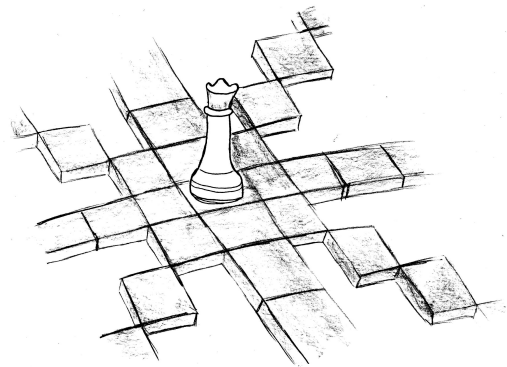


Weitere Algorithmenentwurfstechniken

1 Backtracking

Vier Damen

Beim Schach bedroht die Figur der Dame alle Felder, die sich horizontal, vertikal oder diagonal von der entsprechenden Position der Dame befinden. Im Gegensatz zum normalen Schach betrachten wir hier ein Schachbrett der Grösse 4×4 , auf dem sich mehr als zwei Damen befinden dürfen. Unser Ziel ist es, möglichst viele Damen so zu positionieren, dass keine Dame von einer anderen bedroht wird.



Fassen wir das Schachbrett als (4×4) -Matrix auf, dann können keine zwei Damen in der gleichen Zeile oder Spalte platziert werden, da sie sich ansonsten gegenseitig bedrohten. Also können sich maximal vier Damen auf dem Brett befinden. Wir möchten nun entscheiden, ob eine Platzierung mit genau vier Damen existiert. Wir beschreiben nun, wie eine solche systematisch gefunden werden kann.

Falls eine Platzierung mit vier Damen existiert, dann kann sie durch einen Vektor der Länge 4 repräsentiert werden. Dabei enthält die i -te Komponente die Zeilennummer der Dame, die in der i -ten Spalte positioniert wird. Da der Lösungsvektor eine endliche Länge hat und pro Position nur endlich viele Alternativen erlaubt sind, können wir die Lösungen *systematisch* durchlaufen, um eine Platzierung mit möglichst vielen Damen zu finden. Dabei wird nun *nicht* einfach über alle möglichen Lösungen iteriert. Stattdessen versuchen wir, eine aktuelle Teillösung so zu erweitern, dass die neu entstandene Teillösung noch immer in sich konsistent ist. Kann eine Lösung nicht erweitert werden, dann gehen wir zurück und probieren die alternativen Erweiterungen (sofern existent, ansonsten gehen wir noch weiter zurück). Für das genannte Prinzip des *Backtracking* erhalten wir nun das folgende (sehr grobe) Algorithmengerüst.

BACKTRACK(Teillösung)

- 1 **if** Teillösung ist Lösung für das gesamte Problem **then** Melde Erfolg und STOP
 - 2 **else for each** legale Erweiterung der Teillösung **do**
 - 3 BACKTRACK(erweiterte Lösung)
 - 4 Melde Misserfolg
-

BACKTRACKING

Backtracking kann durch einen *Ablaufbaum* wie folgt visualisiert werden. An jedem Knoten v wird eine Teillösung S gespeichert, und für jede legale Erweiterung S' von S hat v einen Nachfolgerknoten, der S' speichert. Knoten mit Teillösungen ohne legale Erweiterung sind also Blätter. Die Wurzel speichert die (leere) Startlösung.

ABLAUFBAUM

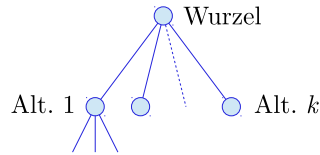


Abb. 4.1: Schematischer Ablauf von Backtracking.

ALGORITHMUS Für das geschilderte Problem verfahren wir nun wie folgt. Wir platzieren initial eine Dame auf der Position (1, 1) (siehe Abb. 4.2a). Für die Dame in der zweiten Spalte bleiben nur die Zeilen 3 und 4 übrig. Wir platzieren zunächst eine Dame auf der Position (3, 2) (siehe Abb. 4.2b). Nun kann aber offenbar keine Dame mehr in der dritten Spalte platziert werden. Folglich nehmen wir die eben vorgenommene Platzierung einer Dame auf (3, 2) zurück und platzieren stattdessen eine Dame auf (4, 2) (siehe Abb. 4.2c). Die einzig mögliche Position einer Dame in der dritten Spalte ist nun (2, 3) (siehe Abb. 4.2d). Auf diese Art fahren wir fort. Sobald eine Lösung nicht mehr erweitert werden kann, machen wir den oder die vorigen Schritt(e) rückgängig. Schliesslich erhalten wir die in Abbildung 4.2e) dargestellte Positionierung mit vier Damen. $\square$

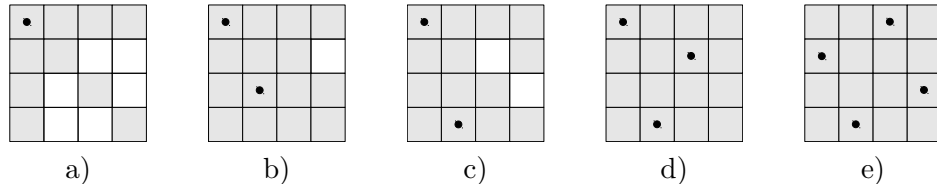


Abb. 4.2: Einige Zwischenschritte beim Backtracking. Die schwarzen Punkte geben die jeweiligen Positionen der Damen an. Grau markierte Felder werden von einer Dame bedroht.

ABLAUFBAUM Es ergibt sich der folgende Ablaufbaum. Neben jedem Knoten ist die aktuelle Teillösung gespeichert. “-” bedeutet dabei, dass die entsprechende Komponente des Vektors noch nicht spezifiziert wurde. Die Zahl in jedem Knoten gibt an, wann der entsprechende Knoten besucht wird.

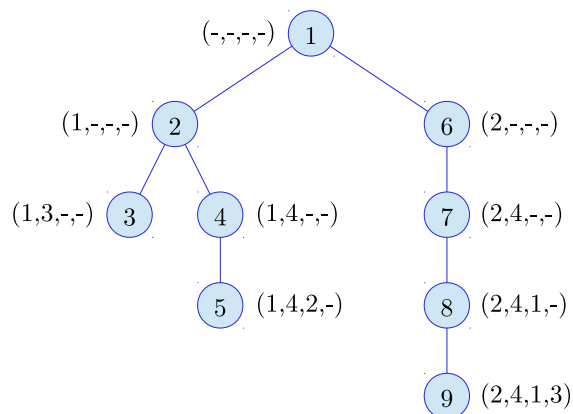


Abb. 4.3: Ablaufbaum für das Vier-Damen-Problem.

LAUFZEIT Hat der Lösungsvektor die Länge n und gibt es pro Komponente k Alternativen, dann werden im schlimmsten Fall ungefähr k^n viele Lösungen ausprobiert, was nicht besser als einfaches Aufzählen aller möglichen Lösungen ist. Backtracking ist also eine Heuristik, die keine Garantie für eine schnelle Laufzeit liefert! Andererseits

erlaubt sie uns, schon in sich unzulässige Teillösungen zu erkennen und nicht weiterzuverfolgen. Wir werden im Folgenden beschreiben, wie man diese Technik so ausbauen kann, dass vielversprechende Lösungen bevorzugt untersucht werden. Auf diese Weise erhoffen wir uns einen Effizienzgewinn in praktischen Anwendungen.

Erfüllbarkeitsproblem der Aussagenlogik

Seien $x_1, \dots, x_n$ boolesche Variablen, d.h., sie sind entweder wahr oder falsch. Wir betrachten eine Formel φ in *konjunktiver Normalform* (KNF). Dabei ist $\varphi = \bigwedge_i C_i$ mit den *Klauseln* $C_i = \bigvee_j x_{ij}$, $x_{ij} \in \{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$. Ein Beispiel ist die Formel

KONJUNKTIVE
NORMALFORM

$$\varphi \equiv (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_4}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4) \quad (1)$$

mit den sechs Klauseln $(x_1 \vee x_2)$, $(\overline{x_1} \vee \overline{x_2})$, $(\overline{x_1} \vee \overline{x_3})$, $(x_1 \vee \overline{x_4})$, $(x_2 \vee x_3 \vee \overline{x_4})$ sowie $(x_1 \vee \overline{x_2} \vee x_4)$. Das *Erfüllbarkeitsproblem der Aussagenlogik* (kurz SAT, engl. *Satisfiability*) besteht nun darin zu entscheiden, ob φ erfüllbar ist, d.h. ob eine Zuweisung $(x_1, \dots, x_n) \in \{\text{wahr}, \text{falsch}\}^n$ existiert, sodass φ wahr ist.

PROBLEM

Eine Lösung kann durch einen Vektor der Länge n repräsentiert werden, dessen Einträge den Wert F (falsch) oder T (wahr) annehmen können. Damit gibt es 2^n mögliche Belegungen. Das Backtracking wird nun wie folgt realisiert. Jedesmal, wenn wir eine Variable x_i setzen, prüfen wir zunächst, wieviele Klauseln erfüllt werden, wenn x_i auf wahr gesetzt wird, und wieviele erfüllt werden, wenn x_i auf falsch gesetzt wird. Wir wählen dann zunächst die Belegung, die mehr Klauseln erfüllt. Sobald wir feststellen, dass durch eine Teilbelegung einige Klauseln nicht mehr erfüllbar sind, nehmen wir die aktuelle Wahl zurück, und versuchen die Alternative. Führt dies ebenfalls zu einer Nicht-Erfüllbarkeit, dann gehen wir einen weiteren Schritt zurück. Wir führen für jeden Schritt die Formel mit, die durch Einsetzung der Belegung übrig bleibt.

ALGORITHMUS

Backtracking betrachtet zuerst die Variable x_1 . Für das obige Beispiel bleiben je nach Wahl von x_1 die folgenden Formeln übrig:

BEISPIEL

$$\varphi|_{x_1=T} \equiv (T) \wedge (\overline{x_2}) \wedge (\overline{x_3}) \wedge (T) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (T), \quad (2)$$

$$\varphi|_{x_1=F} \equiv (x_2) \wedge (T) \wedge (T) \wedge (\overline{x_4}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee x_4). \quad (3)$$

Da die Wahl von $x_1 = T$ zu mehr erfüllten Klauseln führt (nämlich 3 anstatt 2), verfolgen wir zunächst diese Lösung weiter. Wir stellen fest, dass die zusätzliche Wahl von $x_2 = T$ dazu führt, dass die Klausel $(\overline{x_1} \vee \overline{x_2})$ nicht erfüllt wird. Folglich setzen wir $x_2 = F$ und fahren mit der Variablen x_3 fort. Auch dort führt die Wahl von $x_3 = T$ dazu, dass eine Klausel nicht erfüllt ist, nämlich $(\overline{x_1} \vee \overline{x_3})$. Also setzen wir $x_3 = F$ und erhalten die Formel

$$\varphi|_{x_1=T, x_2=F, x_3=F} \equiv (T) \wedge (T) \wedge (T) \wedge (T) \wedge (\overline{x_4}) \wedge (T). \quad (4)$$

Nun stellen wir fest, dass bei der Wahl von $x_4 = T$ die Klausel $(\overline{x_4})$ nicht erfüllt ist, doch bei der Wahl von $x_4 = F$ werden alle sechs Klauseln erfüllt. Somit ist $(x_1, x_2, x_3, x_4) = (T, F, F, F)$ eine erfüllende Belegung für φ . Wir erhalten den in Abbildung 4.4 dargestellten Ablaufbaum. $\square$

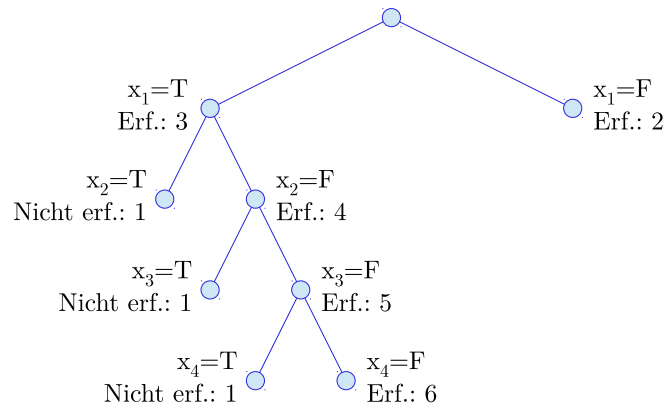


Abb. 4.4: Ablaufbaum für das Erfüllbarkeitsproblem der Aussagenlogik. Neben jedem Knoten ist die Wahl der aktuell betrachteten Variablen vermerkt. Ausserdem findet sich dort die Anzahl der erfüllten bzw. nicht erfüllten Klauseln. Ist die Anzahl der nicht erfüllten Klauseln nicht vermerkt, dann hat die aktuelle Teillösung keine unerfüllten Klauseln.

2 Branch-and-Bound

BRANCH-AND-
BOUND

OBERE
SCHRANKE

UNTERE
SCHRANKE

MODULE

Wir haben im vorigen Abschnitt eine Technik kennengelernt, die es erlaubt, aussichtslose Teillösungen frühzeitig zu verwerfen. Allerdings haben wir sie nur für Entscheidungsprobleme angewendet, bei denen am Ende die Antwort Ja oder Nein steht (“Gibt es eine Platzierung mit vier Damen?”, “Gibt es eine erfüllende Belegung?”). Für Optimierungsprobleme ist *Branch-and-Bound* eine ähnliche Vorgehensweise, allerdings müssen wir überlegen, wann eine Teillösung nicht weiter verfolgt werden soll. O.B.d.A. betrachten wir Maximierungsprobleme, sofern nicht anders genannt. Wie vorher speichert jeder Knoten des Ablaufbaums eine Teillösung. Zusätzlich wird in jedem Knoten eine *obere Schranke* gespeichert. Diese gibt an, welchen Wert eine gültige (ggf. mehrfache) Erweiterung der an diesem Knoten repräsentierten Teillösung maximal erreichen kann. Für den Ablaufbaum verwalten wir ausserdem noch eine *untere Schranke*. Diese ist das Maximum der bereits gefundenen Lösungen und einer globalen unteren Schranke (sofern existent). Stellen wir an einem Knoten fest, dass die obere Schranke kleiner gleich der unteren Schranke ist, dann muss die an diesem Knoten repräsentierte Teillösung nicht weiter verfolgt werden (denn keine Erweiterung kann einen besseren Wert als die derzeit beste Lösung erreichen). Um Branch-and-Bound anzuwenden, müssen zusätzlich drei Module spezifiziert werden:

- 1) *Branch*: Entscheide, wie eine gegebene Teillösung zu erweitern ist.
- 2) *Search*: Welche aktuelle Teillösung soll als nächstes betrachtet und ggf. erweitert werden?
- 3) *Learn*: Was folgt aus den bisherigen Erkenntnissen für erzwungene Teile der Lösung?

PRINZIPIELLES
VORGEHEN

Im Allgemeinen wählt das Search-Modul im Ablaufbaum ein Blatt v mit der höchsten oberen Schranke aus. Ist der Wert der dort gespeicherten Teillösung s_v grösser als die bisherige untere Schranke, so wird diese entsprechend aktualisiert. Ist s_v eine vollständige Lösung, deren Wert mit der unteren Schranke zusammenfällt, dann wird s_v als aktuelles Optimum abgespeichert. Das Branch-Modul versucht nun, den

Knoten v zu erweitern. Das gesamte Verfahren terminiert, sobald die obere Schranke *jedes* Blatts maximal so gross wie die untere Schranke ist (zur Erinnerung: Die untere Schranke gibt die beste bisher erreichte Lösung an). Haben nun alle Blätter eine obere Schranke, die maximal so gross wie die untere Schranke ist, dann kann keine Teillösung so erweitert werden, dass eine bessere als die bisher beste gefundene Lösung entsteht. Die Benutzung des Learn-Moduls ist nicht zwingend, kann aber zu einer besseren Laufzeit führen, da u.U. wesentlich weniger (Teil)lösungen betrachtet werden müssen. Wir erhalten das folgende (grobe) Algorithmengerüst:

BRANCHANDBOUND(Problem P)		BRANCH-AND-BOUND
1	Berechne globale untere Schranke, falls existent.	
2	while globales Optimum nicht gefunden do	
3	if obere Schranke in jedem Blatt $\leq$ untere Schranke then	
4	globales Optimum gefunden. STOP	
5	Wähle Blatt v mit maximaler oberer Schranke	
6	if v hat keine gültige Erweiterung then	
7	Markiere v als nicht erweiterbar	
8	for each gültige Erweiterung der an v gespeicherten Teillösung do	
9	Erzeuge einen Nachfolgerknoten w von v , der die entsprechend erweiterte Teillösung s_w speichert	
10	Schliesse auf erzwungene Teile von s_w	
11	Berechne obere Schranke von s_w	
12	Aktualisiere untere Schranke, falls nötig	

Wir werden diese Technik nun auf einige Optimierungsprobleme anwenden.

Maximale Anzahl erfüllter Klauseln

Wir betrachten die Optimierungsvariante des im vorigen Abschnitt vorgestellten Erfüllbarkeitsproblem der Aussagenlogik. Dabei sei erneut eine Formel in konjunktiver Normalform (KNF) gegeben, zum Beispiel

$$\varphi \equiv (\overline{x_1}) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_4}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4). \quad (5)$$

Man überprüft leicht, dass φ nicht erfüllbar ist. Wir wollen nun zu einer gegebenen Formel φ in KNF eine Belegung finden, die möglichst viele Klauseln erfüllt. Für die obige Formel sind dies genau die Belegungen (x_1, x_2, x_3, x_4) aus

$$\{(T, F, F, F), (F, T, F, F), (F, T, F, T), (F, T, T, F), (F, T, T, T), (F, F, T, F), (F, F, F, F)\}, \quad (6)$$

denn sie erfüllen alle bis auf eine Klausel.

Ist eine Formel φ mit den Variablen $x_1, \dots, x_n$ gegeben, dann kann wie vorher eine Lösung durch einen Vektor $(x_1, \dots, x_n) \in \{T, F\}^n$ repräsentiert werden. Eine Teillösung, die einigen x_i feste Werte zuweist, teilt die Klauseln von φ in genau drei (möglicherweise leere) Gruppen ein: Definitiv *erfüllte* Klauseln, definitiv *unerfüllte*

Klauseln und *unentschiedene* Klauseln. Betrachten wir etwa für das obige Beispiel die Teillösung $x_1 = T$, dann sind die Klauseln $(x_1 \vee x_2)$, $(x_1 \vee \overline{x_4})$ und $(x_1 \vee \overline{x_2} \vee x_4)$ definitiv erfüllt. Die Klausel $(\overline{x_1})$ ist definitiv nicht erfüllt, und alle übrigen Klauseln sind unentschieden.

GLOBALE
 UNTERE
 SCHRANKE

Um Branch-and-Bound für dieses Problem anzuwenden, überlegen wir zunächst, dass in jeder KNF-Formel mindestens $\lceil n/2 \rceil$ Klauseln erfüllt werden können.

Lemma 1. *Sei φ eine Formel in KNF mit den Variablen $x_1, \dots, x_n$. Es gibt eine Belegung, die mindestens $\lceil n/2 \rceil$ Klauseln erfüllt.*

Beweis. Eine Klausel, in der keine Variable sowohl negiert als auch nicht-negiert vorkommt, wird lediglich durch eine einzige Variablenbelegung der in der Klausel vorkommenden Variablen nicht erfüllt (nämlich die, in der alle negierten Variablen auf falsch und alle anderen Variablen auf wahr gesetzt werden). Wird nun in dieser Belegung der Wert jeder Variablen invertiert, dann ist die Klausel erfüllt. Seien nun $B_T = (T, \dots, T)$ und $B_F = (F, \dots, F)$ die zwei Belegungen, die jeder Variablen den Wert T bzw. F zuweisen. Jede Klausel von φ wird entweder durch B_T oder durch B_F oder durch beide erfüllt. Also erfüllt entweder B_T mindestens die Hälfte aller Klauseln, oder aber B_F (würden beide weniger als die Hälfte aller Klauseln erfüllen, dann gäbe es mindestens eine Klausel, die weder von B_T , noch von B_F erfüllt würde, was nicht möglich ist). ■

Wir beschreiben nun die einzelnen Komponenten der Lösungsstrategie.

ALGORITHMUS **Teillösungen:** Wie im vorher diskutierten Entscheidungsproblem kann eine Belegung für eine Formel φ durch einen Vektor aus $\{T, F\}^n$ repräsentiert werden. Jeder Knoten v des Ablaufbaums speichert eine Teillösung $s_v \in \{T, F\}^k$, die den Variablen $x_1, \dots, x_k$ einen Wert zuweist (in der Wurzel ist $k = 0$, und keiner Variablen wird ein Wert zugewiesen). Als Wert der Teillösung s_v definieren wir die Anzahl der von s_v definitiv erfüllten Klauseln.

Obere Schranke: Sei v ein Knoten des Ablaufbaums, der eine Teillösung s_v speichert. Die in v gespeicherte obere Schranke ist die Summe der Anzahl von s_v definitiv erfüllter Klauseln und der Anzahl noch unerfüllter Klauseln.

Untere Schranke: Eine untere Schranke ergibt sich aus dem Maximum der Werte aller bereits gefundenen Teillösungen sowie der globalen unteren Schranke $\lceil n/2 \rceil$.

Branch: Sei v ein Knoten des Ablaufbaums und s_v die in v gespeicherte Teillösung, die den Variablen $x_1, \dots, x_k$ einen Wert zuweist. Ist $k = n$, dann kann s_v nicht erweitert werden. Ansonsten erzeugen wir für v zwei Nachfolgerknoten mit den um $x_{k+1} = T$ bzw. $x_{k+1} = F$ erweiterten Teillösungen.

Search: Als nächstes wählen wir ein Blatt mit einer maximalen oberen Schranke (nicht mit dem maximalen Wert!).

Learn: Für dieses Problem verzichten wir auf den Einsatz des Learn-Moduls.

BEISPIEL Für die Formel (5) von vorhin kann Branch-and-Bound wie folgt angewendet werden. Zunächst wird die untere Schranke auf 4 gesetzt (wegen Lemma 1). Von der

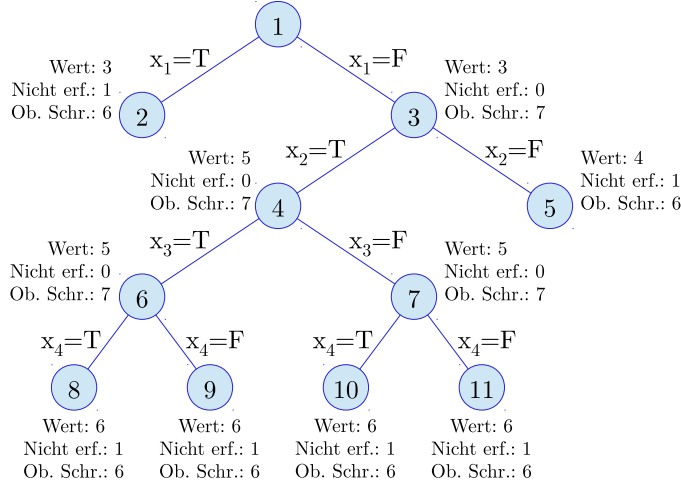


Abb. 4.5: Ablaufbaum für Branch-and-Bound zur Berechnung einer maximalen Menge erfüllbarer Klauseln. Neben jedem Knoten ist der Wert der dort gespeicherten Teillösungen, die Anzahl nicht erfüllter Klauseln sowie die obere Schranke angegeben. Die Zahl in jedem Knoten gibt an, wann der entsprechende Knoten erzeugt wird.

leeren Lösung (Knoten 1 in Abbildung 4.5) ausgehend werden dann die entstehenden Teillösungen berechnet, wenn $x_1 = T$ bzw. $x_1 = F$ gesetzt werden. Dadurch ergeben sich

$$\varphi|_{x_1=T} \equiv (F) \wedge (T) \wedge (\overline{x_2}) \wedge (\overline{x_3}) \wedge (T) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (T), \quad (7)$$

$$\varphi|_{x_1=F} \equiv (T) \wedge (x_2) \wedge (T) \wedge (T) \wedge (\overline{x_4}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee x_4). \quad (8)$$

Diese Teillösungen werden in den Knoten 2 und 3 gespeichert. Offenbar sind in $\varphi|_{x_1=T}$ genau drei Klauseln erfüllt, und eine ist nicht erfüllt. Somit haben alle Erweiterungen dieser Teillösung maximal Wert 6. In $\varphi|_{x_1=F}$ sind ebenfalls drei Klauseln erfüllt, aber keine Klausel ist nicht erfüllt. Also ist die obere Schranke dort 7 und wir fahren mit dieser Teillösung fort. Nun setzen wir zusätzlich $x_2 = T$ bzw. $x_2 = F$ und erhalten

$$\varphi|_{x_1=F, x_2=T} \equiv (T) \wedge (T) \wedge (T) \wedge (T) \wedge (\overline{x_4}) \wedge (T) \wedge (x_4), \quad (9)$$

$$\varphi|_{x_1=F, x_2=F} \equiv (T) \wedge (F) \wedge (T) \wedge (T) \wedge (\overline{x_4}) \wedge (x_3 \vee \overline{x_4}) \wedge (T). \quad (10)$$

In $\varphi|_{x_1=F, x_2=T}$ (Knoten 4) sind fünf Klauseln erfüllt und keine nicht erfüllt, und die obere Schranke ist 7. In $\varphi|_{x_1=F, x_2=F}$ (Knoten 5) dagegen sind nur vier Klauseln erfüllt und mindestens eine ist nicht erfüllt, also ist die obere Schranke lediglich 6. Da die erstgenannte Lösung mindestens fünf Klauseln erfüllt, aktualisieren wir die untere Schranke auf 5 und erweitern den Knoten 4. Auf diese Weise fahren wir fort und entdecken schliesslich den Knoten 8, der eine Lösung mit Wert 6 enthält. Also wird die untere Schranke auf 6 aktualisiert. Die Lösungen in den Knoten 8 bis 11 können nicht erweitert werden. Sobald die Lösung in Knoten 11 entdeckt wurde, ist in allen Blättern des Ablaufbaums (d.h., in den Knoten 2, 5, 8–11) eine obere Schranke gespeichert, die nicht grösser als die untere Schranke ist. Das Verfahren terminiert und liefert die in Knoten 8 gespeicherte Lösung (F, T, T, T) zurück. $\square$

Das Rucksackproblem

Gegeben seien n Objekte $\{1, \dots, n\}$ mit den Nutzwerten $p_i \in \mathbb{N}$ und den Gewichten $g_i \in \mathbb{N}$. Sei weiterhin eine Gewichtsschranke $G \in \mathbb{N}$ gegeben. Eine Auswahl von Objekten $I \subseteq \{1, \dots, n\}$ heisst *Bepackung*. Eine Bepackung heisst *gültig*, wenn das Gesamtgewicht der Objekte in I nicht grösser als G ist, d.h., wenn $\sum_{i \in I} g_i \leq G$ gilt. Der Wert einer Bepackung I ist die Summe der Nutzwerte aller benutzten Objekte, also $\sum_{i \in I} p_i$. Das *Rucksackproblem* besteht nun darin, unter allen möglichen gültigen Bepackungen diejenige mit maximalem Wert zu finden. Wir nehmen der Einfachheit halber an, dass jedes Objekt i ein Gewicht kleiner oder gleich G hat (ansonsten muss es nicht betrachtet werden, da es in keiner gültigen Bepackung vorkommen kann).

Um Branch-and-Bound für das Rucksackproblem anzuwenden, sortieren wir zunächst die Objekte absteigend nach dem Quotienten p_i/g_i , der den Nutzen pro Gewichtseinheit angibt. Je höher diese Zahl ist, desto eher würden wir vermuten, dass das entsprechende Objekt in einer optimalen Lösung vorkommt. Seien also ab sofort die Objekte so sortiert, dass $p_1/g_1 \geq p_2/g_2 \geq \dots \geq p_n/g_n$ gilt.

ALGORITHMUS **Teillösungen:** Jeder Knoten v des Ablaufbaums speichert eine gültige Teillösung $s_v = (I_v, E_v, U_v)$ mit $I_v \cup E_v \cup U_v = \{1, \dots, n\}$. Dabei sind I_v die definitiv benutzten Objekte, E_v die definitiv nicht benutzten Objekte und U_v alle Objekte, für die noch nicht entschieden ist, ob sie benutzt werden sollen oder nicht. In der Wurzel ist $U_v = \{1, \dots, n\}$. Einer Teillösung $s_v = (I_v, E_v, U_v)$ wird der Wert $w(s_v) := \sum_{i \in I_v} p_i$ zugewiesen, also die Summe der Nutzwerte der definitiv benutzten Objekte. Wir definieren das Restgewicht einer Lösung s_v als $R_v = G - \sum_{i \in I_v} g_i$. Da I_v eine *gültige* Bepackung ist, ist $R_v \geq 0$. Schliesslich fordern wir noch, dass jedes Objekt aus U_v (den derzeit unentschiedenen Objekten) höchstens Gewicht R_v hat. Wir werden später sehen, wie wir diese Anforderung aufrechterhalten können.

Obere Schranke: Sei $s_v = (I_v, E_v, U_v)$ die in einem Knoten v des Ablaufbaums gespeicherte Lösung. Ist $U_v = \emptyset$, dann ist die Teillösung vollständig und die obere Schranke ist gleich dem Wert der Bepackung. Sei ansonsten $k \in U_v$ das Element mit maximalem Nutzen pro Gewichtseinheit p_k/g_k . Da alle Objekte aus I_v bereits benutzt wurden, verbleibt ein Restgewicht von R_v . Da s_v nur durch Objekte aus U_v erweitert werden kann und diese pro Gewichtseinheit einen maximalen Nutzen von p_k/g_k besitzen, hat jede legale Erweiterung von s_v höchstens einen Nutzen von $(\sum_{i \in I_v} p_i) + R_v \cdot p_k/g_k = w(s_v) + R_v \cdot p_k/g_k$.

Untere Schranke: Eine untere Schranke ergibt sich aus dem Maximum der Werte aller bereits gefundenen Teillösungen.

Branch: Sei v ein Knoten des Ablaufbaums, der die Teillösung $s_v = (I_v, E_v, U_v)$ speichert. Ist $U_v = \emptyset$, dann ist s_v nicht erweiterbar. Ansonsten wählen wir aus U_v das Objekt k aus, das den grössten Quotienten p_k/g_k hat und erzeugen für v zwei Nachfolgerknoten w_1 und w_2 mit den Lösungen $s_{w_1} = (I_{w_1}, E_{w_1}, U_{w_1})$ bzw. $s_{w_2} = (I_{w_2}, E_{w_2}, U_{w_2})$ mit

$$I_{w_1} = I_v \cup \{k\}, E_{w_1} = E_v, U_{w_1} = U_v \setminus \{k\}, \quad (11)$$

$$I_{w_2} = I_v, E_{w_2} = E_v \cup \{k\}, U_{w_2} = U_v \setminus \{k\}. \quad (12)$$

In der in w_1 gespeicherten Lösung wird das Objekt k also benutzt, in der in w_2 gespeicherten Lösung dagegen nicht. Da v nach Voraussetzung eine Lösung

(I_v, E_v, U_v) mit $\max_{i \in U_v} g_i \leq (G - \sum_{i \in I_v} g_i)$ speichert, ist $\sum_{i \in I_{w_1}} g_i \leq G$, d.h. w_1 speichert eine noch immer gültige Bepackung. Wegen $I_{w_2} = I_v$ ist die in w_2 gespeicherte Bepackung ebenfalls gültig.

Search: Als nächstes wird ein Blatt mit einer maximalen oberen Schranke betrachtet.

Learn: Jedesmal, wenn ein neuer Knoten v mit der Lösung $s_v = (I_v, E_v, U_v)$ erzeugt wird, berechnen wir das verbleibende Gewicht $R_v = G - \sum_{i \in I_v} g_i$. Alle Objekte aus U_v , die ein grösseres Gewicht als R_v haben, können offenbar nicht mehr benutzt werden, da ansonsten die Gewichtsschranke überschritten würde. Also entfernen wir diese Objekte aus U_v und fügen sie stattdessen in E_v , also die Menge der definitiv nicht benutzten Objekte, ein.

Betrachten wir etwa vier Objekte mit den Nutzwerten $p_1 = 36, p_2 = 56, p_3 = 20, p_4 = 12$ sowie den Gewichten $g_1 = 4, g_2 = 7, g_3 = 4, g_4 = 3$. Die Gewichtsschranke sei $G = 10$. Wir berechnen die Quotienten p_i/g_i und erhalten

BEISPIEL

p_i	36	56	20	12
g_i	4	7	4	3
p_i/g_i	9	8	5	4

Offenbar gilt $p_1/g_1 \geq p_2/g_2 \geq p_3/g_3 \geq p_4/g_4$. Der Algorithmus setzt initial die untere Schranke auf 0 und betrachtet danach die leere Teillösung $I_1 = \emptyset, E_1 = \emptyset, U_1 = \{1, 2, 3, 4\}, R_1 = 10$ (Knoten 1 in Abbildung 4.6). Sie hat den Wert 0, und die obere Schranke ist $0 + 10 \cdot 9 = 90$ (aktueller Wert 0 plus Restgewicht 10 multipliziert mit dem maximalen Nutzen pro Gewichtseinheit 9). Wir nutzen nun Objekt 1 zum Verzweigen und erzeugen die Knoten 2 und 3. Wird Objekt 1 benutzt, dann erhalten wir die Teillösung $I_2 = \{1\}, E_2 = \{2\}, U_2 = \{3, 4\}, R_2 = 6$ (Hinweis: E_2 enthält das Objekt 2, weil das Learn-Modul erkennt, dass es wegen des Restgewichts 6 nicht mehr benutzt werden kann). Die Lösung hat Wert 36, und weil dies grösser

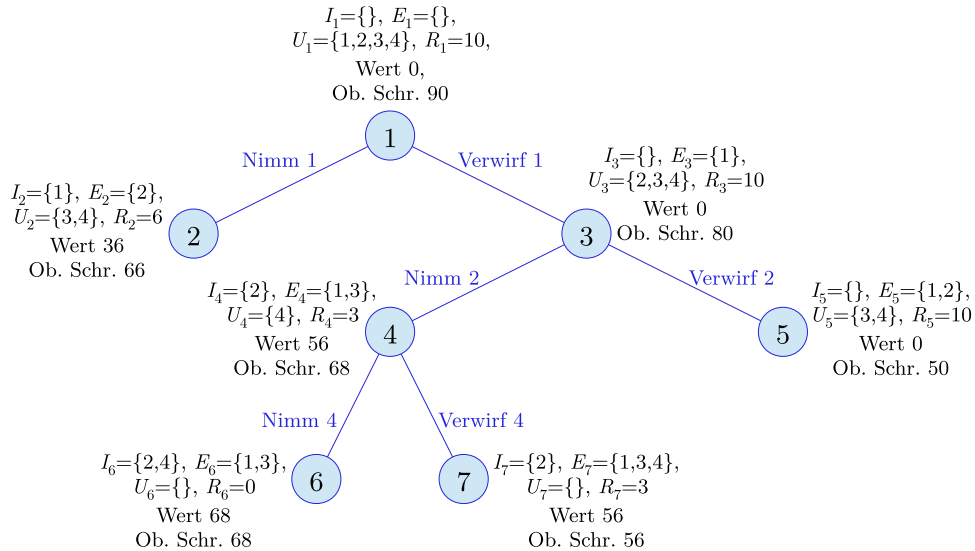


Abb. 4.6: Ablaufbaum zur Beispielinstantz, wenn Branch-and-Bound auf das Rucksackproblem angewendet wird. Die Zahl in jedem Knoten gibt an, wann der entsprechende Knoten erzeugt wird.

als die bisherige untere Schranke ist, wird diese ebenfalls auf 36 gesetzt. Als obere Schranke ergibt sich $36 + 6 \cdot 5 = 66$. Wird Objekt 1 dagegen nicht benutzt, dann ergeben sich $I_3 = \emptyset$, $E_3 = \{1\}$, $U_3 = \{2, 3, 4\}$, $R_3 = 10$. Die Lösung hat Wert 0, also muss die untere Schranke nicht aktualisiert werden. Die obere Schranke ist $0 + 10 \cdot 8 = 80$. Nun muss ein neues Blatt ausgewählt werden, dessen Teillösung erweitert wird (d.h., einer der Knoten 2 oder 3). Wir wählen Knoten 3, da dort eine grössere obere Schranke gespeichert ist. Auf diese Weise fährt der Algorithmus fort, bis der Knoten 7 erzeugt wurde. Danach speichert jedes Blatt eine obere Schranke, die kleiner gleich der unteren Schranke ist. Also terminiert das Verfahren und liefert $I_6 = \{2, 4\}$ zurück. $\square$

VERBESSERUNG DER SCHRANKEN

Verbesserung der Schranken Die Abschätzungen für die oberen und die unteren Schätzungen sind zwar korrekt, allerdings noch ziemlich grob. Wir diskutieren im Folgenden, wie mit geringem Aufwand bessere Abschätzungen erzielt werden können. Dazu betrachten wir das Rucksackproblem unter zwei neuen Blickwinkeln.

GREEDY- STRATEGIE

Bei der ersten Betrachtung des Rucksackproblems mag die folgende *Greedy-Strategie* zur Lösung verlockend klingen. Die Objekte werden erneut nach p_i/g_i , also dem Nutzen pro Gewichtseinheit, absteigend sortiert. Je grösser dieser Quotient ist, desto besser scheint die Wahl des entsprechenden Objekts zu sein. Seien also $p_1/g_1 \geq \dots \geq p_n/g_n$. Wir setzen das Restgewicht auf G und betrachten die Objekte $i = 1, \dots, n$ in genau dieser Reihenfolge. Für das i -te Objekt prüfen wir, ob es noch benutzt werden kann (d.h., ob sein Gewicht noch unterhalb des Restgewichts liegt). Falls ja, dann nehmen wir es in die Auswahl der benutzen Elemente auf und verringern das Restgewicht um g_i .

GREEDY- ALGORITHMUS

GREEDYKNAPSACK($p_1, \dots, p_n, g_1, \dots, g_n, G$)

```

1 Sortiere die Objekte absteigend nach  $p_i/g_i$ 
2  $G_R \leftarrow G$ ;  $I \leftarrow \emptyset$ 
3 for  $i \leftarrow 1$  to  $n$  do
4     if  $g_i \leq G_R$  then  $I \leftarrow I \cup \{i\}$ ;  $G_R \leftarrow G_R - g_i$ 
5 return  $I$ 
```

Da ein Objekt nur dann eingepackt wird, wenn sein Gewicht unter dem Restgewicht liegt und dieses initial auf G gesetzt wird, liefert der Greedy-Algorithmus also offenbar eine gültige Bepackung zurück. Leider kann ihr Wert beliebig schlecht sein:

Theorem 2. Sei $B \in \mathbb{N}$ beliebig. Es gibt eine Menge von n Objekten mit den Nutzwerten $p_1, \dots, p_n \in \mathbb{N}$, den Gewichten $g_1, \dots, g_n \in \mathbb{N}$ und einer Gewichtsschranke $G \in \mathbb{N}$, für die der Greedy-Algorithmus eine Lösung berechnet, deren Wert nur $1/B$ der optimalen Lösung für diese Eingabe beträgt. $\blacksquare$

Der Greedy-Algorithmus kann also eine sehr schlechte Lösung zurückliefern, tut dies aber nicht immer. Wir werden nun aber sehen, dass eine Greedy-Strategie für die im Folgenden vorgestellte Variante des Rucksackproblems grundsätzlich zu einer optimalen Lösung führt.

FRAKTIONALES RUCKSACKPRO- BLEM

Bisher haben wir Objekte entweder eingepackt oder nicht. Das *fraktionale Rucksackproblem* erlaubt es dagegen, Objekte auch nur teilweise einzupacken. Dabei wird angenommen, dass sich das Gewicht und der Nutzen eines Objekts proportional zum angepackten Anteil verhalten. Eine Lösung kann durch einen Vektor $\lambda \in [0, 1]^n$

beschrieben werden, wobei λ_i angibt, welcher Anteil des Objekts i benutzt werden soll (bei $\lambda_i = 0$ wird i überhaupt nicht benutzt, bei $\lambda_i = 1$ vollständig). Das Gewicht von i beträgt entsprechend $\lambda_i g_i$, der Nutzen $\lambda_i p_i$. Das Ziel beim fraktionierten Rucksackproblem besteht also darin, unter allen Vektoren $\lambda \in [0, 1]^n$ mit $\sum_{i=1}^n \lambda_i g_i \leq G$ denjenigen mit maximalem Nutzen $\sum_{i=1}^n \lambda_i p_i$ zu finden. Der folgende Greedy-Algorithmus tut genau das. Er sortiert zunächst erneut die Objekte absteigend nach p_i/g_i , und benutzt danach alle Objekte, bis ein Objekt k gefunden wird, dessen Gewicht grösser als das Restgewicht G_R ist.¹ Von diesem wird $\lambda_k = G_R/g_k$ benutzt, sodass das verbleibende Restgewicht komplett aufgebraucht wird. Sofern weitere Objekte $k+1, \dots, n$ existieren, werden sie nicht benutzt.

FRACTIONALKNAPSACK($p_1, \dots, p_n, g_1, \dots, g_n, G$)

ALGORITHMUS

```

1 Sortiere die Objekte absteigend nach  $p_i/g_i$ 
2  $G_R \leftarrow G; k \leftarrow 1$ 
3 while  $k \leq n$  and  $g_k \leq G_R$  do  $\lambda_k \leftarrow 1; k \leftarrow k + 1$ 
4 if  $k \leq n$  then  $\lambda_k \leftarrow G_R/g_k; k \leftarrow k + 1$ 
5 while  $k \leq n$  do  $\lambda_k \leftarrow 0; k \leftarrow k + 1$ 
6 return  $(\lambda_1, \dots, \lambda_n)$ 
```

Theorem 3. *Der Algorithmus FRACTIONALKNAPSACK berechnet für das fraktionale Rucksackproblem eine optimale Lösung.* ■

OPTIMALITÄT
DER LÖSUNG

Offenbar ist nicht jede Lösung des fraktionierten Rucksackproblems auch eine Lösung des herkömmlichen Rucksackproblems. Umgekehrt gilt dies aber schon: Ist $I \subseteq \{1, \dots, n\}$ eine gültige Bepackung, dann kann diese durch den Vektor $\lambda \in \{0, 1\}^n$ mit $\lambda_i = 1 \Leftrightarrow i \in I$ repräsentiert werden. Die Optimalität kann bei dieser Transformation aber verloren gehen, denn das fraktionale Rucksackproblem erlaubt wesentlich mehr Lösungen. Also bildet der Wert einer optimalen Lösung für das fraktionale Rucksackproblem eine obere Schranke für den Wert einer optimalen Lösung für das herkömmliche Rucksackproblem. Zusammen mit den früheren Erkenntnissen erhalten wir damit die folgende Aussage:

Theorem 4. *Seien n Objekte mit den Nutzwerten $p_1, \dots, p_n \in \mathbb{N}$, den Gewichten $g_1, \dots, g_n \in \mathbb{N}$ und eine Gewichtsschranke $G \in \mathbb{N}$ gegeben. Seien w_{GREEDY} der Wert der vom Greedy-Algorithmus für das herkömmliche Rucksackproblem berechneten Lösung, w_{OPT} der Wert der optimalen Lösung des herkömmlichen Rucksackproblems und w_{FRAC} der Wert der optimalen Lösung des fraktionierten Rucksackproblems. Dann gilt $w_{\text{GREEDY}} \leq w_{\text{OPT}} \leq w_{\text{FRAC}}$.* ■

OBERE UND
UNTERE
SCHRANKE

Wird Branch-and-Bound für das herkömmliche Rucksackproblem angewendet, dann werden Lösungen durch Tripel $s_v = (I_v, E_v, U_v)$ gespeichert. Als untere Schranke hatten wir den Wert der Lösung $w(s_v) = (\sum_{i \in I_v} p_i)$ benutzt, als obere Schranke entsprechend $w(s_v) + R_v \cdot p_k/g_k$ bei einem Restgewicht R_v und einem maximalen Nutzen pro Gewichtseinheit von p_k/g_k . Die Lösung s_v kann aber nur noch durch Objekte aus U_v erweitert werden. Seien nun w_{GREEDY} und w_{FRAC} die Werte der Lösungen der Greedy-Algorithmen für das herkömmliche bzw. für das fraktionale

¹Falls ein solches Objekt nicht existiert, dann ist die Summe aller Gewichte grösser als das Gesamtgewicht und alle Objekte können benutzt werden.

BESSERE
 UNTERE
 SCHRANKE
 BESSERE OBERE
 SCHRANKE

Rucksackproblem, denen als Eingabe genau die Objekte aus U_v und die Gewichts-
 schranke R_v übergeben werden. Offenbar ist $w(s_v) + w_{\text{GREEDY}}$ noch immer eine
 untere Schranke für den maximalen Wert jeder gültigen Erweiterung von s_v , und sie
 ist besser (d.h., grösser oder gleich) als die bisherige untere Schranke $w(s_v)$. Wir be-
 obachten auch, dass $w(s_v) + w_{\text{FRAC}}$ eine gültige obere Schranke für den maximalen
 Wert jeder gültigen Erweiterung von s_v ist, und auch sie ist besser (d.h., kleiner oder
 gleich) als die bisherige obere Schranke. Die Berechnung dieser Schranken dauert ein
 wenig länger, ist aber relativ effizient in Zeit $\mathcal{O}(n \log n)$ durchführbar. Werden die
 Objekte in U_v nach p_k/g_k absteigend sortiert gehalten, dann ist die Laufzeit sogar
 nur linear in $|U_v|$. Durch die verbesserten Schranken erhoffen wir uns, schneller zur
 optimalen Gesamtlösung des herkömmlichen Rucksackproblems zu kommen.