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:

$$
\begin{array}{l}
\texttt{FOR } i := 2 \texttt{ TO } n \texttt{ DO} \\
\qquad g[i, \emptyset] := m[i, 1]; \\
\texttt{FOR } k := 1 \texttt{ TO } n-2 \texttt{ DO} \\
\qquad \texttt{FORALL } S \subseteq \{2, \ldots, n\},\ |S| = k \texttt{ DO} \\
\qquad\qquad \texttt{FORALL } i \in \{2, \ldots, n\} \setminus S \texttt{ DO} \\
\qquad\qquad\qquad g[i, S] := \min_{j \in S} \left(m_{i,j} + g[j, S \setminus \{j\}] \right) \\
g[1, \{2, \ldots, n\}] := \min_{j \in \{2, \ldots, n\}} \left(m_{1,j} + g[j, \{2, \ldots, n\} \setminus \{j\}] \right)
\end{array}
$$

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:

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

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			–		–	–	–

Beispiel (Forts.)

In der Tabelle auf der vorigen Folie fehlen 9 Einträge, die wir nun berechnen:

$$g(3,\{2\}) = m_{3,2} + m_{2,1} = 13 + 5 = 18$$
$$g(4,\{2\}) = m_{4,2} + m_{2,1} = 8 + 5 = 13$$
$$g(2,\{3\}) = m_{2,3} + m_{3,1} = 9 + 6 = 15$$
$$g(4,\{3\}) = m_{4,3} + m_{3,1} = 9 + 6 = 15$$
$$g(2,\{4\}) = m_{2,4} + m_{4,1} = 10 + 8 = 18$$
$$g(3,\{4\}) = m_{3,4} + m_{4,1} = 12 + 8 = 20$$
$$g(4,\{2,3\}) = \min\{m_{4,2} + g(2,\{3\}), m_{4,3} + g(3,\{2\})\} = \min\{23,27\} = 23(2)$$
$$g(3,\{2,4\}) = \min\{m_{3,2} + g(2,\{4\}), m_{3,4} + g(4,\{2\})\} = \min\{31,25\} = 25(4)$$
$$g(2,\{3,4\}) = \min\{m_{2,3} + g(3,\{4\}), m_{2,4} + g(4,\{3\})\} = \min\{29,25\} = 25(4)$$

	$\emptyset$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2,3\}$	$\{2,4\}$	$\{3,4\}$	$\{2,3,4\}$
$i=1$	–	–	–	–	–	–	–	35(2)
$i=2$	5	–	15	18	–	–	25(4)	–
$i=3$	6	18	–	20	–	25(4)	–	–
$i=4$	8	13	15	–	23(2)	–	–	–

Beispiel (Abschluss)

Bitte überprüfen Sie selbst, dass der (hier nicht vorgerechnete) Eintrag rechts oben (35(2)) tatsächlich korrekt ist.

Wie interpretieren wir die Ergebnisse aus der Tabelle?

Der Eintrag 35(2) ganz rechts oben besagt, dass der kürzeste Rundweg die Länge 35 hat und von Stadt 1 zunächst zu Stadt 2 führt. Nun haben wir die Aufgabe von Stadt 2 durch die Städte 3 und 4 zurück zu Stadt 1 zu gelangen, d.h. wir müssen im Eintrag $g(2, \{3, 4\})$ nachsehen, wo wir den Eintrag 25(4) finden. Der Weg führt uns also von Stadt 2 zu Stadt 4, von wo wir dann nur noch Stadt 3 zu besuchen haben, bevor der Ausgangspunkt Stadt 1 wieder angesteuert wird.

Die Lösung ist also $1 \to 2 \to 4 \to 3 \to 1$, mit der Länge

$$m_{1,2} + m_{2,4} + m_{4,3} + m_{3,1} = 10 + 10 + 9 + 6 = 35.$$

String Matching

Das String-Matching-Problem ist das folgende:

Gegeben ist ein *Text* $T \in \Sigma^n$ und ein Muster (*pattern*) $P \in \Sigma^m$, wobei $m \leq n$.

Gesucht sind alle Verschiebungswerte (*shifts*) s, für die gilt:

$$T[s+1, \ldots, s+m] = P[1, \ldots, m]$$

In Worten: Wir suchen alle Vorkommen des Musters P im Text T.

Eine naheliegende Variante wäre, das *kleinste* s, oder auch einfach nur *irgendein* s mit dieser Eigenschaft zu suchen.

Wir wollen hier beim allgemeinsten Problem bleiben, nämlich alle Werte für s zu finden.

Erster Ansatz für String Matching

Die naive Methode, das String-Matching Problem zu lösen, wäre die folgende:

Prüfe für alle $s \in \{0, \ldots, n-m\}$, ob s eine Lösung ist.

Dazu kann man den folgenden Algorithmus verwenden:

```
FOR s := 0 TO n − m DO
        found := TRUE;
        FOR i := 1 TO m DO
                IF T[s + i] ≠ P[i] THEN found := FALSE ENDIF
        ENDFOR
        IF found THEN OUTPUT(s) ENDIF
ENDFOR
```

Und welche Komplexität erhält man auf diese Weise?

Zusätzlicher Speicherplatz wird nicht benötigt.

Als Zeitkomplexität erhält man offenbar $\Theta(m \cdot n)$.

String Matching nach Rabin und Karp

Der Rabin-Karp Algorithmus zum Pattern Matching arbeitet mit einer einfach zu berechnenden sogenannten *Hashfunktion*. Diese soll jedem Pattern der Länge m einen *Hashwert* aus einem vergleichsweise kleinen Zahlenbereich zuordnen. Dann kann man das Matching Problem so lösen:

```
H := Hashwert von P;
FOR s := 0 TO n − m DO
   IF (Hashwert von T[s + 1 ... s + m]) = H THEN Test(s) ENDIF;
```

Die Prozedur `Test` vergleicht wieder $T[s+i]$ mit $P[i]$ für alle i von 1 bis m.

Zur Definition eines Hashwerts für jedes m-buchstabige Pattern über dem Alphabet Σ fassen wir dieses als m-stellige Zahl im Zahlensystem mit Basis $|\Sigma|$ auf und berechnen den Wert dieser Zahl modulo einer geeigneten natürlichen Zahl q.

Rabin-Karp: Berechnung des Hashwerts

Der entscheidende Trick beim Rabin-Karp Algorithmus ist die sehr effiziente Berechnung der Hashwerte durch sukzessive Adaption.

Wir verdeutlichen das Prinzip an einem Beispiel:

Es sei $m = 4$ und $|\Sigma| = 3$. Als Modulus wählen wir $q = 11$. Der Text enthalte das Teilwort $a_5a_4a_3a_2a_1$. Dann ist der Hashwert erst $27a_5 + 9a_4 + 3a_3 + a_2 \mod 11$. Im nächsten Schritt soll daraus $27a_4 + 9a_3 + 3a_2 + a_1 \mod 11$ werden.

Wie geht das effizient?

Wir subtrahieren $27a_5$ (bzw. $5a_5$, weil wir modulo 11 rechnen), multiplizieren dann mit 3 (also mit $|\Sigma|$), und addieren a_1. Der Aufwand für diese Anpassung ist unabhängig von Text- und Patternlänge immer derselbe - also *konstant.*

Jetzt sind wir bereit für einen Pseudo-Code des Algorithmus.

Rabin-Karp: Pseudo-Code

Zunächst initialisieren wir die benötigte Zahl u und berechnen den Hashwert p für das Pattern $P[1 \ldots m]$, sowie t für die ersten m Zeichen des Textes $T[1 \ldots n]$. Dann folgt der Hauptteil:

```
u := |Σ|^(m−1) mod q;
p := 0; t := 0;                          /* Initiale                */
FOR i := 1 TO m DO                       /* Hashwertberechnung */
   p := (|Σ| · p + P[i]) mod q;
   t := (|Σ| · t + T[i]) mod q
ENDFOR;                                  /* Hauptteil */
FOR s := 0 TO n − m DO
   IF p = t THEN test(s) ENDIF;    /* ggfs. Ausgabe s */
   IF s < n − m THEN
      t := ((t − T[s + 1] · u) · |Σ| + T[s + m + 1]) mod q
   ENDIF
ENDFOR
```

Rabin-Karp: Analyse

Die Anzahl der Hashwertberechnungen in beiden Teilen zusammen ist $m + n$, der Aufwand hierfür also in $O(m + n)$. Dazu kommt für jede *Kollision*, d.h. für jedes s, für das $p = t$ erfüllt ist, ein Vergleich mit dem Pattern P, wofür $O(m)$ Schritte nötig sind.

Bei k Kollisionen erhalten wir also eine Gesamtrechenzeit von $O(m + n) + k \cdot O(m)$.

Unter der realistischen Annahme, dass $q > m$ ist und dass die Hashwerte zufällig verteilt im Bereich $0, \ldots, q - 1$ liegen, erhalten wir damit eine lineare Gesamtrechenzeit.

Theoretisch kann für den Rabin-Karp Algorithmus ein worst case im Bereich $\Theta(m \cdot n)$ auftreten – in der Realität erweist sich dieses Verfahren aber als sehr schnell.

Rabin-Karp: Beispiel

Wir suchen das Pattern adac in folgendem Text:

cdcbadccbaadcaadddccaaadacbaab

Dazu schreiben wir den Text zweimal untereinander, aber um $|\Sigma| = 4$ Stellen versetzt, und beachten, dass der Hashwert von adac $0 \cdot 64 + 3 \cdot 16 + 0 \cdot 4 + 2 \cdot 1 \bmod 11 = 50 \bmod 11 = 6$ ist.

```
  cdcbadccbaadcaadddccaaadacbaab
cdcbadccbaadcaadddccaaadacbaab
  984132X11314X4818167316331X
```

An den beiden markierten Stellen werden die Muster ccaa bzw. adac getestet.

Die Werte der Ziffern sind der Reihe nach 64, 16, 4 und 1, und alle Rechnungen werden mod 11 durchgeführt.

Die angegebenen Hashwerte berechnen wir an der Tafel.