt } C_j].$$

Die Klausel C_j wird nur dann nicht durch die zufällige Belegung α erfüllt, wenn alle Variablen, die als positives Literal in C_j vorkommen, auf 0 gesetzt sind und alle Variablen, die als negatives Literal in C_j vorkommen, auf 1 gesetzt sind. Es bezeichne $\ell_j = |C_j|$ die Anzahl an Literalen in C_j. Da die oben genannte Belegung der Variablen aus C_j die einzige ist, die die Klausel nicht erfüllt, gilt

$$\mathbf{Pr}[\alpha \text{ erfüllt } C_j \text{ nicht}] \leq \left(\frac{1}{2}\right)^{\ell_j} \leq \frac{1}{2}.$$

Der erste Schritt ist eine Ungleichung und keine Gleichung, da die Klausel C_j für eine Variable x sowohl das Literal x als auch das Literal $\neg x$ enthalten kann. Da eine solche Klausel durch jede beliebige Belegung erfüllt wird, beträgt die Wahrscheinlichkeit, dass C_j durch α nicht erfüllt wird, in diesem Fall 0. Damit folgt insgesamt

$$\mathbf{E}[Y] = \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}[\alpha \text{ erfüllt } C_j] \geq \sum_{j=1}^{m} w_j \cdot \frac{1}{2} = \frac{1}{2} \sum_{j=1}^{m} w_j \geq \frac{1}{2} \cdot \mathrm{OPT},$$

wobei die letzte Ungleichung daraus folgt, dass das Gesamtgewicht aller Klauseln eine obere Schranke für OPT ist. □

Aus dem Beweis des vorangegangenen Theorems folgt, dass der Approximationsfaktor des Algorithmus RandMaxSAT besser wird, je länger die Klauseln sind. Eine Klausel, die ℓ Literale enthält, ist nur mit einer Wahrscheinlichkeit von höchstens $(1/2)^\ell$ nicht durch die zufällige Belegung der Variablen erfüllt. Betrachten wir beispielsweise das 3-SAT-Problem, bei dem alle Klauseln genau 3 Literale enthalten, so ergibt sich ein Approximationsfaktor von $\frac{7}{8}$. Es ist bekannt, dass dies sogar im Wesentlichen der beste erreichbare Approximationsfaktor für 3-SAT unter der Annahme P≠NP ist.

Der Algorithmus RandMaxSAT und seine Analyse sind vollkommen analog zu dem einfachen Approximationsalgorithmus für das Maximum-Cut Problem. Wir zeigen im Folgenden, wie der Algorithmus *derandomisiert* werden kann. Das bedeutet, wir möchten die eigentlich randomisierten Entscheidungen von RandMaxSAT (d. h. die zufällige Belegung α) nun deterministisch so treffen, dass der Algorithmus den gleichen Approximationsfaktor erreicht. Danach werden wir sehen, wie wir mit einem besseren randomisierten Ansatz einen besseren Approximationsfaktor erreichen können.

4.3.1 Derandomisierung

Wir möchten die Wahl der Belegung α des Algorithmus RandMaxSAT nun deterministisch machen. Als weiteres Hilfsmittel aus der Wahrscheinlichkeitsrechnung führen wir dafür *bedingte Erwartungswerte* ein.

Definition 4.18. *Es sei* $(\Omega, \mathbf{Pr})$ *ein diskreter Wahrscheinlichkeitsraum und* $A \subseteq \Omega$ *ein Ereignis mit* $\mathbf{Pr}[A] > 0$. *Außerdem sei* $X : \Omega \to \mathbb{R}$ *eine diskrete Zufallsvariable, die nur Werte aus einer Menge* $M \subseteq \mathbb{R}$ *annehmen kann. Der* bedingte Erwartungswert von X gegeben A *existiert, falls die Reihe* $\sum_{x \in M} |x| \cdot \mathbf{Pr}[X = x \mid A]$ *konvergiert und ist dann definiert als*

$$\mathbf{E}[X \mid A] = \sum_{x \in M} x \cdot \mathbf{Pr}[X = x \mid A].$$

Intuitiv beschreibt der bedingte Erwartungswert den durchschnittlichen Wert von X unter der Bedingung, dass das Ereignis A eintritt. Sehr nützlich für uns ist die folgende Rechenregel für bedingte Erwartungswerte, die man durch eine einfache Rechnung nachweisen kann:

$$\mathbf{E}[X] = \mathbf{Pr}[A] \cdot \mathbf{E}[X \mid A] + \mathbf{Pr}\left[\overline{A}\right] \cdot \mathbf{E}\left[X \mid \overline{A}\right]. \tag{4.2}$$

Ist B ein weiteres Ereignis, so gilt auch

$$\mathbf{E}[X \mid A] = \mathbf{Pr}[B] \cdot \mathbf{E}[X \mid A \wedge B] + \mathbf{Pr}\left[\overline{B}\right] \cdot \mathbf{E}\left[X \mid A \wedge \overline{B}\right]. \tag{4.3}$$

In der Derandomisierung von RandMaxSAT wählen wir die Belegung der Variablen $x_1, \ldots, x_n$ Schritt für Schritt. In Schritt i bestimmen wir die Belegung von x_i und haben davor schon die Belegung der Variablen $x_1, \ldots, x_{i-1}$ festgelegt. Wir betrachten zunächst den ersten Schritt, in dem wir die Belegung von x_1 festlegen. Mit Y bezeichnen wir wieder die Zufallsvariable, die das Gesamtgewicht der erfüllten Klauseln bei einer uniform zufälligen Belegung α beschreibt. Gemäß (4.2) gilt

$$\begin{aligned}\mathbf{E}[Y] &= \mathbf{Pr}[\alpha_1 = 0] \cdot \mathbf{E}[Y \mid \alpha_1 = 0] + \mathbf{Pr}[\alpha_1 = 1] \cdot \mathbf{E}[Y \mid \alpha_1 = 1] \\ &= \frac{1}{2}\left(\mathbf{E}[Y \mid \alpha_1 = 0] + \mathbf{E}[Y \mid \alpha_1 = 1]\right).\end{aligned}$$

Aus dieser Gleichung folgt direkt, dass einer der beiden bedingten Erwartungswerte in der vorangegangenen Zeile mindestens den Wert $\mathbf{E}[Y]$ annehmen muss. Es sei $b_1 \in \{0, 1\}$ so gewählt, dass $\mathbf{E}[Y \mid \alpha_1 = b_1] \geq \mathbf{E}[Y]$ gilt. Wir setzen x_1 nun auf b_1. Wir fahren nach diesem Schema fort und betrachten Schritt i, in dem wir die Belegung von x_i festlegen. Wir gehen davon aus, dass wir bereits Belegungen $b_1, \ldots, b_{i-1}$ der Variablen $x_1, \ldots, x_{i-1}$ gewählt haben, sodass gilt

$$\mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}] \geq \mathbf{E}[Y].$$

Gemäß (4.3) gilt dann

$$\begin{aligned}\mathbf{E}[Y] &\leq \mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}] \\ &= \mathbf{Pr}[\alpha_i = 0] \cdot \mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}, \alpha_i = 0] \\ &\quad + \mathbf{Pr}[\alpha_i = 1] \cdot \mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}, \alpha_i = 1] \\ &= \frac{1}{2} \cdot \mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}, \alpha_i = 0] \\ &\quad + \frac{1}{2} \cdot \mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_{i-1} = b_{i-1}, \alpha_i = 1].\end{aligned}$$

Mit demselben Argument wie oben muss einer der beiden bedingten Erwartungswerte in der letzten und vorletzten Zeile mindestens den Wert $\mathbf{E}[Y]$ haben. Wir setzen x_i nun auf den Wert b_i, für den das der Fall ist. Auf diese Weise können wir induktiv fortfahren und deterministisch eine Belegung $b_1, \ldots, b_n$ der Variablen $x_1, \ldots, x_n$ finden, sodass das Gesamtgewicht der erfüllten Klauseln mindestens $\mathbf{E}[Y]$ beträgt.

Wir müssen nun noch zeigen, dass das oben beschriebene Verfahren effizient ausgeführt werden kann. Der wesentliche Schritt dafür ist die effiziente Bestimmung der bedingten Erwartungswerte. Dies ist mit der folgenden Formel möglich, die die Linearität des Erwartungswertes nutzt, welche auch für bedingte Erwartungswerte gilt:

$$\mathbf{E}[Y \mid \alpha_1 = b_1, \ldots, \alpha_i = b_i] = \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}[\alpha \text{ erfüllt } C_j \mid \alpha_1 = b_1, \ldots, \alpha_i = b_i].$$

Die bedingten Wahrscheinlichkeiten in der vorangegangenen Formel können wie folgt bestimmt werden: Erfüllt eine der Variablen $x_1, \ldots, x_i$ mit der Belegung $b_1, \ldots, b_i$ bereits die Klausel C_j, so beträgt die bedingte Wahrscheinlichkeit 1. Ansonsten sind alle Literale in C_j, die aus den Variablen $x_1, \ldots, x_i$ bestehen, in der Belegung $b_1, \ldots, b_i$ falsch. Diese können aus der Klausel gestrichen werden. Bleiben danach noch ℓ Literale übrig, so beträgt die bedingte Wahrscheinlichkeit, dass die Klausel durch die zufällige Belegung erfüllt wird, $1 - (1/2)^\ell \geq 1/2$. (Wir nehmen an, dass nicht gleichzeitig x und $\overline{x}$ in der Klausel vorkommen, da die Klausel ansonsten mit Wahrscheinlichkeit 1 erfüllt ist.)

Wir erhalten insgesamt einen deterministischen $\frac{1}{2}$-Approximationsalgorithmus für das Problem Max-SAT. Die Methode der bedingten Erwartungswerte, die wir zur Derandomisierung eingesetzt haben, lässt sich auch auf viele andere randomisierte Algorithmen erweitern. Insbesondere können wir mit dieser Methode auch den Algorithmus RandMaxCut derandomisieren.

4.3.2 Verbesserter Algorithmus

Wir werden nun sehen, dass der Approximationsfaktor $\frac{1}{2}$ durch einen einfachen Trick verbessert werden kann. Anstatt jede Variable uniform zufällig mit 0 oder 1 zu belegen, setzen wir jede Variable mit einer Wahrscheinlichkeit von $p > \frac{1}{2}$ unabhängig von den anderen Variablen auf 1. Wir werden sehen, dass dies für eine geeignete Wahl von p zu einem besseren Approximationsfaktor führt. Zunächst betrachten wir nur Formeln, die keine Klauseln enthalten, die nur aus einem einzigen negativen Literal bestehen. Wir nennen solche Klauseln im Folgenden *negative Einheitsklauseln*.

Lemma 4.19. *Es sei φ eine Formel in konjunktiver Normalform über den Variablen $x_1, \ldots, x_n$ mit den Klauseln $C_1, \ldots, C_m$. Die Formel φ enthalte keine negativen Einheitsklauseln. Eine zufällige Belegung α der Variablen $x_1, \ldots, x_n$, die jede Variable unabhängig mit Wahrscheinlichkeit $p \geq 1/2$ auf 1 setzt, erfüllt jede einzelne Klausel mit einer Wahrscheinlichkeit von mindestens* $\min\{p, 1 - p^2\}$.

Beweis. Wir betrachen eine beliebige Klausel C. Falls es sich bei C um eine Klausel der Länge 1 handelt, so besteht sie laut Voraussetzung aus einem positiven Literal.

Das bedeutet, die Klausel C wird von der zufälligen Belegung mit einer Wahrscheinlichkeit von p erfüllt. Betrachten wir nun den Fall, dass es sich bei C um eine Klausel der Länge $\ell \geq 2$ handelt. Wir bezeichnen mit a und b die Anzahl an positiven bzw. negativen Literalen in C. Dann gilt $\ell = a + b$. Enthält C für eine Variable sowohl das entsprechende positive als auch das negative Literal, so ist die Klausel mit Wahrscheinlichkeit 1 erfüllt. Ansonsten gibt es genau eine Belegung der in C vorkommenden Variablen, die die Klausel C nicht erfüllt: alle Variablen, die positiv in C vorkommen, sind auf 0 gesetzt; alle Variablen, die negativ in C vorkommen, sind auf 1 gesetzt. Die Wahrscheinlichkeit, dass diese Belegung gewählt wird, beträgt $p^b(1-p)^a$. Demzufolge beträgt die Wahrscheinlichkeit, dass C erfüllt ist, in diesem Fall

$$1 - p^b(1-p)^a \geq 1 - p^{a+b} = 1 - p^\ell \geq 1 - p^2,$$

wobei wir im ersten Schritt $1-p \leq p$ ausgenutzt haben. Damit ist das Lemma bewiesen. □

Aus dem Lemma folgt direkt das folgende Theorem.

Theorem 4.20. *Für das Problem* Max-SAT *ohne negative Einheitsklauseln erhalten wir einen* $\min\{p, 1-p^2\}$*-Approximationsalgorithmus, indem wir jede Variable mit Wahrscheinlichkeit von* $p \geq 1/2$ *unabhängig auf* 1 *setzen.*

Beweis. Der Beweis folgt mit dem vorangegangenen Lemma und der Linearität des Erwartungswertes. Sei φ eine Formel ohne negative Einheitsklauseln und sei Y wieder das Gesamtgewicht der Klauseln, die durch die zufällige Belegung α erfüllt werden. Dann gilt

$$\begin{aligned}\mathbf{E}[Y] &= \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}[\alpha \text{ erfüllt } C_j] \geq \sum_{j=1}^{m} w_j \cdot \min\{p, 1-p^2\} \\ &= \min\{p, 1-p^2\} \cdot \sum_{j=1}^{m} w_j \geq \min\{p, 1-p^2\} \cdot \mathrm{OPT}.\end{aligned}$$ □

Der Term $\min\{p, 1-p^2\}$ nimmt sein Maximum an, wenn $p = 1-p^2$ gilt. Dies ist für $p = \frac{1}{2}(\sqrt{5}-1) \approx 0{,}618$ der Fall. Nun zeigen wir noch, wie wir die Annahme eliminieren können, dass es keine negativen Einheitsklauseln gibt.

Sei dazu φ eine beliebige Formel in KNF. Zunächst beobachten wir, dass wir eine zu φ äquivalente Formel erhalten, wenn wir für eine Variable x jedes Vorkommen von x durch $\neg x$ und jedes Vorkommen von $\neg x$ durch x ersetzen. Das bedeutet, wir können ohne Beschränkung der Allgemeinheit annehmen, dass für jede Variable x das Gewicht der Einheitsklausel x mindestens so groß ist wie das Gewicht der Einheitsklausel $\neg x$. Hier haben wir implizit die Konvention getroffen, dass Klauseln, die in φ nicht enthalten sind, das Gewicht 0 haben. Insbesondere bedeutet das, dass nicht der Fall eintreten kann, dass φ die Einheitsklausel $\neg x$ mit positivem Gewicht, aber nicht die Einheitsklausel x enthält. Für eine Variable x_i bezeichnen wir im Folgenden mit v_i das Gewicht der negativen Einheitsklausel $\neg x_i$.

Lemma 4.21. *Es gilt* $\mathrm{OPT} \leq \sum_{j=1}^{m} w_j - \sum_{i=1}^{n} v_i$

Beweis. Jede Belegung erfüllt für jede Variable x_i entweder die Einheitsklausel x_i oder die Einheitsklausel $\neg x_i$. Im besten Fall erfüllt eine Belegung alle Klauseln der Länge mindestens zwei und für jede Variable x_i die Einheitsklausel mit größerem Gewicht. Aus dieser Überlegung ergibt sich direkt das Lemma. □

Theorem 4.22. *Für das Problem* Max-SAT *erhalten wir einen* $\min\{p, 1-p^2\}$*-Approximationsalgorithmus, indem wir jede Variable mit Wahrscheinlichkeit* $p \geq 1/2$ *unabhängig auf* 1 *setzen. Dabei nehmen wir ohne Beschränkung der Allgemeinheit an, dass für jede Variable* x *das Gewicht der Einheitsklausel* x *mindestens so groß ist wie das Gewicht der Einheitsklausel* $\neg x$*. Für* $p = \frac{1}{2}(\sqrt{5}-1)$ *ergibt sich ein Approximationsfaktor von* $\frac{1}{2}(\sqrt{5}-1) \approx 0{,}618$*.*

Beweis. Es sei φ eine beliebige Formel in konjunktiver Normalform. Es sei U die Teilmenge der Klauseln von φ, die alle Klauseln aus φ bis auf die negativen Einheitsklauseln, enthält. Das Gesamtgewicht der Klauseln in U beträgt $\sum_{j=1}^{m} w_j - \sum_{i=1}^{n} v_i$.

Wir setzen nun wieder jede Variable unabhängig von den anderen mit einer Wahrscheinlichkeit von p auf 1. Es sei C eine beliebige Klausel aus U. Handelt es sich bei C um eine positive Einheitsklausel, so ist sie mit Wahrscheinlichkeit p erfüllt. Enthält C für eine Variable x die beiden Literale x und $\neg x$, so ist C mit Wahrscheinlichkeit 1 erfüllt. Ansonsten bezeichne $\ell \geq 2$ die Anzahl an Literalen in C. Ferner seien a und b die Anzahlen an positiven bzw. negativen Literalen in C. Die Wahrscheinlichkeit, dass C von der zufälligen Belegung der Variablen erfüllt wird, beträgt analog zu dem Argument aus Lemma 4.19

$$1 - p^b(1-p)^a \geq 1 - p^{a+b} = 1 - p^\ell \geq 1 - p^2.$$

Das Theorem folgt nun wieder aus der Linearität des Erwartungswertes. Sei Y die Zufallsvariable, die das Gesamtgewicht der erfüllten Variablen angibt. Dann gilt

$$\begin{aligned}
\mathbf{E}[Y] &\geq \sum_{C_j \in U} w_j \cdot \mathbf{Pr}[\alpha \text{ erfüllt } C_j] \geq \sum_{C_j \in U} w_j \cdot \min\{p, 1-p^2\} \\
&= \min\{p, 1-p^2\} \cdot \left(\sum_{j=1}^{m} w_j - \sum_{i=1}^{n} v_i \right) \\
&\geq \min\{p, 1-p^2\} \cdot \mathrm{OPT}.
\end{aligned}$$

□

Auch der Algorithmus aus Theorem 4.22 kann mit der Methode der bedingten Erwartungswerte derandomisiert werden.

Kapitel 5

Lineare Programmierung und Runden

Wir haben in der Vorlesung „Algorithmen und Berechnungskomplexität I“ bereits *lineare Programmierung* als ein wichtiges algorithmisches Werkzeug zur Lösung von Optimierungsproblemen kennengelernt. Auch im Bereich der Approximationsalgorithmen ist dieses Werkzeug von unschätzbarer Bedeutung, da die besten bekannten Approximationsalgorithmen für viele Probleme lineare Programmierung als zentrales Hilfsmittel einsetzen. Wir werden in diesem Kapitel einige solche Algorithmen kennenlernen. Zunächst wiederholen wir jedoch kurz die notwendigen Grundlagen.

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*

$$c_1 x_1 + \ldots + c_d x_d \tag{5.1}$$

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

$$\begin{aligned} a_{11} x_1 + \ldots + a_{1d} x_d &\leq b_1 \\ &\vdots \\ a_{m1} x_1 + \ldots + a_{md} x_d &\leq b_m \end{aligned} \tag{5.2}$$

einhält. Statt dem Relationszeichen $\leq$ erlauben wir 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.

Wir definieren $x = (x_1, \ldots, x_d)^{\mathrm{T}}$ und $c = (c_1, \ldots, c_d)^{\mathrm{T}}$, um die Notation zu vereinfachen. Damit sind $x \in \mathbb{R}^d$ und $c \in \mathbb{R}^d$ Spaltenvektoren und wir können die Zielfunktion (5.1) 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 $b = (b_1, \ldots, b_m)^{\mathrm{T}} \in \mathbb{R}^m$. Dann

entspricht jede Zeile der Matrix einer Nebenbedingung und wir können die Nebenbedingungen (5.2) 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 $c \cdot x$

Zur Illustration zeigen wir nun, wie wir das Problem, einen maximalen Fluss zu berechnen, als lineares Programm formulieren können.

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

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

unter den folgenden Bedingungen maximieren:

$$\forall e \in E : x_e \geq 0 \qquad \text{(Fluss nicht negativ)},$$

$$\forall e \in E : x_e \leq c(e) \qquad \text{(Fluss nicht größer als Kapazität)},$$

$$\forall v \in V \setminus \{s, t\} : \sum_{e=(u,v)} x_e - \sum_{e=(v,u)} x_e = 0 \qquad \text{(Flusserhaltung)}.$$

Die letzten Nebenbedingungen sind als Gleichungen und nicht als Ungleichungen formuliert. Dies ist ohne Erweiterung des Modells möglich, da wir jede Gleichung durch zwei Ungleichungen mit den Relationszeichen $\leq$ und $\geq$ ersetzen können.

Lineare Programme können in polynomieller Zeit gelöst werden. Auch in praktischen Anwendungen ist es oft möglich, sehr große lineare Programme schnell zu lösen. Im Folgenden werden *ganzzahlige lineare Programme* eine große Rolle spielen. Dabei handelt es sich um lineare Programme, bei denen man zusätzlich als Nebenbedingung fordert, dass die Werte aller (oder einiger) Variablen ganzzahlig sein müssen. Man sucht also unter allen ganzzahligen Variablenbelegungen, die die linearen Nebenbedingungen erfüllen, eine mit maximalem oder minimalem Zielfunktionswert. Das Finden einer solchen Variablenbelegung ist NP-schwer, was z. B. daraus folgt, dass das Rucksackproblem als ganzzahliges lineares Programm formuliert werden kann.

5.1 Max-SAT

Wir greifen nun das Problem Max-SAT wieder auf, für das wir bereits in Kapitel 4 einen randomisierten 0,618-Approximationsalgorithmus entworfen haben. Wir werden

sehen, dass mithilfe von linearer Programmierung ein besserer Approximationsfaktor erzielt werden kann.

5.1.1 Randomisiertes Runden

Sei φ eine Eingabe für Max-SAT mit Variablen $x_1, \ldots, x_n$, Klauseln $C_1, \ldots, C_m$ und Gewichten $w_1, \ldots, w_m$. Zunächst formulieren wir die Aufgabe, eine Belegung zu finden, für die das Gesamtgewicht der erfüllten Klauseln maximal wird, als ganzzahliges lineares Programm. Dazu führen wir für jede Aussagenvariable x_i in der Formel φ eine ganzzahlige Variable y_i ein, die nur die Werte 0 oder 1 annehmen darf. Diese Variablen werden die Belegung der Aussagenvariablen $x_1, \ldots, x_n$ codieren. Ferner enthält das ganzzahlige lineare Programm für jede Klausel C_j eine Variable z_j, die ebenfalls nur die Werte 0 oder 1 annehmen kann. Dabei soll $z_j = 1$ genau dann gelten, wenn die durch $y_1, \ldots, y_n$ codierte Belegung der Aussagenvariablen die Klausel C_j erfüllt. Das Gesamtgewicht der erfüllten Klauseln lässt sich dann als $\sum_{j=1}^{m} w_j z_j$ schreiben.

Nun müssen wir durch geeignete Nebenbedingungen die gewünschte Beziehung zwischen den Variablen y_i und z_j herstellen. Wir müssen erreichen, dass die Belegung $z_j = 1$ nur dann möglich ist, wenn die Belegung $y_1, \ldots, y_n$ die Klausel C_j erfüllt. Es bezeichnen P_j und N_j die Indizes der Variablen, die positiv bzw. negativ in C_j vorkommen, d. h.

$$C_j = \left(\bigvee_{i \in P_j} x_i \right) \vee \left(\bigvee_{i \in N_j} \neg x_i \right).$$

Wir fügen für jede Klausel C_j die Nebenbedingung

$$z_j \leq \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i)$$

zu dem ganzzahligen linearen Programm hinzu. Diese garantiert, dass z_j nur dann auf 1 gesetzt werden kann, wenn entweder eine positiv in C_j vorkommende Variable auf 1 oder eine negativ in C_j vorkommende Variable auf 0 gesetzt wurde. Zusammengefasst können wir das Finden einer optimalen Belegung wie folgt als ganzzahliges lineares Programm schreiben:

$$\begin{aligned}
\text{maximiere } & \sum_{j=1}^{m} w_j z_j & & \\
\text{sodass } & \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i) \geq z_j, & & \forall j \in \{1, \ldots, m\}, \\
& y_i \in \{0, 1\}, & & \forall i \in \{1, \ldots, n\}, \\
& 0 \leq z_j \leq 1, & & \forall j \in \{1, \ldots, m\}.
\end{aligned}$$

Auch für die Variablen z_j hätten wir fordern können, dass sie nur die Werte 0 oder 1 annehmen dürfen. Dies ist in diesem Fall aber gar nicht notwendig, da sie aufgrund der Zielfunktion immer auf den größten Wert gesetzt werden, der durch die Nebenbedingungen erlaubt ist. Da alle y_i ganzzahlig sind, sind die z_j dementsprechend in jeder optimalen Variablenbelegung auf 0 oder 1 gesetzt.

Wir haben nun das Problem, eine optimale Variablenbelegung zu finden, auf das Lösen eines ganzzahligen linearen Programms reduziert. Auf den ersten Blick ist dies wenig hilfreich, da es NP-schwer ist, ganzzahlige lineare Programme zu lösen. Wir werden das ganzzahlige lineare Programm nun aber *relaxieren*, d. h. wir werden die Bedingung aufgeben, dass alle oder gewisse Variablen nur ganzzahlige Werte annehmen dürfen. Dann erhalten wir das folgende lineare Programm:

$$\begin{aligned} \text{maximiere } & \sum_{j=1}^{m} w_j z_j & & \\ \text{sodass } & \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i) \geq z_j, & & \forall j \in \{1, \ldots, m\}, \\ & 0 \leq y_i \leq 1, & & \forall i \in \{1, \ldots, n\}, \\ & 0 \leq z_j \leq 1, & & \forall j \in \{1, \ldots, m\}. \end{aligned}$$

Der einzige Unterschied zum ganzzahligen Programm ist, dass wir $y_i \in [0, 1]$ statt $y_i \in \{0, 1\}$ fordern. Nun scheint es wieder so, als hätten wir nicht viel gewonnen. Zwar haben wir ein lineares Programm erhalten, das wir in polynomieller Zeit lösen können, die optimale Lösung ist nun aber im Allgemeinen nicht mehr ganzzahlig und entspricht somit keiner Belegung der Aussagenvariablen $x_1, \ldots, x_n$ in der Formel φ. Dennoch ist die optimale Belegung der Variablen in dem linearen Programm für den Entwurf eines Approximationsalgorithmus nützlich.

Wir bezeichnen im Folgenden mit Z^*_{LP} und Z^*_{GP} die optimalen Zielfunktionswerte des linearen bzw. des ganzzahligen Programms. Es gilt dann $Z^*_{\text{GP}} = \text{OPT}$, wobei OPT das Gesamtgewicht der Klauseln angibt, die in einer optimalen Belegung der Aussagenvariablen $x_1, \ldots, x_n$ erfüllt sind. Da es sich bei dem linearen Programm um eine Relaxierung handelt und alle im ganzzahligen Programm erlaubten Variablenbelegungen auch im linearen Programm erlaubt sind, gilt $Z^*_{\text{LP}} \geq Z^*_{\text{GP}}$. Der optimale Wert des linearen Programms ist also eine obere Schranke für den Wert der optimalen Lösung.

Es bezeichne (y^*, z^*) mit $y^* = (y^*_1, \ldots, y^*_n)$ und $z^* = (z^*_1, \ldots, z^*_m)$ eine optimale Lösung des linearen Programms. Der Algorithmus, den wir im Folgenden analysieren werden, berechnet zunächst eine optimale Lösung des linearen Programms und setzt dann jede Aussagenvariable x_i unabhängig von den anderen mit einer Wahrscheinlichkeit von y^*_i auf 1 und sonst auf 0. Diesen Algorithmus nennen wir *randomisiertes Runden*.

Theorem 5.1. *Randomisiertes Runden ist ein randomisierter* $(1 - \frac{1}{e})$*-Approximationsalgorithmus für* Max-SAT.

Beweis. Genau wie in den Analysen der randomisierten Algorithmen in Abschnitt 4.3 betrachten wir auch in diesem Beweis für jede Klausel separat die Wahrscheinlichkeit, dass sie durch die zufällige Belegung der Variablen erfüllt wird. Mit der oben eingeführten Notation gilt für jede Klausel C_j

$$\mathbf{Pr}\left[C_j \text{ nicht erfüllt}\right] = \prod_{i \in P_j} (1 - y^*_i) \cdot \prod_{i \in N_j} y^*_i. \tag{5.3}$$

Dies folgt wieder daraus, dass es nur eine Belegung der in C_j vorkommenden Variablen gibt, die die Klausel C_j nicht erfüllt: alle positiv auftretenden Variablen sind auf 0

gesetzt und alle negativ auftretenden Variablen auf 1. Das Produkt auf der rechten Seite beschreibt genau die Wahrscheinlichkeit, dass diese Belegung gewählt wird. Um das Produkt abzuschätzen, setzen wir die Ungleichung vom arithmetischen und geometrischen Mittel ein. Diese besagt, dass für beliebige Zahlen $a_1, \ldots, a_n \in \mathbb{R}_{\geq 0}$ das geometrische Mittel höchstens so groß ist wie das arithmetische, also

$$\sqrt[n]{\prod_{i=1}^{n} a_i} \leq \frac{1}{n} \sum_{i=1}^{n} a_i \quad \text{oder äquivalent} \quad \prod_{i=1}^{n} a_i \leq \left(\frac{1}{n} \sum_{i=1}^{n} a_i \right)^n .$$

Angewendet auf (5.3) ergibt sich mit $|C_j| = |P_j| + |N_j|$

$$\begin{aligned} \prod_{i \in P_j} (1 - y_i^*) \cdot \prod_{i \in N_j} y_i^* &\leq \left(\frac{1}{|C_j|} \left(\sum_{i \in P_j} (1 - y_i^*) + \sum_{i \in N_j} y_i^* \right) \right)^{|C_j|} \\ &= \left(\frac{1}{|C_j|} \left(|C_j| + \sum_{i \in P_j} (-y_i^*) + \sum_{i \in N_j} (y_i^* - 1) \right) \right)^{|C_j|} \\ &= \left(1 - \frac{1}{|C_j|} \left(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*) \right) \right)^{|C_j|} . \end{aligned}$$

Jetzt nutzen wir aus, dass das lineare Programm die Nebenbedingung

$$\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*) \geq z_j^*$$

garantiert. Damit ergibt sich insgesamt

$$\mathbf{Pr}\left[C_j \text{ nicht erfüllt}\right] \leq \left(1 - \frac{z_j^*}{|C_j|} \right)^{|C_j|}$$

und dementsprechend

$$\mathbf{Pr}\left[C_j \text{ erfüllt}\right] \geq 1 - \left(1 - \frac{z_j^*}{|C_j|} \right)^{|C_j|} .$$

Wir betrachten nun die Funktion $f(x) = 1 - \left(1 - \frac{x}{|C_j|}\right)^{|C_j|}$ genauer. Man rechnet leicht nach, dass die zweite Ableitung dieser Funktion auf dem Intervall $[0, 1]$ für jedes $|C_j| \geq 1$ nicht positiv ist. Das bedeutet, die Funktion ist konkav. Somit liegt die Gerade g, die durch die Punkte $(0, f(0))$ und $(1, f(1))$ verläuft, an keiner Stelle über der Funktion f (siehe Abbildung 5.1). Wegen $f(0) = 0$ entspricht die Gerade g der Funktion

$$g(x) = f(1) \cdot x = \left(1 - \left(1 - \frac{1}{|C_j|} \right)^{|C_j|} \right) \cdot x.$$

Es gilt also insgesamt

$$\mathbf{Pr}\left[C_j \text{ erfüllt}\right] \geq f(z_j^*) \geq g(z_j^*) = \left(1 - \left(1 - \frac{1}{|C_j|} \right)^{|C_j|} \right) \cdot z_j^* .$$

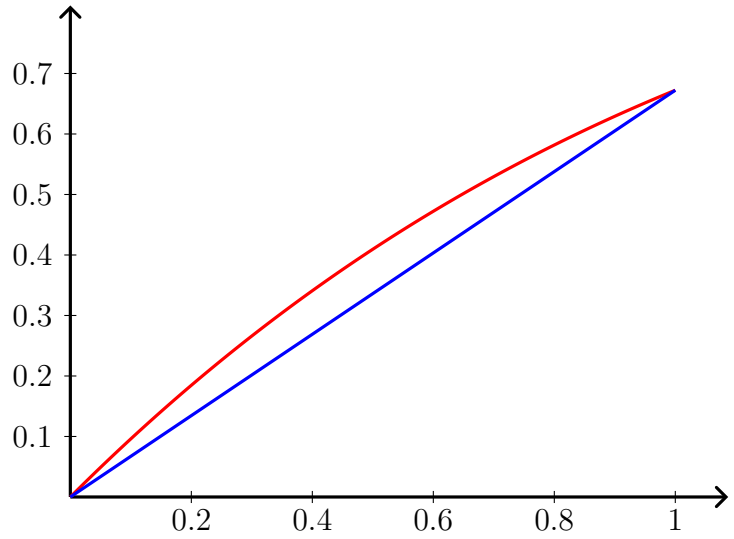


Abbildung 5.1: Die rote Kurve entspricht der Funktion $f(x) = 1 - \left(1 - \frac{x}{|C_j|}\right)^{|C_j|}$ für $|C_j| = 5$. Die blaue Kurve ist die lineare Interpolation $g(x)$, die durch die Punkte $(0, f(0))$ und $(1, f(1))$ verläuft.

Bezeichnen wir wieder mit Y das Gesamtgewicht der erfüllten Klauseln, so ergibt sich mit der Linearität des Erwartungswertes wie in Abschnitt 4.3

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}[C_j \text{ erfüllt}] \\
&\geq \sum_{j=1}^{m} w_j \left(1 - \left(1 - \frac{1}{|C_j|}\right)^{|C_j|}\right) \cdot z_j^* \\
&\geq \min_{k \geq 1} \left(1 - \left(1 - \frac{1}{k}\right)^{k}\right) \cdot \sum_{j=1}^{m} w_j z_j^* \\
&= \min_{k \geq 1} \left(1 - \left(1 - \frac{1}{k}\right)^{k}\right) \cdot Z_{\mathrm{LP}}^* \\
&\geq \min_{k \geq 1} \left(1 - \left(1 - \frac{1}{k}\right)^{k}\right) \cdot \mathrm{OPT}.
\end{aligned}$$

Die Funktion $1 - \left(1 - \frac{1}{k}\right)^k$ ist streng monoton fallend und geht von oben gegen $1 - 1/e$. Damit erhalten wir

$$\mathbf{E}[Y] \geq \left(1 - \frac{1}{e}\right) \cdot \mathrm{OPT}. \qquad \square$$

Es gilt $\left(1 - \frac{1}{e}\right) \approx 0{,}632$. Somit erreicht das randomisierte Runden einen leicht besseren Approximationsfaktor als die Algorithmen, die wir in Abschnitt 4.3 kennengelernt haben. Wir werden im Folgenden sehen, dass eine weitere Verbesserung möglich ist, wenn wir randomisiertes Runden mit dem einfachen Algorithmus RandMaxSAT kombinieren.

5.1.2 Kombination beider Algorithmen

Betrachten wir den Algorithmus RandMaxSAT und das randomisierte Runden genauer, so stellen wir fest, dass der Approximationsfaktor von RandMaxSAT besser wird, je länger die Klauseln sind, und dass der Approximationsfaktor von randomisiertem Runden besser wird, je kürzer die Klauseln sind. Konkret gilt für jede Klausel C_j, dass sie durch das randomisierte Runden mit einer Wahrscheinlichkeit von mindestens $\left(1 - \left(1 - \frac{1}{|C_j|}\right)^{|C_j|}\right) z_j^*$ erfüllt wird und durch den Algorithmus RandMaxSAT mit einer Wahrscheinlichkeit von mindestens $1 - 2^{-|C_j|}$. Kurze Klauseln werden also mit recht hoher Wahrscheinlichkeit durch das randomisierte Runden erfüllt und lange Klauseln durch RandMaxSAT. Der Algorithmus BetterOfTwo führt das randomisierte Runden und den Algorithmus RandMaxSAT je einmal aus und gibt das bessere der beiden Ergebnisse aus.

Theorem 5.2. *Der Algorithmus* BetterOfTwo *ist ein randomisierter* 0,75*-Approximationsalgorithmus für* Max-SAT*.*

Beweis. Wir bezeichnen mit Y_1 und Y_2 das Gesamtgewicht der durch randomisiertes Runden bzw. RandMaxSAT erfüllten Klauseln. Da BetterOfTwo beide Algorithmen ausführt und anschließend das bessere Ergebnis wählt, interessieren wir uns für den Erwartungswert von $\max\{Y_1, Y_2\}$. Der Leser beachte, dass für zwei Zufallsvariablen A und B der Erwartungswert $\mathbf{E}[\max\{A, B\}]$ im Allgemeinen nicht gleich dem Maximum $\max\{\mathbf{E}[A], \mathbf{E}[B]\}$ der Erwartungswerte ist. Da das Maximum zweier Werte aber nie kleiner als der Durchschnitt ist, gilt

$$\mathbf{E}[\max\{Y_1, Y_2\}] \geq \mathbf{E}\left[\frac{1}{2}(Y_1 + Y_2)\right] = \frac{1}{2}\mathbf{E}[Y_1] + \frac{1}{2}\mathbf{E}[Y_2],$$

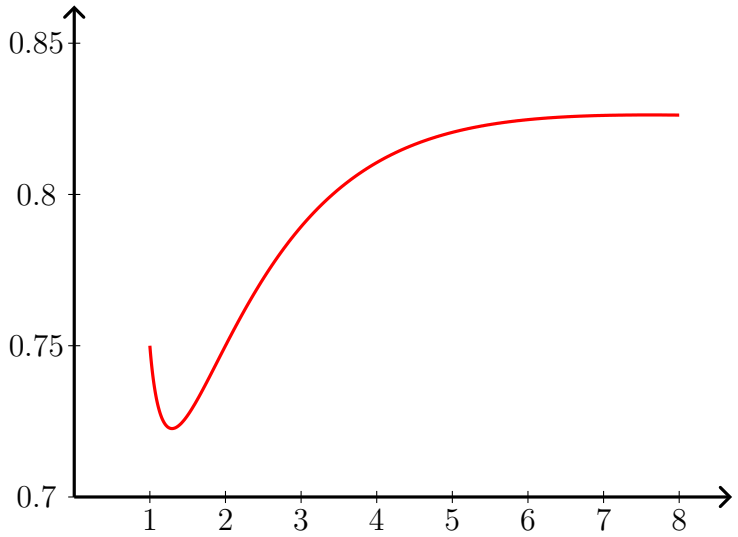
wobei wir im zweiten Schritt die Linearität des Erwartungswertes ausgenutzt haben. Wir setzen nun ein, was wir über die Erwartungswerte von Y_1 und Y_2 aus den Beweisen von Theorem 5.1 und Theorem 4.17 wissen, und erhalten

$$\begin{aligned}\frac{1}{2}\mathbf{E}[Y_1] + \frac{1}{2}\mathbf{E}[Y_2] &\geq \frac{1}{2}\sum_{j=1}^{m} w_j \left(1 - \left(1 - \frac{1}{|C_j|}\right)^{|C_j|}\right) \cdot z_j^* + \frac{1}{2}\sum_{j=1}^{m} w_j(1 - 2^{-|C_j|}) \\ &\geq \sum_{j=1}^{m} w_j z_j^* \left(\frac{1}{2}\left(1 - \left(1 - \frac{1}{|C_j|}\right)^{|C_j|}\right) + \frac{1}{2}(1 - 2^{-|C_j|})\right).\end{aligned}$$

Wir betrachten nun die Funktion

$$f(x) = \frac{1}{2}\left(1 - \left(1 - \frac{1}{x}\right)^{x}\right) + \frac{1}{2}(1 - 2^{-x}),$$

welche in der folgenden Abbildung dargestellt ist.



Die Abbildung lässt vermuten, dass die Funktion für alle ganzzahligen Werte mindestens den Wert 0,75 annimmt. Dass dies tatsächlich der Fall ist, rechnet man leicht nach: Es gilt

$$f(1) = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4} \quad \text{und} \quad f(2) = \frac{1}{2}\left(1 - \frac{1}{4}\right) + \frac{1}{2}\left(1 - \frac{1}{4}\right) = \frac{3}{4}.$$

Für $x \geq 3$ gilt

$$f(x) \geq \frac{1}{2}\left(1 - \frac{1}{e}\right) + \frac{1}{2}\left(1 - 2^{-3}\right) \approx 0{,}753 \geq \frac{3}{4}.$$

Damit gilt insgesamt

$$\mathbf{E}\left[\max\{Y_1, Y_2\}\right] \geq \frac{3}{4} \sum_{j=1}^{m} w_j z_j^* = \frac{3}{4} \cdot Z_{\text{LP}}^* \geq \frac{3}{4} \cdot \text{OPT}. \qquad \square$$

5.1.3 Nichtlineares randomisiertes Runden

Wir werden in diesem Abschnitt sehen, dass auch mit randomisiertem Runden allein ein Approximationsfaktor von $\frac{3}{4}$ für Max-SAT erreicht werden kann, ohne zusätzlich auf RandMaxSAT zurückgreifen zu müssen. Dazu muss das Runden des linearen Programms nichtlinear erfolgen. Es bezeichne (y^*, z^*) mit $y^* = (y_1^*, \ldots, y_n^*)$ und $z^* = (z_1^*, \ldots, z_m^*)$ wieder eine optimale Lösung des linearen Programms, das wir bereits in Abschnitt 5.1.1 beim randomisierten Runden eingesetzt haben. Außerdem sei $f : [0,1] \to [0,1]$ eine zunächst beliebige Funktion. Der Algorithmus RandRunden$_f$ berechnet als erstes die optimale Lösung (y^*, z^*) und setzt dann die Aussagenvariable x_i mit einer Wahrscheinlichkeit von $f(y_i^*)$ auf 1 und sonst auf 0. Für die Identitätsfunktion $f(x) = x$ entspricht dies dem randomisierten Runden, das wir in Abschnitt 5.1.1 kennengelernt und analysiert haben.

Wir analysieren nun den Approximationsfaktor von RandRunden$_f$ für eine beliebige Funktion $f : [0,1] \to [0,1]$ mit

$$1 - 4^{-x} \leq f(x) \leq 4^{x-1} \tag{5.4}$$

für alle $x \in [0,1]$. Wir werden sehen, dass jede solche Funktion sicherstellt, dass jede Klausel C_j mit einer Wahrscheinlichkeit von mindestens $\frac{3}{4}z_j^*$ erfüllt ist.

Theorem 5.3. *Für jede Funktion mit Eigenschaft* (5.4) *ist* RandRunden$_f$ *ein randomisierter* 0,75*-Approximationsalgorithmus für* Max-SAT.

Beweis. Wir zeigen für jede Klausel C_j, dass sie mit einer Wahrscheinlichkeit von mindestens $\frac{3}{4}z_j^*$ erfüllt ist. Das Theorem folgt dann aus dieser Aussage analog zum Beweis von Theorem 5.1.

Sei C_j eine beliebige Klausel. Wieder bezeichnen P_j und N_j die Mengen der Variablen, die positiv bzw. negativ in C_j vorkommen. Die Klausel C_j ist nur dann nicht erfüllt, wenn alle positiv auftretenden Variablen auf 0 und alle negativ auftretenden Variablen auf 1 gesetzt sind. Daraus folgt

$$\mathbf{Pr}\left[C_j \text{ nicht erfüllt}\right] = \prod_{i \in P_j} (1 - f(y_i^*)) \cdot \prod_{i \in N_j} f(y_i^*) \leq \prod_{i \in P_j} 4^{-y_i^*} \cdot \prod_{i \in N_j} 4^{y_i^* - 1},$$

wobei wir Eigenschaft (5.4) ausgenutzt haben. Es gilt

$$\prod_{i \in P_j} 4^{-y_i^*} \cdot \prod_{i \in N_j} 4^{y_i^* - 1} = 4^{-\left(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*)\right)} \leq 4^{-z_j^*}.$$

Die letzte Ungleichung folgt, da das lineare Programm die Ungleichung

$$\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*) \geq z_j^*$$

garantiert. Es gilt somit

$$\mathbf{Pr}\left[C_j \text{ erfüllt}\right] \geq 1 - 4^{-z_j^*}.$$

Da die Funktion $g(x) = 1 - 4^{-x}$ in $[0,1]$ konkav ist, ist die Gerade h, die durch die Punkte $(0, g(0))$ und $(1, g(1))$ geht, eine untere Schranke (im Beweis von Theorem 5.1 haben wir ein analoges Argument genutzt). Wegen $g(0) = 0$ gilt $h(x) = g(1) \cdot x$ und somit insgesamt

$$\mathbf{Pr}\left[C_j \text{ erfüllt}\right] \geq 1 - 4^{-z_j^*} = g(z_j^*) \geq h(z_j^*) = g(1) \cdot z_j^* = \left(1 - \frac{1}{4}\right) z_j^* = \frac{3}{4} z_j^*.$$

Wieder bezeichnen wir mit Y das Gesamtgewicht der erfüllten Klauseln. Insgesamt ergibt sich

$$\begin{aligned} \mathbf{E}\left[Y\right] &= \sum_{j=1}^{m} w_j \cdot \mathbf{Pr}\left[C_j \text{ erfüllt}\right] \\ &\geq \sum_{j=1}^{m} w_j \cdot \frac{3}{4} z_j^* = \frac{3}{4} \cdot \sum_{j=1}^{m} w_j z_j^* \\ &= \frac{3}{4} \cdot Z_{\text{LP}}^* \geq \frac{3}{4} \cdot \text{OPT}. \end{aligned}$$

□

Es stellt sich nun natürlich die Frage, ob durch ein geschickteres randomisiertes Runden ein besserer Approximationsfaktor als 0,75 erreicht werden kann. Dazu betrachten wir zunächst ein Beispiel.

Instanz für MAX-SAT

Wir betrachten die Formel

$$\varphi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2),$$

wobei alle Klauseln das Gewicht 1 haben. Man sieht leicht, dass jede Belegung der Aussagenvariablen genau drei der Klauseln erfüllt. Somit gilt OPT = 3. In der optimalen Lösung des linearen Programms für diese Formel gilt $y_1^* = y_2^* = \frac{1}{2}$ und $z_j = 1$ für alle Klauseln. Das bedeutet, es gilt $Z_{\text{LP}}^* = 4$.

Das obigen Beispiel zeigt, dass es Eingaben φ für MAX-SAT gibt, für die $\text{OPT} \leq \frac{3}{4} Z_{\text{LP}}^*$ gilt. Das bedeutet, wenn wir bei der Analyse eines Approximationsalgorithmus für MAX-SAT als obere Schranke nur den Wert Z_{LP}^* einer optimalen Lösung des linearen Programms benutzen, so können wir keinen besseren Approximationsfaktor als $\frac{3}{4}$ zeigen. Die folgende Definition verallgemeinert diese Idee auf beliebige lineare Programme.

Definition 5.4. *Das* Integrality Gap *eines ganzzahligen Programms ist der Quotient aus dem optimalen Wert des ganzzahligen Programms und dem optimalen Wert des linearen Programms, das man erhält, wenn man die Ganzzahligkeitsbedingungen relaxiert.*

Haben wir für ein Optimierungsproblem eine Formulierung der Instanzen als ganzzahlige Programme gefunden, so bezeichnen wir mit dem Integrality Gap dieser Formulierung *das schlechteste Integrality Gap einer Instanz des Optimierungsproblems. Bei Maximierungsproblemen betrachten wir also das kleinste Integrality Gap über alle Instanzen und bei Minimierungsproblemen das größte.*

Allgemein gilt, dass wir für einen Approximationsalgorithmus für ein Optimierungsproblem keinen besseren Approximationsfaktor als das Integrality Gap zeigen können, wenn wir als einzige Schranke für den Wert einer optimalen Lösung die optimale Lösung des relaxierten linearen Programms benutzen.

5.2 Facility Location ohne Kapazitäten

Wir untersuchen in diesem Abschnitt das Problem *Facility Location*.

Facility Location (ohne Kapazitäten)

Eingabe:	Menge F von Standorten (Facilities), Menge D von Kunden Distanzen $d : F \times D \to \mathbb{R}_{\geq 0}$, Kosten $f : F \to \mathbb{R}_{\geq 0}$ (wir schreiben d_{ij} für $d(i,j)$ und f_i für $f(i)$)
Lösungen:	alle Teilmengen $S \subseteq F$ mit $S \neq \emptyset$
Zielfunktion:	minimiere $\sum_{i \in S} f_i + \sum_{j \in D} \min_{i \in S} d_{ij}$

Anschaulich geht es beim Facility-Location-Problem darum, eine optimale Menge von Fabrikstandorten zu wählen. Es gibt eine Menge von Kunden, die beliefert werden müssen. Die Kosten, um einen Kunden von einer bestimmten Fabrik zu beliefern, sind durch die Distanzen d gegeben, und wir gehen davon aus, dass jeder Kunde von der nächstgelegenen geöffneten Fabrik beliefert wird. Die Gesamtkosten einer Menge von Fabrikstandorten ergeben sich durch die Kosten zum Eröffnen der Standorte (gegeben durch die Funktion f) sowie die Kosten, um alle Kunden zu beliefern. Facility Location ist ein wichtiges NP-schweres Problem, weshalb viel Arbeit in den Entwurf von Approximationsalgorithmen für dieses Problem geflossen ist. Es gibt noch weitere Problemvarianten, bei denen die einzelnen Standorte Kapazitäten haben und jeweils nur eine bestimmte Anzahl an Kunden beliefern können. Wir beschäftigen uns in dieser Vorlesung allerdings ausschließlich mit der Variante ohne Kapazitäten, ohne dies jedes Mal explizit zu erwähnen.

5.2.1 Reduktion auf Set Cover

Zunächst stellen wir fest, dass wir Facility Location als einen Spezialfall von Set Cover auffassen können und sich deshalb auch der Approximationsfaktor des Greedy-Algorithmus aus Theorem 2.2 überträgt. Sei dazu (D, F, d, f) eine Eingabe für Facility Location. Wir konstruieren daraus eine Eingabe für Set Cover, bei der die Kunden die abzudeckende Grundmenge S bilden. Die Familie der Teilmengen von S, die wir zur Abdeckung einsetzen können, nennen wir T. Die Familie T enthält dabei jede mögliche Teilmenge $A \subseteq S$ mit $A \neq \emptyset$ genau $|F|$ mal und zwar für jedes $i \in F$ einmal mit Kosten $f_i + \sum_{j \in A} d_{ij}$. Auf diese Weise können wir jede Menge aus T einem Standort $i \in F$ zuordnen.

Zunächst beobachten wir, dass in einer optimalen Lösung für die konstruierte Instanz von Set Cover keine zwei Mengen $S_1, S_2 \in T$ ausgewählt sein können, die demselben Standort $i \in F$ zugeordnet sind. Diese könnten nämlich durch die Vereinigung $S_1 \cup S_2$ ersetzt werden, was die Kosten um f_i senken würde. Uns interessieren deshalb nur Lösungen für die Instanz von Set Cover, die für jedes $i \in F$ höchstens eine Menge aus T auswählen. Ebenso können wir davon ausgehen, dass die ausgewählten Mengen aus T disjunkt sind, da wir ansonsten Elemente aus den Mengen entfernen könnten, ohne die Kosten zu erhöhen.

Zwischen den Teilmengen von T, die alle Elemente aus S abdecken, paarweise disjunkt sind und für kein $i \in F$ mehr als eine Menge enthalten, auf der einen Seite und den Lösungen für die Instanz von Facility Location auf der anderen Seite existiert eine naheliegende Bijektion. Modulo dieser Bijektion stimmen auch die optimalen Lösungen überein.

Wir wenden nun den Greedy-Algorithmus für Set Cover auf die konstruierte Instanz an. Dieser wählt in jedem Schritt eine Menge aus T mit minimalen relativen Kosten aus und fügt diese der Lösung hinzu. Hat er mit den bislang gewählten Mengen aus T bereits eine Teilmenge $D' \subseteq D$ der Kunden abgedeckt, so bedeutet dies, dass er einen

Standort $i \in F$ und eine Menge $A \subseteq D$ auswählt, für die

$$\frac{f_i + \sum_{j \in A} d_{ij}}{|A \setminus D'|} \tag{5.5}$$

minimal wird. Gemäß Theorem 2.2 erreicht dieser Algorithmus einen Approximationsfaktor von $H_{|D|} = O(\log |D|)$. Allerdings ist nicht klar, dass der Greedy-Algorithmus eine polynomielle Laufzeit besitzt, da die Menge T exponentiell groß ist. Dennoch kann in polynomieller Zeit ein Paar (i, A) mit $i \in F$ und $A \subseteq D$ gefunden werden, für das (5.5) minimal wird. Dazu ist die Beobachtung wichtig, dass für ein Paar (i, s) mit $i \in F$ und $s \in \{1, \ldots, |D \setminus D'|\}$ in polynomieller Zeit eine Menge A mit $|A \setminus D'| = s$ gefunden werden kann, für die (5.5) minimal wird. Dazu genügt es, A als die Menge der s nächsten Nachbarn von i aus der Menge $D \setminus D'$ zu wählen. Der Algorithmus testet dann alle polynomiell vielen Wahlen von (i, s) mit $i \in F$ und $s \in \{1, \ldots, |D \setminus D'|\}$ und findet so die Teilmenge mit den insgesamt kleinsten relativen Kosten.

Theorem 5.5. *Für das Facility-Location-Problem existiert ein $O(\log |D|)$-Approximationsalgorithmus.*

5.2.2 Metrisches Facility Location

Wir betrachten nun den wichtigen Spezialfall, dass die Menge $D \cup F$ in eine Metrik eingebettet ist. Das bedeutet, es gilt insbesondere die Dreiecksungleichung $d_{xy} + d_{yz} \geq d_{xz}$ für alle $x, y, z \in D \cup F$.

Zunächst schreiben wir eine gegebene Eingabe für das Facility-Location-Problem als ganzzahliges lineares Programm. Dazu führen wir Variablen $x_{ij} \in \{0, 1\}$ und $y_i \in \{0, 1\}$ für alle $i \in F$ und $j \in D$ ein. Dabei soll $y_i = 1$ genau dann gelten, wenn Standort i eröffnet wird, und $x_{ij} = 1$ soll genau dann gelten, wenn Kunde j von Standort i beliefert wird. Mithilfe dieser Variablen können wir die Zielfunktion leicht ausdrücken. Wir müssen nur noch Nebenbedingungen zu dem ganzzahligen linearen Programm hinzufügen, die uns garantieren, dass jeder Kunde von einem Standort beliefert wird und zwar nur von einem, der auch geöffnet wurde. Insgesamt ergibt sich das folgende ganzzahlige lineare Programm:

$$\begin{aligned}
\text{minimiere } & \sum_{i \in F} f_i y_i + \sum_{i \in F} \sum_{j \in D} d_{ij} x_{ij} & \\
\text{sodass } & \sum_{i \in F} x_{ij} \geq 1, & \forall j \in D, \\
& x_{ij} \leq y_i, & \forall i \in F,\ \forall j \in D, \\
& y_i \in \{0, 1\}, & \forall i \in F, \\
& x_{ij} \in \{0, 1\}, & \forall i \in F,\ \forall j \in D.
\end{aligned}$$

Die Nebenbedingung $\sum_{i \in F} x_{ij} \geq 1$ garantiert, dass Kunde j von mindestens einem Standort beliefert wird. Da die Zielfunktion minimiert wird, ist sichergestellt, dass in der optimalen Lösung sogar $\sum_{i \in F} x_{ij} = 1$ gilt. Die Nebenbedingung $y_i \geq x_{ij}$ stellt sicher, dass Kunde j nur dann von Standort i beliefert werden kann, wenn dieser auch eröffnet wurde.

Wir betrachten wieder die Relaxierung des ganzzahligen Programms, in der wir nur noch $x_{ij} \in [0,1]$ und $y_i \in [0,1]$ fordern. Dann erhalten wir ein lineares Programm, welches wir in polynomieller Zeit optimal lösen können. Sei (x^*, y^*) eine optimale Lösung dieses linearen Programms. Wir werden diese Lösung wieder durch eine geeignete Rundung ganzzahlig machen. Im Gegensatz zu dem randomisierten Runden bei MAX-SAT ist das Rundungsverfahren, das wir hier einsetzen werden, deterministisch.

Wir bezeichnen mit $\mathrm{OPT}_{\mathrm{LP}} = \sum_{i \in F} f_i y_i^* + \sum_{i \in F} \sum_{j \in D} d_{ij} x_{ij}^*$ die Kosten der optimalen Lösung des linearen Programms. Ferner bezeichnen wir mit $F^* = \sum_{i \in F} f_i y_i^*$ die Kosten für das Eröffnen der Standorte und mit $D^* = \sum_{i \in F} \sum_{j \in D} d_{ij} x_{ij}^*$ die Kosten für das Beliefern der Kunden in der optimalen Lösung des linearen Programms. Wir wandeln die Lösung (x^*, y^*) in zwei Schritten in eine ganzzahlige Lösung um. Diese Schritte nennen wir *Filtern* und *Runden*.

Filtern: Für einen Kunden $j \in D$ seien $D_j^* = \sum_{i \in F} d_{ij} x_{ij}^*$ die Gesamtkosten, um diesen Kunden zu beliefern. Wir definieren die Nachbarschaft N_j von Kunde j als

$$N_j = \{i \in F \mid d_{ij} \leq 2D_j^* \text{ und } x_{ij}^* > 0\}.$$

Es gilt $\sum_{i \in N_j} x_{ij}^* \geq \frac{1}{2}$: Angenommen dies ist nicht der Fall. Dann folgt aus $\sum_{i \in F} x_{ij}^* \geq 1$ direkt $\sum_{i \in F \setminus N_j} x_{ij}^* > \frac{1}{2}$. Dies impliziert den Widerspruch

$$D_j^* = \sum_{i \in F} d_{ij} x_{ij}^* \geq \sum_{i \in F \setminus N_j} d_{ij} x_{ij}^* > 2D_j^* \sum_{i \in F \setminus N_j} x_{ij}^* > D_j^*.$$

Wir transformieren die optimale Lösung (x^*, y^*) nun in eine Lösung (x', y'). Dazu setzen wir zunächst $x' = x^*$ und $y' = y^*$ und nehmen dann Schritt für Schritt Veränderungen an x' und y' vor. Als erstes setzen wir $x'_{ij} = 0$ für alle $j \in D$ und $i \notin N_j$. Dadurch werden im Allgemeinen die Nebenbedingungen $\sum_{i \in F} x'_{ij} \geq 1$ verletzt. Um diese wieder herzustellen, multiplizieren wir x'_{ij} für alle $j \in D$ und $i \in N_j$ mit dem Faktor 2. Die obige Beobachtung $\sum_{i \in N_j} x_{ij}^* \geq \frac{1}{2}$ garantiert, dass dadurch die Bedingungen $\sum_{i \in F} x'_{ij} \geq 1$ wiederhergestellt werden. Nun kann es passieren, dass Bedingungen der Form $y'_i \geq x'_{ij}$ verletzt sind. Auch dies reparieren wir dadurch, dass wir alle y'_i mit 2 multiplizieren.

Durch die oben beschriebene Transformation erhalten wir eine Lösung (x', y') des linearen Programms, die die folgenden Eigenschaften erfüllt:

$$x'_{ij} > 0 \quad \Longrightarrow \quad d_{ij} \leq 2D_j^* \tag{5.6}$$

und

$$y'_i = 2y_i^*. \tag{5.7}$$

Runden: Nun wird die Lösung (x', y') in eine ganzzahlige Lösung (x'', y'') transformiert. Dazu sei $j \in D$ ein Kunde, für den die Verbindungskosten D_j^* minimal sind. Ferner sei $k = \arg\min_{i \in N_j} f_i$ der günstigste Standort in der Nachbarschaft von Kunde j. Wir setzen $y''_k = 1$ und $y''_i = 0$ für alle $i \in N_j \setminus \{k\}$. Das heißt, Standort k wird eröffnet und alle anderen Standorte aus der Nachbarschaft von Kunde j bleiben geschlossen. Wir setzen außerdem $x''_{kj} = 1$ und $x''_{ij} = 0$ für alle $i \neq k$. Das heißt, Kunde j wird in der ganzzahligen Lösung von Standort k beliefert.

Wir betrachten nun noch zusätzlich die *erweiterte Nachbarschaft* $N_j^{\text{erw}} \subseteq D$ von Kunde j. Diese enthält alle Kunden, die in der Lösung (x', y') zumindest teilweise von einem Standort aus N_j bedient werden:

$$N_j^{\text{erw}} = \{\ell \in D \mid x'_{i\ell} > 0 \text{ für ein } i \in N_j\}.$$

In der ganzzahligen Lösung (x'', y'') weisen wir alle Kunden aus der erweiterten Nachbarschaft N_j^{erw} Standort k zu. Wir setzen also $x''_{k\ell} = 1$ und $x''_{i\ell} = 0$ für alle $\ell \in N_j^{\text{erw}}$ und alle $i \neq k$.

Solange es noch einen Kunden gibt, der keinem Standort zugewiesen wurde, fahren wir nach diesem Schema fort. Das heißt, wir wählen einen Kunden $j \in D$, der unter allen noch nicht zugewiesenen Kunden minimale Verbindungskosten D_j^* besitzt. Für diesen Kunden wählen wir den günstigsten Standort k in der Nachbarschaft, öffnen diesen und weisen j sowie alle Kunden in der erweiterten Nachbarschaft von j, die noch nicht zugewiesen sind, Standort k zu.

Aus dem folgenden Theorem folgt, dass die ganzzahlige Lösung (x'', y'') eine 6-Approximation ist. Da diese Lösung effizient gefunden werden kann, handelt es sich bei dem oben beschriebenen Verfahren um einen 6-Approximationsalgorithmus für Facility Location.

Theorem 5.6. *Die Gesamtkosten der ganzzahligen Lösung (x'', y'') sind höchstens 6-mal so groß wie die Kosten OPT_{LP} der optimalen Lösung (x^*, y^*) des linearen Programms.*

Beweis. Zunächst beobachten wir, dass jeder Standort in dem obigen Rundungsverfahren nur höchstens einmal betrachtet wird. Insbesondere wird die Entscheidung, ob ein Standort eröffnet wird oder geschlossen bleibt, niemals revidiert. Betrachten wir dafür einen Schritt des Rundungsverfahrens, in dem Kunde $j \in D$ ausgewählt wurde, da er unter allen noch nicht zugewiesenen Kunden minimale Verbindungskosten D_j^* besitzt. In diesem Schritt wird für alle Standorte aus der Nachbarschaft N_j die Entscheidung getroffen, ob sie geöffnet werden oder geschlossen bleiben. Außerdem werden in diesem Schritt alle Kunden aus der erweiterten Nachbarschaft N_j^{erw} einem Standort zugewiesen. Für alle anderen Kunden $\ell \in D \setminus N_j^{\text{erw}}$ gilt $x'_{i\ell} = 0$ für alle Standorte $i \in N_j$. Das bedeutet, die Standorte aus N_j spielen bei der Betrachtung dieser Kunden keine Rolle mehr.

Wir schätzen nun die Kosten für das Eröffnen der Standorte in der gerundeten Lösung (x'', y'') ab. Bei der Betrachtung eines Kunden $j \in D$ eröffnen wir den Standort $k = \arg\min_{i \in N_j} f_i$. Die Kosten dafür betragen $f_k \leq \sum_{i \in N_j} f_i y'_i$, wobei wir ausgenutzt haben, dass $f_k \leq f_i$ für alle $i \in N_j$ gilt und dass $\sum_{i \in N_j} y'_i \geq \sum_{i \in N_j} x'_{ij} \geq 1$ gilt. Da Standorte nicht mehrfach betrachtet werden, folgt daraus, dass die Kosten für das Eröffnen der Standorte in der Lösung (x'', y'') durch $\sum_{i \in F} f_i y'_i$ nach oben beschränkt sind.

Zuletzt müssen wir noch abschätzen, wie hoch die Verbindungskosten sind. Betrachten wir wieder einen Schritt, in dem ein Kunde $j \in D$ mit minimalen Verbindungskosten D_j^* ausgewählt wird. Dieser wird einem Standort k in seiner Nachbarschaft zugewiesen. Aus der Definition der Nachbarschaft folgt, dass $d_{kj} \leq 2D_j^*$ gilt. Alle Kunden

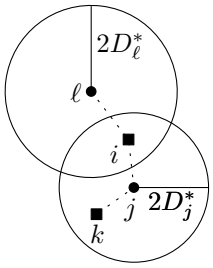


Abbildung 5.2: $d_{k\ell} \leq d_{kj} + d_{ij} + d_{i\ell}$

aus der erweiterten Nachbarschaft von j werden ebenfalls dem Standort k zugewiesen. Sei $\ell \in N_j^{\text{erw}}$ ein solcher Kunde. Dann gibt es einen Standort $i \in N_j$ mit $x'_{i\ell} > 0$ (siehe auch Abbildung 5.2). Wegen (5.6) folgt daraus $d_{i\ell} \leq 2D^*_\ell$. Da die Standorte i und k beide in der Nachbarschaft von Kunde j liegen, beträgt ihr Abstand d_{ik} wegen der Dreiecksungleichung und (5.6) höchstens $d_{ij} + d_{kj} \leq 4D^*_j$. Kunde $j \in D$ besitzt gemäß seiner Wahl unter allen noch nicht zugewiesenen Kunden minimale Verbindungskosten D^*_j. Es gilt also insbesondere $D^*_j \leq D^*_\ell$. Insgesamt beträgt der Abstand zwischen Kunde ℓ und dem Standort k, dem er zugewiesen wird, somit höchstens

$$d_{i\ell} + d_{ik} \leq 2D^*_\ell + 4D^*_j \leq 2D^*_\ell + 4D^*_\ell = 6D^*_\ell.$$

Die Gesamtkosten der Lösung (x'', y'') betragen somit höchstens

$$\sum_{i \in F} f_i y'_i + \sum_{j \in D} 6D^*_j \leq 2 \sum_{i \in F} f_i y^*_i + 6D^* = 2F^* + 6D^* \leq 6(F^* + D^*) = 6 \cdot \text{OPT}_{\text{LP}}. \quad \square$$

In obigem Beweis fällt auf, dass in der gerundeten Lösung (x'', y'') das Eröffnen der Standorte nur höchstens doppelt so viel kostet wie in der optimalen Lösung des linearen Programms, während das Beliefern der Kunden bis zu sechsmal so teuer sein kann. Durch eine geschicktere Wahl der Nachbarschaft, können diese beiden Beiträge balanciert werden, was insgesamt zu einem besseren Approximationsfaktor führt. Sei $\alpha > 1$. Wir definieren dann die α-Nachbarschaft eines Kunden j als

$$N_j^{(\alpha)} = \{i \in F \mid d_{ij} \leq \alpha D^*_j \text{ und } x^*_{ij} > 0\}.$$

Dann gilt $N_j = N_j^{(2)}$. Das Filtern und Runden geschieht nun genauso wie oben beschrieben, nur eben für die Nachbarschaft $N_j^{(\alpha)}$ statt für die Nachbarschaft N_j. Mit analogen Argumenten zeigt man, dass $\sum_{i \in N_j^{(\alpha)}} x^*_{ij} \geq 1 - \frac{1}{\alpha}$ gilt. Statt mit dem Faktor 2 müssen dann beim Filtern alle x'_{ij} mit $j \in D$ und $i \in N_j^{(\alpha)}$ sowie alle y'_i mit $i \in F$ mit dem Faktor $1/(1 - \frac{1}{\alpha}) = \frac{\alpha}{\alpha - 1}$ multipliziert werden.

Dann kann auch der Beweis von Theorem 5.6 entsprechend übertragen werden. Der Leser überlege sich im Detail, dass die Anpassung zu dem Ergebnis führt, dass die Kosten der gerundeten Lösung (x'', y'') durch den folgenden Term nach oben beschränkt sind:

$$\sum_{i \in F} f_i y'_i + \sum_{j \in D} 3\alpha D^*_j \leq \frac{\alpha}{\alpha - 1} \sum_{i \in F} f_i y^*_i + 3\alpha D^* \leq \max\left\{\frac{\alpha}{\alpha - 1}, 3\alpha\right\} \cdot \text{OPT}_{\text{LP}}.$$

Für $\alpha = \frac{4}{3}$ erhält man auf diese Weise einen 4-Approximationsalgorithmus.

5.3 Scheduling auf allgemeinen Maschinen

Wir haben in dieser Vorlesung für das Problem Scheduling auf identischen Maschinen bereits ein PTAS entworfen. Wir betrachten nun eine Verallgemeinerung dieses Problems, in der Jobs auf verschiedenen Maschinen unterschiedliche Größen besitzen können.

Scheduling auf allgemeinen Maschinen

Eingabe:	Menge $J = \{1, \dots, n\}$ von Jobs, Menge $M = \{1, \dots, m\}$ von Maschinen, Jobgröße $p_{ij} \in \mathbb{R}_{>0}$ für alle $i \in M$ und $j \in J$
Lösungen:	alle Abbildungen $\pi : J \to M$ (eine solche Abbildung nennen wir Schedule)
Zielfunktion:	Es sei $L_i(\pi) = \sum_{j \in J : \pi(j) = i} p_{ij}$ die Ausführungszeit von Maschine i in Schedule π. minimiere den Makespan $C(\pi) = \max_{i \in M} L_i(\pi)$

Der einzige Unterschied zum Scheduling auf identischen Maschinen besteht beim Scheduling auf allgemeinen Maschinen darin, dass ein Job $j \in J$ nicht eine feste Größe p_j besitzt, sondern auf jeder Maschine $i \in M$ eine individuelle Größe p_{ij}. Damit können beispielsweise unterschiedlich schnelle Maschinen modelliert werden. Es ist bekannt, dass es unter der Annahme P≠NP für Scheduling auf allgemeinen Maschinen keinen Approximationsalgorithmus gibt, der einen besseren Approximationsfaktor als 3/2 erreicht. Der beste bekannte Approximationsalgorithmus ist ein 2-Approximationsalgorithmus, den wir in diesem Abschnitt besprechen werden.

Wir können eine Instanz des Problems auf naheliegende Weise als ganzzahliges lineares Programm formulieren. Dazu benötigen wir lediglich binäre Variablen $x_{ij} \in \{0, 1\}$, die angeben, ob Job j Maschine i zugewiesen wird, sowie eine reelle Variable t, die den Makespan des Schedules beschreibt. Wir erhalten dann das folgende ganzzahlige lineare Programm:

$$
\begin{aligned}
&\text{minimiere } t \\
&\quad \text{sodass } \sum_{i \in M} x_{ij} = 1, && \forall j \in J, \\
&\qquad \sum_{j \in J} p_{ij} x_{ij} \leq t, && \forall i \in M, \\
&\qquad x_{ij} \in \{0, 1\}, && \forall i \in M,\ \forall j \in J.
\end{aligned}
$$

Versuchen wir mithilfe dieses ganzzahligen linearen Programms einen Approximationsalgorithmus zu entwerfen, indem wir das Programm zunächst relaxieren und dann die optimale Lösung des relaxierten Programms runden, so stellen wir schnell fest, dass dies direkt nicht zum Erfolg führen wird, da das Integrality Gap sehr groß ist.

Beispiel für großes Integrality Gap

Wir betrachten eine Instanz mit einem einzigen Job und m Maschinen. Der Job habe auf allen Maschinen dieselbe Größe 1.

Dann besitzt die optimale ganzzahlige Lösung einen Makespan von 1, da der Job einer beliebigen Maschine zugewiesen werden muss. In der optimalen Lösung des relaxierten linearen Programms wird hingegen jeder Maschine ein $(1/m)$-Bruchteil des Jobs zugewiesen (d. h. $x_{i1} = 1/m$ für alle $i \in M$), sodass die optimale Lösung des relaxierten linearen Programms den Wert $1/m$ besitzt. Damit beträgt das Integrality Gap für diese Instanz m.

Das obige Beispiel zeigt, dass man keinen besseren Approximationsfaktor als m zeigen kann, wenn man nur den Wert der optimalen Lösung des relaxierten linearen Programms als untere Schranke für den optimalen Makespan nutzt. Wir haben bereits beim Scheduling auf identischen Maschinen die Beobachtung ausgenutzt, dass der optimale Makespan nicht kleiner sein kann als der größte Job. Wir könnten dem relaxierten linearen Programm demnach für jeden Job $j \in J$ und jede Maschine $i \in M$ die Nebenbedingung hinzufügen, dass $x_{ij} = 0$ gelten muss, falls $p_{ij} > t$ gilt. Diese Bedingung ist im ganzzahligen Programm automatisch erfüllt und sie würde dazu führen, dass zumindest in dem obigen Beispiel das Integrality Gap von m auf 1 schrumpft. Das Problem ist allerdings, dass sich diese zusätzlichen Nebenbedingungen nicht als lineare Nebenbedingungen formulieren lassen und somit nicht einfach zu dem relaxierten linearen Programm hinzugefügt werden können.

Stattdessen verfolgen wir einen anderen Ansatz und gehen zunächst davon aus, dass wir als Eingabe bereits einen Makespan T erhalten. Wir testen dann, ob das relaxierte lineare Programm eine Lösung mit Wert höchstens T besitzt, wobei wir die oben diskutierten zusätzlichen Nebenbedingungen explizit sicher stellen. Sei dazu $S_T = \{(i,j) \in M \times J \mid p_{ij} \leq T\}$. Wir betrachten nun das folgende lineare Programm LP_T ohne Zielfunktion:

$$\sum_{i \in M} x_{ij} = 1, \qquad \forall j \in J, \tag{5.8}$$

$$\sum_{j \in J} p_{ij} x_{ij} \leq T, \qquad \forall i \in M, \tag{5.9}$$

$$x_{ij} = 0, \qquad \forall (i,j) \notin S_T, \tag{5.10}$$

$$x_{ij} \geq 0, \qquad \forall (i,j) \in S_T. \tag{5.11}$$

Wir haben bei dem linearen Programm keine Zielfunktion angegeben, da wir lediglich testen möchten, ob es überhaupt eine zulässige Lösung besitzt. Dies ist, genauso wie das Optimieren einer linearen Zielfunktion, in polynomieller Zeit möglich.

Besitzt LP_T keine Lösung, so besitzt LP_T insbesondere keine ganzzahlige Lösung. Das bedeutet, dass der Makespan OPT der optimalen Lösung größer als T sein muss. Mithilfe einer binären Suche kann in polynomieller Zeit der Wert

$$T^* = \min\{T \mid \mathrm{LP}_T \text{ besitzt eine Lösung}\}$$

bestimmt werden. Für diesen gilt $T^* \leq \mathrm{OPT}$, da $T^* - \varepsilon < \mathrm{OPT}$ für jedes $\varepsilon > 0$ gilt. Wir zeigen nun, wie aus einer zulässigen Lösung für LP_{T^*} eine ganzzahlige Lösung mit Makespan höchstens $2T^* \leq 2\mathrm{OPT}$ konstruiert werden kann.

Im Gegensatz zu den Algorithmen, die wir bislang in diesem Kapitel besprochen haben, müssen wir uns hier genauer mit der Struktur von Lösungen des linearen Programms

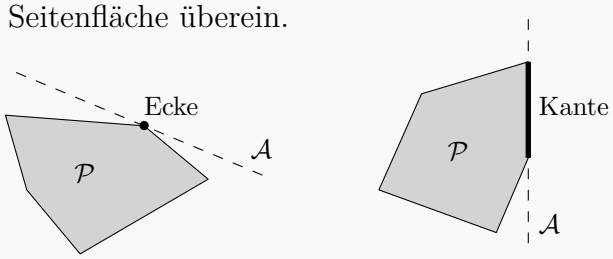
auseinandersetzen und können diese nicht als Blackbox einsetzen. Insbesondere benötigen wir den Begriff einer *zulässigen Basislösung*, den wir in der Vorlesung „Algorithmen und Berechnungskomplexität I“ eingeführt habe. Um diesen Begriff zu wiederholen, rufen wir uns die geometrische Interpretation eines linearen Programms in Erinnerung.

In einem linearen Programm mit d Variablen entspricht jede Variablenbelegung einem Punkt im $\mathbb{R}^d$. Jede lineare Nebenbedingung definiert einen Halbraum im $\mathbb{R}^d$ und der Schnitt all dieser Halbräume entspricht der Menge der zulässigen Lösungen für das lineare Programm. Einen solchen Schnitt nennt man konvexes Polyeder, und das zu einem linearen Programm gehörende konvexe Polyeder bezeichnen wir auch als sein *Lösungspolyeder*.

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 $\mathcal{A}$ gehörenden Halbraum bezeichnet, so sagen wir, dass $\mathcal{A}$ eine *stützende Hyperebene* von $\mathcal{P}$ ist. Für jede stützende Hyperebene $\mathcal{A}$ nennen wird die Menge $\mathcal{P} \cap \mathcal{A}$ *Fläche von* $\mathcal{P}$. Eine nulldimensionale Fläche heißt *Ecke von* $\mathcal{P}$, eine eindimensionale Fläche heißt *Kante von* $\mathcal{P}$ und eine Fläche der Dimension $d-1$ heißt *Seitenfläche von* $\mathcal{P}$.

Beispiel

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



Die Ecken des Lösungspolyeders entsprechen den zulässigen Basislösungen (die äquivalente algebraische Definition von zulässigen Basislösungen benötigen wir an dieser Stelle nicht). Geometrisch wird eine Ecke durch den Schnitt von d linear unabhängigen Hyperebenen definiert. Dies bedeutet, dass in einer Ecke mindestens d der Nebenbedingungen mit Gleichheit erfüllt sind. In „Algorithmen und Berechnungskomplexität I“ haben wir außerdem bewiesen, dass eine Lösung genau dann eine zulässige Basislösung ist, wenn sie ein *Extrempunkt* des Lösungspolyeders ist. Dabei heißt ein Punkt $x \in \mathcal{P}$ Extrempunkt, wenn es kein $y \in \mathbb{R}^d$ mit $y \neq 0$ gibt, für das $x + y \in \mathcal{P}$ und $x - y \in \mathcal{P}$ gilt.

Des Weiteren haben wir gesehen, dass es zu jedem zulässigen und beschränkten linearen Programm (d. h. zu jedem linearen Programm, dessen Lösungspolyeder nicht leer ist und in dem der Wert der Zielfunktion nicht beliebig gut wird) eine zulässige Basislösung gibt, die optimal ist, wenn alle Variablen nur Werte größer oder gleich Null annehmen können. Eine solche kann in polynomieller Zeit gefunden werden. Wir können also insbesondere in polynomieller Zeit eine zulässige Basislösung von LP_{T^*} berechnen.

Lemma 5.7. *In einer zulässigen Basislösung von* LP_{T^*} *besitzen höchstens* $n + m$ *Variablen einen Wert echt größer als Null.*

Beweis. Die Anzahl an Variablen in LP_{T^*} beträgt nm. Das bedeutet, in einer zulässigen Basislösung müssen mindestens nm Nebenbedingungen mit Gleichheit erfüllt sein. Es gibt n Bedingungen vom Typ (5.8) und m Bedingungen vom Typ (5.9). Selbst wenn diese alle mit Gleichheit erfüllt sind, müssen noch mindestens $nm-(n+m)$ viele Bedingungen vom Typ (5.10) oder (5.11) mit Gleichheit erfüllt sein, d. h. die entsprechenden Variablen x_{ij} müssen auf Null gesetzt sein. Somit besitzen höchstens $n + m$ der Variablen x_{ij} einen Wert ungleich Null. □

Sei nun x eine zulässige Basislösung von LP_{T^*}. Zu dieser Lösung definieren wir den *Allokationsgraphen* G. Dabei handelt es sich um einen bipartiten Graphen mit Knotenmenge $M \cup J$ und Kantenmenge $E = \{(i, j) \in M \times J \mid x_{ij} > 0\}$. Eine Kante (i, j) in dem Allokationsgraphen besagt also, dass Job j in der zulässigen Basislösung x zumindest teilweise Maschine i zugewiesen wird. Wir nennen einen Graphen mit Knotenmenge V einen *Pseudobaum*, falls dieser zusammenhängend ist und höchstens $|V|$ Kanten enthält. Da jeder Spannbaum eines solchen Graphen genau $|V|-1$ Kanten enthält, handelt es sich bei einem Pseudobaum entweder um einen Spannbaum oder um einen Spannbaum mit einer zusätzlichen Kante, die einen Kreis schließt. Das folgende Korollar folgt direkt aus Lemma 5.7.

Korollar 5.8. *Falls der Allokationsgraph* G *zusammenhängend ist, so ist er ein Pseudobaum.*

Wir nennen einen Graphen einen *Pseudowald*, wenn jede seiner Zusammenhangskomponenten ein Pseudobaum ist.

Lemma 5.9. *Der Allokationsgraph* G *ist ein Pseudowald.*

Beweis. Es sei $G' = (V', E')$ eine beliebige Zusammenhangskomponente von G mit $V' = M' \cup J'$ für $M' \subseteq M$ und $J' \subseteq J$. Wir betrachten nun ein eingeschränktes Schedulingproblem, bei dem nur noch die Jobs aus J' und die Maschinen aus M' vorhanden sind. Wir bezeichnen mit LP'_{T^*} das zu LP_{T^*} analoge lineare Programm, das sich für die eingeschränkte Instanz ergibt. Außerdem bezeichnen wir mit x' die Einschränkung der Lösung x auf die Variablen, die in LP'_{T^*} noch auftreten. Dann ist x' eine zulässige Lösung von LP'_{T^*}.

Wir zeigen nun, dass es sich bei x' sogar um eine zulässige Basislösung von LP'_{T^*} handelt. Angenommen, dies ist nicht der Fall. Dann ist x' insbesondere kein Extrempunkt und somit gibt es einen vom Nullvektor verschiedenen Vektor y', sodass $x' + y'$ und $x' - y'$ zulässige Lösungen von LP'_{T^*} sind. Wir erweitern den Vektor y' zu einem Vektor $y \in \mathbb{R}^{nm}$, indem wir $y_{ij} = 0$ für alle $i \notin M'$ oder $j \notin J'$ setzen. Dann sind $x + y$ und $x - y$ zulässige Lösungen von LP_{T^*} und somit ist x kein Extrempunkt und damit auch keine zulässige Basislösung von LP_{T^*} im Widerspruch zu seiner Definition.

Da es sich bei x' um eine zulässige Basislösung von LP'_{T^*} handelt und der Graph G' zusammenhängend ist, ist er laut Korollar 5.8 ein Pseudobaum. □

Mithilfe des Allokationsgraphen runden wir die Lösung x nun in eine ganzzahlige Lösung $\hat{x}$. Für einen Job $j \in J$, der in x bereits vollständig einer Maschine $i \in M$ zugewiesen ist (für den also $x_{ij} = 1$ gilt), übernehmen wir diese Zuweisung in $\hat{x}$. Solange es noch einen solchen Job gibt, entfernen wir ihn und die entsprechende Kante aus dem Allokationsgraphen. Den Graphen, der danach übrigbleibt, nennen wir H. Die Knotenmenge von H ist $J' \cup M$, wobei $J' \subseteq J$ die Menge der Jobs bezeichnet, die in der Lösung x auf mehr als eine Maschine verteilt werden.

In dem Graphen H berechnen wir ein einseitig perfektes Matching, d. h. eine Kantenmenge N, sodass jeder Job aus J' genau eine inzidente Kante in N und jede Maschine aus M höchstens eine inzidente Kante in N besitzt.

Lemma 5.10. *Der Graph H besitzt ein einseitig perfektes Matching, welches in polynomieller Zeit gefunden werden kann.*

Beweis. Zunächst entfernen wir aus H alle isolierten Maschinenknoten. Im Folgenden nutzen wir aus, dass alle Blätter in dem Pseudowald H Maschinenknoten sind. Dies liegt daran, dass wir Jobknoten mit Grad 1 bei der Erstellung von H entfernt haben (diese entsprechen genau den Jobs, die bereits in x komplett einer Maschine zugewiesen sind). Solange es in dem Graphen H noch ein Blatt gibt, führen wir die folgende Operation aus: Wähle ein beliebiges Blatt $i \in M$ aus H aus. Dieses besitzt genau einen adjazenten Knoten $j \in J$. Weise Job j Maschine i zu (d. h. füge die Kante (i, j) zu N hinzu) und entferne die Knoten i und j und ihre inzidenten Kanten aus dem Graphen H. Lösche anschließend alle isolierten Maschinenknoten, die dadurch entstanden sind.

Werden aus einem Pseudobaum iterativ alle Blätter auf die oben beschriebene Art und Weise entfernt, so bleibt am Ende ein leerer Graph oder ein Kreis übrig. Da es sich bei H um einen bipartiten Pseudowald handelt, verbleibt somit am Ende eine (möglicherweise leere) Menge von Kreisen gerader Länge. Von jedem dieser Kreise nehmen wir jede zweite Kante zu N hinzu. Durch dieses Verfahren wird insgesamt sichergestellt, dass jeder Knoten aus J' genau eine inzidente Kante und jeder Knoten aus M höchstens eine inzidente Kante erhält. □

Das einseitig perfekte Matching bildet zusammen mit den bereits in x festgelegten Jobs die ganzzahlige Lösung $\hat{x}$.

Theorem 5.11. *Der Makespan von $\hat{x}$ beträgt höchstens $2T^*$.*

Beweis. Auf jeder Maschine $i \in M$ verursachen die Zuweisungen in x eine Ausführungszeit von höchstens T^*. Insbesondere beträgt die Gesamtausführungszeit aller Jobs, die in der Lösung x komplett Maschine i zugewiesen sind, höchstens T^*. All diese Jobs erhält Maschine i auch in der Lösung $\hat{x}$. Zusätzlich wird ihr beim Runden der fraktional zugewiesenen Jobs wegen Lemma 5.10 höchstens ein weiterer Job j zugewiesen. Für diesen Job j gilt, sofern er existiert, dass die Kante (i, j) zum Graphen G gehört. Demnach gilt $x_{ij} > 0$ und somit $p_{ij} \leq T^*$. Insgesamt ergibt sich damit für jede Maschine $i \in M$ eine Ausführungszeit von höchstens $2T^*$. □

Kapitel 6

Semidefinite Programmierung und Runden

Wir haben im letzten Kapitel gesehen, dass lineare Programmierung ein wichtiges und vielseitiges Werkzeug zum Entwurf von Approximationsalgorithmen ist. In diesem Kapitel werden wir dieses Werkzeug erweitern und sehen, dass für einige Probleme bessere Approximationsgarantien erreicht werden können, wenn man statt auf lineare Programme auf Programme mit bestimmten nichtlinearen Zielfunktionen und Nebenbedingungen zurückgreift.

Wir betrachten zunächst noch einmal das Maximum-Cut Problem, für das wir bereits einen einfachen randomisierten 2-Approximationsalgorithmus kennengelernt haben. Die Eingabe bestehe aus einem ungerichteten Graphen $G = (V, E)$ mit Knotenmenge $V = \{v_1, \ldots, v_n\}$ und Kantengewichten $w : E \to \mathbb{R}_{>0}$ (wir schreiben im Folgenden w_{ij} für $w(v_i, v_j)$ und setzen $w_{ij} = 0$ für $(v_i, v_j) \notin E$). Führt man für jeden Knoten v_i eine Variable x_i ein, so entspricht die gegebene Instanz des Maximum-Cut Problems dem folgenden Programm:

$$\begin{aligned} \text{maximiere } & \frac{1}{2}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij}(1 - x_i x_j) & \\ \text{sodass } & x_i^2 = 1, & \forall i \in \{1, \ldots, n\}, \\ & x_i \in \mathbb{Z}, & \forall i \in \{1, \ldots, n\}. \end{aligned}$$

In diesem Programm sind sowohl die Zielfunktion als auch die Nebenbedingungen nicht linear, sondern quadratisch in den Variablen. Die Nebenbedingungen stellen sicher, dass die Variablen x_i nur Werte aus der Menge $\{1, -1\}$ annehmen. Andersherum ist jede Belegung der Variablen mit Werten aus dieser Menge eine zulässige Lösung für das Programm. Das Ziel beim Maximum-Cut Problem ist es, eine Partition der Knotenmenge V in zwei Seiten U und $V \setminus U$ zu berechnen, sodass das Gesamtgewicht der Kanten, die über den Schnitt laufen, maximal wird. Die Variable x_i codiert, ob der Knoten v_i zu der Menge U gehört. Wir interpretieren $x_i = 1$ als $v_i \in U$ und $x_i = -1$ als $v_i \in V \setminus U$. Gehören v_i und v_j zu derselben Seite der Partition, so gilt $x_i = x_j$ und damit $(1 - x_i x_j) = 0$. Gehören v_i und v_j zu verschiedenen Seiten der Partition, so gilt $x_i \neq x_j$ und damit $(1 - x_i x_j) = 2$. Daraus folgt, dass die Zielfunktion genau dem Gesamtgewicht der Kanten entspricht, die über den Schnitt laufen.

Anstatt einfach nur die Ganzzahligkeitsbedingung aufzugeben, betrachten wir eine andere Art von Relaxierung. Wir ersetzen jede Variable x_i durch einen Vektor $z_i \in \mathbb{R}^n$ und das Produkt zweier Variablen durch das Skalarprodukt der entsprechenden Vektoren. Wir erhalten dann das folgende *Vektorprogramm*:

$$\begin{aligned} \text{maximiere } & \frac{1}{2}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij}(1 - z_i \cdot z_j) & \\ \text{sodass } & z_i \cdot z_i = 1, & \forall i \in \{1,\dots,n\}, \\ & z_i \in \mathbb{R}^n, & \forall i \in \{1,\dots,n\}. \end{aligned}$$

Dieses Vektorprogramm kann als Relaxierung des obigen ganzzahligen quadratischen Programms aufgefasst werden, denn eine Lösung $x_1,\dots,x_n \in \{1,-1\}$ kann in dem Vektorprogramm nachgebildet werden, indem $z_i = x_i e$ für jedes i gesetzt wird, wobei $e \in \mathbb{R}^n$ einen beliebigen Einheitsvektor bezeichnet. Dann stimmen die Zielfunktionswerte der Lösungen $(x_1,\dots,x_n)$ und $(z_1,\dots,z_n)$ überein.

Allgemein besitzt ein Vektorprogramm n Variablen $z_i \in \mathbb{R}^n$. Das heißt, jede der n Variablen ist selbst ein n-dimensionaler reeller Vektor. Sowohl die Zielfunktion als auch die Nebenbedingungen sind linear in den Skalarprodukten dieser Vektoren. Ein Vektorprogramm mit m Nebenbedingungen lässt sich also allgemein wie folgt schreiben:

$$\begin{aligned} \text{max/min } & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}(z_i \cdot z_j) & \\ \text{sodass } & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}(z_i \cdot z_j) = b_k, & \forall k \in \{1,\dots,m\}, \\ & z_i \in \mathbb{R}^n, & \forall i \in \{1,\dots,n\}. \end{aligned} \tag{6.1}$$

Genau wie bei linearen Programmen können wir statt Gleichungen auch Ungleichungen als Nebenbedingungen in einem Vektorprogramm zulassen.

Im nächsten Abschnitt werden wir uns zunächst genauer mit Vektorprogrammen beschäftigen und sehen, dass ein Vektorprogramm in polynomieller Zeit (im Wesentlichen) optimal gelöst werden kann. In den anschließenden Abschnitten werden wir für das Maximum-Cut Problem und einige weitere Probleme Approximationsalgorithmen kennenlernen, die darauf basieren, die optimale Lösung eines Vektorprogramms geeignet zu runden.

6.1 Semidefinite Programmierung

Um uns mit Vektorprogrammen im Detail beschäftigen zu können, benötigen wir zunächst einige Begriffe und Ergebnisse aus der linearen Algebra. Für eine Matrix $M \in \mathbb{R}^{m\times n}$ bezeichnen wir mit $M^{\mathrm{T}} \in \mathbb{R}^{n\times m}$ die zugehörige transponierte Matrix. Die Zeilen der Matrix M entsprechen also den Spalten der Matrix M^{T}. Eine Matrix $M \in \mathbb{R}^{n\times n}$ heißt *symmetrisch*, wenn $M = M^{\mathrm{T}}$ gilt. Alle Vektoren $v \in \mathbb{R}^n$ fassen wir im Folgenden als Spaltenvektoren auf, d. h. $v^{\mathrm{T}}v \in \mathbb{R}$ entspricht dem Skalarprodukt des Vektors v mit sich selbst und $vv^{\mathrm{T}} \in \mathbb{R}^{n\times n}$ ist eine $(n \times n)$-Matrix.

Definition 6.1. *Eine Matrix $M \in \mathbb{R}^{n\times n}$ heißt* positiv semidefinit*, wenn $x^{\mathrm{T}}Mx \geq 0$ für alle $x \in \mathbb{R}^n$ gilt. Wir schreiben dann $M \succeq 0$.*

Das folgende Lemma kann mit elementaren Methoden der linearen Algebra gezeigt werden.

Lemma 6.2. *Für eine symmetrische Matrix $M \in \mathbb{R}^{n\times n}$ sind die folgenden Aussagen äquivalent.*

1. *M ist positiv semidefinit.*
2. *M hat keine negativen Eigenwerte.*
3. *Es gilt $M = V^{\mathrm{T}}V$ für eine Matrix $V \in \mathbb{R}^{n\times n}$.*

Wir lernen nun mit *semidefiniten Programmen* noch eine weitere Art von nichtlinearen Programmen kennen. Die Variablen eines solchen Programms bezeichnen wir mit x_{ij} für $i, j \in \{1, \ldots, n\}$ und wir fordern $x_{ij} = x_{ji}$. Die Zielfunktion und die Nebenbedingungen sind linear in diesen Variablen. Der einzige Unterschied zu einem linearen Programm ist eine zusätzliche Nebenbedingung, die nur solche Variablenbelegungen zulässt, für die die Matrix $X = (x_{ij})$ positiv semidefinit ist. Ein semidefinites Programm mit m Nebenbedingungen besitzt demnach die folgende Form:

$$\begin{aligned} \max/\min \ & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & & \\ \text{sodass} \ & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}x_{ij} = b_k, & \forall k \in \{1,\ldots,m\}, & \\ & x_{ij} = x_{ji}, & \forall i,j \in \{1,\ldots,n\}, & \\ & X = (x_{ij}) \succeq 0. & & \end{aligned} \tag{6.2}$$

Auch bei semidefiniten Programmen können statt Gleichungen Ungleichungen als Nebenbedingungen zugelassen werden.

Lemma 6.3. *Das Vektorprogramm* (6.1) *und das semidefinite Programm* (6.2) *sind in dem Sinne äquivalent, dass eine Lösung für das eine in polynomieller Zeit in eine Lösung für das andere mit demselben Zielfunktionswert überführt werden kann und umgekehrt. Insbesondere können damit optimale Lösungen ineinander überführt werden.*

Beweis. Betrachten wir eine Lösung $(z_1, \ldots, z_n)$ für das Vektorprogramm (6.1). Wir konstruieren eine Matrix $Z \in \mathbb{R}^{n\times n}$, wobei Spalte i durch den Vektor z_i gegeben ist. Es sei $X = (x_{ij}) = Z^{\mathrm{T}}Z \in \mathbb{R}^{n\times n}$. Es gilt dann $x_{ij} = z_i \cdot z_j = x_{ji}$ und somit ist die Matrix X symmetrisch. Mit Lemma 6.2 folgt dann, dass sie positiv semidefinit ist. Die Matrix X ist somit eine zulässige Lösung für das semidefinite Programm (6.2) und sie besitzt denselben Zielfunktionswert wie die Lösung $(z_1, \ldots, z_n)$ im Vektorprogramm.

Nun betrachten wir eine Lösung $X = (x_{ij})$ für das semidefinite Programm. Um hieraus eine Lösung für das Vektorprogramm zu konstruieren, berechnen wir zunächst eine Cholesky-Zerlegung von X, d. h. wir berechnen eine Matrix Z mit $X = Z^{\mathrm{T}}Z$. Streng

genommen, kann eine solche Zerlegung im Allgemeinen nicht in polynomieller Zeit berechnet werden, da die Einträge von Z irrational sein können. Es ist jedoch möglich, die Einträge in polynomieller Zeit mit einem kleinen additiven Fehler zu approximieren. Da dieser Fehler beliebig klein gemacht werden kann, werden wir nicht weiter darauf eingehen und der Einfachheit halber davon ausgehen, dass wir die Zerlegung exakt berechnet haben. Bezeichnen wir mit z_i die i-te Spalte von Z, so ist $(z_1, \ldots, z_n)$ eine zulässige Lösung für das Vektorprogramm mit demselben Zielfunktionswert wie die Lösung X für das semidefinite Programm (es gilt $x_{ij} = z_i \cdot z_j$). □

Semidefinite Programme können im Allgemeinen nicht in polynomieller Zeit optimal gelöst werden, da die optimale Lösung irrational sein kann. Ein weiteres Problem ist, dass es semidefinite Programme gibt, in denen alle Koeffizienten kleine Zahlen sind, der Wert der optimalen Lösung jedoch doppelt exponentiell von der Anzahl der Variablen abhängt. In den für uns relevanten semidefiniten Programmen tritt dieses Problem allerdings nicht auf. Mithilfe der Ellipsoidmethode erhält man das folgende Theorem.

Theorem 6.4. *Die optimale Lösung eines semidefiniten Programms kann für jedes $\varepsilon > 0$ mit einem additiven Fehler von höchstens ε berechnet werden. Die Laufzeit hierfür ist polynomiell in der Eingabegröße und $\log(R/\varepsilon)$ beschränkt, sofern die Menge der zulässigen Lösungen nichtleer und in dem Hyperwürfel $[-R, R]^n$ enthalten ist.*

Mit Lemma 6.3 überträgt sich dieses Theorem auch auf Vektorprogramme.

6.2 Maximum-Cut Problem

Wir kommen nun auf das Maximum-Cut Problem und die oben diskutierte Relaxierung als Vektorprogramm zurück:

$$\begin{aligned} \text{maximiere } & \frac{1}{2}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij}(1 - z_i \cdot z_j) & \\ \text{sodass } & z_i \cdot z_i = 1, & \forall i \in \{1, \ldots, n\}, \\ & z_i \in \mathbb{R}^n, & \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{6.3}$$

Mithilfe von Theorem 6.4 können wir in polynomieller Zeit die optimale Lösung dieser Relaxierung mit einem beliebig kleinen additiven Fehler berechnen. Wir ignorieren diesen additiven Fehler und gehen davon aus, dass wir die optimale Lösung $(z_1, \ldots, z_n)$ berechnet haben. Wir bezeichnen mit W^* den Wert dieser Lösung. Mit der Beobachtung, die wir am Anfang dieses Kapitels gemacht haben, dass es sich bei dem Vektorprogramm um eine Relaxierung des ganzzahligen quadratischen Programms handelt, folgt $W^* \geq \text{OPT}$, wobei OPT den Wert des optimalen Schnittes bezeichnet.

Unser Ziel ist es nun, die optimale Lösung des Vektorprogramms zu einer ganzzahligen Lösung zu runden, sodass sich die Zielfunktion nur um einen konstanten Faktor verschlechtert. Das Rundungsverfahren, das wir einsetzen werden, ist randomisiert, und es

basiert auf der folgenden Beobachtung. Der Beitrag der Kante zwischen den Knoten v_i und v_j zur Zielfunktion in der optimalen Lösung des Vektorprogramms beträgt

$$\frac{w_{ij}}{2}(1 - z_i \cdot z_j) = \frac{w_{ij}}{2}(1 - \|z_i\|\|z_j\| \cos(\theta_{ij})) = \frac{w_{ij}}{2}(1 - \cos(\theta_{ij})),$$

wobei θ_{ij} den Winkel zwischen den Vektoren z_i und z_j bezeichnet. Der Beitrag steigt in dem relevanten Intervall $[0, \pi]$ für θ_{ij} also streng monoton von 0 (für $\theta_{ij} = 0$) auf w_{ij} (für $\theta_{ij} = \pi$). Damit wir durch das Runden nicht allzu viel verlieren, sollte die Wahrscheinlichkeit, dass v_i und v_j in der gerundeten Lösung durch den Schnitt getrennt werden, mit dem Winkel θ_{ij} ansteigen. Je größer der Winkel θ_{ij} ist, desto größer sollte die Wahrscheinlichkeit sein, dass v_i und v_j voneinander getrennt werden.

Das gewünschte Verhalten erhalten wir auf einfache Art und Weise. Die Vektoren z_i liegen aufgrund der Nebenbedingungen $z_i \cdot z_i = 1$ auf der Einheitssphäre $\mathcal{S}^{n-1} = \{x \in \mathbb{R}^n \mid \|x\| = 1\}$. Wir teilen diese Sphäre nun in zwei Hälften ein, indem wir eine uniform zufällige Hyperebene durch den Mittelpunkt von $\mathcal{S}^{n-1}$ (den Koordinatenursprung) legen. Diese Hyperebene partitioniert die Menge der Vektoren z_i und damit auch die Menge der Knoten v_i in zwei Teile. Dieser Algorithmus ist im folgenden Pseudocode noch einmal formal beschrieben.

SDPMaxCut

1 Löse das Vektorprogramm (6.3).
2 Wähle $r \in \mathcal{S}^{n-1}$ uniform zufällig.
3 Setze $U = \{v_i \mid z_i \cdot r \geq 0\}$.
4 **return** $(U, V \setminus U)$;

Wir müssen noch diskutieren, wie ein uniform zufälliger Vektor $r \in \mathcal{S}^{n-1}$ effizient ausgewählt werden kann. Dazu erzeugen wir zunächst n Zahlen $\hat{r}_1, \ldots, \hat{r}_n$ unabhängig zufällig gemäß einer Standardnormalverteilung. Streng genommen ist dies natürlich nicht in endlicher Zeit möglich, da eine standardnormalverteilte Zahl mit Wahrscheinlichkeit 1 irrational ist. Bei einer Implementierung des Algorithmus muss man sich also mit einer Approximation begnügen und die Zahlen $\hat{r}_1, \ldots, \hat{r}_n$ nach einer endlichen Zahl von Dezimalstellen abschneiden. Standardnormalverteilte Zufallsvariablen, die nach einer endlichen Anzahl an Stellen abgeschnitten sind, können effizient erzeugt werden. Wir ignorieren im Folgenden den Einfluss dieser Approximation, da er beliebig klein gemacht werden kann, und gehen der Einfachheit halber davon aus, dass es sich bei $\hat{r}_1, \ldots, \hat{r}_n$ um nicht-abgeschnittene standardnormalverteilte Zufallsvariablen handelt. Es sei dann $\hat{r} = (\hat{r}_1, \ldots, \hat{r}_n)$ und $r = \hat{r}/\|\hat{r}\|$.

Lemma 6.5. *Der Vektor r ist uniform zufällig auf der Sphäre $\mathcal{S}^{n-1}$ verteilt.*

Beweis. Aus der Definition der Standardnormalverteilung und der Tatsache, dass die Zufallsvariablen $\hat{r}_1, \ldots, \hat{r}_n$ unabhängig sind, ergibt sich, dass die Dichte des Vektors $\hat{r}$ an der Stelle $(x_1, \ldots, x_n) \in \mathbb{R}^n$

$$\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x_i^2}{2}\right) = \frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2}\sum_{i=1}^{n} x_i^2\right)$$

beträgt. Diese Dichte hängt nur davon ab, wie weit der Punkt $(x_1, \ldots, x_n)$ vom Ursprung des Koordinatensystems entfernt ist, nicht aber von seiner genauen Position. Demzufolge ist die Dichte von $\hat{r}$ radialsymmetrisch. Der Vektor $r = \hat{r}/\|\hat{r}\|$ hat dieselbe Richtung wie der Vektor $\hat{r}$. Aus der Radialsymmetrie folgt somit das Lemma. □

Eine wichtige Eigenschaft von normalverteilten Zufallsvektoren ist, dass Projektionen auf Unterräume wieder normalverteilt sind. Konkret nutzen wir die folgende Eigenschaft: Seien $e_1 \in \mathbb{R}^n$ und $e_2 \in \mathbb{R}^n$ zwei orthogonale Einheitsvektoren. Dann sind die Projektionen von $\hat{r}$ auf e_1 und e_2 unabhängige standardnormalverteilte Zufallsvariablen. Es bezeichne r' die Projektion von $\hat{r}$ auf die Ebene, die von e_1 und e_2 aufgespannt wird. Da die Projektionen von $\hat{r}$ auf e_1 und e_2 unabhängige standardnormalverteilte Zufallsvariablen sind, folgt mit Lemma 6.5, dass der Vektor $r'/\|r'\|$ uniform zufällig auf der Sphäre $\mathcal{S}^1$ (also auf dem Rand des Einheitskreises) in der durch e_1 und e_2 aufgespannten Ebene verteilt ist. Diese Eigenschaft ist für das folgende Lemma wesentlich.

Lemma 6.6. *Für jedes Paar v_i und v_j von Knoten gilt*

$$\mathbf{Pr}\left[v_i \text{ und } v_j \text{ werden durch den Schnitt } (U, V \setminus U) \text{ getrennt}\right] = \frac{\theta_{ij}}{\pi}.$$

Beweis. Für $z_i = z_j$ beträgt die Wahrscheinlichkeit, dass v_i und v_j getrennt werden, null. Dies entspricht der Aussage des Lemmas. Sei nun $z_i \neq z_j$. Wir projizieren den Vektor $\hat{r}$ auf die Ebene, die durch die Vektoren z_i und z_j aufgespannt wird. Es bezeichne r' diese Projektion. Wir haben oben diskutiert, dass $r'/\|r'\|$ ein uniform zufälliger Einheitsvektor in dieser Ebene ist. Um zu bestimmen, wann die Knoten v_i und v_j in dem Schnitt $(U, V \setminus U)$ getrennt werden, betrachten wir Abbildung 6.1. Dort haben wir zu den Vektoren z_i und z_j jeweils eine orthogonale Gerade durch den Koordinatenursprung eingezeichnet. Die zu z_i orthogonale Gerade berührt den Einheitskreis in den Punkten A und C, und die zu z_j orthogonale Gerade berührt den Einheitskreis in den Punkten B und D. Durch diese vier Punkte wird der Einheitskreis in vier Sektoren eingeteilt.

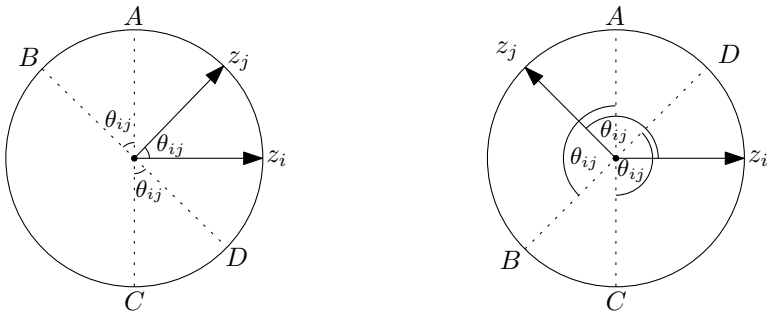


Abbildung 6.1: Die Knoten v_i und v_j werden durch den Schnitt genau dann getrennt, wenn der Vektor $r'/\|r'\|$ in den Kreissektor zwischen A und B oder in den Kreissektor zwischen C und D fällt. Wir gehen dabei o. B. d. A. davon aus, dass von z_i ausgehend der Abstand zu z_j im Uhrzeigersinn mindestens so groß ist wie gegen den Uhrzeigersinn.

Sei $r'' = r'/\|r'\|$. Die Wahrscheinlichkeit, dass r'' exakt einen der Punkte A, B, C oder D trifft, ist 0. Wir können im Folgenden davon ausgehen, dass dieses Ereignis nicht eintritt.

- Fällt der Vektor r'' in den Kreissektor AB, so gilt $r' \cdot z_i < 0$ und $r' \cdot z_j > 0$.
- Fällt der Vektor r'' in den Kreissektor BC, so gilt $r' \cdot z_i < 0$ und $r' \cdot z_j < 0$.
- Fällt der Vektor r'' in den Kreissektor CD, so gilt $r' \cdot z_i > 0$ und $r' \cdot z_j < 0$.
- Fällt der Vektor r'' in den Kreissektor DA, so gilt $r' \cdot z_i > 0$ und $r' \cdot z_j > 0$.

Die Knoten v_i und v_j werden durch den Schnitt also genau dann getrennt, wenn r'' entweder in den Kreissektor AB oder in den Kreissektor CD fällt. Beide haben einen Mittelpunktswinkel von θ_{ij}. Da r'' uniform zufällig auf dem Einheitskreis verteilt ist, beträgt die Wahrscheinlichkeit, dass es in einen dieser beiden Kreissektoren fällt, somit $2\theta_{ij}/(2\pi) = \theta_{ij}/\pi$. □

Oben haben wir bereits $W^* \geq \text{OPT}$ als den optimalen Wert des Vektorprogramms eingeführt. Wir bezeichnen nun mit W das Gesamtgewicht des Schnittes, den der Algorithmus SDPMaxCut berechnet. Sei außerdem

$$\alpha = \frac{2}{\pi} \cdot \min_{0<\theta\leq\pi} \frac{\theta}{1-\cos(\theta)}. \tag{6.4}$$

Lemma 6.7. *Es gilt* $\mathbf{E}[W] \geq \alpha W^* \geq \alpha\text{OPT}$.

Beweis. Wie schon bei der Analyse des einfachen randomisierten Algorithmus RandMaxCut nutzen wir die Linearität des Erwartungswertes. Es gilt

$$\begin{aligned}\mathbf{E}[W] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij} \cdot \mathbf{Pr}[v_i \text{ und } v_j \text{ werden durch den Schnitt } (U, V\setminus U) \text{ getrennt}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} w_{ij} \cdot \frac{\theta_{ij}}{\pi}, \end{aligned} \tag{6.5}$$

wobei wir für die zweite Gleichung Lemma 6.6 genutzt haben. Aus der Definition von α folgt

$$\theta_{ij} \geq \alpha \cdot \frac{\pi}{2}(1-\cos(\theta_{ij})).$$

Setzen wir dies in (6.5) ein, so erhalten wir

$$\mathbf{E}[W] \geq \alpha \cdot \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{w_{ij}}{2}(1-\cos(\theta_{ij})) = \alpha W^* \geq \alpha\text{OPT}. \qquad \square$$

Lemma 6.8. *Es gilt* $\alpha \geq 0{,}878$.

Beweis. Das Lemma folgt durch eine Kurvendiskussion. Die Funktion $\frac{\theta}{1-\cos(\theta)}$ nimmt ihr Minimum in dem Intervall $[0, \pi]$ an der Stelle $\theta = 2{,}3311\ldots$ an. Für diesen Wert von θ gilt

$$\frac{2}{\pi} \cdot \frac{\theta}{1-\cos(\theta)} \geq 0{,}878. \qquad \square$$

Zusammengefasst erhalten wir das folgende Theorem.

Theorem 6.9. *Der Algorithmus* SDPMaxCut *ist ein* 0,878*-Approximationsalgorithmus für das Maximum-Cut Problem.*

6.3 Max-2SAT

Wir betrachten nun den Spezialfall des Problems Max-SAT, bei dem jede Klausel nur höchstens zwei Literale enthalten darf. Diesen bezeichnen wir mit Max-2SAT. Man kann für diesen Spezialfall in polynomieller Zeit testen, ob eine gegebene Formel eine Belegung der Variablen besitzt, die alle Klauseln erfüllt. Es ist jedoch NP-schwer, eine Belegung zu finden, die für eine gegebene Formel so viele Klauseln wie möglich erfüllt. Sind die Klauseln gewichtet, so ist es natürlich ebenfalls NP-schwer eine Belegung der Variablen zu finden, sodass das Gesamtgewicht der erfüllten Klauseln maximal wird. Wir haben in Abschnitt 5.1.2 für das Problem Max-SAT einen 0,75-Approximationsalgorithmus kennengelernt. Wir werden nun sehen, dass wir für den Spezialfall Max-2SAT einen besseren Approximationsfaktor mithilfe von semidefiniter Programmierung erreichen können.

Sei im Folgenden φ eine Eingabe für Max-2SAT mit Variablen $x_1, \dots, x_n$, Klauseln $C_1, \dots, C_m$ und Gewichten $w_1, \dots, w_m$. Zunächst formulieren wir diese Instanz als quadratisches Programm. Dazu führen wir Variablen $y_0, y_1, \dots, y_n$ ein, die nur die Werte $+1$ oder -1 annehmen können. Die Interpretation dieser Variablen ist wie folgt: die Aussagenvariable x_i ist genau dann auf 1 gesetzt, wenn $y_i = y_0$ gilt. Wir definieren den Wert $v(C_j)$ einer Klausel C_j als 1, wenn die Belegung der Aussagenvariablen, die durch die Variablen $y_0, y_1, \dots, y_n$ codiert wird, die Klausel erfüllt, und sonst als 0. Man überprüft leicht, dass für Klauseln der Länge 1

$$v(x_i) = \frac{1 + y_0 y_i}{2} \quad \text{und} \quad v(\neg x_i) = \frac{1 - y_0 y_i}{2}$$

gilt. Betrachten wir nun eine Klausel der Länge 2 mit zwei positiven Literalen x_i und x_j. Es gilt

$$\begin{aligned} v(x_i \vee x_j) &= 1 - v(\neg x_i) v(\neg x_j) = 1 - \frac{1 - y_0 y_i}{2} \cdot \frac{1 - y_0 y_j}{2} \\ &= 1 - \frac{1 - y_0 y_i - y_0 y_j + y_0^2 y_i y_j}{4} \\ &= \frac{3}{4} + \frac{y_0 y_i + y_0 y_j - y_i y_j}{4} \\ &= \frac{1 + y_0 y_i}{4} + \frac{1 + y_0 y_j}{4} + \frac{1 - y_i y_j}{4}, \end{aligned}$$

wobei wir $y_0^2 = 1$ genutzt haben. Die wesentliche Erkenntnis für uns ist, dass wir den Wert $v(x_i \vee x_j)$ als Linearkombination von Termen der Form $(1 + y_a y_b)$ und $(1 - y_a y_b)$ für $a, b \in \{0, 1, \dots, n\}$ darstellen können. Der Leser überzeuge sich als Übung davon, dass dies auch dann noch der Fall ist, wenn die betrachtete Klausel der Länge zwei ein positives und ein negatives oder zwei negative Literale enthält.

Die Zielfunktion, die wir maximieren möchten, ist $\sum_{j=1}^{m} w_j v(C_j)$. Da sich die Werte von allen Klauseln als Linearkombinationen von Termen der Form $(1+y_a y_b)$ und $(1-y_a y_b)$ für $a, b \in \{0, 1, \ldots, n\}$ darstellen lassen, können wir geeignete Koeffizienten a_{ij} und b_{ij} finden, sodass

$$\sum_{j=1}^{m} w_j v(C_j) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (a_{ij}(1 + y_i y_j) + b_{ij}(1 - y_i y_j))$$

gilt. Wir erhalten dann als quadratisches Programm

$$\begin{aligned} \text{maximiere } & \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (a_{ij}(1 + y_i y_j) + b_{ij}(1 - y_i y_j)) \\ \text{sodass } & y_i^2 = 1, && \forall i \in \{0, 1, \ldots, n\}, \\ & y_i \in \mathbb{Z}, && \forall i \in \{0, 1, \ldots, n\}. \end{aligned}$$

Auch hier betrachten wir wieder die Relaxierung als Vektorprogramm:

$$\begin{aligned} \text{maximiere } & \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (a_{ij}(1 + z_i \cdot z_j) + b_{ij}(1 - z_i \cdot z_j)) \\ \text{sodass } & z_i \cdot z_i = 1, && \forall i \in \{0, 1, \ldots, n\}, \\ & z_i \in \mathbb{R}^{n+1}, && \forall i \in \{0, 1, \ldots, n\}. \end{aligned}$$

Der Approximationsalgorithmus SDP2SAT für Max-2SAT, den wir betrachten, arbeitet vollkommen analog zu dem Algorithmus SDPMaxCut für das Maximum-Cut Problem. Das heißt, zuerst berechnet er eine optimale Lösung $(z_0, \ldots, z_n)$ für das Vektorprogramm (auch hier ignorieren wir wieder, dass dies nur mit einem additiven Fehler möglich ist, den wir beliebig klein machen können). Den Wert dieser Lösung bezeichnen wir mit W^*. Anschließend wählt SDP2SAT einen uniform zufälligen Vektor r auf der Einheitssphäre $\mathcal{S}^n = \{x \in \mathbb{R}^{n+1} \mid \|x\| = 1\}$ und setzt $y_i = +1$ für $i \in \{0, \ldots, n\}$ genau dann, wenn $r \cdot z_i \geq 0$ gilt. Wir bezeichnen mit W das Gesamtgewicht, der durch diese Lösung erfüllten Klauseln.

Lemma 6.10. *Es gilt* $\mathbf{E}[W] \geq \alpha W^*$ *für den in* (6.4) *definierten Wert von* α.

Beweis. Wir nutzen die Linearität des Erwartungswertes und erhalten

$$\mathbf{E}[W] = 2 \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (a_{ij} \cdot \mathbf{Pr}[y_i = y_j] + b_{ij} \cdot \mathbf{Pr}[y_i \neq y_j]).$$

Wir bezeichnen wieder mit θ_{ij} den Winkel zwischen z_i und z_j. Dann folgt analog zu der Analyse von SDPMaxCut für alle $i, j \in \{0, \ldots, n\}$

$$\mathbf{Pr}[y_i \neq y_j] = \frac{\theta_{ij}}{\pi} \geq \frac{\alpha}{2}(1 - \cos(\theta_{ij})).$$

Man kann ebenfalls zeigen (Übung für den Leser), dass

$$\mathbf{Pr}[y_i = y_j] = 1 - \frac{\theta_{ij}}{\pi} \geq \frac{\alpha}{2}(1 + \cos(\theta_{ij}))$$

gilt. Nutzen wir genau wie bei der Analyse von SDPMaxCut aus, dass $z_i \cdot z_j = \cos(\theta_{ij})$ gilt, so erhalten wir insgesamt

$$\mathbf{E}[W] \geq \alpha \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (a_{ij} \cdot (1 + \cos(\theta_{ij})) + b_{ij} \cdot (1 - \cos(\theta_{ij}))) = \alpha W^*. \qquad \square$$

Zusammengefasst erhalten wir das folgende Ergebnis.

Theorem 6.11. *Der Algorithmus* SDP2SAT *ist ein* 0,878*-Approximationsalgorithmus für das Problem* Max-2SAT.

6.4 Correlation Clustering

Als letzte Anwendung für semidefinite Programmierung beschäftigen wir uns nun noch mit dem Problem Correlation Clustering. Die Eingabe für dieses Problem besteht aus einer Menge $X = \{x_1, \ldots, x_n\}$ von Datenpunkten sowie zwei symmetrischen Funktionen $w^+ : X \times X \to \mathbb{R}_{>0}$ und $w^- : X \times X \to \mathbb{R}_{>0}$. Für zwei Datenpunkte x_i und x_j ist $w^+_{i,j} := w^+(x_i, x_j)$ ein Maß für ihre Ähnlichkeit. Das heißt, je größer $w^+_{i,j}$ ist, desto ähnlicher sind sich die Datenpunkte x_i und x_j. Andersherum ist $w^-_{i,j} := w^-(x_i, x_j)$ ein Maß dafür, wie verschieden die Datenpunkte x_i und x_j sind. Je größer $w^-_{i,j}$ desto verschiedener sind x_i und x_j. Je nach Anwendung sind die Werte $w^+_{i,j}$ und $w^-_{i,j}$ natürlich korreliert. Wir betrachten hier aber allgemeine Funktionen.

Das Ziel ist es, eine Partition $\mathcal{P} = (P_1, \ldots, P_k)$ der Menge X zu berechnen, wobei die Zahl k der Klassen nicht vorgegeben ist. Für eine Partition $\mathcal{P}$ sei

$$\delta^=(\mathcal{P}) = \{(i,j) \mid i < j,\ i \text{ und } j \text{ sind in derselben Klasse der Partition } \mathcal{P}\}$$

und

$$\delta^{\neq}(\mathcal{P}) = \{(i,j) \mid i < j,\ i \text{ und } j \text{ sind in verschiedenen Klassen der Partition } \mathcal{P}\}.$$

Die zu maximierende Zielfunktion können wir dann schreiben als

$$v(\mathcal{P}) := \sum_{(i,j) \in \delta^=} w^+_{ij} + \sum_{(i,j) \in \delta^{\neq}} w^-_{ij}.$$

Wir möchten also eine Partition finden, sodass zum einen die Ähnlichkeit innerhalb der Klassen und zum anderen die Unähnlichkeit zwischen verschiedenen Klassen möglichst groß ist. Dies sind im Allgemeinen gegenläufige Forderungen, denn die erste Summe in der Zielfunktion wird maximal, wenn alle Datenpunkte in derselben Klasse liegen, und die zweite Summe wird maximal, wenn jeder Datenpunkt seine eigene Klasse bildet. Beim Correlation Clustering wird der beste Kompromiss dieser beiden Extremfälle gesucht.

Aus der obigen Diskussion ergibt sich direkt ein einfacher $\frac{1}{2}$-Approximationsalgorithmus. Wir haben gesehen, dass die Partition $\mathcal{P}$, die alle Datenpunkte einer einzigen

Klasse zuweist, den Wert $v(\mathcal{P}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij}^{+}$ besitzt. Auf der anderen Seite erreicht die Partition $\mathcal{P}'$, die jeden Datenpunkt seiner eigenen Klasse zuweist, einen Wert von $v(\mathcal{P}') = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij}^{-}$. Wegen

$$\mathrm{OPT} \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij}^{+} + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij}^{-} = v(\mathcal{P}) + v(\mathcal{P}')$$

muss somit entweder $v(\mathcal{P}) \geq \mathrm{OPT}/2$ oder $v(\mathcal{P}') \geq \mathrm{OPT}/2$ gelten.

Wir entwerfen nun einen Approximationsalgorithmus, der einen besseren Approximationsfaktor als $\frac{1}{2}$ erreicht. Dazu formulieren wir die gegebene Instanz von Correlation Clustering wie folgt als Vektorprogramm:

$$\begin{aligned} \text{maximiere } & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Big(w_{ij}^{+}(y_i \cdot y_j) + w_{ij}^{-}(1 - y_i \cdot y_j) \Big) \\ \text{sodass } & y_i \in \{e_1, \ldots, e_n\}, && \forall i \in \{1, \ldots, n\}. \end{aligned}$$

In diesem Vektorprogramm bezeichnen $e_1, \ldots, e_n$ die kanonischen Einheitsvektoren des $\mathbb{R}^n$. Das heißt, der Vektor e_i besitzt eine 1 an Position i und alle anderen Einträge sind 0. Wegen $e_i \cdot e_j = 1$ für $i = j$ und $e_i \cdot e_j = 0$ für $i \neq j$ modelliert dies genau die gegebene Instanz von Correlation Clustering. Dabei gehören zwei Datenpunkte x_i und x_j genau dann zu derselben Klasse der Partition, wenn $y_i = y_j$ gilt.

Da das obige Vektorprogramm wegen der Bedingungen an die Wahl der y_i nicht effizient gelöst werden kann, relaxieren wir es und erhalten:

$$\begin{aligned} \text{maximiere } & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Big(w_{ij}^{+}(z_i \cdot z_j) + w_{ij}^{-}(1 - z_i \cdot z_j) \Big) \\ \text{sodass } & z_i \cdot z_i = 1, && \forall i \in \{1, \ldots, n\}, \\ & z_i \cdot z_j \geq 0, && \forall i, j \in \{1, \ldots, n\}, \\ & z_i \in \mathbb{R}^n, && \forall i \in \{1, \ldots, n\}. \end{aligned}$$

Wir bezeichnen mit $(z_1, \ldots, z_n)$ eine optimale Lösung des Vektorprogramms und mit W^* ihren Wert. Dann gilt $W^* \geq \mathrm{OPT}$, da es sich um eine Relaxierung des obigen Vektorprogramms handelt, das die Instanz von Correlation Clustering exakt modelliert.

Wir runden die Lösung $(z_1, \ldots, z_n)$ nun auf ähnliche Art und Weise wie in den Algorithmen SDPMaxCut und SDP2SAT, wählen aber unabhängig zwei uniform zufällige Vektoren r_1 und r_2 von der Einheitssphäre $\mathcal{S}^{n-1}$. Diese beiden Vektoren induzieren eine Partition der Menge der Datenpunkte in vier Klassen:

$$\begin{aligned} P_1 &= \{x_i \mid r_1 \cdot z_i \geq 0, r_2 \cdot z_i \geq 0\}, \\ P_2 &= \{x_i \mid r_1 \cdot z_i \geq 0, r_2 \cdot z_i < 0\}, \\ P_3 &= \{x_i \mid r_1 \cdot z_i < 0, r_2 \cdot z_i \geq 0\}, \\ P_4 &= \{x_i \mid r_1 \cdot z_i < 0, r_2 \cdot z_i < 0\}. \end{aligned}$$

Wir zeigen nun, dass die Partition $\mathcal{P} = (P_1, P_2, P_3, P_4)$ im Erwartungswert einen Wert von mindestens $\frac{3}{4}W^* \geq \frac{3}{4}\mathrm{OPT}$ erreicht. Dazu benötigen wir das folgende Lemma, das mit einer elementaren Kurvendiskussion gezeigt werden kann.

Lemma 6.12. *Für $\theta \in [0, \pi/2]$ gilt*

$$\left(1 - \frac{\theta}{\pi}\right)^2 \geq \frac{3}{4}\cos(\theta) \quad \textit{und} \quad \left(1 - \left(1 - \frac{\theta}{\pi}\right)^2\right) \geq \frac{3}{4}(1 - \cos(\theta)).$$

Theorem 6.13. *Es gilt $\mathbf{E}[v(\mathcal{P})] \geq \frac{3}{4}W^* \geq \frac{3}{4}\mathrm{OPT}$.*

Beweis. Für $i, j \in \{1, \ldots, n\}$ sei $X_{ij} \in \{0, 1\}$ eine Zufallsvariable, die angibt, ob die Datenpunkte x_i und x_j zu derselben Klasse der Partition $\mathcal{P}$ gehören (in diesem Fall gilt $X_{ij} = 1$ und ansonsten $X_{ij} = 0$). Aus Lemma 6.6 erhalten wir die Eigenschaft, dass die Vektoren z_i und z_j mit einer Wahrscheinlichkeit von $\frac{\theta_{ij}}{\pi}$ durch ein uniform zufälliges $r \in \mathcal{S}^{n-1}$ getrennt werden, wobei θ_{ij} wieder den Winkel zwischen z_i und z_j bezeichnet. Da r_1 und r_2 unabhängig sind, ergibt sich

$$\begin{aligned}\mathbf{E}[X_{ij}] &= \mathbf{Pr}[X_{ij} = 1] \\ &= \mathbf{Pr}[z_i \text{ und } z_j \text{ werden weder durch } r_1 \text{ noch durch } r_2 \text{ getrennt}] \\ &= \left(1 - \frac{\theta_{ij}}{\pi}\right)^2.\end{aligned}$$

Den Wert $v(\mathcal{P})$ der Partition $\mathcal{P}$ können wir schreiben als

$$v(\mathcal{P}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(w_{ij}^+ X_{ij} + w_{ij}^-(1 - X_{ij})\right).$$

Mit der Linearität des Erwartungswertes folgt

$$\begin{aligned}\mathbf{E}[v(\mathcal{P})] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(w_{ij}^+ \mathbf{E}[X_{ij}] + w_{ij}^-(1 - \mathbf{E}[X_{ij}])\right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(w_{ij}^+ \left(1 - \frac{\theta_{ij}}{\pi}\right)^2 + w_{ij}^- \left(1 - \left(1 - \frac{\theta_{ij}}{\pi}\right)^2\right)\right).\end{aligned}$$

Die Nebenbedingungen $z_i \cdot z_j \geq 0$ stellen sicher, dass alle Winkel θ_{ij} in dem Intervall $[0, \pi/2]$ liegen. Wir können also Lemma 6.12 benutzen und erhalten

$$\begin{aligned}\mathbf{E}[v(\mathcal{P})] &\geq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(w_{ij}^+ \cdot \frac{3}{4}\cos(\theta_{ij}) + w_{ij}^- \cdot \frac{3}{4}(1 - \cos(\theta_{ij}))\right) \\ &= \frac{3}{4} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(w_{ij}^+(z_i \cdot z_j) + w_{ij}^-(1 - (z_i \cdot z_j))\right) = \frac{3}{4} \cdot W^*.\end{aligned}$$

□

Kapitel 7

Lokale Suche

Lokale Suche ist eine einfache aber für viele Probleme erfolgreiche Technik zum Entwurf von Approximationsalgorithmen. Ein lokaler Suchalgorithmus startet zunächst mit einer beliebigen Lösung für die gegebene Probleminstanz und verbessert diese dann solange lokal, bis ein lokales Optimum erreicht ist, also eine Lösung, in der keine lokale Verbesserung mehr möglich ist. Wie genau der Begriff einer lokalen Verbesserung definiert ist, hängt vom betrachteten Optimierungsproblem ab. Oft gibt es für ein Problem sogar mehrere naheliegende Definitionen. Das Ziel besteht darin, den Begriff der lokalen Verbesserung so zu definieren, dass zum einen jedes lokale Optimum einen guten Approximationsfaktor erreicht und zum anderen ein lokales Optimum mit polynomiell vielen lokalen Verbesserungen erreicht werden kann. In diesem Kapitel werden wir für verschiedene Probleme Approximationsalgorithmen kennenlernen, die auf lokaler Suche basieren.

7.1 Spannbäume mit minimalem Maximalgrad

Wir betrachten eine Variante des Spannbaumproblems, bei der es darum geht, in einem ungewichteten Graphen einen Spannbaum mit möglichst kleinem Maximalgrad zu finden.

Minimum-Degree Spanning Tree Problem (MDST-Problem)

Eingabe:	ungerichteter Graph $G = (V, E)$
Lösungen:	alle Spannbäume T von G
Zielfunktion:	minimiere den Maximalgrad von T

Es ist NP-schwer zu entscheiden, ob ein gegebener Graph einen Spannbaum mit Maximalgrad 2 besitzt. Ein solcher Spannbaum entspricht nämlich einem Hamiltonpfad, also einem Pfad, der jeden Knoten des Graphen genau einmal enthält. Zu entscheiden, ob ein Graph einen solchen Hamiltonpfad besitzt, ist ein bekanntes NP-vollständiges Problem, und somit ist auch das MDST-Problem NP-schwer.

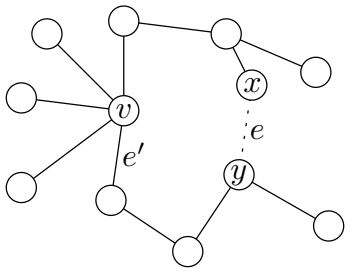


Abbildung 7.1: Der Austausch von e' durch e ist eine lokale Verbesserung.

7.1.1 Erster Algorithmus

Für einen Spannbaum T von G und einen Knoten $v \in V$ bezeichne $d_T(v)$ den Grad von v im Spannbaum T. Außerdem sei $\Delta(T) = \max_{v \in V} d_T(v)$, $n = |V|$ und OPT der optimale Maximalgrad. Wir definieren nun, was wir unter einer lokalen Verbesserung verstehen werden. Sei dazu $v \in V$ ein Knoten und sei $e = (x, y) \in E$ eine Kante, deren Hinzunahme zum Spannbaum T einen Kreis C erzeugt, der v enthält. Ferner sei $\max\{d_T(x), d_T(y)\} \leq d_T(v) - 2$. Existieren ein Knoten v und eine Kante e mit diesen Eigenschaften, dann verändern wir den Spannbaum, indem wir die Kante e hinzufügen und eine Kante $e' \in C$ entfernen, die zu v inzident ist. Sei T' der resultierende Spannbaum. Diese Operation ist in Abbildung 7.1 dargestellt.

Die oben beschriebene lokale Verbesserung verringert den Grad von v (und eines weiteren Knotens) um eins und erhöht die Grade von x und y um jeweils eins. Aufgrund der Annahme über die Grade der Knoten haben die drei Knoten x, y und v im Spannbaum T' jeweils einen Grad von höchstens $d_T(v) - 1$. Der Maximalgrad dieser drei Knoten wird durch die lokale Verbesserung also um eins reduziert.

Ein naheliegender lokaler Suchalgorithmus wäre es nun, mit einem beliebigen Spannbaum zu starten und solange lokale Verbesserungen durchzuführen, bis keine mehr möglich ist. Dabei beschränken wir uns nicht auf lokale Verbesserungen an Knoten $v \in V$ mit $d_T(v) = \Delta(T)$, sondern erlauben beliebige lokale Verbesserungen. Für einen Spannbaum kann in polynomieller Zeit getestet werden, ob eine lokale Verbesserung möglich ist. Der wesentliche Parameter in der Laufzeit der lokalen Suche ist demzufolge die Anzahl an lokalen Verbesserungen, die durchgeführt werden, bis ein lokales Optimum erreicht ist. Man kann zeigen, dass stets nach endlich vielen Schritten ein lokales Optimum erreicht wird (der Leser überlege sich, warum dies so ist), allerdings ist keine polynomielle Schranke für die Anzahl der Schritte bekannt. Deshalb ändern wir den Algorithmus leicht ab und schränken die erlaubten lokalen Verbesserungen ein.

Es sei $\ell = \lceil \log_2(n) \rceil$. Der Algorithmus LocalMDST startet mit einem beliebigen Spannbaum. In jedem Schritt testet er für den aktuellen Spannbaum T, ob eine lokale Verbesserung möglich ist, bei der der involvierte Knoten v einen Grad zwischen $\Delta(T) - \ell$ und $\Delta(T)$ besitzt. Ist eine solche lokale Verbesserung möglich, wird eine beliebige davon ausgeführt. Ansonsten terminiert der Algorithmus und gibt den aktuellen Spannbaum T aus. Sprechen wir im Folgenden von einem lokalen Optimum, so ist damit stets ein lokales Optimum bezüglich der Einschränkung gemeint, dass in

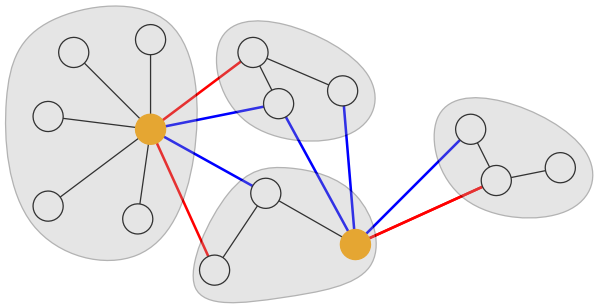


Abbildung 7.2: In diesem Beispiel bestehe der Spannbaum T aus den schwarzen und roten Kanten. Die Menge X bestehe aus den roten Kanten. Diese werden aus T entfernt, was zu vier Zusammenhangskomponenten führt. Die blauen Kanten bilden gemeinsam mit den roten Kanten die Menge E^X der Kanten aus dem Graphen zwischen zwei Zusammenhangskomponenten. Jede dieser Kanten ist inzident zu mindestens einem orangefarbenen Knoten. Die orangefarbenen Knoten können demnach als Menge S gewählt werden. Aus Lemma 7.1 folgt, dass in jedem Spannbaum einer der beiden orangefarbenen Knoten mindestens Grad 2 besitzt.

einer lokalen Verbesserung nur Knoten mit einem Grad zwischen $\Delta(T) - \ell$ und $\Delta(T)$ als Knoten v gewählt werden dürfen.

Wir werden nun eine obere Schranke für den Maximalgrad $\Delta(T)$ eines jeden lokal optimalen Spannbaums zeigen. Dazu benötigen wir zunächst eine untere Schranke für OPT. Um eine solche Schranke zu finden, müssen wir intuitiv eine möglichst kleine Menge von Knoten im Graphen G identifizieren, die in jedem Spannbaum insgesamt viele inzidenten Kanten besitzen müssen. Das folgende Lemma, welches in Abbildung 7.2 illustriert wird, macht diese Überlegung formal.

Wir sagen im Folgenden, dass eine Kante e zu einer Menge $S \subseteq V$ inzident ist, wenn mindestens einer der Endknoten von e zu der Menge S gehört.

Lemma 7.1. *Es sei T ein beliebiger Spannbaum und es sei X eine Teilmenge der Kanten von T. Entfernen wir die Kanten X aus dem Spannbaum T, so erhalten wir $|X| + 1$ Zusammenhangskomponenten. Wir bezeichnen mit $E^X \subseteq E$ die Kanten des Graphen G, die zwischen zwei dieser Zusammenhangskomponenten verlaufen. Ferner sei $S \subseteq V$ so gewählt, dass jede Kante aus E^X inzident zu S ist. Dann gilt* $\mathrm{OPT} \geq \lceil |X|/|S| \rceil$.

Beweis. Jeder Spannbaum muss mindestens $|X|$ Kanten aus der Menge E^X enthalten, da er sonst nicht zusammenhängend sein könnte. Da jede Kante aus E^X mindestens einen Endknoten in der Menge S besitzt, bedeutet dies, dass der Gesamtgrad der Knoten aus S mindestens $|X|$ beträgt. Der Durchschnittsgrad der Knoten aus S beträgt somit mindestens $|X|/|S|$. Daraus folgt, dass es einen Knoten in S geben muss, dessen Grad mindestens $|X|/|S|$ beträgt. Da alle Knotengrade ganzzahlig sind, folgt das Lemma. □

Mithilfe des vorangegangenen Lemmas können wir das folgende Theorem zeigen.

Theorem 7.2. *Ist T ein lokal optimaler Spannbaum, so gilt $\Delta(T) \leq 2 \cdot \mathrm{OPT} + \ell$.*

Beweis. Für $i \in \mathbb{N}$ bezeichne $S_i = \{v \in V \mid d_T(v) \geq i\}$ die Menge der Knoten, die im Spannbaum T mindestens Grad i besitzen. Wir nutzen nun die Mengen S_i, um eine geeignete Menge X für die Anwendung von Lemma 7.1 zu finden.

Beobachtung 7.3. *Es gibt ein $i \in \{\Delta(T) - \ell + 1, \ldots, \Delta(T)\}$ mit $|S_{i-1}| \leq 2|S_i|$.*

Beweis. Nehmen wir an, dass die Aussage nicht gilt. Dann folgt

$$|S_{\Delta(T)-\ell}| > 2^\ell \cdot |S_{\Delta(T)}| \geq n$$

im Widerspruch dazu, dass $S_{\Delta(T)-\ell}$ als Teilmenge von V nur höchstens n Knoten enthalten kann. □

Beobachtung 7.4. *Für jedes i gibt es mindestens $(i-1)|S_i| + 1$ Kanten im Spannbaum T, die zu S_i inzident sind.*

Beweis. Die Summe der Grade der Knoten aus S_i beträgt mindestens $i \cdot |S_i|$ und es kann nur höchstens $|S_i| - 1$ viele Kanten geben, für die beide Endknoten aus S_i stammen. Letzteres folgt, da T ein Spannbaum und somit kreisfrei ist. Dies bedeutet insgesamt, dass es mindestens

$$i \cdot |S_i| - (|S_i| - 1) = (i-1)|S_i| + 1$$

viele verschiedene Kanten gibt, die zu S_i inzident sind. □

In der folgenden Beobachtung betrachten wir die Situation, dass wir alle Kanten aus T entfernen, die inzident zu S_i sind. Wir zeigen, dass aufgrund der lokalen Optimalität von T danach alle Kanten, die zwei verschiedene Zusammenhangskomponenten verbinden, zu S_{i-1} inzident sind.

Beobachtung 7.5. *Sei T ein lokal optimaler Spannbaum und sei $i \geq \Delta(T) - \ell$ beliebig. Ferner sei X die Menge der Kanten aus T, die inzident zu S_i sind. Dann ist jede Kante aus E^X inzident zu S_{i-1}.*

Beweis. Angenommen es existiert eine Kante $e = (x, y) \in E^X$, für die $d_T(x) \leq i - 2$ und $d_T(y) \leq i - 2$ gilt. Die Kante e kann nicht zum Spannbaum T gehören, da sie nicht inzident zu S_i ist. Mithilfe von e ist es möglich, in dem Spannbaum T eine lokale Verbesserung durchzuführen. Fügt man nämlich die Kante e dem Spannbaum T hinzu, so entsteht ein Kreis. Dieser enthält neben der Kante e auch einen Knoten $v \in S_i$, da der Kreis verschiedene Zusammenhangskomponenten verbindet und alle Kanten des Spannbaums T, die zwischen verschiedenen Zusammenhangskomponenten verlaufen, inzident zu S_i sind. Der Austausch einer zu v inzidenten Kante auf dem Kreis durch e wäre wegen $i \geq \Delta(T) - \ell$ eine lokale Verbesserung im Spannbaum T. □

Aus den drei Beobachtungen folgt das Theorem. Zunächst wählen wir ein $i \in \{\Delta(T) - \ell + 1, \ldots, \Delta(T)\}$ mit $|S_{i-1}| \leq 2|S_i|$, welches wegen Beobachtung 7.3 existiert. Dann definieren wir X als die Menge der Kanten aus T, die inzident zu S_i sind, und wenden Lemma 7.1 an. Da jede Kante aus E^X gemäß Beobachtung 7.5 inzident zu S_{i-1} ist, folgt

$$\text{OPT} \geq \left\lceil \frac{|X|}{|S_{i-1}|} \right\rceil \geq \frac{|X|}{|S_{i-1}|}.$$

Gemäß Beobachtung 7.4 gilt $|X| \geq (i-1)|S_i| + 1$ und damit

$$\text{OPT} \geq \frac{(i-1)|S_i| + 1}{|S_{i-1}|} \geq \frac{(i-1)|S_i| + 1}{2|S_i|} > \frac{i-1}{2} \geq \frac{\Delta(T) - \ell}{2}.$$

Das Theorem folgt aus dem Umstellen dieser Ungleichung nach $\Delta(T)$. □

Aus Theorem 7.2 lässt sich nicht direkt eine Aussage über den Approximationsfaktor von LocalMDST ableiten, da zusätzlich zum Faktor 2 noch ein additiver Term $\ell = \lceil \log_2(n) \rceil$ hinzukommt. Ist beispielsweise $\text{OPT} \geq \ell$, so erreicht der Algorithmus einen Approximationsfaktor von 3. Ist OPT kleiner als ℓ, so ist der Approximationsfaktor schlechter.

Es bleibt noch zu zeigen, dass der Algorithmus LocalMDST eine polynomielle Laufzeit besitzt. Da jede lokale Verbesserung in polynomieller Zeit durchgeführt werden kann, ist der entscheidende Parameter die Anzahl an lokalen Verbesserungen bis zu einem lokalen Optimum.

Theorem 7.6. *Die Anzahl an lokalen Verbesserungen, die von* LocalMDST *durchgeführt werden, ist durch $O(n^4)$ beschränkt.*

Beweis. Eine allgemeine Technik zur Analyse der Schrittanzahl einer lokalen Suche, die wir auch in diesem Beweis nutzen werden, ist der Einsatz einer Potentialfunktion. Eine solche Funktion ordnet jeder Lösung einen reellen Wert zu. Kann man dann zeigen, dass die Werte aller Lösungen aus einem beschränkten Intervall stammen und dass der Wert in jedem Schritt um mindestens einen gewissen Betrag oder einen gewissen Faktor sinkt, so kann abhängig von der Intervalllänge und dem Betrag bzw. Faktor eine obere Schranke für die Anzahl an Schritten gezeigt werden.

Konkret ordnen wir einem Spannbaum T das Potential

$$\Phi(T) = \sum_{v \in V} 3^{d_T(v)}$$

zu. Für jeden Spannbaum T gilt $\Phi(T) \leq n3^n$, da kein Knoten einen größeren Grad als n besitzen kann. Außerdem minimiert jeder Hamiltonpfad das Potential. Ein solcher besitzt zwei Knoten mit Grad 1 sowie $n-2$ Knoten mit Grad 2 und somit ein Potential von $2 \cdot 3^1 + (n-2) \cdot 3^2 = 9n - 12 > n$, wobei wir $n \geq 2$ annehmen. Somit ist das Potential jedes Spannbaums größer als n.

Wir zeigen nun, dass sich das Potential mit jeder lokalen Verbesserung um mindestens den Faktor $1 - \frac{2}{27n^3}$ verringert. Dazu betrachten wir wie oben eine lokale Verbesserung,

in der eine Kante e', die inzident zu einem Knoten v mit $d_T(v) \geq \Delta(T) - \ell$ ist, durch eine Kante $e = (x, y)$ ersetzt wird, für die $\max\{d_T(x), d_T(y)\} \leq d_T(v) - 2$ gilt. Durch diesen Schritt verändern sich nur die Grade der Knoten x, y und v sowie der des anderen Endknotens von Kante e'. Der Grad des anderen Endknotens von e' sinkt um eins (wenn es sich nicht um x oder y handelt). Dadurch sinkt das Potential. Wir werden diese Potentialreduktion jedoch nicht berücksichtigen, da bereits durch die Veränderung der Grade von x, y und v eine genügend große Potentialreduktion erreicht wird.

Die Potentialreduktion beträgt mindestens

$$(3^{d_T(v)} - 3^{d_T(v)-1}) + (3^{d_T(x)} - 3^{d_T(x)+1}) + (3^{d_T(y)} - 3^{d_T(y)+1}).$$

Sei $i = d_T(v)$. Wegen $d_T(x) \leq i - 2$ und $d_T(y) \leq i - 2$ können wir diesen Term durch

$$(3^i - 3^{i-1}) + 2(3^{i-2} - 3^{i-1}) = 3^i - 3 \cdot 3^{i-1} + 2 \cdot 3^{i-2} = 2 \cdot 3^{i-2} = \frac{2}{9} \cdot 3^i \geq \frac{2}{9} \cdot 3^{\Delta(T)-\ell}$$

nach unten abschätzen. Wegen

$$3^\ell = 3^{\lceil \log_2(n) \rceil} \leq 3 \cdot 3^{\log_2(n)} \leq 3 \cdot 2^{2\log_2(n)} = 3n^2$$

beträgt die Reduktion des Potential in dem betrachteten Schritt mindestens

$$\frac{2}{9} \cdot 3^{\Delta(T)-\ell} \geq \frac{2}{27n^2} \cdot 3^{\Delta(T)} \geq \frac{2}{27n^3} \cdot \Phi(T),$$

wobei wir im letzten Schritt $\Phi(T) \leq n3^{\Delta(T)}$ genutzt haben. Damit ist insgesamt gezeigt, dass sich das Potential mit jeder lokalen Verbesserung um mindestens den Faktor $1 - \frac{2}{27n^3}$ reduziert.

Sei T_t der Spannbaum, der nach t lokalen Verbesserungen von LocalMDST erreicht wird. Für diesen gilt

$$\Phi(T_t) \leq \left(1 - \frac{2}{27n^3}\right)^t \cdot n3^n.$$

Für $t = \frac{27}{2}n^4 \ln(3)$ ergibt dies

$$\Phi(T_t) \leq \left(\left(1 - \frac{2}{27n^3}\right)^{\frac{27}{2}n^3}\right)^{\ln(3)n} \cdot n3^n \leq e^{-\ln(3)n} \cdot n3^n = n.$$

In dieser Rechnung haben wir genutzt, dass die Ungleichung $(1 - x) \leq e^{-x}$ für alle $x \in [0, 1]$ gilt. Da wir bereits oben argumentiert haben, dass das Potential für jeden Spannbaum größer als n ist, muss die Anzahl lokaler Verbesserungen kleiner als $\frac{27}{2}n^4 \ln(3) = O(n^4)$ sein. □

Die Wahl von ℓ als $\lceil \log_2(n) \rceil$ ist zu einem gewissen Grad beliebig. Für jedes $b > 1$ führt die Wahl $\ell = \lceil \log_b(n) \rceil$ zu einem polynomiellen Algorithmus. Für jedes lokale Optimum gilt dann $\Delta(T) \leq b \cdot \mathrm{OPT} + \lceil \log_b(n) \rceil$.

7.1.2 Verbesserter Algorithmus

Wir werden den Algorithmus aus dem vorangegangenen Abschnitt nun so abändern, dass er in polynomieller Zeit einen Spannbaum T mit $\Delta(T) \leq \mathrm{OPT} + 1$ ausgibt. Da das MDST-Problem NP-schwer ist, ist dies unter der Annahme P $\neq$ NP das bestmögliche Ergebnis. Als erstes nutzen wir Lemma 7.1, um eine hinreichende Bedingung für $\Delta(T) \leq \mathrm{OPT} + 1$ herzuleiten.

Lemma 7.7. *Es sei T ein beliebiger Spannbaum und $k = \Delta(T)$. Es sei ferner $D_k \subseteq \{v \in V \mid d_T(v) = k\}$ mit $D_k \neq \emptyset$ und $D_{k-1} \subseteq \{v \in V \mid d_T(v) = k-1\}$. Es bezeichne X die Menge der Kanten aus T, die inzident zu $D_{k-1} \cup D_k$ sind, und $\mathcal{C}$ bezeichne die Menge der $|X|+1$ Zusammenhangskomponenten, die entstehen, wenn die Kanten aus X aus dem Spannbaum T entfernt werden. Es bezeichne $E^X \subseteq E$ die Menge der Kanten aus G, die zwei Komponenten aus $\mathcal{C}$ verbinden. Ist jede Kante aus E^X inzident zu $D_{k-1} \cup D_k$, so gilt $\Delta(T) \leq \mathrm{OPT} + 1$.*

Beweis. Wir wenden Lemma 7.1 mit X und $S = D_{k-1} \cup D_k$ an und erhalten

$$\mathrm{OPT} \geq \left\lceil \frac{|X|}{|D_{k-1}| + |D_k|} \right\rceil.$$

Die Summe der Knotengrade aller Knoten aus $D_{k-1} \cup D_k$ im Spannbaum T beträgt $(k-1)|D_{k-1}| + k|D_k|$. Da T kreisfrei ist, können innerhalb von $D_{k-1} \cup D_k$ nur höchstens $|D_{k-1}| + |D_k| - 1$ Kanten verlaufen. Damit folgt

$$|X| \geq (k-1)|D_{k-1}| + k|D_k| - (|D_{k-1}| + |D_k| - 1) = (k-1)|D_k| + (k-2)|D_{k-1}| + 1.$$

Insgesamt folgt

$$\begin{aligned} \mathrm{OPT} &\geq \left\lceil \frac{(k-1)|D_k| + (k-2)|D_{k-1}| + 1}{|D_{k-1}| + |D_k|} \right\rceil \\ &= \left\lceil (k-1) - \frac{|D_{k-1}| - 1}{|D_{k-1}| + |D_k|} \right\rceil \\ &\geq k - 1, \end{aligned}$$

wobei wir für die letzte Ungleichung genutzt haben, dass der Bruch in der zweiten Zeile kleiner als 1 ist. Mit $k = \Delta(T)$ folgt das Lemma. □

Unser Ziel ist es nun, die lokale Suche so zu steuern, dass ein Spannbaum erreicht wird, der die Bedingung aus Lemma 7.7 erfüllt. Sobald ein solcher erreicht wird, stoppt der Algorithmus. Wir teilen die Schritte des Algorithmus in Phasen und Teilphasen ein. Ist T mit $k = \Delta(T)$ der Spannbaum zu Beginn einer Phase, so ist es das Ziel der Phase, den Spannbaum durch lokale Verbesserungen so umzubauen, dass der Maximalgrad auf $k-1$ sinkt. In jeder Teilphase wird für einen Knoten mit Grad k der Grad auf $k-1$ verringert, ohne dass dabei neue Knoten mit Grad k entstehen.

Um eine Intuition für den Algorithmus zu erhalten, wenden wir Lemma 7.7 an, wobei wir zunächst D_k und D_{k-1} als die Menge aller Knoten mit Grad k bzw. $k-1$ im

ImprovedLocalMDST

1. Wähle beliebigen Spannbaum T von $G = (V, E)$.
2. **while** true **do**
3. $k := \Delta(T)$ // Beginn einer neuen Phase
4. **while** $\exists v \in V : d_T(v) = k$ **do**
5. $D_k = \{v \in V \mid d_T(v) = k\}$ // Beginn einer Teilphase
6. $D_{k-1} = \{v \in V \mid d_T(v) = k - 1\}$
7. $X =$ Menge der zu $D_{k-1} \cup D_k$ inzidenten Kanten
8. $\mathcal{C} =$ Menge der Zusammenhangskomponenten nach der Entfernung von X
9. Alle Knoten aus D_{k-1} seien unmarkiert.
10. **if** $\forall e \in E^X : e$ ist inzident zu $D_{k-1} \cup D_k$ **then return** T
11. **for each** $e = (x, y) \in E^X : x, y \notin D_{k-1} \cup D_k$ **do**
12. Sei C der Kreis, den Kante e erzeugt, wenn sie T hinzu gefügt wird.
13. **if** $C \cap D_k = \emptyset$ **then**
14. Markiere alle $v \in C \cap D_{k-1}$ als reduzierbar durch e.
15. $D_{k-1} := D_{k-1} \setminus C$
16. Passe X und $\mathcal{C}$ an das neue D_{k-1} an.
17. **else**
18. Sei $v \in C \cap D_k$ beliebig.
19. **if** x oder y markiert **then**
20. Reduziere den Grad von x und/oder y durch eine lokale Verschiebung und propagiere die lokalen Verschiebungen rekursiv soweit notwendig.
21. Reduziere den Grad von v durch eine lokale Verbesserung mit $e = (x, y)$.
22. Verlasse die **for**-Schleife und gehe zu Schritt 4.

Abbildung 7.3: der Algorithmus ImprovedLocalMDST als Pseudocode

Spannbaum T wählen. Mit den Kanten aus der Menge E^X versuchen wir nun, eine lokale Verbesserung zu konstruieren. Jede Kante $e = (x, y)$ aus E^X, die nicht zum Spannbaum T gehört, schließt einen Kreis, wenn sie zu T hinzugefügt wird. Ist jede solche Kante zu $D_{k-1} \cup D_k$ inzident, so ist die Bedingung aus Lemma 7.7 erfüllt und wir haben bereits einen Spannbaum T mit $\Delta(T) \leq \mathrm{OPT} + 1$ gefunden.

Ansonsten sei $e = (x, y)$ eine Kante mit $d_T(x) \leq k - 2$ und $d_T(y) \leq k - 2$. Der Kreis muss mindestens einen Knoten aus $D_{k-1} \cup D_k$ enthalten, da er verschiedene

Zusammenhangskomponenten durchläuft. Liegt ein Knoten $v \in D_k$ auf dem Kreis, so kann eine lokale Verbesserung durchgeführt werden, bei der eine zu v inzidente Kante auf dem Kreis durch e ersetzt wird. Danach betragen die Grade von v, x und y jeweils höchstens $k-1$ und die Teilphase ist abgeschlossen.

Befinden sich auf dem Kreis keine Knoten aus D_k, so muss mindestens ein Knoten $v \in D_{k-1}$ enthalten sein. Durch eine lokale Verbesserung wie oben kann dann zwar erreicht werden, dass der Grad von v auf $k-2$ sinkt, dafür erhöhen sich aber die Grade von x und y auf möglicherweise bis zu $k-1$. Durch eine solche lokale Verbesserung reduziert man also nicht unbedingt die Knotengrade, man kann aber den Grad $k-1$ von einem Knoten zu einem oder zwei anderen verschieben. Wir sprechen deshalb im Folgenden auch von einer lokalen Verschiebung statt von einer lokalen Verbesserung. Solche lokalen Verschiebungen sind in dem Algorithmus, den wir betrachten werden, ein wichtiges Hilfsmittel. In dem Algorithmus werden wir einige Knoten v mit Grad $k-1$ als reduzierbar durch eine Kante e markieren. Dies bedeutet, dass eine lokale Verschiebung möglich ist, bei der die Kante e hinzugefügt wird und der Grad von Knoten v von $k-1$ auf $k-2$ sinkt.

Abbildung 7.3 zeigt den Algorithmus ImprovedLocalMDST im Pseudocode. Dieser Algorithmus verhält sich zunächst wie oben beschrieben, er setzt also D_k und D_{k-1} auf die Menge aller Knoten mit Grad k bzw. $k-1$ im Spannbaum T und berechnet die Mengen X und $\mathcal{C}$ entsprechend. Dann betrachtet er alle Kanten aus E^X, die nicht zu $D_{k-1} \cup D_k$ inzident sind, nacheinander. Für jede solche Kante $e = (x, y)$ wird der Kreis C betrachtet, den sie erzeugt, wenn sie dem Spannbaum T hinzugefügt wird. Dieser Kreis enthält, wie oben diskutiert, mindestens einen Knoten aus $D_{k-1} \cup D_k$. Enthält er keinen Knoten aus D_k, so werden alle Knoten aus D_{k-1} auf dem Kreis als reduzierbar durch die Kante e markiert, denn der Grad jedes Knotens aus $D_{k-1} \cap C$ kann durch eine lokale Verschiebung auf $k-2$ reduziert werden (dies gilt für jeden Knoten aus $D_{k-1} \cap C$ für sich genommen, aber nicht für alle gleichzeitig). Die lokalen Verschiebungen werden an dieser Stelle noch nicht durchgeführt. Alle markierten Knoten werden aus D_{k-1} entfernt. Die Mengen X und $\mathcal{C}$ werden entsprechend angepasst.

Enthält der Kreis C hingegen einen Knoten v aus D_k, so wird eine lokale Verbesserung durchgeführt, indem die Kante e dem Spannbaum hinzugefügt wird und eine auf dem Kreis C zu v inzidente Kante gelöscht wird. Die Wahl von e stellt sicher, dass die Endknoten von e nicht zu $D_{k-1} \cup D_k$ gehören. Sind sie beide unmarkiert, so haben sie jeweils einen Grad von höchstens $k-2$ und die oben genannte lokale Verbesserung kann problemlos durchgeführt werden. Sie schließt die Teilphase ab, da ein Knoten mit Grad k eliminiert wurde. Sind hingegen x und/oder y markiert, so haben sie einen Grad von $k-1$. Durch die lokale Verbesserung würde ihr Grad auf k steigen. Um dies zu verhindern, führen wir zunächst die lokalen Verschiebungen aus, die zu den Markierungen geführt haben. Ist also beispielsweise der Knoten x als durch eine Kante $e' = (x', y')$ reduzierbar markiert, so wird die entsprechende lokale Verschiebung durchgeführt, was den Grad von Knoten x auf $k-2$ reduziert. Führt man also für x und/oder y die entsprechenden lokalen Verschiebungen durch, so sinkt ihr Knotengrad auf $k-2$ und die anvisierte lokale Verbesserung über v und e kann durchgeführt werden.

Wir haben allerdings noch nicht berücksichtigt, dass auch x' und/oder y' markiert sein können. Dies entspricht der folgenden Situation: Auf dem Kreis, den $e = (x, y)$

erzeugt, liegt ein Knoten v mit $d_T(v) = k$. Es gilt $d_T(x) = k - 1$ und Knoten x ist als durch $e' = (x', y')$ reduzierbar markiert. Bevor die lokale Verbesserung an v durch Einfügen von e durchgeführt werden kann, muss erst der Grad von x durch eine lokale Verschiebung auf $k-2$ reduziert werden. Ist auch x' als durch eine Kante $e'' = (x'', y'')$ reduzierbar markiert, so gilt $d_T(x') = k - 1$, das heißt, der Grad von x' würde durch die lokale Verschiebung auf k erhöht werden. Damit dies nicht passiert, führen wir zunächst an x' eine lokale Verschiebung durch Einfügen von e'' durch, die den Grad von x' um eins reduziert. Allerdings könnten auch x'' und/oder y'' markiert sein und Grad $k - 1$ haben. Dann müssen wir rekursiv weiter verfahren.

Theorem 7.8. *Der Algorithmus* ImprovedLocalMDST *terminiert nach polynomiell vielen Schritten mit einem Spannbaum T mit $\Delta(T) \leq \mathrm{OPT} + 1$.*

Beweis. Der Algorithmus terminiert nur dann, wenn die Bedingung aus Lemma 7.7 erfüllt ist. Damit folgt $\Delta(T) \leq \mathrm{OPT} + 1$ für den Spannbaum T, den der Algorithmus ausgibt, sofern er terminiert. Wir müssen nun nur noch zeigen, dass ImprovedLocalMDST stets nach polynomiell vielen Schritten terminiert.

Betrachten wir die Kaskade von rekursiven Aufrufen in Zeile 20 genauer. Wir werden die folgende Invariante per Induktion zeigen: Zu jedem Zeitpunkt gibt es für jeden markierten Knoten v eine Menge von lokalen Verschiebungen innerhalb der Zusammenhangskomponente aus $\mathcal{C}$, in der er sich befindet, die den Grad von v auf $k - 2$ reduziert und keine Knoten mit einem Grad größer als $k - 1$ erzeugt. Am Anfang ist diese Invariante trivialerweise erfüllt, da es noch keine markierten Knoten gibt.

Betrachten wir nun die Situation, dass ein Knoten v in Zeile 14 als durch $e = (x, y)$ reduzierbar markiert wird. Da e verschiedene Zusammenhangskomponenten aus $\mathcal{C}$ verbindet, liegen x und y in verschiedenen Zusammenhangskomponenten. Wir können somit die Induktionsvoraussetzung unabhängig auf x und y anwenden. Wir erhalten dann insgesamt eine Menge M von lokalen Verschiebungen, nach deren Ausführung x und y Grad höchstens $k - 2$ besitzen und kein Knotengrad auf mehr als $k - 1$ erhöht wurde. Nach diesen lokalen Verschiebungen kann die durch v und e beschriebene lokale Verschiebung durchgeführt werden. Da nach dem Markieren von v alle Knoten aus $C \cap D_{k-1}$ aus der Menge D_{k-1} entfernt und X und $\mathcal{C}$ entsprechend angepasst werden, werden die Zusammenhangskomponenten von x, y und v vereinigt. Damit gilt die Invariante weiterhin, denn alle Verschiebungen aus M sowie die durch v und e beschriebene Verschiebung finden in derselben Zusammenhangskomponente statt.

Nun folgt einfach, dass der Algorithmus polynomielle Laufzeit besitzt, denn in jeder Iteration der for-Schleife wird entweder mindestens ein Knoten markiert oder eine lokale Verbesserung durchgeführt, die die Teilphase abschließt. Die Zahl der Teilphasen und Phasen ist jeweils durch die Anzahl der Knoten des Graphen nach oben beschränkt. □

7.2 Facility Location ohne Kapazitäten

Wir kommen in diesem Abschnitt noch einmal auf das Problem Facility Location zurück. Für die metrische Variante dieses Problems haben wir in Abschnitt 5.2 einen 4-

Approximationsalgorithmus kennengelernt, der lineare Programmierung als Hilfsmittel einsetzt. Wir werden nun sehen, dass mithilfe von lokaler Suche ein deutlich besserer Approximationsfaktor erreicht werden kann.

Im Folgenden sei eine Eingabe (D, F, d, f) für Facility Location mit metrischen Abständen d gegeben. Eine Lösung ist eine nichtleere Menge $S \subseteq F$ von Standorten. Dabei wird in der Zielfunktion implizit davon ausgegangen, dass jeder Kunde dem nächstgelegenen Standort aus S zugewiesen wird. Wir werden im folgenden die Zuweisung explizit als Teil der Lösung ansehen, sodass eine Lösung in diesem Abschnitt als (S, σ) beschrieben wird, wobei $S \subseteq F$ die geöffneten Standorte und $\sigma : D \to S$ die Zuweisungen bezeichnet. Die Zielfunktion können wir dann als

$$\sum_{i \in S} f_i + \sum_{j \in D} d_{\sigma(j)j}$$

schreiben. Es ist klar, dass es für eine Menge S von Standorten stets optimal ist, jeden Kunden dem nächstgelegenen Standort aus S zuzuweisen. Der lokale Suchalgorithmus, den wir betrachten werden, wird auch stets diese optimale Zuweisung durchführen. Für die Analyse des Algorithmus ist es jedoch nützlich, auch die Möglichkeit zu haben, andere Zuweisungen zu betrachten.

Der lokale Suchalgorithmus startet mit einer beliebigen Lösung. In jedem Schritt testet er, ob die aktuelle Lösung (S, σ) eine lokale Verbesserung erlaubt und führt diese gegebenenfalls durch. Dabei gibt es die folgenden drei Arten von lokalen Verbesserungen.

1. Der Menge S wird ein weiterer Standort $i \in F \setminus S$ hinzugefügt. Die Zuweisung σ wird optimal für die Standortmenge $S \cup \{i\}$ gewählt.
2. Aus der Menge S wird ein Standort $i \in S$ gelöscht. Die Zuweisung σ wird optimal für die Standortmenge $S \setminus \{i\}$ gewählt.
3. Aus der Menge S wird ein Standort $i \in S$ gelöscht und ihr wird ein Standort $i' \in F \setminus S$ hinzugefügt. Die Zuweisung σ wird optimal für die Standortmenge $(S \setminus \{i\}) \cup \{i'\}$ gewählt.

Natürlich bezeichnen wir nur solche Schritte als lokale Verbesserungen, die die Kosten der Lösung reduzieren. Nur solche werden vom lokalen Suchalgorithmus durchgeführt. In Situationen, in denen nicht klar ist, ob die Zielfunktion durch einen Schritt sinkt, sprechen wir auch von einer lokalen Transformation anstatt von einer lokalen Verbesserung.

Für eine Lösung (S, σ) bezeichnen wir mit $F(S) = \sum_{i \in S} f_i$ die Kosten für das Eröffnen der Standorte und mit $D(\sigma) = \sum_{j \in D} d_{\sigma(j)j}$ die Kosten für das Beliefern der Kunden. Wir fixieren eine beliebige optimale Lösung (S^*, σ^*) für die gegebene Instanz und bezeichnen mit $F^* = F(S^*)$ die Kosten für das Eröffnen der Standorte in S^* und mit $D^* = D(\sigma^*) = \sum_{j \in D} \min_{i \in S^*} d_{ij}$ die Kosten für das Beliefern der Kunden. Dann gilt $\mathrm{OPT} = F^* + D^*$.

Wir werden nun zunächst für jede lokal optimale Lösung (S, σ) die Kosten für das Beliefern der Kunden beschränken.

Lemma 7.9. *Für jede lokal optimale Lösung (S, σ) gilt*

$$D(\sigma) \leq F^* + D^* = \mathrm{OPT}.$$

Beweis. Da die lokale Suche in jedem Schritt die Zuweisung σ so wählt, dass sie für die aktuelle Standortmenge S optimal ist, können wir davon ausgehen, dass jeder Kunde in der Zuweisung σ an den nächstgelegenen Standort aus S angeschlossen ist. Gilt $S = S^*$, so ist die Lösung (S, σ) demnach optimal. Wir betrachten im Folgenden nur noch den Fall, dass $S \neq S^*$ gilt.

Die lokale Optimalität der Lösung (S, σ) impliziert gewisse Ungleichungen, aus denen das Lemma folgt. Dafür betrachten wir zunächst einen beliebigen Standort $i \in S^* \setminus S$, sofern ein solcher existiert. Da die Lösung (S, σ) lokal optimal ist, gilt für keine der oben beschriebenen lokalen Transformationen, dass sie die Kosten der Lösung reduziert. Insbesondere ist es nicht vorteilhaft, den Standort i der Menge S hinzuzufügen und die Zuweisung σ entsprechend anzupassen. Wir betrachten eine Zuweisung $\sigma' : D \to S \cup \{i\}$ mit

$$\sigma'(j) = \begin{cases} i & \text{falls } \sigma^*(j) = i, \\ \sigma(j) & \text{sonst.} \end{cases}$$

Die Zuweisung σ' stimmt also im Wesentlichen mit der Zuweisung σ überein. Lediglich Kunden, die in der optimalen Lösung dem Standort i zugewiesen sind, werden in σ' ebenfalls dem Standort i zugewiesen. Die Zuordnung σ' ist im Allgemeinen keine optimale Zuordnung für die Standortmenge $S \cup \{i\}$. Die lokale Optimalität von (S, σ) impliziert, dass es selbst für eine optimale Zuweisung keine Verbesserung wäre, Standort i zu S hinzuzufügen. Dies bedeutet insbesondere, dass auch die Lösung $(S \cup \{i\}, \sigma')$ nicht besser sein kann als die Lösung (S, σ). Die Kosten f_i für das Eröffnen des zusätzlichen Standortes i sind also mindestens genauso groß wie die Kostenersparnis in der Zuordnung:

$$f_i \geq D(\sigma) - D(\sigma') = \sum_{j:\sigma^*(j)=i} (d_{\sigma(j)j} - d_{\sigma^*(j)j}). \tag{7.1}$$

Ungleichung (7.1) gilt auch für alle Standorte $i \in S \cap S^*$. Dies liegt daran, dass σ jeden Kunden dem nächstgelegenen Standort aus S zuweist. Wegen $i \in S$ gilt deshalb $d_{\sigma(j)j} \leq d_{\sigma^*(j)j}$ für alle Kunden j. Demzufolge ist jeder Summand auf der rechten Seite kleiner oder gleich 0. Wegen $f_i \geq 0$ ist die Ungleichung in diesem Fall erfüllt.

Summieren wir (7.1) über alle Standorte $i \in S^*$, so erhalten wir

$$\sum_{i \in S^*} f_i \geq \sum_{i \in S^*} \sum_{j:\sigma^*(j)=i} (d_{\sigma(j)j} - d_{\sigma^*(j)j}).$$

Da σ^* jeden Kunden genau einem Standort aus S^* zuordnet, können wir die Doppelsumme auf der rechten Seite wie folgt ersetzen:

$$\sum_{i \in S^*} f_i \geq \sum_{j \in D} (d_{\sigma(j)j} - d_{\sigma^*(j)j}).$$

Die linke Seite dieser Ungleichung entspricht F^*. Die rechte entspricht $D(\sigma) - D(\sigma^*) = D(\sigma) - D^*$. Es ergibt sich $F^* \geq D(\sigma) - D^*$, woraus das Lemma folgt. □

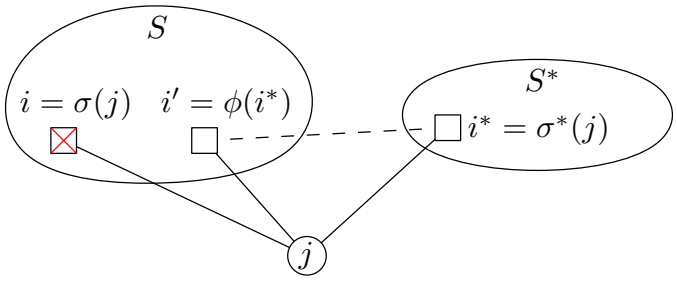


Abbildung 7.4: Wird Standort i geschlossen, so wird Kunde j an Standort $i' = \phi(\sigma^*(j))$ angeschlossen.

Wir müssen nun noch die Kosten für das Eröffnen der Standorte in einer lokal optimalen Lösung (S, σ) beschränken. Hierfür betrachten wir die lokalen Transformationen genauer, bei denen ein Standort $i \in S$ entfernt oder durch einen anderen Standort ersetzt wird. Zunächst beschreiben wir eine spezielle Möglichkeit, wie die Zuordnung σ nach dem Löschen von i angepasst werden kann. Betrachten wir dafür wieder die optimale Lösung (S^*, σ^*) und einen Kunden j mit $\sigma(j) = i$. Für diesen Kunden müssen wir einen neuen Standort finden. In der optimalen Lösung ist Kunde j an den Standort $i^* = \sigma^*(j)$ angebunden. Dieser gehört im Allgemeinen nicht zur Menge S. Deshalb betrachten wir einen Standort aus der Menge S, der den kleinsten Abstand zu i^* besitzt. Diesen nennen wir $i' = \phi(i^*)$. Ist $i \neq i'$, so können wir den Kostenanstieg begrenzen, der durch die Neuzuweisung von Kunde j zu Standort i' statt i entsteht. Abbildung 7.4 stellt die Situation dar.

Lemma 7.10. *Es sei $j \in D$ ein Kunde, $i = \sigma(j)$ und $i' = \phi(\sigma^*(j))$. Ferner gelte $i \neq i'$. Weisen wir Kunde j dem Standort i' statt dem Standort i zu, so erhöhen sich die Kosten der Zuweisung um höchstens $2d_{\sigma^*(j)j}$.*

Beweis. Der Beweis folgt im Wesentlichen durch mehrfache Anwendung der Dreiecksungleichung und Ausnutzung der Definition von ϕ. Aus der Wahl von ϕ folgt $d_{i'i^*} \leq d_{ii^*}$. Aus dieser Ungleichung folgt mit zweifacher Anwendung der Dreiecksungleichung

$$d_{i'j} \leq d_{i'i^*} + d_{i^*j} \leq d_{ii^*} + d_{i^*j} \leq (d_{ij} + d_{i^*j}) + d_{i^*j} = d_{ij} + 2d_{i^*j}.$$

Dies impliziert

$$d_{i'j} - d_{ij} \leq 2d_{i^*j},$$

was der Aussage des Lemmas entspricht. □

Mithilfe des vorangegangenen Lemmas können wir nun die folgende Aussage zeigen.

Lemma 7.11. *Für jede lokal optimale Lösung (S, σ) gilt*

$$F(S) \leq F^* + 2D^* \leq 2 \cdot \mathrm{OPT}.$$

Beweis. Die Beweisstrategie ist ähnlich zu der aus Lemma 7.9. Wir werden aus der lokalen Optimalität der Lösung (S, σ) Ungleichungen ableiten, aus denen die gewünschte Schranke für die Kosten $F(S)$ folgt.

Betrachten wir zunächst das Löschen eines Standortes $i \in S$. Alle Kunden, die von σ dem Standort i zugewiesen werden, müssen dann neu zugewiesen werden. Wir nennen einen Standort $i \in S$ *sicher*, wenn es keinen Standort $i^* \in S^*$ mit $\phi(i^*) = i$ gibt. Ein sicherer Standort i ist also für keinen Standort der optimalen Lösung S^* der nächstgelegene Standort aus S. Für jeden sicheren Standort i können wir Lemma 7.10 anwenden, um die Kosten, die durch das Schließen von Standort i entstehen, nach oben abzuschätzen. In Lemma 7.10 wird jeder Kunde $j \in D$ mit $\sigma(j) = i$ dem zu $\sigma^*(j)$ nächstgelegenen Standort aus S zugewiesen. Diesen bezeichnen wir mit $\phi(\sigma^*(j))$. Aus der lokalen Optimalität von (S, σ) folgt, dass die Kostenreduktion durch das Schließen eines Standortes geringer ist als der Kostenanstieg durch die veränderte Zuweisung. Aus Lemma 7.10 folgt für jeden sicheren Standort i somit

$$f_i \leq \sum_{j:\sigma(j)=i} d_{\phi(\sigma^*(j))j} - d_{ij} \leq \sum_{j:\sigma(j)=i} 2d_{\sigma^*(j)j}$$

oder äquivalent dazu

$$-f_i + \sum_{j:\sigma(j)=i} 2d_{\sigma^*(j)j} \geq 0. \tag{7.2}$$

Als nächstes betrachten wir Standorte, die unsicher sind. Sei i ein solcher Standort und sei $R = \{i^* \in S^* \mid \phi(i^*) = i\}$ die Menge der optimalen Standorte, für die i ein nächstgelegener Standort aus S ist. Sei ferner $i' \in R$ ein Standort, der von allen Standorten aus R den kleinsten Abstand zu i besitzt.

Für jeden Standort $i^* \in R \setminus \{i'\}$ betrachten wir die lokale Transformation, die i^* zu der Menge S hinzufügt. Da dies keine lokale Verbesserung ist, müssen die Kosten für das Hinzufügen von i^* größer sein als der Gewinn durch die verbesserte Zuweisung. Analog zu Lemma 7.9 folgt

$$f_{i^*} \geq \sum_{j:\sigma^*(j)=i^*,\sigma(j)=i} (d_{\sigma(j)j} - d_{\sigma^*(j)j}),$$

wobei die Summe auf der rechten Seite die Kostenreduktion angibt, die man erzielt, wenn man jeden Kunden j mit $\sigma^*(j) = i^*$ und $\sigma(j) = i$ Standort i^* zuweist. Äquivalent dazu ist die Ungleichung

$$f_{i^*} + \sum_{j:\sigma^*(j)=i^*,\sigma(j)=i} (d_{\sigma^*(j)j} - d_{\sigma(j)j}) \geq 0. \tag{7.3}$$

Nun betrachten wir noch die lokale Transformation, die Standort i gegen Standort i' tauscht. Dies ist natürlich nur dann sinnvoll, wenn $i \neq i'$ gilt. Um die Kosten für diese lokale Transformation abzuschätzen, müssen wir uns noch überlegen, wie sich die Zuweisung der Kunden zu den Standorten ändert. Wir arbeiten mit der folgenden (nicht notwendigerweise optimalen) Möglichkeit, die Zuweisung zu ändern: Für alle Kunden $j \in D$ mit $\sigma(j) \neq i$ behalten wir die Zuweisung bei. Jeder Kunde $j \in D$ mit $\sigma(j) = i$, für den $\sigma^*(j) \notin R$ gilt, wird Standort $\phi(\sigma^*(j))$ zugewiesen. Jeder Kunde $j \in D$ mit $\sigma(j) = i$, für den $\sigma^*(j) \in R$ gilt, wird Standort i' zugewiesen. Wir

nutzen wieder die lokale Optimalität der Lösung (S, σ), die besagt, dass die Kostenveränderung durch das Eröffnen von i' statt i plus die Kostenveränderung durch die neue Zuweisung nicht negativ sein kann:

$$f_{i'} - f_i + \sum_{j:\sigma(j)=i,\sigma^*(j)\notin R} (d_{\phi(\sigma^*(j))j} - d_{ij}) + \sum_{j:\sigma(j)=i,\sigma^*(j)\in R} (d_{i'j} - d_{ij}) \geq 0.$$

Jeden Term in der ersten Summe können wir mithilfe von Lemma 7.10 durch $2d_{\sigma^*(j)j}$ nach oben abschätzen. Dies impliziert

$$f_{i'} - f_i + \sum_{j:\sigma(j)=i,\sigma^*(j)\notin R} 2d_{\sigma^*(j)j} + \sum_{j:\sigma(j)=i,\sigma^*(j)\in R} (d_{i'j} - d_{ij}) \geq 0. \tag{7.4}$$

Diese Ungleichung gilt trivialerweise auch für $i = i'$, da sie dann zu der Aussage

$$\sum_{j:\sigma(j)=i,\sigma^*(j)\notin R} 2d_{\sigma^*(j)j} \geq 0$$

degeneriert.

Als nächstes addieren wir alle Ungleichungen, die wir für den unsicheren Standort i' erhalten haben, also (7.3) für jedes $i^* \in R \setminus \{i'\}$ sowie (7.4). Wir erhalten

$$\begin{aligned} &\sum_{i^*\in R\setminus\{i'\}} \left(f_{i^*} + \sum_{j:\sigma^*(j)=i^*,\sigma(j)=i} (d_{\sigma^*(j)j} - d_{\sigma(j)j}) \right) \\ &+ f_{i'} - f_i + \sum_{j:\sigma(j)=i,\sigma^*(j)\notin R} 2d_{\sigma^*(j)j} + \sum_{j:\sigma(j)=i,\sigma^*(j)\in R} (d_{i'j} - d_{ij}) \geq 0. \end{aligned} \tag{7.5}$$

In dem obigen Term kommen nur Kunden $j \in D$ mit $\sigma(j) = i$ vor. Wir zeigen, dass der Gesamtbeitrag eines jeden solchen Kunden j höchstens $2d_{\sigma^*(j)j}$ beträgt. Für Kunden j mit $\sigma^*(j) \notin R$ beträgt der Beitrag genau $2d_{\sigma^*(j)j}$. Gilt $\sigma^*(j) = i'$, so ergibt sich der Gesamtbeitrag zu $d_{i'j} - d_{ij} \leq 2d_{i'j}$. Gilt $i^* = \sigma^*(j) \in R \setminus \{i'\}$, so beträgt der Beitrag $d_{i^*j} + d_{i'j} - 2d_{ij}$. Aus der Wahl von i' folgt $d_{i'i} \leq d_{i^*i}$. Daraus folgt mit zweifacher Anwendung der Dreiecksungleichung

$$\begin{aligned} d_{i^*j} + d_{i'j} - 2d_{ij} &\leq d_{i^*j} + (d_{i'i} + d_{ij}) - 2d_{ij} \\ &\leq d_{i^*j} + (d_{i^*i} + d_{ij}) - 2d_{ij} \\ &\leq d_{i^*j} + ((d_{i^*j} + d_{ij}) + d_{ij}) - 2d_{ij} = 2d_{i^*j}. \end{aligned}$$

Insgesamt ergibt sich aus (7.5) somit

$$-f_i + \sum_{i^*\in R} f_{i^*} + \sum_{j:\sigma(j)=i} 2d_{\sigma^*(j)j} \geq 0. \tag{7.6}$$

Um den Beweis abzuschließen, genügt es nun (7.2) für alle sicheren Standorte $i \in S$ und (7.6) für alle unsicheren Standorte $i \in S$ zu addieren. Mit der Beobachtung, dass jeder Standort $i^* \in S^*$ für genau ein $i \in S$ in der Menge R enthalten ist, ergibt sich

$$-\sum_{i\in S} f_i + \sum_{i^*\in S^*} f_{i^*} + \sum_{j\in D} 2d_{\sigma^*(j)j} \geq 0,$$

was wir auch als $-F(S)+F^*+2D^* \geq 0$ formulieren können. Dies ist genau die Aussage des Lemmas. □

Fassen wir Lemma 7.9 und Lemma 7.11 zusammen, so erhalten wir das folgende Theorem.

Theorem 7.12. *Die Kosten jedes lokalen Optimums* (S, σ) *sind durch* $2F^* + 3D^* \leq 3 \cdot \mathrm{OPT}$ *nach oben beschränkt.*

Genau wie in Theorem 5.6 ist auffällig, dass die Vorfaktoren von F^* und D^* nicht identisch sind. Auch hier ist dies der Schlüssel zur Verbesserung der Analyse. Sei $I = (D, F, d, f)$ eine Eingabe für Facility Location. Wir modifizieren diese, indem wir die Kosten für das Eröffnen der Standorte mit einem Faktor $\mu > 0$ skalieren. Konkret setzen wir $\tilde{f}_i = f_i/\mu$ für jedes $i \in F$ und betrachten die Instanz $I' = (D, F, d, \tilde{f})$. Die optimale Lösung für die Instanz I hat bezüglich I' dieselben Kosten D^* für das Beliefern der Kunden und Kosten von F^*/μ für das Eröffnen der Standorte (sie ist aber im Allgemeinen nicht optimal für I'). Gemäß Lemma 7.9 besitzt jedes lokale Optimum (S, σ) der Instanz I' deshalb Belieferungskosten von höchstens $F^*/\mu + D^*$ und gemäß Lemma 7.11 Eröffnungskosten von höchstens $F^*/\mu + 2D^*$[1]. Die Eröffnungskosten der Lösung (S, σ) bezüglich f betragen dann höchstens $\mu(F^*/\mu + 2D^*)$. Da sich die Belieferungskosten in I und I' nicht unterscheiden, besitzt jede Lösung (S, σ), die lokal optimal in I' ist, somit in I Kosten von höchstens

$$D(\sigma) + F(S) \leq \left(\frac{F^*}{\mu} + D^*\right) + \mu\left(\frac{F^*}{\mu} + 2D^*\right) = \left(1 + \frac{1}{\mu}\right) F^* + (1 + 2\mu)D^*. \quad (7.7)$$

Für $\mu = 1/\sqrt{2}$ betragen die Kosten höchstens $(1 + \sqrt{2})(F^* + D^*) \leq 2.415 \cdot \mathrm{OPT}$.

Leider können wir aus diesen Überlegungen aber noch nicht den Schluss ziehen, dass lokale Suche zu einem $(1+\sqrt{2})$-Approximationsalgorithmus führt, da exponentiell viele lokale Verbesserungen benötigt werden könnten, um ein lokales Optimum zu erreichen. Deshalb schränken wir uns auf Instanzen ein, in denen alle Kosten f_i und Distanzen d_{ij} ganzzahlig sind und wir modifizieren den lokalen Suchalgorithmus so, dass nur noch lokale Verbesserungen durchgeführt werden, die den Wert der Zielfunktion um mindestens einen Faktor $(1 - \delta) < 1$ reduzieren. Wir nennen den resultierenden Algorithmus LocalFacilityLocation. Betragen die initialen Kosten M, so betragen die Kosten nach k Schritten der lokalen Suche höchstens $(1 - \delta)^k M$. Ist k so groß gewählt, dass dieser Term kleiner als eins ist, so muss er wegen der Ganzzahligkeit null sein. Danach können keine weiteren lokalen Verbesserungen mehr durchgeführt werden. Allerdings gilt Theorem 7.12 nicht mehr, da wir nur noch eine Teilmenge der lokalen Verbesserungen erlauben. Man kann allerdings die Beweise von Lemma 7.9 und Lemma 7.11 an die veränderte Situation anpassen, indem man den Faktor $(1 - \delta)$ in den entsprechenden Ungleichungen berücksichtigt. Man erhält dann nach einigen Rechnungen die abgeschwächten Ungleichungen

$$D(\sigma) \leq F^* + D^* + |F|\delta(D(\sigma) + F(S)) \quad (7.8)$$

und

$$F(S) \leq F^* + 2D^* + |F|\delta(D(\sigma) + F(S)). \quad (7.9)$$

[1]Dies gilt, da wir in den Beweisen der beiden Lemmas nicht ausgenutzt haben, dass (S^*, σ^*) eine optimale Lösung ist. Sämtliche Argumente lassen sich auch auf suboptimale Lösungen anwenden.

Theorem 7.13. *Mithilfe des Algorithmus* LocalFacilityLocation *kann in polynomieller Zeit für jedes $\varepsilon > 0$ eine $(1 + \sqrt{2} + \varepsilon)$-Approximation für das metrische Facility Location Problem berechnet werden.*

Beweis. Aus den abgeschwächten Ungleichungen (7.8) und (7.9) ergibt sich für jedes lokale Optimum (S, σ) die Ungleichung

$$(1 - 2|F|\delta)(D(\sigma) + F(S)) \leq 2F^* + 3D^*$$

und damit eine obere Schranke für den Approximationsfaktor von $3/(1 - 2|F|\delta)$. Für $\delta = \varepsilon/(12|F|)$ gilt

$$\frac{3}{1 - 2|F|\delta} = \frac{3}{1 - \varepsilon/6} = \frac{3}{1 - (\varepsilon/3)/2} \leq 3\left(1 + \frac{\varepsilon}{3}\right) = 3 + \varepsilon,$$

wobei wir für die Ungleichung ausgenutzt haben, dass $1/(1 - x/2) \leq 1 + x$ für alle $x \in [0, 1]$ gilt. Skalieren wir wieder die Eröffnungskosten mit $\sqrt{2}$, so können wir den Approximationsfaktor analog zu (7.7) auf $1 + \sqrt{2} + \varepsilon$ verbessern.

Wir haben oben bereits argumentiert, dass jedes $k \in \mathbb{N}$ mit $(1 - \delta)^k M < 1$ eine obere Schranke für die Anzahl der lokalen Verbesserungen ist. Sei $k = \alpha/\delta$ für ein $\alpha > 0$. Dann gilt

$$(1 - \delta)^k M = (1 - \delta)^{\alpha/\delta} M \leq e^{-\alpha} M.$$

Für $\alpha = \ln(M) + 1$ ist dieser Term kleiner als 1. Die Zahl der lokalen Verbesserungen ist also durch

$$\frac{\ln(M) + 1}{\delta} = \frac{12|F|(\ln(M) + 1)}{\varepsilon}$$

nach oben beschränkt. Wir können die Kosten M der Startlösung zum Beispiel durch die Summe aller f_i und d_{ij} nach oben beschränken, weshalb dies für jede Konstante $\varepsilon > 0$ eine polynomielle Schranke ist. □

Kapitel 8

Primal-Duale Algorithmen

In diesem Kapitel lernen wir mit primal-dualen Algorithmen noch eine weitere grundlegende Technik zum Entwurf von Approximationsalgorithmen kennen, die bei vielen Problemen zum Einsatz kommt. Um diese Technik zu verstehen, müssen wir zunächst definieren, was wir unter einem dualen linearen Programm verstehen. Anschließend werden wir die Technik einsetzen, um Approximationsalgorithmen für Set Cover, Feedback Vertex Set und das Steinerwaldproblem zu entwerfen.

8.1 Duale Lineare Programme

Zunächst müssen wir definieren und verstehen, was das duale Programm eines linearen Programms ist. Betrachten wir dafür das folgende lineare Programm als Beispiel.

$$\begin{array}{rlrrrrr} \text{minimiere} & f(x) = 4x_1 + & 7x_2 + & 2x_3 & & & \\ \text{sodass} & 3x_1 + & 7x_2 + & x_3 & \geq & 10 & \\ & 2x_1 + & 5x_2 + & 3x_3 & \geq & 6 & \\ & 2x_1 + & x_2 - & x_3 & \geq & 5 & \\ & 4x_1 + & x_2 + & 3x_3 & \geq & 20 & \\ & 4x_1 + & 13x_2 + & x_3 & \geq & 4 & \\ & x_1, x_2, x_3 \geq 0 & & & & & \end{array} \tag{8.1}$$

Wir suchen nach einer Möglichkeit, untere Schranken für den optimalen Wert dieses linearen Programms zu zeigen. Aus den Nebenbedingungen können wir verschiedene Schranken herleiten. Die erste Nebenbedingung liefert uns direkt eine Schranke, denn da die Variablen nur nichtnegative Werte annehmen dürfen, gilt für alle Lösungen $x = (x_1, x_2, x_3)$ des linearen Programms

$$\begin{aligned} f(x) &= 4x_1 + 7x_2 + 2x_3 \\ &\geq 3x_1 + 7x_2 + x_3 \geq 10. \end{aligned}$$

Somit kann auch die optimale Lösung keinen kleineren Wert als 10 haben. Wir erhalten eine bessere Schranke, indem wir die zweite und dritte Nebenbedingung addieren:

$$\begin{aligned} f(x) &= 4x_1 + 7x_2 + 2x_3 \\ &\geq 4x_1 + 6x_2 + 2x_3 \\ &= (2x_1 + 5x_2 + 3x_3) + (2x_1 + x_2 - x_3) \\ &\geq 6 + 5 = 11. \end{aligned}$$

Eine noch bessere Schranke erhalten wir, wenn wir die vierte und fünfte Ungleichungen jeweils mit 1/2 gewichtet addieren:

$$\begin{aligned} f(x) &= 4x_1 + 7x_2 + 2x_3 \\ &= \frac{1}{2}(4x_1 + x_2 + 3x_3) + \frac{1}{2}(4x_1 + 13x_2 + x_3) \\ &\geq \frac{1}{2}(20 + 4) = 12. \end{aligned}$$

Zur Herleitung dieser unteren Schranken haben wir jeweils eine gewichtete Summe der Nebenbedingungen gebildet. Dabei haben wir die Gewichte so gewählt, dass für jede Variable x_i der Vorfaktor kleiner oder gleich dem Vorfaktor in der Zielfunktion ist. Die untere Schranke ergibt sich dann aus der entsprechend gewichteten Summe der rechten Seiten. Haben wir mit 12 die bestmögliche untere Schranke gefunden, die sich mit dieser Methode herleiten lässt? Interessanterweise können wir das Finden einer unteren Schranke wieder als lineares Programm formulieren. Dazu führen wir für jede Nebenbedingung eine Variable ein, die sagt, mit welcher Gewichtung sie in die untere Schranke eingeht. In unserem Beispiel erhalten wir fünf Variablen $y_1, \ldots, y_5 \geq 0$ und die bestmögliche untere Schranke, die man mit dieser Methode herleiten kann, lässt sich mit dem folgenden linearen Programm bestimmen.

$$\begin{array}{rrrrrrrrr} \text{maximiere} & g(y) = 10y_1 + & 6y_2 + & 5y_3 + & 20y_4 + & 4y_5 & & & \\ \text{sodass} & 3y_1 + & 2y_2 + & 2y_3 + & 4y_4 + & 4y_5 & \leq & 4 & \\ & 7y_1 + & 5y_2 + & y_3 + & y_4 + & 13y_5 & \leq & 7 & (8.2) \\ & y_1 + & 3y_2 - & y_3 + & 3y_4 + & y_5 & \leq & 2 & \\ & & y_1, \ldots, y_5 \geq 0 & & & & & & \end{array}$$

Sei $y = (y_1, \ldots, y_5)$ eine Lösung dieses linearen Programms mit einem Zielfunktionswert $g(y)$ und sei $x = (x_1, x_2, x_3)$ eine Lösung von (8.1) mit einem Zielfunktionswert $f(x)$. Dann gilt

$$\begin{aligned} f(x) &= 4x_1 + 7x_2 + 2x_3 \\ &\geq (3y_1 + 2y_2 + 2y_3 + 4y_4 + 4y_5)x_1 \\ &\quad + (7y_1 + 5y_2 + y_3 + y_4 + 13y_5)x_2 \\ &\quad + (y_1 + 3y_2 - y_3 + 3y_4 + y_5)x_3 \\ &= (3x_1 + 7x_2 + x_3)y_1 + (2x_1 + 5x_2 + 3x_3)y_2 + (2x_1 + x_2 - x_3)y_3 \\ &\quad + (4x_1 + x_2 + 3x_3)y_4 + (4x_1 + 13x_2 + x_3)y_5 \end{aligned}$$

$$\geq 10y_1 + 6y_2 + 5y_3 + 20y_4 + 4y_5 = g(y).$$

Somit ist der Wert $g(y)$ jeder Lösung y von (8.2) eine untere Schranke für den Wert jeder Lösung x von (8.1). Insbesondere gilt dies für den Wert $g(y^*)$ der optimalen Lösung y^* von (8.2).

Die optimale Lösung von (8.2) ist $y^* = \left(0, 0, \frac{2}{5}, \frac{4}{5}, 0\right)$. Es folgt für jede Lösung x von (8.1)

$$\begin{aligned} f(x) &= 4x_1 + 7x_2 + 2x_3 \\ &\geq 4x_1 + \frac{6}{5}x_2 + 2x_3 \\ &= \frac{2}{5}(2x_1 + x_2 - x_3) + \frac{4}{5}(4x_1 + x_2 + 3x_3) \\ &\geq \frac{2}{5} \cdot 5 + \frac{4}{5} \cdot 20 = 18. \end{aligned}$$

Wir haben einen Beweis dafür erhalten, dass der optimale Wert von (8.1) nicht kleiner als 18 sein kann. Interessanterweise ist dies in diesem Beispiel sogar eine scharfe Schranke, d. h. es gibt eine Lösung x für (8.1) mit Wert 18 nämlich $x = \left(\frac{7}{2}, 0, 2\right)$.

Das lineare Programm (8.2) wird als das zu (8.1) *duale lineare Programm* bezeichnet. Zur Unterscheidung wird das lineare Programm (8.1) dann auch als *primales lineares Programm* bezeichnet. Wir verallgemeinern dieses Konzept nun von dem konkreten Beispiel auf allgemeine lineare Programme in einer bestimmten Form. Sei dazu das folgende lineare Programm mit n Variablen $x_1, \ldots, x_n$ und m Nebenbedingungen gegeben:

$$\begin{aligned} \text{minimiere} \quad & f(x) = \sum_{j=1}^{n} c_j x_j \\ \text{sodass} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad \forall i \in \{1, \ldots, m\}, \\ & x_j \geq 0, \qquad\qquad \forall j \in \{1, \ldots, n\}. \end{aligned} \tag{8.3}$$

Genauso wie in obigem Beispiel können wir auch für dieses lineare Programm untere Schranken für den optimalen Zielfunktionswert zeigen, indem wir gewichtete Summen von Nebenbedingungen betrachten. Wir führen dazu m Variablen $y_1, \ldots, y_m \geq 0$ ein, die die Gewichtungen angeben, mit denen die Nebenbedingungen in die untere Schranke eingehen. Als untere Schranke ergibt sich dann die entsprechend gewichtete Summe der rechten Seiten. Genau wie im Beispiel müssen wir sicher stellen, dass in der gewichteten Summe der Nebenbedingungen der Vorfaktor jeder Variable x_j höchstens c_j ist. Daraus ergeben sich n Nebenbedingungen. Insgesamt erhalten wir das folgende duale lineare Programm:

$$\begin{aligned} \text{maximiere} \quad & g(y) = \sum_{i=1}^{m} b_i y_i \\ \text{sodass} \quad & \sum_{i=1}^{m} a_{ij} y_i \leq c_j, \quad \forall j \in \{1, \ldots, n\}, \\ & y_i \geq 0, \qquad\qquad \forall i \in \{1, \ldots, m\}. \end{aligned} \tag{8.4}$$

Wir erhalten die folgende Aussage, die als *schwache Dualität* bekannt ist.

Theorem 8.1. *Sei $x = (x_1, \ldots, x_n)$ eine Lösung für das primale lineare Programm* (8.3) *und sei $y = (y_1, \ldots, y_m)$ eine Lösung für das duale lineare Programm* (8.4)*. Dann gilt $f(x) \geq g(y)$.*

Beweis. Es gilt

$$f(x) = \sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i = g(y), \tag{8.5}$$

wobei die erste Ungleichung daraus folgt, dass y eine Lösung für (8.4) ist, und die zweite Ungleichung daraus, dass x eine Lösung für (8.3) ist. □

Für die Analyse und den Entwurf der Approximationsalgorithmen in diesem Kapitel spielt Theorem 8.1 eine entscheidende Rolle. Eine bemerkenswerte Eigenschaft, die für viele Bereiche der Optimierung von großer Wichtigkeit ist und die wir deshalb der Vollständigkeit halber erwähnen, ist die folgende *starke Dualität*.

Theorem 8.2. *Das primale lineare Programm* (8.3) *hat genau dann einen beschränkten optimalen Wert, wenn dies für das duale lineare Programm* (8.4) *der Fall ist. Sei ferner x^* eine optimale Lösung für das primale lineare Programm und sei y^* eine optimale Lösung für das duale lineare Programm, dann gilt $f(x^*) = g(y^*)$.*

Es ist also kein Zufall, dass die untere Schranke, die wir mithilfe des dualen LPs in unserem Beispiel gezeigt haben, scharf ist. Dies ist vielmehr immer der Fall.

Theorem 8.1 und 8.2 implizieren die folgenden *Komplementaritätsbedingungen*.

Theorem 8.3. *Sei $x = (x_1, \ldots, x_n)$ eine Lösung für das primale lineare Programm* (8.3) *und sei $y = (y_1, \ldots, y_m)$ eine Lösung für das duale lineare Programm* (8.4)*. Die Lösungen x und y sind genau dann optimale Lösungen für das primale bzw. duale lineare Programm, wenn die folgenden beiden Bedingungen gelten:*

$$\forall j \in \{1, \ldots, n\} : x_j > 0 \Rightarrow \sum_{i=1}^{m} a_{ij} y_i = c_j \tag{8.6}$$

und

$$\forall i \in \{1, \ldots, m\} : y_i > 0 \Rightarrow \sum_{j=1}^{n} a_{ij} x_j = b_i. \tag{8.7}$$

Beweis. Seien x und y optimale Lösungen. Dann gilt $f(x) = g(y)$ gemäß Theorem 8.2. Aus der Ungleichungskette (8.5) wird also eine Gleichungskette. Die beiden Ungleichungen sind aber nur dann mit Gleichheit erfüllt, wenn die Bedingungen (8.6) und (8.7) gelten.

Seien x und y beliebige Lösungen, die die Bedingungen (8.6) und (8.7) erfüllen, so folgt umgekehrt aus (8.5), dass $f(x) = g(y)$ gilt. Aus Theorem 8.1 folgt, dass dies nur dann möglich ist, wenn x und y optimal sind. □

8.2 Set Cover

Wir betrachten noch einmal das Set Cover Problem aus Abschnitt 2.2. Sei eine Eingabe mit Grundmenge $S = \{1, \ldots, n\}$, mit Teilmengen $S_1, \ldots, S_m$ und mit Kosten $w_1, \ldots, w_m \in \mathbb{R}_{>0}$ gegeben. Es gelte $\bigcup_{j=1}^{m} S_j = S$. Wir können das Problem, eine optimale Lösung zu finden, als ganzzahliges lineares Programm formulieren:

$$\begin{aligned} \text{minimiere } & \sum_{j=1}^{m} w_j x_j \\ \text{sodass } & \sum_{j:i\in S_j} x_j \geq 1, \quad \forall i \in \{1, \ldots, n\}, \\ & x_j \in \{0, 1\}, \quad \forall j \in \{1, \ldots, m\}. \end{aligned}$$

In diesem linearen Programm codieren die binären Variablen $x_1, \ldots, x_m$, welche Teilmengen ausgewählt werden und die Nebenbedingungen stellen sicher, dass jedes Element aus S in mindestens einer ausgewählten Menge enthalten ist. Relaxieren wir dieses ganzzahlige Programm, so erhalten wir das folgende lineare Programm:

$$\begin{aligned} \text{minimiere } & \sum_{j=1}^{m} w_j x_j \\ \text{sodass } & \sum_{j:i\in S_j} x_j \geq 1, \quad \forall i \in \{1, \ldots, n\}, \\ & x_j \geq 0, \qquad \forall j \in \{1, \ldots, m\}. \end{aligned} \tag{8.8}$$

Wir haben in dieser Relaxierung die Bedingung $x_j \leq 1$ weggelassen, da diese in der optimalen Lösung des linearen Programms automatisch erfüllt ist. Als nächstes stellen wir das zu (8.8) duale lineare Programm gemäß dem in Abschnitt 8.1 beschriebenen Verfahren auf:

$$\begin{aligned} \text{maximiere } & \sum_{i=1}^{n} y_i \\ \text{sodass } & \sum_{i\in S_j} y_i \leq w_j, \quad \forall j \in \{1, \ldots, m\}, \\ & y_i \geq 0, \qquad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{8.9}$$

Erfahrungsgemäß ist es schwierig, eine Intuition dafür zu gewinnen, warum ein lineares Programm das duale eines anderen ist. Wir können in diesem Beispiel auch keine gute Intuition dafür geben, warum gerade (8.9) das zu (8.8) duale Programm ist. Hier muss man sich in der Regel auf die rein mechanische Übersetzung aus Abschnitt 8.1 verlassen. Erhält man dadurch das duale lineare Programm, so ist es jedoch sinnvoll, sich zu überlegen, welchem Problem es entspricht. Man kann (8.9) so interpretieren, dass man jedem Objekt $i \in S$ aus der Grundmenge einen Preis y_i zuweisen möchte. Die Summe aller Preise soll möglichst groß werden, wobei jede Teilmenge S_j die Nebenbedingung induziert, dass der Gesamtpreis der in S_j enthaltenen Objekte höchstens w_j betragen darf.

Wir betrachten nun den folgenden Algorithmus, der gleichzeitig eine Lösung I für Set Cover und eine Lösung y des dualen linearen Programms konstruiert.

PrimalDual-SC

1. $y := 0$; // gültige Lösung des dualen LPs
2. $I := \emptyset$;
3. **while** $(\exists i \notin \bigcup_{j \in I} S_j)$
4. Erhöhe y_i solange, bis für ein j mit $i \in S_j$ gilt $\sum_{\ell \in S_j} y_\ell = w_j$.
5. $I := I \cup \{j\}$;
6. **return** I

Die Bedingung in der while-Schleife stellt sicher, dass der Algorithmus erst terminiert, wenn durch die Auswahl I alle Objekte abgedeckt sind. Sind noch nicht alle Objekte abgedeckt, so wird ein Objekt i gewählt, das noch nicht abgedeckt ist. Für dieses wird in Zeile 4 eine Menge S_j mit $i \in S_j$ ausgewählt und in Zeile 5 wird diese Menge der Auswahl I hinzugefügt. Zur Auswahl der Menge S_j wird die kleinste Erhöhung von y_i bestimmt, die dazu führt, dass eine der Nebenbedingungen des dualen LPs mit Gleichheit erfüllt ist (diese Erhöhung kann auch null sein). Da y_i nicht über diesen Wert hinaus erhöht wird, ist sichergestellt, dass y zu jedem Zeitpunkt eine gültige Lösung des dualen LPs ist.

Theorem 8.4. *Es bezeichne $f = \max_i |\{j \mid i \in S_j\}|$ die maximale Anzahl an Vorkommen eines Elementes in den Teilmengen. Der Algorithmus* PrimalDual-SC *ist ein f-Approximationsalgorithmus für Set Cover.*

Beweis. Es bezeichne OPT den Wert einer optimalen Lösung für die gegebene Instanz von Set Cover und es bezeichne Z^* den optimalen Wert des primalen LPs (8.8). Dann gilt $Z^* \leq \text{OPT}$. Es bezeichne y die Lösung des dualen LPs, die PrimalDual-SC konstruiert. Da jede Lösung des dualen LPs eine untere Schranke für den optimalen Wert des primalen LPs impliziert, gilt $\sum_{i=1}^{n} y_i \leq Z^*$. Für den Beweis des Theorems genügt es also zu zeigen, dass $\sum_{j \in I} w_j \leq f \cdot \sum_{i=1}^{n} y_i$ gilt, denn dann folgt insgesamt

$$\sum_{j \in I} w_j \leq f \cdot \sum_{i=1}^{n} y_i \leq f \cdot Z^* \leq f \cdot \text{OPT}.$$

Aus dem Algorithmus folgt, dass wir eine Menge S_j nur dann zu der Auswahl I hinzufügen, wenn die Bedingung $\sum_{i \in S_j} y_i = w_j$ erfüllt ist. Dies impliziert

$$\sum_{j \in I} w_j = \sum_{j \in I} \sum_{i \in S_j} y_i = \sum_{i \in S} y_i \cdot |\{j \in I \mid i \in S_j\}| \leq f \cdot \sum_{i \in S} y_i,$$

woraus das Theorem wie oben diskutiert folgt. □

Algorithmen, die nach dem Schema von PrimalDual-SC ablaufen, nennt man *primal-duale Algorithmen*. Diese haben gemeinsam, dass sie stets eine gültige Lösung des dualen LPs mitführen und dabei eine (im Allgemeinen nicht optimale) ganzzahlige Lösung für das primale LP konstruieren. Dabei werden die Lösungen für das primale und duale LP vom Algorithmus explizit konstruiert, ohne auf allgemeine Algorithmen zur Lösung

von linearen Programmen zurückzugreifen. Die Dualitätstheorie ist wesentlich für die Analyse der Algorithmen.

Alternativ kann die Kostenabschätzung der von PrimalDual-SC berechneten Lösung auch über die im Folgenden dargestellte abgeschwächte Komplementaritätsbedingung erfolgen. Die Lösung I, die der Algorithmus PrimalDual-SC konstruiert, entspricht einer ganzzahligen Lösung $x \in \{0,1\}^m$ des primalen LPs (8.8). Für diese Lösung x und die berechnete Lösung y des dualen LPs stellt der Algorithmus die folgende Bedingung sicher:

$$\forall j \in \{1,\ldots,m\} : x_j > 0 \Rightarrow \sum_{i \in S_j} y_i = w_j.$$

Dies liegt daran, dass ein j nur dann der Menge I hinzugefügt wird, wenn die Gleichung $\sum_{i \in S_j} y_i = w_j$ erfüllt ist. Da jedes Objekt i in höchstens f Mengen enthalten ist und jedes x_j aus $\{0,1\}$ gewählt wird, folgt ebenfalls die Bedingung

$$\forall i \in \{1,\ldots,n\} : y_i > 0 \Rightarrow \sum_{j : i \in S_j} x_j \leq f.$$

Dies erinnert an die Komplementaritätsbedingungen aus Theorem 8.3. Tatsächlich können wir allgemein das folgende Ergebnis beweisen.

Theorem 8.5. *Sei $x = (x_1,\ldots,x_n)$ eine Lösung für das primale lineare Programm* (8.3) *und sei $y = (y_1,\ldots,y_m)$ eine Lösung für das duale lineare Programm* (8.4)*. Ferner seien $\alpha \geq 1$ und $\beta \geq 1$ so gewählt, dass die folgenden beiden Bedingungen gelten:*

$$\forall j \in \{1,\ldots,n\} : x_j > 0 \Rightarrow \frac{c_j}{\alpha} \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j$$

und

$$\forall i \in \{1,\ldots,m\} : y_i > 0 \Rightarrow b_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq \beta \cdot b_i.$$

Dann gilt $f(x) \leq \alpha\beta \cdot g(y)$.

Beweis. Analog zu (8.5) folgt

$$\begin{aligned} f(x) = \sum_{j=1}^{n} c_j x_j &\leq \alpha \cdot \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \alpha \cdot \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \\ &\leq \alpha\beta \cdot \sum_{i=1}^{m} b_i y_i = \alpha\beta \cdot g(y). \end{aligned}$$

□

Theorem 8.4 folgt aus Theorem 8.5 mit $\alpha = 1$ und $\beta = f$.

8.3 Feedback Vertex Set

Wir betrachten nun das NP-schwere Problem Feedback Vertex Set. Dabei ist ein ungerichteter Graph mit Knotengewichten gegeben und das Ziel ist es, eine Teilmenge von Knoten mit möglichst geringem Gewicht zu entfernen, sodass der resultierende Graph kreisfrei ist. In der folgenden Definition nutzen wir für einen Graphen $G = (V, E)$ und eine Menge $S \subseteq V$ die Notation $G[V \setminus S]$. Damit ist der von $V \setminus S$ induzierte Teilgraph von G gemeint, also der Graph, der aus G hervorgeht, indem alle Knoten aus S und alle zu S inzidenten Kanten gelöscht werden.

Feedback Vertex Set

Eingabe:	ungerichteter Graph $G = (V, E)$, Knotengewichte $w : V \to \mathbb{R}_{>0}$
Lösungen:	alle $S \subseteq V$ sodass $G[V \setminus S]$ kreisfrei ist
Zielfunktion:	minimiere $\sum_{v \in S} w(v)$

Die Bedingung, dass $G[V \setminus S]$ kreisfrei ist, kann man äquivalent auch so formulieren, dass jeder Kreis im Graphen G mindestens einen Knoten aus der Menge S enthalten muss. Es bezeichne $\mathcal{C}$ die Menge aller Kreise im Graphen G. Ferner sei $w_i = w(i)$ für $i \in V$. Dann ergibt sich die folgende Formulierung von Feedback Vertex Set als ganzzahliges lineares Programm:

$$\begin{aligned} \text{minimiere } & \sum_{i \in V} w_i x_i \\ \text{sodass } & \sum_{i \in C} x_i \geq 1, \quad \forall C \in \mathcal{C}, \\ & x_i \in \{0, 1\}, \quad \forall i \in V. \end{aligned} \tag{8.10}$$

In diesem ganzzahligen linearen Programm gibt es eine binäre Variable x_i für jeden Knoten $i \in V$, die angibt, ob der Knoten i zu der Menge S gehört. Die Nebenbedingungen stellen sicher, dass für jeden Kreis im Graphen mindestens ein Knoten ausgewählt wird. Im Unterschied zu den linearen Programmen, die wir bisher in der Vorlesung kennengelernt haben, ist die Anzahl an Nebenbedingungen in (8.10) im Allgemeinen nicht polynomiell in der Eingabelänge beschränkt, da es in einem Graphen exponentiell viele Kreise geben kann. Dies ist an dieser Stelle kein Problem, da wir in dem primal-dualen Algorithmus, den wir betrachten werden, das lineare Programm nie explizit lösen müssen. Wir relaxieren das ganzzahlige LP, indem wir die Bedingung $x_i \in \{0, 1\}$ durch die Bedingung $x_i \geq 0$ ersetzen. Dann ergibt sich das folgende duale LP:

$$\begin{aligned} \text{maximiere } & \sum_{C \in \mathcal{C}} y_C \\ \text{sodass } & \sum_{C : i \in C} y_C \leq w_i, \quad \forall i \in V, \\ & y_C \geq 0, \qquad \forall C \in \mathcal{C}. \end{aligned} \tag{8.11}$$

Im dualen LP gibt es für jede Nebenbedingung des primalen LPs eine Variable, also für jeden Kreis $C \in \mathcal{C}$. Im dualen LP ist demnach die Anzahl an Variablen im Allgemeinen

nicht polynomiell beschränkt. Der primal-duale Algorithmus wird aber stets nur eine polynomielle Zahl an Variablen auf einen Wert größer als null setzen.

Der folgende Algorithmus ergibt sich als fast kanonische Anpassung aus dem Algorithmus PrimalDual-SC. Lediglich der Grund für die Zeilen 3 und 8 ist nicht direkt ersichtlich. Das Entfernen von Knoten von Grad 1 ist für den Algorithmus zunächst irrelevant, da diese Knoten ohnehin auf keinem Kreis vorkommen können. Es wird sich allerdings später als nützlich herausstellen. Es sei noch angemerkt, dass das Entfernen eines Knotens mit Grad 1 einen neuen Knoten mit Grad 1 erzeugen kann. Dieser wird dann ebenfalls gelöscht usw. Nach dem Ausführen von Zeile 3 oder 8 ist also sichergestellt, dass der Graph G keinen Knoten mit Grad 1 mehr enthält.

PrimalDual-FVS

1. $y := 0$; // gültige Lösung des dualen LPs
2. $S := \emptyset$;
3. Entferne Knoten mit Grad 1 aus G, bis kein solcher Knoten mehr existiert.
4. **while** ($\exists$ Kreis C^* in G)
5. Erhöhe y_{C^*} solange, bis $\sum_{C:i\in C} y_C = w_i$ für ein $i \in C^*$ gilt.
6. $S := S \cup \{i\}$;
7. Entferne Knoten i und alle zu i inzidenten Kanten aus G.
8. Entferne Knoten mit Grad 1 aus G, bis kein solcher Knoten mehr existiert.
9. **return** S

Wir beginnen die Analyse analog zu der des Algorithmus PrimalDual-SC. Es bezeichne OPT den Wert einer optimalen Lösung für die gegebene Instanz von Feedback Vertex Set und es bezeichne Z^* den optimalen Wert des primalen LPs, also der relaxierten Variante von (8.8). Dann gilt $Z^* \leq \text{OPT}$. Es bezeichne y die Lösung des dualen LPs, die PrimalDual-FVS konstruiert. Da jede Lösung des dualen LPs eine untere Schranke für den optimalen Wert des primalen LPs impliziert, gilt $\sum_{C\in\mathcal{C}} y_C \leq Z^*$. Aus dem Algorithmus folgt, dass wir einen Knoten i nur dann zu der Menge S hinzufügen, wenn die Bedingung $\sum_{C:i\in C} y_C = w_i$ erfüllt ist. Dies impliziert

$$\sum_{i\in S} w_i = \sum_{i\in S} \sum_{C:i\in C} y_C = \sum_{C\in\mathcal{C}} y_C \cdot |S \cap C|.$$

Der Term $|S \cap C|$ gibt an, wie viele Knoten des Kreises C in der Lösung S ausgewählt wurden. Ist dies für jeden Kreis C mit $y_C > 0$ durch α nach oben beschränkt, so ergibt sich

$$\sum_{i\in S} w_i \leq \alpha \cdot \sum_{C\in\mathcal{C}} y_C \leq \alpha \cdot Z^* \leq \alpha \cdot \text{OPT},$$

da der Wert jeder Lösung y des dualen Programms eine untere Schranke für Z^* ist. Der Algorithmus berechnet dann also eine α-Approximation. Leider kann ohne Weiteres keine gute obere Schranke für $|S \cap C|$ angegeben werden, da es Kreise geben kann, von denen sehr viele Knoten ausgewählt werden.

Um einen guten Approximationsfaktor zu erhalten, müssen wir den Algorithmus ändern. Statt in Zeile 5 eine beliebige Variable y_C des dualen LPs zu erhöhen, müssen wir sorgfältig auswählen, welche Variable erhöht werden soll. Wir wählen im folgenden Algorithmus einen Kreis, der möglichst wenige Knoten mit Grad ≥ 3 enthält. Dabei sei $n = |V|$ die Anzahl an Knoten im Eingabegraphen G.

PrimalDual-FVS2

1. $y := 0$; // gültige Lösung des dualen LPs
2. $S := \emptyset$;
3. Entferne Knoten mit Grad 1 aus G, bis kein solcher Knoten mehr existiert.
4. **while** ($\exists$ Kreis C in G)
5. Finde Kreis C^* mit höchstens $2\lceil \log_2 n \rceil$ Knoten mit Grad ≥ 3.
6. Erhöhe y_{C^*} solange, bis $\sum_{C: i \in C} y_C = w_i$ für ein $i \in C^*$ gilt.
7. $S := S \cup \{i\}$;
8. Entferne Knoten i und alle zu i inzidenten Kanten aus G.
9. Entferne Knoten mit Grad 1 aus G, bis kein solcher Knoten mehr existiert.
10. **return** S

Das folgende Lemma zeigt, dass in Zeile 5 des Algorithmus PrimalDual-FVS2 stets effizient ein Kreis C^* mit der gewünschten Eigenschaft gefunden werden kann.

Lemma 8.6. *In jedem Graphen G mit n Knoten, der keine Knoten mit Grad 1 aber mindestens eine Kante enthält, existiert ein Kreis, der höchstens $2\lceil \log_2 n \rceil$ Knoten mit Grad ≥ 3 enthält. Ein solcher kann in polynomieller Zeit gefunden werden.*

Beweis. Zunächst entfernen wir aus G alle Knoten mit Grad 2, indem wir nach und nach jeden Knoten mit Grad 2 durch eine direkte Kante zwischen seinen beiden adjazenten Knoten ersetzen. Es bezeichne G' den resultierenden Graphen. Jeder Kreis in G' entspricht einem Kreis in G.

Enthält G keinen Knoten mit Grad ≥ 3, so ist die zu zeigende Aussage wahr. Ansonsten enthält G' mindestens einen Knoten. Wir wählen einen beliebigen Knoten in G' aus und führen eine Breitensuche von diesem Knoten aus durch. Wir stoppen die Breitensuche, sobald sie einen Kreis findet. Da jeder Knoten in G' Grad ≥ 3 hat, besitzt jeder Knoten im Breitensuchbaum, solange noch kein Kreis gefunden wurde, mindestens zwei Kinder. Die Wurzel besitzt sogar drei Kinder. Auf Ebene 1 gibt es also drei Knoten und solange noch kein Kreis gefunden wurde, ist jede Ebene des Breitensuchbaumes mindestens doppelt so groß wie die Ebene davor. Auf Ebene $i \geq 1$ befinden sich dann mindestens $3 \cdot 2^{i-1}$ Knoten. Wenn bis Ebene k noch kein Kreis gefunden wurde, so befinden sich auf den Ebenen 0 bis k zusammen mindestens

$$1 + \sum_{i=1}^{k} 3 \cdot 2^{i-1} = 1 + 3 \cdot (2^K - 1) = 3 \cdot 2^K - 2$$

Knoten. Für $K = \lceil \log_2((n+2)/3) \rceil$ gilt $3 \cdot 2^K - 2 \geq n$. Damit stoppt die Breitensuche spätestens auf Tiefe K mit dem ersten gefundenen Kreis. Wird der Kreis zwischen

zwei Knoten mit Tiefe K geschlossen, so enthält er maximal $2K + 1$ viele Knoten. In G entspricht dies einem Kreis mit höchstens $2K + 1$ Knoten mit Grad ≥ 3. Ferner gilt für $n \geq 2$ (Übungsaufgabe)

$$2K + 1 = 2\lceil \log_2((n+2)/3) \rceil + 1 \leq 2\lceil \log_2 n \rceil. \qquad \square$$

Theorem 8.7. PrimalDual-FVS2 *ist ein* $(4\lceil \log_2 n \rceil)$*-Approximationsalgorithmus für Feedback Vertex Set.*

Beweis. Wir greifen die obige Rechnung auf:

$$\sum_{i \in S} w_i = \sum_{i \in S} \sum_{C : i \in C} y_C = \sum_{C \in \mathcal{C}} y_C \cdot |S \cap C|.$$

Um diesen Term abzuschätzen, sind nur die Kreise C relevant, die von PrimalDual-FVS2 in Zeile 5 ausgewählt werden, da nur für diese $y_C > 0$ gelten kann. Per Definition gilt für jeden solchen Kreis C, dass er höchstens $2\lceil \log_2 n \rceil$ Knoten mit einem Grad ≥ 3 enthält. Dies schließt zunächst nicht aus, dass er sehr viele Knoten von Grad 2 enthält. Diese gehören zu dem Zeitpunkt, zu dem C betrachtet wird, noch nicht zu S, da sie im Graphen noch vorhanden sind. In späteren Schritten können sie aber der Menge S hinzugefügt werden.

Die Knoten mit Grad ≥ 3 teilen den Kreis C in Segmente ein: zwischen zwei Knoten mit Grad ≥ 3 befindet sich ein (möglicherweise leeres) Segment von aufeinanderfolgenden Knoten mit Grad 2. Es gibt höchstens $2\lceil \log_2 n \rceil$ solcher Segmente auf dem Kreis C. Wir argumentieren nun, dass von jedem dieser Segmente im Laufe des Algorithmus höchstens ein Knoten der Menge S hinzugefügt wird. Betrachten wir dazu ein Segment P und den ersten Schritt, in dem ein Knoten $i \in P$ der Menge S hinzugefügt wird. In diesem Schritt wird i aus dem Graphen entfernt. Danach werden in diesem Schritt auch alle anderen Knoten aus P nacheinander entfernt. Dies liegt daran, dass P ein Pfad bestehend aus Knoten mit Grad 2 ist. Wird ein Knoten auf diesem Pfad gelöscht, so haben seine Nachbarn danach nur noch Grad 1 und werden ebenfalls gelöscht. Dieser Prozess setzt sich fort, bis alle Knoten des Pfades gelöscht wurden.

Für einen Kreis C mit $y_C > 0$ werden also höchstens alle Knoten mit Grad ≥ 3 sowie ein Knoten mit Grad 2 pro Segment zu der Menge S hinzugefügt. Damit gilt $|S \cap C| \leq 4\lceil \log_2 n \rceil$ und somit

$$\sum_{i \in S} w_i = \sum_{C \in \mathcal{C}} y_C \cdot |S \cap C| \leq 4\lceil \log_2 n \rceil \cdot \sum_{C \in \mathcal{C}} y_C \leq 4\lceil \log_2 n \rceil \cdot Z^* \leq 4\lceil \log_2 n \rceil \cdot \mathrm{OPT}. \qquad \square$$

Auch Theorem 8.7 kann aus Theorem 8.5 abgeleitet werden. Sei $x \in \{0,1\}^n$ die ganzzahlige Lösung des primalen LPs, die der von PrimalDual-FVS2 konstruierten Lösung S entspricht, und sei y die konstruierte Lösung des dualen LPs. PrimalDual-FVS2 stellt sicher, dass die folgenden Bedingungen gelten:

$$\forall i \in V : x_i > 0 \Rightarrow \sum_{C : i \in C} y_C = w_i.$$

und

$$\forall C \in \mathcal{C} : y_C > 0 \Rightarrow \sum_{i \in C} x_i \leq 4\lceil \log_2 n \rceil.$$

Damit kann Theorem 8.5 mit $\alpha = 1$ und $\beta = 4\lceil \log_2 n \rceil$ angewendet werden.

8.4 Das Kürzeste-Wege-Problem

Wir betrachten in diesem Abschnitt das Problem einen kürzesten s-t-Pfad in einem ungerichteten Graphen zu finden. Dazu sei ein ungerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}_{>0}$ und zwei ausgezeichneten Knoten $s \in V$ und $t \in V$ gegeben. Das Ziel ist es, einen möglichst kurzen s-t-Pfad zu finden.

Kürzeste-Wege-Problem

Eingabe:	ungerichteter Graph $G = (V, E)$, Kantengewichte $w : E \to \mathbb{R}_{>0}$ Startknoten $s \in V$, Zielknoten $t \in V$
Lösungen:	alle s-t-Pfade P in G
Zielfunktion:	minimiere $\sum_{e \in P} w(e)$

Dass wir ausgerechnet dieses Problem betrachten, mag verwundern, denn im Gegensatz zu den anderen Problemen aus dieser Vorlesung handelt es sich beim Kürzesten-Wege-Problem um ein Problem, das effizient gelöst werden kann. Dennoch werden wir untersuchen, wie dieses Problem mithilfe eines primal-dualen Algorithmus gelöst werden kann. Dies dient als Vorbereitung auf den folgenden Abschnitt über das Steinerwaldproblem.

Zunächst formulieren wir das Problem als ganzzahliges lineares Programm. Dazu bezeichne $\mathcal{S} = \{S \subseteq V \mid s \in S, t \notin S\}$ die Menge aller *s-t-Schnitte* des Graphen G. Wir führen für jede Kante $e \in E$ eine Variable $x_e \in \{0, 1\}$ ein, die angibt, ob sie auf dem s-t-Pfad enthalten ist. Es bezeichne $\delta(S)$ die Menge der Kanten, die aus der Menge S hinausführen. Es gilt also

$$\delta(S) = \{(x, y) \in E \mid |\{x, y\} \cap S| = 1\}.$$

Ferner nutzen wir in dem folgenden linearen Programm wieder die Notation $w_e = w(e)$:

$$\begin{aligned} \text{minimiere} \quad & \sum_{e \in E} w_e x_e \\ \text{sodass} \quad & \sum_{e \in \delta(S)} x_e \geq 1, \quad \forall S \in \mathcal{S}, \\ & x_e \in \{0, 1\}, \quad \forall e \in E. \end{aligned} \tag{8.12}$$

Genau wie bei Feedback Vertex Set besitzt dieses LP im Allgemeinen exponentiell viele Nebenbedingungen. Diese erzwingen, dass in jeder Lösung aus jedem s-t-Schnitt mindestens eine Kante hinausführt. Man überzeugt sich leicht davon, dass jeder s-t-Pfad diese Eigenschaft erfüllt. Wir betrachten umgekehrt für eine Lösung x des ganzzahligen LPs den Graphen $G' = (V, E')$ mit $E' = \{e \in E \mid x_e = 1\}$, der aus den ausgewählten Kanten besteht. Da aus jedem s-t-Schnitt mindestens eine Kante hinausführt, beträgt der maximale Fluss von s nach t gemäß dem Max-Flow-Min-Cut-Theorem mindestens 1. Damit muss in E' ein s-t-Pfad enthalten sein. Ist x eine optimale Lösung, so entspricht E' sogar einem s-t-Pfad. Insgesamt impliziert dies,

dass jeder kürzeste s-t-Pfad eine optimale Lösung des ganzzahligen LPs darstellt und umgekehrt.

Wir relaxieren das LP, indem wir die Bedingung $x_e \in \{0,1\}$ durch die Bedingung $x_e \geq 0$ ersetzen. Das zu dieser Relaxierung duale LP besitzt für jeden s-t-Schnitt S eine Variable y_S und für jede Kante $e \in E$ eine Nebenbedingung. Es sieht wie folgt aus:

$$\begin{aligned} \text{maximiere } & \sum_{S \in \mathcal{S}} y_S \\ \text{sodass } & \sum_{S: e \in \delta(S)} y_S \leq w_e, \quad \forall e \in E, \\ & y_S \geq 0, \qquad\qquad \forall S \in \mathcal{S}. \end{aligned} \tag{8.13}$$

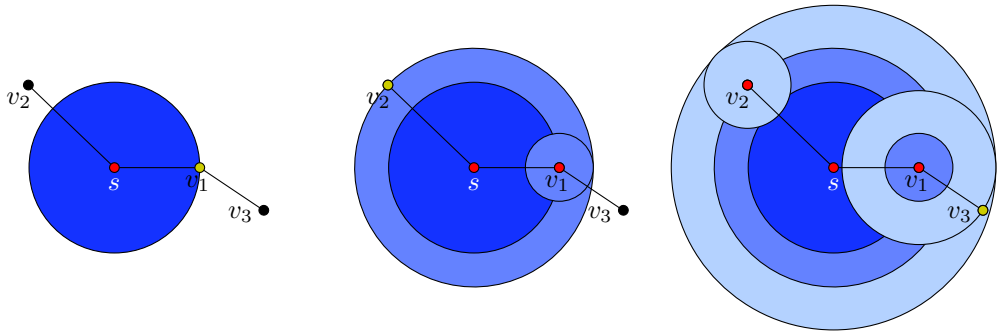
Um eine Intuition für dieses duale LP zu gewinnen, stellen wir uns vor, dass jeder Schnitt $S \in \mathcal{S}$ von einer Begrenzung mit einer Dicke von y_S umgeben ist. Jede Kante e muss lang genug sein, um alle Begrenzungen für Schnitte mit $e \in \delta(S)$ durchqueren zu können. Deshalb muss die Gesamtdicke dieser Begrenzungen durch w_e nach oben beschränkt sein.

Wir betrachten den folgenden primal-dualen Algorithmus. Dieser führt eine Lösung y des dualen LPs mit und erzeugt eine ganzzahlige Lösung $F \subseteq E$ für das primale LP. Solange es in F noch keinen s-t-Pfad gibt, werden F weitere Kanten hinzugefügt. Diese werden mit dem typischen Schema eines primal-dualen Algorithmus ausgewählt. Dazu wird eine Variable y_S des dualen LPs solange erhöht, bis eine der Nebenbedingungen scharf wird. Die zu dieser Nebenbedingung gehörende Kante wird dann der Menge F hinzugefügt. Für die Analyse des Algorithmus ist es wichtig, welche Variable y_S erhöht wird. Wir wählen S als die Menge der Knoten, die mit den Kanten aus F von s aus erreicht werden können. Der letzte Schritt im Algorithmus unterscheidet sich etwas von den primal-dualen Algorithmen, die wir bislang gesehen haben. Anstatt direkt die Menge F zurückzugeben, wird diese gegebenenfalls noch ausgedünnt, indem nur ein beliebiger in F enthaltener s-t-Pfad ausgegeben wird. Der Algorithmus ist im folgenden Pseudocode noch einmal ausführlich dargestellt.

PrimalDual-ShortestPath

1. $y := 0$; // gültige Lösung des dualen LPs
2. $F := \emptyset$;
3. **while** (es existiert kein s-t-Pfad in (V, F))
4. Sei S die Menge der von s erreichbaren Knoten in (V, F).
5. Erhöhe y_S solange, bis für eine Kante $e \in \delta(S)$ gilt $\sum_{S': e \in \delta(S')} y_{S'} = w_e$.
6. $F := F \cup \{e\}$;
7. Es sei P ein s-t-Pfad in (V, F).
8. **return** P

Abbildung 8.1 stellt das Verhalten des Algorithmus in einem Beispiel dar. Wir untersuchen nun den Approximationsfaktor. Dazu zeigen wir zunächst die folgende Aussage.



(a) $S = \{s\}$ (b) $S = \{s, v_1\}$ (c) $S = \{s, v_1, v_2\}$

Abbildung 8.1: Beispiel für das Verhalten von PrimalDual-ShortestPath: Im ersten Schritt gilt $S = \{s\}$ und der Algorithmus erhöht die Dicke y_S der Begrenzung solange, bis eine Nebenbedingung scharf wird. Das ist zuerst für die Kante (s, v_1) der Fall. Diese wird eingefügt, wodurch S um v_1 erweitert wird. Dann wird die Dicke y_S der Begrenzung für $S = \{s, v_1\}$ erhöht. Anschaulich bedeutet das, dass der Kreis um s weiter wächst und um v_1 ein neuer Kreis wächst. Die nächste Nebenbedingung, die scharf wird, gehört zu der Kante (s, v_2). Dementsprechend wird v_2 der Menge S hinzugefügt. Im letzten Schritt wachsen die Kreise um s und v_1 weiter und um v_2 wächst ein neuer Kreis.

Lemma 8.8. *Zu jedem Zeitpunkt bilden die Kanten aus F einen Baum auf der Menge der von s aus erreichbaren Knoten.*

Beweis. Zu Beginn gilt $F = \emptyset$ und damit ist die Aussage wahr. Wir betrachten nun einen Schritt, in dem eine Kante $e = (x, y)$ der Menge F hinzugefügt wird. Es gilt $e \in \delta(S)$ für die Menge S, der von s erreichbaren Knoten in (V, F). Sei $x \in S$ und $y \notin S$. Da der Knoten y von s aus über die Kanten F nicht erreicht werden kann, kann die Hinzunahme der Kante e keinen Kreis schließen (man beachte, dass der Graph G ungerichtet ist). Damit ist die Invariante gezeigt. □

Wir zeigen nun, dass PrimalDual-ShortestPath eine optimale Lösung berechnet.

Theorem 8.9. *Die Ausgabe von* PrimalDual-ShortestPath *ist stets ein kürzester s-t-Pfad.*

Beweis. Der Beweis des Theorems folgt mit den Methoden, die wir bereits in den Analysen der anderen primal-dualen Algorithmen eingesetzt haben. Da der Algorithmus eine Kante e nur dann zu der Menge F hinzufügt, wenn die entsprechende Nebenbedingung scharf ist, gilt

$$\sum_{e \in P} w_e = \sum_{e \in P} \sum_{S : e \in \delta(S)} y_S = \sum_{S \in \mathcal{S}} y_S \cdot |P \cap \delta(S)|.$$

Es bleibt zu zeigen, dass $|P \cap \delta(S)| = 1$ für jedes S mit $y_S > 0$ gilt. Dies impliziert nämlich aufgrund der schwachen Dualität

$$\sum_{e \in P} w_e = \sum_{S \in \mathcal{S}} y_S \leq \text{OPT}.$$

Um die fehlende Aussage zu zeigen, führen wir einen Beweis durch Widerspruch. Nehmen wir an, es gibt einen Schnitt S mit $y_S > 0$ und $|P \cap \delta(S)| \geq 2$. Wir folgen dem Pfad P Kante für Kante: nach einer ungeraden Anzahl an Kanten aus $\delta(S)$ befindet sich der Pfad außerhalb von S, nach einer gerade Anzahl an Kanten aus $\delta(S)$ innerhalb von S. Wegen $t \notin S$ muss $|P \cap \delta(S)|$ somit ungerade sein. Damit gilt im betrachteten Fall sogar $|P \cap \delta(S)| \geq 3$. Wir betrachten das Teilstück P' von P zwischen der ersten und zweiten Kante aus $|P \cap \delta(S)|$ inklusive dieser Kanten (siehe Abbildung 8.2). Der Pfad P' verläuft bis auf seinen Start- und Endknoten komplett außerhalb von S.

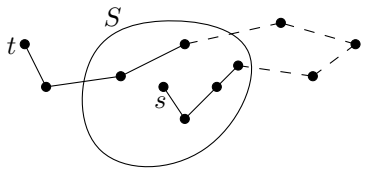


Abbildung 8.2: Die gestrichelten Kanten bilden den Pfad P'.

Wir betrachten nun den Zeitpunkt, zu dem y_S vom Algorithmus erhöht wird, und bezeichnen mit F' die Kanten, die der Algorithmus zu diesem Zeitpunkt bereits ausgewählt hat. Die Kanten aus F' bilden gemäß Lemma 8.8 einen Spannbaum auf S. Dies bedeutet, dass es in F' insbesondere einen Weg zwischen dem Start- und Endknoten von P' gibt. Damit enthält $F' \cup P'$ einen Kreis. Für die Menge F der vom Algorithmus am Ende ausgewählten Kanten gilt $F \supseteq F' \cup P'$. Damit enthält auch diese Menge einen Kreis, was Lemma 8.8 widerspricht. □

Nebenbei seit angemerkt, dass der Algorithmus PrimalDual-ShortestPath allen Teilnehmerinnen und Teilnehmern der Vorlesung gut bekannt sein sollte. Er verhält sich nämlich genau so wie der Dijkstra-Algorithmus zur Berechnung eines kürzesten s-t-Pfades.

8.5 Steinerwaldproblem

Bei dem Steinerwaldproblem ist ein ungerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}_{>0}$ gegeben. Zusätzlich sind Knotenpaare $(s_1, t_1), \ldots, (s_k, t_k)$ mit $s_i, t_i \in V$ gegeben. Wir suchen eine Menge $F \subseteq E$ von Kanten mit möglichst kleinem Gesamtgewicht, sodass es für jedes Paar (s_i, t_i) einen s_i-t_i-Weg in (V, F) gibt.

Steinerwaldproblem

Eingabe:	ungerichteter Graph $G = (V, E)$, Kantengewichte $w : E \to \mathbb{R}_{>0}$ Knotenpaare $(s_1, t_1), \ldots, (s_k, t_k)$ mit $s_i, t_i \in V$
Lösungen:	alle $F \subseteq E$, sodass es in (V, F) für jedes i einen s_i-t_i-Weg gibt
Zielfunktion:	minimiere $\sum_{e \in F} w(e)$

Das Kürzeste-Wege-Problem kann als Steinerwaldproblem mit nur einem Paar von Knoten aufgefasst werden. Das Steinerwaldproblem ist NP-schwer. Wir können es

ähnlich zum Kürzeste-Wege-Problem als ganzzahliges lineares Programm modellieren. Dazu sei $\mathcal{S}_i = \{S \subseteq V \mid |S \cap \{s_i, t_i\}| = 1\}$ die Menge aller s_i-t_i-Schnitte in G und $\delta(S)$ sei so definiert wie im letzten Abschnitt. Ferner sei $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_k$. Wieder nutzen wir die Notation $w_e = w(e)$ und führen eine Variable $x_e \in \{0, 1\}$ für jede Kante des Graphen ein. Wir erhalten dann das folgende lineare Programm:

$$\begin{aligned} \text{minimiere} \quad & \sum_{e \in E} w_e x_e \\ \text{sodass} \quad & \sum_{e \in \delta(S)} x_e \geq 1, \quad \forall S \in \mathcal{S}, \\ & x_e \in \{0, 1\}, \qquad \forall e \in E. \end{aligned} \tag{8.14}$$

Die Nebenbedingungen stellen sicher, dass es für jedes i in der ausgewählte Menge von Kanten einen s_i-t_i-Weg gibt. Mit den gleichen Argumenten wie für das Kürzeste-Wege-Problem kann auch hier wieder gezeigt werden, dass jede optimale Lösung des ganzzahligen LPs einer optimalen Lösung für das Steinerwaldproblem entspricht. Wir relaxieren das Programm, indem wir die Bedingungen $x_e \in \{0, 1\}$ durch die Bedingungen $x_e \geq 0$ ersetzen, und erhalten das folgende duale LP, welches eine Variable y_S für jeden Schnitt $S \in \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_k$ enthält:

$$\begin{aligned} \text{maximiere} \quad & \sum_{S \in \mathcal{S}} y_S \\ \text{sodass} \quad & \sum_{S : e \in \delta(S)} y_S \leq w_e, \quad \forall e \in E, \\ & y_S \geq 0, \qquad \forall S \in \mathcal{S}. \end{aligned} \tag{8.15}$$

Sowohl das primale als auch das duale LP sind identisch mit dem entsprechenden LP für das Kürzeste-Wege-Problem. Der einzige Unterschied ist, dass die Menge $\mathcal{S}$ nun für jedes i die s_i-t_i-Schnitte enthält. Wir können dementsprechend auch wieder fast denselben primal-dualen Algorithmus anwenden.

PrimalDual-Steinerwald

1. $y := 0$; // gültige Lösung des dualen LPs
2. $F := \emptyset$;
3. **while** (es gibt i, für das kein s_i-t_i-Pfad in (V, F) existiert)
4. Sei S eine Zusammenhangskomponente in (V, F) mit $|S \cap \{s_i, t_i\}| = 1$.
5. Erhöhe y_S solange, bis für eine Kante $e \in \delta(S)$ gilt $\sum_{S' : e \in \delta(S')} y_{S'} = w_e$.
6. $F := F \cup \{e\}$;
7. **return** F

Bei der Analyse dieses Algorithmus mit den Standardmethoden, die wir in diesem Kapitel kennengelernt haben, tritt ein Problem auf. Es gilt

$$\sum_{e \in F} w_e = \sum_{e \in F} \sum_{S : e \in \delta(S)} y_S = \sum_{S \in \mathcal{S}} y_S \cdot |F \cap \delta(S)|.$$

Wenn wir einen Approximationsfaktor von α zeigen wollen, müssen wir zeigen, dass $|F \cap \delta(S)| \leq \alpha$ für jedes $y_S > 0$ gilt. Wir betrachten eine Clique auf $k+1$ Knoten. Ein Knoten entspreche dabei $s_1 = \ldots = s_k$ und jedes t_i entspreche einem der anderen Knoten. Die Kantengewichte seien alle 1. Zu Beginn muss der Algorithmus PrimalDual-Steinerwald die Menge S so wählen, dass sie aus genau einem der $k+1$ Knoten besteht. Nennen wir diesen v. Er setzt dann $y_S = 1$. Im weiteren Verlauf wird $y_{S'}$ für kein anderes S' mehr erhöht und die resultierende Menge F ist ein Stern mit Zentrum v. Für $S = \{v\}$ gilt dann $|F \cap \delta(S)| = k$. Mit der Standardanalyse lässt sich also bestenfalls ein Approximationsfaktor von k zeigen, welcher trivial zu erreichen ist.

Wir werden diesen Approximationsfaktor nun dadurch verbessern, dass wir mehrere y_S gleichzeitig erhöhen. Dies führt zu dem folgenden Algorithmus.

PrimalDual-Steinerwald2

1. $y := 0$; // gültige Lösung des dualen LPs
2. $F := \emptyset$;
3. $\ell := 0$;
4. **while** (es gibt ein i, für das kein s_i-t_i-Pfad in (V, F) existiert)
5. $\quad \ell := \ell + 1$;
6. $\quad$ Sei $\mathcal{C}$ die Menge der Zusammenhangskomponenten S von (V, F) mit $|S \cap \{s_i, t_i\}| = 1$ für ein i.
7. $\quad$ Erhöhe y_S gleichmäßig für alle $S \in \mathcal{C}$, bis für ein $S \in \mathcal{C}$ und eine Kante $e_\ell \in \delta(S)$ gilt $\sum_{S': e_\ell \in \delta(S')} y_{S'} = w_{e_\ell}$.
8. $\quad F := F \cup \{e_\ell\}$;
9. $F' := F$;
10. **for** $(k = \ell;\ k \geq 1;\ k{-}{-})$
11. $\quad$ **if** ($F' \setminus \{e_k\}$ ist gültige Lösung)
12. $\quad\quad F' := F' \setminus \{e_k\}$;
13. **return** F'

Der wesentliche Unterschied zu PrimalDual-Steinerwald ist, dass in diesem Algorithmus gleichzeitig für alle Zusammenhangskomponenten $S \in \mathcal{C}$ die Variable y_S erhöht wird. Ansonsten ist das Vorgehen identisch: sobald die erste Nebenbedingung scharf wird, wird die entsprechende Kante in die Lösung F eingefügt. Die Variable ℓ dient ausschließlich der Benennung der Kanten. Die i-te eingefügte Kante erhält die Bezeichnung e_i. In der for-Schleife wird die Lösung wieder bereinigt und überflüssige Kanten werden aus der Lösung F entfernt. Der Algorithmus PrimalDual-Steinerwald2 besitzt eine einfache intuitive Darstellung, die in Abbildung 8.3 erläutert wird.

Das folgende Lemma, dessen Beweis wir weiter unten besprechen werden, gibt eine Schranke dafür an, wie viele Kanten in einer Iteration insgesamt aus allen Schnitten $S \in \mathcal{C}$ hinausführen.

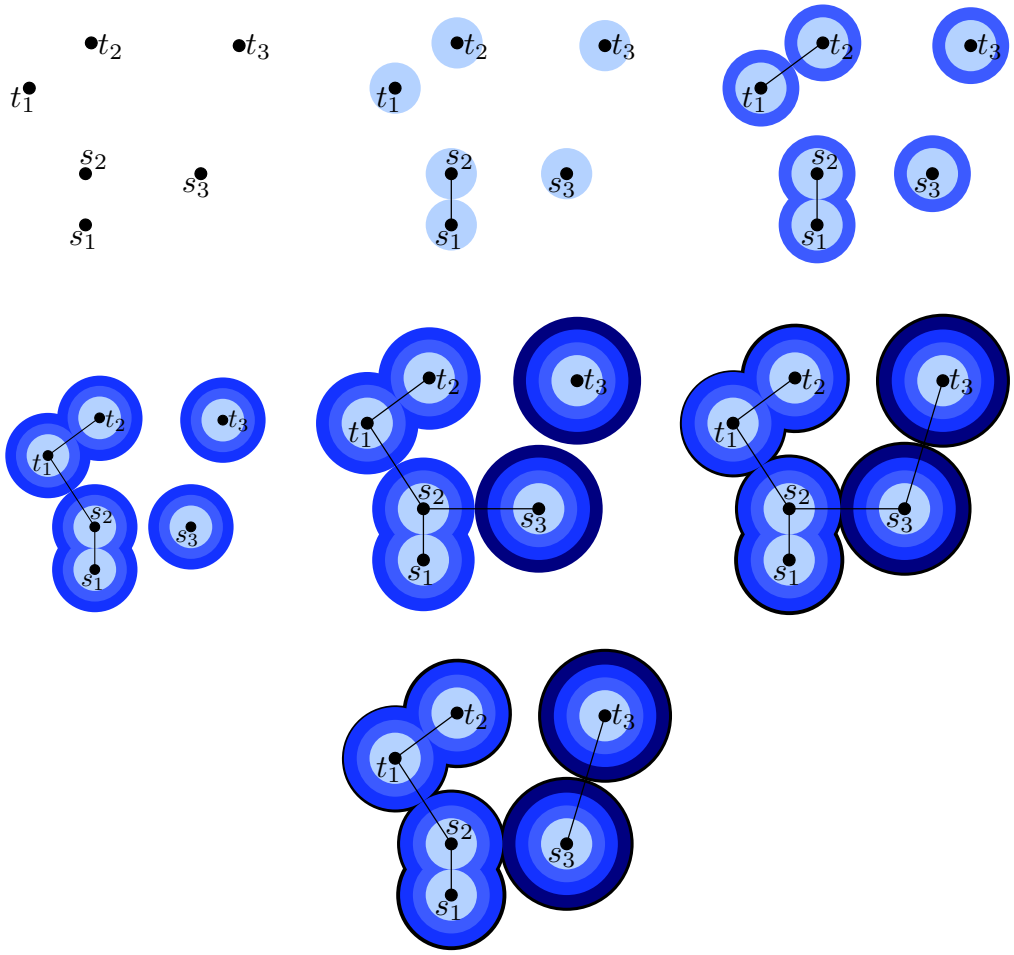


Abbildung 8.3: Beispiel für das Verhalten von PrimalDual-Steinerwald2: Im Algorithmus wird um jeden Knoten herum ein Grenzbereich definiert, dessen Dicke in jedem Schritt anwächst, wenn der Knoten sich in einer Komponente $S \in \mathcal{C}$ befindet. Sobald sich die Grenzbereiche zweier Knoten x und y treffen, wird die Kante (x, y) der Menge F hinzugefügt, sofern $(x, y) \in E$ und x und y sich derzeit noch in verschiedenen Zusammenhangskomponenten befinden. In der sechsten Abbildung sind alle s_i-t_i-Paare verbunden und der Algorithmus bricht die while-Schleife mit der aktuellen Kantenmenge F ab. Diese wird noch bereinigt und überflüssige Kanten werden gelöscht. Das Ergebnis F' ist in der siebten Abbildung dargestellt.

Lemma 8.10. *In jeder Iteration von* PrimalDual-Steinerwald2 *gilt*

$$\sum_{S \in \mathcal{C}} |F' \cap \delta(S)| \leq 2|\mathcal{C}|.$$

Mithilfe dieses Lemmas können wir den Approximationsfaktor von PrimalDual-Steinerwald2 beschränken.

Theorem 8.11. PrimalDual-Steinerwald2 *ist ein 2-Approximationsalgorithmus für das Steinerwaldproblem.*

Beweis. Es gilt wie üblich

$$\sum_{e \in F'} w_e = \sum_{e \in F'} \sum_{S : e \in \delta(S)} y_S = \sum_{S \in \mathcal{S}} y_S \cdot |F' \cap \delta(S)|.$$

Unser Ziel ist es, die Ungleichung

$$\sum_{S \in \mathcal{S}} y_S \cdot |F' \cap \delta(S)| \leq 2 \sum_{S \in \mathcal{S}} y_S \tag{8.16}$$

zu zeigen. Anders als in den vorherigen Analysen gilt aber nicht, dass $|F' \cap \delta(S)|$ für jedes S mit $y_S > 0$ durch 2 nach oben beschränkt ist. Wir zeigen (8.16) stattdessen per Induktion über die Anzahl an Iterationen von PrimalDual-Steinerwald2. Zu Beginn ist $y_S = 0$ für alle $S \in \mathcal{S}$ und damit ist die Ungleichung erfüllt. Sei (8.16) nun zu Beginn einer Iteration erfüllt. In der betrachteten Iteration wird y_S für jedes $S \in \mathcal{C}$ um denselben Wert ε erhöht. Dadurch vergrößert sich die linke Seite von (8.16) um

$$\varepsilon \cdot \sum_{S \in \mathcal{C}} |F' \cap \delta(S)|$$

und die rechte um

$$2\varepsilon |\mathcal{C}|.$$

Wegen Lemma 8.10 ist die Erhöhung der linken Seite kleiner als die der rechten. Damit bleibt die Ungleichung erfüllt und es folgt insgesamt

$$\sum_{e \in F'} w_e = \sum_{S \in \mathcal{S}} y_S \cdot |F' \cap \delta(S)| \leq 2 \sum_{S \in \mathcal{S}} y_S \leq 2 \cdot \mathrm{OPT}. \qquad \square$$

Zum Beweis von Lemma 8.10 benötigen wir noch die folgende Aussage.

Lemma 8.12. *Die Menge F der von* PrimalDual-Steinerwald2 *ausgewählten Kanten ist zu jedem Zeitpunkt kreisfrei.*

Beweis. Zu Beginn gilt $F = \emptyset$ und die Aussage ist erfüllt. Die Gültigkeit für alle weiteren Zeitpunkte folgt induktiv. Sei F zu Beginn einer Iteration kreisfrei. Die Kante e_ℓ, die der Menge F hinzugefügt wird, stammt aus $\delta(S)$ für eine Zusammenhangskomponente $S \in \mathcal{C}$. Damit verbindet e_ℓ zwei Zusammenhangskomponenten und kann keinen Kreis schließen. $\square$

Beweis von Lemma 8.10. Wir betrachten die Iteration, in der für ein ℓ die Kante e_ℓ der Menge F hinzugefügt wird. Es sei $F_\ell = \{e_1, \ldots, e_{\ell-1}\}$ die Menge der Kanten, aus denen F zu Beginn dieser Iteration besteht. Ferner sei $H = F' \setminus F_\ell$. Dann ist $F_\ell \cup H = F_\ell \cup F'$ eine Lösung für das Steinerwaldproblem, da F' bereits alleine eine Lösung ist. Ferner ist $(F_\ell \cup H) \setminus \{e\}$ für keine Kante $e \in H$ eine Lösung. Dies folgt daraus, dass in der for-Schleife, in der überflüssige Kanten gelöscht werden, die Kanten in umgekehrter Einfügereihenfolge getestet werden. Aus $e \in H$ folgt $e = e_j$ für ein $j > \ell - 1$. Das bedeutet, zu dem Zeitpunkt, zu dem getestet wird, ob e gelöscht werden kann, sind noch alle Kanten aus F_ℓ (und alle Kanten aus H) in F' vorhanden. Wäre also $(F_\ell \cup H) \setminus \{e\}$ eine Lösung, so wäre e dementsprechend gelöscht worden.

Wir erzeugen einen neuen Graphen G', indem wir in dem Graphen (V, F_ℓ) jede Zusammenhangskomponente zu einem Knoten kontrahieren. Dadurch entsteht eine Menge V' von Knoten. Die Kanten in G' werden durch die Kanten aus H erzeugt. Keine Kante aus H kann beide Endknoten in derselben Zusammenhangskomponente von (V, F_ℓ) haben, da die Knotenmenge $F_\ell \cup H \subseteq F$ sonst im Widerspruch zu Lemma 8.12 nicht kreisfrei wäre. Somit verbindet jede Kante aus H zwei Zusammenhangskomponenten in (V, F_ℓ). Wir fügen für jede Kante e aus H eine Kante in G' ein, die die Knoten verbindet, die den Zusammenhangskomponenten entsprechen, die e verbindet. Dadurch entstehen keine Mehrfachkanten, da es wegen der Kreisfreiheit zwischen jedem Paar von Zusammenhangskomponenten nur höchstens eine Kante in H geben kann.

Für einen Knoten $v \in V'$ bezeichne $d(v)$ seinen Grad im Graphen G'. Wir teilen die Knotenmenge V' in aktive und passive Knoten ein. Ein aktiver Knoten entspricht einer Zusammenhangskomponente S mit $S \in \mathcal{C}$, also $|S \cap \{s_i, t_i\}| = 1$ für ein i. Alle anderen Knoten heißen passiv. Es bezeichne A die Menge der aktiven Knoten und P bezeichne die Menge der passiven Knoten v mit $d(v) > 0$. Die gewünschte Ungleichung

$$\sum_{S \in \mathcal{C}} |F' \cap \delta(S)| \leq 2|\mathcal{C}| \tag{8.17}$$

können wir mithilfe dieser Definitionen umschreiben. Die rechte Seite entspricht $2|A|$. Für jedes $S \in \mathcal{C}$ gilt $|F' \cap \delta(S)| = d(v)$, wobei $v \in V'$ den Knoten bezeichne, der die Zusammenhangskomponente S repräsentiert, wegen $F' \cap \delta(S) \subseteq H$. Somit ist (8.17) äquivalent zu

$$\sum_{v \in A} d(v) \leq 2|A|. \tag{8.18}$$

Es gilt

$$\sum_{v \in A} d(v) = \sum_{v \in A \cup P} d(v) - \sum_{v \in P} d(v).$$

Wäre $d(v) \geq 2$ für jedes $v \in P$, so gilt (8.18):

$$\sum_{v \in A} d(v) = \sum_{v \in A \cup P} d(v) - \sum_{v \in P} d(v) \leq 2(|A| + |P|) - 2|P| = 2|A|,$$

wobei wir für die Abschätzung von $\sum_{v \in A \cup P} d(v)$ ausgenutzt haben, dass der Graph G' aufgrund seiner Kreisfreiheit nur höchstens $|A| + |P| - 1 \leq |A| + |P|$ Kanten enthalten kann. Somit ist die Summe der Knotengerade durch $2(|A| + |P|)$ nach oben beschränkt.

Es bleibt zu zeigen, dass alle passiven Knoten entweder gar keine inzidente Kante haben oder mindestens zwei. Angenommen, es existiert ein passiver Knoten $v \in V'$ mit Grad 1. Sei $e \in H \subseteq F'$ die zu v inzidente Kante. Da e zu F' gehört, ist e notwendig, um die Zulässigkeit der Lösung zu garantieren (sonst wäre Kante e aus F' gelöscht worden). Dies bedeutet, die Kante e muss auf einem s_i-t_i-Pfad in der Lösung liegen. Dies impliziert $|S \cap \{s_i, t_i\}| = 1$ für die Zusammenhangskomponente S, die v repräsentiert. Damit wäre v ein aktiver Knoten im Widerspruch zur Wahl von v. □

Kapitel 9

Traveling-Salesman-Problem

Das Traveling-Salesman-Problem (TSP) haben wir bereits in früheren Vorlesungen kennengelernt. In diesem Abschnitt zeigen wir zunächst, dass für das allgemeine TSP kein Approximationsalgorithmus existiert, der eine sinnvolle Approximationsgüte liefert. Anschließend lernen wir für zwei Spezialfälle des TSP einen 3/2-Approximationsalgorithmus sowie ein Approximationsschema kennen.

Traveling-Salesman-Problem (TSP)

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

9.1 Allgemeines TSP

Was ist die beste Approximationsgüte, die für das 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 9.1. *Falls $P \neq NP$, so existiert kein 2^n-Approximationsalgorithmus für das TSP.*

Beweis. Wir haben uns in der Vorlesung „Algorithmen und Berechnungskomplexität II“ mit dem Hamiltonkreis-Problem HC 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 gesehen, 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 den 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, wenn 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 Codierungslänge 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 beträgt. 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 demzufolge 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 9.1 ist willkürlich. Tatsächlich kann man mit dem obigem Beweis sogar für jede in Polynomialzeit berechenbare Funktion $r : \mathbb{N} \to \mathbb{R}_{\geq 1}$ zeigen, dass es keinen $r(n)$-Approximationsalgorithmus für das TSP gibt, falls P $\neq$ 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 $\neq$ NP für ein Problem keinen r-Approximationsalgorithmus gibt, zeigt man, dass man mithilfe eines solchen Algorithmus ein NP-schweres Problem in polynomieller Zeit lösen kann.

9.2 Metrisches TSP

Es gibt verschiedene 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 bei vielen Anwendungen gegeben ist, ist die Dreiecksungleichung. Das bedeutet, dass

$$d(x, z) \leq d(x, y) + d(y, z)$$

für alle Knoten $x, y, z \in V$ 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 und ist die Distanz zweier Punkte durch ihren euklidischen Abstand gegeben, so ist diese Eigenschaft erfüllt.

Definition 9.2. *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. 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-schwer ist. Dies folgt direkt daraus, dass das TSP auch dann noch NP-schwer ist, wenn alle Distanzen zwischen verschiedenen Knoten entweder 1 oder 2 sind. Dann ist die Dreiecksungleichung automatisch erfüllt. Somit ist gezeigt, dass auch das metrische TSP noch NP-schwer ist.

9.2.1 2-Approximationsalgorithmus

In folgendem Algorithmus benutzen wir den Begriff eines *Eulerkreises*. Dabei handelt es sich um einen Kreis in einem Graphen, der jede Kante genau einmal enthält, der Knoten aber mehrfach besuchen darf. Ein Graph mit einem solchen Kreis wird auch *eulerscher Graph* genannt (siehe Abbildung 9.1).

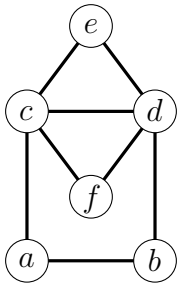


Abbildung 9.1: Dieser Graph enthält den Eulerkreis $(a, c, e, d, f, c, d, b, a)$.

Das Problem, einen Eulerkreis zu finden, sieht auf den ersten Blick so ähnlich aus wie das NP-schwere 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, falls 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 besitzt. Außerdem kann man in einem solchen Graphen auch in polynomieller Zeit einen Eulerkreis berechnen. Diese Behauptungen zu beweisen, überlassen wir dem Leser als Übung.

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

Metric-TSP

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

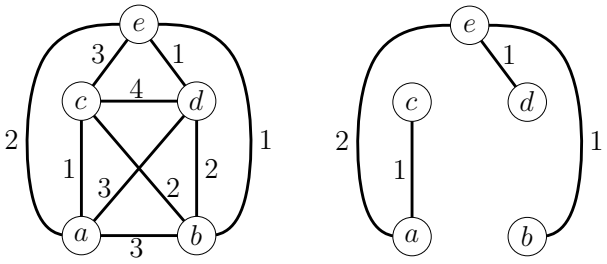
1. Sei $G = (V, E)$ ein vollständiger ungerichteter Graph mit Knotenmenge V. Berechne einen minimalen Spannbaum T von G bezüglich der Distanzen d.
2. Erzeuge einen Multigraphen G', der nur die Kanten aus T enthält und jede davon genau zweimal. In G' besitzt jeder Knoten geraden Grad.
3. Finde einen Eulerkreis 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 der gefundene Eulerkreis A in G' (der alle Knoten aus V mindestens einmal enthält) in einen Hamiltonkreis umgewandelt. 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) (siehe Abbildung 9.2). Bilden die Distanzen d eine Metrik auf V, so ist durch die Dreiecksungleichung garantiert, dass der Kreis durch das Überspringen von bereits besuchten Knoten nicht länger wird.

Theorem 9.3. *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 $C^\star \subseteq E$ 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



(a) Eingabe (b) Spannbaum

(c) Multigraph G' mit Eulerkreis $A = (c, a, e, d, e, b, e, a, c)$ (d) Lösung C

Abbildung 9.2: Beispiel für den Algorithmus METRIC-TSP

erhalten wir damit

$$\mathrm{OPT} = d(C^\star) = d(P) + d(e) \geq d(P) \geq d(T). \tag{9.1}$$

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 (9.1) impliziert dies, dass der Algorithmus stets eine 2-Approximation berechnet. □

Es stellt sich 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, wenn nur 1 und 2 als Distanzen erlaubt sind. Wir gehen davon aus, dass n gerade ist. Es gelte

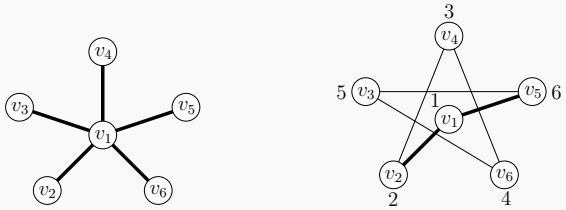
$$\forall i \in \{2, \ldots, n\} : d(v_1, v_i) = 1$$

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 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.

9.2.2 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 dem folgenden 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*. Ein Matching M eines Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, sodass 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 vollständigen Graphen mit einer geraden Anzahl an Knoten ein perfektes Matching mit minimalem Gewicht in polynomieller Zeit berechnet werden kann. Wie der Algorithmus dafür aussieht, wollen wir hier aber nicht diskutieren.

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 $\tilde{G} = (V, T \cup M)$. Dabei sei jede Kante $e \in T \cap M$ 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 9.4. *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 Spannbaum T. Für $v \in V$ bezeichne $d_T(v)$ den Grad des Knotens v im Spannbaum T. Dann ist

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

eine gerade Zahl, da jede Kante von T 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 Beweis von Theorem 9.3 bereits eine solche untere Schranke gezeigt, nämlich das Gewicht des minimalen Spannbaumes. Wir zeigen nun noch eine weitere Schranke.

Lemma 9.5. *Es sei $V' \subseteq V$ beliebig, sodass $|V'|$ gerade ist. Außerdem sei M ein perfektes Matching auf V' mit minimalem Gewicht $d(M)$. Dann gilt $d(M) \leq \mathrm{OPT}/2$.*

Beweis. Es sei $C^\star$ 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 $C^\star$ besucht werden. Das heißt, wir nehmen in $C^\star$ Abkürzungen und überspringen die Knoten, die nicht zu V' gehören. Wegen der Dreiecksungleichung kann sich die Länge der Tour nicht vergrößern und es gilt $\mathrm{OPT} = d(C^\star) \geq d(C')$.

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 auch für das perfekte Matching M mit minimalem Gewicht $d(M)$. □

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

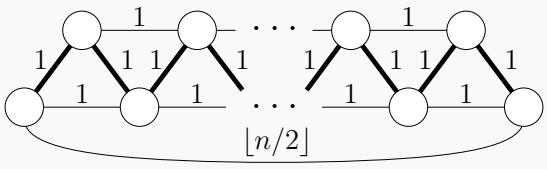
$$d(C) \leq d(T) + d(M) \leq \mathrm{OPT} + \frac{1}{2} \cdot \mathrm{OPT} = \frac{3}{2} \cdot \mathrm{OPT}.$$

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 1,5 der bestmögliche Approximationsfaktor, der für das metrische TSP unter der Annahme P $\neq$ NP erreicht werden kann? Diese Frage ist bis heute ungeklärt. Man kennt bisher keinen besseren Approximationsalgorithmus als den Christofides-Algorithmus. Auf der anderen Seite ist bekannt, dass es keinen $\frac{123}{122}$-Approximationsalgorithmus für das metrische TSP geben kann, falls P $\neq$ NP gilt. Die Lücke zwischen $\frac{123}{122} \approx 1{,}008$ und 1,5 zu schließen, ist ein großes offenes Problem der theoretischen Informatik und kombinatorischen Optimierung.

9.3 Euklidisches TSP

Wir betrachten nun das *euklidische TSP*. Dabei handelt es sich um den Spezialfall des metrischen TSP, bei dem die Knoten $v_1, \ldots, v_n \in \mathbb{R}^d$ Punkte im d-dimensionalen Raum sind und der Abstand zweier Punkte durch den euklidischen Abstand gegeben ist. Wir beschränken uns auf den zweidimensionalen Fall, ohne dies jedes Mal explizit zu erwähnen. Für zwei Punkte $a = (x_a, y_a) \in \mathbb{R}^2$ und $b = (x_b, y_b) \in \mathbb{R}^2$ gilt dann $d(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$. Auch dieser Spezialfall des TSP ist noch NP-schwer. Interessanterweise ist aber unklar, ob das euklidische TSP überhaupt in NP liegt, da die Länge einer gegebenen Tour eine Summe von Quadratwurzeln ist, die selbst dann irrational sein kann, wenn alle Koordinaten ganzzahlig sind. Es ist unklar, ob für eine Summe von Quadratwurzeln effizient geprüft werden kann, ob sie eine gewisse Schranke überschreitet.

Es existiert ein PTAS für das euklidische TSP. In diesem Abschnitt werden wir die wesentlichen Ideen, die diesem PTAS zugrunde liegen, diskutieren und ein Approximationsschema erhalten, mit dessen Hilfe für jedes konstante $\varepsilon > 0$ eine $(1+\varepsilon)$-Approximation in Zeit $O(n^{O(\log \log n)})$ berechnet werden kann. Wir werden nicht mehr im Detail diskutieren, wie der Exponent $O(\log \log n)$ in der Laufzeit auf $O(1)$ reduziert werden kann, um aus diesem Approximationsschema ein PTAS zu machen. Genauso wie die Approximationsschemata, die wir in Kapitel 3 konstruiert haben, basiert das Approximationsschema für das euklidische TSP darauf, die Instanz geeignet zu runden und dann mittels dynamischer Programmierung zu lösen. Die Umsetzung dieser Idee unterscheidet sich aber deutlich.

Zunächst zeigen wir, dass wir nicht beliebige Instanzen betrachten müssen, sondern nur solche die *ε-wohlgerundet* sind. Bei einer solchen Instanz sind die Koordinaten aller Punkte ganzzahlig aus dem Bereich $\{0, 1, \ldots, 8n/\varepsilon\}$ und jedes Paar von Punkten hat mindestens Abstand 4. Im Folgenden gehen wir davon aus, dass ein konstantes $\varepsilon > 0$

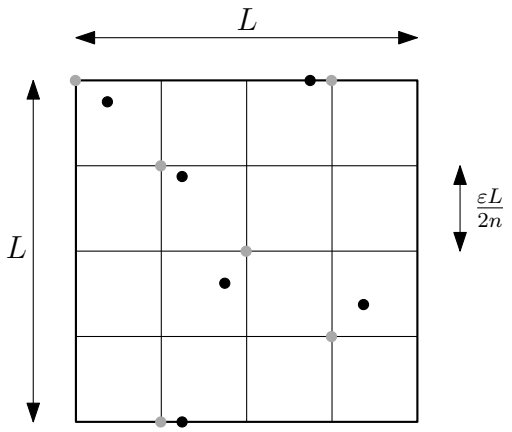


Abbildung 9.3: Die schwarzen Punkte entsprechen in diesem Beispiel der Eingabe V. Die grauen Punkte entsprechen der gerundeten Punktmenge V'.

gegeben ist, sodass $8n/\varepsilon$ ganzzahlig ist. Wir lassen das ε weg, wenn es aus dem Kontext hervorgeht, und sprechen einfach über wohlgerundete Instanzen.

Theorem 9.6. *Existiert ein Approximationsschema für ε-wohlgerundete Instanzen des euklidischen TSP mit einer Laufzeit von $O(f(n,\varepsilon))$, so existiert auch ein Approximationsschema für allgemeine Instanzen des euklidischen TSP mit einer Laufzeit von $O(f(n,\frac{3\varepsilon}{10}) + \text{poly}(n,\frac{1}{\varepsilon}))$.*

Beweis. Es sei $V = \{v_1, \ldots, v_n\} \subseteq \mathbb{R}^2$ eine beliebige Eingabe für das euklidische TSP mit $v_i = (x_i, y_i)$, und es sei $\varepsilon' > 0$ eine beliebige Konstante. Das Ziel ist es, für die Eingabe V eine $(1+\varepsilon')$-Approximation zu berechnen. Dazu transformieren wir V in eine ε-wohlgerundete Eingabe für ein später noch zu wählendes ε und nutzen das Approximationsschema, um für diese Eingabe eine ε-Approximation zu berechnen.

Wir betrachten zunächst das kleinste achsenparallele Quadrat, das die Eingabe V enthält. Dieses besitzt eine Seitenlänge von

$$L = \max\{\max_i x_i - \min_i x_i, \max_i y_i - \min_i y_i\}.$$

Wir teilen dieses Quadrat nun in ein Gitter ein, indem wir im Abstand von $\varepsilon L/(2n)$ horizontale und vertikale Linien einfügen. Anschließend runden wir jeden Punkt v_i auf den nächsten Gitterpunkt v_i'. Es sei V' die Menge der so gerundeten Punkte (siehe Abbildung 9.3). Wir verändern jede Koordinate jedes Punktes durch das Runden um maximal $\varepsilon L/(4n)$. Für jedes Paar von Punkten v_i und v_j gilt somit

$$|d(v_i,v_j) - d(v_i',v_j')| \leq d(v_i,v_i') + d(v_j,v_j') \leq 2 \cdot \sqrt{2\left(\frac{\varepsilon L}{4n}\right)^2} \leq \frac{\varepsilon L}{n},$$

wobei wir im ersten Schritt die Dreiecksungleichung ausgenutzt haben. Die Distanz zweier Punkte ändert sich durch das Runden also um höchstens $\varepsilon L/n$. Es sei noch angemerkt, dass es passieren kann, dass V' weniger Punkte als V enthält, da mehrere Punkte aus V auf denselben Gitterpunkt gerundet werden.

Im nächsten Schritt verschieben wir das Gitter und die Punkte aus V' so, dass die linke untere Ecke des Gitters im Ursprung des Koordinatensystems liegt, und wir skalieren das Gitter und die Punkte aus V' so, dass sie ganzzahlige Koordinaten erhalten. Dazu multiplizieren wir (nach dem Verschieben) jede Koordinate mit dem Wert $8n/(\varepsilon L)$. Die Punktmenge, die wir so erhalten, nennen wir V''. Die Koordinaten aller Punkte aus V'' sind ganzzahlig und durch $L \cdot \frac{8n}{\varepsilon L}$ nach oben beschränkt. Ferner beträgt der minimale Abstand zweier Punkte aus V'' mindestens $\frac{\varepsilon L}{2n} \cdot \frac{8n}{\varepsilon L} = 4$. Die Menge V'' ist somit ε-wohlgerundet.

Sei nun π die $(1+\varepsilon)$-Approximation für die Eingabe V'', die von dem Approximationsschema für wohlgerundete Instanzen berechnet wird. Wir können π auf kanonische Art auch als Tour auf V' auffassen, da es eine eindeutige Zuordnung zwischen den Punkten aus V' und V'' gibt. Die Tour π kann sogar als Tour auf V aufgefasst werden. Hierbei ist allerdings zu beachten, dass V mehr Punkte als V' enthalten kann, da mehrere Punkte von V auf denselben Gitterpunkt gerundet werden können. Ist dies der Fall, so besucht die Tour π diesen Gitterpunkt aus V' bzw. V'' nur einmal. Wir erweitern π dann, indem wir an der Stelle, an der π den Gitterpunkt besucht, alle Punkte aus V, die auf diesen Gitterpunkt abgebildet werden, in einer beliebigen Reihenfolge besuchen. Auf diese Weise ergibt sich aus π auch eine Tour auf V. Wir bezeichnen mit $d(\pi)$, $d'(\pi)$ und $d''(\pi)$ die Längen der Tour π bezogen auf V, V' bzw. V''.

Da wir π als $(1+\varepsilon)$-Approximation für die Eingabe V'' berechnet haben, gilt

$$d''(\pi) \leq (1+\varepsilon) \cdot \mathrm{OPT}(V'').$$

Da wir die optimale Tour auf V auch als Tour auf V' betrachten können, bei der wir ggf. Punkte überspringen, die auf denselben Gitterpunkt gerundet werden, können wir $\mathrm{OPT}(V')$ mithilfe von $\mathrm{OPT}(V)$ nach oben beschränken. Es gilt

$$\mathrm{OPT}(V'') = \frac{8n}{\varepsilon L} \cdot \mathrm{OPT}(V') \leq \frac{8n}{\varepsilon L} \cdot (\mathrm{OPT}(V) + \varepsilon L),$$

wobei die letzte Ungleichung daraus folgt, dass sich die Distanz zwischen zwei Punkten durch das Runden um höchstens $\varepsilon L/n$ ändert und jede Tour auf V genau n Kanten enthält. Analog folgt auch für die Tour π die erste Ungleichung in der folgenden Rechnung:

$$\begin{aligned} d(\pi) &\leq d'(\pi) + \varepsilon L \\ &= \frac{\varepsilon L}{8n} \cdot d''(\pi) + \varepsilon L \\ &\leq \frac{\varepsilon L}{8n} \cdot (1+\varepsilon) \cdot \mathrm{OPT}(V'') + \varepsilon L \\ &\leq \frac{\varepsilon L}{8n} \cdot (1+\varepsilon) \cdot \frac{8n}{\varepsilon L} \cdot (\mathrm{OPT}(V) + \varepsilon L) + \varepsilon L \\ &= (1+\varepsilon) \cdot \mathrm{OPT}(V) + ((1+\varepsilon)\varepsilon + \varepsilon)L. \end{aligned}$$

Wir nutzen nun noch die Beobachtung, dass $\mathrm{OPT}(V) \geq L$ gilt, da es zwei Punkte in V mit Abstand mindestens L gibt. Dies führt insgesamt zu der Abschätzung

$$d(\pi) \leq (1+\varepsilon) \cdot \mathrm{OPT}(V) + ((1+\varepsilon)\varepsilon + \varepsilon) \cdot \mathrm{OPT}(V) = (1 + 3\varepsilon + \varepsilon^2) \cdot \mathrm{OPT}(V).$$

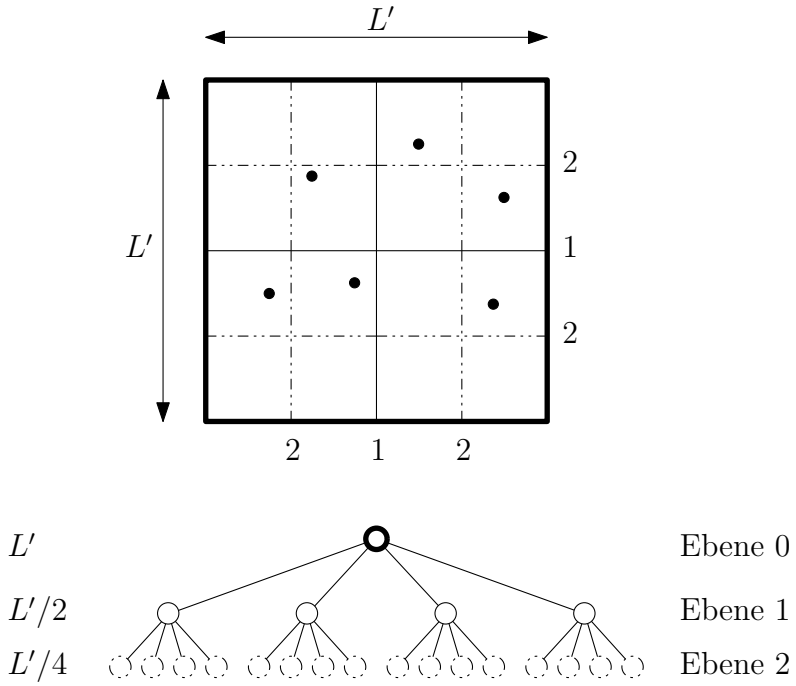


Abbildung 9.4: Beispiel für die ersten drei Ebenen der rekursiven Partition von V. Die Zahlen rechts und unter der Partition bezeichnen die Ebenen der Linien, die wir später definieren werden.

Damit ist gezeigt, dass es auch für allgemeine Instanzen ein Approximationsschema gibt, wenn es ein solches für wohlgerundete Instanzen gibt. Möchten wir eine $(1+\varepsilon')$-Approximation für die Instanz V berechnen, so genügt es mit dem obigen Argument, ε hinreichend klein zu wählen, sodass $3\varepsilon+\varepsilon^2 \leq \varepsilon'$ gilt. Dies ist für $\varepsilon = \frac{3\varepsilon'}{10}$ der Fall. □

Im Rest dieses Abschnittes beschäftigen wir uns mit dem Entwurf eines Approximationsschemas für wohlgerundete Instanzen des euklidischen TSP. Der erste Schritt besteht darin, dass wir eine gegebene wohlgerundete Instanz V mit $n = |V|$ durch ein Gitter rekursiv partitionieren. Sei dazu L wie oben die Seitenlänge des kleinsten achsenparallelen Quadrats, das V enthält, und sei L' die kleinste Zweierpotenz größer oder gleich $2L$. Wir betrachten nun ein zunächst beliebiges achsenparalleles Quadrat mit Seitenlänge L', das alle Punkte aus V enthält. Dieses teilen wir in vier Quadrate mit Seitenlänge $L'/2$ ein. Diese teilen wir wieder in jeweils vier Quadrate mit Seitenlänge $L'/4$ ein usw. Wir stoppen die Einteilung, sobald die Quadrate eine Seitenlänge von 1 haben. Eine solche Partition kann auf naheliegende Weise als Baum dargestellt werden. Die Wurzel des Baumes repräsentiert das gesamte Quadrat mit Seitenlänge L'. Jeder innere Knoten v hat genau vier Kinder, die die Teilquadrate repräsentieren, in die das Quadrat, das v repräsentiert, eingeteilt wird. Wir sagen, dass die Wurzel auf Ebene 0 liegt und zählen dann die Ebenen des Baumes von oben nach unten durch. Die Knoten auf Ebene i repräsentieren also Quadrate mit Seitenlänge $L'/2^i$. Abbildung 9.4 zeigt ein Beispiel für die ersten drei Ebenen einer solchen Partition.

Die Anzahl an Ebenen in der Partition beträgt $\log_2(L')+1 \leq \log_2(L)+3 = O(\log n)$, wobei wir ausgenutzt haben, dass für wohlgerundete Punkte $L = O(n)$ gilt. Weiter-

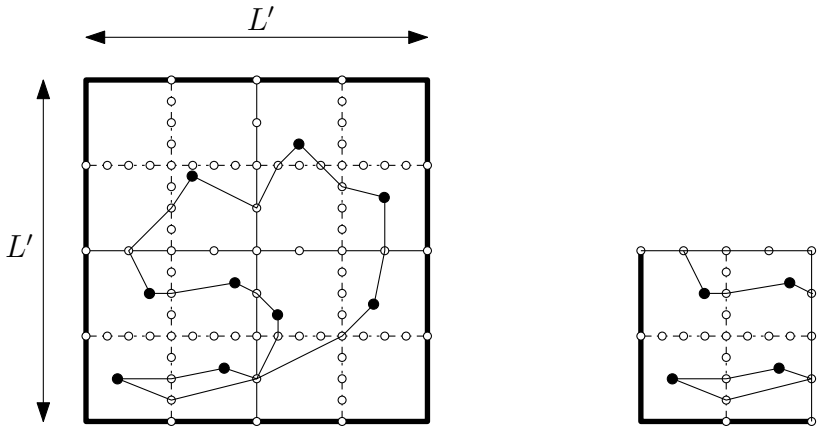


Abbildung 9.5: Links: Darstellung der Portale für $m = 4$ sowie einer 3-leichten Portaltour. Rechts: Partielle Portaltour des Quadrates links unten auf Ebene 1.

hin beträgt der minimale Abstand zweier Punkte aus der wohlgerundeten Eingabe V laut Voraussetzung mindestens 4 und damit enthalten die Quadrate auf der untersten Ebene mit Seitenlänge 1 jeweils nur höchstens einen Punkt aus V.

Wir werden für die Eingabe V eine Tour berechnen, die die Quadrate in der Partition nur an bestimmten dafür vorgesehenen *Portalen* betritt und verlässt. Dazu definieren wir zunächst für jedes Quadrat mit Ausnahme des Quadrates auf Ebene 0 eine Menge von Portalen. Sei $m = 2^z \geq 2$ ein später zu bestimmender Parameter. Für jedes Quadrat auf Ebene $i \geq 1$ betrachten wir seine beiden *inneren Seiten*, d. h. die Seiten, die keine Seiten eines Quadrates auf Ebene $i - 1$ sind (siehe Abbildung 9.6 für eine Illustration). Wir fügen auf jeder inneren Seite jeweils $m + 1$ äquidistant verteilte Portale ein. Weiterhin sind alle Portale der Quadrate auf Ebene $i-1$ auch Portale der entsprechenden Quadrate auf Ebene i. Abbildung 9.5 zeigt ein Beispiel.

Für die gegebene Eingabe V betrachten wir nur noch *Portaltouren*. Dabei handelt es sich um Touren, die alle Punkte aus V enthalten und die optional auch Portale (ggf. auch mehrfach) enthalten dürfen. Außerdem darf eine Portaltour nur über ein Portal von einem Quadrat in ein anderes gelangen. Eine Portaltour heißt *r-leicht*, wenn sie jede Seite jedes Quadrates aus der Partition höchstens r-mal überquert (siehe Abbildung 9.5 für ein Beispiel).

Theorem 9.7. *Eine optimale r-leichte Portaltour mit Parameter m kann für eine wohlgerundete Eingabe in Zeit $O((mr)^{O(r)} n \log n)$ berechnet werden.*

Beweis. Wir nutzen dynamische Programmierung, um das Theorem zu beweisen. Für eine Portaltour und ein Quadrat der Partition bezeichnen wir den Teil der Tour, der innerhalb des Quadrates verläuft, als *partielle Portaltour*. Da die Portaltour das Quadrat mehrfach betreten und verlassen kann, besteht eine partielle Portaltour im Allgemeinen aus mehreren knotendisjunkten (bezogen auf die Punkte aus V nicht auf die Portale) Wegen, die zusammen alle Punkte aus V in dem Quadrat enthalten (siehe Abbildung 9.5 für ein Beispiel). Enthält das betrachtete Quadrat bereits alle Punkte aus V, so entspricht die partielle Portaltour der gesamten Portaltour.

Eine r-leichte Portaltour überquert die vier Seiten jedes Quadrates zusammen nur maximal $4r$-mal. Somit besteht die partielle Portaltour jedes Quadrates aus maximal $2r$ knotendisjunkten Wegen. Wir betrachten zunächst nur solche Quadrate, die nicht alle Punkte aus V enthalten. Für jedes solche Quadrat Q definieren wir in dem dynamischen Programm mehrere Teilprobleme. Jedes dieser Teilprobleme ist durch eine Zahl $\ell \in \{1, 2, \ldots, 2r\}$ und eine Multimenge $\{(a_1, b_1), \ldots, (a_\ell, b_\ell)\}$ von Portalpaaren von Q beschrieben. In dem Teilproblem, das durch die Parameter Q, ℓ und $\{(a_1, b_1), \ldots, (a_\ell, b_\ell)\}$ beschrieben wird, suchen wir die kürzeste partielle r-leichte Portaltour innerhalb des Quadrates Q, die aus genau ℓ knotendisjunkten Wegen besteht, sodass für jedes i genau einer dieser Wege die Portale a_i und b_i verbindet. Da das Quadrat Q höchstens $4m$ Portale besitzt, gibt es für jedes a_i und b_i maximal $4m$ mögliche Wahlen. Somit beträgt die Anzahl an Teilproblemen für jedes Quadrat höchstens

$$\sum_{\ell=1}^{2r} (4m)^{2\ell} \leq (4m)^{4r+1} = m^{O(r)}.$$

Die rekursive Partition des Quadrates mit Seitenlänge L' besitzt, wie oben diskutiert, $O(\log n)$ viele Ebenen. Auf jeder Ebene kann es nur maximal $n = |V|$ Quadrate geben, die mindestens einen Punkt aus V enthalten. Somit gibt es insgesamt nur $O(n \log n)$ nichtleere Quadrate in der Partition. In dem dynamischen Programm müssen auch die leeren Quadrate betrachtet werden. Da sich die Lösungen der Teilprobleme für zwei leere Quadrate auf derselben Ebene aber nicht unterscheiden, genügt es, für jede Ebene nur ein leeres Quadrat zu betrachten. Insgesamt besitzt das dynamische Programm damit $O(m^{O(r)} n \log n)$ relevante Teilprobleme, die gelöst werden müssen.

Sei Q ein beliebiges Quadrat auf der untersten Ebene der Partition. Da die Instanz wohlgerundet ist, enthält Q höchstens einen Punkt aus V. Wir betrachten das Teilproblem, das durch die Parameter Q, ℓ und $\{(a_1, b_1), \ldots, (a_\ell, b_\ell)\}$ beschrieben wird. Enthält das Quadrat Q keinen Punkt aus V, so ist die kürzeste partielle Portaltour für die gegebenen Parameter einfach dadurch gegeben, dass jedes a_i auf direktem Wege mit dem entsprechenden b_i verbunden wird. Enthält das Quadrat Q einen Punkt aus V, so muss dieser auf einem der a_i-b_i-Wege besucht werden. Alle anderen Portale werden auf direktem Wege verbunden. Da es nur ℓ Möglichkeiten für die Wahl von i gibt, kann das Teilproblem in polynomieller Zeit gelöst werden.

Sei Q nun ein beliebiges Quadrat, das nicht auf der untersten Ebene der Partition liegt, und seien Q_1, Q_2, Q_3 und Q_4 die Quadrate, in die Q in der Partition eingeteilt wird. Wir nennen die vier Seiten von $Q_1, \ldots, Q_4$, die nicht auf den Seiten von Q liegen innere Seiten (siehe Abbildung 9.6). Wir betrachten das Teilproblem, das durch die Parameter Q, ℓ und $\{(a_1, b_1), \ldots, (a_\ell, b_\ell)\}$ beschrieben wird, und möchten die Lösung für dieses Teilproblems auf die Lösungen der Quadrate $Q_1, \ldots, Q_4$ zurückführen. Dazu beobachten wir, dass die optimale r-leichte partielle Portaltour für Q für die gegebenen Parameter jede innere Seite höchstens r-mal überqueren kann. Jede innere Seite besitzt $m + 1$ Portale, sodass es höchstens $(m + 2)^{4r}$ Möglichkeiten gibt, die inneren Portale auszuwählen, an denen die Überquerungen stattfinden können. (Wären es genau r Überquerungen pro innerer Seite, so gäbe es $(m + 1)^{4r}$ Möglichkeiten. Der Term $(m + 2)^{4r}$ trägt der Tatsache Rechnung, dass es auch weniger Überquerungen

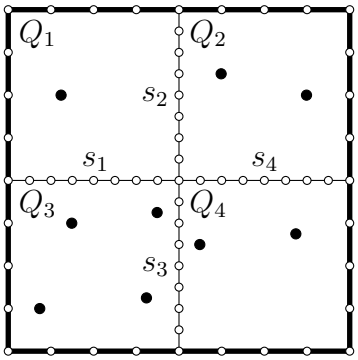


Abbildung 9.6: Darstellung der Teilquadrate $Q_1, \ldots, Q_4$ sowie der inneren Seiten $s_1, \ldots, s_4$ dieser Teilquadrate (Q_1 besitzt beispielsweise s_1 und s_2 als innere Seiten)

geben kann.) Zusätzlich zu der Auswahl der inneren Portale legen wir noch fest, auf welchen a_i-b_i-Wegen die inneren Portale besucht werden und in welcher Reihenfolge sie besucht werden. Für die Auswahl der a_i-b_i-Wege gibt es $\ell^{4r} \leq (2r)^{4r}$ Möglichkeiten und für die Reihenfolge gibt es $(4r)!$ Möglichkeiten. Alles in allem haben wir höchstens

$$(m+2)^{4r} \cdot (2r)^{4r} \cdot (4r)! = (mr)^{O(r)}$$

viele Möglichkeiten, die oben diskutierten Wahlen zu treffen.

Stehen die inneren Portale fest, sowie auf welchen a_i-b_i-Wegen sie in welcher Reihenfolge besucht werden, so ergibt sich für jedes der Quadrate $Q_1, \ldots, Q_4$ ein eindeutig beschriebenes Teilproblem, dessen Lösung wir in dem dynamischen Programm vorher berechnet haben. Wir wissen dann nämlich für jedes dieser Quadrate, in wie viele Wege die partielle Portaltour zerfällt und welche Portale diese Wege verbinden. Wir testen alle $(mr)^{O(r)}$ möglichen Wahlen und übernehmen die, die zu der günstigsten partielle Portaltour von Q führt. Pro Teilproblem führt dies zu einer Laufzeit von $(mr)^{O(r)}$. Da die Gesamtanzahl relevanter Teilprobleme $O(m^{O(r)} n \log n)$ beträgt, folgt das Theorem. □

Wir haben in diesem Abschnitt bisher argumentiert, dass wir uns bei dem Entwurf eines Approximationsschemas auf wohlgerundete Instanzen beschränkten können und dass wir für solche Instanzen mithilfe von dynamischer Programmierung optimale r-leichte Portaltouren in Zeit $O((mr)^{O(r)} n \log n)$ berechnen können. Wir müssen nun noch zeigen, dass die optimale r-leichte Portaltour für geeignete Wahlen von m und r eine gute Approximation der optimalen TSP-Tour ist.

Wir haben oben ein Quadrat mit Seitenlänge L' betrachtet, das alle Punkte aus V enthält. Dieses Quadrat ist nicht eindeutig, da $L' \geq 2L > L$ gilt. Da bei wohlgerundeten Instanzen die Koordinaten aller Punkte aus V ganzzahlig und nicht negativ sind, stammen alle Koordinaten aus dem Bereich $\{0, 1, \ldots, L\}$. Dementsprechend enthält jedes Quadrat mit Seitenlänge L', dessen linke untere Ecke an einer Position $(a, b) \in Z^2$ für $Z = \{-L'/2 + 1, -L'/2 + 2, \ldots 0\}$ liegt, die Punktmenge V. Wir nennen die oben beschriebene Partition der Instanz in Quadrate im Folgenden (a, b)-*Partition*, wenn die linke untere Ecke des initialen Quadrats mit Seitenlänge L' an Position (a, b) liegt.

Lemma 9.8. *Für ε-wohlgerundete Instanzen gilt: Werden a und b unabhängig uniform zufällig aus der Menge Z gewählt, so besitzt die (a, b)-Partition mit einer Wahrscheinlichkeit von mindestens $1/2$ eine r-leichte Portaltour mit Kosten höchstens $(1+\varepsilon)\cdot\mathrm{OPT}$ für die Parameter $m = O(\log(L')/\varepsilon)$ und $r = 2m + 2$.*

Beweis. Sei eine beliebige optimale Tour für die Instanz V gegeben. Wir werden diese nun zunächst in eine Portaltour umwandeln. Solange es in der Tour noch eine Kante (x, y) gibt, die die Grenze zweier Quadrate an einem Punkt z überschreitet, der kein Portal ist, ersetzen wir die Kante (x, y) durch die Kanten (x, p) und (p, y), wobei p das Portal auf der entsprechenden Grenze ist, das den kleinsten Abstand zu z besitzt. Wegen der Dreiecksungleichung ist der Weg x, p, y nicht länger als der Weg x, z, p, z, y und somit erhöht sich die Länge der Tour durch diese Transformation um maximal $2d(p, z)$.

Wir klären nun zunächst die Frage, wie stark sich die Tourlänge insgesamt dadurch erhöhen kann, dass wir alle Grenzübertritte auf die nächstgelegenen Portale verschieben. Die Verlängerung der Tour wird von zwei Faktoren beeinflusst, nämlich zum einen dadurch, wie oft die optimale Tour Grenzen zwischen verschiedenen Quadraten überschreitet, und zum anderen dadurch, wie weit die nächstgelegenen Portale von den Grenzübertritten entfernt sind. Wir betrachten zunächst eine beliebige vertikale oder horizontale Linie ℓ an einer x- bzw. y-Koordinate $i \in \mathbb{N}$. Enthält diese Linie Grenzen zwischen Quadraten der (a, b)-Partition, so definieren wir die Ebene der Linie als das kleinste i, sodass ℓ innere Seiten von Quadraten auf Ebene i enthält. Wir haben diese Ebenen bereits in Abbildung 9.4 illustriert. Je größer die Ebene einer Linie ist, desto kleiner ist der Abstand der Portale auf der Linie. Wir wollen nun zeigen, dass die meisten Grenzübertritte der optimalen Tour sehr wahrscheinlich auf Linien mit einer hohen Ebene passieren.

Zunächst beschäftigen wir uns mit der Frage, wie viele Grenzübertritte in der optimalen Tour überhaupt auftreten können. Für eine horizontale oder vertikale Linie ℓ bezeichne $t(\ell)$ wie oft die optimale Tour die Linie ℓ überschreitet. Es sei außerdem T die Summe aller $t(\ell)$ über alle horizontalen und vertikalen Linien ℓ, die sich an ganzzahligen x- bzw. y-Koordinaten befinden.

Lemma 9.9. *Es gilt $T \leq 2 \cdot \mathrm{OPT}$.*

Beweis. Wir betrachten für jede Kante der optimalen Tour, wie viel sie zu T beitragen kann. Sei eine beliebige Kante der optimalen Tour zwischen zwei Punkten (x_1, y_1) und (x_2, y_2) gegeben. Da wir nur Linien an ganzzahligen x- bzw. y-Koordinaten betrachten, kann die Kante maximal einen Beitrag von $|x_1 - x_2| + |y_1 - y_2| + 2$ zu T leisten. Sei s die Länge der betrachteten Kante, also $s = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Für $a, b \geq 0$ gilt $a^2 + b^2 \geq 2ab$ wegen $a^2 + b^2 - 2ab = (a - b)^2 \geq 0$. Daraus folgt

$$\sqrt{2(a^2 + b^2)} \geq \sqrt{a^2 + b^2 + 2ab} = \sqrt{(a + b)^2} = a + b.$$

Daraus folgt

$$|x_1 - x_2| + |y_1 - y_2| + 2 \leq \sqrt{2(|x_1 - x_2|^2 + |y_1 - y_2|^2)} + 2$$

$$\begin{aligned} &= \sqrt{2}s + 2 \\ &\leq 2s, \end{aligned}$$

wobei die letzte Ungleichung ausnutzt, dass $s \geq 4$ gilt, da die Instanz wohlgerundet ist. Das Lemma folgt nun einfach daraus, dass wir die oben gezeigte Ungleichung über alle Kanten der optimalen Tour summieren. □

Nun widmen wir uns der Frage, auf welchen Ebenen die Grenzübertritte der optimalen Tour stattfinden. Dazu beobachten wir zunächst, dass es genau 2^{i-1} vertikale und 2^{i-1} horizontale Linien auf Ebene $i \geq 1$ gibt (siehe Abbildung 9.4). Die zufällige Wahl von a und b aus der Menge Z impliziert für jede horizontale oder vertikale Linie ℓ und für jedes $i \geq 1$

$$\mathbf{Pr}\left[\text{Linie } \ell \text{ befindet sich auf Ebene } i\right] \leq \frac{2^{i-1}}{|Z|} = \frac{2^{i-1}}{L'/2} = \frac{2^i}{L'}.$$

Wir betrachten nun alle Kreuzungen der optimalen Tour mit der Linie ℓ und schätzen die erwartete Verlängerung der optimalen Tour ab, die dadurch entsteht, dass wir alle diese Kreuzungen jeweils auf das nächstgelegene Portal verschieben. Da die Portale einer Linie auf Ebene i den Abstand $\frac{L'}{2^i m}$ haben, beträgt diese höchstens

$$\begin{aligned} &\sum_{i=1}^{\log_2(L')} \mathbf{Pr}\left[\text{Linie } \ell \text{ befindet sich auf Ebene } i\right] \cdot t(\ell) \cdot \frac{L'}{2^i m} \\ &\leq \sum_{i=1}^{\log_2(L')} \frac{2^i}{L'} \cdot t(\ell) \cdot \frac{L'}{2^i m} \\ &= \sum_{i=1}^{\log_2(L')} \frac{t(\ell)}{m} = \frac{t(\ell)}{m} \cdot \log_2(L'). \end{aligned}$$

Wir wählen nun m als die kleinste Zweierpotenz größer oder gleich $\frac{4}{\varepsilon}\log_2(L')$. Dann ist der vorangegangene Term durch $\frac{\varepsilon}{4}t(\ell)$ nach oben beschränkt. Summieren wir dies über alle Linien ℓ, so erhalten wir

$$\sum_{\ell} \frac{\varepsilon}{4} t(\ell) = \frac{\varepsilon}{4} \sum_{\ell} t(\ell) = \frac{\varepsilon T}{4} \leq \frac{\varepsilon \cdot \mathrm{OPT}}{2}.$$

Aus der Markow-Ungleichung folgt, dass die Wahrscheinlichkeit, dass sich die Tourlänge um mehr als $\varepsilon \cdot \mathrm{OPT}$ erhöht, höchstens $1/2$ beträgt.

Als letztes müssen wir uns dem Problem widmen, dass die Portaltour, die wir erhalten, die Seite eines Quadrates beliebig oft kreuzen kann. Für $r = 2m + 2$ erhalten wir eine r-leichte Portaltour, wenn wir es schaffen, dass die Tour jedes Portal auf der Grenze zwischen zwei Quadraten höchstens zweimal überquert. Sei eine beliebige Portaltour gegeben. Solange es noch ein Portal p gibt, das zwischen zwei Quadraten mehr als zweimal überschritten wird, können wir zwei Kreuzungen entfernen, ohne die Länge der Tour zu verändern. Dazu zerlegen wir die Tour in verschiedene Segmente, die jeweils mit einem Besuch von p beginnen und enden. Wir müssen dann lediglich diese Segmente in der richtigen Reihenfolge durchlaufen, um eine Tour derselben Länge zu erhalten, die p weniger oft überschreitet. Abbildung 9.7 zeigt ein Beispiel, anhand dessen der Leser sich von der Korrektheit der Aussage überzeugen sollte. □

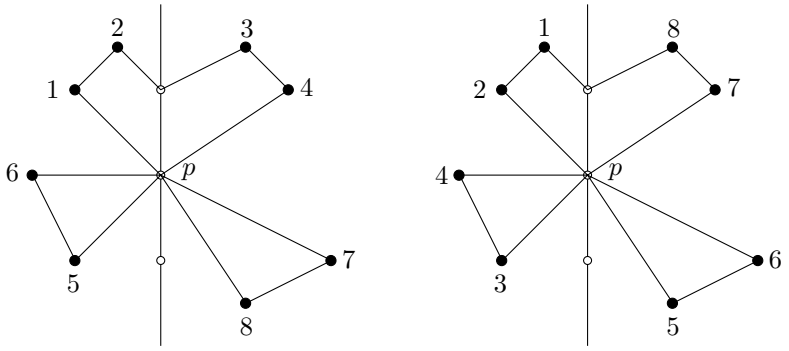


Abbildung 9.7: Beispiel wie durch das Entfernen zweier Kreuzungen an einem Portal p die Tour verkürzt werden kann

Aus unseren Vorüberlegungen folgt nun relativ leicht das folgende Theorem.

Theorem 9.10. *Es existiert ein Approximationsschema für das euklidische TSP, das für jede Konstante $\varepsilon > 0$ in Zeit $O(n^{O(\log\log n)})$ eine $(1+\varepsilon)$-Approximation berechnet.*

Beweis. Gemäß Theorem 9.6 können wir uns auf den Entwurf eines Approximationsschemas für wohlgerundete Instanzen beschränken. Theorem 9.7 besagt, dass eine optimale r-leichte Portaltour mit Parameter m für eine wohlgerundete Eingabe in Zeit $O((mr)^{O(r)} n \log n)$ berechnet werden kann. Setzen wir $m = O(\log(L')/\varepsilon)$ und $r = 2m + 2$ und nutzen aus, dass für wohlgerundete Instanzen $L' = O(n)$ gilt, so beträgt die Laufzeit

$$
\begin{aligned}
&O((2m^2+2m)^{O(2m+2)} n \log n) \\
&= O(m^{O(m)} n \log n) \\
&= O\left(\left(\frac{\log(L')}{\varepsilon}\right)^{O(\log(L')/\varepsilon)} n \log n\right) \\
&= O\left(\left(\frac{\log(n)}{\varepsilon}\right)^{O(\log(n)/\varepsilon)} n \log n\right) \\
&= O\left(\left(\frac{n^{\frac{\log\log n}{\log n}}}{\varepsilon}\right)^{O(\log(n)/\varepsilon)} n \log n\right) \\
&= O\left(n^{O(\frac{\log\log n}{\varepsilon})} \cdot \left(\frac{1}{\varepsilon}\right)^{O(\log(n)/\varepsilon)}\right).
\end{aligned}
$$

Aus Lemma 9.8 folgt, dass für die gewählten Parameter m und r mit einer Wahrscheinlichkeit von mindestens $1/2$ eine r-leichte Portaltour mit Kosten höchstens $(1+\varepsilon)\cdot\mathrm{OPT}$ existiert, wenn die Koordinaten a und b uniform zufällig aus Z gewählt werden. Da die Menge $Z = \{-L'/2+1, -L'/2+2, \ldots 0\}$ nur $L'/2 = O(n)$ Elemente enthält, können wir alle möglichen Wahlen von a und b in polynomieller Zeit testen und das beste Ergebnis nehmen. □

Kapitel 10

Untere Schranken für Approximierbarkeit

Zum Abschluss der Vorlesung beschäftigen wir uns noch einmal mit der Frage, wie man zeigen kann, dass gewisse Approximationsfaktoren für bestimmte Probleme nicht erreicht werden können. Zwei solche Beispiele haben wir in der Vorlesung bereits kennengelernt: In Theorem 3.10 haben wir gezeigt, dass es unter der Annahme P $\neq$ NP für das Problem Scheduling auf identischen Maschinen kein FPTAS gibt, und in Theorem 9.1 haben wir unter derselben Annahme gezeigt, dass es für das allgemeine TSP keinen 2^n-Approximationsalgorithmus gibt.

10.1 Reduktionen von NP-schweren Problemen

Die Grundidee zum Beweis der beiden oben zitierten Ergebnisse ist dieselbe. Wir haben gezeigt, dass wir mithilfe eines FPTAS für Scheduling auf identischen Maschinen oder mithilfe eines 2^n-Approximationsalgorithmus für das TSP ein NP-schweres Problem in polynomieller Zeit optimal lösen können. Wir werden in diesem Abschnitt zwei weitere solche Reduktionen diskutieren.

Theorem 10.1. *Sei $\varepsilon > 0$ beliebig. Unter der Annahme $P \neq NP$ existiert für das Bin-Packing-Problem kein $(3/2 - \varepsilon)$-Approximationsalgorithmus.*

Beweis. Sei A ein $(3/2-\varepsilon)$-Approximationsalgorithmus für das Bin-Packing-Problem. Wir zeigen, dass wir mithilfe von A das NP-vollständige Problem Partition in polynomieller Zeit optimal lösen können. Die Eingabe I für Partition besteht aus einer Menge $s_1, \ldots, s_n \in \mathbb{N}$ von Zahlen, für die entschieden werden soll, ob es eine Auswahl $J \subseteq \{1, \ldots, n\}$ gibt, für die $\sum_{i \in J} s_i = \sum_{i \notin J} s_i = T/2$ mit $T = \sum_{i=1}^{n} s_i$ gilt.

Wir konstruieren aus einer solchen Eingabe für Partition eine Eingabe für das Bin-Packing-Problem. Dabei übernehmen wir die Objektgrößen $s_1, \ldots, s_n$ und setzen die Kapazität der Bins auf $T/2$. Die so konstruierte Instanz des Bin-Packing-Problems bezeichnen wir mit I'. Ferner bezeichnen wir mit $\mathrm{OPT}(I')$ die minimale Anzahl an Bins, die wir benötigen, um die Objekte zu packen. Die Konstruktion stellt sicher,

dass $\mathrm{OPT}(I') = 2$ genau dann gilt, wenn die gegebene Eingabe für Partition eine Lösung besitzt. Ansonsten gilt $\mathrm{OPT}(I') \geq 3$.

Besitzt die Eingabe für Partition eine Lösung, so gibt A auf der Instanz I' für das Bin-Packing-Problem eine Lösung mit höchstens $(3/2 - \varepsilon) \cdot \mathrm{OPT}(I') = 3 - 2\varepsilon < 3$ vielen Bins aus. Da die Anzahl an Bins ganzzahlig ist, berechnet A also eine Lösung mit zwei Bins. Besitzt die Eingabe für Partition keine Lösung, so existiert auch keine Lösung für I' mit zwei Bins. Der Approximationsalgorithmus A gibt also eine Lösung mit mindestens drei Bins aus.

Wir können das NP-vollständige Problem Partition demnach in polynomieller Zeit lösen, indem wir die Instanz I' berechnen, auf diese den Approximationsalgorithmus A anwenden und anschließend testen, ob die ausgegebene Lösung mit zwei Bins auskommt. □

Als nächstes kommen wir noch einmal auf das Problem Scheduling auf allgemeinen Maschinen zurück. Für dieses Problem haben wir in Abschnitt 5.3 einen 2-Approximationsalgorithmus kennengelernt. Dies ist der derzeit beste bekannte Approximationsalgorithmus. Es ist ebenfalls bekannt, dass es für kein $\varepsilon > 0$ einen $(3/2 - \varepsilon)$-Approximationsalgorithmus für Scheduling auf allgemeinen Maschinen gibt. Es ist eine offene Frage, die Lücke zwischen der oberen und der unteren Schranke zu schließen. Wir zeigen nun die untere Schranke. Zunächst fangen wir mit dem Beweis einer etwas schwächeren Schranke an. Dazu benötigen wir als Ausgangspunkt für unsere Reduktion das folgende Entscheidungsproblem.

Dreidimensionales Matching (3-DM)

Eingabe:	Mengen $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$ Menge $F \subseteq A \times B \times C$ von Tripeln
Frage:	Existiert $F' \subseteq F$, sodass jedes Element aus $A \cup B \cup C$ in genau einem Tripel aus F' enthalten ist? Eine solche Menge nennen wir *dreidimensionales Matching*.

Es ist bekannt, dass 3-DM NP-vollständig ist.

Theorem 10.2. *Sei $\varepsilon > 0$. Unter der Annahme $P \neq NP$ existiert für das Problem Scheduling auf allgemeinen Maschinen kein $(4/3 - \varepsilon)$-Approximationsalgorithmus.*

Beweis. Es sei eine Instanz $I = (A, B, C, F)$ für 3-DM gegeben. Wir setzen $n = |A| = |B| = |C|$ und $m = |F|$. Für $m < n$ besitzt die Instanz I keine Lösung, wir können also im Folgenden $m \geq n$ annehmen. Basierend auf der Instanz I konstruieren wir eine Instanz I' für Scheduling auf allgemeinen Maschinen. Zunächst erzeugen wir für jedes Tripel aus der Menge F eine Maschine in der Instanz I', wir setzen also $M = F$ Des Weiteren fügen wir für jedes Element $j \in A \cup B \cup C$ einen Job in die Instanz I' ein. Wir setzen $p_{ij} = 1$ für alle Maschinen $i = (a, b, c) \in M = F$ mit $j \in \{a, b, c\}$. Für alle anderen Maschinen i setzen wir $p_{ij} = \infty$. Die Maschinen entsprechen also den Tripeln aus F und die Jobs den Elementen aus $A \cup B \cup C$. Ein Job hat auf den Tripeln, in denen er vorkommt, eine Länge von 1 und sonst ein Länge von ∞. Wir

schließen die Reduktion ab, indem wir noch $m - n$ Dummy-Jobs hinzufügen. Jeder dieser Dummy-Jobs hat auf jeder Maschine eine Länge von 3. Die Reduktion erzeugt insgesamt $3n + (m - n)$ Jobs und m Maschinen.

Gibt es in der Instanz I ein dreidimensionales Matching $F' \subseteq F$, so gibt es in der Instanz I' einen Schedule π mit Makespan 3. Es gilt $|F'| = n$ und $|F \setminus F'| = m - n$. Alle Maschinen aus $F \setminus F'$ erhalten in π einen der Dummy-Jobs. Damit haben diese Maschinen alle eine Ausführungszeit von 3 und alle Dummy-Jobs sind verteilt. Eine Maschine $i = (a, b, c) \in F'$ erhält in π die Jobs a, b und c. Diese haben alle eine Größe von 1 auf i. Somit haben auch alle Maschinen aus F' eine Ausführungszeit von 3. Ferner sind alle Jobs verteilt, da es sich bei F' um ein dreidimensionales Matching handelt. Insgesamt ist π somit ein Schedule mit Makespan 3.

Gibt es in der Instanz I' einen Schedule π mit Makespan 3, so gibt es in der Instanz I ein dreidimensionales Matching. Da es sich bei π um einen Schedule mit Makespan 3 handelt, müssen die Dummy-Jobs in π so verteilt sein, dass jeder Dummy-Job der einzige Job auf der Maschine ist, der er in π zugewiesen ist. Damit bleiben noch $m - (m - n) = n$ Maschinen für die anderen $3n$ Jobs, die den Elementen aus $A \cup B \cup C$ entsprechen. Sei $F' \subseteq F$ die Menge dieser Maschinen. Wir zeigen, dass F' ein dreidimensionales Matching ist. Da jede Maschine in F' eine Ausführungszeit von höchstens 3 besitzt, müssen die $3n$ Jobs, die den Elementen aus $A \cup B \cup C$ entsprechen, auf den Maschinen aus F' so verteilt sein, dass jede Maschine aus F' genau drei Jobs mit Größe 1 erhält. Das bedeutet, eine Maschine $i = (a, b, c) \in F'$ erhält genau die Jobs a, b und c. Da durch die Maschinen aus F' alle Jobs aus $A \cup B \cup C$ abgedeckt werden, handelt es sich bei F' um ein dreidimensionales Matching.

Es bezeichne $\mathrm{OPT}(I') \in \mathbb{N} \cup \{\infty\}$ den optimalen Makespan für die Instanz I'. Zusammengefasst folgt aus den beiden vorangegangenen Absätzen Folgendes: Besitzt die Instanz I für 3-DM eine Lösung, so gilt $\mathrm{OPT}(I') = 3$. Besitzt die Instanz I für 3-DM keine Lösung, so gilt $\mathrm{OPT}(I') > 3$ und damit wegen der Ganzzahligkeit $\mathrm{OPT}(I') \geq 4$. Das Theorem folgt nun genau wie Theorem 10.1: Sei A ein $(4/3-\varepsilon)$-Approximationsalgorithmus für Scheduling auf allgemeinen Maschinen. Besitzt die Instanz I für 3-DM eine Lösung, so berechnet A einen Schedule π mit einem Makespan $C(\pi)$ von höchstens $(4/3 - \varepsilon) \cdot \mathrm{OPT}(I') = 4 - 3\varepsilon < 4$. Wegen der Ganzzahligkeit gilt dann $C(\pi) = 3$. Besitzt die Instanz I keine Lösung, so berechnet A einen Schedule π mit $C(\pi) \geq 4$. Wir können also anhand der Ausgabe von A in polynomieller Zeit entscheiden, ob die Instanz I eine Lösung besitzt oder nicht. □

Wir können die Grundidee der obigen Reduktion übernehmen, um die folgende stärkere Aussage zu zeigen.

Theorem 10.3. *Sei $\varepsilon > 0$. Unter der Annahme $P \neq NP$ existiert für das Problem Scheduling auf allgemeinen Maschinen kein $(3/2 - \varepsilon)$-Approximationsalgorithmus.*

Beweis. Wir modifizieren die Reduktion aus dem Beweis von Theorem 10.2 zunächst wie folgt. Wir setzen die Ausführungszeit der Dummy-Jobs von 3 auf 2 und wir fügen nur noch für die Elemente aus $B \cup C$ Jobs in die Instanz I' ein. Für die Elemente aus A werden keine Jobs mehr erzeugt. Man sieht mit analogen Argumenten wie im Beweis

	Theorem 10.2	Theorem 10.3	Theorem 10.3
$T_1 = (1, 4, 7)$	1,4,7	4,7	Dummy
$T_2 = (1, 4, 8)$	Dummy	Dummy	4,8
$T_3 = (1, 5, 7)$	Dummy	Dummy	5,7
$T_4 = (2, 5, 8)$	2,5,8	5,8	Dummy
$T_5 = (2, 6, 9)$	Dummy	Dummy	Dummy
$T_6 = (3, 6, 9)$	3,6,9	6,9	6,9

Abbildung 10.1: Beispiel für die Reduktionen aus den Theoremen 10.2 und 10.3: Es sei $F = \{T_1, \ldots, T_6\}$ die Menge der Maschinen bzw. der Tripel. Es sei $A = \{1, 2, 3\}$, $B = \{4, 5, 6\}$ und $C = \{7, 8, 9\}$. Die zweite Spalte zeigt eine Zuweisung der Jobs aus der Reduktion in Theorem 10.2, die dem dreidimensionalen Matching $\{T_1, T_4, T_6\}$ entspricht. Die dritte Spalte zeigt eine Zuweisung der Jobs aus der Reduktion in Theorem 10.3, die ebenfalls dem dreidimensionalen Matching $\{T_1, T_4, T_6\}$ entspricht. Die vierte Spalte zeigt, dass das Weglassen der Jobs aus A, wie zu Beginn des Beweises von Theorem 10.3 diskutiert, dazu führt, dass optimale Schedules im Allgemeinen keinen dreidimensionalen Matchings mehr entsprechen.

von Theorem 10.2, dass die so konstruierte Instanz I' einen Schedule mit Makespan 2 besitzt, wenn es für die Instanz I von 3-DM eine Lösung gibt. Die andere Implikation ist aber nicht mehr gewährleistet, denn es kann nun auch dann einen Schedule mit Makespan 2 geben, wenn die Instanz I keine Lösung besitzt (siehe Abbildung 10.1 für eine Erläuterung).

Wenn man sich die letzte Spalte in Abbildung 10.1 anschaut, so stellt man Folgendes fest: Ein Schedule mit Makespan 2 entspricht im Allgemeinen keinem dreidimensionalen Matching mehr, da es nicht möglich ist, die Jobs aus A auf die ausgewählten (d. h. nicht mit Dummy-Jobs belegten) Maschinen zu verteilen. Um die Reduktion wieder korrekt zu machen, müssen wir also sicherstellen, dass es in jedem Schedule mit Makespan 2 für jeden Job aus A genau eine ausgewählte Maschine gibt. Dazu differenzieren wir die Menge der Dummy-Jobs. Für $a \in A$ sei

$$k_a = |\{T \in F \mid T \text{ ist von der Form } (a, *, *)\}|$$

die Anzahl der Tripel, in denen a enthalten ist. Statt der $m - n$ undifferenzierten Dummy-Jobs erzeugen wir für jedes $a \in A$ genau $k_a - 1$ Dummy-Jobs. Diese haben eine Größe von 2 auf allen Maschinen, die Tripeln entsprechen, die a enthalten, und eine Größe von ∞ auf allen anderen Maschinen. Es gilt $\sum_{a \in A}(k_a - 1) = |F| - n = m - n$, und somit ändert sich die Zahl der Dummy-Jobs nicht. Damit ist die Reduktion vollständig beschrieben.

Gibt es ein dreidimensionales Matching F', so gibt es einen Schedule π mit Makespan 2. Dieser wird ähnlich wie im Beweis von Theorem 10.2 erzeugt. Dabei werden die Jobs aus $B \cup C$ auf den Maschinen aus F' verteilt, sodass jede solche Maschine eine Ausführungszeit von 2 besitzt. Die Dummy-Jobs werden auf den anderen Maschinen wie folgt verteilt. Für jedes $a \in A$ gibt es in $F \setminus F'$ genau $k_a - 1$ viele Maschinen, die einem Tripel entsprechen, das a enthält. Die $k_a - 1$ zu a gehörenden Dummy-Jobs werden auf diesen Maschinen verteilt.

Gibt es einen Schedule π mit Makespan 2, so gibt es auch ein dreidimensionales Matching. In π gibt es für jedes $a \in A$ genau eine Maschine, die einem Tripel entspricht, das a enthält, und die keinen der $k_a - 1$ zu a gehörenden Dummy-Jobs erhält. Damit ist sichergestellt, dass es genau eine Maschine gibt, die einem Tripel entspricht, das a enthält, und die keine Dummy-Jobs enthält. Mit denselben Argumenten wie in Theorem 10.2 folgt diese Aussage auch für die Jobs aus B und C. Damit bildet die Menge der Maschinen, die keine Dummy-Jobs enthalten, ein dreidimensionales Matching.

Der Beweis kann nun analog zum Beweis von Theorem 10.2 abgeschlossen werden. □

10.2 Reduktionen mithilfe des PCP-Theorems

Konnten in der Anfangszeit der Komplexitätstheorie viele untere Schranken für die Approximierbarkeit von verschiedenen Optimierungsproblemen gezeigt werden, so stagnierte der Fortschritt zu Beginn der 1990er Jahre für viele grundlegende Optimierungsprobleme. So waren beispielsweise weder für Clique noch für Max-SAT nichttriviale untere Schranken für die Approximierbarkeit bekannt. Dies änderte sich mit der Entwicklung des PCP-Theorems, dem wir die letzte Vorlesung in diesem Semester widmen wollen.

Bei dem PCP-Theorem handelt es sich zunächst lediglich um eine alternative Charakterisierung der Klasse NP, die auf den ersten Blick nichts mit Approximationsalgorithmen oder Approximierbarkeit zu tun hat. Um es zu formulieren, erinnern wir uns an die Vorlesung „Algorithmen und Berechnungskomplexität II“, in der wir das folgende Theorem bewiesen haben.

Theorem 10.4. *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann in der Klasse* NP *enthalten, wenn es eine deterministische Turingmaschine V (einen* Verifizierer*), deren Worst-Case-Laufzeit polynomiell beschränkt ist, und ein Polynom p gibt, sodass für jede Eingabe $x \in \Sigma^*$ gilt*

$$x \in L \iff \exists y \in \{0,1\}^* : |y| \leq p(|x|) \text{ und } V \text{ akzeptiert } x\#y.$$

Dabei sei $\#$ ein beliebiges Zeichen, das zum Eingabealphabet des Verifizierers, aber nicht zu Σ gehört.

Dieses Theorem formalisiert die Intuition, dass ein Problem genau dann zu NP gehört, wenn die Korrektheit einer gegebenen Lösung effizient überprüft werden kann. Betrachten wir als Beispiel das Problem SAT. Eine Eingabe x für dieses Problem entspricht einer Formel in konjunktiver Normalform und $x \in$ SAT gilt per Definition genau dann, wenn die Formel erfüllbar ist. Wir konstruieren nun einen Verifizierer V, der als Eingabe eine Formel x sowie eine Variablenbelegung y erhält und ausgibt, ob y die Formel x erfüllt. Ist x erfüllbar, so gibt es eine Variablenbelegung y, für die der Verifizierer $x\#y$ akzeptiert. Ist x nicht erfüllbar, so akzeptiert der Verifizierer $x\#y$ für kein y. Für eine erfüllbare Formel x gibt es also ein polynomiell langes Zertifikat y, das effizient überprüft werden kann und belegt, dass $x \in$ SAT gilt.

Wir schränken den Verifizierer nun dahingehend ein, dass er nicht mehr das vollständige Zertifikat lesen darf. Dafür erlauben wir, dass er randomisiert arbeiten und mit einer gewissen Wahrscheinlichkeit das falsche Ergebnis liefern darf.

Definition 10.5. *Seien $r : \mathbb{N} \to \mathbb{N}$ und $q : \mathbb{N} \to \mathbb{N}$ zwei Funktionen. Ein $(r(n), q(n))$-Verifizierer V ist ein polynomieller Algorithmus, der bei Eingabe $x\#y$ mit $n = |x|$ wie folgt arbeitet:*

1. *V erzeugt unabhängig und uniform zufällig $r(n)$ viele Zufallsbits.*
2. *Abhängig von x und den erzeugten Zufallsbits berechnet V deterministisch $q(n)$ viele Positionen aus $\{1, 2, \ldots, |y|\}$ im Zertifikat y und liest diese aus.*
3. *Abhängig von x und den erzeugten Zufallsbits wählt V deterministisch eine Funktion $f : \{0,1\}^{q(n)} \to \{0,1\}$.*
4. *V setzt die $q(n)$ gelesenen Bits des Zertifikates y in die Funktion f ein und akzeptiert genau dann, wenn f den Wert 1 annimmt.*

Ein $(r(n), q(n))$-Verifizierer ist bei einer Eingabe der Länge n also dahingehend eingeschränkt, dass er nur $q(n)$ viele Bits des Zertifikates lesen darf. Welche er liest, hängt von den Zufallsbits ab, wovon er $r(n)$ viele erzeugen darf. Darauf basierend muss er entscheiden, ob er die Eingabe akzeptiert oder verwirft. Basierend darauf können wir nun die Klasse PCP definieren.

Definition 10.6. *Seien s und c mit $s < c$ sowie $r : \mathbb{N} \to \mathbb{N}$ und $q : \mathbb{N} \to \mathbb{N}$ beliebig. Eine Sprache $L \subseteq \Sigma^*$ gehört genau dann zu der Klasse $\mathrm{PCP}_{c,s}(r(n), q(n))$, wenn es einen polynomiellen $(r(n), q(n))$-Verifizierer und ein Polynom p gibt, sodass für jede Eingabe $x \in \Sigma^*$ die folgenden Aussagen gelten:*

$$x \in L \quad \Rightarrow \quad \exists y \in \{0,1\}^*, |y| \leq p(|x|) : \mathbf{Pr}\left[V \text{ akzeptiert } x\#y\right] \geq c$$

und

$$x \notin L \quad \Rightarrow \quad \forall y \in \{0,1\}^* : \mathbf{Pr}\left[V \text{ akzeptiert } x\#y\right] \leq s.$$

Sei $L \in \mathrm{NP}$ beliebig und sei p ein Polynom, das die Zertifikatslänge von L gemäß Theorem 10.4 beschreibt. Dann gilt $L \in \mathrm{PCP}_{1,0}(0, p(n))$. Es existiert also ein Verifizierer, der stets das richtige Ergebnis ausgibt, keine Zufallsbits benötigt und $p(n)$ viele Bits des Zertifikates liest. Das PCP-Theorem lautet wie folgt.

Theorem 10.7. *Es gibt eine Konstante k und eine Funktion $r(n) = O(\log n)$, sodass $\mathrm{NP} \subseteq \mathrm{PCP}_{1,1/2}(r(n), k)$.*

Die Aussage von Theorem 10.7 ist bemerkenswert. Selbst extrem eingeschränkte Verifizierer sind mächtig genug, um alle Sprachen aus NP zu entscheiden. Für jede Sprache L aus NP genügt es, nur konstant viele Positionen im Zertifikat y zu lesen, um $x \in L$ mit dem richtigen Zertifikat sicher zu erkennen und $x \notin L$ für jedes mögliche Zertifikat y zumindest mit Wahrscheinlichkeit $1/2$ zu verwerfen. Dabei dürfen nur logarithmisch viele Zufallsbits genutzt werden.

Der Beweis des PCP-Theorems ist sehr komplex und wir werden ihn hier nicht besprechen. Zum besseren Verständnis schauen wir uns aber noch einmal das Beispiel SAT an. Wir hatten oben bereits besprochen, dass man als Zertifikat für eine Eingabe für dieses Problem eine Variablenbelegung nehmen kann, die der Verifizierer dann darauf testet, ob sie die Formel erfüllt. Bedeutet das PCP-Theorem nun, dass es genügt, nur die Belegung konstant vieler Variablen im Zertifikat zu lesen, um mit genügend hoher Wahrscheinlichkeit die richtige Aussage zu liefern? Dies ist nicht der Fall, denn ist x eine Formel und y eine Variablenbelegung, die x nicht erfüllt, so erkennt man dies im Allgemeinen nicht, wenn man nur die Belegung von konstant vielen Variablen liest. Der Schlüssel zum Beweis des PCP-Theorems liegt in einer geeigneten Wahl des Zertifikats und des Verifizierers.

Mithilfe des PCP-Theorems können starke Aussagen über die Approximierbarkeit von vielen Optimierungsproblemen getroffen werden. Bis zur Entwicklung des PCP-Theorems war es nicht ausgeschlossen, dass es für Max-SAT ein PTAS gibt. Basierend auf dem PCP-Theorem zeigen wir nun, dass dies nicht der Fall ist.

Theorem 10.8. *Es gibt eine Konstante $\varepsilon > 0$, für die es keinen $(1-\varepsilon)$-Approximationsalgorithmus für* Max-SAT *gibt, falls $P \neq NP$.*

Beweis. Wir wählen eine beliebige NP-vollständige Sprache L aus. Mithilfe des PCP-Theorems werden wir zeigen, dass ein $(1-\varepsilon)$-Approximationsalgorithmus für Max-SAT für ein geeignetes $\varepsilon > 0$ impliziert, dass L in polynomieller Zeit entschieden werden kann. Sei x mit $n = |x|$ eine Eingabe für L. Für eine Konstante k und eine Funktion $r(n) = O(\log n)$ gilt $L \in \mathrm{PCP}_{1,1/2}(r(n), k)$ gemäß Theorem 10.7. Es gibt nur $N = 2^{r(n)} = 2^{O(\log n)} = n^{O(1)}$ viele verschiedene Möglichkeiten, die ausgewählten Zufallsbits zu belegen. Für jede Möglichkeit, liest der Verifizierer nur k viele Bits des Zertifikates y. Demzufolge sind nur kN viele Positionen in y überhaupt relevant und wir können uns auf Zertifikate der Länge kN beschränken. Wir können ein Zertifikat dann durch die Aussagenvariablen $y_1, \ldots, y_{kN} \in \{0,1\}$ beschreiben.

Wir konstruieren nun eine Eingabe φ für das Problem Max-SAT mit den Aussagenvariablen $y_1, \ldots, y_{kN}$. Die Klauseln gehen aus den Funktionen f hervor, die der Verifizierer berechnet. Gemäß Definition 10.5 gibt es für jede Wahl z der Zufallsbits eine Funktion f_z^x auf den gelesenen Zertifikatsbits, die der Verifizierer auswertet. Es gilt gemäß Theorem 10.7

- $x \in L \Rightarrow$ Es gibt ein Zertifikat y, das der Verifizierer für jede Belegung der Zufallsbits akzeptiert. Dieses Zertifikat $y_1, \ldots, y_{kN}$ führt also dazu, dass alle Funktionen f_z^x den Wert 1 annehmen.
- $x \notin L \Rightarrow$ Für jedes Zertifikat y führt höchstens die Hälfte der Belegungen der Zufallsbits dazu, dass der Verifizierer akzeptiert. Für jede Belegung $y_1, \ldots, y_{kN}$ nimmt also höchstens die Hälfte der Funktionen f_z^x den Wert 1 an.

Die Funktionen f_z^x hängen jeweils von höchstens k Aussagenvariablen ab. Sie können somit mit höchstens 2^k Klauseln der Länge k in konjunktiver Normalform dargestellt werden. Die Instanz φ von Max-SAT, die wir erzeugen, besteht aus den Darstellungen

der Funktionen f_z^x in konjunktiver Normalform für alle N möglichen Belegungen z der Zufallsbits. Dann gilt

- $x \in L \Rightarrow$ Alle Klauseln in φ erfüllbar.
- $x \notin L \Rightarrow$ Für jede Belegung der Aussagenvariablen nimmt höchstens die Hälfte der Funktionen f_z^x den Wert 1 an. Für mindestens die Hälfte der Funktionen ist also mindestens eine Klausel nicht erfüllt. Damit gibt es für jede Variablenbelegung mindestens $N/2$ nicht erfüllte Klauseln.

Die Gesamtzahl an Klauseln beträgt höchstens $2^k N$. Für Eingaben $x \notin L$ sind in jeder Belegung mindestens $N/2$ davon nicht erfüllt. Sei nun ein α-Approximationsalgorithmus für Max-SAT mit $\alpha > 1 - \frac{1}{2^{k+1}}$ gegeben. Ist φ erfüllbar und sei $A \leq N2^k$ die Anzahl an Klauseln in φ, so liefert der Approximationsalgorithmus eine Variablenbelegung, in der mindestens

$$\alpha A > A - \frac{A}{2^{k+1}} \geq A - \frac{N2^k}{2^{k+1}} = A - \frac{N}{2}$$

Klauseln erfüllt sind. Es sind in der Lösung, die A berechnet, also weniger als $N/2$ Klauseln nicht erfüllt. Dies impliziert $x \in L$. Damit folgt mit analogen Argumenten wie im Beweis von Theorem 10.1, dass L in polynomieller Zeit entschieden werden kann. □

Mit mehr Arbeit kann die Schranke noch deutlich verbessert werden. Unter der Annahme P $\neq$ NP kann beispielsweise für das Problem Max-3SAT, in dem jede Klausel die Länge mindestens 3 hat, gezeigt werden, dass es keinen Approximationsalgorithmus mit einem Approximationsfaktor größer als 7/8 gibt. Dies ist eine scharfe Schranke, da der einfache Algorithmus RandMaxSAT diesen Approximationsfaktor erreicht.

Zum Abschluss betrachten wir das Problem Max-Clique.

Max-Clique

Eingabe:	ungerichteter Graph $G = (V, E)$
Lösungen:	alle $S \subseteq V$, für die der induzierte Teilgraph $G[S]$ vollständig ist (Eine Menge S mit dieser Eigenschaft nennen wir *Clique*.)
Zielfunktion:	maximiere $\|S\|$

Gibt man einfach einen beliebigen Knoten des Graphen aus, so erreicht man trivialerweise einen Approximationsfaktor von $1/n$ für $n = |V|$. Der beste bekannte Approximationsalgorithmus für Max-Clique ist nur marginal besser und erreicht einen Approximationsfaktor von $O\left(\frac{\log^3 n}{n(\log\log n)^2}\right)$. Dennoch waren vor dem PCP-Theorem keine nichttrivialen unteren Schranken für die Approximierbarkeit von Max-Clique bekannt.

Theorem 10.9. *Falls P $\neq$ NP, so gibt es für keine Konstante $\varepsilon > 0$ einen $(\frac{1}{2} + \varepsilon)$-Approximationsalgorithmus für* Max-Clique.

Beweis. Der Beweis verläuft nach einem ähnlichen Muster wie der von Theorem 10.8. Wieder starten wir mit einer NP-vollständigen Sprache L und wenden das PCP-Theorem an. Wir nutzen dieselben Notationen wie im Beweis von Theorem 10.8 und konstruieren nun einen Graphen $G = (V, E)$ anstelle einer Formel φ.

Der Graph G enthält für jede der N möglichen Belegungen der Zufallsbits höchstens 2^k Knoten, und zwar einen für jede mögliche Belegung der k gelesenen Zertifikatsbits, die dazu führt, dass der Verifizierer akzeptiert. Die Knoten, die zu derselben Belegung der Zufallsbits gehören, nennen wir im Folgenden auch eine Gruppe. Für jede Belegung der Zufallsbits bildet jede Gruppe eine unabhängige Menge (es gibt also keine Kanten innerhalb einer Gruppe). Wir verbinden nur Knoten aus verschiedenen Gruppen. Jeder Knoten codiert die Belegung von k der Zertifikatsbits. Zwei Knoten aus verschiedenen Gruppen sind genau dann durch eine Kante verbunden, wenn sich die durch sie codierten Belegungen der Zertifikatsbits nicht widersprechen. Jede Clique in G codiert demnach eine Belegung einer Teilmenge der Zertifikatsbits.

Gilt $x \in L$, so gibt es eine Belegung y der Zertifikatsbits, sodass der Verifizierer für jede Belegung der Zufallsbits akzeptiert. Aus jeder Gruppe gibt es einen Knoten, der der Belegung y entspricht. Diese Knoten bilden gemeinsam eine Clique der Größe N.

Gilt $x \notin L$, so akzeptiert der Verifizierer jedes Zertifikat y höchstens mit einer Wahrscheinlichkeit $1/2$. Das heißt, für jede Belegung y gibt es höchstens in der Hälfte der Gruppen einen Knoten. Damit ist die Größe der größten Clique durch $N/2$ nach oben beschränkt.

Das Theorem folgt nun genauso wie Theorem 10.8. □

Auch für Max-Clique kann mit sehr viel mehr Aufwand mithilfe des PCP-Theorems eine noch bessere untere Schranke für die Approximierbarkeit gezeigt werden. Unter der Annahme P $\neq$ NP gibt es für kein $\varepsilon > 0$ einen $\Omega(n^{-1+\varepsilon})$-Approximationsalgorithmus für Max-Clique.

Literaturverzeichnis

[1] Michael R. Garey und David S. Johnson: **Computers and Intractability; A Guide to the Theory of NP-Completeness**. W. H. Freeman & Co., New York, NY, USA, 1990.

[2] Bernhard Korte und Jens Vygen: **Kombinatorische Optimierung**. Springer, Heidelberg, 2. Auflage, 2012.

[3] Vijay V. Vazirani: **Approximation Algorithms**. Springer, Heidelberg, 2001.

[4] David P. Williamson und David B. Shmoys: **The Design of Approximation Algorithms**. Cambridge University Press, New York, NY, USA, 1. Auflage, 2011.