Für einen Graphen mit m Kanten und n Knoten ergibt sich in obiger (ausführlicher) Codierung ein Platzbedarf von

$$(6m-1) + 2m(\log_{10} n + 1).$$

Dabei kann man an ein paar Stellen sparen (z.B. „v" oder „{", „}" weglassen) aber auch mehr Platz investieren (z.B. in ASCII codieren bzw. binär statt dezimal codieren). So wäre auch

$$(2m-1) + 2m(\log_2 n + 1) \quad \text{denkbar.}$$

Was ist wirklich wichtig dabei?!

(i) Die Kantenliste ist sparsamer als die Inzidenzmatrix, denn wenn n nicht zu klein ist, dann ist

$$n \geq 2 + 2(\log_2 n + 1).$$

(was heißt „nicht zu klein"? $n \geq 8$ reicht!)

Also ist auch

$$mn > (2m-1) + 2m(\log_2 n + 1).$$

(ii) Unabhängig von der Codierung ist für die Größe des Speicherplatzes der zweite Ausdruck wichtig, denn

$$2m(\log_2 n + 1) \leq (2m-1) + 2m(\log_2 n + 1)$$
$$\leq 4m(\log_2 n + 1) \quad \text{für } n \geq 2.$$

(iii) Letztlich kommt es also gar nicht so sehr auf die Vorfaktoren an (die sind codierungsabhängig!), sondern auf den Ausdruck

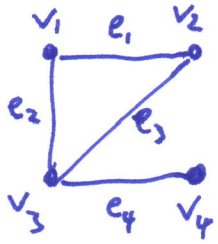
$$m \log n.$$

(iv) In $m \log n$ steckt das Wesen der Kantenliste:
Zähle für alle m Kanten die Nummern der beteiligten Knoten auf!

Wie wir im nächsten Abschnitt sehen werden, lohnt sich dafür eine eigene Notation:

Die Kantenliste benötigt $\Theta(m \log n)$ Speicherplatz.

(4) Adjazenzliste



V_1 : V_2, V_3 ;
V_2 : V_1, V_3 ;
V_3 : V_1, V_2, V_3 ;
V_4 : V_4 ;

Das ist etwas praktischer als die Kantenliste, wenn man für Graphenalgorithmen direkten Zugriff auf die Nachbarn eines Knotens benötigt! Man muss nicht die Nachbarn erst mühsam aus einer Liste heraussuchen.

Länge:

Jede Kante taucht doppelt auf, einmal für jeden Knoten.

So also:

$$2n + 4m + n(\log_{10} n + 1) + 2m(\log_{10} n + 1)$$

- d.h. $\Theta(n \log n + m \log n)$.

Im allgemeinen sind Graphen mit vielen isolierten Knoten (ohne Kanten!) uninteressant, d.h. z.B. $m \geq \frac{n}{2}$

$m \geq n$ o.ä.

Also wieder $\Theta(m \log n)$.

Jetzt wollen wir noch etwas mehr: den direkten Zugriff auf die Nachbarn der Knoten, wenn wir sie brauchen!

Also:

Liste (!) $\quad \underset{\downarrow}{v_2}, \underset{\downarrow}{v_3}; \underset{\downarrow}{v_1}, \underset{\downarrow}{v_3}; \underset{\downarrow}{v_1}, \underset{\downarrow}{v_2}, \underset{\downarrow}{v_4}; \underset{\downarrow}{v_3}$

Zugriff auf Nachbarn jeweils ab Semikolon.

Algorithmisch: Gehe Liste durch, zähle Semikolons → das dauert

Datenstruktur: Speichere die Stelle ab, an der die Nachbarn von v_3 zu finden sind.

→ Zeiger (engl. "Pointer")

Bekannt bei Webseiten:

Speicherinhalt : z.B. YouTube-Video (etliche MB)

Zeiger : URL (einige Byte)

„Man muss nicht alles wissen, man muss nur wissen wo's steht!"

Für den direkten Zugriff brauchen wir n Zeiger; jeder codiert eine Speicherzelle, d.h. die Nummer eines Bits in der Liste.

So ein Zeiger braucht also selber

$$\log_2\left(2n + 4m + n(\log_2 n + 1)\right) + 2m\left(\log_2 n + 1\right)$$

für $n \geq 10$, $m \geq n$

$$\leq \log_2\left(9m\left(\log_2 n + 1\right) + 1\right)$$

$$\leq \log_2 9 + \log_2 m + \log_2\left(\log_2 n + 1\right) + 1 \leq 2\log_2 m$$

Bits

Insgesamt benötigen wir also

$$\Theta(n \log_2 m) \quad \text{Bits.}$$

Jetzt ist $m \leq n^2$,

also $\log_2 m \leq \log_2 n^2 = 2 \log_2 n$,

d.h. $n \log m \leq 2n \log n$,

und der insgesamt verbrauchte Speicherplatz ist

$$\Theta(n \log m + m \log n) = \Theta(m \log n).$$

3.7 Wachstum von Funktionen

Im letzten Abschnitt haben wir Funktionen abgeschätzt und auf die „wesentlichen" Bestandteile reduziert, um Größenordnungen und Wachstumsverhalten zu beschreiben. Ein bisschen formaler:

Definition 3.9 (Θ-Notation)

Seien $f, g : \mathbb{N} \to \mathbb{R}$ Funktionen.

Dann gilt $f \in \Theta(g) :\Leftrightarrow$ Es gibt positive Konstanten c_1, c_2, n_0 mit

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0.$$

Man sagt: f wächst asymptotisch in derselben Größenordnung wie g.

Beispiele: $2n^2 - 1 \in \Theta(n^2)$

$$\frac{n^3}{1000} + n^2 + n \log n \in \Theta(n^3)$$