Algorithmen und Datenstrukturen 2

Sándor P. Fekete
LaTeX Version: Arne Schmidt

23. Juni 2026

Institut für Betriebsysteme und Rechnerverbund
Abteilung Algorithmik

Inhaltsverzeichnis

1 Einführung: Knapsack-Probleme **1**
1.1 Aufgabenstellungen 1
1.2 Ein Beispiel 7

2 Dynamisches Programmieren **10**
2.1 Subset Sum 10
2.2 Dynamic Programming für Rucksackprobleme 11

3 Branch and Bound **15**
3.1 Linear Programming 20
3.2 Integer Programming 20

4 Approximationen **22**

5 Komplexität **25**
5.1 Einstieg 25
5.2 Ein Beispiel mit Logik 26

6 Graphenproblem: Vertex Cover **31**

7 Hashing **34**
7.1 Hashfunktionen 34
7.2 Kollisionen 35
7.2.1 Verkettete Liste 36
7.2.2 Offene Addressierung 37
7.3 Universelles Hashing 39

Literaturverzeichnis **40**

1 Einführung: Knapsack-Probleme

1.1 Aufgabenstellungen

Beispiel 1.1 (Eine Klausursituation)**.** Informatikstudent Knut Donald sitzt in einer Klausur, welche 150 Minuten dauert. Es gibt maximal 100 Punkte und es werden 50 Punkte benötigt um die Klausur zu bestehen. Nach 30 Minuten hat Knut 6 sichere Punkte und 10 Punkte, die er nicht kann - und einen Überblick über die restlichen Teilaufgaben:

i Aufgabe	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
z_i Zeit	20	32	40	8	16	4	32	40	8	32	28	20	16	20	40	24
p_i Punkte	3	3	10	5	2	4	2	9	2	5	3	9	10	3	10	4

Es gibt also noch 84 Punkte zu holen, jedoch werden dafür 380 Minuten benötigt. Es verbleiben allerdings nur 120 Minuten. Kann Knut die Klausur bestehen? Wir suchen also eine Menge $S \subseteq \{1, ..., 16\}$ mit $\sum_{i \in S} z_i \leq 120$ und $\sum_{i \in S} p_i \geq 44$, d.h. eine Teilmenge der Aufgaben, die in den 120 Minuten gelöst werden können und mindestens 44 Punkte einbringen.

Wir beobachten, dass viele Punkte alleine noch nicht reichen. Auch die Zeit muss berücksichtigt werden. Wir suchen wertvolle Aufgabe, d.h. möglichst viele Punkte pro Zeiteinheit oder möglichst wenig Zeit pro Punkt. Mit scharfen Blick erkennt man: Aufgabe 6 ist sehr wertvoll, denn wir benötigen nur eine Minute pro Punkt; Aufgaben 4 und 13 sind wertvoll, da wir nur 1,6 Minuten pro Punkt benötigen; Aufgabe 7 ist wertlos, denn wir verbrauchen 16 Minuten pro Punkt!

Insgesamt ist die ganze Sache unübersichtlich! Es gibt $2^{16} = 65.536$ mögliche Lösungen. Alle Möglichkeiten zu testen dauert zu lange. Wir brauchen ein schnelles Verfahren.

Wir betrachten die folgende Strategie. Sortiere die Aufgaben nach Wert $\left(\frac{z_i}{p_i}\right)$. Nimm Aufgaben der Reihenfolge entsprechend auf, falls sie noch in der Zeit schaffbar sind. Für unser Beispiel liefert diese Strategie folgendes:

i	6	4	13	12	3	9	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	40	8	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	10	2	10	9	4	5	3	3	2	3	3	2
$\frac{z_i}{p_i}$	1	1,6	1,6	$2,\overline{2}$	4	4	4	$4,\overline{4}$	6	6,4	$6,\overline{6}$	$6,\overline{6}$	8	$9,\overline{3}$	$10,\overline{6}$	16

Dies liefert $S = \{6, 4, 13, 12, 9, 3, 16\}$ mit $\sum_{i \in S} z_i = 120$ und $\sum_{i \in S} p_i = 44$. Knut kann die Klausur also gerade so bestehen.

Betrachten wir die Klausursituation als *Entscheidungsproblem* einmal genauer.

Problem 1.2 (Rucksackproblem, 0-1-Knapsack)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe $z_i > 0$ Gewinn $p_i > 0$
- Größenschranke Z
- Gewinnschranke P

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$\sum_{i \in S} p_i \geq P$$

Es interessiert uns also nur, ob ein gewisser Wert erreicht werden kann. Ein Algorithmus für Problem 1.2 gibt uns also irgendeine Lösung, die die Bedingungen erfüllt. Man ist aber unter Umständen an einer Lösung interessiert, die den höchsten Wert erzielt. Das motiviert folgendes *Optimierungsproblem*:

Problem 1.2′ (Maximum Knapsack)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe $z_i > 0$ und Gewinn $p_i > 0$
- Größenschranke Z

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$\sum_{i \in S} p_i = Maximal$$

Bei Problem 1.2 und 1.2′ müssen die Objekte entweder ganz oder gar nicht genommen werden. Gegebenenfalls können die Objekte aber geteilt werden. Beispielsweise gibt es bei Klausuraufgaben Teilpunkte für teilweise gelöste Aufgaben. Dadurch erhalten wir folgendes Problem:

Problem 1.3 (Fractional Knapsack)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe $z_i > 0$ und Gewinn $p_i > 0$
- Größenschranke Z

Gesucht:

Für jedes Objekt ein Wert

$$x_i \in [0, 1]$$

sodass

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

Dieses Problem lässt sich mit dem Greedy-Ansatz optimal lösen, d.h. für jede Instanz gibt uns der Algorithmus eine Lösung mit einem bestmöglichen Wert zurück. Wie oben beschrieben, nehmen wir die nach $\frac{z_i}{p_i}$ sortierten Objekte der Reihe nach auf und nehmen das erste Objekt, das nicht mehr passt, anteilig mit auf. Algorithmus 1.4 zeigt den Pseudocode für diese Strategie.

Algorithmus 1.4 Greedy-Algorithmus für Fractional Knapsack

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$

Ausgabe: $x_1, ..., x_n \in [0, 1]$
mit

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

1: Sortiere {1,...,n} nach $\frac{z_i}{p_i}$ aufsteigend;
Dies ergibt die Permutation $\pi(1), ..., \pi(n)$.
Setze $j = 1$.
2: **while** $(\sum_{i=1}^{j} z_{\pi(i)} \leq Z)$ **do**
3: $\quad x_{\pi(j)} := 1$
4: $\quad j := j + 1$
5: Setze $x_{\pi(j)} := \dfrac{Z - \sum_{i=1}^{j-1} z_{\pi(i)}}{z_{\pi(j)}}$
6: **return**

Satz 1.5. *Algorithmus 1.4 liefert eine optimale Lösung für Problem 1.3.*

Beweis. O.B.d.A. sei $\sum_{i=1}^{n} z_i \geq Z$. Zunächst zeigen wir, dass $\sum_{i=1}^{n} x_i z_i \leq Z$ ist. Sei j^* der letzte Index, für den die *while*-Bedingung überprüft wird. Nach Konstruktion ist daher $\sum_{i=1}^{j^*-1} x_{\pi(i)} z_{\pi(i)} \leq Z$. Außerdem ist $x_{\pi(j^*)} = \frac{Z - \sum_{i=1}^{j^*-1} x_{\pi(i)} z_{\pi(i)}}{z_{\pi(j^*)}}$, also ist $x_{\pi(j^*)} z_{\pi(j^*)} = Z - \sum_{i=1}^{j^*-1} x_{\pi(i)} z_{\pi(i)}$ und damit $x_{\pi(j^*)} z_{\pi(j^*)} + \sum_{i=1}^{j^*-1} x_{\pi(i)} z_{\pi(i)} = Z$. Da $x_{\pi(j^*+1)} = \cdots = x_{\pi(n)} = 0$, ist somit auch $\sum_{i=1}^{n} x_i z_i = Z$.

Es bleibt noch zu zeigen, dass die berechnete Lösung bestmöglich ist. Wähle j^* wie oben. Es gilt $0 \leq x_{\pi(j^*)} < 1$. Angenommen x wäre nicht optimal; dann betrachte unter allen optimalen Lösungen eine solche Lösung x^*, für die $\sum_{i=1}^{n} |x_i - x_i^*|$ minimal ist. Wegen $\sum_{i=1}^{n} x_i p_i < \sum_{i=1}^{n} x_i^* p_i$ und $p_i > 0$ müsste ein $k \in \{1, \ldots, n\}$ existieren, sodass $x^*_{\pi(k)} > x_{\pi(k)}$; da für alle $k \leq j^* - 1$ sowohl $x_{\pi(k)} = 1$ als auch $x^*_{\pi(k)} \leq 1$ gälte, müsste $j^* \leq k \leq n$ gelten. Wäre nun $x^*_{\pi(j)} \geq x_{\pi(j)}$ für alle $j \leq j^*$, so folgte aus $\sum_{i=1}^{n} x_i z_i = Z$ und $x^*_{\pi(k)} > x_{\pi(k)}$ sowie $z_k > 0$ der Widerspruch $\sum_{i=1}^{n} x_i^* z_i > Z$; also gäbe es ein $\ell \leq j^*$ mit $x^*_{\pi(\ell)} < x_{\pi(\ell)}$.

Sei nun $\varepsilon \leq \min\left\{x_{\pi(\ell)} - x^*_{\pi(\ell)}, \frac{z_{\pi(k)}}{z_{\pi(\ell)}} \cdot \left(x^*_{\pi(k)} - x_{\pi(k)}\right)\right\}$. Dann wäre $x^*_{\pi(\ell)} + \varepsilon \leq x_{\pi(\ell)}$ und $x^*_{\pi(k)} - \varepsilon \frac{z_{\pi(\ell)}}{z_{\pi(k)}} \geq x_{\pi(k)}$. Wir erhielten eine Lösung $\tilde{x} = (x_1^*, \ldots, x^*_{\pi(\ell)} + \varepsilon, \ldots, x^*_{\pi(k)} - \varepsilon \frac{z_{\pi(\ell)}}{z_{\pi(k)}}, \ldots, x_n^*)$ mit Wert $\sum_{i=1}^{n} \tilde{x}_i p_i = \sum_{i=1}^{n} x_i^* p_i + \varepsilon p_{\pi(\ell)} - \varepsilon p_{\pi(k)} \frac{z_{\pi(\ell)}}{z_{\pi(k)}}$. Wegen $\ell \leq k$ wäre $\varepsilon \frac{p_{\pi(\ell)}}{z_{\pi(\ell)}} \geq \varepsilon \frac{p_{\pi(k)}}{z_{\pi(k)}}$, woraus $\varepsilon p_{\pi(\ell)} - \varepsilon p_{\pi(k)} \frac{z_{\pi(\ell)}}{z_{\pi(k)}} \geq 0$ folgte. Somit gälte $\sum_{i=1}^{n} \tilde{x}_i p_i \geq \sum_{i=1}^{n} x_i^* p_i$, d.h. $\tilde{x}$ wäre auch optimal. Wegen

$$\begin{aligned}
\sum_{i=1}^{n} |\tilde{x}_i - x_i| &= \sum_{i \neq \pi(\ell),\, \pi(k)} |x_i^* - x_i| + \left|\tilde{x}_{\pi(\ell)} - x_{\pi(\ell)}\right| + \left|\tilde{x}_{\pi(k)} - x_{\pi(k)}\right| \\
&= \sum_{i \neq \pi(\ell),\, \pi(k)} |x_i^* - x_i| + \left|x^*_{\pi(\ell)} + \varepsilon - x_{\pi(\ell)}\right| + \left|x^*_{\pi(k)} - \varepsilon \frac{z_{\pi(\ell)}}{z_{\pi(k)}} - x_{\pi(k)}\right| \\
&= \sum_{i \neq \pi(\ell),\, \pi(k)} |x_i^* - x_i| + \left|x^*_{\pi(\ell)} - x_{\pi(\ell)}\right| - \varepsilon + \left|x^*_{\pi(k)} - x_{\pi(k)}\right| - \varepsilon \frac{z_{\pi(\ell)}}{z_{\pi(k)}} \\
&= \sum_{i=1}^{n} |x_i^* - x_i| - \varepsilon - \varepsilon \frac{z_{\pi(\ell)}}{z_{\pi(k)}}
\end{aligned}$$

hätten wir damit eine Optimallösung $\tilde{x}$, für die $\sum_{i=1}^{n} |\tilde{x}_i - x_i| < \sum_{i=1}^{n} |x_i^* - x_i|$ gilt. Das ist ein Widerspruch zur Annahme. Also muss x optimal sein. □

Betrachten wir erneut Beispiel 1.1.

i	6	4	13	12	3	9	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	40	8	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	10	2	10	9	4	5	3	3	2	3	3	2
$\frac{z_i}{p_i}$	1	1,6	1,6	$2,\overline{2}$	4	4	4	$4,\overline{4}$	6	6,4	$6,\overline{6}$	$6,\overline{6}$	8	$9,\overline{3}$	$10,\overline{6}$	16

Um den Algorithmus besser zu verfolgen, geben wir Zwischenwerte nach jeder for-Schleife an. Dazu gehört, welches Objekt genutzt wird, zu welchen Teilen dieser gepackt wird und welches Gesamtgewicht und welchen Gesamtwert unsere aktuelle Packung besitzt.

j	$\pi(j)$	$x_{\pi(j)}$	$\sum_{i=1}^{j} x_{\pi(i)} z_{\pi(i)}$	$Z - \sum_{i=1}^{j} x_{\pi(i)} z_{\pi(i)}$	$\sum_{i=1}^{j} x_{\pi(i)} p_{\pi(i)}$
1	6	1	4	116	4
2	4	1	12	108	9
3	13	1	28	92	19
4	12	1	48	72	28
5	3	1	88	32	38
6	9	1	96	24	40
7	15	$\frac{6}{10}$	120	0	46

Mit dem Algorithmus erhalten wir also eine Lösung mit Wert

$$\sum_{i=1}^{n} p_i x_i = 46$$

mit

$$x_6 = x_4 = x_{13} = x_{12} = x_9 = x_3 = 1$$
$$x_{15} = 0{,}6$$
$$x_8 = x_{16} = x_{10} = x_1 = x_{14} = x_5 = x_{11} = x_2 = x_7 = 0$$

Problematisch ist, dass fraktionale Lösungen nicht immer erlaubt sind. Zum Beispiel sollte ein Kunsträuber seinen Rucksack nicht mit kleingeschnittenen Gemälden vollpacken. Dadurch verliert er Wert.

Schauen wir uns weitere Problemvarianten an. Eine Variante betrachtet das mehrfache Nutzen von Objekten. Beispielsweise kann in einem Supermarkt ein Objekt mehrfach eingekauft werden.

Problem 1.6 (Integer Knapsack)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe z_i Gewinn p_i
- Größenschranke Z

Gesucht:

Für jedes Objekt ein Wert

$$x_i \in \mathbb{N}$$

mit

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

Ein Spezialfall des Knapsack-Problems ist, wenn alle Gewinndichten identisch sind. Es spielt also weniger eine Rolle, welche Objekte ich benutze, sondern nur, dass sie in den Rucksack passen. Dabei ist man daran interessiert, den Rucksack so voll wie möglich zu packen.

Problem 1.7.
Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe z_i
- Größenschranke Z

Gesucht:
Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$\sum_{i \in S} z_i = Maximal$$

Ein Spezialfall von Problem 1.7 fragt nun, ob diese Schranke Z exakt erreicht werden kann. Dieses Problem ist auch unter dem Namen Subset Sum bekannt, siehe Problem 1.8. Auch dieses Problem kann weiter verfeinert werden, indem man fragt, ob alle Objekt so aufgeteilt werden können, sodass beide Teile das gleiche Gewicht besitzt. Dieses Problem wird auch Partition genannt. Siehe Problem 1.9.

Problem 1.8 (Subset Sum)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe z_i
- Zielgröße Z

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i = Z$$

Problem 1.9 (Partition)**.**

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe z_i

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i = \sum_{i \notin S} z_i$$

1.2 Ein Beispiel

Beispiel 1.10.
Ist die Instanz $\{z_1, ..., z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$ für Partition lösbar? Das Gesamtgewicht dieser Objekte ist 240. Wir suche also eine Teilmenge, dessen Gewicht 120 ergibt. Das ist allerdings nicht möglich. Doch wie können wir zeigen, dass es keine Lösung gibt?

Wir testen alle Möglichkeiten durch. O.B.d.A. können wir annehmen, dass die 57 in S enthalten ist. Wir haben also 57,___ und __. Bleibt noch ein Gewicht von 63 übrig.

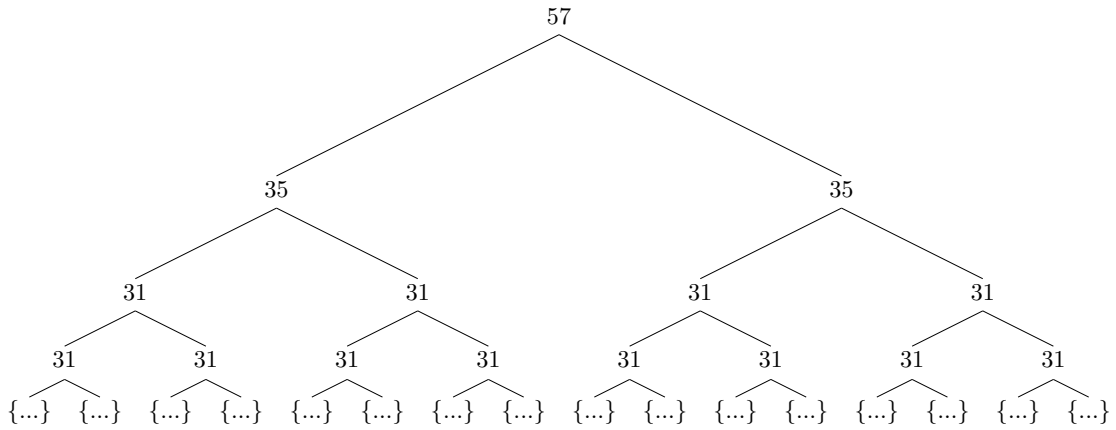
Betrachte nun die 35. Wenn wir die 35 zu S hinzufügen, müssen wir noch ein Gewicht von 28 mit den übrigen Objekten erreichen. Durch schnelles Ausprobieren sieht man schnell, dass das weder mit der 20, der 17, der 13 oder der 7 funktioniert. Die 35 darf also nicht in S sein. Damit haben wir 57,___ und 35,___

Betrachte nun die 31. Fügen wir die 31 zu S hinzu, müssen wir noch ein Gewicht von 32 erzeugen. Durch Ausprobieren sieht man wieder schnell, dass kann nicht mit den übrigens Elementen erzeugt werden. Damit können beide Objekte mit Gewicht 31 nicht

in S liegen. Es gilt nun also 57, ___ und 35, 31, 31, ___

Betrachte nun die 29. Ist die 29 in S enthalten, bleibt ein Gewicht von 34 über. Auch dieses kann nicht mit der 17, 13 und der 7 erzeugt werden. Die 29 kann also nicht in S liegen. Damit ist haben wir 57, ___ und 35, 31, 31, 29, ___

Auf der rechten Seite befinden sich nun eine Summe von > 120. Wir haben also durch systematisches Ausprobieren herausgefunden, dass eine Partition nicht möglich ist. Das Grundprinzip, das wir benutzt haben, ist eine *Rückwärtsrekursion*, d.h. wir nehmen eine Instanz und teilen sie in kleinere Instanzen auf. Der Vorteil ist: es funktioniert. Allerdings müssen exponentiell viele Teilmengen durchprobiert werden. Durch das Ausprobieren können wir einen Entscheidungsbaum konstruieren: Soll eine Zahl in die rechte Teilmenge, so wird eine Kante nach rechts-unten gezeichnet. Soll sie in die linke, so wird eine Kante nach links-unten gezeichnet.



Wie können wir das verbessern?

- Schneide Teilbäume ab wenn möglich. Das erlaubt zwar Einsparungen, bleibt aber im schlimmsten Fall exponentiell.
- Betrachte nicht Rückwärtsrekursion (d.h. Stufenweise ausprobieren, wobei größere Teilinstanzen auf kleinere Teilinstanzen reduziert werden), sondern Vorwärtsrekursion. Baue größere Lösungen aus kleineren auf!

Beispiel 1.10A.
Schauen wir uns folgende Situation an:
Fragen:

- Wie lautet eine mögliche Kombination?
- Wie gut oder schlecht ist der Greedy-Algorithmus 1.4 für 0-1-Knapsack?
- Wie kann man den Algorithmus erweitern/modifizieren, damit er eine bessere Leistung garantiert?

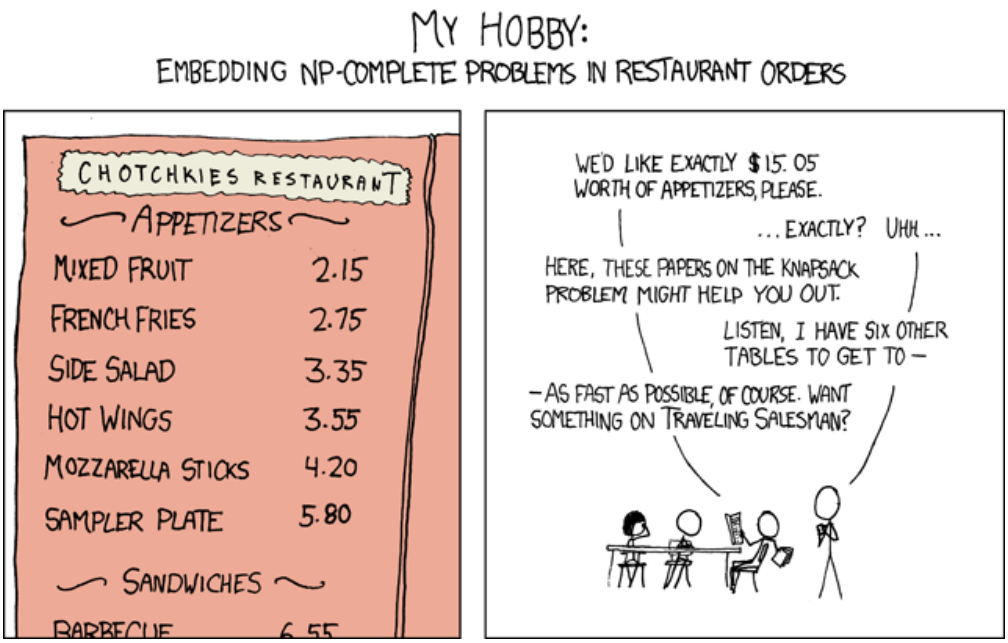


Abbildung 1.1: xkcd #287 [4]

Große Fragen:

- Wie bekommt man diese Probleme besser in den Griff?
- Wie schwierig kann das werden - bzw. wie lange benötigt das Lösen?

2 Dynamisches Programmieren

In diesem Kapitel betrachten wir das Verfahren *Dynamic Programming* (deutsch: dynamische Programmierung), um Probleme aus dem vorherigen Kapitel zu lösen. Wir starten mit dem Entscheidungsproblem SUBSET SUM und verallgemeinern den benutzten Algorithmus danach, um das Optimierungsproblem MAXIMUM KNAPSACK zu lösen.

2.1 Subset Sum

Betrachte die folgenden Zahlen in aufsteigender Reihenfolge:

$$[z_1, ..., z_9] = [7, 13, 17, 20, 29, 31, 31, 35, 57]$$

Teste für Zahl x, ob sie mit den ersten i Summanden erzeugt werden kann.

$$\mathscr{S}(x,i) = \begin{cases} 1, & \text{falls x mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$$

$$\mathscr{S}(x,0) = \begin{cases} 1, & \text{für } x = 0 \\ 0, & \textit{sonst} \end{cases}$$

$$\mathscr{S}(x,1) = \begin{cases} 1, & \text{für } x = 0 \text{ oder } x = 7 \\ 0, & \textit{sonst} \end{cases}$$

$$\mathscr{S}(x,2) = \begin{cases} 1, & \text{für } x = 0, 7, 13, 20 \\ 0, & \textit{sonst} \end{cases}$$

$$\mathscr{S}(x,3) = \begin{cases} 1, & \text{für } x = 0, 7, 13, 20, 17, 24, 30, 37 \\ 0, & \textit{sonst} \end{cases}$$

Die Werte von $\mathscr{S}(x,i)$ können wir in einer Tabelle festhalten:

i \ x	0	1	2	3	3	5	6	7	...	13	14	15	16	17	18	19	20	...	240
0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	...	0
1	1	0	0	0	0	0	0	1	...	0	0	0	0	0	0	0	0	...	0
2	1	0	0	0	0	0	0	1	...	1	0	0	0	0	0	0	1	...	0
3	1	0	0	0	0	0	0	1	...	1	0	0	0	1	0	0	1	...	0

Das lässt sich verallgemeinern. Wir erhalten folgende Update-Regeln:

$$\mathscr{S}(x,0) = 0, \text{ für alle } x \in \{1, ..., Z\}; \mathscr{S}(0,0) = 1$$
$$\mathscr{S}(x, i-1) = 1 \Rightarrow \mathscr{S}(x,i) = 1$$
$$\mathscr{S}(x - z_i, i-1) = 1 \Rightarrow \mathscr{S}(x,i) = 1$$

Algorithmus 2.1 Dynamic Programming für Subset Sum

Eingabe: Zahlen $z_1, ..., z_n$ mit $\sum_{i=1}^{n} z_i = Z$

Ausgabe: Boolesche Funktion $\mathscr{S} : \{0, ..., Z\} \times \{0, ..., n\} \to \{0, 1\}$

mit $\mathscr{S}(x, i) = \begin{cases} 1, & \text{falls x mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$

1: $\mathscr{S}(0, 0) := 1$
2: **for** (x=1) to Z **do**
3: $\quad \mathscr{S}(x, 0) := 0;$
4: **for** $(i = 1)$ to n **do**
5: $\quad$ **for** $(x = 0)$ to $z_i - 1$ **do**
6: $\quad\quad \mathscr{S}(x, i) := \mathscr{S}(x, i - 1);$
7: $\quad$ **for** $(x = z_i)$ to Z **do**
8: $\quad\quad$ **if** $((\mathscr{S}(x, i - 1) = 1)$ OR $(\mathscr{S}((x - z_i), i - 1) = 1))$ **then**
9: $\quad\quad\quad \mathscr{S}(x, i) := 1;$
10: $\quad\quad$ **else**
11: $\quad\quad\quad \mathscr{S}(x, i) := 0;$
12: **return**

Satz 2.2. *Algorithmus 2.1 löst das Problem* Subset Sum. *Die Lauzeit ist* $O(Z \cdot n)$.

Beweis. Wir beweisen den Satz mit vollständiger Induktion über die Anzahl der Elemente n. Für $n = 0$ ist die Initialisierung korrekt. Für den Induktionsschritt nehmen wir an, dass $\mathscr{S}(x, i - 1)$ für alle Zahlen x korrekt ist. Dann betrachten wir im Schritt 2 (Zeilen 7-9) die Möglichkeiten der Erzeugung von x mit $z_1, ..., z_i$.

(I) z_i wird nicht verwendet $\Rightarrow x$ kann mit $z_1, ..., z_{i-1}$ erzeugt werden.

(II) z_i wird verwendet $\Rightarrow x - z_i$ kann mit $z_1, ..., z_{i-1}$ erzeugt werden.

Genau das macht der Algorithmus in Schritt 2. Daher gilt auch, dass $\mathscr{S}(x, i)$ korrekt ist. Die Laufzeit ist klar. □

2.2 Dynamic Programming für Rucksackprobleme

Bei Rucksackproblemen haben wir nicht nur Objekte i einer gewissen Größe z_i, sondern noch mehr:

- Objekte $1, ..., n$
- Kosten $z_1, ..., z_n$
- Nutzen $p_1, ..., p_n$

Auch darauf lässt sich Dynamic Programming anwenden! Betrachte dazu nicht nur $\mathscr{S}(x,i)$ sondern auch $P(x,i)$. Das ist der höchste Nutzen, der sich mit den Objekten $1,\ldots,i$ erzielen lässt, wenn die Kosten auf x begrenzt sind. Nehmen wir an, $P(x,i-1)$ ist für alle $x \in \{0,...,Z\}$ bekannt. Betrachte nun Objekt i mit Kosten z_i und Nutzen p_i. Dann gibt es folgende Möglichkeiten:

1. Objekt i ist zu teuer für Kostenschranke x, falls $z_i > x$
2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung
3. Objekt i ist nicht zu teuer und bringt eine Verbesserung

Wir erhalten also folgende Gleichung:

$$P(x,i) = \begin{cases} P(x,i-1), & \text{falls } z_i > x \\ max\{P(x,i-1), P(x-z_i,i-1)+p_i\}, & \text{sonst} \end{cases}$$

Das ist eine Rekursionsgleichung, die sogenannte „Bellman-Rekursion“ nach Richard Bellman (1957) [1]. Man löst hierbei die kleineren Teilprobleme so gut wie möglich - und zwar per Vorwärtsrekursion.

Beispiel 2.4 (Knapsackproblem)**.**
Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Benutzen wir die Rekursionsgleichung, erhalten wir folgende Tabelle für $P(x,i)$:

i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Beachte:

- Dominierte Teilmengen werden ignoriert.
- Nur Verbesserungen werden berücksichtigt!

Algorithmus 2.5 Dynamic Programming für Maximum Knapsack

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

mit $P(x,i) = max \sum_{j=1}^{i} p_j y_j$ mit $\sum_{j=1}^{i} z_j y_j \leq x,$ für $y_j \in \{0, 1\}$

```
1:  for (x = 0) to Z do
2:      P(x, 0) := 0
3:  for (i = 1) to n do
4:      for (x = 0) to (z_i − 1) do
5:          P(x, i) := P(x, i − 1)
6:      for (x = z_i) to Z do
7:          if ((P(x − z_i, i − 1) + p_i) > P(x, i − 1)) then
8:              P(x, i) := P(x − z_i, i − 1) + p_i
9:          else
10:             P(x, i) := P(x, i − 1)
```

Satz 2.6. *Algorithmus 2.5 berechnet besten Lösungswert für das Rucksackproblem in einer Laufzeit O(nZ).*

Das sieht polynomiell aus, ist aber tatsächlich nicht unbedingt polynomiell in der Größe des sinnvoll codierten Inputs: Z kann groß sein und mit $O(\log Z)$ Bits codiert werden. Das nennt man *pseudopolynomiell.*

Ein weiterer Nachteil: Wir berechnen eine ganze Tabelle, was unter Umständen viel Speicherplatz verbraucht. Eventuell interessiert einen aber nur der Optimalwert, wodurch man nur die vorhergehende Zeile benötigt. Wir speichern also je nur eine Zeile und überschreibe sie geeignet (und vorsichtig). Hier kann man auch Codezeilen sparen! Wir erhalten:

Algorithmus 2.7 Dynamic Programming für Maximum Knapsack - sparsamer

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \to \mathbb{R};\ x \mapsto P(x)$

mit $P(x) = \max \sum_{j=1}^{n} p_j y_j$ mit $\sum_{j=1}^{n} z_j y_j \leq x,$ für $y_j \in \{0, 1\}$

```
1:  for (x = 0) to Z do
2:      P(x) := 0
3:  for (i = 1) to n do
4:      for (x = Z) downto z_i do
5:          if ((P(x − z_i) + p_i) > P(x)) then
6:              P(x) := P(x − z_i) + p_i
7:  return p* := P(Z)
```

Satz 2.8. *Algorithmus 2.7 berechnet einen besten Lösungswert für das Rucksackproblem in einer Laufzeit O(nZ).*

Bisher haben wir nur den Wert einer optimalen Lösung bestimmt. Doch was sind Möglichkeiten auch eine solche Lösung zu erhalten? Ähnlich zum bisher gesehenen können wir die Teillösungen in den einzelnen Zellen speichern und mitschleppen. Allerdings kostet das viel Speicher und unter Umständen viel Zeit. Eine einfachere Möglichkeit ist das Merken der Position von welcher Zelle man gekommen ist. Dafür braucht man sich nur merken, welcher Fall der Rekursionsgleichung eingetroffen ist. Für Maximum Knapsack reicht hier eine Tabelle mit Booleschen Werten: *true*, falls ein Objekt genommen wurde, *false*, andernfalls.

Andere Variante:
Dynamic Programming nicht mit Fokus auf Kosten z_i, sondern auf den Nutzen p_i.
Das ist ähnlich, aber nicht

Maximieren des Nutzens für mögliche Kosten

sondern

Minimieren der Kosten für möglichen Nutzen.

3 Branch and Bound

Im Subset-Sum-Beispiel 1.10 haben wir gesehen, dass sich auch beim Enumerieren Arbeit sparen lässt. Dabei haben wir Teilbäume abgeschnitten, bei denen wir wussten, dass wir dort nichts mehr erreichen können. Das betrachten wir in diesem Kapitel. Für ein Maximierungsproblem lassen sich folgende Grundideen festhalten.

1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
   - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum
   - Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Baum
3. Beide Schranken sollten möglichst einfach und schnell zu berechnen sein
4. Wenn der erreichbare Wert in einem Teilbaum kleiner bleiben muss als der im ganzen Baum bereits erreichte, können wir den aktuellen Teilbaum abschneiden

Betrachtet man ein Minimierungsproblem, so kehrt sich der Sinn der Schranken um: Untere Schranken geben den erreichbaren Zielfunktionswert an; Obere Schranken einen erreichten Zielfunktionswert.

Wir betrachten noch einmal Beispiel 2.4 mit $Z = 9$ und folgenden sieben Objekten:

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

1. Wie können wir das enumerieren?
   - Wir probieren nacheinander für $i = 1, ..., 7$, ob $x_i = 0$ oder $x_i = 1$.
   - Wir gehen die Möglichkeiten baumartig durch (siehe Abbildung 3.1).
   - Wir laufen DFS-mäßig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.
2. Welche unteren und oberen Schranken haben wir zur Verfügung? Für die *untere Schranke* eignet sich Greedy. Verwende unter den noch nicht zugewiesenen Objekten in aufsteigendem Kosten/Nutzen-Verhältnis Objekte, solange sie passen. Hier ohne fixierte Objekte:

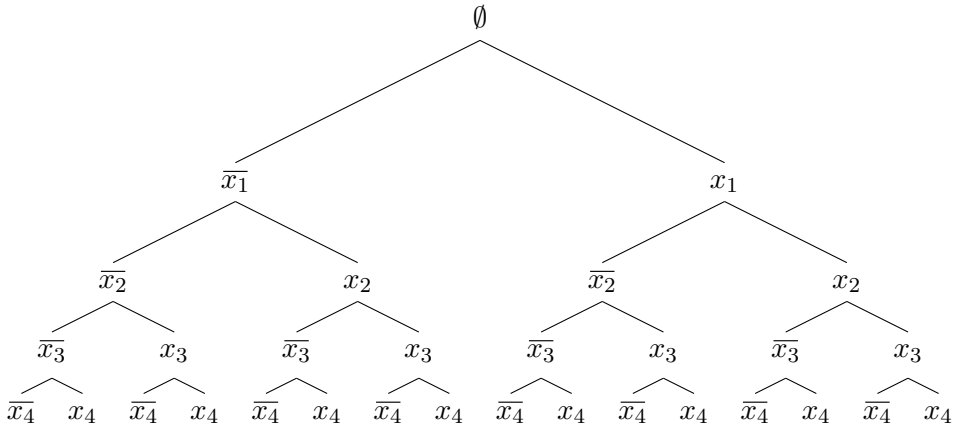


Abbildung 3.1: Enumerationsbaum für vier Objekte.

j	$\pi(j)$	$x_{\pi(j)}$	$z_{\pi(j)}$	$\sum_{i=1}^{j} x_{\pi(i)} z_{\pi(i)}$	$\sum_{i=1}^{j} x_{\pi(i)} p_{\pi(i)}$
1	1	1	2	2	6
2	2	1	3	5	11
3	3	0	6	5	11
4	4	0	7	5	11
5	5	0	5	5	11
6	6	0	9	5	11
7	7	1	3	9	14

Es ist also ein Gesamtnutzen von 14 erreichbar.

Für die *obere Schranke* können wir Greedy für das einfachere Problem Fractional Knapsack benutzen, d.h. Algorithmus 1.4.

i=1 : $\sum_{i \in S} z_i = 2$, $\sum_{i \in S} p_i = 6$

i=2 : $\sum_{i \in S} z_i = 5$, $\sum_{i \in S} p_i = 11$

Bleibt: Z-$\sum_{i \in S} z_i = 4$, also $x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$
Damit: $\sum_{i \in S} x_i z_i = 9$, $\sum_{i \in S} x_i p_i = 11 + \frac{2}{3} \cdot 8 = 16, \bar{3}$
Da unsere Eingabewerte ganzzahlig sind, muss auch die Optimallösung ganzzahlig sein. Wir können unsere Lösung also abrunden und erhalten so eine obere Schranke von 16.

3. Diese Schranken sind einfach zu berechnen.
   WICHTIG: Für die obere Schranke betrachten wir ein Hilfsproblem, was einfach ist, weil wir weniger strenge Bedingungen verlangen. So etwas nennt man *Relaxierung*

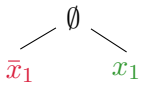
4. Nun betrachten wir den Enumerationsbaum mit jeweils:

S: positiv fixiert ($x_i = 1$)
$\overline{S}$: negativ fixiert ($x_i = 0$)
Wir nennen die lokale obere Schranke U, und die globale untere Schranke P.

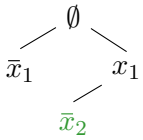
Leerer Baum $S = \emptyset$, $\overline{S} = \emptyset$, U=16, P=14

Objekt 1 $\overline{S} = \{1\}$, $S = \emptyset$

Da x_1 ausgeschlossen wurde, kann eine neue obere Schranke berechnet werden. Greedy für Fractional Knapsack liefert: U=13. Also ist hier nicht mehr als Gesamtwert 13 erreichbar. $x_1 = 0$ ist also eine Sackgasse: Wir brauchen diesen Teilbaum nicht weiter zu durchlaufen.
Also: $\overline{S} = \emptyset$, $S = \{1\} \Rightarrow U = 16$, $P = 14$

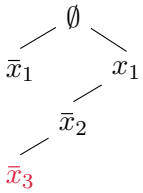


Objekt 2 (raus) $\overline{S} = \{2\}$, $S = \{1\} \Rightarrow U = 15$, $P = 14$



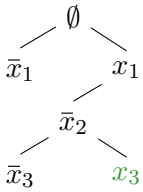
Objekt 3 (raus) $\overline{S} = \{2,3\}$, $S = \{1\} \Rightarrow U = 15$, $P = 15$

Hier gibt es wegen $U \leq P$ keinen Platz mehr für Verbesserungen $\Rightarrow$ Sackgasse

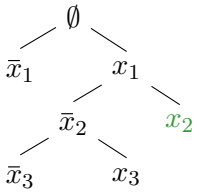


Objekt 3 (rein) $\overline{S} = \{2\}$, $S = \{1,3\} \Rightarrow U = 15$, $P = 15$

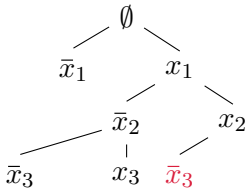
Auch hier können wir uns nicht mehr verbessern. $\Rightarrow$ Sackgasse



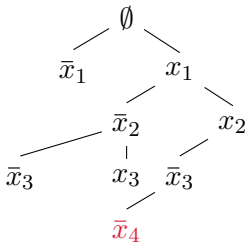
Objekt 2 (rein) $\overline{S} = \emptyset$, $S = \{1,2\} \Rightarrow U = 16$, $P = 15$



Objekt 3 (raus) $\overline{S} = \{3\}$, $S = \{1, 2\} \Rightarrow U = 16$, $P = 15$

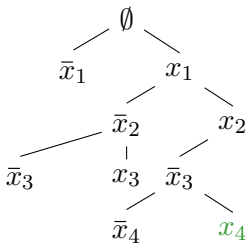


Objekt 4 (raus) $\overline{S} = \{3, 4\}$, $S = \{1, 2\} \Rightarrow U = 15$, $P = 15 \Rightarrow$ Sackgasse

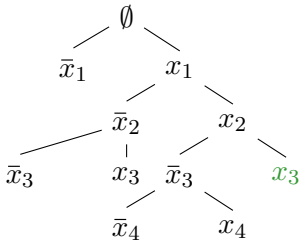


Objekt 4 (rein) $\overline{S} = \{3\}$, $S = \{1, 2, 4\}$

Diese Auswahl überschreitet Z und ist daher unzulässig, sowie alle weiteren Entscheidungen in diesem Teilbaum. ⇒ Sackgasse



Objekt 3 (rein) $\overline{S} = \emptyset$, $S = \{1, 2, 3\} \Rightarrow$ Ebenfalls unzulässig ⇒ Sackgasse



Damit haben wir den gesamten Entscheidungsbaum durchlaufen und dabei das Optimum $S = \{1, 4\}$ mit Wert 15 identifiziert.

Allgemein lässt sich das Branch-and-Bound-Verfahren wie in Algorithmus 3.1 beschreiben.

Algorithmus 3.1 Branch-And-Bound als Unterroutine

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$ (global: Kosten/Nutzenwerte, Kostenschranke)
P (global: bester bekannter Lösungswert)
ℓ (nächster Index, über den verzweigt werden soll)
$x_j = b_j$ für $j = 1, ..., \ell - 1$ mit $b_j \in \{0, 1\}$ (bislang fixierte Binärvariable)

Ausgabe: $\max \left\{ \sum_{j=1}^{\ell-1} b_j p_j + \sum_{j=\ell}^{n} x_j p_j \mid \sum_{j=1}^{\ell-1} b_j z_j + \sum_{j=\ell}^{n} x_j z_j \leq Z, x_j \in \{0, 1\} \right\}$

Also: Lösung des Knapsackproblems mit den ersten $\ell - 1$ Variablen fixiert

1: **procedure** BRANCH-AND-BOUND(ℓ)
2: **if** $(\sum_{j=1}^{\ell-1} b_j z_j > Z)$ **then return** ▷ unzulässig
3: Compute $L := LB(b_1, \ldots, b_{\ell-1})$
4: **if** $L > P$ **then** $P := L$ ▷ Lösungswert verbessert
5: **if** $(\ell > n)$ **then return** ▷ Blatt im Baum erreicht
6: $U := UB(b_1, \ldots, b_{\ell-1})$ ▷ (Obere Schranke berechnen)
7: **if** $(U > P)$ **then**
8: $b_\ell := 0$; BRANCH-AND-BOUND($\ell + 1$);
9: $b_\ell := 1$; BRANCH-AND-BOUND($\ell + 1$);
10: **return**

Dabei ist $UB(b_1, ..., b_{\ell-1})$ eine geeignete obere Schranke bei fixierten Variablen $b_1, ..., b_{\ell-1}$:

$$max \left\{ \sum_{j=1}^{\ell-1} b_j p_j + \sum_{j=\ell}^{n} x_j p_j \mid \sum_{j=1}^{\ell-1} b_j z_j + \sum_{j=\ell}^{n} x_j z_j \leq Z, x_j \in \{0, 1\} \right\}$$

Berechnung erfolgt mit dem Greedy-Algorithmus 1.4! $LB(b_1, \ldots, b_{\ell-1})$ ist eine geeignete untere Schranke, welche man beispielsweise mit einem Greedy-Algorithmus für ganzzahliges Knapsack berechnen kann. Diesen Algorithmus sehen wir gleich. Alternativ kann man für eine untere Schranke schauen, ob die aktuelle Belegung eine Verbesserung gebracht hat und aktualisiert P entsprechend.

Satz 3.2. *Algorithmus 3.1 (als rekursiv arbeitende Unterroutine) berechnet eine optimale Lösung für das Knapsackproblem in einer Worst-Case-Laufzeit $O(2^n f(n))$, wobei $f(n)$ die Zeit für die Berechnung der Schranken ist.*

Beweis. Es werden systematisch alle Teillösungen durchprobiert. (Dabei werden Teilmengen nur dann ausgelassen, wenn sie erwiesenermaßen unzulässig sind (Zeile 2) oder keine Verbesserung bringen (Zeile 4)
Die Zahl der Rekursionsaufrufe (Schritt 9 und 11) ist insgesamt 2^n, die sonstigen Berechnungen benötigen jeweils $f(n)$. □

Die Branch-and-Bound-Methode ist natürlich nicht nur auf Knapsack-Probleme limitiert. Land und Doig [3] haben gezeigt, wie das Verfahren im Allgemeinen genutzt werden kann. Gerade für Integer Programming bieten sich Branch-and-Bound-basierte Methoden an.

3.1 Linear Programming

Ein lineares Programm (kurz: LP) für ein Maximierungsproblem besteht aus einer Zielfunktion $\max c^T x$ mit Kostenvektor $c \in \mathbb{R}^n$ und Variablenvektor $x \in \mathbb{R}^n$, sowie aus Nebenbedingungen $Ax \leq b$ mit $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$. Somit ergibt sich die Form:

$$
\begin{array}{rccccccc}
\max & c_1 x_1 & + & \dots & + & c_n x_n & & \\
\text{subject to:} & & & & & & & \\
& A_{1,1} x_1 & + & \dots & + & A_{1,n} x_n & \leq & b_1 \\
& A_{2,1} x_1 & + & \dots & + & A_{2,n} x_n & \leq & b_2 \\
& \vdots & & \vdots & & \vdots & & \vdots \\
& A_{m,1} x_1 & + & \dots & + & A_{m,n} x_n & \leq & b_m \\
& x_1 & , & \dots & , & x_n & \in & \mathbb{R}
\end{array}
$$

Der Kostenvektor kann in einem LP statt zu maximiert auch minimiert werden, falls wir ein Minimierungsproblem formulieren möchten. LPs lassen sich effizient lösen, d.h. es existiert ein Algorithmus, der das Problem in polynomieller Zeit bzgl. der Inputgröße löst.

3.2 Integer Programming

Wenn die Variablen auf ganzzahlige Werte beschränkt sind, spricht man von einem Ganzzahligen Programm (engl: Integer Program, kurz: IP). Das ist meist notwendig, wenn man mit diskreten Problemen wie Knapsack zu tun hat. Liegt noch allgemeiner ein Mix aus ganzzahligen und fraktionalen Variablen vor, spricht man von einem Mixed Integer Program (MIP). Im Gegensatz zum Lösen von LPs ist das Lösen von IPs und MIPs NP-schwer, es gibt also wahrscheinlich keinen polynomiellen Algorithmus hierfür. (Details zu den Komplexitätsklassen P und NP gibt es in Kapitel 5.) Dennoch gibt es da draußen sehr fortschrittliche MIP-Solver, die auch komplexe Instanzen schnell optimal lösen können. Zusammen mit der Tatsache, dass viele algorithmische Probleme unkompliziert als MIP formuliert werden können, macht dies MIPs zu einem in der Praxis äußerst relevanten Tool zum Lösen algorithmischer Probleme.

Ein IP, welches Maximum Knapsack beschreibt, sieht wie folgt aus.

$$
\begin{array}{rccccccc}
\max & p_1 x_1 & + & \dots & + & p_n x_n & & \\
\text{subject to:} & & & & & & & \\
& z_1 x_1 & + & \dots & + & z_n x_n & \leq & Z \\
& x_1 & , & \dots & , & x_n & \in & \{0,1\}
\end{array}
$$

Um dieses IP zu lösen, betrachtet man zunächst die Relaxierung, d.h. wir wählen $x \in [0,1]^n$ und erhalten somit ein LP, welches effizient gelöst werden kann. Nachdem wir eine fraktionale Lösung erhalten haben, testen wir, ob diese Lösung ganzzahlig ist. Falls sie es ist, sind wir fertig. Andernfalls wählen wir eine Variable x_i und lösen das LP einmal für $x_i = 0$ und für $x_i = 1$. Dort kann der Branch-Vorgang erneut stattfinden.

Diese grundlegende Technik der Relaxierung und einem darauf basiertem Branching bildet üblicherweise auch die Grundlage des Lösungsprozesses in MIP-Solvern.

4 Approximationen

In Kapitel 1 haben wir gesehen, dass der Greedy Algorithmus für FRACTIONAL KNAPSACK (Algorithmus 1.4) das Problem optimal löst. Doch wie gut ist Greedy in der ganzzahligen Variante (siehe Algorithmus 4.2)? Wie das nachfolgende Beispiel zeigt, kann Greedy beliebig schlecht sein.

Beispiel 4.1.
Sei $n = 2$, sowie $z_1 = 1$, $p_1 = 1 + \varepsilon$, $z_2 = N$, $p_2 = N$ und $Z = N$. GREEDY$_0$ liefert $S = \{1\}$, was einem Lösungswert von $1 + \varepsilon$ entspricht. Optimal ist allerdings $S = \{2\}$ mit einem Lösungswert von N. Für $N \gg 1 + \varepsilon$ ist der Greedywert einen beliebigen großen Faktor vom Optimum entfernt, denn

$$\frac{\text{ALG}}{\text{OPT}} = \frac{1+\varepsilon}{N} \overset{N\to\infty}{\to} 0$$

Algorithmus 4.2 GREEDY$_0$

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$
Ausgabe: $S \subseteq \{1, ..., n\}$ und G_0
mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$G_0 := \sum_{i \in S} p_i = Inklusionsmaximal$$

1: Sortiere {1,...,n} nach $\frac{z_i}{p_i}$ aufsteigend;
Dies ergibt die Permutation $\pi(1), ..., \pi(n)$.
Setze $j = 1$.
2: **while** $(j \leq n)$ **do**
3: **if** $\left(\sum_{i=1}^{j-1} z_{\pi(i)} x_{\pi(i)} + z_{\pi(j)} \leq Z\right)$ **then**
4: $x_{\pi(j)} := 1$
5: $j := j + 1$
6: **return**

Kann der Algorithmus verbessert werden, sodass wir auch Garantien bekommen, wie gut unser Algorithmus ist? Das Problem von GREEDY$_0$ ist, das es möglicherweise viele Objekte gibt, die einen hohen Kosten-Nutzen-Quotienten besitzen, aber den Rucksack insgesamt nur sehr wenig auslasten. Eine Strategie ist daher zusätzlich zur Greedy-Lösung das Objekt mit dem höchsten Nutzen zu betrachten (siehe Algorithmus 4.3).

Algorithmus 4.3 Greedy-Approximation

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$
Ausgabe: $S \subseteq \{1, ..., n\}$ mit $\sum_{i \in S} z_i \leq Z$ und Wert $G'_0 := \sum_{i \in S} p_i$
1: $G'_0 := max\{G_0, max\{p_i \mid z_i \leq Z, i \in \{1, ..., n\}\}\}$
2: **return**

Satz 4.4. *Algorithmus 4.3 berechnet eine Lösung mit G'_0, im Vergleich mit dem Optimalwert OPT gilt: $G'_0 \geq \frac{1}{2}OPT$.*

Beweis. Nach Konstruktion ist $G'_0 \geq G_0$ und $G'_0 \geq max\{p_i\} =: p^*$. Außerdem ist $G_0 + p^* \geq FractionalKP \geq OPT$.
Daher ist $2G'_0 \geq G_0 + p^* \geq OPT \Rightarrow G'_0 \geq \frac{1}{2}OPT$ □

Definition 4.5 (Approximationsalgorithmus)**.**

1. Für ein Maximierungsproblem MAX ist ein Algorithmus ALG ein *c-Approximationsalgorithmus* für MAX, wenn für jede Instanz I von MAX
   a) ALG in *polynomieller Zeit* in der Größe von I eine zulässige Lösung mit Wert ALG liefert
   b) für den Vergleich mit dem zugehörigen Optimalwert $\text{OPT}(I)$ gilt: $\text{ALG}(I) \geq c \cdot \text{OPT}(I)$ (dabei ist $c \leq 1$)
2. Entsprechend für Minimierungsproblem wiederum:
   a) ALG liefert in *polynomieller Zeit* in der Größe von I eine zulässige Lösung mit Wert $\text{ALG}(I)$
   b) es gilt für den Vergleich mit dem zugehörigen Optimalwert $\text{OPT}(I)$: $\text{ALG}(I) \leq c \cdot \text{OPT}(I)$ (dabei ist $c \geq 1$)

Frage: Wie gut kann man Knapsack approximieren?

Algorithmus 4.6 GREEDY_k

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$ [Parameter k fixiert]
Ausgabe: $S \subseteq \{1, ..., n\}$ mit $\sum_{i \in S} z_i \leq Z$ und Wert $G_k := \sum_{i \in S} p_i$
1: $G_k := 0$, $S := \emptyset$
2: **for all** $\overline{S} \subseteq \{1, ..., n\}$ mit $|\overline{S}| \leq k$ **do**
3: **if** $(\sum_{i \in \overline{S}} z_i \leq Z)$ **then**
4: $G_k := max\{G_k, \sum_{i \in \overline{S}} p_i + \text{GREEDY}_0(\{z_i | i \notin \overline{S}\}, Z - \sum_{i \in \overline{S}} z_i, \{p_i | i \notin \overline{S}\})\}$;
5: Update S;
6: **return** G_k, S

Satz 4.7. GREEDY_k *ist ein* $(1 - \frac{1}{k+1})$*-Approximationsalgorithmus.*

Beweis (nach [5]). Man sieht leicht, dass die Laufzeit höchstens $O(n^{k+2})$ ist. Für festes k ist das polynomiell. Sei OPT ein optimaler Lösungswert, erzielt mit einer Teilmenge $S^* \subseteq \{1, ..., n\}$. Wir unterscheiden:

1. $|S^*| \leq k$. Dann testet $\textsc{Greedy}_k$ S^* als ein $\overline{S}$ und findet also das Optimum.

2. $|S^*| > k$. Seien $i_1, ..., i_k$ die ersten k Objekte in S^* mit den höchsten Gewinnen $p_{i_1}, \ldots, p_{i_k}$. Diese Teilmenge wird von $\textsc{Greedy}_k$ als $\overline{S}$ getestet und per Greedy erweitert. Seien $i_{k+1}, ..., i_{k+l}$ die bis hierhin hinzugenommenen Objekte und i_m das erste Objekt aus S^*, das nicht mehr passt. (Sollte Greedy alle Elemente aus S^* packen, ist dies das Optimum.) Wir können argumentieren, dass

   a) $\sum_{j=1}^{k} p_{i_j} + \sum_{j=1}^{l} p_{i_{k+j}} + p_{i_m} \geq OPT$

   b) $\sum_{j=1}^{k} p_{i_j} + \sum_{j=1}^{l} p_{i_{k+j}} \leq G_k$

   Außerdem ist $p_{i_m} \leq p_{i_j}$ für alle $j = 1, ..., k$. Deswegen gilt:

   c) $p_{i_m} \leq \frac{1}{k} G_k$

   Damit ist:
   $OPT \underset{(a)}{\leq} \sum_{j=1}^{k} p_{i_j} + \sum_{j=1}^{l} p_{i_{k+j}} + p_{i_m} \underset{(b)(c)}{\leq} G_k + \frac{1}{k} G_k = \frac{k+1}{k} G_k$
   und wir bekommen
   $G_k \geq \frac{k}{k+1} OPT = (1 - \frac{1}{k+1}) OPT$ □

Man kann zeigen:

- Für $\textsc{Greedy}_k$ ist der Faktor $(1 - \frac{1}{k+1})$ bestmöglich.
- Wenn man G_0 in Zeile 4 von Algorithmus 1.25 durch G'_0 ersetzt, bekommt man einen Approximationsalgorithmus mit Gütegarantie $(1 - \frac{1}{k+2})$.

Man sieht: Für jedes feste $\varepsilon > 0$ gibt es einen polynomiellen Algorithmus für Knapsack, der eine $(1 - \varepsilon)$-Approximation liefert. Das motiviert:

Definition 4.8 (PTAS)**.** Ein polynomielles Approximationsschema (Engl.: Polynomial-time approximation scheme. Kurz: PTAS) für ein Optimierungsproblem ist eine Familie von Algorithmen, die für jedes beliebige, aber feste $\varepsilon > 0$ einen $(1 - \varepsilon)$-Approximationsalgorithmus (bzw. $(1 + \varepsilon)$) liefert.

Korollar 4.9. $\{\textsc{Greedy}_k \mid k \in \mathbb{N}\}$ ist ein PTAS für Knapsack.

5 Komplexität

5.1 Einstieg

Wir haben bisher verschiedene algorithmische Aspekte von Knapsack und Co. kennengelernt:

(A) Heuristisch:	Einfache Probiermethoden, die oft, aber nicht immer, ganz ordentliche Lösungen liefern. $\rightarrow$ GREEDY$_0$
(B) Exakt:	Algorithmen, die immer optimale Lösungen finedn, aber manchmal lange dafür brauchen. $\rightarrow$ *Dynamic-Programming* $\rightarrow$ *Branch-and-Bound*
(C) Approximativ:	Algorithmen, die polynomieller Zeit Lösungen finden, die nicht unbedingt optimal, aber gut sind. $\rightarrow$ GREEDY$_k$

Geht das noch besser? Können wir einen Algorithmus finden, der

1. immer (= für jede Instanz)
2. schnell (= polynomiell in der Codierungsgröße der Instanz)
3. eine optimale Lösung

berechnet?

Also: Gibt es einen Perfekten Algorithmus für Knapsack?

Definition 5.1. Ein algorithmisches Problem Π gehört zur Klasse P, wenn es für Π einen Algorithmus mit polynomieller Laufzeit gibt, der immer eine optimale Lösung findet.

Deutlich einfacher ist das Verifizieren (oder NachPrüfen) einer gültigen Lösung!

Definition 5.2.
Ein Problem Π gehört zur Klasse NP, wenn es für jede beliebige Instanz I von Π die Gültigkeit einer Lösung in polynomieller Zeit nachgeprüft werden kann.

Beobachtung 5.3. Offensichtlich gilt SubsetSum $\in NP$ und 0-1-Knapsack $\in NP$, denn wir müssen nur prüfen, ob die Gewichtsschranke eingehalten bzw. erreicht wird und im zweiten Fall, ob die Wertschranke überschritten wird. Allerdings ist nicht klar, ob SubsetSum $\in P$ oder 0-1-Knapsack $\in P$.

In der Literatur steht NP für „Nichtdeterministisch polynomiell“. Für viele Probleme ist Verifizierung der Existenz einer Lösung relativ leicht - der Nichtexistenz ($\rightarrow coNP$) aber relativ schwer (zumindest anschaulich).

Problem 5.4.
Gilt $P \neq NP$?

5.2 Ein Beispiel mit Logik

Beispiel 5.5.
Wir betrachten die folgende Knapsack-Instanz mit $n = 12$, $z_i = p_i$, $Z = 111444$ und den folgenden Objekten:

$$\begin{aligned}
z_1 = p_1 &= 100110 \\
z_2 = p_2 &= 100001 \\
z_3 = p_3 &= 10101 \\
z_4 = p_4 &= 10010 \\
z_5 = p_5 &= 1001 \\
z_6 = p_6 &= 1110 \\
z_7 = p_7 &= 200 \\
z_8 = p_8 &= 100 \\
z_9 = p_9 &= 20 \\
z_{10} = p_{10} &= 10 \\
z_{11} = p_{11} &= 2 \\
z_{12} = p_{12} &= 1
\end{aligned}$$

Gibt es eine Menge $S \subseteq \{1, ...12\}$ mit $\sum_{i \in S} z_i = \sum_{i \in S} p_i = Z$? Das entspricht dem Entscheidungsproblem SubsetSum. Man sieht:

1. Man muss 1 oder 2 auswählen, aber nicht beide. $\leftarrow$ 1.Ziffer
2. Man muss 3 oder 4 auswählen, aber nicht beide. $\leftarrow$ 2.Ziffer
3. Man muss 5 oder 6 auswählen, aber nicht beide. $\leftarrow$ 3.Ziffer
4. Man muss 1, 3 oder 6 auswählen, dann kann man mit 7 und 8 die Zahl 4 erzeugen. $\leftarrow$ 4.Ziffer
5. Man muss 1, 4 oder 6 auswählen, dann kann man mit 9 und 10 die Zahl 4 erzeugen. $\leftarrow$ 5.Ziffer
6. Man muss 2, 3 oder 5 auswählen, dann kann man mit 11 und 12 die Zahl 4 erzeugen. $\leftarrow$ 6.Ziffer

Es gibt genau dann ein S mit $\sum_{i \in S} p_i = Z$, wenn sich alle sechs Bedingungen erfüllen lassen!

Setzen wir einfach einige Boolesche Variablen:

$$x_1 := \begin{cases} 1, \text{ Objekt 1 wird gewählt} \\ 0, \text{ Objekt 1 wird nicht gewählt, also Objekt 2} \end{cases}$$

$$x_2 := \begin{cases} 1, \text{ Objekt 3 wird gewählt} \\ 0, \text{ Objekt 3 wird nicht gewählt, also Objekt 4} \end{cases}$$

$$x_3 := \begin{cases} 1, \text{ Objekt 5 wird gewählt} \\ 0, \text{ Objekt 5 wird nicht gewählt, also Objekt 6} \end{cases}$$

Damit:
Es gibt ein $S \subseteq \{1, ..., 12\}$ mit $\sum_{i \in S} p_i = Z$
$\Leftrightarrow$ Wir können die logischen Variablen so belegen, dass die folgende Formel erfüllt wird:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

Also:
Wenn wir einen perfekten Algorithmus haben, der also in polynomieller Zeit die Existenz von S entscheiden kann, dann können wir den auch verwenden, um die logische Formel zu knacken - und umgekehrt. Das geht für jede logische Formel vom Typ 3SAT!

Definition 5.6. (3-Satisfiability (3SAT))

Gegeben: Eine Boolesche Formel, zusammengesetzt aus:

- n Boolesche Variablen $x_1, ..., x_n$, aus denen wir Literale ℓ_i der Form x_k oder $\bar{x}_k$ bilden können.
- m Klauseln, jede zusammengesetzt aus genau drei Literalen $C_j = (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ mit $1 \leq j \leq m$.

Gesucht: Eine alle m Klauseln erfüllende (engl: satisfying) Wahrheitsbelegung der n Variablen.

Beobachtung 5.7. Wenn eine 3SAT-Instanz eine erfüllende Wahrheitsbelegung hat, lässt sich eine solche leicht verifizieren. Gibt es keine Lösung, so ist das nicht ohne weiteres schnell nachzuweisen.

Problem 5.8. Gibt es für 3SAT einen Algorithmus, der für jede Instanz I in polynomieller Zeit (in m und n von I) entscheidet, ob I erfüllbar ist?

Satz 5.9. *Wenn* 0-1-Knapsack $\in P$ *ist, dann ist auch* $3SAT \in P$.

Beweis. Analog zu Beispiel 5.5: Baue aus einer 3SAT-Instanz I_{3SAT} eine Instanz I_{KN} von 0-1-Knapsack mit folgenden Eigenschaften:

1. Die Codierungsgröße von I_{KN} ist polynomiell beschränkt durch die Codierungsgröße von I_{3SAT}.

2. Es gibt für I_{KN} eine Teilmenge mit Lösungswert $Z \Leftrightarrow I_{3SAT}$ ist erfüllbar.

Wenn wir einen polynomiellen Algorithmus für Knapsack haben, dann können wir damit in polynomieller Zeit jede Instanz I_{3SAT} von 3SAT entscheiden:

1. Baue zu I_{3SAT} eine Instanz I_{KN} von Knapsack.

2. Löse I_{KN}.

3. Betrachte die Lösung und verwende Eigenschaft 2 um Lösbarkeit von I_{3SAT} zu entscheiden.

Wie konstruieren wir eine Instanz I_{3SAT} aus I_{KN}? Wir betrachten dazu folgenden Algorithmus.

Algorithmus 5.10 3SATtoKnapsack

Eingabe: Instanz I_{3SAT} von 3SAT
Ausgabe: Instanz I_{KN} von 0-1-KNAPSACK

$Z := \sum_{i=0}^{n-1} 10^{m+i} + \sum_{i=0}^{m-1} 4 \cdot 10^i$
$P := Z$
Erstelle Objekte $z_1, \ldots, z_{2n+2m}$ und $p_1, \ldots, p_{2n+2m}$
for $i = 1$ **to** n **do** ▷ Initialisiere Gewichte für Variablen
    $p_{2i-1} := z_{2i-1} := 10^{n+m-i}$
    $p_{2i} := z_{2i} := 10^{n+m-i}$
for $j = 1$ **to** m **do** ▷ Betrachte jede Klausel
    for $i = 1$ **to** n **do** ▷ Setze Bit von Variablen, wenn in Klausel präsent
        if x_i taucht als positives Literal in C_j auf **then**
            $p_{2i-1} := z_{2i-1} := z_{2i-1} + 10^{m-j}$
        else if x_i taucht als negatives Literal in C_j auf **then**
            $p_{2i} := z_{2i} := z_{2i} + 10^{m-j}$
    $p_{2n+2j-1} := z_{2n+2j-1} := 2 \cdot 10^{m-j}$ ▷ Gewichte um auf 4 aufzufüllen,
    $p_{2n+2j} := z_{2n+2j} := 1 \cdot 10^{m-j}$ ▷ wenn Klausel erfüllt wurde.
return

In Abbildung 5.1 ist ein Beispiel dieser Reduktion gegeben. Man erkennt, dass für jede Variable x_i entweder Objekt $2i-1$ oder $2i$ gewählt werden kann, aber nicht beide. Im Klauselbereich können wir die Vieren in Z nur dann erzeugen, wenn wir mit den Variablengewichten in jeder Spalte mindestens eine Eins ausgewählt haben.

Damit ist I_{3SAT} genau dann erfüllbar, wenn I_{KN} eine Lösung hat. □

Satz 5.11 (Satz von Cook 1971)**.** *Wenn $3SAT \in P$, dann gilt $P = NP$.*

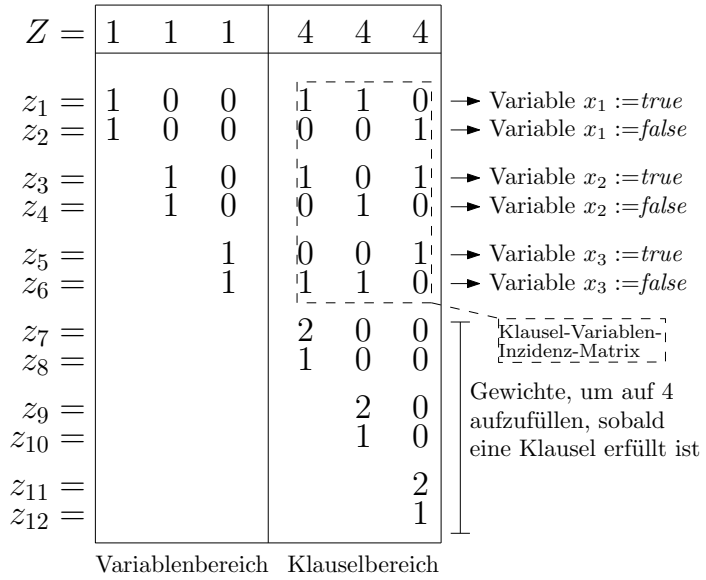


Abbildung 5.1: Beispiel einer Reduktion von der 3SAT-Instanz $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$ auf eine Instanz von 0-1-Knapsack. Es gilt $P := Z$ und $p_i := z_i$ für alle $i \in \{1, \ldots, 12\}$.

Beweisidee:
Man kann zeigen, dass sich jedes Problem in NP als äquivalentes 3SAT-Problem codieren lässt.

Korollar 5.12. Wenn $Knapsack \in P$, dann gilt $P = NP$.

Definition 5.13.

1. Ein Problem Π in NP heißt NP-vollständig, wenn $\Pi \in P \Rightarrow P = NP$ gilt.
2. Ein Problem Π heißt NP-schwer, wenn $\Pi \in P \Rightarrow P = NP$ gilt.

Mit dieser Definition und bekannten Sätzen folgt sofort folgendes Korollar.

Korollar 5.14.

1. 3SAT ist NP-vollständig.
2. Knapsack$_{\leq}$ ist NP-vollständig.
3. Knapsack ist NP-schwer.

Konsequenzen?!

	Idealer Finanzberater	Perfekter Algorithmus
(1)	ehrlicher	immer
(2)	intelligenter	schnell
(3)	Investmentbänker	optimale Lösung

In Zeiten der Finanzkrise: Der Schnitt ist leer!
Bei NP-vollständigkeit: Der Schnitt ist leer! (Bei $P \neq NP$)
Was können wir in schwierigen Situationen tun?

(A) Auf Glück vertrauen

(B) Hart arbeiten

(C) Erwartungen herabschrauben

(D) Mit Schicksal hadern und diskutieren

Hier:

(A) nicht immer: Heuristiken

(B) nicht schnell: Exakte Algorithmen (Branch-and-Bound)

(C) nicht optimal: Approximationsanalyse

(D) nicht NP-vollständig: Komplexitätsanalyse

6 Graphenproblem: Vertex Cover

Graphenprobleme sind auch wichtig, interessant und können auch schwierig sein.

Definition 6.1 (Vertex Cover)**.**
Gegeben: Ein Graph $G =< V, E >$
Gesucht: Eine Knotenmenge $S \subseteq V$ möglichst kleiner Kardinalität, sodass für jede Kante $e = \{u, v\} \in E$ gilt: $u \in S$ oder $v \in S$.

Beispiel 6.2.
Betrachte den Graphen G aus Abbildung 6.1 (auch bekannt unter dem Namen *Petersen-Graph*) mit einem Vertex Cover der Größe 6.

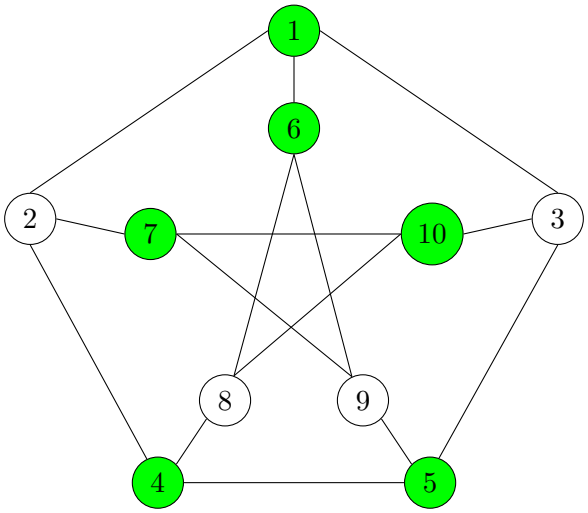


Abbildung 6.1: Der Petersen-Graph mit einem möglichen Vertex Cover.

Wir beobachten, dass wir den Graphen in zwei Kreise aufteilen können: einen äußeren (Knoten 1 bis 5) und einen inneren (Knoten 6 bis 10) Kreis. Für die Kreise der Länge 5 brauchen wir jeweils 3 Knoten im Vertex Cover. Also müssen in G mindestens 6 Knoten im Vertex Cover vorhanden sein.

Beispiel 6.3.
Betrachte nun den Graphen in Abbildung 6.2. Wie groß ist ein minimales Vertex Cover? Man kann schnell erkennen, dass wir mindestens zehn Knoten benötigen. Betrachtet man den Graphen ohne rote Kanten, benötigen wir einen Knoten pro obere Kante und zwei pro Dreieck. Für diese Instanz sind zehn Knoten ausreichend.

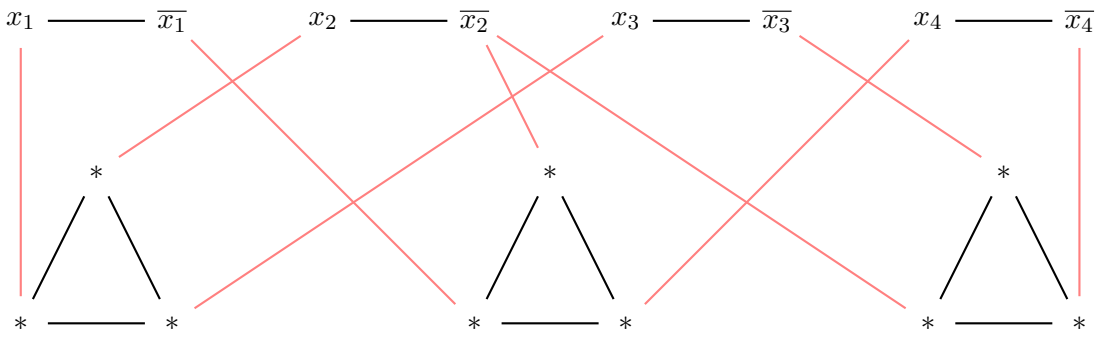


Abbildung 6.2: Ein Graph, der aus der Reduktion der 3SAT-Instanz $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$ entstanden ist

Mit einem scharfen Blick erkennt man:
$\exists VC$ mit 10 Knoten $\Leftrightarrow (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$

Satz 6.4. *Vertex Cover ist NP-schwer.*

Beweis. Wir geben eine Reduktion von 3SAT an:
Eine allgemeine Konstruktionsmethode, die aus jeder 3SAT-Instanz I_{3SAT} eine VC-Instanz I_{VC} konstruiert, sodass

1. Die Größe von I_{VC} polynomiell in der Größe von I_{3SAT} beschränkt ist.
2. I_{VC} ein VC der Größe $n + 2m$ hat $\Leftrightarrow$ I_{3SAT} ist erfüllbar. □

Da Vertex Cover NP-schwer ist, bieten sich also die bekannten Methoden (Heuristiken, Exakt und Approximationen) an. Wie kann man Vertex Cover approximieren? Wir wissen, dass für jede Kante einer der beiden Enden im Vertex Cover liegen muss. Wir können folgendes beobachten: Nimmt man sich unabhängige Kanten, d.h. die Endpunkte überlappen sich nicht, erhalten wir eine untere Schranke. Genauer:

Definition 6.5. Sei $G = (V, E)$ ein Graph. Eine Menge M von Kanten aus E heißt *inklusionsmaximales Matching*, wenn

1. keine zwei Kanten in M existieren, die einen Knoten gemeinsam haben, und
2. keine Kante zu M hinzugefügt werden kann, sodass immer noch 1. gilt.

Satz 6.6. *Sei M ein inklusionsmaximales Matching und OPT_{VC} der Wert eines kleinsten Vertex Covers in einem Graphen G. Dann gilt $|M| \leq \text{OPT}_{VC} \leq 2|M|$.*

Beweis. Die linke Seite der Ungleichung geht aus der Beobachtung hervor. Betrachten wir also die rechte Seite $\text{OPT}_{VC} \leq 2|M|$. Wir konstruieren uns folgende Knotenmenge VC_M: Für jede Kante $e = \{v, w\} \in M$ nehmen wir v und w in VC_M auf. Es ist klar,

dass $|VC_M| \leq 2|M|$ gilt. Bleibt zu zeigen, dass VC_M ein Vertex Cover ist. Angenommen VC_M ist kein Vertex Cover. Dann muss eine Kante f existieren, dessen Endpunkte nicht in VC_M liegen. Das bedeutet wiederum, dass wir M mit f erweitern können. Also war M nicht inklusionsmaximal, was einen Widerspruch darstellt. □

Korollar 6.7. Es existiert eine 2-Approximation für Vertex Cover.

7 Hashing

Wir betrachten nun die Dynamische Verwaltung von Daten, wobei jeder Datensatz eindeutig durch einen Schlüssel charakterisiert wird. Viele Anwendungen benötigen nur einfache Daten-Zugriffsmechanismen (*dictionary operations*) wie Suchen oder Löschen eines Schlüssels x und Einfügen eines neuen Datensatzes d mit Schlüssel x. Das Problem: Die Menge potentieller Schlüssel (Universum) kann sehr groß sein, die auftretende Schlüsselmenge S kann aber sehr klein sein und ist im Allgemeinen vorher nicht bekannt. Man möchte nun also durch Berechnungen feststellen, wo der Datensatz mit Schlüssel x gespeichert ist.

Definition 7.1. Eine *Hashtabelle* der Größe m besteht aus einem Array T mit Speicherzellen $T[0]$ bis $T[m-1]$. Die Datensätze werden in dieser Hashtabelle abgespeichert.

Eine *Hashfunktion* h liefert für jeden Schlüssel $x \in U$ eine Adresse in der Hashtabelle, d.h. $h : U \to \{0, \ldots, m-1\}$.

Vorteil beim Hashing ist, dass alle Operation im Mittel konstante Komplexität besitzen. Zum Vergleich: Balancierte Suchbäume, wie AVL-Bäume, B-Bäume oder Rot-Schwarz-Bäume, besitzen eine Komplexität von $O(\log n)$.

Definition 7.2 (Belegungsfaktor)**.** Der Quotient $\beta := \frac{n}{m}$ heißt Belegungsfaktor oder Auslastungsfaktor einer Hashtabelle der Größe m

Wie wir sehen werden, lässt sich der Aufwand für dictionary operations als Funktion in β abschätzen. Die Anzahl aktueller Schlüssel geht also nur indirekt in den Aufwand ein.

Wie schon erwähnt ist die Anzahl möglicher Schlüssel viel größer als die Hashtabelle, also $|U| \gg m$. Nach dem Schubfachprinzip gibt es verschiedene Schlüssel x_1 und x_2, die auf die gleiche Adresse abgebildet werden. Sind zwei solche Schlüssel in der aktuellen Schlüsselmenge, entsteht eine Adresskollision. Diese muss behoben werden.

7.1 Hashfunktionen

Definition 7.3 (Hashverfahren)**.** Ein Hashverfahren ist gegeben durch:

- eine Hashtabelle,
- eine Hashfunktion, und
- eine Strategie zur Auflösung möglicher Adresskollisionen.

Eine gute Hashfunktion sollte

- surjektiv sein, d.h. den ganzen Wertebereich umfassen,
- die zu speichernden Schlüssel (möglichst) gleichmäßig verteilen, d.h. für alle Speicherplätze i und j sollte gelten $|h^{-1}(i)| \approx |h^{-1}(j)|$,
- effizient berechenbar sein.

Wir nehmen nun an, dass die Schlüssel durchnummeriert sind, also $U = \{0, 1, \ldots, N-1\}$. Häufig wird eine Division- und Kongruenzmethode benutzt. Also: $h(x) :\equiv x \mod m$ woraus folgt $|h^{-1}(i)| \in \left\{ \left\lfloor \frac{N}{m} \right\rfloor, \left\lceil \frac{N}{m} \right\rceil \right\}$.

Problem: Die Daten sind oft nicht gleichverteilt! Wichtig ist eine geeignete Wahl von m. Ist m eine Zweierpotenz, wählt $x \mod m$ nur die letzten $\log m$ Bits. Ist m hingegen eine Primzahl, so beeinflusst $x \mod m$ alle Bits.

Beispiel 7.4. Sei $m = 11$ und $S = \{49, 22, 6, 52, 76, 34, 13, 29\}$ Wir erhalten folgende Hashwerte: $h(49) \equiv 49 \mod 11 = 5$, $h(22) \equiv 22 \mod 11 = 0$, 6, 8, 10, 1, 2 und 7

7.2 Kollisionen

Wie bereits erwähnt, ist es unvermeidbar, dass Kollisionen auftreten. Was können wir tun, falls eine Kollision auftritt und wie wahrscheinlich ist das?

Problem 7.5 (Geburtstagsparadoxon)**.** Bei wie vielen (zufällig gewählten) Person ist es wahrscheinlich, dass hiervon zwei am selben Datum (Tag und Monat) Geburtstag haben?

Wir nehmen dazu an, dass die Daten unabhängig sind und dass $\text{Prob}(h(x) = j) = \frac{1}{m}$. Dann ist die Wahrscheinlichkeit, dass das i-te Datum nicht mit den ersten $i-1$ Daten kollidiert, sofern diese kollisionsfrei sind $\frac{m-(i-1)}{m}$. Es ist also egal, welche Speicherplätze die ersten $i-1$ Daten belegen; $m - i + 1$ der m Möglichkeiten sind gut. Damit gilt: $\text{Prob}(n \text{ Daten kollisionsfrei}) = \frac{m-1}{m} \cdot \frac{m-2}{m} \ldots \frac{m-n+1}{m}$

Beispiel 7.6. Sei $m = 365$. Dann ist Prob(23 Daten kollisionsfrei)≈ 0.49 und Prob(50 Daten kollisionsfrei)≈ 0.03.

Satz 7.7. *Die Wahrscheinlichkeit, dass $2m^{1/2}$ Daten kollisionsfrei eingefügt werden können, liegt bei höchstens 36,79%.*

Beweis. $\text{Prob}(2m^{1/2} \text{ Daten kollisionsfrei}) = \frac{m-1}{m} \ldots \frac{m-m^{1/2}}{m} \ldots \frac{m-2m^{1/2}+1}{m} \leq 1 \cdot \left(\frac{m-m^{1/2}}{m}\right)^{m^{1/2}} = (1 - \frac{1}{m^{1/2}})^{m^{1/2}} \approx \frac{1}{e} \approx 0.3679$ □

Man erkennt, dass Hashing mit Kollisionen leben muss und dass man Strategien zu Kollisionsbehandlung benötigt. Es gibt folgende Möglichkeiten zur Kollisionsbehandlung:

1. Verkettete Listen: Jede Komponente der Hashtabelle enthält einen Zeiger auf eine Überlaufliste.
2. offene Adressierung: Im Kollisionsfall nach fester Regel alternativen freien Platz in der Hashtabelle suchen. Für jeden Schlüssel ist die Reihenfolge, in der Speicherplätze betrachtet werden, vorgegeben: Sondierungsfolge.

7.2.1 Verkettete Liste

Realisierung: Jede Komponente der Hashtabelle enthält einen Zeiger auf paarweise disjunkte lineare Listen. Die i-te Liste $L(i)$ enthält alle Schlüssel $x \in S$ mit $h(x) = i$. Der Vorteil ist, dass alle Operationen unterstützt werden. Außerdem darf n größer als m sein (für $n \gg m$ ist *Rehashing* ratsam). Ein Nachteil ist, dass zusätzlicher Speicher für die Zeiger benötigt wird. Die Operationen im Überblick:

search(x) Berechne $h(x)$ und suche in Liste $L(h(x))$.

insert(x) Nach erfolgloser Suche: Berechne $h(x)$ und füge x in Liste $L(h(x))$ ein.

delete(x) Nach erfolgreicher Suche: Berechne $h(x)$, suche x in Liste $L(h(x))$ und entferne x.

Beispiel 7.8. Sei $m = 7$ und $h(x) = x \mod m$.

Betrachte die Schlüsselmenge $S = \{2, 5, 12, 15, 19, 43, 53\}$. Einfügen in die Hashtabelle liefert die in Abbildung 7.1 gefüllte Hashtabelle.

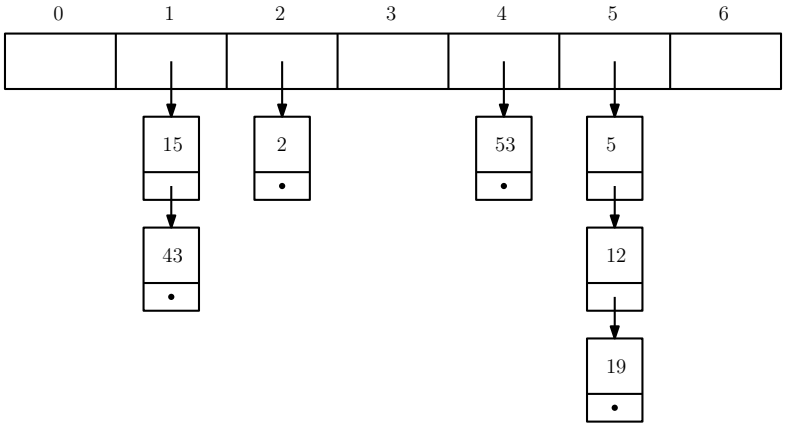


Abbildung 7.1: Gefüllte Hashtabelle mit verketteten Listen.

Satz 7.9. *Bei zufälligen Daten und ideal streuenden Hashfunktionen enthält eine beliebige Liste $L(j)$ im Erwartungsfall $\frac{n}{m} = \beta$ Elemente.*

Beweis. Betrachte Zufallsvariable $X_{ij} = \begin{cases} 1, i\text{-tes Datum kommt in Liste } L(j) \\ 0, \text{sonst} \end{cases}$.

Es gilt $\text{Prob}(X_{ij} = 1) = \frac{1}{m}$, woraus folgt: $E(X_{ij}) = 1 \cdot \frac{1}{m} + 0 \cdot \frac{m-1}{m} = \frac{1}{m}$. Sei nun $X_j = \sum_{i=1}^{n} X_{ij}$, d.h. X_j zählt, wie viele Daten in Liste $L(j)$ kommen. Dann ist $E(X_j) = E(\sum_{i=1}^{n} X_{ij}) = E(X_{1j}) + \cdots + E(X_{nj}) = \frac{n}{m} = \beta$ □

Damit müssen wir für eine erfolglose Suche in Liste $L(j)$ durchschnittlich (inklusive NIL-Zeiger) $1 + \beta$ Objekte betrachten. Beispielsweise für $n = 0.95m$ ist das 1.95. Für eine erfolgreiche Suche müssen wir die Position in der Liste berücksichtigen. Sei l die Länge

von Liste $L(j)$. Da jede Position in der Liste die gleiche Wahrscheinlichkeit $\frac{1}{l}$ besitzt, befindet sich das gesuchte Element im Erwartungsfall an Position $\frac{1}{l}(1+2+\cdots+l) = \frac{l+1}{2}$. Im Durchschnitt ist die Listenlänge $1 + \frac{n-1}{m}$ (Liste enthält sicher das Datum, und die anderen $n-1$ Daten sind zufällig verteilt). Damit erhalten wir eine erwartete Suchdauer von $\frac{1}{2}(1 + \frac{n-1}{m} + 1) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\beta}{2}$. Für $n = 0.95m$ ist das 1.475.

7.2.2 Offene Addressierung

Betrachten wir nun die zweite Variante zur Kollisionsbehandlung: die offene Adressierung. Im Kollisionfall wird nach einer festen Regel ein alternativer freier Platz in der Hashtabelle gesucht (Sondierungsfolge). Voraussetzung: Auswertung von h gilt als eine Operation. Sei $t(i,j)$ die Position des i-ten Versuchs zum Einfügen von Datum x mit $h(x) = j$. Anforderung an t sind:

- t ist in $O(1)$ berechenbar,
- $t(0,j) = j$,
- $t(\cdot, j) : \{0,\ldots,m-1\} \to \{0,\ldots,m-1\}$ bijektiv

Damit können wir nun unsere Operationen definieren:

search(x) Berechne $j := h(x)$. Suche x an den Position $t(0,j),\ldots,t(m-1,j)$. Brich ab, falls x oder eine freie Stelle gefunden wurde. Im letzteren Fall gibt es kein Datum mit Schlüssel x.

insert(x) Nach erfolgloser Suche: Finde freien Platz (sonst Overflow) und x dort eintragen.

Die Operation delete(x) kann nicht ohne Weiteres ausgeführt werden. Würden wir einen Schlüssel einfach Löschen, entstehen Lücken, welche bei einer Search-Operation zu Fehlern führen würde. Eine Möglichkeit ist, Felder mit *besetzt*, *noch nie besetzt* oder *wieder frei* zu markieren. Eine Suche würde nur bei *noch nie besetzt* Feldern frühzeitig abbrechen. Im Laufe der Zeit wird es allerdings keine Position mehr geben, die noch nie besetzt wird, wodurch das Hashing ineffizient wird. Wir betrachten beim offenen Hashing also nur search und insert Operationen.

Lineares Sondieren

Schauen wir uns zunächst lineares Sondieren an. Hier ist $t(i,j)$ in der Form $(ci + j)$ mod m mit $c \in \mathbb{R}$. Es wird auf den Hashwert j also ein linearer Zähler addiert.

Beispiel 7.10. Sei $m = 19$ und $j = h(x) = 7$. Wir erhalten die Sondierungsfolge: 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 0, 1, 2, 3, 4, 5, 6.

Mann kann schnell sehen, dass immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen. Diese Abschnitte heißen *Cluster*. Durch diese Clusterbildung entstehen erhöhte Suchzeiten.

Beispiel 7.11. Betrachte eine Hashtabelle mit $m = 19$ Feldernm wobei Positionen 2, 5, 6, 9, 10, 11, 12 und 17 belegt sind.

$h(x)$:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
landet an Position:	0	1	3	3	4	7	7	7	8	13	13	13	13	13	14	15	16	18	18
W.keit:	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$	$\frac{1}{19}$	0	0	$\frac{3}{19}$	$\frac{1}{19}$	0	0	0	0	$\frac{5}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$

Ein Wunsch beim Hashing ist es, dass zu jedem Zeitpunkt alle Positionen die gleiche Wahrscheinlichkeit haben, besetzt zu werden.

Beobachtung 7.12. Das geht numerisch nicht genau. Im Beispiel 7.11 gibt es elf freie Plätze, aber 19 Hashwerte. Also haben alle Wahrscheinlichkeit $\frac{k}{19}$ und nicht $\frac{k}{11}$.

Definition 7.13 (Modell des idealen Hashings)**.** Beim *idealen Hashing* haben alle $\binom{m}{n}$ Möglichkeiten, die n besetzten Plätze für m Schlüssel auszuwählen, die gleiche Wahrscheinlichkeit.

Man sieht schnell: Lineares Sondieren ist weit vom idealen Hashing entfernt.

Quadratisches Sondieren

Eine weitere Möglichkeit zum Finden eines freien Platzes bieten quadratisches Sondieren. Hier wächst der additive Term nicht linear, sondern quadratisch. Beispielhaft kann $t(i, j)$ die Form $j + (-1)^{i+1} \cdot \lfloor \frac{i+1}{2} \rfloor^2 \mod m$ besitzen.

Beispiel 7.14. Sei $m = 19$, $j = h(x) = 7$ und $t(i, j) = j + (-1)^{i+1} \cdot \lfloor \frac{i+1}{2} \rfloor^2 \mod m$. Damit erhalten wir die Sondierungsfolge 7, 8, 6, 11, 3, 16, 17, 4, 10, 13, 1, 5, 9, 18, 15, 14, 0, 12, 2.

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv? Antwort: Nein. Aber immer wenn $m \equiv 3 mod 4$ und m eine Primzahl ist. Den Beweis dazu findet man in der Zahlentheorie.

Insgesamt ist quadratisches Sondieren besser geeignet als lineares Sondieren. Allerdings sind für großes m die ersten Werte noch nah an j.

Multiplikatives Sondieren

Beim multiplikativen Sondieren werden Versuchszähler und Hashfunktion miteinander multipliziert: $t(i, j) := i \cdot j \mod m$ mit $1 \leq i \leq m - 1$. Es muss darauf geachtet werden, dass die benutzte Hashfunktion nicht auf 0 abbildet, ansonsten kann bei einer Kollision gegebenenfalls kein freier Platz gefunden werden. Benutzen wir beispielsweise $h(x) = x \mod (m - 1) + 1$. Diese Funktion ergibt einen Wert zwischen 1 und $m - 1$.

Beispiel 7.15. Sei $m = 19$ und $j = h(x) = 7$. Wir erhalten die folgende Sondierungsfolge:

$i \cdot j$	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126
$i \cdot j \mod 19$	7	14	2	9	16	4	11	18	6	13	1	8	15	3	10	17	5	12

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv? Antwort: Nein, aber immer wenn m eine Primzahl ist und $j \neq 0$. Das können wir durch einen Widerspruch beweisen. Angenommen, dass gilt nicht, dann gäbe es $1 \leq i_1 < i_2 \leq m-1$ mit

$$\begin{aligned} & i_1 j \equiv i_2 j \mod m \\ \Rightarrow\ & j(i_2 - i_1) \equiv 0 \mod m \\ \Rightarrow\ & j(i_2 - i_1) \text{ ist Vielfaches von } m \end{aligned}$$

Das heißt die Primfaktorzerlegung von $j(i_2 - i_1)$ muss m enthalten. Das ist ein Widerspruch dazu, dass $1 \leq j \leq m-1$ und $1 \leq i_1 < i_2 \leq m-1$.

Doppeltes Hashing

Eine letzte Möglichkeit, die wir hier vorstellen wollen, ist doppeltes Hashing. Hier benutzen wir zwei Hashfunktionen $h_1(x)$ und $h_2(x)$, welche miteinander additiv verknüpft werden. Eine der beiden Hashfunktionen wird mit dem Versuchszähler multiplikativ verknüpft. Damit erhalten wir beispielsweise $h(i, x) := h_1(x) + i \cdot h_2(x)$ mit $h_1(x) := x \mod m$, $h_2(x) := x \mod (m-2) + 1$ und $0 \leq i \leq m-1$.

Beispiel 7.16. Mit $m = 19$ und $x = 47$ erhalten wir $h_1(47) = 47 \mod 19 = 9$ und $h_2(47) = 47 \mod 17 + 1 = 14$. Damit ergibt sich die Sondierungfolge: 9, 4, 18, 13, 8, 3, 17, 12, 7, 12, 7, 2, 16, 11, 6, 1, 15, 10, 5, 0, 14.

Beobachtung 7.17. $i \cdot h_2(x)$ durchläuft für $1 \leq i \leq m-1$ die Werte $1, \ldots, m-1$ in irgendeiner Reihenfolge; ergänzt wird $0 = 0 \cdot h_2(x)$. Durch den Summanden $h_1(x)$ wird der Anfang zufällig verschoben. Doppeltes Hashing kommt dem idealen Hashing am nächsten.

7.3 Universelles Hashing

Trotz hervorragender Average-Case-Komplexität von $\Theta(1)$, gibt es ein sehr schlechtes Worst-Case-Verhalten mit Laufzeit $\Theta(n)$. Dies kommt dadurch zu Stande, dass bei festgelegter Hashfunktion eine bestimmte Schlüsselmenge existiert, die dieses ungünstige Verhalten hervorrufen kann. Aushilfe verschafft universelles Hashing: Zur Laufzeit wird eine Hashfunktion aus einer Klasse von Hashfunktionen (mit besonderen Eigenschaften) ausgewählt. Damit erreicht man, bezogen auf die Zufallsauswahl der Hashfunktion, für jede Schlüsselmenge $S \subseteq U$ ein gutes Laufzeitverhalten im Mittel.

Literaturverzeichnis

[1] R. Bellman, R.E. Bellman, and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[3] AH Land and AG Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28, 1960.

[4] Randall Munroe. NP-Complete. *https://xkcd.com/287/*.

[5] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *J. ACM*, 22(1):115–124, 1975.