Kap. 7.3 Enumerationsverfahren

Kap. 7.4 Branch-and-Bound

Kap. 7.5 Dynamische Programmierung

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

ACHTUNG: Die VO am Dienstag, dem **21.7. entfällt**

24. VO (vorletzte) DAP2 SS 2009 16. Juli 2009

DAP2-Klausur

- **Stoff der Klausur:** Stoff der Vorlesung und der Übungen (*nicht: Kap. 3.3: Externe Sortierverfahren*),
  - für die Klausur am 31. Juli 2009 gilt: *nicht Kapitel 7: Optimierung*
- **Zeit:** 10:15 Uhr – 11:45 Uhr (90 Minuten)
  - bitte ab 10:00 Uhr vor Ort sein
- **Wo?** Für die Hörsaaleinteilung beachten Sie bitte die Information auf den Web-Seiten

Anmeldung für IKT, ET/IT bei Problemen mit BOSS

- bitte mit Anmeldeformular bis spätestens 24.7. anmelden
- dabei: Studienrichtung auswählen (aber nur aus systemischen Gründen)

Überblick

- Enumerationsverfahren
  - Beispiel: 0/1-Rucksackproblem
- Branch-and-Bound
  - Beispiel: 0/1-Rucksackproblem
  - Beispiel: ATSP
- Dynamische Programmierung
  - Beispiel: 0/1-Rucksackproblem

Kap. 7.3: Enumerationsverfahren

Enumerationsverfahren

- Exakte Verfahren, die auf einer vollständigen Enumeration beruhen
- Eignen sich für kombinatorische Optimierungs-probleme: hier ist die Anzahl der zulässigen Lösungen endlich.

Definition: Kombinatorisches Optimierungsproblem

Gegeben sind:

- endliche Menge E (Grundmenge)
- Teilmenge I der Potenzmenge 2^{E} von E (zulässige Mengen)
- Kostenfunktion c: $\mathrm{E} \rightarrow \mathrm{K}$

Gesucht ist: eine Menge $I^* \in I$, so dass

$$c(I^*) = \sum_{e \in I^*} c(e)$$

so groß (klein) wie möglich ist.

Enumerationsverfahren

„Exhaustive Search“

- **Idee:** Enumeriere über die Menge aller zulässigen Lösungen und bewerte diese mit der Kostenfunktion
- Die am besten bewertete Lösung ist die optimale Lösung.

- **Problem:** Bei NP-schwierigen OP ist die Laufzeit dieses Verfahrens nicht durch ein Polynom in der Eingabegröße beschränkt (sondern exponentiell).

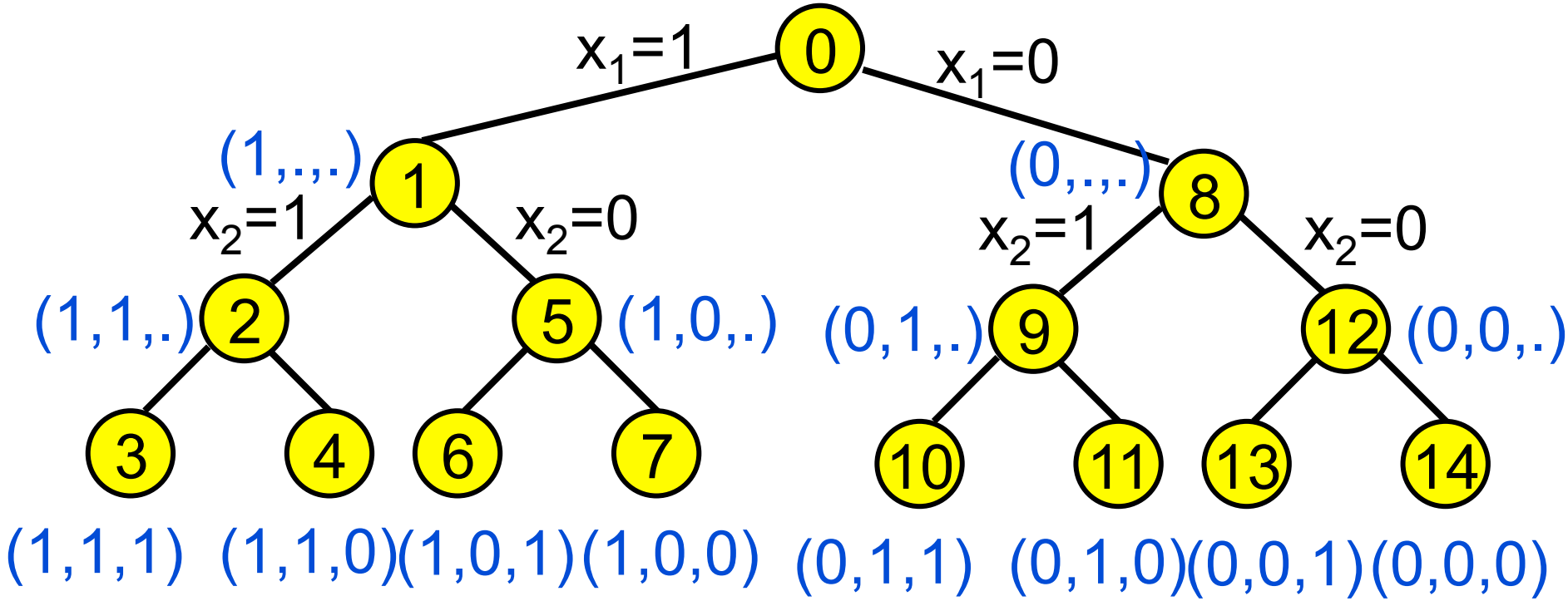
Enumerationsverfahren für 0/1-Rucksackproblem

- **Idee:** Enumeriere über alle Teilmengen einer N-elementigen Menge.

- Für alle Gegenstände i=1…N:
  - Fixiere zusätzlich zu den bisherigen auch noch Gegenstand i: Löse das kleinere Problem, das nur noch aus N-i Gegenständen besteht.

Divide & Conquer-Prinzip

Bsp: Enumeration durch Backtracking: Suchbaum für N=3



„Backtracking-Verfahren“

Realisierung der Enumeration für 0/1-Rucksackproblem

- x: Lösungsvektor mit $x_i=1$ genau dann wenn Gegenstand i eingepackt wird, und $x_i=0$ sonst
- z: Anzahl der bereits fixierten Variablen in x
- xcost: Gesamtkosten (Wert) der Lösung x
- xweight: Gesamtgewicht der Lösung x

- Enum(z,xcost,xweight,x)
- Aufruf: Enum(0,0,0,x)

Algorithmus: Enum(z,xcost, xweight,x)

```
(1)  if xweight ≤ K then {
(2)      if xcost > maxcost then {
(3)          maxcost:=xcost; bestx:=x
(4)      }
(5)      for i:=z+1,…,N do {
(6)          x[i]:=1
(7)          Enum(i, xcost+c[i], xweight+w[i], x)
(8)          x[i]:=0
(9)  } }
```

Enumerationsverfahren: Beispiel

Reihenfolge der ausgewerteten Rucksäcke für $n=3$:

- 0 0 0
- 1 0 0
- 1 1 0
- 1 1 1
- 1 0 1
- 0 1 0
- 0 1 1
- 0 0 1

Diskussion Enumerationsverfahren

- Laufzeit: $O(2^N)$

- Wegen der exponentiellen Laufzeit sind Enumerationsverfahren i.A. nur für kleine Instanzen geeignet. Bereits für $N \geq 50$ ist das Verfahren nicht mehr praktikabel.

Diskussion Enumerationsverfahren

- **Zusatzidee:** k-elementige Mengen, die nicht in den Rucksack passen, müssen nicht aufgezählt werden

- In vielen Fällen kann die Anzahl der ausprobierten Möglichkeiten deutlich verringert werden.

- Systematische Verkleinerung des Suchraumes bieten Branch-and-Bound Algorithmen

Kap. 7.4: Branch-and-Bound

Branch-and-Bound

- **Idee:** Eine spezielle Form der beschränkten Enumeration, die auf dem Divide&Conquer Prinzip basiert.

- Frühes Ausschließen ganzer Lösungsgruppen in der Enumeration durch „Bounding“.

- Man sagt: L ist eine **untere Schranke** für eine Instanz P eines OPs, wenn für den optimalen Lösungswert gilt: $c_{opt}(P) \geq L$ ist. U ist eine **obere Schranke**, wenn gilt $c_{opt}(P) \leq U$.

Branch-and-Bound

- **Idee:** An den entstehenden Knoten des Suchbaums wird geprüft, ob die dazugehörige Teillösung x' weiterverfolgt werden muss.

- Hierzu wird an jedem Knoten eine untere Schranke L der Teillösung (lower bound) und eine obere Schranke U (upper bound) berechnet.

- L (für Max.probleme) kann mit einer beliebigen Heuristik (z.B. Greedy) berechnet werden

- U (für Max.probleme) gibt eine Schranke für den besten Wert an, der von x' erreicht werden kann.

Gerüst für Branch-and-Bound Algorithmen für Maximierungsprobleme

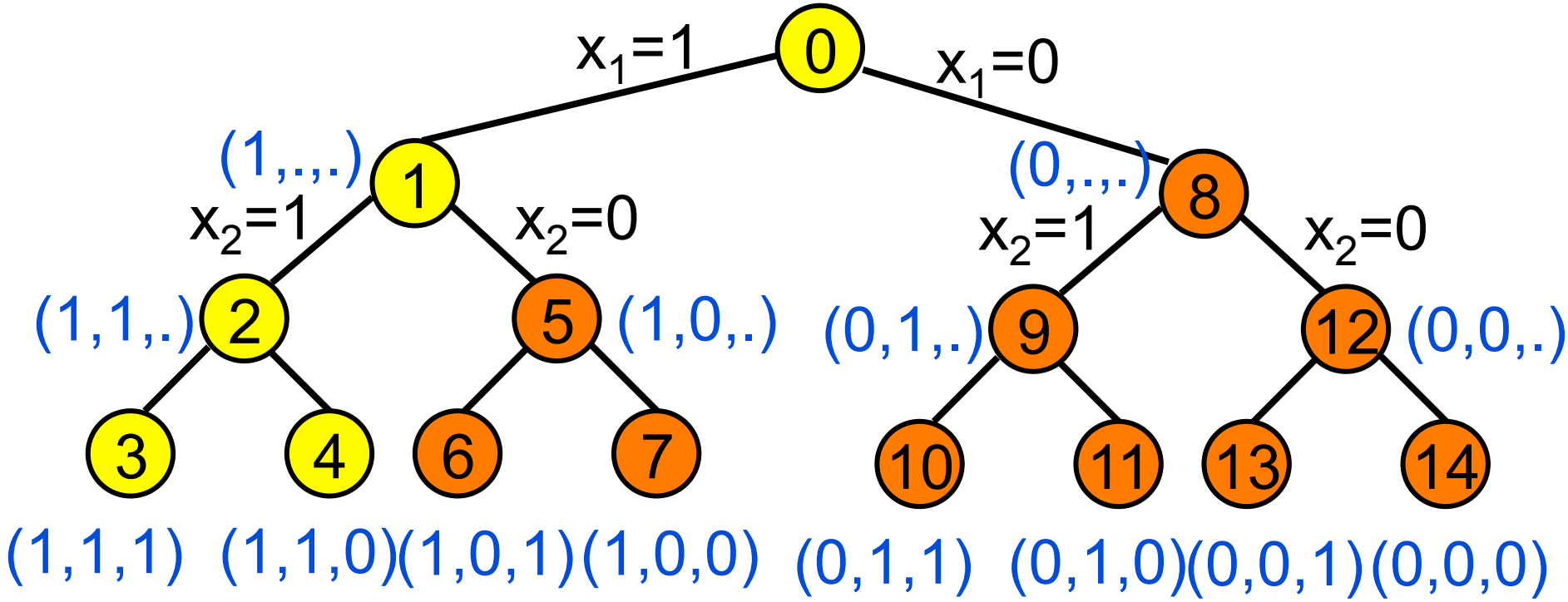
- Berechne eine zulässige Startlösung mit Wert L und eine obere Schranke U für alle möglichen Lösungen.
- Falls $L = U \rightarrow$ STOP: gefundene Lösung ist optimal.

- **Branching:** Partitioniere die Lösungsmenge (in zwei oder mehr Teilprobleme).

- **Search:** Wähle eines der bisher erzeugten Teilprobleme.

- **Bounding:** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei $L:=\max(L,L_i)$ der Wert der besten bisher gefundenen Lösung. Falls $U_i \leq L$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Bsp: Enumeration durch Backtracking: Suchbaum für N=3



Mittels Branch-and-Bound versucht man ganze Teilbäume im Suchbaum abzuschneiden ohne diese auswerten zu müssen

Beispiel: Branch-and-Bound für 0/1- Rucksackproblem

- Berechnung einer zulässigen Startlösung:
- z.B. Greedy-Heuristik $\rightarrow$ L

- Berechnung einer oberen Schranke U: später!

Beispiel: Branch-and-Bound für 0/1- Rucksackproblem

- **Branching:** Wir zerlegen das Problem in zwei Teilprobleme: Wir wählen Gegenstand i und wählen es für Teilproblem T_{1i} aus (d.h. $x_i=1$) und für T_{2i} explizit nicht aus (d.h. $x_i=0$).
- Für T_{1i} muss das Gewicht von Gegenstand i von der Rucksackgröße abgezogen werden und der Wert zum Gesamtwert hinzuaddiert werden.
- Danach streichen wir Gegenstand i aus unserer Liste.

Branch-and-Bound für 0/1-Rucksackproblem

- **Search:** Wähle eines der bisher erzeugten Teilprobleme, z.B. dasjenige mit der besten (größten) oberen Schranke (wir hoffen, dass wir hier die beste Lösung finden werden).

- **Bounding:** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei $L:=\max(L,L_i)$ der Wert der besten bisher gefundenen Lösung. Falls $U_i \leq L$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Berechnung einer oberen Schranke für 0/1-Rucksackproblem

- Sortierung nach Nutzen f_i := Wert c_i, / Gewicht w_i
- Seien $i_1,...,i_n$ die in dieser Reihenfolge sortierten Gegenstände
- Berechne das maximale r mit $w_1+\ldots+w_r \leq K$
- Gegenstände $1,\ldots,r$ werden eingepackt (also $x_i=1$ für $i=1,\ldots,r$)
- Danach ist noch Platz für $K-(w_1+\ldots+w_r)$ Einheiten an Gegenständen. Diesen freien Platz „füllen“ wir mit $(K-(w_1+\ldots+w_r)) / w_{r+1}$ Einheiten von Gegenstand $r+1$ auf.

Behauptung: Es existiert keine bessere Lösung!

Berechnung einer oberen Schranke für 0/1-Rucksackproblem

- **Begründung:** Die ersten r Gegenstände mit dem höchsten Nutzen pro Einheit sind im Rucksack enthalten. Und zwar jeweils so viel davon, wie möglich ist.
- **Achtung:** vom letzten Gegenstand $r+1$ wird eventuell ein nicht-ganzzahliger Teil eingepackt
- deswegen ist die generierte Lösung i.A. nicht zulässig (sie erfüllt die Ganzzahligkeitsbedingung i.A. nicht)
- **Aber es gilt:** der berechnete Lösungswert der Gegenstände im Rucksack ist mindestens so groß wie die beste Lösung.

Berechnung einer oberen Schranke für unser Beispiel

K=17

Gegenstand	a	b	c	d	e	f	g	h
Gewicht	3	4	4	6	6	8	8	9
Wert	3	5	5	10	10	11	11	13
Nutzen	1	1,25	1,25	1,66	1,66	1,37	1,37	1,44

- Sortierung nach Nutzen:= Wert $c_{i,}$ / Gewicht $w_{i:}$ d,e,h,f,g,b,c,a
- Wir packen also in den Rucksack: $x_d=x_e=1$
- Platz frei für 5 Einheiten: $x_h = 5/9$ Einheiten dazu
- Wert: 10+10+5/9(13)<20+7,3=27,3
- Obere Schranke für beste Lösung: 27

Berechnung einer oberen Schranke für 0/1-Rucksackproblem

- FALL: Es besteht bereits eine Teillösung, denn wir befinden uns mitten im Branch-and-Bound Baum.
- Betrachte jeweils die aktuellen Teilprobleme, d.h. sobald ein Gegenstand als „eingepackt“ fixiert wird, wird K um dessen Gewicht reduziert. Falls $x_i=0$ fixiert wird, dann wird der Gegenstand aus der Liste der Gegenstände gestrichen.
- Berechnung der oberen Schranke auf den noch nicht fixierten Gegenständen: wie vorher

Branch-and-Bound für ATSP

- ATSP: wie TSP, jedoch auf gerichtetem Graphen: die Kantenkosten sind hier nicht symmetrisch: $c(u,v)$ ist i.A. nicht gleich $c(v,u)$.

Branch-and-Bound für ATSP

Asymmetrisches Handlungsreisendenproblem, ATSP

- **Gegeben:** Vollständiger **gerichteter Graph** $G=(V,E)$ mit Kantenkosten c_e (geg. durch Distanzmatrix zwischen allen Knotenpaaren)

- **Gesucht:** Tour T (Kreis, der jeden Knoten genau einmal enthält) mit minimalen Kosten $c(T)=\sum c_e$

I.A. gilt hier $c(u,v) \neq c(v,u)$

Gerüst für Branch-and-Bound Algorithmen für Minimierungsprobleme

- Berechne eine zulässige Startlösung mit Wert U und eine untere Schranke L für alle möglichen Lösungen.
- Falls $L = U \rightarrow$ STOP: gefundene Lösung ist optimal.

- **Branching:** Partitioniere die Lösungsmenge (in zwei oder mehr Teilprobleme).

- **Search:** Wähle eines der bisher erzeugten Teilprobleme.

- **Bounding:** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei U der Wert der besten bisher gefundenen Lösung. Falls $U \leq L_i$ braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Gerüst für Branch-and-Bound Algorithmen für ATSP

- **Branching:** Partitioniere die Lösungsmenge, indem jeweils für eine ausgewählte Kante $e=(u,v)$ zwei neue Teilprobleme erzeugt werden:
- T_{1e}: (u,v) wird in Tour aufgenommen; in diesem Fall können auch alle anderen Kanten (u,w) und (w,v) für alle $w \in V$ ausgeschlossen werden.
- T_{2e}: (u,v) wird nicht in Tour aufgenommen.

Gerüst für Branch-and-Bound Algorithmen für ATSP

- **Bounding:** Berechne untere Schranke, z.B. durch:
- Für alle Zeilen u der Distanzmatrix berechne jeweils das Zeilenminimum.
- Denn: Zeile u korrespondiert zu den von u ausgehenden Kanten. In jeder Tour muss u genau eine **aus**gehende Kante besitzen: d.h. ein Wert in Zeile u wird auf jeden Fall benötigt.

ATSP-Beispiel

$$D=\begin{pmatrix} \infty & 5 & \textcircled{1} & 2 & 1 & 6 \\ 6 & \infty & 6 & 3 & 7 & \textcircled{2} \\ \textcircled{1} & 4 & \infty & 1 & 2 & 5 \\ 4 & \textcircled{3} & 3 & \infty & 5 & 4 \\ 2 & 5 & \textcircled{1} & 2 & \infty & 5 \\ 6 & \textcircled{2} & 6 & 4 & 5 & \infty \end{pmatrix}$$

○ Zeilenminimumsumme := $1+2+1+3+1+2 = 10$

alternativ: Spaltenminimumsumme := $1+2+1+1+1+2 = 8$

Gerüst für Branch-and-Bound Algorithmen für ATSP

- **Bounding:** Berechne untere Schranke, z.B. durch:
- Für alle Zeilen u der Distanzmatrix berechne jeweils das Zeilenminimum.
- Denn: Zeile u korrespondiert zu den von u ausgehenden Kanten. In jeder Tour muss u genau eine **aus**gehende Kante besitzen: d.h. ein Wert in Zeile u wird auf jeden Fall benötigt.
- Dasselbe kann man für alle Spalten machen: In jeder Tour muss u genau eine **ein**gehende Kante besitzen.
- L := max (Zeilenminsumme, Spaltenminsumme)

Kap. 7.4 Dynamische Programmierung

Dynamische Programmierung

Dynamic Programming, DP

- **Idee:** Zerlege das Problem in kleinere Teil-probleme P_i ähnlich wie bei Divide & Conquer.
- Allerdings: die P_i sind hier abhängig voneinander (im Gegensatz zu Divide & Conquer)
- Dazu: Löse jedes Teilproblem und speichere das Ergebnis E_i so ab dass E_i zur Lösung größerer Probleme verwendet werden kann.

Dynamische Programmierung

Allgemeines Vorgehen:

- Wie ist das Problem sinnvoll einschränkbar bzw. zerlegbar? Definiere den Wert einer optimalen Lösung rekursiv.

- Bestimme den Wert der optimalen Lösung „bottom-up“.

Beispiel: All Pairs Shortest Paths

All-Pairs Shortest Paths (APSP)	
Gegeben:	gerichteter Graph G = (V,A) Gewichtsfunktion w : A!R_0^+
Gesucht:	ein kürzester Weg von u nach v für jedes Paar $u, v \in V$

Algorithmus von Floyd-Warshall

Idee: Löse eingeschränkte Teilprobleme:

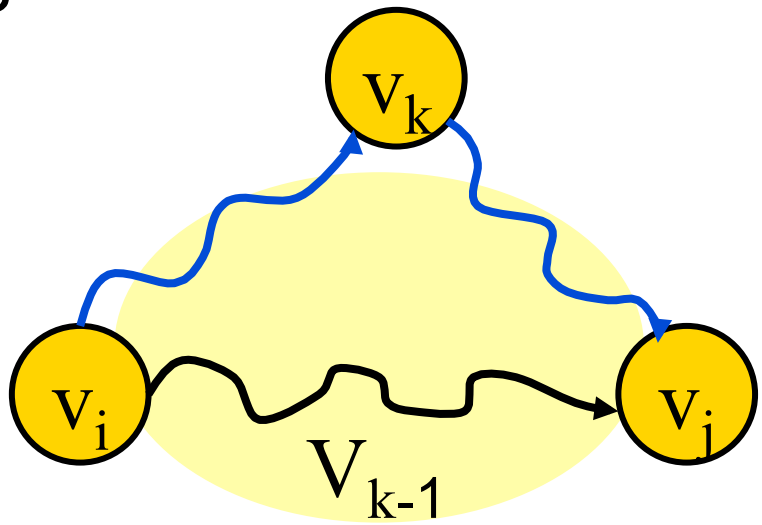
- Sei $V_k := \{v_1, \ldots, v_k\}$ die Menge der ersten k Knoten
- Finde kürzeste Wege, die nur Knoten aus V_k als Zwischenknoten benutzen dürfen
- Zwischenknoten sind alle Knoten eines Weges außer die beiden Endknoten
- Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Weges von v_i nach v_j, der nur Knoten aus V_k als Zwischenknoten benutzt

Berechnung von $d_{ij}^{(k)}$

Berechnung von $d_{ij}^{(k)}$:

- $d_{ij}^{(0)} = w(v_i, v_j)$
- Bereits berechnet: $d_{ij}^{(k-1)}$ für alle $1 \leq i,j \leq n$

Für einen kürzesten Weg von v_i nach v_j gibt es zwei Möglichkeiten:



Berechnung von $d_{ij}^{(k)}$

Zwei Möglichkeiten für kürzesten Weg von v_i nach v_j:

- Der Weg p benutzt v_k nicht als Zwischenknoten: dann ist $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
- Der Weg p benutzt v_k als Zwischenknoten: dann setzt sich p aus zwei kürzesten Wegen über v_k zusammen, die jeweils nur Zwischenknoten aus V_{k-1} benutzen: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

- $d_{ij}^{(k)} := \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
- **Bellmansche Optimalitätsgleichung:** der Wert einer optimalen Lösung eines Problems wird als einfache Funktion der Werte optimaler Lösungen von kleineren Problemen ausgedrückt.

Algorithmus von Floyd-Warshall

ist dynamische Programmierung:

(1) **for** i:=1 **to** n **do**

(2) **for** j:=1 **to** n **do**

(3) $d_{ij}^{(0)} := w(v_i, v_j\}$

(4) } }

(5) **for** k:=1 **to** n **do**

(6) **for** i:=1 **to** n **do**

(7) **for** j:=1 **to** n **do**

(8) $d_{ij}^{(k)} := \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

(9) } } }

DP für 0/1-Rucksackprobleme

Rucksackproblem HIER:

- **Geg.:** n Gegenstände mit Größe (Gewicht) w_i und Wert c_i, und ein Rucksack der Größe (Gewichtslimit) K, wobei $w_i, c_i, K \in Z$.

- **Gesucht:** Menge der in den Rucksack gepackten Gegenstände mit maximalem Gesamtwert; dabei darf das Gesamtgewicht den Wert K nicht überschreiten.

ACHTUNG HIER: Alle Eingabewerte sind ganzzahlig!

DP für 0/1-Rucksackproblem

Wie können wir ein Rucksackproblem sinnvoll einschränken?

- Betrachte nur die ersten k der Objekte
- **Frage:** Was nützt uns die optimale Lösung des eingeschränkten Problems für die Lösung des Problems mit $k+1$ Objekten?
- **Lösung:** Variiere die Anzahl der betrachteten Objekte und die Gewichtsschranke

Dynamische Programmierung für das 0/1-Rucksackproblem

- **Def.:** Für ein festes $i \in \{1,\ldots,n\}$ und ein festes $W \in \{0,\ldots,K\}$ sei **R(i,W)** das eingeschränkte Rucksackproblem mit den ersten i Objekten mit Gewichten w_j, Werten c_j $(j=1,\ldots,i)$ und Gewichtslimit W.
- Wir legen eine Tabelle $T(i,W)$ an für die Werte $i=0,\ldots,n$ und $W=0,\ldots,K$, wobei an Position $T(i,W)$ folgende Werte gespeichert werden:
- $F(i,W)$: optimaler Lösungswert für Problem $R(i,W)$
- $D(i,W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dynamische Programmierung für das 0/1-Rucksackproblem

- $D(i,W)=1$: wenn es in $R(i,W)$ optimal ist, das i-te Element einzupacken
- $D(i,W)=0$: sonst

- $F(n,K)$ ist dann der Wert einer optimalen Rucksackbepackung für unser Originalproblem

- $F(i,W)$: optimaler Lösungswert für Problem $R(i,W)$
- $D(i,W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert $F(i,W)$ aus bereits bekannten Lösungen der Probleme $F(i-1,W')$ berechnen?

$$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$$

entweder i-tes Element wird eingepackt: dann muss ich die optimale Lösung betrachten, die in $R(i-1,W-w_i)$ gefunden wurde.

oder i-tes Element wird nicht eingepackt: dann erhält man den gleichen Lösungswert wie bei $R(i-1,W)$.

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert $F(i,W)$ aus bereits bekannten Lösungen der Probleme $F(i-1,W')$ berechnen?

$$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$$

Bellmansche Optimalitätsgleichung

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert $F(i,W)$ aus bereits bekannten Lösungen der Probleme $F(i-1,W')$ berechnen?

$$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$$

Randwerte:

- $F(0,W) := 0$ für alle $W:=0,\ldots,K$
- Falls $W<w_i$, dann passt Gegenstand i nicht mehr in den Rucksack für das eingeschränkte Problem mit Gewicht $W \leq K$.

Algorithmus DP für Rucksackproblem

```
(1) for W:=0 to K do F(0,W) := 0        // Initialisierung
(2) for i:=1 to n do {
(3)    for W:=0 to w_i-1 do   // Gegenstand i zu groß
(4)        F(i,W) := F(i-1,W); D(i,W) := 0
(5)    for W:=w_i to K do {
(6)        if F(i-1,W-w_i)+c_i > F(i-1,W) then {
(7)            F(i,W):= F(i-1,W-w_i)+c_i); D(i,W) := 1 }
(8)        else { F(i,W):= F(i-1,W);          D(i,W) := 0 }
(9) } }
```

Beispiel:

K=9

n=7

i	1	2	3	4	5	6	7
c_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

i \ W	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Frage: Wie können wir aus dem optimalen Lösungswert die Lösung berechnen?

Wir haben in $T[i,W]$ gespeichert:

- $F(i,W)$ und $D(i,W)$
- Wir wissen: $F(n,K)$ enthält den optimalen Lösungswert für unser Originalproblem

Für Problem $R(n,K)$: Starte bei $F(n,K)$:

- Falls $D(n,K)=0$, dann packen wir Gegenstand n nicht ein. Gehe weiter zu Problem $R(n-1,K)$.
- Falls $D(n,K)=1$, dann packen wir Gegenstand n ein. Damit ist das für die ersten n-1 Gegenstände erlaubte Gewichtslimit $K-w_n$. Gehe weiter zu Problem $R(n-1,K-w_n)$

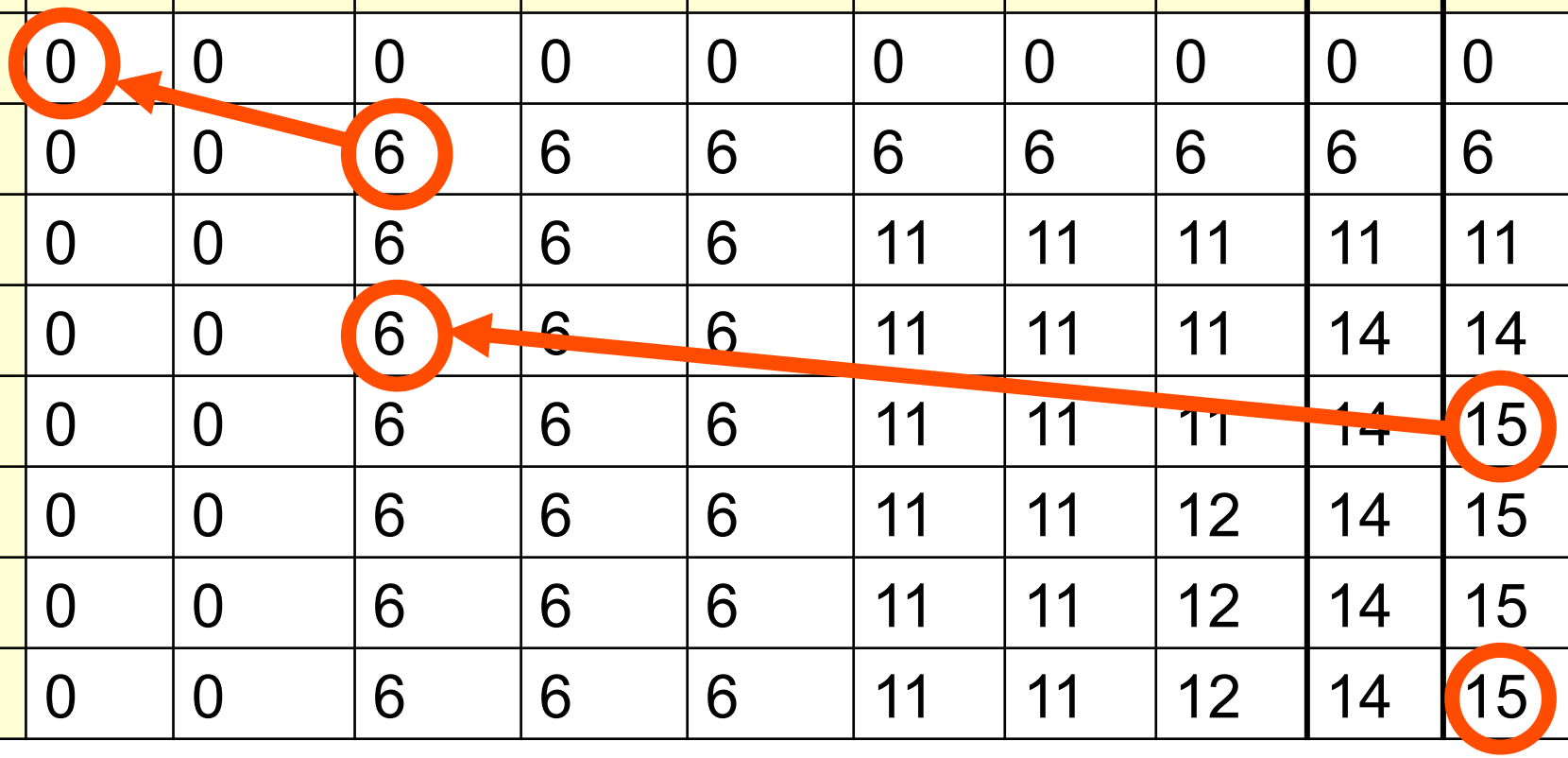
Beispiel:

K=9

n=7

i	1	2	3	4	5	6	7
c_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

i \ W	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15



Die optimale Lösung besteht aus den Objekten 1 und 4

Analyse DP für Rucksackproblem

- Laufzeit der Berechnung der Werte $F(i,W)$ und $D(i,W)$: $O(nK)$
- Laufzeit der Berechnung der Lösung aus der Tabelle: $O(n)$

- Frage: Ist diese Laufzeit polynomiell?

- Antwort: NEIN, denn die Rechenzeit muss auf die Länge (genauer die Bitlänge) der Eingabe bezogen werden.

Analyse DP für Rucksackproblem

- Antwort: NEIN, denn die Rechenzeit muss auf die Länge (genauer die Bitlänge) der Eingabe bezogen werden.
- Wenn alle Zahlen der Eingabe nicht größer als 2^n sind und $K=2^n$, dann ist die Länge der Eingabe $\Theta(n^2)$, denn: ($2n+1$ Eingabezahlen)*(Kodierungslänge)
- Aber die Laufzeit ist von der Größenordnung $n2^n$ und damit exponentiell in der Inputlänge.

- Wenn aber alle Zahlen der Eingabe in $O(n^2)$ sind, liegt die Eingabelänge im Bereich zwischen $\Omega(n)$ und $O(n \log n)$. Dann ist die Laufzeit in $O(n^3)$ und damit polynomiell.

Analyse DP für Rucksackproblem

- Rechenzeiten, die exponentiell sein können, aber bei polynomiell kleinen Zahlen in der Eingabe polynomiell sind, heißen **pseudopolynomiell.**

- **Theorem:** Das 0/1-Rucksackproblem kann mit Hilfe von Dynamischer Programmierung in pseudopolynomieller Zeit gelöst werden.

- In der Praxis kann man Rucksackprobleme meist mit Hilfe von DP effizient lösen, obwohl das Problem NP-schwierig ist (da die auftauchenden Zahlen relativ klein sind).

Analyse DP für Rucksackproblem

- Das besprochene DP-Verfahren heißt **Dynamische Programmierung durch Gewichte**.
- Denn wir betrachten die Lösungswerte als Funktion der Restkapazitäten im Rucksack.

- Durch Vertauschung der Rollen von Gewicht und Wert kann auch **Dynamische Programmierung durch Werte** durchgeführt werden.
- Dort betrachtet man alle möglichen Lösungswerte und überlegt, wie man diese mit möglichst wenig Gewicht erreichen kann.

Analyse DP für Rucksackproblem

- In der Praxis kann man Rucksackprobleme meist mit Hilfe von DP effizient lösen, obwohl das Problem NP-schwierig ist (da die auftauchenden Zahlen i.A. polynomiell in n sind).