Noch einmal TSP

Wir hatten das TSP-Problem (*Problem der Handlungsreisenden*) schon als Beispiel für die Branch-and-Bound Methode kennengelernt. Nun wollen wir noch einen zweiten Algorithmus angeben, der nach dem Prinzip des Dynamischen Programmierens vorgeht.

Das Problem sei wie folgt gegeben:

Eine $n \times n$-Entfernungsmatrix, die für alle $i, j \in \{1, \ldots, n\}$ angibt, wie weit der Weg von Stadt i zu Stadt j ist. Die Einträge dieser Matrix bezeichnen wir mit $m_{i,j}$.

Hierbei erlauben wir auch $m_{i,j} \neq m_{j,i}$, d.h. die Matrix ist nicht notwendig symmetrisch. Auch dürfen Einträge den Wert ∞ annehmen. Das heißt dann, es gibt keinen direkten Weg.

Gesucht ist nun eine Permutation π der Zahlen $\{1, \ldots, n\}$ so, dass der durch $\pi(1), \ldots, \pi(n)$ und zurück zu $\pi(1)$ gegebene Rundweg minimale Länge hat.

TSP mit Dynamischem Programmieren

Die Idee ist die folgende:

Wir suchen eine Permutation π der Zahlen $1, \ldots, n$, wobei wir annehmen, dass $\pi(1) = 1$ gilt.

Als Kosten des Weges der durch π definiert ist, erhalten wir:

$$c(\pi) = \sum_{k=1}^{n-1} m_{\pi(k),\pi(k+1)} + m_{\pi(n),\pi(1)}$$

oder, weil $\pi(1) = 1$ gelten soll:

$$c(\pi) = m_{1,\pi(2)} + \sum_{k=2}^{n-1} m_{\pi(k),\pi(k+1)} + m_{\pi(n),1}$$

Wir halten eine Tabelle mit Einträgen $g(i, S)$ für $S \subseteq \{2, \ldots, n\}$ und $i \in \{1, \ldots, n\}$, wobei der Eintrag $g(i, S)$ die Kosten eines kürzesten Weges angeben soll, der von Stadt i startend, alle Städte der Menge S genau einmal besucht und dann zu Stadt 1 zurückkehrt.

Formel für $g(i, S)$

Die Formel für $g(i, \emptyset)$ ist leicht zu formulieren:

$$g(i, \emptyset) = m_{i,1}$$

Wenn $S \neq \emptyset$ gilt, können wir alle Möglichkeiten für die erste Stadt $j \in S$, die besucht werden soll, durchprobieren. Der Weg dahin kostet $m_{i,j}$. Danach müssen wir noch alle Städte aus $S \setminus \{j\}$ besuchen und schließlich zu Stadt 1 zurückkehren. Die Kosten für diese Aufgabe können wir rekursiv berechnen, sie sind gegeben durch $g(j, S \setminus \{j\})$. Also erhalten wir:

$$g(i, S) = \min_{j \in S} \left(m_{i,j} + g(j, S \setminus \{j\}) \right)$$

Die gesuchten Gesamtkosten des kürzesten Weges durch alle n Städte sind damit gegeben durch $g(1, \{2, \ldots, n\})$.

Pseudo-Code

Wir berechnen $g(i, S)$ der Reihe nach – zunächst für $S = \emptyset$, dann für $|S| = 1$, $|S| = 2$, usw. bis $|S| = n - 2$. Zum Schluss kann wie gewünscht $g(1, \{2, \ldots, n\})$ berechnet werden:

```
FOR i := 2 TO n DO
        g[i, ∅] := m[i, 1];
FOR k := 1 TO n − 2 DO
        FORALL S ⊆ {2, ..., n}, |S| = k DO
                FORALL i ∈ {2, ..., n} \ S DO
                        g[i, S] := min_{j∈S} (m_{i,j} + g[j, S \ {j}])
g[1, {2, ..., n}] := min_{j∈{2,...,n}} (m_{1,j} + g[j, {2, ..., n} \ {j}])
```

Analyse

Termination und Korrektheit sollten klar sein.

Zur Effizienz:

Der Algorithmus nutzt eine Tabelle mit Einträgen $g(i, S)$ für $i \in \{1, \ldots, n\}$ und $S \subseteq \{2, \ldots, n\}$. Die übrigen gespeicherten Daten sind Laufvariablen i und k.

Damit ergibt sich ein Speicherbedarf in $O(n \cdot 2^n)$.

Jeder der $n \cdot 2^n$ Einträge in der Tabelle wird einmal berechnet, und zwar durch Ermittlung eines Minimums von maximal $n - 1$ bekannten Werten.

Also erhalten wir die Zeitkomplexität $O(n^2 \cdot 2^n)$. Das ist eine erhebliche Verbesserung gegenüber allen naiven Ansätzen, deren Zeitkomplexität im Bereich $n!$ liegt.

Beispiel

Wir betrachten das selbe Beispiel wie bei dem Branch-and-Bound Ansatz noch einmal. Die Matrix M war:

$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

Die Menge $\{2,3,4\}$ hat 8 Teilmengen, also berechnen wir eine Tabelle mit 4 Zeilen und 8 Spalten:

	$\emptyset$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2,3\}$	$\{2,4\}$	$\{3,4\}$	$\{2,3,4\}$
$i=1$	–	–	–	–	–	–	–	?
$i=2$	5	–			–	–		–
$i=3$	6		–		–		–	–
$i=4$	8			–		–	–	–