Praktische Ansätze zur Lösung des Rucksackproblems

Clemens Rawald — 30. Mai 2018

„Schwierige Problemstellungen“
Seminar Sommersemester 2018 · Fachhochschule Wedel

Das Rucksack Problem *(engl. Knapsack Problem)* gehört zu den klassischen NP-vollständigen Problemen, dies wurde unter anderem von Richard Karp (1972) gezeigt. Es handelt sich dabei jedoch nicht nur um ein rein akademisches Problem, auch viele praktische Anwendungen wie z.B. Beladungsprobleme, Finanzmanagement, Verschnittprobleme lassen sich durch das Rucksack Problem darstellen. Auf die zwei grundlegenden Varianten, Branch and Bound und Dynamische Programmierung zum Lösen des Rucksack Problems wird im nachfolgendem eingegangen.

1 Einleitung

1.1 Das 0-1 Rucksack Problem

Gegeben sei eine endliche Kapazität C des Rucksacks und eine endliche Menge von Elementen indiziert von 1 bis n. Jedes dieser Elemente verfügt noch über zwei Eigenschaften, zum einen den Wert v_i *(engl. value)* und zum anderen ein Gewicht w_i *(engl. weight)*. Gesucht sei nun die Teilmenge von Elementen, die die Kapazität des Rucksacks nicht überschreitet aber den Wert maximiert.

Mathematisch formuliert:

$$\begin{aligned} maximiere \sum_{i=1}^{n} v_i x_i \\ so\ dass \sum_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0,1\}, v_i > 0, C \geq w_i > 0, C > 0 \end{aligned}$$

1.2 Varianten

Es gibt viele verschiedene Varianten des Rucksack Problems: bounded, unbounded, multi-dimensional, multi-objective und multiple-choice, um einige aufzuzählen. Bei bounded ist die Anzahl, die von einem Element i in den Rucksack gepackt werden darf, beschränkt. Beim 0-1 Rucksack Problem, was somit auch eine bounded Variante ist, ist es 1, dies kann jedoch auch eine beliebige andere natürliche Zahl sein. Bei unbounded hingegen fällt diese Restriktion weg, jedes Element kann beliebig oft in Rucksack hinzugefügt werden, aber natürlich nur unter der Bedingung, dass die Kapazität nicht überschritten wird. Die multi-dimensional Variante fügt weiterte Restriktionen hinzu, d.h. der Rucksack hat mehrere Kapazitätsbeschränkungen. Im Gegensatz dazu fügt die multi-objective Variante mehre Ziele hinzu, die maximiert werden sollen. Zu guter Letzt noch die multiple-choice Variante, dort hat man statt nur einen Rucksack mehrere Rucksäcke, die unabhängig von einander gefüllt werden können. Es ist auch möglich mehrere dieser Varianten zu kombinieren. Im Nachfolgendem wird sich auf das 0-1 Rucksack Problem beschränkt, denn dies ist die am bestem untersuchte Variante des Rucksack Problems und vermeidet unnötige Komplexität bei den hier beschriebenen Algorithmen.

1.3 Bruteforce

Gegeben sei eine Menge von n Elementen, indem man für jede Teilemenge den Wert berechnet, wird auch sicher die Optimale Lösung gefunden. Das Problem ist jedoch, dass die Menge aller Teilmengen gleich der Potenzmenge $\mathcal{P}$ ist und die Potenzmenge eine Mächtigkeit von 2^n hat. Somit ist dies eine sehr ineffiziente Herangehensweise und wird schon für kleine n unpraktikabel. Beispiel: Gegeben sei eine Probleminstanz mit 50 Elementen und es können 1.000.000 Lösungen pro Sekunde berechnet werden, dann braucht der Algorithmus ca. 36 Jahre, um diese Probleminstanz zu lösen.

1.4 Greedy-Algorithmus

Eine gegebene Menge von n Elementen wird mittels Profitablitätsindex $\frac{v_i}{w_i}$, absteigend sortiert. Es wird das Element mit dem Index b bestimmt, so dass $\sum_{i=1}^{b-1} v_i \leq C \wedge \sum_{i=1}^{b} v_i > C$ gilt. Die vorhergehende Bedingung sagt aus, dass wir so lange Elemente hinzufügen, bis wir das Element mit den Index b erreichen, welches die Kapazitätsbeschränkung überschreiten würde. Wichtig dabei ist jedoch, dass auch keine weiteren Elemente hinter dem $b$tem Index hineingepackt werden, auch wenn diese die Kapazitätsbeschränkung nicht überschreiten würden. Das Element mit dem Index b wird auch als das Break Item bzw. Critical Item bezeichnet. Die Laufzeit des Algorithmus liegt in der

Größenordnung von $\mathcal{O}(n \log(n))$, welche polynomiell ist und gute Ergebnisse in der Praxis erzielt, es handelt sich dabei jedoch um eine Heuristik, d.h. die Lösung kann beliebig weit vom Optimum abweichen.

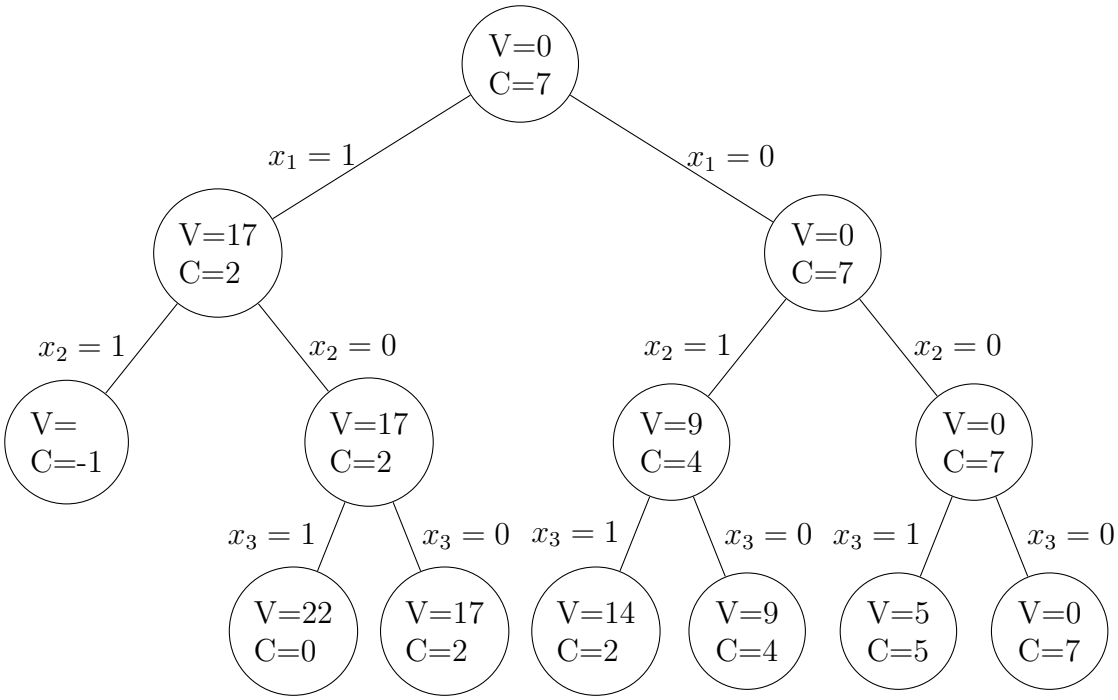
1.5 Backtracking

Auch diese Herangehensweise in algorithmisch sehr einfach, es handelt sich hierbei um eine schlichte Tiefensuche, die sich das Prinzip des „Divide and Conquer“ zunutze macht. Im Gegensatz zum Bruteforce Ansatz müssen wir nicht mehr alle möglichen Lösungen betrachten, sondern können im Falle, dass wir eine umgültige Teillösung haben schon vorzeitig abbrechen. Jedoch hat der Algorithmus noch immer exponentielle Laufzeit, d.h. auch dieser Algorithmus erweist sich als nicht praktikabel für große n.

1.6 Beispiel

$$\begin{aligned} \textit{maximiere } & 17x_1 + 9x_2 + 5x_3 \\ \textit{so dass } & 5x_1 + 3x_2 + 2x_3 \leq 7 \end{aligned}$$

Die Werte der ersten Gleichung entsprechen dem jeweiligen v_i eines Elements, die der zweiten Gleichung dem jeweiligen w_i. Des Weiteren ist der zweiten Gleichung zu entnehmen, dass der Rucksack eine Kapazität C von 7 hat.



Darstellung einer Beispiel Tiefensuche mittels Backtracking Algorithmus: Jeder Knoten enthält den aktuellen Wert V und die verbleibende Kapazität C des Rucksacks. Wie man der oberen Darstellung entnimmt, kann der Algorithmus auch schon frühzeitig mit der weiteren Berechnung eines Teilbaums aufhören, wenn die Kapazität überschritten wird. Dies tritt bei $x_1 = 1$ und $x_2 = 1$ auf, dort wird die Kapazität des Rucksacks um 1 überschritten. Somit muss in diesem Fall x_3 nicht mehr betrachtet werden. Die optimale Lösung ist 22 mit $x_1 = 1$ $x_2 = 0$ und $x_3 = 1$.

2 Branch and Bound

2.1 Allgemein

Es handelt sich um eine allgemeine Optimierungsstrategie, sie setzt auch auf das bewährte Prinzip des „Divide and Conquer“, jedoch kommt im Gegensatz zum Backtracking nun auch noch ein Bounding zum Einsatz. Ein weiterer Unterschied besteht darin, dass eine andere Expansions Strategie gewählt werden kann. Sie ist nicht nur auf die Tiefensuche beschränkt, Breitensuche bzw. Bestensuche sind auch möglich. Als bounding Algorithmus kann z.B. eine abgewandelte Version des zuvor vorgestellten Greedy Algorithmus genommen werden.

2.2 LP-Relexation

Die Beschränkung, dass nur ganze Teile von Elementen eingepackt werden dürfen, wird aufgehoben $x \in [0, 1]$. Es werden die Elemente wieder mithilfe des Profitbilitätsindex absteigend sortiert und dann das Break Item mit dem Index b bestimmt. *(siehe Greedy)* Der Optimale Wert V für einen Rucksack mit der Kapazität C kann nun wie folgt bestimmt werden:

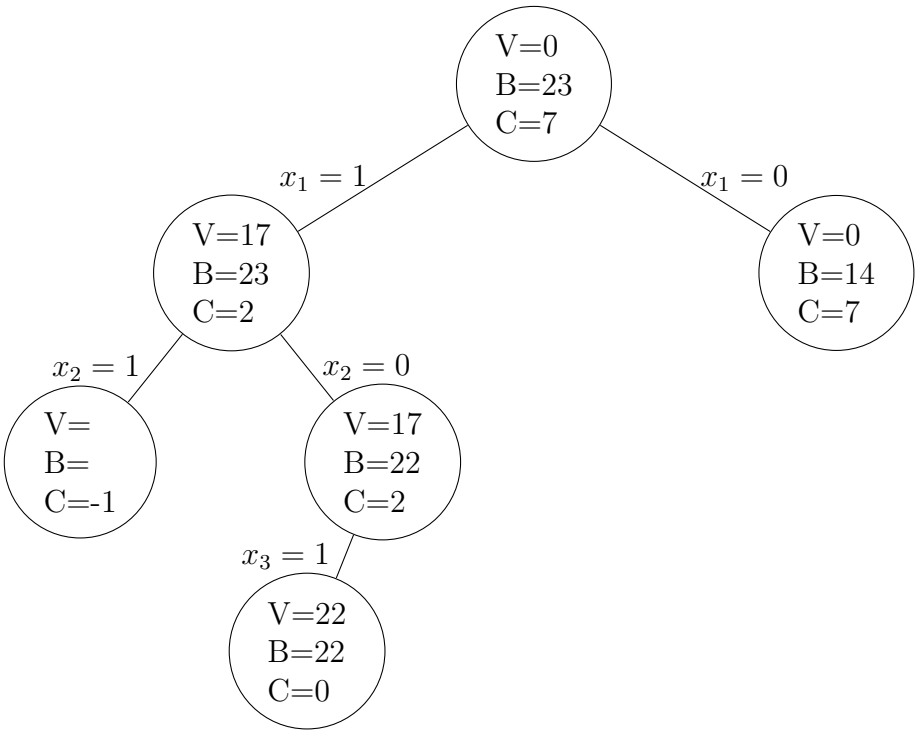
$$V = \sum_{i=1}^{b-1} v_i + (C - \sum_{i=1}^{b-1} w_i)/w_b v_b$$

Hierbei handelt es sich auch um die Lösung des fraktionalen Rucksack Problems, aber nicht für die 0-1 Variante des Rucksack Problems. Es handelt sich jedoch um ein Upper bound, dies ist leicht zu erkennen, denn den Wert $\lfloor V \rfloor$ können wir nicht überschreiten.

2.2.1 Beispiel

$$\begin{aligned} maximiere\ 17x_1 + 9x_2 + 5x_3 \\ so\ dass\ 5x_1 + 3x_2 + 2x_3 \leq 7 \end{aligned}$$

Es handelt sich um das gleiche Beispiel wie zuvor beim Backtrack Ansatz zum besserem Vergleich. Um den Unterschied zwischen Branch and Bound und Backtracking besser deutlich zu machen, wird weiterhin via Tiefensuche expandiert. Es kommt jedoch nun noch das Bounding hinzu.



Der oberen Darstellung ist zu entnehmen, dass jeder Knoten nun noch über einen Wert B verfügt. Dieser Wert wird mittels des Bounding Algorithmus für jeden Teilbaum, der betrachtet wird, berechnet. Zu diesem Wert muss nun noch der tatsächliche Wert, der sich bisher in dem Rucksack befindet, hinzuaddiert werden. Algorithmisch gesehen besteht kein Unterschied zwischen der Komplexität von Backtracking und Branch and Bound, aber wie man in diesem Beispiel sehen kann, wird der Suchaufwand beträchtig reduziert. Im Allgemeinem eignet sich die Branch and Bound Algorithmus am bestem für Probleme, bei denen v_i und w_i der Elemente nicht stark korrelieren, denn dort können viele Lösungsmöglichkeiten aufgrund des Boundings schon früh ausgeschlossen werden. Somit lässt sich auch sagen, dass es unterschiedlich schwere Instanzen von Rucksack Problemen gibt. Sie werden um so schwerer, so stärker v_i und w_i der Elemente mit anderer korrelieren, was oft in der Praxis der Fall ist.

3 Dynamische Programmierung

3.1 Allgemein

Es handelt sich auch hierbei um eine Optimierungs Technik, die auf verschiedene Probleme angewendet werden kann. Die Idee ist es wiederum, dass Problem in kleinere Teilprobleme zu zerlegen. Hierbei gibt es aber einen entscheidenden Unterschied, denn die Teilprobleme sind voneinander abhängig und die Lösungen werden gespeichert, um Neuberechnungen zu vermeiden. Darüber hinaus wird die Lösung auch nicht via „Top-Down“ sondern via „Bottom-Up“ berechnet, indem die gespeicherten Teillösungen von kleineren Teilproblemen zu Lösungen übergeordneter Probleme zusammengesetzt werden.

3.2 Algorithmus

Die Funktion $\mathrm{DP}(i, j)$ ist wie folgt definiert:
i sei der maximal zugelassene Index von einem Element, dass in den Rucksack gepackt werden darf und j sei der Wert der genau erreicht werden soll. Das Ergebnis der Funktion ist das minimale Gewicht. Das heißt: $\mathrm{DP}(3, 12)$ sagt aus, es dürfen die Elemente mit den Indizes $1, 2$ und 3 in den Rucksack gepackt werden. Gesucht ist die Teilmenge, die genau den Wert 12 erreicht und für die das Gewicht für den Wert 12 minimal ist. Falls keine Teilmenge existiert beträgt das Gewicht ∞. Für ein allgemeines Rucksack Problem bedeutet dies, dass $\mathrm{DP}(n, j)\ \forall j \in [0, V_{sum}]\ V_{sum} = \sum\limits_{i=1}^{n} v_i$ gelöst werden muss. Im Anschluss muss nur noch das Größte j bestimmt werden, so dass $\mathrm{DP}(n, j) \leq C$ gilt. Unter der Annahme dass $\mathrm{DP}(i-1, j)\ \forall\ j$ gelöst werden kann, können auch alle $\mathrm{DP}(i, j)$ wie folgt bestimmt werden:

$$\mathrm{DP}(i, j) = \begin{cases} \min(\mathrm{DP}(i-1, j), \mathrm{DP}(i-1, j-v_i) + w_i), & \text{if } v_i \leq j \\ \mathrm{DP}(i-1, j), & \text{otherwise} \end{cases}$$

Daraus folgt auch das $\mathrm{DP}(0, 0) = 0$ und $\mathrm{DP}(0, j) = \infty\ j > 0$

3.3 Beispiel

$$\begin{aligned} \textit{maximiere}\ & 2x_1 + 3x_2 + 5x_3 + 7x_4 + 4x_5 \\ \textit{so dass}\ & 1x_1 + 4x_2 + 3x_3 + 6x_4 + 5x_5 \leq 12 \end{aligned}$$

Index \ Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	0	∞	1	4	∞	5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	0	∞	1	4	∞	3	∞	4	7	∞	8	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	0	∞	1	4	∞	3	∞	4	7	7	8	∞	9	∞	10	13	∞	14	∞	∞	∞	∞
5	0	∞	1	4	5	3	6	4	7	7	8	9	9	12	10	13	14	14	15	18	∞	19

Die Zeilen werden nacheinander mithilfe der Funktion DP berechnet. Die Berechnung findet via „Bottom-Up“ statt, um wiederholtes Berechnen von gleichen Werten zu vermeiden. Die eingezeichneten Pfeile zeigen an, wie sich die Optimale Lösung zusammensetzt. Dies wäre auch eine Möglichkeit, wie man die Zusammensetzung einer Lösung abspeichern könnte. Dieser Ansatz benötigt aber zusätzlichen Speicher. Eleganter ist die Variante, die sich den zuvor vorgestellten Algorithmus zu Nutze macht. Man schaue sich einen Tabellen Eintrag an, wenn $(i, j) = (i - 1, j)$ gilt kann man daraus folgern, dass es besser war, das Element i nicht in den Rucksack zu packen. Dies entspricht somit dem otherwise Fall $\mathrm{DP}(i, j) = \mathrm{DP}(i - 1, j)$. Andernfalls gilt, dass der Eintrag $(i, j) < (i - 1, j)$ ist, somit wurde das Element i in den Rucksack gepackt. Wenn man vom Eintrag $(i - 1, j)$ noch v_i Spalten nach links geht kommt man zum Eintrag $(i - 1, j - v_i)$. Die vorherige beste Teillösung ohne das Element i. Wenn man diese Schritte wiederholt kann man für alle Elemente bestimmen, ob sie in den Rucksack gepackt wurden oder nicht.
Die Optimale Lösung ist 14 (siehe Tabelleneintrag $5, 14$). Dort ist das Gewicht 10, welches kleiner ist als 12 und j ist mit 14 somit maximal. Die Belegungen von x_1 bis x_5 lassen sich wie oben beschrieben bestimmen.

$$
\begin{aligned}
(5, 14) &= (4, 14) && \Rightarrow x_5 = 0 \\
(3, 14) &\neq (4, 14) && \Rightarrow x_4 = 1 \\
(3, 7) &\neq (2, 7) && \Rightarrow x_3 = 1 \\
(2, 2) &= (1, 2) && \Rightarrow x_2 = 0 \\
(1, 2) &\neq (0, 2) && \Rightarrow x_2 = 1
\end{aligned}
$$

3.4 Pseudopolyminell

Wenn man sich die Komplexität des eben vorgestellten Algorithmus betrachtet, erkennt man, dass man die komplette Tabelle berechnen muss, um die Lösung zu ermitteln. Die Berechnung eines Feldes ist konstant, daraus folgt die Größe der Tabelle gibt die Laufzeit und den Speicherbedarf an.

Laufzeit: $\mathcal{O}(nV_{sum})$

Speicher: $\mathcal{O}(nV_{sum})$

Somit könnte man denken, dass es sich um einen polynomiellen Algorithmus handelt. Dies ist aber nicht der Fall, denn V_{sum} ist unabhänging von der Anzahl der Gegenstände i, die sich im Rucksack befinden können und kann

beliebig groß sein. In praktischen Anwendungen kann dies eventuell kein Problem sein, weil v_i, w_i und C beschränkt sind. In diesem Fall hätten wir dann einen echten polynomiellen Algorithmus für das 0-1 Rucksack Problem. Anderenfalls handelt es sich nur um einen pseudopolyminellen Algorithmus.

3.5 PTAS/FPTAS

Beim Fully Polynomial Time Approximation Scheme handelt es sich um eine Erweiterung des eben vorgestelltem Algorithmus DP. Dieser ist im allgemeinem pseudopolyminell, durch die Erweiterung soll er polyminell werden. Dies hat aber zur Folge, dass es sich nur noch um eine Approximation Scheme handelt. Der Algorithmus liefert somit eine Lösung, die nicht optimal sein muss. Aber im Gegensatz zu einer Heuristik ist der Fehler begrenzt, d.h. es lässt sich eine Aussage darüber machen, wie weit man maximal von der optimalen Lösung entfernt sind.

1. $v_{max} = \max\{v_1, \ldots, v_n\}, \varepsilon \in (0,1), x = \frac{v_{max}\varepsilon}{n}$
2. $v'_i = \left\lfloor \frac{v_i}{x} \right\rfloor$
3. wie führen den Algorithmus DP mit der neuen Eingabe aus.
4. wir erhalten die neue Lösung

Warum die Werte so definiert sind, wie sie es sind, wird sich in den beiden folgenden Abschnitten zeigen.

3.5.1 Laufzeit

Laufzeit ohne FPTAS

$$\begin{aligned} nV_{sum} &\leq n(nv_{max}),\ V_{sum} \leq nv_{max} \\ &= n^2 v_{max} \end{aligned}$$

Laufzeit mit FPTAS

$$\begin{aligned} n^2 v'_{max} &= n^2 \left\lfloor \frac{v_{max}}{x} \right\rfloor \\ &= n^2 \left\lfloor \frac{v_{max}}{\frac{v_{max}\varepsilon}{n}} \right\rfloor \\ &= n^2 \left\lfloor \frac{v_{max} n}{v_{max}\varepsilon} \right\rfloor \\ &= n^2 \left\lfloor \frac{n}{\varepsilon} \right\rfloor \leq n^2 \frac{n}{\varepsilon} = \frac{n^3}{\varepsilon} \end{aligned}$$

Nach allen Substitutionen erhält man eine Laufzeit von $\frac{n^3}{\varepsilon}$. Somit kann die untere Grenze von $\varepsilon > 0$ erklärt werden, denn durch 0 darf nicht geteilt werden, bzw. wenn man sich den $\lim_{\varepsilon \to 0} \frac{n^3}{\varepsilon} = \infty$ anschaut wird die Laufzeit desto

größer je kleiner ε wird und tendiert in Richtung unendlich. Des Weiteren erkennt man, dass es sich hierbei um ein Fully Polynomial Time Approximation Scheme handelt. Dies bedeutet nämlich, dass das ε in einer polynomiellen Abhängigkeit vorkommt, hier $\frac{1}{\varepsilon}$. Würde es in einer nicht polynomiellen Abhängigkeit vorkommen, wäre es nur ein Polynomial Time Approximation Scheme. Der Vorteil eines FPTAS gegenüber eines PTAS ist, dass Änderungen an ε nur polynomielle Auswirkungen auf die Laufzeit haben.

3.5.2 Fehler

$$
\begin{aligned}
OPT &= \text{optimale Lösung} \\
APX &= \text{approximierte Lösung} \\
v_i' &= \left\lfloor \frac{v_i}{x} \right\rfloor \\
v_i &\mod x \leq x - 1 \leq x
\end{aligned}
$$

$$
\begin{aligned}
APX &\geq OPT - nx \\
&= OPT - n\frac{v_{max}\varepsilon}{n} \\
&= OPT - v_{max}\varepsilon,\ OPT \geq v_{max} \\
&\geq OPT - OPT\varepsilon \\
&= (1-\varepsilon) * OPT
\end{aligned}
$$

Nachdem man auch hier allen Substitutionen gemacht hat, sieht man warum $\varepsilon < 1$ gelten muss, denn wenn $\varepsilon >= 1$ ist folgt daraus, dass $APX <= 0$ ist. Diese Aussage ist nicht hilfreich, weil ein leerer Rucksack sicher immer den Wert 0 erreicht, nämlich dann, wenn man nichts hinein tut.

4 Zusammenfassung

Obwohl das Rucksack Problem NP-vollständig ist, gibt es gute praktische Verfahren, um selbst große Probleminstanzen zu lösen. Wobei jedoch zu beachten ist, dass die hier vorgestellten Verfahren verschiedene Stärken und Schwächen aufweisen. Der Branch and Bound Ansatz ist sehr schnell bei Problem Instanzen bei denen v_i und w_i wenig korrelieren. Dynamische Programmierung kann sehr große Instanzen mit mehreren tausenden Elementen lösen, vorausgesetzt man hat die entsprechende Rechenleistung. In der Praxis ist es auch üblich, Verfahren zu kombinieren. Eine Möglichekeit ist z.B. Branch and Bound zu Vorverarbeitung zu nutzten, um im Anschluss mit

Hilfe von Dynamischer Programmierung das reduzierte Problem zu lösen. Die hier vorgestellten Herangehensweisen zum Lösen des Rucksack Problems sind sehr allgemeine Optimierungsmethoden und lassen sich zum Teil auch auf andere Problemstellungen übertragen.

Literatur

[1] Ernst-Peter Beisel. Verfahren zur lösung von 0-1-rucksack problemen, Oktober 2010.

[2] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[3] David Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, 1995.

[4] David Pisinger. David pisinger's optimization codes, 1995.

[5] Vijay V. Vazirani. *Approximation Algorithms*, pages 68–70. Springer-Verlag New York, Inc., New York, NY, USA, 2001.