HANSER

Leseprobe

zu

„Praktische Komplexitätstheorie in Beispielen“

von Jochen Kripfganz und Holger Perlt

E-Book-ISBN: 978-3-446-46532-9
E-Pub-ISBN: 978-3-446-46708-8

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-46532-9
sowie im Buchhandel

Vorwort und Hinweise zur Nutzung des Buches

Das vorliegende Buch **„Praktische Komplexitätstheorie in Beispielen“** möchte den Leser anhand ausgewählter Texte und interaktiver Beispiele in das Gebiet der Komplexitätsuntersuchungen algorithmischer Berechnungen einführen. Es ist offensichtlich, dass gegenwärtig und zukünftig die Anforderungen an die Verarbeitung immer größerer Datenmengen weiter steigen werden. Mithilfe intelligenter Algorithmen sollen innere Zusammenhänge entdeckt bzw. verifiziert oder Problemstellungen gelöst werden. Damit sind gesicherte Bewertungen dieser Algorithmen von entscheidender Bedeutung, sowohl was die Korrektheit als auch die Laufzeit betrifft. Insbesondere der zweite Aspekt ist Gegenstand unserer Betrachtungen. Wir verstehen unser Buch nicht als umfassende Darstellung dieses umfangreichen Gebietes der theoretischen Informatik. Wir haben einige Problemstellungen herausgesucht, welche unserer Meinung nach die wesentlichen Aspekte charakterisieren. In den einzelnen Abschnitten wird dann auf die weiterführende Literatur verwiesen.

Das Buch ist vollständig im notebook-Format des Programmsystems **Mathematica** (*Wolfram Research, Inc. (2012)*) geschrieben. Dies erlaubt eine effektive Integration von Text, Formeln und interaktiven Beispielen. Es ist mit dem kostenlosen **Wolfram Player** (ab Version 12) zu benutzen. Der Leser wird durch **Hyperlinks** oder dynamische **Buttons** durch das Buch geführt. Die einzelnen Kapitel sind als separate notebooks konzipiert, um die Übersichtlichkeit beim Lesen zu sichern. Von jedem Kapitel kommt man wieder über einen derartigen Button zurück zum übergeordneten Abschnitt. Zu jedem Kapitel gehören Beispiele, welche die besprochenen Verfahren demonstrieren. Durch Veränderung bestimmter Parameter kann der Leser diese Aufgaben interaktiv verändern und so im vorgegebenen Rahmen eigene Probleme lösen. Es ist dieser integrative Aspekt, der uns diese neuartige Form nahelegte. Wir hoffen, dass der Leser davon in möglichst hohem Maße profitiert.

Das Buch im nb-Format können Sie sich hier herunterladen:

https://plus.hanser-fachbuch.de/

Entpacken Sie das Archiv **Praktische_Komplexitätstheorie.zip** in ein Verzeichnis Ihrer Wahl. Dabei sollte die Hierarchie

Buch.nb

Beispiele

Kapitel

beibehalten werden.

Das Buch wird als Datei Buch.nb mit dem Wolfram Player geöffnet.

Zum Download und zur Installation des kostenlosen Wolfram Players gehen Sie bitte auf die Seite *https://www.wolfram.com/player/* und befolgen die dort gegebenen Hinweise. Der Player ist für die Betriebssysteme Windows/Linux/MacOS verfügbar.

Die Autoren

Prof. Dr. Jochen Kripfganz hat auf dem Gebiet der Theoretischen Elementarteilchenphysik gearbeitet und besitzt umfangreiche Lehrerfahrung. Gegenwärtig verantwortet er die Lehrveranstaltung Theoretische Informatik an der Staatlichen Studienakademie Leipzig.

Dr. Holger Perlt hat an der Universität Leipzig Physik studiert. Er arbeitet auf dem Gebiet der Theoretischen Elementarteilchenphysik, insbesondere zu störungstheoretischen und nichtstörungstheoretischen Methoden. Derzeit hält er die Vorlesungen zu Datenstrukturen und Algorithmen sowie zur Numerik an der Staatlichen Studienakademie Leipzig.

Beide Autoren arbeiten seit den 1990er-Jahren mit dem Programmsystem ***Mathematica*** der Firma Wolfram Research, Inc. Sie sind Autoren des Buches „Arbeiten mit Mathematica“ (erschienen im Hanser Verlag) und des Mathematica-Programmpaketes „OperationsResearch“ (vertrieben als Drittprodukt von Wolfram Research).

Inhalt

1 Einführung

Wenn wir eine Berechnung durchführen, dann lösen wir dabei, ohne das unbedingt so zu nennen, ein algorithmisches Problem. Ist das Problem wohldefiniert, so gehört zu jedem zulässigen Input (aus einer Menge M_1) eindeutig ein bestimmter Output (aus einer Menge M_2). Anders ausgedrückt, wollen wir also eine Abbildung f: $M_1 \rightarrow M_2$ berechnen. Wenn es einen Algorithmus (eine Rechenvorschrift bzw. eine Folge von Handlungsanweisungen) gibt, der das leistet, dann nennen wir das Problem ein algorithmisches Problem bzw. das Problem heißt berechenbar. In diesem Sinne ist das Backen eines Kuchens ein algorithmisches Problem.

Es ist nun keinesfalls so, dass alle wohldefinierten Probleme auch berechenbar sind. Nehmen wir als Beispiel folgende Fragestellung, das sogenannte Halteproblem: Zu einem beliebigen Computerprogramm (ohne offensichtliche Syntaxfehler) soll vorab ermittelt werden, ob dieses Programm auf einer wiederum beliebigen (aber zulässigen) Eingabe nach endlicher Zeit hält, oder sich in eine Endlosschleife verabschiedet. Interessanterweise können wir zur Lösung dieser Aufgabe keinen Algorithmus angeben, d. h. kein Programm schreiben, das diese Aufgabe allgemein löst (für spezielle Fälle ist das natürlich möglich). Wir können es höchstens jeweils ausprobieren, und sind dann aber nur erfolgreich, wenn das Programm auf der jeweiligen Eingabe tatsächlich hält. Ansonsten müssen wir das Programm irgendwann abbrechen, ohne ausschließen zu können, dass das Programm später doch noch gehalten hätte.

Mit Fragestellungen dieser Art beschäftigt sich die Berechenbarkeitstheorie, die nicht Gegenstand dieses Buches ist. Unglücklicherweise gibt es aber eine ganze Reihe von praktisch relevanten Problemen, die in obigem Sinne sehr wohl berechenbar sind, praktisch aber leider nicht. Das liegt daran, dass (zumindest mit den bekannten Methoden) die Berechnungszeit exponentiell mit der Größe der Eingabe anwächst.

Ein Beispiel ist das Problem des Handelsreisenden (Rundreiseproblem): Ein Handelsreisender möchte an einem Tag n Kunden besuchen, und danach nach Hause zurückkehren. Das möchte er natürlich so schnell wie möglich erledigen. Es zeigt sich, dass die Rechenzeit für die Bestimmung der kürzesten Rundreise schon bei

einigen Hundert Kunden das Alter des Universums (ca. 10^{13} Mrd. Jahre) übersteigt, und damit offensichtlich nicht effizient möglich ist. Eine Skalierung gängiger Rechner-Architekturen hilft hier auch nicht wesentlich weiter.

Mit Problemen dieser Art und auch ihrer näherungsweisen Behandlung werden wir uns in diesem Buch detailliert beschäftigen. Zunächst folgt aber eine methodische Einführung in das Gebiet der Algorithmen und Datenstrukturen.

2 Algorithmen und Datenstrukturen

2.1 Algorithmen

Allgemein kann man einen **Algorithmus** als eine Handlungsvorschrift bezeichnen, die ausgehend von bestimmten Vorbedingungen ein definiertes Ziel erreicht. In unserem Zusammenhang könnte man präzisieren: Ein Algorithmus beschreibt in einer endlichen Zahl grundlegender Anweisungen den Übergang von einem definierten Anfangs- in einen definierten Endzustand. Diese Beschreibung kann in freier Textform, in strukturierter Textform (Pseudocode) oder in einer Programmsprache erfolgen. Letztendlich soll diese Formulierung zu einem lauffähigen Computerprogramm führen.

Die folgenden Unterabschnitte geben eine kurze Einführung in dieses Gebiet, soweit es für unser Buch sinnvoll erscheint. Ansonsten verweisen wir auf die umfangreiche Standardliteratur (siehe z. B. *Sedgewick (2014)*, *Schöning (2011)* oder *Ottmann u. a. (2012)*).

2.1.1 Landau-Notation

Für die Laufzeit-Einschätzung von Algorithmen ist die Bestimmung der Größenordnung von Funktionen von wesentlicher Bedeutung (**Landau-Notation**).

- **Die O Notation**
  Wir betrachten zwei Funktionen *f(x)*, *g(x)*. Gibt es eine Konstante $c > 0$ und ein $x_0 \geq 0$, so dass gilt

  $$\exists c > 0 \exists x_0 \geq 0 \forall x > x_0 : f(x) \leq c\, g(x)$$

  dann schreiben wir $f(x) = O(g(x))$.

- **Die Ω Notation**
  Wir betrachten zwei Funktionen *f(x)*, *g(x)*. Gibt es eine Konstante $c > 0$ und ein $x_0 \geq 0$, so dass gilt

  $$\exists c > 0 \exists x_0 \geq 0 \forall x > x_0 : f(x) \succeq cg(x)$$

  dann schreiben wir $f(x) = \Omega(g(x))$.
- **Die Θ Notation**
  Wir betrachten zwei Funktionen *f(x)*, *g(x)*. Gibt es eine Konstante $c > 0$ und ein $x_0 \geq 0$, so dass gilt

  $$\exists c > 0 \exists x_0 \geq 0 \forall x > x_0 : f(x) \succeq c\,g(x) \text{ und } f(x) \preceq cg(x)$$

  dann schreiben wir $f(x) = \Theta(g(x))$.

2.1.2 Optimierungsprobleme/Entscheidungsprobleme

Die Probleme, die in diesem Buch behandelt werden, kann man in zwei große Gruppen unterteilen. **Optimierungsprobleme** suchen unter allen gültigen Lösungen eines Problems die beste(n) aus. Eine mathematische Formulierung in der Menge der reellen Zahlen $\mathbb{R}$ wäre:

$$\min_x f(x)$$

mit
$c_i(x) \leq 0$, i=1,... ,k
$h_j(x) = 0$,j=1,... ,m

- $f(x)$: $\mathbb{R}^n \rightarrow \mathbb{R}$: Zielfunktion
- $x \in \mathbb{R}^n$: zu bestimmende Lösung
- $c_j(x)$, $h_j(x) \in \mathbb{R}$: Randbedingungen

Die **Gültigkeit der Lösung** ergibt sich durch die Befolgung der Randbedingungen. Sind die $x \in \mathbb{R}^n$ ganz oder teilweise diskret (zum Beispiel ganze Zahlen), spricht man von diskreter Optimierung.

Es ist von großer Bedeutung (und teils auch sehr schwierig) zu beweisen, dass ein gewählter Algorithmus das Optimierungsziel nach endlich vielen Schritten erreicht (Optimalität). Dabei spielt die Größe des Problems eine entscheidende Rolle. Es kann sein, dass die Zahl der Schritte, die durch diesen Algorithmus festgelegt sind, mit der Problemgröße so stark steigt, dass das Erreichen des Ziels in einer akzeptablen Zeit unmöglich ist. Dann behilft man sich mit Algorithmen, die wesentlich weniger Schritte benötigen, aber die Optimalität nicht gewährleisten.

Derartige Algorithmen nennt man **Heuristiken** (für das betrachtete Problem). Die Schwierigkeit besteht nun darin abzuschätzen, wie groß der Abstand der heuristischen Lösung von der optimalen Lösung ist. Die wohl prominenteste Aufgabenstellung, die das Verhältnis von Heuristik und „optimalem" Algorithmus beleuchtet, ist das **Rundreiseproblem**.

Entscheidungsprobleme sind Fragestellungen, die nur mit „ja" oder „nein" zu beantworten sind.

Es gibt einen engen Zusammenhang zwischen Optimierungs- und Entscheidungsproblemen. So könnte man die obige Formulierung dahingehend ändern:

$$\exists ? x : f(x) \preceq g, g \in \mathbb{R}$$

mit
$c_j(x) \leq 0,\ i=1,\ldots,k$
$h_j(x) = 0, j=1,\ldots,m$

Man kann die Algorithmen in Klassen unterteilen, die sich durch grundlegende Abläufe unterscheiden. Einige seien an dieser Stelle aufgeführt.

2.1.3 Greedy-Algorithmen

Algorithmen dieser Klasse treffen Entscheidungen, die jeweils lokal das beste Resultat ergeben. Einmal getroffene Entscheidungen werden nicht zurückgenommen. Daraus folgt, dass derartige Algorithmen im Allgemeinen nicht die global günstigste Lösung finden. Greedy-Algorithmen werden in Optimierungsproblemen eingesetzt. Die Optimalität des Algorithmus muss für jede Anwendung bewiesen werden. Zum Beispiel findet der Kruskal-Algorithmus zur Bestimmung **minimaler Spannbäume** das Optimum. Andererseits sind Greedy-Algorithmen nicht geeignet, das **Rundreiseproblem** zu lösen. Hier werden diese als Heuristiken eingesetzt. Dabei wird zum Beispiel der erhaltene minimale Spannbaum einmal vollständig umfahren, d.h die Rundreise durchläuft jede Kante des Spannbaumes zweimal. Verbindungen zwischen mehrfach besuchten Knoten werden durch Bewertung mittels des euklidischen Abstandes aufgebrochen und durch direkte Verbindungen (welche im Originalgraph, aber nicht im minimalen Spannbaum enthalten sind) ersetzt. Dies geschieht solange, bis die Zulässigkeit des Rundreiseproblems erfüllt ist. Man erhält so einen **Hamilton-Kreis** mit möglichst kleiner Gesamtlänge.

Ein anderer interessanter Fall ist das **Rucksackproblem**. Dabei gibt es eine Menge von Objekten, welche jeweils zwei Eigenschaften besitzen, sagen wir Preis und Nutzen. Zudem ist ein Behälter gegeben, der ein Maximalgewicht aufnehmen

kann. Gesucht ist nun eine Auswahl aus der Objektmenge, sodass der Gesamtnutzen maximal wird, aber das Gesamtgewicht das zulässige Maximalgewicht nicht überschreitet. Kann man die Mengenobjekte stückeln (fraktionieren), so gibt es Greedy-Algorithmen, die das Optimum finden. Ist allerdings nur „nehmen“ (1) oder „nicht-nehmen“ (0) erlaubt, sind die Greedy-Algorithmen nicht optimal!

Um die Optimalität eines Greedy-Algorithmus für das fraktionale Rucksackproblem zu verstehen, genügt es, die Menge der Objekte absteigend nach dem Verhältnis Preis/Gewicht zu sortieren.

Dann werden dieser geordneten Liste nach und nach die Objekte entnommen und in den Behälter getan – das ist das Greedy-Verfahren. Das Objekt, das als Ganzes genommen, das Gesamtgewicht den Grenzwert überschreiten lässt, wird dann so geteilt, dass das Gewicht wieder zulässig ist. Das geht für das 0-1-Rucksackproblem natürlich nicht.

Ein sehr einfaches Beispiel kann dies demonstrieren. Gegeben seien drei Objekte mit folgenden Preisen (p_i) und Gewichten (w_i):

i	1	2	3
p_i	60	100	120
w_i	10	20	30
p_i/w_i	6	5	4

Das maximale Gewicht für den Behälter sei W=50. Gesucht ist eine gültige Belegung des Behälters $\{x_1, x_2, x_3\}$ mit maximalem Preis:

$$x_1p_1 + x_2p_2 + x_3p_3 \rightarrow \text{maximal}$$

$$x_1w_1 + x_2w_2 + x_3w_3 \preceq W$$

Fraktionales Rucksackproblem: $x_i \in Q = \left\{z / n \,\|\, z \in \mathbb{Z} \wedge n \in \mathbb{N}\{0\}\right\}$

Greedy: Der Reihe nach bezüglich (p_i/w_i): $\{x_1 = 1, x_2 = 1, x_3 = 2/3\} \Rightarrow$ Gesamtgewicht = 50, Gesamtnutzen = 240 $\Rightarrow$ Optimum

0-1-Rucksackproblem: $x_i \in \{0,1\}$

Greedy: Der Reihe nach bezüglich (p_i/w_i): $\{x_1 = 1, x_2 = 1, x_3=0\} \Rightarrow$ Gesamtgewicht = 30, Gesamtnutzen = 160 $\Rightarrow$ kein Optimum

Kombinatorik: $\{x_1 = 0, x_2 = 1, x_3 = 1\} \Rightarrow$ Gesamtgewicht = 50, Gesamtnutzen = 220 $\Rightarrow$ Optimum

2.1.4 Iterative und rekursive Algorithmen

Iterative Algorithmen lösen ein Problem durch eine endliche Zahl von Schleifendurchläufen. Dabei ist die Abfolge der elementaren Anweisungen innerhalb einer Schleife identisch, die gesuchte Größe wird pro Durchlauf geändert. Die Lösung liegt dann nach Abarbeitung der Schleife vor.

Iterative Algorithmen können auch ineinander geschachtelt sein. Damit ist klar, dass deren Laufzeit proportional zum Produkt der entsprechenden Zahl der Schleifendurchläufe wächst.

Rekursive Algorithmen beziehen sich vollständig oder teilweise auf sich. Bekannte mathematische Formulierungen sind:

$$T(n) = aT\left(\frac{n}{b}\right) + F(n) \tag{2.1}$$

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_k x_{n-k} + b_k \tag{2.2}$$

$$P_n(x) = a_1(n,x)P_{n-1}(x) + \ldots + a_{n-k}(n,x)P_{n-k}(x) \tag{2.3}$$

Formel 2.1 (mit a≥1, b>1) dient zur Bestimmung der Laufzeit T(n) rekursiver Algorithmen der Größe n. Dabei sind:

- a – die Zahl der Aufteilung des betrachteten Problems in Unterprobleme,
- 1/b – der Verkleinerungsfaktor der Problemgröße,
- F(n) – Kosten, die durch die Division des Problems und die Kombination der Teillösungen entstehen.

In Abhängigkeit vom Laufzeitverhalten von f(n) kann man Abschätzungen für T(n) erhalten, wobei nur bestimmte Klassen von f(n) erlaubt sind. Dies ist Inhalt des sogenannten **Master-Theorems**. Das Verhältnis der anwachsenden Zahl von Operationen (a) zu der sich verringernden Argumentgröße (n/b) bestimmt die Effizienz der Rekursivität und damit die Laufzeit.

Folgende Fälle für f(n) werden durch das Master-Theorem abgedeckt:

f(n)	T(n)
$O\left(n^{\log_b a-\epsilon}\right)$, $\epsilon > \Theta$	$\Theta\left(n^{\log_b a}\right)$
$\Theta\left(n^{\log_b a}\right)$	$\Theta\left(n^{\log_b a}\log_b n\right)$
$\Omega\left(n^{\log_b a+\epsilon}\right)$, $\epsilon > \Theta$, und ist a $f\left(\frac{n}{b}\right) \preceq \alpha f(n)$, $\Theta < \alpha < 1$ und $n \to \infty$	$\Theta\left(f(n)\right)$

Formel 2.2 und Formel 2.3 bestimmen eine Größe aus der Kenntnis von k Vorgängern. Bekannte Gleichungen dieser Art sind die zur Berechnung der Fibonacci-Zahlen (2.2) oder zur Bestimmung spezieller orthogonaler Polynome vom Grade n (2.3).

Rekursive Algorithmen sind meist elegant und kurz. Sofern diese aber nicht explizit gelöst werden können (was vor allem die Fälle (2.2) und (2.3) betrifft), sind sie in ihrer Umsetzung in Computerprogramme im Allgemeinen langsam und speicherintensiv. Der Selbstaufruf der entsprechenden Routinen benötigt jeweils eigene, disjunkte Speicherbereiche, und die Zusammenführung der Teilresultate kostet Zeit. Man kann aber im Prinzip jeden rekursiven Algorithmus in eine iterative Form bringen.

Ein anschauliches Beispiel für die Gegenüberstellung von Iteration und Rekursion kann mit der **Fibonacci-Folge** demonstriert werden:

$$F(0) = 0,\ F(1) = 1,\ F(n) = F(n-1) + F(n-2) \text{ für } n \geq 2$$

Iteration:

Ein Pseudocode für die iterative Formulierung ist:

```
fun fib_it(n)
    if(n=0||n=1) return n
    f1=0
    f2=1
    do(i=2,n)
        k=f1+f2
        f1=f2
        f2=k
    end do
    return k
end fun
```

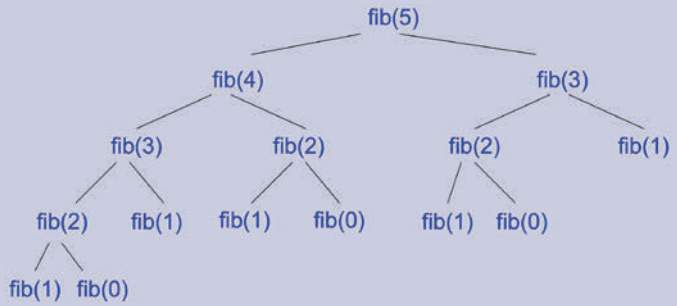
Es ist offensichtlich, dass der Aufwand linear mit der Zahl n wächst.

Rekursion:

Der Pseudocode ist sehr übersichtlich und elegant. Er entspricht genau der „originalen“ Definition.

```
fib(0)=0
fib(1)=1
fib(n)=fib(n-1)+fib(n-2)
```

Es lohnt sich allerdings, einen Blick auf die Abarbeitungsfolge dieses Algorithmus zu werfen. Für n=5 erhält man



Es werden also 3x fib(0), 5x fib(1), 3x fib(2), 2x fib(3), 1x fib(4) und 1x fib(5) aufgerufen und ausgewertet. Allgemein wächst der Aufwand exponentiell mit 2^n.

2.1.5 Branch&Bound-Algorithmen

Branch&Bound-Algorithmen beruhen darauf, auf eine effiziente Weise alle zulässigen Lösungen eines kombinatorischen Optimierungsproblems aufzulisten und mithilfe von bekannten Schranken Lösungen dabei auszuschließen, die als Optimum nicht in Frage kommen können. Insbesondere kann man damit Probleme der diskreten Optimierung sehr gut behandeln. Zur Bestimmung der Schranken können wiederum Greedy-Algorithmen angewendet werden.

Betrachten wir das obige triviale Rucksack-Problem für den 0-1-Fall:

i	1	2	3
p_i	60	100	120
w_i	10	20	30
p_i/w_i	6	5	4

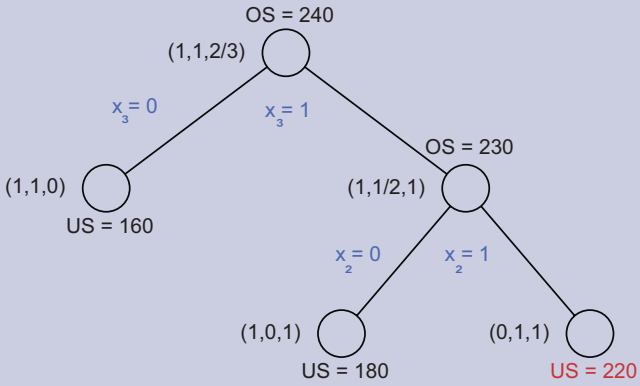
Das maximale Gewicht für den Behälter sei W = 50.

Die obere Schranke wird mit dem Greedy-Algorithmus für das fraktionelle Rucksackproblem bestimmt ($x_1 = 1, x_2 = 1, x_3 = 2/3$) – der Wert ist 240. Nun wird bezüglich der Variable x_3 aufgeteilt, denn diese ist ja nicht gültig: linker Branch: ($x_1, x_2, x_3 = 0$) und rechter Branch : ($x_1, x_2, x_3 = 1$) . Das Verfahren läuft nun rekursiv weiter – in diesem Beispiel jedoch nur sehr kurz. Der linke Branch ist relativ schnell bestimmt ($x_1 = 1, x_2 = 1, x_3 = 0$), es ergibt sich eine untere Schranke von 160. Auch der rechte Branch ist bald beendet. Über die Greedy-Lösung ($x_1 = 1, x_2 = 1/2, x_3 = 1$) (mit Gesamtgewicht 230) gelangt man zu den Verzweigungen ($x_1, x_2 = 0, x_3 = 1$) und ($x_1, x_2 = 1, x_3 = 1$), welche dann zur Lösung ($x_1 = 0, x_2 = 1, x_3 = 1$) führen. Das zugehörige Gesamtgewicht beträgt 220. Die Lösung ist größer als die bisherige Unterschranke, kleiner als die obere Schranke und damit zulässig.

Man kann diese Art von Algorithmen auch als Entscheidungsbäume darstellen. Jeder Knoten dieses Graphen stellt eine Unterteilung des Problems in disjunkte Teilprobleme dar (Branch). Dabei muss diese Unterteilung sicherstellen, dass der gesamte mögliche Lösungsraum abgedeckt wird. Die dadurch entstehenden Zweige werden durch die Berechnung und den Vergleich von unteren und oberen Schranken der Zielfunktion bewertet und gegebenenfalls nicht weiter verfolgt (Bound). Dies geschieht rekursiv. Im schlechtesten Fall müssen alle möglichen Lösungen verfolgt werden, in vielen Fällen werden aber oft ganze Zweige „abgeschnitten“, was zu einer erheblichen Beschleunigung des Verfahrens führen kann. Es gibt unterschiedliche Arten des Traversierens derartiger Entscheidungsbäume:

- **Tiefensuche:** zuerst entlang eines Weges bis zum Ende suchen.
- **Breitensuche:** zuerst die näheren Knoten besuchen.

Das obige Beispiel des 0-1-Rucksacks kann als Baum wie folgt dargestellt werden (OS: obere Schranke, US: untere Schranke).



Es ist offensichtlich, dass der Branch&Bound-Algorithmus alle zulässigen Lösungen findet.

2.1.6 Dynamische Programmierung

Dieses Verfahren überführt mehrstufige Entscheidungsprozesse in eine rekursive Form. Es werden parallel stufenweise Teillösungen gebildet, die ausgeschieden werden, wenn sie nicht eindeutig zu einer besseren Lösung führen als eine bereits vorhandene Teillösung. Einmal berechnete Teillösungen werden gespeichert und können bei Bedarf wiederverwendet werden. Dynamische Programmierung wird bei Optimierungsproblemen eingesetzt.