ADS: Algorithmen und Datenstrukturen 2

Teil 6

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

15. Mai 2019

[Letzte Aktualisierung: 05/06/2019, 08:53]

Das 0/1-Rucksackproblem

- Objekte $\{1, 2, \ldots, n\}$
- mit Volumina $t_1, t_2, \ldots, t_n$ und Werten $p_1, p_2, \ldots, p_n$
- Rucksack hat Gesamtvolumen c.

Gesucht: Ein 0/1-Vektor $a_1, ..., a_n \in \{0, 1\}$ mit

$$\sum_{i=1}^{n} a_i t_i \leq c$$

Optimierungsproblem: finde $a \in A$, so dass der Gesamtwert

$$f(a) = \sum_{i=1}^{n} a_i p_i$$

maximal wird.

Beispiel

i	1	2	3	4	5
p_i	8	2	7	8	3
t_i	3	1	4	5	2

$c = 10$

- optimale Lösung für 0/1-Rucksack hat Wert 20
  Objekte $\{1, 2, 3, 5\}$, also $x_1 = x_2 = x_3 = x_5 = 1$, $x_4 = 0$
- es gibt 2^n 0/1-Vektoren, alles auszuprobieren kostet also $O(n \times 2^n)$ Zeit
- Lösungsprinzip: Dynamische Programmierung

Dynamische Programmierung (DP)

1. Lösen eines Problem durch Zurückführen auf einfachere Teilprobleme, die gelöst und zur Gesamtlösung zusammengefügt werden.
2. In DP sind diese Teilprobleme i.A. *"überlappend"*, d.h. sie können ihrerseits gemeinsame Teilprobleme haben.
3. Um mehrfache Berechnung von Teilproblemlösungen einzusparen, werden die Teilergebnisse abgespeichert. ($\Rightarrow$ Effizienz)

Anmerkungen:

- Zusammensetzung aus Teillösungen in (1) funktioniert nur für bei geigneten Problemen und geeigneter Zerlegung, nicht allgemein!
- Divide-and-Conquer ist ein weiteres algorithmisches Prinzip, auf das (1) zutrifft. Im Gegensatz zu DP, wird bei D&C i.A. in *nicht-überlappende* Teilprobleme zerlegt, die *unabhängig* gelöst werden.

Prinzip Dynamische Programmierung (für Optimierung)

1. Charakterisiere den Lösungsraum.
2. Definiere rekursiv, wie eine optimale Lösung aus kleineren optimalen Lösungen zusammengesetzt wird.
3. Lege eine Berechnungsreihenfolge fest, so dass die Lösungen von kleineren Teilproblemen berechnet werden, bevor diese für die Lösung eines grösseren benötigt werden.

Voraussetzung für (2): Bellmannsches Optimalitätsprinzip

Die optimale Lösung eines (Teil-)Problems (der Grösse n) setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.

Bellmannsches Optimalitätsprinzip für das Rucksackproblem

Es sei $O_i \subseteq 1, 2, ..., n$ eine optimale Packung bei Kapazitätsgrenze c.

Dann gilt:

- Entweder $i \in O_i$, dann ist $O_i \setminus \{i\}$ eine optimale Packung aus $\{1, ..., i-1\}$ bei Kapazitätsgrenze $c - t_i$ oder
- $i \notin O_i$, dann ist O_i eine optimale Packung aus $\{1, ..., i-1\}$ bei Kapazitätsgrenze c

Algorithmus für das 0/1-Rucksackproblem

Es sei $f(i,t)$ der maximale Wert beim Packen des Rucksacks mit i Objekten aus $\{1,2,...,n\}$ bei Kapazitätsgrenze c.

$f(i,t)$ kann dann rekursiv wie folgt berechnet werden

Berechnung von $f(i,t)$:

$$f(i,t) = \begin{cases} 0 & \textit{falls } i = 0 \\ f(i-1,t) & \textit{falls } i > 0 \textit{ und } t \leq t_i \\ \max\{f(i-1,t), f(i-1,t-t_i)+p_i\} & \textit{sonst} \end{cases}$$

Beispieltabelle $f(i, t)$

Es sei n $= 5$, $p_i = (8, 2, 7, 8, 3)$ $t_i = (3, 1, 4, 5, 2)$, c$= 10$

		t									
		1	2	3	4	5	6	7	8	9	10
i	1	0	0	8	8	8	8	8	8	8	8
	2	2	2	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)	10 (1,2)
	3	2 (2)	2 (2)	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	15 (1,3)	17 (1,2,3)	17 (1,2,3)	17 (1,2,3)
	4	2 (2)	2 (2)	8 (1)	10 (1,2)	10 (1,2)	10 (1,2)	15 (1,3)	17 (1,2,3)	18 (1,2,4)	18 (1,2,4)
	5	2 (2)	3 (5)	8 (1)	10 (1,2)	11 (1,5)	13 (1,2,5)	15 (1,3)	17 (1,2,3)	18 (1,3,5)	20 (1,2,3,5)

Noch mal Rucksackproblem

Nehmen wir an, jeder Gegenstand kann **mehrfach** eingepackt werden.
Welchen **maximalen** Wert können wir erreichen?

Definiere

- `kosten[j]` ... Volumen von `j`
- `wert[j]` ... Wert von `j`

- Sei W_i der maximale Wert von Gegenständen mit Volumen i
- Wenn bei Volumen i der letzte eingepackte Gegenstand j ist, so ist der beste Wert $\texttt{wert}[j] + W_{i-\texttt{kosten}[j]}$
- Also:

$$W_i = \max_j \left(\texttt{wert}[j] + W_{i-\texttt{kosten}[j]}\right)$$

Rucksackproblem

- Effiziente Berechnung, indem die Operationen in einer zweckmäßigen Reihenfolge ausgeführt werden.
- Variablen
  - `i` Kapazität ($1 \leq$ `i` $\leq$ `M`)
  - `j` Gegenstand ($1 \leq$ `j` $\leq$ `N`)
- Felder
  - `best_wert[i]` ... bester Wert bei Kosten `i` ($\hat{=} W_i$)
  - `best_obj[i]` ... ausgewähltes `j` für besten Wert bei Kosten `i`
- ```
  for ( j = 1 ; j <= N ; j ++ ) {
     for ( i = 1 ; i <= M ; i++ ) {
        if ( i >= kosten[j] ) {
           if ( best_wert[i]
                < best_wert[i - kosten[j]] + wert[j] ) {
              best_wert[i] = best_wert[i - kosten[j]] + wert[j];
              best_obj[i]  = j ;
  }}}}
  ```
  ```

Rucksackproblem

Bezeichnung	A	B	C	D	E
Wert	4	5	10	11	13
Kosten	3	4	7	8	9

Lösung des Rucksack-Beispiels (mit max. Kapazität 17):

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

Rucksackproblem: Beispiel

- Das erste Zeilenpaar zeigt den maximalen Wert (den Inhalt der Felder `best_wert` und `best_obj`), 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_obj berechnet werden. Per Definition ist best_obj[M] in ihm enthalten, und der restliche Inhalt ist der gleiche wie im optimalen Rucksack der Größe M - kosten [best_obj [M]] usw.

Rucksackproblem: Eigenschaften

- Rucksack-Problems mit DP: O($N \cdot M$)
- $\Rightarrow$ Rucksack-Problem leicht, wenn die Kapazität M klein ist;
  sonst kann die Laufzeit aber sehr groß werden.
- Problem: die Effizienz dieses Verfahrens hängt davon ab, wieviele *verschiedene Werte* die Kosten annehmen können.
  Z.B. kritisch, wenn Kosten keine kleinen ganzen Zahlen sind.
  (Man kann das Verfahren auf reell-wertige Kosten erweitern, bekommt aber dann schnell dieses Problem.)
- Grundsätzlich gilt bei diesem Algorithmus, dass optimale Entscheidungen nicht geändert werden müssen, nachdem sie einmal getroffen wurden.
  ($\hat{=}$ *Bellmansches Optimalitätsprinzip*)
- Zur Erinnerung: DP funktioniert, immer wenn (dem Algorithmus eine Problemzerlegung zugrunde gelegt wird, bei der) dieses Prinzip gilt.

Greedy und das 0/1-Rucksackproblem

- Kanonischer Greedy für 0/1-Rucksack erreicht Wert 19, Objekte $\{1, 4, 5\}$, also $x_1 = x_4 = x_5 = 1$, $x_2 = x_3 = 0$.
- Mengensystem erlaubter Objektkombinationen ist generell *kein Matroid*. (Austauscheigenschaft nicht erfüllt).
- Daher liefert der kanonische Greedy-Algorithmus nicht immer die optimale Lösung.
- Aber **fraktionales Rucksackproblem** kann mit Greedy-Ansatz gelöst werden.

Fraktionales Rucksackproblem

- Wie 0/1-Rucksackproblem, aber $x_i \in [0, 1]$.
- Beliebige Bruchteile eines Objekts dürfen eingepackt werden.

Optimale Lösung wird gefunden mit folgendem Greedy-Ansatz:

- Berechne von jedem Objekt i den *Nutzen*, also das Verhältnis aus Gewinn und Volumen $p_i/t_i =: u_i$.
- Sortiere Objekte absteigend nach Nutzen (spezifischer Gewinn): $u_1 \geq u_2 \geq \cdots \geq u_n$.
- Finde maximales i, so dass die Objekte $\{1, \ldots, i\}$ vollständig in den Rucksack passen, $\sum_{j=1}^{i} t_j \leq c$.
- Setze $x_j = 1$ für $j \leq i$, $x_j = 0$ sonst.
- Falls $i \neq n$, setze

$$x_{i+1} = \left(c - \sum_{j=1}^{i} t_j \right) / t_{i+1}$$

Beispiel

i	1	2	3	4	5
p_i	8	2	7	8	3
t_i	3	1	4	5	2
u_i	$\frac{8}{3}$	2	$\frac{7}{4}$	$\frac{8}{5}$	$\frac{3}{2}$

$c = 10$

- Greedy für fraktionalen Rucksack erreicht Wert 20.2
  $x_1 = x_2 = x_3 = 1$, $x_4 = \frac{2}{5}$, $x_5 = 0$.

Ganzzahlige und relaxierte Probleme

- Wichtige Unterscheidung zwischen ganzzahligen und reellzahligen ("kontinuierlichen") Optimierungsproblemen.
- Einschränkung auf ganze Zahlen macht viele Probleme qualitativ schwerer lösbar (Beispiel hier: 0/1-Rucksack).
- Das einem ursprünglich ganzzahligen Problem zugeordnete kontinuierliche Problem heißt *Relaxierung* oder *relaxiertes Problem*.
- Relaxierung wird oft betrachet, weil sie eine untere/obere Schranke an die Werte der Kosten-/Gewinnfunktion liefert.

Branch and Bound

Bisher behandelte Strategie für Optimierungsprobleme:

- Greedy-Verfahren, Dynamische Programmierung

Was tun, wenn dieses nicht anwendbar ist?

- Naiver Ansatz ("Brute Force"): vollständige Aufzählung aller möglichen Lösungen.
- *Branch and Bound*: Identifiziere möglichst große Teilmengen des Lösungsraums, die keine optimale Lösung enthalten können, und überspringe diese beim Aufzählen.

Beispiel für Branch and Bound

Aufzählen von Lösungen

$x_1 \ldots x_5$	Ges.Wert	Ges.Vol.
00000	0	0
10000	8	3
01000	7	4
11000	15	7
001**	≤ 13	
...		

0/1 - Rucksack, $c = 10$

i	1	2	3	4	5
p_i	8	7	8	3	2
t_i	3	4	5	2	1

Keine Lösung x mit $x_1 = x_2 = 0$ und $x_3 = 1$ kann wertvoller als 8+3+2=13 sein. Diese Lösungen brauchen daher nicht aufgezählt zu werden, denn sie sind alle schlechter als der bisher gesehene Maximalwert 15.

Grundidee (Minimierungsproblem)

1. Strukturiere den Lösungsraum so, dass er einen Baum darstellt.

   (Die Wurzel entspricht der leeren Lösung, die zu jeder beliebigen Lösung erweitert werden kann.)

2. Sobald ein neuer Knoten erzeugt wurde, berechne für diesen Teilbaum die untere Schranke b (bound).

   (Keine Lösung, die sich oberhalb des betreffenden Knotens befindet, kann einen besseren Wert als b haben)

3. Vergleiche die bounds aller bisherigen Baumblätter und expandiere dasjenige mit dem kleinsten bound.

   (Wenn ein Ast so weit entwickelt worden ist, dass das entsprechende Blatt eine vollständige und im Vergleich mit allen anderen bounds minimale Lösung darstellt, dann ist diese Lösung optimal.)

Traveling Salesman Problem (TSP)

- **Gegeben:** n Städte, paarweise Distanzen:
  d_{ij} Entfernung von Stadt i nach Stadt j.
- **Gesucht:** Rundreise mit minimaler Länge durch alle Städte, also Permutation $\pi : (1, ..., n) \rightarrow (1, ..., n)$, für die

$$c(\pi) \quad = \quad \left[\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} \right] \quad + \quad d_{\pi(n)\pi(1)}$$

  minimal.

- NP-hart (d.h. exponentiell, ausser P=NP)
- Naiver Algorithmus: Alle $(n-1)!$ Reihenfolgen betrachten.

Branch and Bound für TSP

- Operiere auf Kantenmengen. Zulässige Kantenmengen X: maximal je eine eingehende und eine ausgehende Kante pro Stadt, keine Zyklen kürzer als n (Anzahl Städte).
- Untere Schranke: Kosten der gewählten Kanten *plus* Kosten der billigsten (eingehenden und ausgehenden) Kanten der unverbundenen Städte.

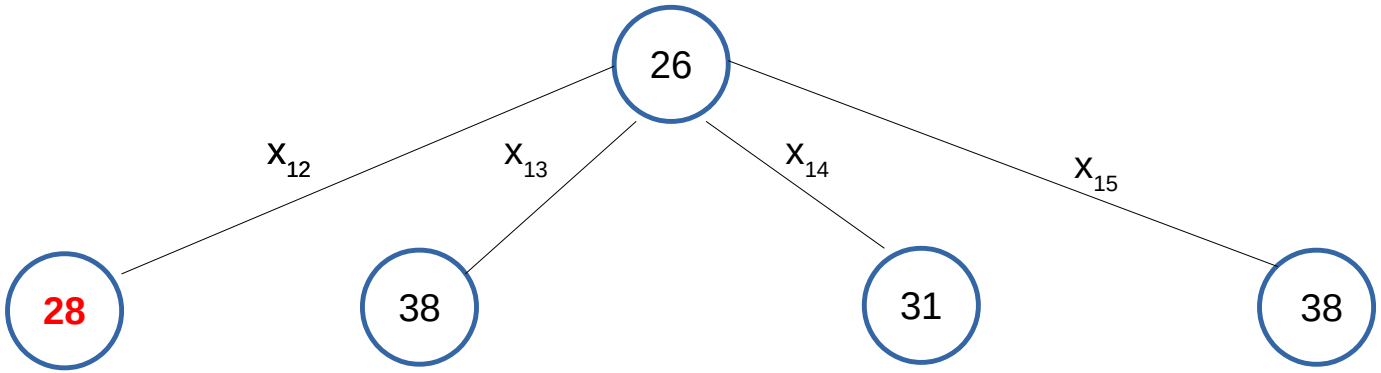
Branch and Bound für TSP - zunächst symmetrisch

Minimale Distanz ist Summe der Zeilen- bzw. Spalten-Minima

	1	2	3	4	5
1	-	5	13	8	17
2	5	-	9	4	14
3	13	9	-	6	7
4	8	4	6	-	11
5	17	14	7	11	-

Kosten für Verzweigung (bound) sind Kosten für gewählten Weg plus minimale Kosten für Zeilen/Spalten in Matrix ohne diesen Weg

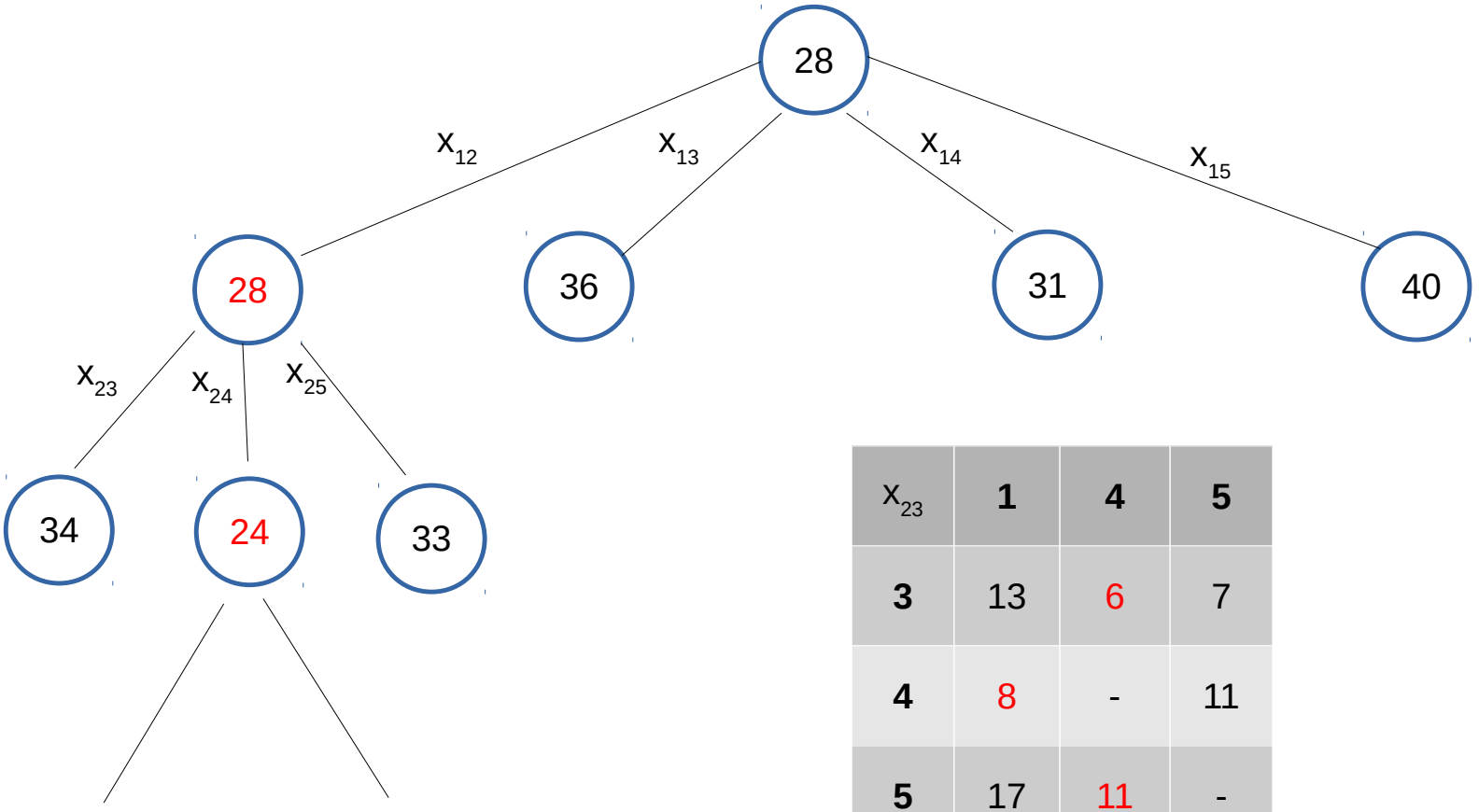
Beispiel symmetrisch 1



x_{12}	1	3	4	5
2	-	9	4	14
3	13	-	6	7
4	8	6	-	11
5	17	7	11	-

x_{13}	1	2	4	5
2	5	-	4	14
3	-	9	6	7
4	8	4	-	11
5	17	14	11	-

Beispiel symmetrisch 2



x_{23}	**1**	**4**	**5**
3	13	6	7
4	8	-	11
5	17	11	-

Branch and Bound für TSP - asymmetrisch

Beispiel für untere Schranke bei TSP:

	1	2	3	4	5
1	-	5	13	8	17
2	7	-	9	4	14
3	12	10	-	6	7
4	8	4	9	-	11
5	15	14	8	12	-

Betrachte Teilmenge von Lösungen, die Kanten $(1,2)$ und $(2,3)$ enthalten.
Kosten der vorhandenen Kanten:
$\gamma = d_{12} + d_{23}$
minimale Kosten für ausgehende Kanten:
$\beta_{\text{aus}} = d_{34} + d_{41} + d_{54}$
minimale Kosten für eingehende Kanten:
$\beta_{\text{ein}} = d_{41} + d_{34} + d_{35}$
Wert der unteren Schranke
$\gamma + \max\{\beta_{\text{aus}}, \beta_{\text{ein}}\}$

Beispiel asymmetrisch

	1	**2**	**3**	**4**	**5**
1	-	5	13	8	17
2	7	-	9	4	14
3	12	10	-	6	7
4	8	4	9	-	11
5	15	14	8	12	-

Noch mal Traveling Salesman Problem (TSP)

Dynamische Programmierung für TSP

- Sei $g(i, S)$ Länge des kürzesten Weges von Stadt i über jede Stadt in der Menge S (jeweils genau *ein* Besuch) nach Stadt 1.
- Lösung des TSP also: $g(1, \{2, ..., n\})$.
  Anmerkung: Stadt 1 kann beliebig gewählt werden, da Rundreise gesucht wird.
- Es gilt:

$$g(i, S) = \begin{cases} d_{i1} & \text{falls } S = \varnothing \\ \min_{j \in S} \left[d_{ij} + g(j, S \setminus \{j\}) \right] & \text{sonst} \end{cases}$$

Dynamische Programmierung für TSP - Beispiel

Es seien n= 4 und D =

	1	2	3	4
1	-	5	13	8
2	7	-	9	14
3	12	10	-	6
4	8	4	9	-

(Wie vorheriges Beispiel, aber auf 4 Städte verkürzt)

Dynamische Programmierung für TSP - Beispiel (2)

$$|S| = 0 : g(2, \{\}) = 7, g(3, \{\}) = 12, g(4, \{\}) = 8$$

$|S| = 1 :$
$g(2, \{3\}) = d_{23} + g(3, \{\}) = 9 + 12 = 21$
$g(2, \{4\}) = d_{24} + g(4, \{\}) = 14 + 8 = 22$
$g(3, \{2\}) = d_{32} + g(2, \{\}) = 10 + 7 = 17$
$g(3, \{4\}) = d_{34} + g(4, \{\}) = 6 + 8 = 14$
$g(4, \{2\}) = d_{42} + g(2, \{\}) = 4 + 7 = 11$
$g(4, \{3\}) = d_{43} + g(3, \{\}) = 9 + 12 = 21$

$|S| = 2 :$
$g(2, \{3, 4\}) = min\{d_{23} + g(3, \{4\}), d_{24} + g(4, \{3\})\} = min\{9 + 14, 14 + 21\} = 23$
$g(3, \{2, 4\}) = min\{d_{32} + g(2, \{4\}), d_{34} + g(4, \{2\})\} = min\{10 + 22, 6 + 11\} = 17$
$g(4, \{2, 3\}) = min\{d_{42} + g(2, \{3\}), d_{43} + g(3, \{2\})\} = min\{4 + 21, 9 + 17\} = 25$

Dynamische Programmierung für TSP - Beispiel (3)

$|S| = 3$:
$g(1, \{2, 3, 4\}) =$
$min\{d_{12} + g(2, \{3, 4\}), d_{13} + g(3, \{2, 4\}, d_{14} + g(4, \{2, 3\}\} =$
$min\{5 + 23, 13 + 17, 8 + 25\} = 28$

Die optimale Rundreise mit 1 als Startpunkt lautet also:
$\pi = (1, 2, 3, 4)$ mit der Länge 28

Bei jeder Berechnung $g(i, S)$ kann das j gespeichert werden, das als Minimum angenommen wird.
Die Rundreise π lässt sich dann wie folgt konstruieren:

- starte mit 1
- berechne j aus $g(1, S_0)$
- berechne π rekursiv weiter

Wann funktioniert DP?

1. Es muss nicht immer möglich sein, die Lösungen kleinerer Probleme so zu kombinieren, dass sich die Lösung eines größeren Problems ergibt.
2. Die Anzahl der zu lösenden Probleme kann unvertretbar groß sein.
3. Es ist noch nicht gelungen, genau anzugeben, welche Probleme mit Hilfe der dynamischen Programmierung in effizienter Weise gelöst werden können. Es gibt viele "schwierige" Probleme, für die sie nicht anwendbar zu sein scheint, aber auch viele "leichte" Probleme, für die sie weniger effizient ist als Standardalgorithmen.