Techniken für die Exakte Lösung von Rucksackproblemen

Ulrich Pferschy
Universität Graz

Inhalt:

1. **Einleitung**
2. **Dynamisches Programmieren**
3. **Algorithmus auf Bit-Basis**
4. **Bounded Knapsack Problem**
5. **Die Champions**

1. Inhalt

• **Rucksackproblem / Knapsack Problem (KP)**

⇒ **gegeben:** n Objekte/items j mit Profit p_j und Gewicht w_j, Rucksack Kapazität c.

⇒ **gesucht:** Finde eine Teilmenge von items mit Gesamtgewicht $\leq c$ und maximalem Gesamtprofit.

$$
\begin{aligned}
(KP) \quad z^* := \text{maximize} \; & \sum_{j=1}^{n} p_j x_j \\
\text{s.t.} \; & \sum_{j=1}^{n} w_j x_j \leq c \\
& x_j \in \{0,1\}, \quad j = 1, \ldots, n.
\end{aligned}
$$

⇒ **Motivation:**

- Auswahl von Gütern/Projekten unter einer Budgetrestriktion
- Optimale Portfoliogestaltung
- Unterproblem bei Integer Programming (Column Generation)
- ...

⇒ **Literatur:**

H. Kellerer, U. Pferschy, D. Pisinger,
Knapsack Problems
Springer, 2004, 546 pages, EUR 107.–
`www.diku.dk/knapsack`

⇒ **Komplexität:**

- einfachstes integer programming problem (ILP)
- $\mathcal{NP}$–hartes Problem
- in der Praxis meist sehr gut lösbar
- Effiziente Approximationsschemata
- (KP) mit "=" Bedingung:

$$\sum_{j=1}^{n} w_j x_j = c$$

nicht approximierbar für einen Faktor von n^{ε}.

⇒ **2. Dynamisches Programmieren:**

Eigenschaft der *optimalen Substruktur* bzw.
Bellmannsches Optimalitätsprinzip:

Jeder Teil einer optimalen Lösung ist selbst optimale Lösung eines Unterproblems
$\Longrightarrow$ Optimallösung aufbauen durch iteratives Lösen geeigneter Unterprobleme

$z_j(d)$: Optimallösung eines Problems mit item Menge $\{1, \ldots, j\}$, $j \leq n$, und Kapazität $d \leq c$.

$$
z_j(d) = \begin{cases} z_{j-1}(d) & \text{if } d < w_j \,, \\ \max\{z_{j-1}(d),\, z_{j-1}(d - w_j) + p_j\} & \text{if } d \geq w_j \,. \end{cases}
$$

offensichtlich array der Dimension $n \cdot c$

Problem: Speicherung aller Teillösungen!

Version 1:

für jeden entry $z_j(d)$:
Speicherung der zugehörigen Lösungsmenge ($\leq n$ items)

$\Longrightarrow$ **Zeit:** $O(n^2c)$ **Speicher:** $O(n^2c)$

Version 2:

Beobachtung:

Zur Berechnung von $z_{j+1}(\cdot)$ wird nur $z_j(\cdot)$ benötigt

$\Longrightarrow$ nur 2 arrays: $z_{alt}(d), z_{neu}(d)$

bei jeder Update-Operation muß Lösungsmenge kopiert werden

$\Longrightarrow$ **Zeit:** $O(n^2c)$ **Speicher:** $O(nc)$

Version 3:

Beobachtung:

beim Update einer Lösungsmenge wird nur ein einziges Element zugefügt

$\Longrightarrow$ Dynamische Speicherstruktur

Neues Element mit Pointer zur bisherigen Lösung

$\Longrightarrow$ **Zeit:** $O(nc)$ **Speicher:** $O(nc)$

Version 4:

Überlegung: einfach nur letztes Element speichern

$\Longrightarrow$ Rekonstruktion der Lösung durch Backtracking

Problem: Elemente können mehrfach vorkommen! unzulässig!

Beispiel: $n = 3$, $c = 5$,

j	1	2	3
p_j	4	3	2
w_j	4	3	1

$z_j(d)$:

$d\backslash j$	1	2	3	items
0	0	0	0	$\emptyset$
1	0	0	2	3
2	0	0	2	3
3	0	3	3	2
4	4	4	5	2,3
5	4	4	6	1,3

Ausweg: n Iterationen: immer nur ein Element der Lösung bestimmen

$\Longrightarrow$ **Zeit:** $O(n^2c)$ **Speicher:** $O(n+c)$

Version 5:

Grundidee: Divide and Conquer

Beobachtung:
Mit Dynamischem Programmieren wird für alle Kapazitäten $\leq c$ der optimale Lsüngswert ohne Zusatzspeicher berechnet.

- Partition der item Menge N in 2 Hälften N_1, N_2
- Rekursive Lösung der Teilproblem
- Kombination der Lösungen zu Gesamtlösung
- Bestimmung der Lösungsmenge am Rekursionsende mit $|N_i| = 1$.

$\Longrightarrow$ **Zeit:** $O(nc)$ **Speicher:** $O(n+c)$

[Pferschy, 1999]

Algorithm Divide & Conquer D&C (N, C)**:**

Divide

Partition N into two disjoint subsets N_1 and N_2 with $|N_1| \approx |N_2|$.
Perform **solve** (N_1, C) returning $v(N_1, c),\ c = 0, \ldots, C$.
Perform **solve** (N_2, C) returning $v(N_2, c),\ c = 0, \ldots, C$.
Find C_1, C_2 with $C_1 + C_2 = C$ and $v(N_1, C_1) + v(N_2, C_2) = v(N, C)$.

Conquer

if $|N_1| \leq 1$ **then**
Combine $s(N_1, C_1)$ with the current solution set
if $|N_2| \leq 1$ **then**
Combine $s(N_2, C_2)$ with the current solution set

Recursion

if $|N_1| > 1$ **then** perform **D&C** (N_1, C_1)
if $|N_2| > 1$ **then** perform **D&C** (N_2, C_2)

(C1) **solve** (N, C):
DP bestimmt fr jeden Gewichtswert $c = 0, \ldots, C$
den optimalen Profitwert $v(N, c)$ in $O(|N| \cdot C)$ Zeit und $O(|N| + C)$ Speicher.

(C2) $|N| \leq 1$: $s(N, C)$ ist trivial.

(C3) Für jede Partition N_1, N_2 of N gibt es C_1, C_2,
sodass $C_1 + C_2 = C$ und $v(N_1, C_1) + v(N_2, C_2) = v(N, C)$.

Theorem:
Unter den Bedingungen (C1)–(C3) kann die optimale Lösungsmenge durch $\mathbf{D\&C}$ (N, C) in $O(|N| \cdot C)$ Zeit und $O(|N| + C)$ Speicher bestimmt werden.

Auf viele Probleme aus der Rucksackumgebung anwendbar.

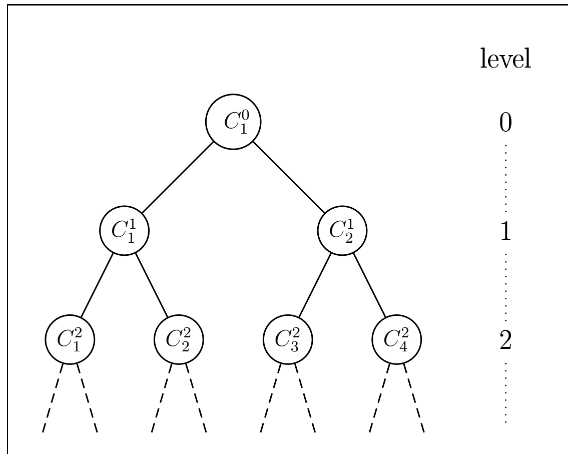


Fig. 3.4. A binary rooted tree representing the recursive structure of Recursive-DP by nodes

Zeit für $\mathbf{D\&C}$ (N', C') ohne Rekursion ist $O(|N'| \cdot C')$.

Betrachte den Binärbaum der rekursiven Aufrufe:
Ein Knoten hat *level* ℓ wenn es $\ell - 1$ Knoten am Pfad zur Wurzel gibt.
Die Wurzel hat level 0.
Ein Aufruf $\mathbf{D\&C}$ (N', C') in level ℓ hat $|N'| \approx |N|/2^\ell$.

Sei C_j^ℓ, $j = 1, \ldots, 2^\ell$, das Gewicht C' von Knoten j in level ℓ.
Wegen der Konstruktion von C_1 und C_2 gilt für jedes level ℓ

$$\sum_{j=1}^{2^\ell} C_j^\ell = C$$

Laufzeit für alle Knoten in level ℓ

$$\sum_{j=1}^{2^\ell} \frac{|N|}{2^\ell} \cdot C_j^\ell = \frac{|N|}{2^\ell} \cdot C$$

Summe über alle levels

$$\sum_{\ell=0}^{\log |N|} \frac{|N|}{2^\ell} \cdot C \leq 2|N| \cdot C$$

Dynamisches Programmieren mit Listen:

Statt fixem array betrachte Liste L von *Zuständen* / states:

Zustand (p, w) $\iff$ es gibt Lösung mit Profit p und Gewicht w.

sehr einfache Implementierung:

```
L = (0,0)
for i = 1 to n
    L' := L
    add (p_j, w_j) to every state in L'
    delete all states in L' with w > c
    L := merge (L, L')
end for
```

Dominanz:
Gilt für zwei Zustände $(p_1, w_1), (p_2, w_2)$:

$$p_1 \geq p_2 \text{ und } w_1 \leq w_2$$

$\implies$ Zustand 1 *dominiert* Zustand 2

Organisation der Listen L:

Liste von undominierten Zuständen in aufsteigender Reihenfolge von Profit und Gewicht.

merge (L, L'): einfaches Scannen beider Listen

Gesamtlaufzeit:

B: Anzahl nicht dominierter Zustände ($B \leq c$)

Laufzeit: $O(nB)$

worst-case: $O(nc)$

expected-case:

Theorem: [Beier, Vöcking, 2003]
Sind $p_j \sim U[0, 1]$ gleichverteilt und w_j *beliebig*, dann ist $B \in O(n^3)$.

$\implies O(n^4)$ erwartete Laufzeit!

3. Algorithmus auf Bit-Basis

Subset Sum Problem:

Spezialfall von (KP) mit $p_j = w_j$:

$$(SSP) \qquad \text{maximize} \quad \sum_{j=1}^{n} w_j x_j$$
$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_j \le c$$
$$x_j \in \{0,1\}, \quad j = 1, \dots, n.$$

Wort-Länge des Codierung einer Instanz sei W.
Alle W Bits können parallel in konstanter Zeit manipuliert werden.

Wenn $w_j = \ell W$ (ganzes Vielfaches von W):

$\Longrightarrow$ einfacher Shift:
$k-$tes Wort wird mit $(k+\ell)-$tem Wort oder-verknüpft.

Wenn $w_j = \ell W + r$:

$\Longrightarrow$ $k-$tes Wort wird oder-verknüpft mit
$(k-\ell)-$tes Wort right shift um r bits
$(k-\ell-1)-$tes Wort left shift um $W-r$ bits

Beispiel:

Da $W \in \Theta(\log c) \Longrightarrow$
Zeit: $O(nc/\log c)$ **Speicher:** $O(n + c/\log c)$ [Pisinger,'03]

auch auf Rucksackproblem übertragbar (viel mühsamer).

Transformation in (KP):

triviale Version:

b_j binäre Variablen für den item Typ

$\Longrightarrow \sum_{j=1}^{n} b_j$ items in (KP)

verbesserte Version:

Binärcodierung von x_j mit $n_j := \lfloor \log b_j \rfloor$

profit	p_j	$2p_j$	$4p_j$	$8p_j$	$\dots$	$2^{n_j-1}p_j$	$(n_j - 2^{n_j} + 1)p_j$
weight	w_j	$2w_j$	$4w_j$	$8w_j$	$\dots$	$2^{n_j-1}w_j$	$(n_j - 2^{n_j} + 1)w_j$

$$\sum_{j=1}^{n}(n_j+1) = n + \sum_{j=1}^{n} \lfloor \log b_j \rfloor \in O(n \log c) \text{ items in (KP)}$$

polynomial in the encoded input size.

Word RAM Algorithmus für SSP:

Beobachtung: Dynamisches Programmieren notiert nur,
welche Gewichte erreicht werden können $\Longrightarrow \{0,1\}$ Eintrag.

$$z_{j+1}(d) = 1 \Longleftrightarrow \quad z_j(d) = 1 \ \vee \ z_j(d - w_{j+1}) = 1$$

Bit-weise Darstellung des arrays $z_j(d)$

Elegante Möglichkeit der Update-Operation für item $j+1$:
Sei z_j^s right shift von z_j um w_{j+1} bits

$$z_{j+1} := z_j \text{ or } z_j^s$$

Beispiel: $c = 9$, $w_1 = 3$, $w_2 = 5$

d	0 1 2 3 4 5 6 7 8 9
z_1	1 0 0 1 0 0 0 0 0 0
z_1^s	0 0 0 0 0 1 0 0 1 0
z_1 or $z_1^s = z_2$	1 0 0 1 0 1 0 0 1 0

4. Bounded Knapsack Problem:

Variante von (KP) mit b_j Kopien von item Typ j:

$$(BKP) \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$
$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_j \le c$$
$$x_j \in \{0,1,\dots,b_j\}, \quad j = 1, \dots, n.$$

noch besser: Ausnützen der Problemstruktur

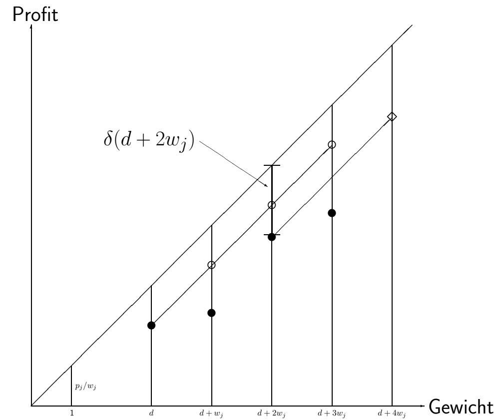
Ziel: Dynamisches Programmieren mit *einem* Durchgang pro item Typ.

Idee 1: Wenn item j ein update erzeugt,
d.h. $z_{j-1}(d) < z_{j-1}(d - w_j) + p_j =: z_j(d)$

$\Longrightarrow$ weiterer update-Versuch durch zweite Kopie an der Stelle
$z_j(d + w_j)$ bereits von $z_j(d)$ aus.

$\Longrightarrow$ *Folge von bis zu* b_j *update-Operationen*

Idee 2: Wenn b_j updates in Folge: was weiter?
Suche den *besten* Startpunkt zwischen $d + w_j$ und $d + b_j w_j$ für weitere
updates $\Longrightarrow$ Liste aller möglichen Startpunkte mitführen.



Einträge in der Liste:

$\delta(d) := d \ \ p_j/w_j - z_{j-1}(d)$

vertikaler Abstand (in Profit-Richtung) zwischen bisheriger Lösung und einer fiktiven Lösung nur aus item Typ j.

Position d mit minimalem $\delta(d)$ ist beste Startposition
für weitere Updates.

Sortierte Liste der $\delta(d)$ kann beim Updaten mitgeführt werden.

Listenoperationen benötigen insgesamt konstante Zeit pro item Typ.

$\Longrightarrow$ Gleiche Komplexität wie (KP)

Zeit: $O(nc)$ **Speicher:** $O(n+c)$

[Pferschy, 1999]

5. Die Championsleague:

aufbauend auf *core-Konzept*:

Vergleich Optimallösung x^* vs. LP-Relaxation x^{LP}:

x^* und x^{LP} unterscheiden sich nur an relativ wenigen Stellen.

Struktur der LP-Relaxation:
Sortiere items nach absteigender *Effizienz*:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$$

füge items in dieser Reihenfolge in den Rucksack ein.

$$\begin{aligned} x^{LP} &= [1,\ldots,1,|\ \underbrace{1,1,1,x,0,0,0}\ |,0,\ldots,0] \\ x^* &= [1,\ldots,1,|\ \underbrace{0,1,0,1,1,0,1}_{\text{core } C}\ |,0,\ldots,0] \end{aligned}$$

split item s mit $x_s^{LP} = x$

Variablen außerhalb des core sind fixiert.

offenes Restproblem: $|C|$ items, Kapazität $c - \sum_{j<C} w_j$.
viel kleiner, meist sehr schnell lösbar.

Core-Algorithmus, Grundschema:

```
choose C := {s − δ, ..., s + δ}
compute an upper bound U ≥ z*
solve the core-problem exactly
if Σ_{j<C} p_j + z*_C ≥ U stop, solution is optimal
else δ = 2 * δ, restart
alternative else
    solve complete problem exactly
stop.
```

Problem: Größe von C ist unbekannt!
Verschiedene Versuche für Wahl von C:
z.B. $|C| = 50, |C| = 100, |C| = \sqrt{n}$

Verbesserung:

Expanding Core Algorithm

Grundidee: [Pisinger, 1995]

Beginne mit sehr kleinem Core

Erweitere diesen sukzessive in beide Richtungen:

(i) Einfügen eines Elements (mit niedrigerer Effizienz)
$\Longrightarrow$ ein weiterer Schritt im dynamischen Programm

(ii) Entfernen eines Elements j (mit höherer Effizienz)
$\Longrightarrow$ DP mit $z(d) = \max\{z(d), z(d+w_j) - p_j\}$

Abbruchbedingung:

(1) Obere Schranke U erreicht

(2) Alle items im DP berücksichtigt

Champion 1: Algorithmus Combo

[Martello, Pisinger, Toth, 1999]

basierend auf dynamischem Programmieren mit Listen

Anwendung von oberen Schranken für jeden Zustand der Liste L:
für $(p, w) \in L$ wird zugehörige obere Schranke U'
für das Restproblem bestimmt.
Vergleich mit unterer Schranke z^ℓ (beste bisherige Lösung):
Wenn $U' \leq z^\ell \Longrightarrow$ Zustand streichen.

Versucht den betriebenen Aufwand
(= Verbesserung oberer und unterer Schranken)
an Schwierigkeit der Instanz anzupassen.

Messkriterium: Anzahl der undominierten Zustände Z

1. Starte mit sehr kleinem C
   Führe Expanding Core aus.
2. $|Z| \geq M_1$: Skaliere Gewichte & Kapazität durch ggT$(w_1, \ldots, w_n)$
3. $|Z| \geq M_2$: Bestimme verbesserte obere Schranke U durch Kardinalitätsbedingungen:
   $$\sum_{j=1}^{n} x_j \leq k, \text{ wobei } k := \min\{h \mid \sum_{j=1}^{h} w_j > c\} - 1$$
   bei aufsteigender Sortierung der Gewichte.
   Liefert in der LP-Relaxation verbesserte Schranken.
   untere Kardinalitätsschranken analog.
4. $|Z| \geq M_3$: Versuche bessere untere Schranken
   (= zulässige Lösungen) zu finden.
   Kombination von Zuständen der DP-Liste mit einzelnen "guten" items außerhalb des Core.
   $\Longrightarrow$ Chance zum Abbruch, falls obere Schranke erreicht.

Alternative Bestimmung des core:

Statt relativer "Qualität" eines items wird sein absoluter Beitrag gemessen.

Bestimme Wirkung von item j gegenüber der LP-Relaxation

Sei $r = p_s/w_s$ die Effizienz des split items:

$$loss(j) = |p_j - rw_j|$$

gibt Veränderung der Zielfunktion bei Austausch von split item mit item j an.

weiterer Vergleich zwischen Optimallösung und LP-Relaxation:
Setze $\Gamma := z^{LP} - z^*$ und $dif(x) := \{i \mid x_i \neq x_i^{LP}\}$

Es gilt für jede zulässige Lösung x:

$$z^{LP} - z(x) = r\left(c - \sum_{i=1}^{n} w_i x_i\right) + \sum_{i \in dif(x)} loss(i) \leq \Gamma$$

Champion 2:
[Beier, Vöcking, 2005]

Überlegung:

$$\sum_{i \in dif(x)} loss(i) \leq \Gamma \implies loss(i) \leq \Gamma, \forall i \in x$$

$\Longrightarrow$ nur items j mit $loss(j) \leq \Gamma$ kommen in der Lösung vor.

Beachte: Sortierung nach $loss$-Wert $\neq$ Sortierung nach Effizienz!

Algorithmus:

führe Expanding Core aus, wobei items nach steigendem $loss$-Wert vom split item ausgehend eingefügt werden.

Dynamisches Programmieren mit Listen

Start mit $\Gamma := z^{LP} - z^{\ell}$
bei jeder Verbesserung von z^{ℓ} auch Γ updaten

Zustände mit $loss > \Gamma$ eliminieren

Abbruchkriterium:
erstes item mit $loss > \Gamma$ gefunden.

Aufteilung der Berechnung auf zwei Listen
geschickte Balancierung:
- Liste 1: wächst am Anfang, mehr Einträge mit geringem $loss$
- Liste 2: erst später, weniger Einträge, höherer $loss$

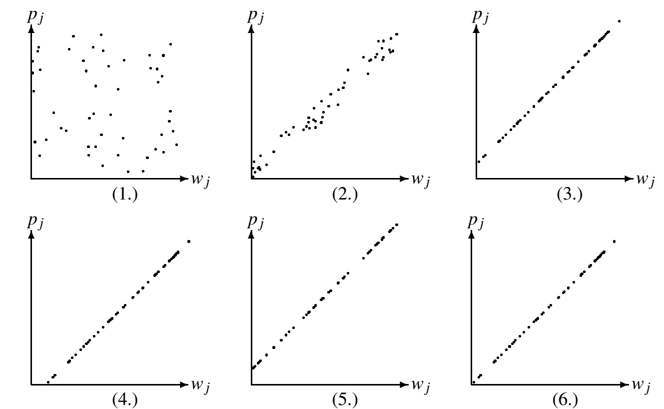
Testen von Rucksackalgorithmen

klassische Instanzen sehr gut lösbar

bei "einfacheren" Instanzen ist Champion 2 besser als Combo.

bei "komplizierten" Instanzen noch keine vollständigen Test, Combo teilweise besser.

5.5 Computational Experiments 1



Bemühungen "schwierige" Test-Instanzen zu erzeugen.

(i) sehr große Koeffizienten (64 Bit integers)

(ii) "kombinatorische" Struktur mit schlechten Schranken