

Theoretische Informatik SoSe 2003 – eine Kurzzusammenfassung

Bernd Gärtner

3. Juli 2003

Inhaltsverzeichnis

1	Elementare Wahrscheinlichkeitstheorie	2
1.1	Grundbegriffe	2
1.2	Coupon Collector's Problem	2
2	Suchbäume	2
2.1	Zufällige Suchbäume	2
2.2	Randomisierte Suchbäume	3
3	UNION-FIND-Datenstrukturen	3
3.1	Schnelle FINDs	3
3.2	Schnelle UNIONS	4
4	Schnitte in Graphen	4
4.1	Der Algorithmus von Stoer, Wagner, Nagamochi, Ibaraki	4
4.2	Randomisierter MinCut	4
5	Matchings	5
5.1	Augmentierende Wege	5
5.2	Maximum Matching in bipartiten Graphen	5
5.3	Perfekte Matchings in regulären bipartiten Graphen	6
5.4	Das Stable Marriage Problem	6
5.5	Der Heiratssatz	6
6	Online-Algorithmen	6
6.1	Das k -Paging-Problem	7
6.1.1	Deterministische Algorithmen	7
6.1.2	Randomisierte Algorithmen	7
7	External-Memory-Algorithmen	8
7.1	Sortieren	8

8	Algorithmen für NP-schwere Probleme	8
8.1	NP-Vollständige und NP-schwere Probleme	9
8.2	Das Rundreiseproblem	9
8.2.1	Exakte Lösungen	9
8.2.2	Approximationen	9
8.3	LP-Relaxierungen	10
8.3.1	MAX-SAT	10
8.3.2	Set Cover	10
8.4	Das Rucksackproblem	10

1 Elementare Wahrscheinlichkeitstheorie

1.1 Grundbegriffe

Wiederholung von Konzepten: Wahrscheinlichkeitsraum, Zufallsvariablen, Erwartungswert, Linearität des Erwartungswertes, bedingte Wahrscheinlichkeit, bedingter Erwartungswert, das Partitionstheorem, unfaire Münzwürfe (Bernoulli-Experiment)

1.2 Coupon Collector's Problem

In jeder Runde ziehen wir ein zufälliges von n Sammelbildern. Was ist die erwartete Anzahl von Runden, bis wir alle n Bilder haben? Bezeichne Y_i die erwartete Anzahl von Runden, die notwendig sind, um das i -te neue Bild zu ziehen. Nach der Analyse des Bernoulli-Experiments gilt

$$E(Y_i) = \frac{n}{n - i + 1},$$

woraus folgt, dass die gesuchte Rundenzahl im Erwartungswert

$$\sum_{i=1}^n \frac{n}{n - i + 1} = n \sum_{i=1}^n \frac{1}{i} = nH_n = O(n \log n)$$

beträgt. Bei $n = 100$ zum Beispiel müssen im Schnitt etwa 519 Bilder gezogen werden.

2 Suchbäume

2.1 Zufällige Suchbäume

Rekursive Definitionen von

- binären Suchbäumen
- perfekt balancierten Suchbäumen
- zufälligen Suchbäumen

Abschätzung diverser Parameter zufälliger Suchbäume mit n Elementen:

Parameter	Wert
Erwartete Tiefe des kleinsten Knotens	$H_n - 1 \leq \ln n$
Erwartete Tiefe des Knotens vom Rang i	$H_i + H_{n-i+1} \leq 2 \ln n$
Erwartete Gesamttiefe	$2(n+1)H_n - 4n = O(n \log n)$
Erwartete Höhe	$\leq 4.3 \ln n$

Zusammenhang zu Quicksort: Abschätzung für erwartete Gesamttiefe ist zugleich Abschätzung für erwartete Anzahl von Vergleichen in randomisiertem Quicksort.

Die Moral: zufällige Suchbäume verhalten sich im Durchschnitt fast so gut wie perfekt balancierte Suchbäume.

2.2 Randomisierte Suchbäume

Problem zufälliger Suchbäume: Verhalten nur gut bei zufälliger Einfügereihenfolge; uns interessiert aber die benutzerbestimmte Einfügereihenfolge.

Lösung durch *Treaps*: binärer Suchbaum bzgl. der Schlüssel, Heap bzgl. Prioritäten, die beim Einfügen zufällig gewählt werden. Nachteil: Rotationen notwendig, um Heap-Eigenschaft aufrechtzuerhalten.

Es gilt: für *jede* Einfügereihenfolge ist der entstehende Treap (oder auch *randomisierter Suchbaum*) ein zufälliger Suchbaum; die Sätze über das gute Verhalten der zufälligen Suchbäume sind also auf Treaps anwendbar.

3 UNION-FIND-Datenstrukturen

Problem: Aufrechterhalten einer Menge von Mengen unter den Operationen UNION (Vereinigung zweier Mengen) und FIND (Berechnen der Menge, die ein gegebenes Element enthält).

3.1 Schnelle FINDs

Feld, das für jedes Element i den Namen der Menge speichert, die i enthält. FIND hat Komplexität $O(1)$, UNION $O(n)$.

Verbesserung: verkette die Elemente einer Menge durch zusätzliche Zeiger. Beim Vereinigen zweier Mengen durchlaufe stets die *kleinere* und setze für alle ihre Elemente den Namen der Menge neu.

Es gilt: Die ersten m UNIONS kosten $O(m \log m)$ Zeit.

Beweis mit *amortisierter Analyse*: jedes Element, das durchlaufen wird, bezahlt einen Franken, ist danach aber in einer mindestens doppelt so grossen Menge. Es gibt nur m zahlende Elemente, jedes zahlt $O(\log m)$ Franken.

3.2 Schnelle UNIONS

Speicherung der Mengen als Bäume, Element in der Wurzel ist Namensgeber der Menge. UNION hat Komplexität $O(1)$, FIND hat Komplexität proportional zur Höhe des Baumes der Menge, im schlimmsten Fall $O(n)$.

Verbesserung 1: Mache beim UNION den Baum geringerer Höhe zum Kind des Baumes grösserer Höhe. Dann hat jeder Baum Höhe $O(\log n)$, und jedes FIND geht in $O(\log n)$.

Verbesserung 2: Mache beim FIND jedes durchlaufene Element zu einem Kind der Wurzel (Pfadverkürzen). Dann braucht eine Folge von m UNION- und FIND-Operationen Zeit $O(m \log^* n)$, wobei $\log^* n \leq 5$ für alle in der Praxis vorkommenden n .

4 Schnitte in Graphen

$G = (V, E, w)$ gewichteter Graph, $|V| = n, |E| = m$; für $S \subseteq V$ heisst das Paar $C = (S, V \setminus S)$ *Schnitt*. Das Gewicht $w(S)$ von S ist das Gesamtgewicht aller Kanten, die Knoten in S mit Knoten in $V \setminus S$ verbinden. Gesucht ist der Schnitt minimalen Gewichts.

Beobachtung: wenn der minimale Schnitt Knoten s und t nicht trennt, so können wir s und t *verschmelzen*; der resultierende Graph auf $n - 1$ Knoten hat den gleichen minimalen Schnitt.

4.1 Der Algorithmus von Stoer, Wagner, Nagamochi, Ibaraki

Finde minimalen (s, t) -Schnitt C' für beliebige s, t (MinCut-Phase); verschmelze s und t und berechne rekursiv den minimalen Schnitt C'' im Verschmelzungsgraphen. Nach der Beobachtung ist das Gewicht des global minimalen Schnittes gleich $\min(w(C'), w(C''))$.

MinCut-Phase: Berechnet Reihenfolge der Knoten; vorletzter und letzter Knoten in der Reihenfolge sind s und t . Als nächster Knoten wird jeweils derjenige gewählt, dessen Kanten zu den vorher gewählten Knoten in der Summe das grösste Gewicht haben.

Laufzeit insgesamt: $O((n^2 + nm) \log n)$ (verbesserbar auf $O(nm + n^2 \log n)$ mit Hilfe von Fibonacci-Heaps (bessere Prioritätsschlangen).

4.2 Randomisierter MinCut

Solange der Graph noch mehr als zwei Knoten hat, wähle jeweils eine Kante mit Wahrscheinlichkeit proportional zu ihrem Gewicht und verschmelze ihre beiden Knoten. Wenn nur noch zwei Knoten vorhanden sind, gib das Gewicht ihrer verbindenden Kante als Lösung aus.

Dieser Algorithmus liefert den minimalen Schnitt, wenn in keinem Schritt zwei Knoten verschmolzen worden sind, die vom minimalen Schnitt getrennt werden. Dieses 'schlechte' Ereignis hat in einem Graphen mit k Knoten Wahrscheinlichkeit höchstens $2/k$. Es folgt,

dass der Algorithmus mit Wahrscheinlichkeit mindestens

$$\frac{1}{\binom{n}{2}}$$

den minimalen Schnitt berechnet, und zwar in Zeit $O(n^2)$. Lassen wir den Algorithmus $O(n^2 \log n)$ -mal laufen, erhalten wir einen $O(n^4 \log n)$ -Algorithmus mit Fehlerwahrscheinlichkeit nur noch

$$\frac{1}{n^c}$$

für beliebiges vorgegebenes c .

Dieser Algorithmus ist schlechter als der exakte Algorithmus, aber durch *Bootstrapping* können wir die Laufzeit schrittweise verbessern. Idee: sobald der Graph zu klein wird (Anzahl der Knoten erreicht einen Wert t), fahren wir mit dem exakten Algorithmus fort. Für $t = n^{2/3}$ erhalten wir dann bereits einen Algorithmus mit Laufzeit $O(n^{8/3} \log n)$, was besser ist als die Laufzeit des exakten Algorithmus. Setzen wir bei der Schranke t nun diesen verbesserten Algorithmus ein, erhalten wir für $t = n^{3/4}$ bereits $O(n^{5/2} \log n)$. Dies können wir fortsetzen, um schliesslich für jedes $\varepsilon > 0$ einen $O(n^{2+\varepsilon})$ -Algorithmus zu erhalten.

5 Matchings

Matching in $G = (V, E)$: Menge von Kanten $M \subseteq E$, so dass jeder Knoten zu höchstens einer Kante inzident ist. Ziel: finde grösstmögliches (maximum) Matching.

5.1 Augmentierende Wege

Augmentierender Weg: Folge von Knoten; erster und letzter sind frei (nicht inzident zu Matching-Kante); aufeinanderfolgende Knoten sind durch Kanten verbunden, wobei sich Nicht-Matching- und Matching-Kanten abwechseln.

Klar ist: wenn es einen augmentierenden Weg gibt, kann das Matching vergrössert werden durch 'Flippen' der Kanten des Weges. Es gilt aber sogar

Satz: M ist genau dann ein maximum Matching, wenn es keinen augmentierenden Weg bzgl. M gibt.

5.2 Maximum Matching in bipartiten Graphen

Solange es geht, finde augmentierenden Weg und vergrössere das Matching. Das Finden eines augmentierenden Weges kann auf das Finden eines gerichteten Pfades zwischen Quelle und Senke in einem leicht konstruierbaren Hilfsgraphen zurückgeführt werden.

Laufzeit: $O(n^3)$ (n Augmentierungsschritte, jeder geht in $O(n^2)$ Zeit)

5.3 Perfekte Matchings in regulären bipartiten Graphen

Satz: Jeder k -reguläre bipartite Graph hat ein perfektes Matching (alle Knoten sind inzident zu einer Matching-Kante).

Beweis durch Algorithmus: Weise allen Kanten Gewichte zwischen 0 und k zu, so dass die Gewichtssumme an jedem Knoten k ist. Zu Beginn sind alle Gewichte gleich 1. Ziel: alle Gewichte sind 0 oder k , und das definiert ein perfektes Matching.

Algorithmus: Solange es eine Kante mit Gewicht ungleich 0 oder k gibt, so gibt es einen Kreis aus solchen Kanten. Alternierend erhöhen und erniedrigen wir die Kantengewichte entlang des Kreises um 1, so dass die Kantensummen an den Knoten gleich bleiben. Machen wir dies so, dass wir diejenigen Kanten erhöhen, die in der Summe bereits das grössere Gewicht haben, kann gezeigt werden, dass die Potentialfunktion

$$\sum_{e \in E} w(e)(k - w(e))$$

um mindestens die Länge des Kreises abnimmt. Nach endlich vielen Schritten ist das Potential somit auf Null gefallen, und dies entspricht dem gewünschten Ergebnis, dass alle Gewichte entweder 0 oder k sind.

Laufzeit: $O(mk)$.

5.4 Das Stable Marriage Problem

Gegeben sind n Frauen und n Männer; jede Frau / jeder Mann hat eine nach Sympathie geordnete Liste aller Männer / Frauen. Ziel ist es, eine stabile Paarung zu finden. Das bedeutet, es gibt keine Frau F und keinen Mann M , die nicht zusammen sind, sich aber gegenseitig sympathischer finden als ihre jeweiligen Partner.

Die Existenz einer stabilen Paarung kann mit einem Algorithmus gezeigt werden.

5.5 Der Heiratssatz

Dies ist eine Version des Stable Marriage Problem ohne Sympathiewerte. Gegeben sind n Frauen und m Männer, $n \leq m$; ferner gibt es eine Liste von Frau/Mann-Paaren, die mögliche Partnerschaften darstellen. Unter welchen Bedingungen ist es möglich, alle Frauen unter die Haube zu bringen? Die Antwort gibt der

Heiratssatz: es gibt genau dann ein Matching, das alle Frauen abdeckt, wenn jede Teilmenge \mathcal{F} der Frauen insgesamt mindestens $|\mathcal{F}|$ Männer kennt. Der Beweis geht durch Induktion über die Anzahl der Frauen.

6 Online-Algorithmen

Hier geht es darum, Algorithmen für Probleme zu finden, bei denen die Eingabe nicht vollständig bekannt ist, sondern stückweise kommt. Die Worst-Case-Analyse ist hier nicht sehr sinnvoll, sondern wir verwenden die *kompetitive* Analyse, bei der die Leistung eines *Online*-Algorithmus zur Leistung eines optimalen *Offline*-Algorithmus in Beziehung

gesetzt wird. Der Offline-Algorithmus kennt im Gegensatz zum Online-Algorithmus die komplette Eingabe, kann also besser sein als der Online-Algorithmus.

6.1 Das k -Paging-Problem

Ein Cache mit Platz für k Seiten muss unter Anfragen an Speicherseiten aufrechterhalten werden. Jede angefragte Seite muss—wenn sie nicht im Cache ist—in den cache geladen werden, und eine andere Seite muss den Cache dafür verlassen. Ziel ist es, die Anzahl der *cache misses* (Zugriffe auf Seiten nicht im Cache) zu minimieren.

6.1.1 Deterministische Algorithmen

Beobachtung: Im schlechtesten Fall führt bei jedem deterministischen Algorithmus jeder Zugriff zu einem cache miss (denn man muss ja immer nur die Seite anfragen, die gerade den Cache verlassen hat).

Sei σ eine Folge von Anfragen und sei $\text{OPT}(\sigma)$ die minimale Anzahl von cache misses, die in einem offline-Algorithmus für σ auftreten müssen. Ein deterministischer Online-Algorithmus A heisst c -kompetitiv, falls es ein b gibt, so dass für alle σ

$$A(\sigma) \leq c \cdot \text{OPT}(\sigma) + b$$

gilt, wobei $A(\sigma)$ die Anzahl der cache misses von A auf σ ist.

Satz 1: Der Algorithmus LRU (least recently used), der stets diejenige Seite aus dem cache wirft, die am längsten nicht angefragt wurde, ist k -kompetitiv.

Satz 2: Kein deterministischer Paging-Algorithmus kann c -kompetitiv sein für $c < k$.

6.1.2 Randomisierte Algorithmen

Für deterministischer Algorithmen ist das Problem also gelöst. Es bleiben randomisierte Verfahren. Ein randomisierter Online-Algorithmus A heisst c -kompetitiv, falls es ein b gibt, so dass für alle σ

$$E(A(\sigma)) \leq c \cdot \text{OPT}(\sigma) + b,$$

wobei $E(A(\sigma))$ die *erwartete* Anzahl von cache misses ist, die A auf σ produziert.

Der randomisierte Algorithmus MARKER ist $2H_k$ -kompetitiv. Da $2H_k = O(\log k)$, ist der kompetitive Faktor dieses Algorithmus *exponentiell besser* als der Faktor jedes deterministischen Algorithmus.

MARKER geht in Runden vor: zu Beginn einer Runde sind alle k Speicherseiten unmarkiert. Wird eine Seite im Cache angefragt, oder wird eine Seite in den Cache geladen, wird sie markiert; im letzteren Fall verlässt eine *zufällige* unmarkierte Seite den Cache. Sind bereits alle Seiten markiert, wenn eine neue Seite geladen wird, so werden zunächst alle Markierungen gelöscht, und es beginnt eine neue Runde.

7 External-Memory-Algorithmen

Hier geht es um Algorithmen für Eingaben, die zu gross sind, um in den Hauptspeicher zu passen. Während der Verarbeitung müssen also Teile von der Platte nachgeladen und Zwischenergebnisse auf die Platte geschrieben werden. Das Komplexitätsmass ist die Anzahl der Ein/Ausgabeoperationen zwischen Hauptspeicher und Festplatte. Wir definieren die folgenden Symbole:

- N ist die Anzahl der zu verarbeitenden Dateneinheiten;
- M ist die Anzahl der Dateneinheiten, die im Hauptspeicher verarbeitet werden können;
- B ist die Blockgrösse (Anzahl der Dateneinheiten, die bei einer Ein/Ausgabeoperation transferiert werden können).

Ferner setzen wir

$$n := \frac{N}{B}, \quad m := \frac{M}{B}.$$

7.1 Sortieren

Satz: Eine Menge von N Dateneinheiten kann mit

$$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) = O(n \log_m n)$$

Ein/Ausgabeoperationen sortiert werden.

Der Algorithmus dazu ist eine Variante von *Mergesort*. Hier ist die wesentliche Idee. In einem ersten Schritt wird die Eingabe blockweise geladen, sortiert und wieder auf die Platte geschrieben. Nun haben wir n sortierte Blöcke auf der Platte. Jeweils m von diesen laden wir in den Speicher und *vermischen* sie zu einem sortierten *Superblock*, den wir blockweise wieder auf die Platte schreiben. Nachdem wir dies für alle Blöcke gemacht haben, gibt es $\approx n/m$ sortierte Superblöcke. In jeder Runde vermischen wir nun jeweils m der Superblöcke zu einem neuen Superblock, wobei wir beobachten, dass wir von jedem der m Superblöcke stets nur einen Block im Speicher halten müssen.

Da die Anzahl der Superblöcke in jeder Runde um den Faktor m abnimmt, haben wir nach $O(\log_m n)$ Runden die gesamte Eingabe sortiert. Jede Runde (Mischvorgang) benötigt dabei $O(n)$ Ein/Ausgabeoperationen.

8 Algorithmen für NP-schwere Probleme

Hier geht es darum, Probleme, die beweisbar schwer sind, 'trotzdem' zu lösen, entweder in nicht ganz so exponentieller Zeit, oder approximativ in polynomieller Zeit.

8.1 NP-Vollständige und NP-schwere Probleme

P ist die Klasse der Entscheidungsprobleme, die in polynomieller Zeit gelöst werden können. NP ist die Klasse der Entscheidungsprobleme, bei denen es genau für jede Ja-Antwort einen Beweis gibt, der in polynomieller Zeit verifiziert werden kann. (Beispiel: wir können beim Rundreiseproblem schnell prüfen, ob es eine Rundreise der Länge höchstens k gibt, wenn wir die kürzestmögliche Rundreise als Beweis in der Hand haben.)

Ein Entscheidungsproblem in NP ist NP-vollständig, wenn man alle Probleme in NP in polynomieller Zeit auf dieses Problem reduzieren kann. Ein Optimierungsproblem ist NP-schwer, wenn ein polynomieller Algorithmus für das Problem polynomielle Algorithmen für alle Probleme in NP nach sich ziehen würde.

8.2 Das Rundreiseproblem

Gegeben ist ein *vollständiger* gewichteter Graph $G = (V, E, w)$, $|V| = n$, mit nichtnegativen Kantengewichten. Gesucht ist eine Rundreise durch alle Knoten, die das Gesamtgewicht der bereisten Kanten minimiert. Das Problem ist NP-schwer.

8.2.1 Exakte Lösungen

Die offensichtliche Lösung probiert alle $(n - 1)!$ Rundreisen durch, die bei einem festen Knoten beginnen und enden. Für jede wird in $O(n)$ Zeit die Länge berechnet und dann das Minimum ausgegeben. Dies ist ein $O(n!)$ -Algorithmus.

Man kann dies mit *dynamischem Programmieren* verbessern. Sei $V = \{0, \dots, n - 1\}$. Für $S \subseteq \{1, \dots, n - 1\}$ und $i \in S$ definiere $W(S, i)$ als Länge der kürzesten Reise von 0 nach i unter Benutzung genau der Knoten in $0 \cup S$. Die Länge der kürzesten Rundreise ist dann

$$\min_{i=1}^{n-1} (W(\{1, \dots, n - 1\}, i) + w(\{0, i\})).$$

Ausserdem gilt (für $|S| \geq 2$)

$$W(S, i) = \min_{k \in S \setminus \{i\}} (W(S \setminus \{i\}, k) + w(\{k, i\})),$$

so dass die Werte $W(S, i)$ für alle S und i in Zeit $O(n^2 2^n)$ induktiv berechnet werden können.

8.2.2 Approximationen

Wir nehmen an, dass die Gewichtsfunktion w die Dreiecksungleichung erfüllt.

Der erste Algorithmus berechnet zunächst den minimal-spannenden Baum und verdoppelt diesen. Im resultierenden Doppelbaum hat jeder Knoten geraden Grad, so dass es einen Eulerkreis gibt (Kreis, der alle Kanten genau einmal durchläuft). Wir geben dann die ersten Auftreten der Knoten entlang des Eulerkreises als Rundreise aus. Dabei machen wir gegenüber dem Eulerkreis nur Abkürzungen, so dass die Rundreise höchstens doppelt so lang ist wie der minimal-spannende Baum. Dieser wiederum ist höchstens so

lang wie eine optimale Rundreise. Das heisst, die berechnete Rundreise ist höchstens doppelt so lang wie die optimale: wir haben eine 2-Approximation.

Der Algorithmus von *Christofides* verbessert den Faktor auf $3/2$. Anstatt den minimalspannenden Baum zu verdoppeln, fügt er ein perfektes Matching kleinsten Gewichts auf den Knoten ungeraden Grades im Baum hinzu. Da das Matching vom Gewicht her höchstens die Hälfte der optimalen Rundreise ausmacht, hat der resultierende Eulerkreis eine Länge, die höchstens um den Faktor 1.5 über dem Optimum liegt.

8.3 LP-Relaxierungen

In einem linearen Programm wollen wir eine lineare Zielfunktion in n Variablen unter m linearen Ungleichungen maximieren oder minimieren. Dieses Problem liegt in P. Verlangen wir aber, dass die Variablen ganzzahlige Werte annehmen, so wird das Problem NP-schwer. Oft kann man aber ein schweres Problem als ganzzahliges lineares Programm formulieren, die LP-Relaxierung (Weglassen der Ganzzahligkeitsbedingungen) lösen, und dann versuchen, diese Lösung irgendwie auf ganzzahlige Werte zu runden.

8.3.1 MAX-SAT

Gegeben eine boolesche Formel in konjunktiver Normalform, finde eine Belegung der Variablen, die möglichst viele der Klauseln erfüllt. Der triviale randomisierte Algorithmus (belege die Variablen zufällig) erreicht einen erwarteten Approximationsfaktor von $1/2$ (d.h. wenn s Klauseln erfüllbar sind, erfüllt der Algorithmus im Schnitt mindestens $s/2$).

Fasst man das Problem als ganzzahliges LP auf (mit Variablen in $\{0, 1\}$) und löst die Relaxierung, erhält man Variablenwerte zwischen 0 und 1. Diese kann man als Wahrscheinlichkeiten interpretieren, mit denen die zugehörigen booleschen Variablen auf *wahr* gesetzt werden. Es zeigt sich, dass dieser Algorithmus einen Approximationsfaktor von mindestens $1 - 1/e \approx 0.63$ erreicht. Der bessere der beiden Algorithmen (trivial und *randomisiertes Runden*) erreicht sogar Faktor $3/4$.

8.3.2 Set Cover

Gegeben eine Grundmenge mit n Elementen und ein System von Teilmengen, finde die minimale Anzahl von Teilmengen, die die Grundmenge überdecken. Mittels LP-Relaxierung und randomisiertem Runden kann eine Anzahl von Teilmengen gefunden werden, die zwar im Erwartungswert die optimale Grösse hat, jedes Element aber nur mit Wahrscheinlichkeit $1 - 1/e$ abdeckt. Indem wir den Prozess logarithmisch oft wiederholen und die erhaltenen partiellen 'überdeckungen' vereinigen, erhalten wir mit hoher Wahrscheinlichkeit eine Überdeckung, deren Grösse um den Faktor $O(\log n)$ über der optimalen Grösse liegt.

8.4 Das Rucksackproblem

Ein Dieb findet bei einem Einbruch n Gegenstände mit Werten w_1, \dots, w_n und Gewichten g_1, \dots, g_n vor. Sein Rucksack kann Gesamtgewicht b aufnehmen. Welche Gegenstände muss

der Dieb auswählen, um einerseits den Wert seiner Beute zu maximieren, andererseits aber die Rucksackkapazität nicht zu überschreiten?

Wie beim Rundreiseproblem gibt es eine exakte Lösung mittels dynamischem Programmieren, die zu *pseudopolynomieller* Laufzeit $O(n \cdot \text{opt})$ führt, wobei opt der vom Dieb erreichbare Wert ist. Dieser ist im allgemeinen aber nicht polynomiell in der Grösse der Eingabe.

Den exakten Algorithmus nehmen wir nun als Basis für einen Approximationsalgorithmus, der für festes $\varepsilon > 0$ einen Wert berechnet, der höchstens um einen Faktor $(1 - \varepsilon)$ unter dem optimalen Wert liegt.

Der Algorithmus teilt zunächst alle Werte durch den Faktor $k = \max(1, \varepsilon w_{\max}/n)$ und rundet ab, um wieder ganzzahlige Werte zu erzielen. w_{\max} ist dabei das grösste Gewicht in der ursprünglichen Instanz. Die skalierte Instanz wird nun mit dem exakten Algorithmus gelöst, und die ausgewählte Teilmenge von Gegenständen wird auch als Lösung für das Originalproblem ausgegeben. Man kann zeigen, dass der Optimalwert der skalierten Instanz polynomiell gross ist, so dass der exakte Algorithmus ein polynomieller Algorithmus ist. Andererseits kann gezeigt werden, dass wirklich ein Approximationsfaktor von $(1 - \varepsilon)$ gilt.