

Stand der Dinge:

- Dynamische Programmierung vermeidet Mehrfachberechnung von Zwischenergebnissen
- Bei Rekursion einsetzbar
- Häufig einfache bottom-up Implementierung möglich

Das Subset Sum Problem:

- Algorithmus für schwieriges Problem
- Laufzeit hängt von Eingabewert ab

Fraktionales Rucksack-Problem

- Gegeben sind n Gegenstände. Der i -te Gegenstand besitzt Wert v_i und Gewicht g_i . Ausserdem ist eine Gewichtsschranke W gegeben.
- Zulässige Lösungen sind Zahlen $a_i \in [0,1]$ mit

$$\sum_{i=1}^n a_i g_i \leq W.$$

- Gesucht ist eine zulässige Lösung a_1, \dots, a_n mit möglichst großem Gesamtwert

$$\sum_{i=1}^n a_i v_i.$$

Gierige Lösung des fraktionierten Rucksack-Problems

Algorithmus Gieriges-Einpacken:

1. Sortiere die Verhältnisse v_i/g_i absteigend. Sei

$$v_{\pi(1)}/g_{\pi(1)} \geq v_{\pi(2)}/g_{\pi(2)} \geq \dots v_{\pi(n)}/g_{\pi(n)}$$

für Permutation π auf $(1, \dots, n)$.

2. Bestimme maximales k , so dass noch gilt $\sum_{i=1}^k g_{\pi(i)} \leq W$.
3. Setze $a_{\pi(1)} = a_{\pi(2)} = \dots = a_{\pi(k)} = 1$ und setze

$$a_{\pi(k+1)} = \frac{W - \sum_{i=1}^k g_{\pi(i)}}{g_{\pi(k+1)}}.$$

Alle anderen a_i setze auf 0.

Gieriges Einpacken ist optimal

Satz 17.4: Gieriges Einpacken löst das fraktionale Rucksackproblem optimal.

Das Rucksackproblem:

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Dynamisches Programmieren – Rucksack

Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

Dynamisches Programmieren – Rucksack

Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen und haben Gesamtwert 13
- Optimal?

Dynamisches Programmieren – Rucksack

Beispiel:

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen und haben Gesamtwert 13
- Optimal?
- Objekt 2, 3 und 4 passen und haben Gesamtwert 15 !

Das Rucksackproblem (Optimierungsversion):

- Eingabe: n Objekte $\{1, \dots, n\}$;
Objekt i hat ganzz. pos. Größe $g[i]$ und Wert $v[i]$;
Rucksackkapazität W
- Ausgabe: Menge $S \subseteq \{1, \dots, n\}$ mit $\sum_{i \in S} g[i] \leq W$ und
maximalem Wert $\sum_{i \in S} v[i]$

Herleiten einer Rekursion:

- Sei O optimale Lösung
- Bezeichne $\text{Opt}(i,w)$ den Wert einer optimalen Lösung aus Objekten 1 bis i bei Rucksackgröße w

Unterscheide, ob Objekt n in O ist:

- Fall 1 (n nicht in O):
 $\text{Opt}(n,W) = \text{Opt}(n-1,W)$
- Fall 2 (n in O):
 $\text{Opt}(n,W) = v[n] + \text{Opt}(n-1,W-g[n])$

Rekursion:

- $\text{Opt}(i,0) = 0$ für $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$ dann $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,
$$\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$$

Dynamisches Programmieren – Rucksack

Rekursion:

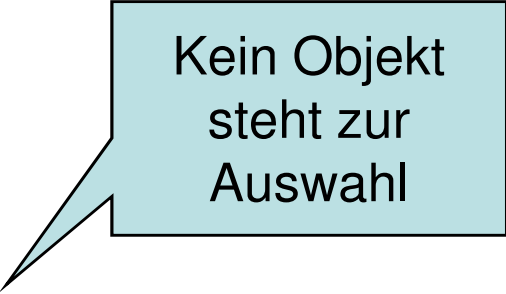
- $\text{Opt}(i,0) = 0$ für $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$ dann $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$

Kein Objekt
passt in den
Rucksack

Dynamisches Programmieren – Rucksack

Rekursion:

- $\text{Opt}(i,0) = 0$ für $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$ dann $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$

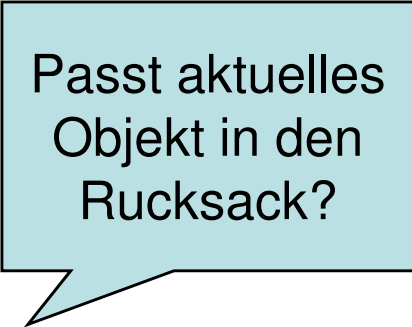


Kein Objekt
steht zur
Auswahl

Dynamisches Programmieren – Rucksack

Rekursion:

- $\text{Opt}(i,0) = 0$ für $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$ dann $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$

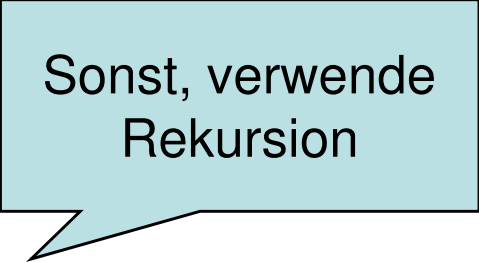


Passt aktuelles
Objekt in den
Rucksack?

Dynamisches Programmieren – Rucksack

Rekursion:

- $\text{Opt}(i,0) = 0$ für $0 \leq i \leq n$
- $\text{Opt}(0,i) = 0$ für $0 \leq i \leq W$
- Wenn $w < g[i]$ dann $\text{Opt}(i,w) = \text{Opt}(i-1,w)$
- Sonst,
 $\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$



Sonst, verwende
Rekursion

Rucksack(n,W)

1. Initialisiere Feld $A[0,\dots,n][0,\dots,W]$ mit $A[0,i] = 0$ für alle $0 \leq i \leq n$ und $A[j,0] = 0$ für alle $0 \leq j \leq W$
2. **for** $i \leftarrow 1$ **to** n **do**
3. **for** $j \leftarrow 1$ **to** W **do**
4. Berechne $A[i,j]$ nach Rekursion
5. **return** $A[n,W]$

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0								
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0
	0	1							W

g[1] > W:
Also $\text{Opt}(i,w) = \text{Opt}(i-1,w)$

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2			
0	0	0	0	0	0	0	0	0	0
	0	1							W

$$\text{Opt}(i,w) = \max\{\text{Opt}(i-1,w), v[i] + \text{Opt}(i-1,w-g[i])\}$$

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2	2		
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0								
1	0	0	0	0	0	2	2	2	
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0							
	0							
	0							
	0							
	0							
	0							
	0							
1	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0
	0	1						W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
0									
0									
0									
0									
0									
0	0	0							
1	0	0	0	0	2	2	2	2	
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0	4	4				
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
0									
0									
0									
0									
0									
1	0	0	0	4	4	4	4	4	
0	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0								
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
0									
0									
0									
0									
0									
0	1	1							
0	0	0	4	4	4	4	4	6	
1	0	0	0	0	2	2	2	2	
0	0	0	0	0	0	0	0	0	
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1	4					
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
0									
0									
0									
0									
0	0	1	1	4	5	5	5	5	
1	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0								
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0								
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0								
	0								
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
0									
0	2	3	5	6	7	9	10	10	
0	2	3	5	6	7	9	10	10	
0	1	3	4	5	7	8	8	8	
0	1	1	4	5	5	5	5	6	
0	0	0	4	4	4	4	4	6	
1	0	0	0	0	0	2	2	2	
0	0	0	0	0	0	0	0	0	
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0								
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe	Wert
	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamisches Programmieren – Rucksack

Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Optimaler Lösungswert für W=8

		Größe	Wert
		g	v
1	5	2	
2	3	4	
	1	1	
	2	3	
	1	2	
	7	3	
	4	7	
n	3	3	

Dynamisches Programmieren – Rucksack

Beispiel:

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Optimaler Lösungswert für W=8

		Größe	Wert
		g	v
1		5	2
2		3	4
		1	1
		2	3
		1	2
		7	3
		4	7
n		3	3

Satz 19.6

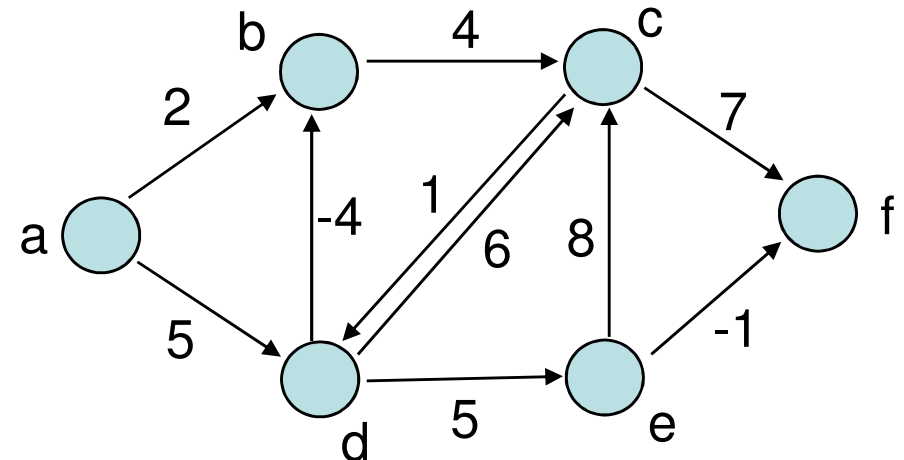
Algorithmus Rucksack berechnet in $\Theta(nW)$ Zeit den Wert einer optimalen Lösung, wobei n die Anzahl der Objekte ist und W die Größe des Rucksacks.

Dynamische Programmierung - APSP

All Pairs Shortest Path (APSP):

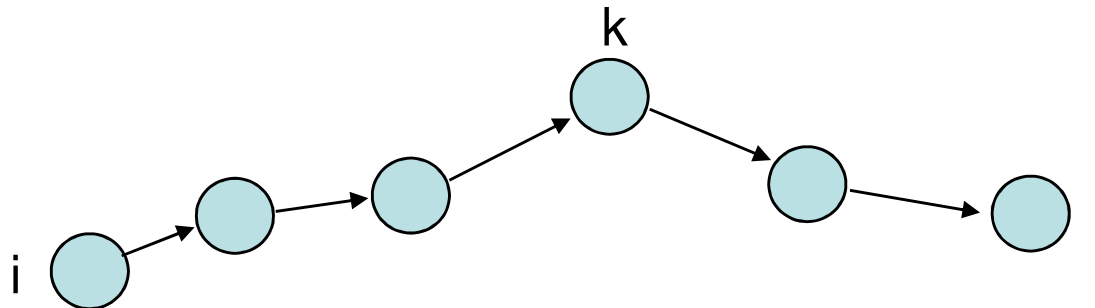
- Eingabe: Gewichteter Graph $G=(V,E)$
- Ausgabe: Für jedes Paar von Knoten $u,v \in V$ die Distanz von u nach v sowie einen kürzesten Weg

	a	b	c	d	e	f
a	0	1	5	5	10	9
b	∞	0	4	5	10	9
c	∞	-3	0	1	6	5
d	∞	-4	0	0	5	4
e	∞	5	8	9	0	-1
f	∞	∞	∞	∞	∞	0



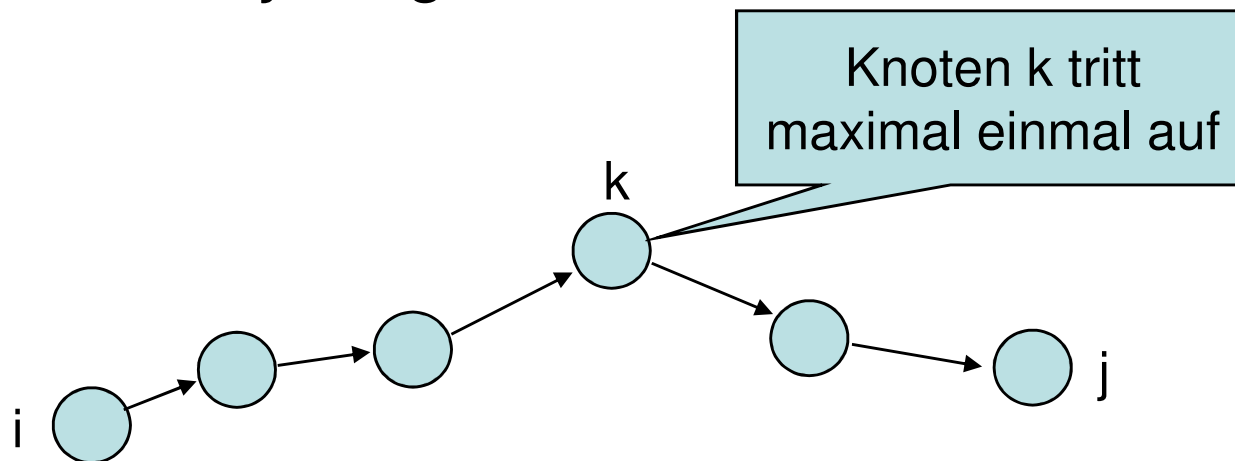
Zur Erinnerung:

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte i - j -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:



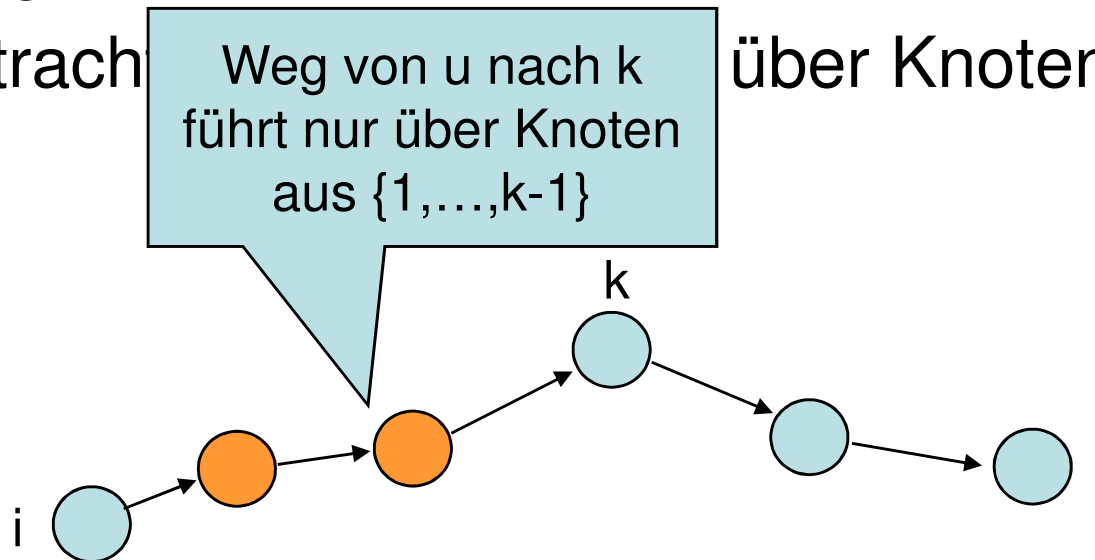
Zur Erinnerung:

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte i - j -Weg, der nur über Knoten aus $\{1, \dots, k\}$ läuft:



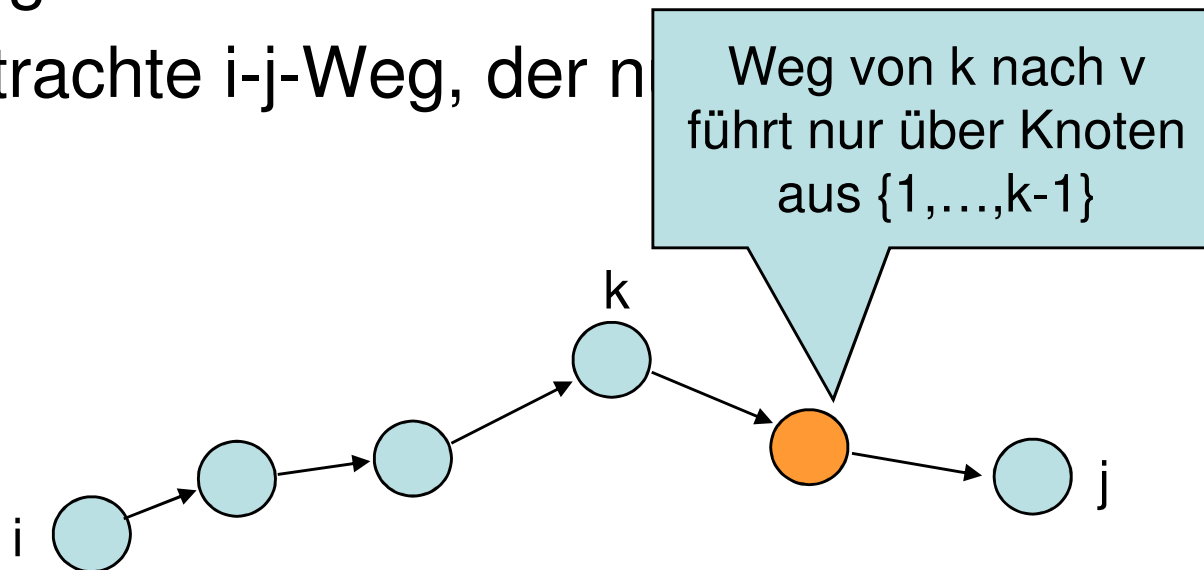
Zur Erinnerung:

- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte den kürzesten Weg von i nach k über Knoten aus $\{1, \dots, k\}$ läuft:



Zur Erinnerung:

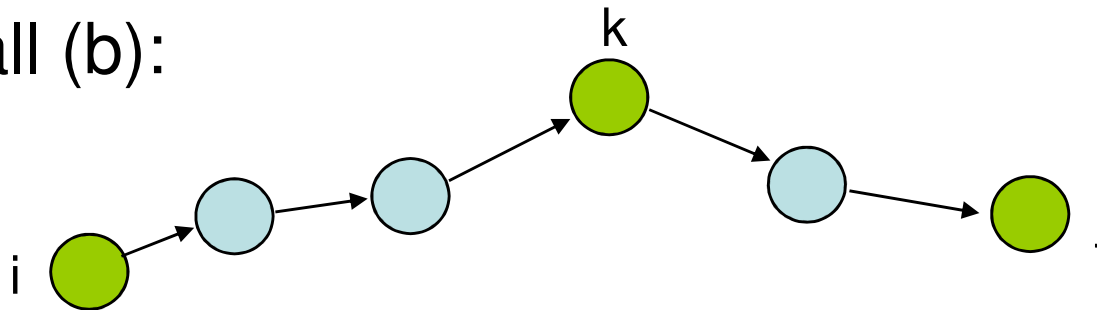
- Sei G ein Graph ohne negative Zyklen und sei j von i aus erreichbar. Dann gibt es einen kürzesten i - j -Weg, der keinen Knoten doppelt benutzt.
- Wir können also annehmen, dass jeder Knoten in jedem Weg maximal einmal vorkommt
- Betrachte i - j -Weg, der nicht über Knoten v aus $\{1, \dots, k\}$ läuft:



Dynamische Programmierung - APSP

- Kürzester i - j -Weg über Knoten aus $\{1, \dots, k\}$ ist
- (a) kürzester i - j -Weg über Knoten aus $\{1, \dots, k-1\}$ oder
- (b) kürzester i - k -Weg über Knoten aus $\{1, \dots, k-1\}$ gefolgt von kürzestem k - j -Weg über Knoten aus $\{1, \dots, k-1\}$

Fall (b):



Die Rekursion:

- Sei $d_{ij}^{(k)}$ die Länge eines kürzesten i - j -Wegs mit Knoten aus $\{1, \dots, k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{falls } k=0 \\ \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{falls } k \geq 1 \end{cases}$$

- Matrix $D^{(n)} = (d_{ij}^{(n)})$ enthält die gesuchte Lösung

Dynamische Programmierung - APSP

Floyd-Warshall(W, n)

1. $D^{(0)} \leftarrow W$
2. **for** $k \leftarrow 1$ **to** n **do**
3. **for** $i \leftarrow 1$ **to** n **do**
4. **for** $j \leftarrow 1$ **to** n **do**
5. $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6. **return** $D^{(n)}$

Dynamisches Programmieren – Rucksack

Teile & Herrsche:

- Aufteilen der Eingabe in mehrere Unterprobleme
- Rekursives lösen der Unterprobleme
- Zusammenfügen

Gierige Algorithmen:

- Konstruiere Lösung Schritt für Schritt
- In jedem Schritt optimiere einfaches, lokales Kriterium

Dynamische Programmierung:

- Formuliere Problem rekursiv
- Vermeide mehrfache Berechnung von Teilergebnissen
- Verwende „bottom-up“ Implementierung