Rucksackproblem

Simon Hünting ALDS 22/23

Prof. Dr. Meyer

Inhaltsverzeichnis

1. **Definition**

Das Rucksack-Problem (engl. Knapsack problem) ist ein Problem der kombinatorischen Optimierung. Bei einer Menge von Gegenständen, die jeweils ein Gewicht und einen Wert haben, besteht das Ziel darin, die Anzahl der einzelnen Gegenstände zu bestimmen, die in eine Sammlung aufgenommen werden sollen, so dass das Gesamtgewicht kleiner oder gleich einem bestimmten Grenzwert ist und der Gesamtwert so groß wie möglich ist. Der Name leitet sich von dem Problem ab, das sich jemandem stellt, der einen Rucksack mit einer bestimmten Größe hat und ihn mit den wertvollsten Gegenständen füllen muss.

In der gebräuchlichsten Version des Problems gibt es zwei Beschränkungen: das Fassungsvermögen des Rucksacks und die Anzahl der Gegenstände. Ziel ist es, den Gesamtwert der Gegenstände zu maximieren, ohne das Fassungsvermögen des Rucksacks zu überschreiten. Diese Version wird als 0/1-Knapsackproblem bezeichnet, da jeder Gegenstand entweder in den Rucksack aufgenommen werden muss (1) oder nicht (0). Da es nur zwei Entscheidungsmöglichkeiten 0 und 1 gibt handelt es sich um ein binäres Problem.

Es gibt noch weitere Varianten des Knapsack-Problems, z. B. das begrenzte Knapsack-Problem, bei dem jeder Gegenstand mehrfach aufgenommen werden kann, und das unbegrenzte Knapsack-Problem, bei dem es keine Begrenzung für die Anzahl der Gegenstände gibt, die aufgenommen werden können. Diese Varianten können mit unterschiedlichen Algorithmen gelöst werden, aber die Grundidee ist dieselbe: die Kombination von Gegenständen zu finden, die den Wert maximiert und dabei die Einschränkungen des Problems einhält.

Das Knapsack-Problem ist ein NP-hartes Problem, d. h. es gibt keinen bekannten Algorithmus, der es in polynomialer Zeit für alle möglichen Eingaben lösen kann. Daher wurden verschiedene Algorithmen entwickelt, um das Problem annähernd zu lösen oder für bestimmte Arten von Eingaben in angemessener Zeit annähernde Lösungen zu finden. Zu diesen Algorithmen gehören gierige Algorithmen, die bei jedem Schritt lokal optimale Entscheidungen treffen, und Algorithmen der dynamischen Programmierung, die das Problem in kleinere Teilprobleme aufteilen und die Lösungen dieser Teilprobleme speichern, um eine Neuberechnung zu vermeiden.

Zusammenfassung:

Das Rucksackproblem ist eine Art des Branch-and-Bound-Verfahren, es gibt also nicht eine oder alle Lösungen gibt, sondern die beste gesucht wird. Es handelt sich um ein Optimierungsproblem, um für eine bestimmte Zielfunktion eine optimale Lösung zu finden.

2. **Herangehensweise Rucksackproblem**

Am Beispiel der Urlaubsplanung am Strand lässt sich das Problem besser bildlich erklären. Man kann in seinen Koffer nur 20 Kilogram einpacken und muss sich deshalb entscheiden, welche Klamotten man einpackt. Hat man zum Beispiel 40 Kg an Klamotten zuhause, zwischen denen man sich entscheiden kann, muss man die Hälfte zuhause lassen. Da der Urlaub am Strand in der Wärme stattfinden soll, nimmt eine Winterjacke 1 Kg weg und hat einen extrem niedrigen Nutzen, während Badehose und Sonnencreme weniger wiegen und einen hohen Nutzen haben. So entscheidet der Algorithmus, sich gegen Winterjacke und pro Sonnencreme/Badehose.

Mathematische Herangehensweise:

Gegeben sind beim Rucksackproblem eine Menge von Objekten mit Gewichten („weights“) w[1…n] und Werten/Preisen/Nutzen („values“) v[1…n]. Es gibt eine maximale Tragkraft T.

Gesucht ist die Auswahl S $\subseteq${1,…,n} von Objekten, wo das Gesamtgewicht der ausgewählten Objekte w[i] $\leq$ T ist und der Gesamtwert der ausgewählten Objekte v[i] maximal ist.

Die erste Herangehensweise wäre es sich alle möglichen Lösungen anzugucken und am Ende die beste von denen auszuwählen. Es handelt sich um die erschöpfende Suche, deren Motto rohe Gewalt und die Laufzeit Ω = $2^n \times n$ beträgt. Erster Lösungsansatz ist Teilmengen aufzuzählen, um auf die Laufzeit Ω = 2^n zu kommen. Dabei wird bei jeder Zahl aus {1,…,n} entschieden, ob sie in der Teilmenge S ist oder nicht. Jede Entscheidung ist ein Knoten um Entscheidungsbaum bei der entweder $i \in S$ ist, oder nicht. Am Beispiel von n = 3, man nimmt drei Teile mit in den Urlaub, wird dieser Entscheidungsbaum verdeutlicht. (s. Abbildung 1)

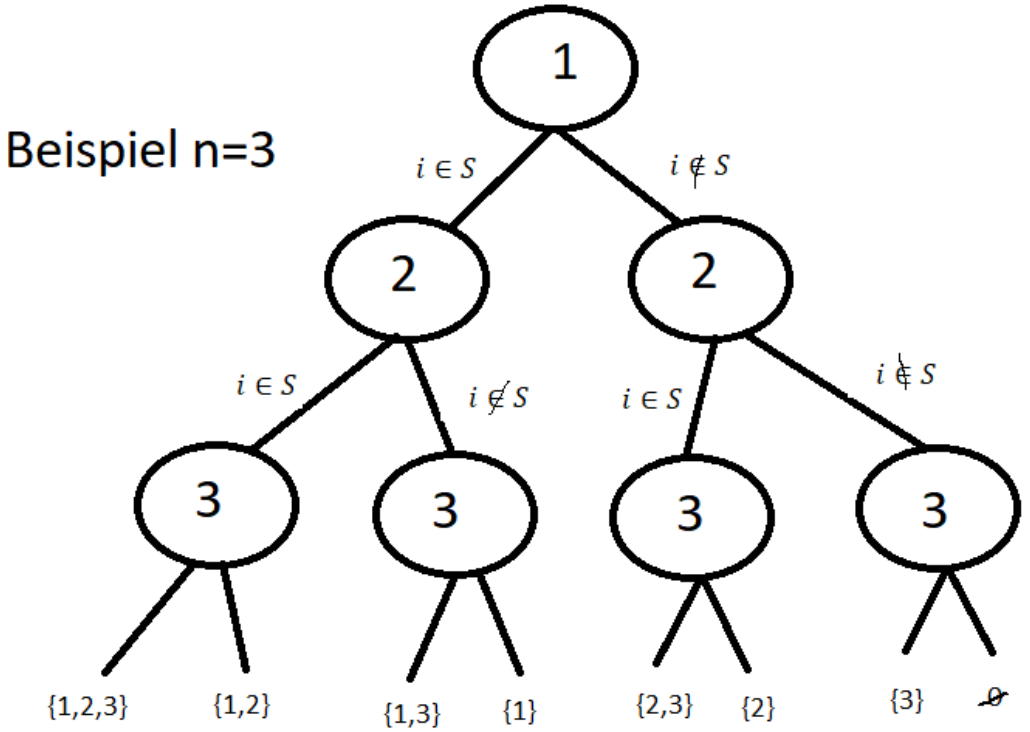

Abbildung 1

Hier handelt es sich allerdings nicht um eine optimale Lösung, da man jeden Zweig abarbeiten muss. Auch eine Sortierung nach weight und value hilft beim Rucksackproblem nicht, da man beide Punkte berücksichtigen muss und so keine einfache Sortierung möglich ist, so kann für einen Winterurlaub eine Ski Set von großem Wert sein, allerdings alleine schon 20 Kg erreichen und man kann keine Wechselklamotten oder ähnliches mitnehmen. Es müssen also sämtliche Kombinationen durchprobiert werden und der Wert von jedem Zweig verglichen werden. Wurde der bis dahin höchste Nutzen erreicht muss dieser zwischengespeichert werden und es werden weiter Kombinationen getestet.

Branch-and-Bound verfolgt den Lösungsansatz, dass Zweige bereits im Voraus ausgeschlossen werden können, da selbst die bestmögliche

Kombination, geringeren Nutzen hat, als der bis dahin zwischengespeicherte höchste Nutzen. (s. Abbildung 2)

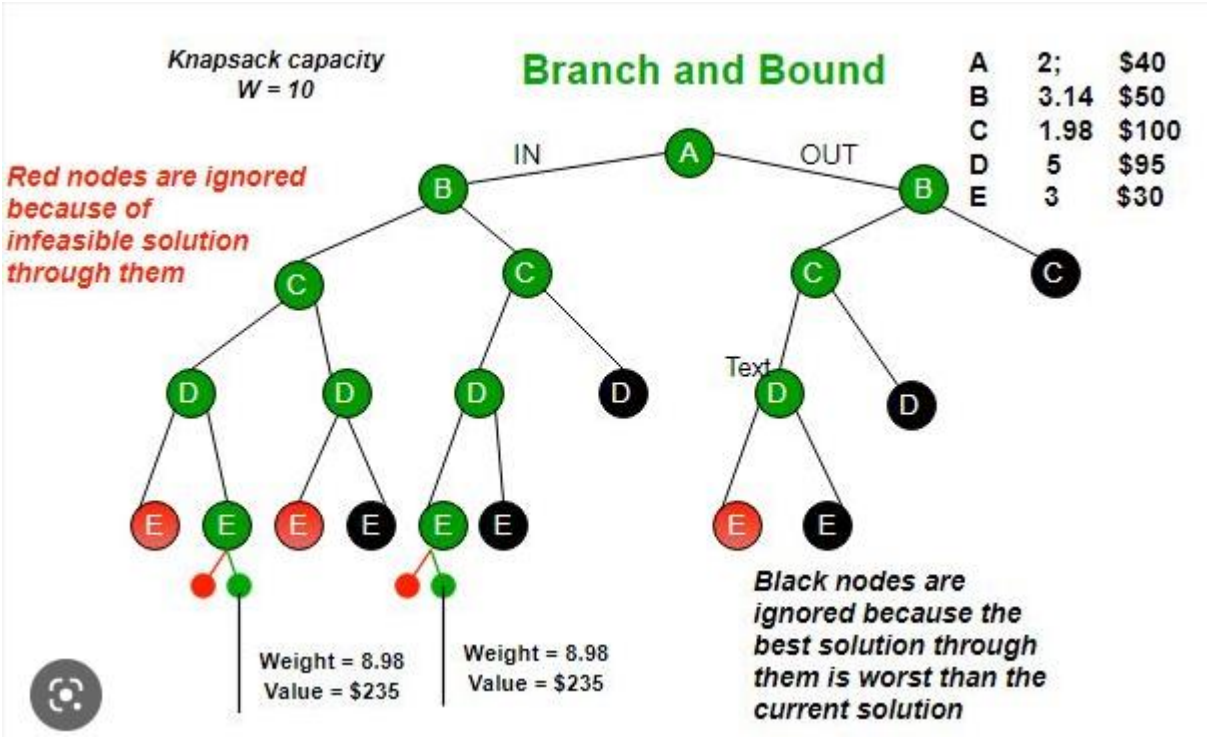

Abbildung 2 Branch & Bound (https://www.geeksforgeeks.org/branch-and-bound-algorithm/)

Sämtliche roten Punkte können nicht durchgeführt werden, da in diesem Beispiel das maximale Gewicht von 10 überschritten wird, sämtliche schwarze Punkte werden wie beschrieben ignoriert, da deren Nutzen nicht höher sein kann, als der bisherige höchste Nutzen von 235$. Lediglich die grünen Zweige können durchlaufen werden.

Man geht bei diesem binären Entscheidungsbaum so vor, dass man ganz links startet und nach rechts durchläuft. Der erste Zweig ist also, A-B-C-D-E. Da E das Maximum übertrifft setzt Backtracking ein und man geht im Ast zurück zum Knotenpunkt D und guckt sich den zweiten Punkt E an, welcher hier mit einem Gewicht von 8,98 hinterlegt ist. Danach geht man durch Backtracking wieder zurück und durchläuft die nächsten Zweige(ausgeschlossen der roten und schwarzen) bis man den optimalen Nutzen gefunden hat. Aus diesem Verfahren wird klar, dass ein Optimum

garantiert wird. Der Preis für dieses Optimum ist, dass die Null-Eins Entscheidung im Entscheidungsbaum zu 2^n Kombinationen führt, die in der Regel zwar nicht alle durchprobiert werden müssen, die Laufzeitkomplexität jedoch bei $O(2^n)$ bleibt. Geht man den Worst-Case durch würde man jeden einzelnen Zweig ablaufen und erst der ganz Rechte Weg würde zu optimalem Nutzen führen.

3. Problembeschreibung

Aufgabe ist es eine optimale Auswahl von unterschiedlichen Elementen mit verschiedenen Eigenschaften unter gegebenen Beschränkungen (Größen- oder Volumenbeschränkungen) zu finden. Es findet ein Raketenstart zur Raumstation statt und es können auf Grund von monetären Entscheidungen nur maximal 645 Kg zusätzliche Last mitgenommen werden. Zur Auswahl stehen acht Objekte mit folgenden Kennzahlen:

Objekt-Nr.	Gewicht in kg	Profit in 1.000 Euro
1	191	201
2	239	141
3	66	50
4	249	38
5	137	79
6	54	73
7	153	232
8	148	48

Es soll die optimale Kostellation gefunden werden, mit welchen Objekten zur Station geflogen werden kann. Es wird also eine Entscheidungsbaum durchlaufen, der $n = 8$ Tiefen hat.

4. **Implementierung in Java**

Zur Implementierung benötigt man zunächst eine Klasse (Knapsack) in der wir uns befinden und die beiden statischen Attribute bestValue und bestSelection deklariert sind, um die besten Zwischenlösungen zu speichern.

Es wird eine weitere Klasse (Item) mit Konstruktor erstellt, damit die acht Objekte erzeugt werden können, die potenziell mit zur Station genommen werden. Enthalten sind entsprechende Methoden um auf diese zugreifen und sich diese nachher über toString in der Konsole auszugeben. Instanzen sind die Objektnummer, das Gewicht und der Nutzen in Form von monetärem Profit.

Die Methode knapsack() enthält die Parameter selectedItems, selectedValue, availableItems, currentWeight und weightLimit und sorgt dafür dass der Entscheidungsbaum abgelaufen wird. Wofür die Parameter genau stehen, ist in den Kommentaren im Code hinterlegt. Es wird geguckt, ob alle Items verarbeitet sind (availableItems=0 ?) und selectedValue mit dem alten bestValue verglichen. Ist selectedValue > bestValue wird das neue bestValue bestimmt. Bei höchstem Value werden der Liste bestSelection die selectedItems hinzugefügt. Ist der Optimalfall noch nicht eingetreten wird der restliche Baum mit Backtracking abgegangen, im Java Code die else Abfrage ab Zeile 89.

In der main-method werden die acht Objekte initialisiert und die optimale Packliste bestimmt.

Java-Code

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Date: 2.12.2022
 * Das ist Code zur Demonstration des Rucksackproblems
 * @author simon
 * @version 1.0
 *
 *
 */
public class Knapsack {
  /*
   * statische Attribute, sollen bis dato beste Zwischenlösung speichern
   * @param bestValue
   * @param bestSelection
   */
  static int bestValue;
  static List<Item> bestSelection;

  public static class Item {

      private int objectNr;  //@param objectNr -> Objektnummer für Weltraummaterial
      private int value;       //@param value -> Profit in 1000€
      private int weight;      //@param weight -> Gewicht in Kg

      //Konstruktor mit Methoden um einzelne Items aufzurufen
      public Item(int objectNr, int weight, int value) {

        this.objectNr = objectNr;
        this.weight = weight;
        this.value = value;

      }
      private int getObjectNr() {
            return objectNr;
        }
        private void setObjectNr(int objectNr) {
            this.objectNr = objectNr;
        }
        private int getValue() {
            return value;
        }
        private void setValue(int value) {
            this.value = value;
        }
```

```
49        private int getWeight() {
50              return weight;
51        }
52        private void setWeight(int weight) {
53              this.weight = weight;
54        }
55        //@return wirft Itemdetails zurück
56        public String toString() {
57        return this.objectNr + " " + this.value + " " + this.weight;
58       }
59     }
60     /*
61      * Operation knapsack, zum Herausfinden des bestmöglichen Values
62  unter maximalem Gewicht
63      * @param selectedValue ausgewählter Profit
64      * @param availableValue verfügbarer Profit
65      * @param currentWeight momentan erreichtes Gewicht
66      * @param weightLimit max Gewicht von 645Kg
67      */
68     public static void knapsack(List<Item> selectedItems,
69                                         int selectedValue,
70
71                                         List<Item> availableItems,
72
73                                         int availableValue,
74
75                                         int currentWeight,
76
77                                         int weightLimit)
78
79        {
80           if (availableItems.isEmpty())            //alle Items sind
81  verarbeitet, Ergebnis wird als Bound/Messlatte gemerkt
82           {
83                 if (selectedValue > bestValue)
84                 {
85                       bestValue = selectedValue;
86                       bestSelection.clear();
87                       bestSelection.addAll(selectedItems);
88                 }
89                 else
90                 {
91                       Item item = availableItems.remove(0);
92                       int weight = item.getWeight();
93                       int value = item.getValue();
94                       if ((currentWeight < weightLimit) &&
95                             (selectedValue + availableValue > bestValue))
96                             {
97                             //neues Optimum nicht möglich, Item wird
98  genommen und weiter nach verfügbaren Items geguckt
```

```
                        if (currentWeight + weight <= weightLimit)
                            {
                            //Platz für Objekt in Knapsack
                            selectedItems.add(item);
                        knapsack(selectedItems,
                        selectedValue+value,
                        availableItems, availableValue-value,
                        currentWeight+weight, weightLimit);
                            selectedItems.remove(item);
//Entfernen des Items
                            }
                        knapsack(selectedItems, selectedValue,
                                availableItems, availableValue,
                                currentWeight, weightLimit);
                        }
                    availableItems.add(item);  //verfügbare Items zu
Liste hinzufügen
                    }
                }
    }

    public static void main(String[] args) {

        List<Item> items = new ArrayList<Item>();

        //@param items Liste mit Wert aus Aufgabenstellung befüllen
        items.add(new Item(1,201,191));
        items.add(new Item(2,141,239));
        items.add(new Item(3,50,66));
        items.add(new Item(4,38,249));
        items.add(new Item(5,79,137));
        items.add(new Item(6,73,54));
        items.add(new Item(7,232,153));
        items.add(new Item(8,48,148));

        knapsack(new ArrayList<Item>(),0,items,0,0,645);
        System.out.println("Die optimale Packliste:");
        int value = 0;
        int weight = 0;
        //optimale Packliste bestimmen
        for (Item item : bestSelection)
        {
            System.out.println(item);
            value += item.getValue();
            weight += item.getWeight();
        }
        System.out.println("Wert: "+value+" Gewicht: "+weight);
    }
}
```

5. Quellen:

Java Algorithmen und Datenstrukturen- Mit einer Einführung in die funktionale Programmiersprache Clojure (Manfred Meyer, unter Mitwirkung von Burkhard Neppert)

https://www.baeldung.com/java-knapsack

https://www.youtube.com/watch?v=ujhi2h70qIw&ab_channel=AlgorithmenundDatenstrukturen

https://www.geeksforgeeks.org/branch-and-bound-algorithm/