Hausarbeit Rucksackproblem

Das Rucksackproblem

Diskrete Mathematik
Dualen Hochschule Baden-Württemberg
Stuttgart

von
Paul Walker und Tom Hofer

Matrikelnummer: 3610783, 4775319
Abgabedatum: 30.06.2022

Inhaltsverzeichnis

1 Einleitung

Das Rucksackproblem ist ein kombinatorisches Optimierungsproblem. Das Problem ist viel erforscht, da es für viele Anwendungen einen praktischen Nutzen hat. Diese Hausarbeit soll eine Übersicht über das Rucksackproblem bieten und verschiedene Lösungs- und Approximationsverfahren vorstellen. Neben dem 0-1-Rucksackproblem werden auch weitere Probleme in der Kategorie der Rucksackprobleme vorgestellt.

2 Problemstellung

Ein Rucksack hat eine bestimmte Tragekapazität (z.b. 6 Kg) ein Dieb packt bei einem Wohnungsraub sein Diebesgut in diesen Rucksack. In der Wohnung befinden sich Gegenstände mit unterschiedlichem Gewicht und Geldwert. Gegenstände können verschieden oft vorhanden sein (z.b. Schmuck 4kg 500€ 1stk, Elektrogeräte 3kg 400€ 3stk, Schuhe 2kg 300€ 1stk, Geld 1kg 200€ 2stk). Der Dieb möchte bei dem Wohnungsraub einen möglichst großen Gewinn erzielen, er möchte also den Geldwert der in den Rucksack gepackten Gegenstände maximieren, da der Rucksack aber nur eine bestimmte Kapazität hat, muss zuerst ein Optimierungsproblem gelöst werden. Dieses Problem nennt sich das *Rucksackproblem* (oder Englisch: *Knapsack Problem*)

In diesem Beispiel wäre die beste Kombination: 1stk Geld, 1stk Schuhe, 1stk Elektrogeräte damit wird ein Gesamtgewinn von 900€ und einem Gesammtgewicht von 6kg erreicht.

Mathematisch formuliert: Es gibt $m \in \mathbb{N}$ Gegenstände. Sei $c_i \in \mathbb{N} : i \in I$ der Wert des Gegenstandes i, $a_i \in \mathbb{N} : i \in I$ das Gewicht des Gegenstandes i und $u_i \in \mathbb{N} : i \in I$ wie oft der Gegenstand vorhanden ist jeweils mit $i \in I = \{1, 2, \ldots, m\}$. Die Tragekapazität des Rucksacks ist $b \in \mathbb{N}$. Dann wird für den maximal erreichbaren Geldwert in Abhängigkeit zur Tragekapazität des Rucksacks $f(b)$ definiert:

$$f(b) := \max(\sum_{i=1}^{m} c_i x_i : \sum_{i=1}^{m} a_i x_i \leq b, 0 \leq x_i \leq u_i, x_i \in \mathbb{N}, i \in I)$$

Das hier dargestellte Rucksackproblem ist ein begrenztes Rucksackproblem. Beim unbegrenzten Rucksackproblem kann x_i jeden beliebigen wert in $\mathbb{N}$ annehmen und ist nicht von u_i begrenzt.

Das am häufigsten auftretende Rucksackproblem ist das 0-1-Rucksackproblem. Ein Begrenztes Rucksackproblem kann zu einem 0-1-Rucksackproblem vereinfacht werden, indem $u_i = 1$ gesetzt wird. Dann gilt:

$$f(b) := \max(\sum_{i \in I} c_i x_i : \sum_{i \in I} a_i x_i \leq b, x_i = \{0, 1\}, i \in I)$$

Anschaulich bedeutet das, dass jeder Gegenstand nur genau einmal vorhanden ist und daher auch nur einmal mitgenommen werden kann. Da das 0-1-Rucksackproblem das

Häufigste in der Kategorie der Rucksackprobleme ist, soll der Fokus im Folgenden auf dem 0-1-Rucksackproblem liegen.

Das Rucksackproblem gehört zur Kategorie der am schwersten zu lösenden Probleme, den $\mathcal{NP}$-vollständigen Problemen [1]. In der Praxis gibt es aber einige gute Lösungs- und Approximationsverfahren.

2.1 $\mathcal{NP}$-vollständige Probleme

Ein Algorithmus ist in polynomer Zeit lösbar, wenn für dessen Zeitkomplexität $O(n^k)$ gilt. Dabei ist k eine positive ganze Zahl und n die Komplexität des Eingabewertes. Diese Algorithmen haben die Zeitkomplexität $\mathcal{P}$ (Polynome Zeit).

Algorithmen, die sich nicht innerhalb dieser Zeitkomplexität beschreiben lassen, also eine höhere Zeitkomplexität als die Polynome Zeit $\mathcal{P}$ haben nennt man $\mathcal{NP}$-vollständige Probleme. Diese Probleme lassen sich nur mithilfe von Nichtdeterministischen Operationen in polynomer Zeit lösen. Am Modell einer Turing-Maschine erklärt bedeutet Nichtdeterminismus, dass der Nächste Zustand nicht unbedingt durch die vorhergehenden Zustände bestimmt ist. Während eine Nichtdeterministische Turing-Maschine die richtige Auswahl in polynomer Zeit berechnen kann, muss eine deterministische Turing-Maschine alle Möglichkeiten ausprobieren, um die richtige Auswahl zu erhalten und benötigt dafür deutlich länger als polynome Zeit. $\mathcal{NP}$-vollständige Probleme sind daher für einen Computer die am schwersten zu lösenden Probleme.

Ob sich $\mathcal{NP}$-vollständige Probleme allgemein in $\mathcal{P}$ Probleme umformen lassen, ist eine bisher ungelöste Frage [2, Kap. 15].

3 Anwendungen

Das Rucksackproblem tritt in der realen Welt häufiger bei Optimierungsproblemen, speziell bei Problemen in der Ressourcen Allokation auf. Man denke beispielsweise an einen LKW, der ein bestimmtes Transportvolumen hat und Güter, die bei verschiedenem Volumen einen unterschiedlichen Gewinn erzielen. Oder ein Containerschiff das ein bestimmtes Gewicht tragen kann und Container mit unterschiedlichem Gewicht und Gewinn.

Zusätzlich zu den praktischen Anwendungen gibt es auch theoretische Anwendungen für das Rucksackproblem. Rucksackprobleme sind ganzzahlige lineare Optimierungsprobleme. Andere ganzzahlige lineare Optimierungsprobleme können in Rucksackprobleme umgeformt werden [1], die umgeformte Version des Problems hat dabei dieselben Lösungen wie das Originalproblem [1]. Das heißt es können die bekannten Lösungsverfahren zur Lösung des Rucksackproblems auch bei anderen ganzzahligen linearen Optimierungsproblemen verwendet werden.

Eine Problemvariante des Rucksackproblems, die Subset Sum wird in der Kryptographie verwendet, beispielsweise beim Merkle-Hellman-Kryptosystem.

4 Techniken zur effizienteren Lösung

Da alle Rucksackprobleme $\mathcal{NP}$-schwer sind ist es unwahrscheinlich, dass jemals ein polynomieller Lösungsalgorithmus gefunden wird. Deshalb müssen Algorithmen eingesetzt werden, die den Lösungsraum ganz oder teilweise auflisten. Um diese Algorithmen effizienter zu gestalten wurden einige Techniken entwickelt, die den Lösungsraum eingrenzen können und so eine effizientere Lösung schaffen. Sie nutzen speziellen strukturellen Eigenschaften von Rucksackproblemen aus und ermöglichen es auch größere Probleme in Bruchteilen einer Sekunde zu lösen. Im Folgenden werden einiger dieser Techniken betrachtet.

Die in 5 und 6 vorgestellten Lösungs- und Approximationsverfahren nutzen die hier vorgestellten Techniken entweder oder können durch sie noch weiter beschleunigt werden.

4.1 Branch-and-bound

Im Grunde eine komplette Auflistung des Lösungsraumes in Form eines Baumes. Hier wird die Rucksack-Kapazität bei jeder Baumebene vergrößert. Anstatt bei jeder Baumebene alle Möglichkeiten zu bilden, werden allerdings Zweige, die nicht zu einer besseren Lösung führen können, direkt aussortiert. Dies geschieht durch die Verwendung von Grenzen [4].

4.2 Dynamische Programmierung

Die Auflistung des Lösungsraumes erfolgt ähnlich einer Breitensuche, allerdings werden zusätzliche Dominanz regeln eingeführt. Durch das Hinzufügen von Grenzbedingungen zur dynamischen Programmierung können fortgeschrittene Formen von Branch-and-bound Algorithmen erzeugt werden [4]. Dieses Verfahren wird in 5.2 genauer dargestellt.

4.3 State space relaxation

State space relaxation ist eine abgeschwächte Form der dynamischen Programmierung. Bei dieser Technik werden die Koeffizienten mit einem festen Wert multipliziert, um die Zeitliche und räumliche Komplexität zu verringern [4]. Dadurch wird allerdings auch die Genauigkeit verringert und ein zuvor optimaler Algorithmus wird zu einem Näherungsalgorithmus. Diese Technik wird von dem *voll-polynomiellen Approximationsverfahren* in 6.2 verwendet.

4.4 Preprocessing

Als Preprocessing wird das Festlegen von Variablen des Lösungsraumes, vor dem Start des eigentlichen Lösungsalgorithmus bezeichnet. Dies wird durch die Durchführung von Grenztests ermöglicht, durch die einige Werte für die Lösungsvariable ausgeschlossen

werden können [4]. So Werden Gegenstände deren Gewicht über die Rucksack-Kapazität direkt aussortiert.

5 Lösungsansverfahren

5.1 Brute Force

Bei n zur Auswahl stehender Elemente, gibt es 2^n Möglichkeiten verschiedene Kombinationen von Gegenständen daraus zu wählen. Bei jeder Iteration wird dann überprüft, ob die jetzige Kombination von Gegenständen die beste bisher ist. Ist sie das, wird die bisher beste Kombination ersetzt und der neue beste Wert gespeichert [6]. Der Algorithmus hat damit eine Komplexität von $O(n2^n)$. Er ist damit der schlechtest mögliche Algorithmus.

5.2 Dynamische Optimierung

Das im Folgenden vorgestellte Verfahren funktioniert zur Lösung des 0-1-Rucksackproblems. In der Literatur wird dieser Algorithmus verschieden genannt. In [5] wird dieser Algorithmus *Algorithmus von Gilmore und Gomoroy* genannt, in [2] wird der Algorithmus entweder *pseudopolynomieller Algorhitmus* oder *Algorithmus von Bellman und Dantzig* [2] genannt, in [6, 3] und in anderen Publikationen wird der Algorithmus einfach *Algorithmus mit Dynamischer Optimierung* genannt. Diese Bezichnung wird hier auch verwendet.

Der hier vorgestellte basiert auf *dynamischer Optimierung*. Dafür wird das Rucksackproblem zuerst vollständig reduziert und dann schrittweise wieder zum Originalproblem aufgebaut.

Begonnen wird mit dem einfachsten Problem: einem Rucksack mit Kapazität 0 mit einem Gegenstand zu füllen. Die Kapazität wird schrittweise erhöht bis die Originalkapazität erreicht ist. Es wird bei jeder Erhöhung der Kapazität geprüft, ob der Gegenstand hinzugefügt werden kann und ob dies einen Vorteil gegenüber der vorherigen Iteration bringt. Der Gegenstand wird hinzugefügt oder nicht basierend darauf was den Wert im Rucksack am höchsten werden lässt. Dann wird der nächste Gegenstand dazu genommen und die Kapazität des Rucksacks wieder auf 0 gesetzt. Wichtig ist dabei, dass die Gegenstände im Wert absteigend ausgewählt werden. Mit einem Gegenstand mehr wird wieder mit der Erhöhung der Rucksack Kapazität von 0 bis zur Originalkapazität begonnen. Das wird so lange wiederholt, bis versucht wurde alle Gegenstände hinzuzufügen [5]. Der maximal erzielbare Wert im Rucksack steht im unteren rechten Feld der Matrix, und ist das Ergebnis des Algorithmus.

In Code-Auszug 1 ist dieser Algorithmus beispielhaft in Python implementiert. Die Tabelle 2 zeigt das Ergebnis dieses Algorithmus für die Eingabe aus Tabelle 1.

Um herauszufinden welche Gegenstände in den Rucksack aufgenommen wurden, muss die Matrix schrittweise zurückverfolgt werden. Begonnen wird dabei mit dem unteren rechten Eintrag der Matrix. Ein Gegenstand i ist im Rucksack vorhanden, wenn im

Wert	Gewicht	Index
500€	4kg	1
400€	3kg	2
300€	2kg	3
200€	1kg	4

Tabelle 1: Werte

	0kg	1kg	2kg	3kg	4kg	5kg	6kg
0	0€	0€	0€	0€	0€	0€	0€
1	0€	0€	0€	0€	500€	500€	500€
2	0€	0€	0€	400€	500€	500€	500€
3	0€	0€	300€	400€	500€	700€	800€
4	0€	200€	300€	500€	600€	700€	900€

Tabelle 2: Lösungsmatrix

Vergleich zum vorherigen Gegenstand ein Gewinn erzielt wurde, also $m(i,j) \neq m(i-1,j)$ mit m als Lösungsmatrix (siehe Tabelle 2). Dieser Gegenstand wird dann dem Rucksack hinzugefügt und die verbleibende Kapazität im Rucksack verringert. Der nächste Eintrag in der Matrix der betrachtet werden muss ist dann also $m(i-1, j-weight[i])$.
Wenn $m(i,j) = m(i-1,j)$ gilt, dann ist der Gegenstand i nicht im Rucksack vorhanden. Der nächste Matrixeintrag der betrachtet werden muss ist dann $m(i-1,j)$. Das wird wiederholt bis der obere linke Matrixeintrag erreicht ist.

Der Algorithmus hat die Komplexität $O(nb)$, wobei b die maximale Kapazität des Rucksacks darstellt. Algorithmen mit einer solchen Komplexität werden *pseudo-polynomial* genannt [5].

Code-Auszug 1: Gilmore Gomoroy in Python

```python
from numpy import zeros

def gilmore_gomoroy(v, b):
  """
  Parameters:
  v (array) list of (value, weight) tuples
  b (int) knapsack capacity

  Returns
  m (array): solution matrix
  """

  # Matrix with solution
  m = zeros((len(v)+1, b+1))

  for i in range(1, len(v)+1):
    for j in range(1, b+1):
      weight = v[i-1][1]
      value = v[i-1][0]
      if weight > j:
        m[i, j] = m[i-1, j]
      else:
        m[i, j] = max(m[i-1, j], m[i-1, j-weight] + value)

  return m
```

6 Annäherungsverfahren

6.1 Greedy Algorithmus

Bei vielen Optimierungsproblemen wird ein Greedy Algorithmus eingesetzt, um das Problem zu lösen oder die Lösung zumindest gut annähern zu können [6].

Für das 0-1-Rucksackproblem Problem gibt es mehrere Greedy Strategien.

1. Schritt für Schritt Gegenstände mit absteigendem Wert wählen und in den Rucksack Packen.
2. Schritt für Schritt Gegenstände mit aufsteigendem Gewicht in den Rucksack Packen.
3. Schritt für Schritt Gegenstände mit aufsteigendem Wert pro Gewicht in den Rucksack Packen.

Dabei erzielt die letzte Strategie in der Praxis die besten Ergebnisse [6].
Die Zeitkomplexität dieses Algorithmus ist $O(n \log n)$

6.2 Voll-polynomielles Approximationsverfahren

Für das Rucksackproblem gibt es kein Polynomielles Lösungsverfahren. Mithilfe des Lösungsverfahrens mit dynamischer Optimierung kann aber ein voll polynomielles Approximationsverfahren geschaffen werden [2].

Die Laufzeit des Verfahrens mit dynamischer Optimierung hängt direkt von der Kapazität des Rucksacks ab. Daher liegt es nahe, die Gewichte und die Kapazität zu halbieren und abzurunden.

$$a_i^* := \left\lfloor \frac{a_i}{t} \right\rfloor \ : \ i \in I \ ; \ b^* := \left\lfloor \frac{b}{t} \right\rfloor$$

Damit kann die Laufzeit des Algorithmus in Abschnitt 5.2 von $O(nb)$ auf $O(n\frac{b}{t})$ reduziert werden. Dieser Algorithmus ist allerdings weiterhin *pseudo-polynomial*, da die Abhängigkeit von b weiterhin besteht.

Dieses Approximationsverfahren kann allerdings in ein voll polynomielles Verfahren umgewandelt werden, wenn $t = \epsilon \cdot \frac{b}{n}$. Demnach ergibt sich ein, von einem beliebigen $\epsilon > 0$ abhängiger Approximationsalgorithmus mit

$$O(n\frac{b}{t}) = O(n\frac{b}{\epsilon \cdot \frac{b}{n}}) = O(n^2 \cdot \frac{1}{\epsilon})$$

Nun Läuft das Verfahren in polynomer Zeit (siehe Abschnitt 2.1), das Approximationsverfahren ist also *voll-polynomiell*.

7 Arten von Rucksackproblemen

Neben den bereits vorgestellten Rucksackproblemen, dem begrenzten Rucksackproblem und dem 0-1-Rucksackproblem, gibt es noch zahlreiche weitere Formen von Rucksackproblemen. Im Folgenden sollen beispielhaft die Hauptcharakteristika einiger dieser Varianten erklärt werden.

7.1 Unbegrenztes Rucksackproblem

Dieses Rucksackproblem ist ähnlich zu den beiden bereits bekannten Varianten. Hier kann allerdings jeder Gegenstand beliebig oft gewählt werden.

$$f(b) := \max(\sum_{i=1}^{m} c_i x_i : \sum_{i=1}^{m} a_i x_i \leq b, 0 \leq x_i, x_i \in \mathbb{N}, i \in I)$$

7.2 Multiple-choice Rucksackproblem

Die Gegenstände sind in k Klassen K_j, $j = \{1, ..., k\}$ aufgeteilt. Aus jeder Klasse muss genau ein Gegenstand gewählt werden. Man stelle sich hier vor das ein Rucksack zum Wandern gepackt werden soll und ein Getränk aus mehreren Marken ausgewählt werden muss, eine Packung Müsliriegel aus mehreren Marken ausgewählt werden muss und so weiter.

$$f(b) := \max(\sum_{j=1}^{k} \sum_{i \in K_j} c_{ji} x_{ji} : \sum_{j=1}^{k} \sum_{i \in K_j} a_{ji} x_{ji} \leq b,$$

$$(\sum_{i \in K_j} x_{ji} = 1 : j = \{1, ..., k\}),$$

$$(x_{ji} \in \{0, 1\} : j = \{1, ..., k\}, i \in K_j),$$

$$i \in I)$$

7.3 Multi-constrained Knapsack Problem

Bezeichnet ein Rucksackproblem mit mehreren einschränkenden Größen, zum Beispiel Gewicht und Volumen. Allgemein formuliert gibt es n einschränkende Grenzen b_j, $j = \{1, ..., n\}$:

$$f(b) := \max(\sum_{i=1}^{m} c_i x_i : \sum_{i=1}^{m} a_{ji} x_i \leq b_j, j = \{0, ..., n\}, 0 \leq x_i, x_i \in \mathbb{N}, i \in I)$$

7.4 Subset sum Problem

Bezeichnet ein 0-1-Rucksackproblem, bei dem für alle Gegenstände Wert und Gewicht proportional zueinander sind $a_i \sim c_i : i \in I$ mit a_i als Gewicht, c_i als Wert und $I = \{1, 2, \ldots, m\}$. Dadurch müssen Wert und Gewicht nicht mehr separat voneinander betrachtet werden. Das Problem kann jetzt als Suche nach der Menge derjenigen Werte betrachtet werden, deren Summe so groß wie möglich, aber nicht größer als die Kapazität des Rucksacks ist.

$$f(b) := \max(\sum_{i=1}^{m} c_i x_i : \sum_{i=1}^{m} a_i x_i \leq b, x_i \in \{0, 1\}, i \in I)$$

7.5 Das Rückgeldproblem

Bei diesem Problem muss die Kapazität des Rucksacks, mit so wenig Gegenständen wie möglich exakt erreicht werden. Das namengebende Beispiel ist ein Kassierer der, mit so wenig Münzen wie möglich, einen bestimmten Rückgeldbetrag auszahlen muss. Jeder Gegenstand (Münze) ist unendlich oft vorhanden.

$$f(b) := \max(\sum_{i=1}^{m} x_i : \sum_{i=1}^{m} a_i x_i = b, 0 \leq x_i, x_i \in \mathbb{N}, i \in I)$$

Literatur

[1] Ernst-Peter Beisel. *Verfahren zur Lösung von 0-1-Rucksack Problemen. Theorie und Implementierung*. 2010. URL: http://www2.math.uni-wuppertal.de/~beisel/Rucksack/mainKnapsack.pdf (besucht am 21. 06. 2022).

[2] Bernhard Korte und Jens Vygen, Hrsg. *Kombinatorische Optimierung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018. ISBN: 978-3-662-57690-8. DOI: 10.1007/978-3-662-57691-5.

[3] Michail G. Lagoudakis. *The 0-1 Knapsack Problem. an Introductory Survey*. Techn. Ber. P.O. Box 44330, Lafayette, LA 70504: The Center for Advanced Computer StudiesUniversity of Southwestern Louisiana, 1996.

[4] David Pisinger. „Algorithms for Knapsack Problems“. Diss. Dept. of Computer Science, University of Copenhagen, 1995.

[5] Guntram Scheiterhauer. *Zuschnitt- und Packungsoptimierung. Problemstellungen, Modellierungstechniken, Lösungsmethoden*. Hrsg. von Ulrich Sandten & Kerstin Hoffmann. Vieweg+Teubner, 2008. ISBN: 978-3-8351-0215-6.

[6] Maya Hristakeva & Dipti Shrestha. *Different Approaches to Solve the 0/1 Knapsack Problem*. URL: http://micsymposium.org/mics_2005/papers/paper102.pdf (besucht am 23. 06. 2022).