Algorithmen und Datenstrukturen 2

Sommersemester 2007
11. Vorlesung

Peter F. Stadler

Universität Leipzig
Institut für Informatik
studla@bioinf.uni-leipzig.de

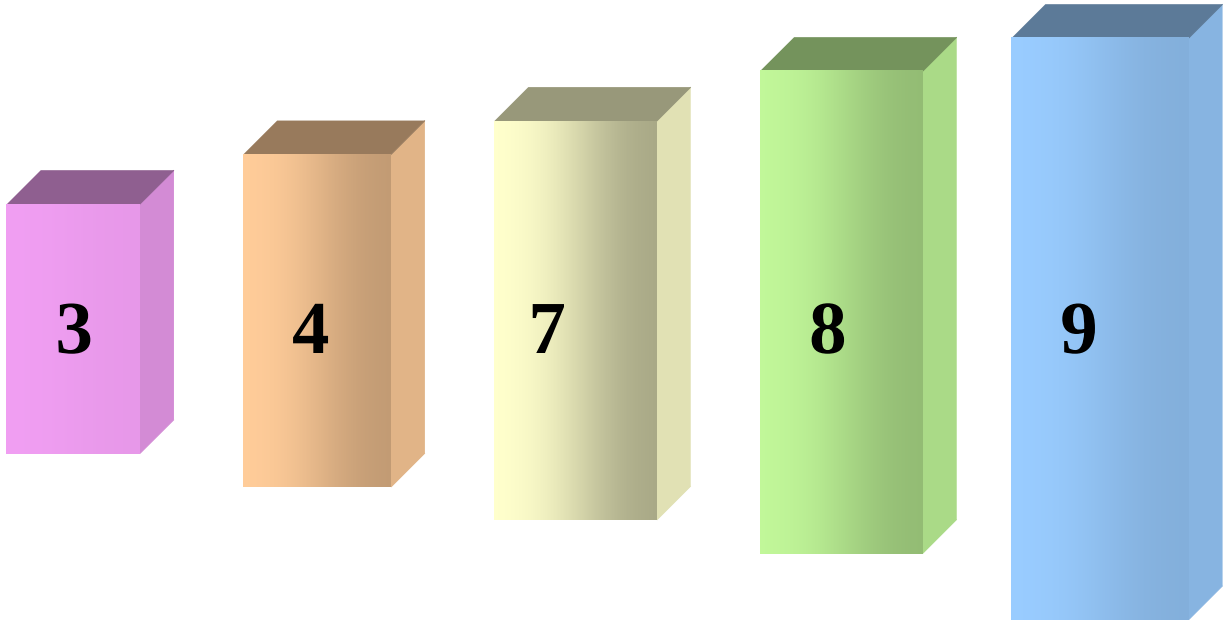
Das Rucksack-Problem

Ein Dieb, der einen Safe ausraubt, findet in ihm *N Typen* von Gegenständen unterschiedlicher Größe und unterschiedlichen Werts, hat aber nur einen kleinen Rucksack der *Größe M* zur Verfügung, um die Gegenstände zu tragen.

Das *Rucksack-Problem* besteht darin, diejenige Kombination von Gegenständen zu finden, die der Dieb für seinen Rucksack auswählen sollte, so dass der *Gesamtwert* der von ihm geraubten Gegenstände *maximal* wird.

Beispiel: Der Rucksack besitzt ein Fassungsvermögen von 17, der Safe enthält viele Gegenstände mit unterschiedlichen Größen und den angegebenen Werten.

Abbildung zum Rucksack-Problem



Wert	4	5	10	11	13
Bezeichnung	A	B	C	D	E

Der Dieb kann dann 5 Gegenstände A (jedoch nicht 6) mitnehmen, so dass die gesamte Beute den Wert 20 hat, oder er kann seinen Rucksack mit einem D und einem E füllen, was einen Gesamtwert von 24 ergibt, oder er kann andere Kombinationen ausprobieren. Doch für welche Kombination wird der *Gesamtwert maximal*?

Bedeutung des Rucksack-Problems auch im kommerziellen Bereich:

Z. B. ist es für eine Reederei von Interesse, die beste Möglichkeit zu kennen, wie ein Lastkraftwagen oder ein Transportflugzeug mit Gütern für die Verschiffung beladen werden kann.

Bei solchen Anwendungsfällen können auch andere Varianten des Problems auftreten: Es könnte z. B. sein, dass von jedem Gegenstand nur eine begrenzte Anzahl vorhanden ist.

Für viele solche Varianten ist der gleiche Ansatz geeignet.

Zur Lösung des Rucksack-Problems mit Hilfe der *dynamischen Programmierung* berechnet man die beste Kombination für alle Größen eines Rucksacks bis M.

Algorithmus

cost [i]: der größte Wert, der mit einem Rucksack mit dem Fassungsvermögen i erzielt werden kann

best [i]: das letzte Element, das hinzugefügt wurde, um das Maximum zu realisieren

- Zuerst berechnet man für *alle Größen des Rucksacks den maximalen Wert*, wenn nur Elemente vom *Typ A* verwendet werden.
- Danach berechnet man den maximalen Wert, wenn nur *Elemente A und B* verwendet werden.
- usw.

Die Lösung reduziert sich auf eine einfache Berechnung von cost [i].

Annahme: Auswahl des Elements j für den Rucksack, dann wäre der beste Gesamtwert, der erzielt werden könnte val [j] (für das Element) + cost [i - size [j]] (um den Rest des Rucksacks aufzufüllen) . Wenn dieser Wert den besten Wert übersteigt, der ohne ein Element j erreicht werden kann, aktualisiert man cost [i] und best [i]; andernfalls lässt man diese Größen unverändert.

Berechnung

Diese Berechnung kann in sehr effizienter Weise realisiert werden, indem die *Operationen in einer zweckmäßigen Reihenfolge* ausgeführt werden:

```
for ( j = 1 ;  j <=  N ; j ++ ) {
   for ( i = 1 ; i <= M ;  i++ ) {
      if  ( i >= size[j] ) {
         if ( cost[i] < cost[i - size[j]] + val[j] ) {
            cost[i] =  cost[i - size[j]] + val[j] ;
            best[i] = j ;
         }
      }
   }
}
```

Schleife über die Objekttypen

Schleife über die Größe des Rucksacks

Lösung des Rucksack-Beispiels

	k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j = 1																		
	cost [k]	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best [k]	-	-	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j = 2																		
	cost [k]	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best [k]	-	-	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j = 3																		
	cost [k]	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best [k]	-	-	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j = 4																		
	cost [k]	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best [k]	-	-	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j = 5																		
	cost [k]	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best [k]	-	-	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Erklärung zur Lösung des Beispiels

Das erste Zeilenpaar zeigt den maximalen Wert (den Inhalt der Felder cost und best), wenn nur Elemente A benutzt werden;

das zweite Zeilenpaar zeigt den maximalen Wert, wenn nur Elemente A und B verwendet werden, usw.

Der *höchste Wert*, der mit einem Rucksack der Größe 17 erreicht werden kann, *ist 24*.

Im Verlaufe der Berechnung dieses Ergebnisses hat man auch viele Teilprobleme gelöst, z. B. ist der größte Wert, der mit einem Rucksack der Größe 16 erreicht werden kann, 22, wenn nur Elemente A, B und C verwendet werden.

Der tatsächliche Inhalt des optimalen Rucksacks kann mit Hilfe des Feldes best berechnet werden. Per Definition ist best[M] in ihm enthalten, und der restliche Inhalt ist der gleiche wie im optimalen Rucksack der Größe

M - size [best [M]] usw.

Eigenschaften

Für die Lösung des Rucksack-Problems mit Hilfe der *dynamischen Programmierung* wird eine zu $N\,M$ proportionale Zeit benötigt.

Somit kann das Rucksack-Problem leicht gelöst werden, wenn M *nicht groß* ist; für große Fassungsvermögen kann die Laufzeit jedoch unvertretbar groß werden.

Eine *grundlegende Schwierigkeit* ist es, dass das Verfahren nicht anwendbar ist, wenn M *und die Größen oder Werte* z. B. *reelle Zahlen* anstatt ganzer Zahlen sind.

Wenn jedoch die Fassungsvermögen sowie die Größen und Werte der Gegenstände ganze Zahlen sind, so gilt das grundlegende Prinzip, dass *optimale Entscheidungen* nicht geändert werden müssen, nachdem sie einmal getroffen wurden.

Jedesmal, wenn dieses allgemeine Prinzip zur Anwendung gebracht werden kann, ist *die dynamische Programmierung anwendbar.*

Greedy-Algorithmus I

Greedy-Algorithmen sind mit dem *dynamischen Programmieren* verwandt, jedoch einfacher.

Die Grundsituation ist dieselbe:
Es geht um ein *Optimierungsproblem*; es soll sukzessiv eine optimale Lösung - in Bezug auf eine gegebene *Bewertungsfunktion* - konstruiert werden.

Während man beim *dynamischen Programmieren* solche optimale Lösung für alle kleineren Teilprobleme konstruiert und mit Hilfe dieser *in einer Tabelle eingetragenen Daten* die nächst größere optimale Lösung konstruiert, *verzichtet* man bei *Greedy auf die Buchführung* mittels einer Tabelle. Stattdessen wird der nächste Erweiterungsschritt zu einer (hoffentlich) optimalen Lösung lediglich auf Grund der lokal verfügbaren Informationen getätigt.

Greedy-Algorithmus I

Greedy-Algorithmen sind mit dem *dynamischen Programmieren* verwandt, jedoch einfacher.

Die Grundsituation ist dieselbe:
Es geht um ein *Optimierungsproblem*; es soll sukzessiv eine optimale Lösung - in Bezug auf eine gegebene *Bewertungsfunktion* - konstruiert werden.

Während man beim *dynamischen Programmieren* solche optimale Lösung für alle kleineren Teilprobleme konstruiert und mit Hilfe dieser *in einer Tabelle eingetragenen Daten* die nächst größere optimale Lösung konstruiert, *verzichtet* man bei *Greedy auf die Buchführung* mittels einer Tabelle. Stattdessen wird der nächste Erweiterungsschritt zu einer (hoffentlich) optimalen Lösung lediglich auf Grund der lokal verfügbaren Informationen getätigt.

Greedy-Algorithmus II

Annahme: Es gibt eine Gewichtsfunktion w, die die „*Güte“ einer Lösung* (auch einer Teillösung) misst. Es soll *eine Lösung mit maximalen w-Wert* konstruiert werden.

1. Starte mit der leeren Lösung.
2. Erweitere die bisher konstruierte Teillösung wie folgt:

> Wenn zur Erweiterung dieser Teillösung k Erweiterungsmöglichkeiten zur Verfügung stehen, die auf die vergrößerten *Teillösungen* $l_1, \ldots, l_k$ führen, wähle diejenige Teillösung l_i *mit* $w(l_i)$ *maximal*.

Der Name Greedy = gefräßig erklärt sich dadurch, dass ein Greedy-Algorithmus nach der Methode „*Nimm immer das größte Stück“* vorgeht.

Dieses Greedy-Prinzip kann in vielen Fällen funktionieren und tatsächlich auf eine *optimale Lösung* führen - ohne den Aufwand, der bei dynamischem Programmieren betrieben werden muss.

Teilmengensystem

Sei E eine endliche Menge, U eine Menge von Teilmengen von E.
(E,U) heißt *Teilmengensystem*, falls gilt:
1. $\{\} \in U$;
2. $A \subseteq B, B \in U \Rightarrow A \in U$

Das zu (E, U) gehörige *Optimierungsproblem* besteht darin, für eine beliebige *Gewichtsfunktion* $w : E \rightarrow Z$ eine in U maximale Menge T *(bzgl.* $\subseteq$ *)* zu finden, deren *Gesamtgewicht*

$$w(T) = \sum_{e \in T} w(e)$$

maximal ist.

Kanonischer Greedy-Algorithmus

Der einem Teilmengensystem (E, U) und Gewichtsfunktion $w : E \rightarrow Z$ zugeordnete *kanonische Greedy-Algorithmus* für diese Aufgabe arbeitet wie folgt:

Ordne alle Elemente in E nach absteigendem Gewicht:

$w(e_1) \geq \ldots \geq w(e_n)$

T = ∅;

for (k = 1; k <= n; k++)

  if (T ∪ {e_k} ∈ U)

    T = T ∪ {e_k};

Ausgabe der Lösung T;

(Soll alternativ eine maximale Menge in U mit minimalem Gewicht gefunden werden, so ordne man zu Beginn die Elemente von E nach aufsteigendem Gewicht).

Dieser Algorithmus liefert allerdings nicht immer die optimale Lösung.

Beispiel

Sei $E = \{ e_1, e_2, e_3 \}$

$U = \{ \varnothing, \{e_1\}, \{e_2\}, \{e_3\}, \{e_2,e_3\} \}$

mit $w(e_1) = 3$

$w(e_2) = w(e_3) = 2$

Dann liefert der *kanonische Greedy-Algorithmus* die Lösung

$T = \{e_1\}$ mit $w(T) = 3$,

während die optimale Lösung dagegen $T' = \{e_2,e_3\}$ mit $w(T') = 4$ ist.

Austauscheigenschaft

Satz: Sei (E,U) ein Teilmengensystem. Der kanonische *Greedy-Algorithmus* liefert für das zugehörige Optimierungsproblem (in Bezug auf jede beliebige Gewichtsfunktion w : E $\rightarrow$ Z) die *optimale Lösung, genau dann wenn (E,U) ein Matroid ist.*

Diese algebraische Struktur verlangt ein *zusätzliches Axiom* – die *Austauscheigenschaft*:

$$A, B \in U, |A| < |B| \Rightarrow \exists x \in B - A : A \cup \{ x \} \in U$$

Das obige Beispiel stellt kein Matroid dar, denn die Austauscheigenschaft ist bei

$A = \{ e_1 \}$ und $B = \{ e_2 , e_3 \}$ verletzt.

Dijkstra-Algorithmus I

Berechnung des kürzesten Weges.

Beweis der Optimalität der Lösung:

a) zugrundeliegende matroide Struktur zeigen
b) zeigen, dass kanonischer Greedy, angesetzt auf die matroide Struktur und eine Gewichtsfunktion w, die gleichen Lösungen berechnet, wie der Dijkstra-Algorithmus

a) matroide Struktur
   i) Teilmengensystem (E,U)
      - Grundmenge E: Menge aller Pfade (zyklenfrei) vom Startknoten s ausgehend
      - Menge der Lösungen U: Menge von Pfadmengen über E, wobei für jede einzelne Pfadmenge gilt, dass die enthaltenen Pfade auf *verschiedene* Endknoten führen.

      Es gilt:

      $\emptyset \in U$ sowie $A \subseteq B, B \in U \Rightarrow A \subseteq U$ also (E,U) ist Teilmengensystem

Dijkstra-Algorithmus II

ii) Austauscheigenschaft

Seien A, B $\in$ U zwei Pfadmengen mit |A| < |B|.

Dann gibt es genau |A| verschiedene Endknoten in A und in B befindet sich mindestens ein Pfad mit einem weiteren Endknoten, der nicht in A vorkommt.

Dieser Pfad kann zu A hinzugefügt werden und das Resultat liegt immer noch in der Menge U.

$\Rightarrow$ (E,U) ist Matroid

b) kanonischer Greedy hat gleiche Lösungen wie Dijkstra

kanonischer Greedy:

Ordne die Elemente nach aufsteigendem (bzw. absteigenden) Gewicht:

$w(e_1) \leq ... \leq w(e_n)$

```
T = ∅;
for (k = 1; k <= n; k++)
   if (T ∪ {e_k} ∈ U)   { T = T ∪ {e_k}; }
```

Dijkstra-Algorithmus III

Liege dem kanonischen Greedy-Algorithmus das dargestellte Teilmengensystem (E,U) zugrunde.

Die Gewichtsfunktion auf der Grundmenge, also auf den von Startknoten s ausgehenden Pfaden, sei die Summe der Längen der Kanten, die auf dem Pfad liegen:

$$w'(p) = \Sigma_{k \text{ liegt auf } p}\ w(k)$$

Der kanonische Greedy-Algorithmus für dieses Matroid und diese Gewichtsfunktion w' erzeugt in seinem Ablauf exakt die gleichen Ergebnisse wie der Dijkstra-Algorithmus.

(Der Dijkstra-Algorithmus ist nur effizienter, da man nicht alle von s ausgehenden Pfade erzeugen und anschließend nach aufsteigenden w‘-Werten sortieren muss.)

Auftragsplanung

Gegeben:

- Menge A von n "*Aufträgen*", jeweils in einer Zeiteinheit zu bewältigen,
- zu jedem *Auftrag i* gibt es *Gewinn* p_i und *Termin* d_i, an dem er abgeschlossen sein muss (sonst kein Gewinn)

Gesucht: Menge M von Aufträgen, so dass

a) M sich so sortieren lässt, dass jeder Auftrag vor seinem Abschlusstermin erledigt wird und

b) der Gesamtgewinn maximal ist.

Beispiel

Auftragsmenge E = {a, b, c, d, e}

Wert	**Termin**
$p_a = 13,$	$d_a = 2$
$p_b = 7,$	$d_b = 1$
$p_c = 9,$	$d_c = 1$
$p_d = 3,$	$d_d = 2$
$p_e = 5,$	$d_e = 1$

Lösung	**Gewinn**
∅	0
a	13
b	7
c	9
d	3
e	5
a,b	20
a,c	22
a,d	16
a,e	18
b,d	10
c,d	12
d,e	8

Greedy Lösung

Ordne Aufträge e_k absteigend nach Gewinn: $p_1 \geq ... \geq p_n$

```
T = ∅;
for (k = 1; k <= n; k++)
  if ( T∪{e_k} ist eine zulässige Lösung )
    T = T ∪ {e_k};
```

Ausgabe der Lösung `T;`

In $T=\{e_1, e_2, ..., e_i\}$ seien die Aufträge nach ihrem Abschlusstermin geordnet:

$d_1 \leq d_2 \leq ... \leq d_i$

Lösung T ist genau dann zulässig, wenn gilt:

$d_1 \geq 1, d_2 \geq 2, ..., d_i \geq i$