RUB

Vorlesung Datenstrukturen

Graphdarstellungen

Maike Buchin

20.6.2017

Graphen

RUB

Motivation: Graphen treten häufig als Abstraktion von Objekten (Knoten) und ihren Beziehungen (Kanten) auf.

Beispiele: soziale Netzwerke, Strassennetzwerke, Zitationsgraph

Beim **Zitationsgraph** repräsentieren Knoten wissenschaftliche Artikel und eine gerichtete Kanten gibt an, dass ein Artikel einen anderen zitiert. Hier stellt sich z.B. die Frage, wie man effizient eingehende Kanten bestimmt?

Die Effizienz dieser (und anderer) elementarer Operationen hängt stark von der **Darstellung** des Graphen ab.

Wir legen den Schwerpunkt auf **gerichtete Graphen**. da sich ungerichtete als doppelt gerichtete darstellen lassen.
Viele Datenstrukturen erlauben auch Mehrfachkanten und Schleifen darzustellen.

Operationen

RUB

folgende Operationen auf Graphen sind wünschenswert:

- *Zugriff auf Informationskomponente:* Knoten oder Kanten
- *Navigation:* Zugriff auf ein- und ausgehende Kanten
- *Kantenabfragen:* Ist (u, v) eine Kante?
- *Konstruktion, Umwandlung, Ausgabe* des Graphen
- Aktualisierung: Einfügen & Löschen von Knoten & Kanten

Wir betrachten verschiedene Darstellungen, die diese Operationen verschieden effizient umsetzen

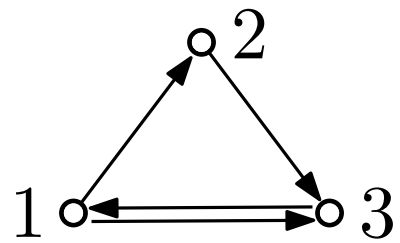
Notation: es ist immer $m = \#\text{Kanten}$ und $n = \#\text{Knoten}$
Es ist immer $0 \leq m \leq n^2$, also $m \in O(n^2)$.
Häufig ist aber auch $m \in O(n)$, d.h. G ist *dünn besetzt.*

8.1 Ungeordnete Kantenfolgen

RUB

Einfachste Darstellung: ungeordnete Liste der Kanten.
Jede Kante speichert zwei Knotenindizes und mögliche weitere Information (z.B. Kantengewichte).
Diese Darstellung wird häufig für Ein- und Ausgabe in Algorithmen benutzt.

Beispiel:



$$\{(1,2),(2,3),(3,1),(1,3)\}$$

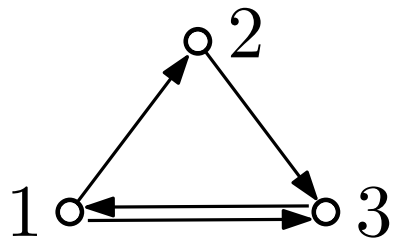
Eigenschaften:

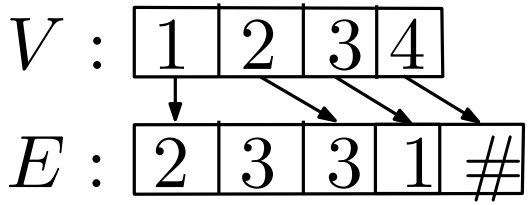
- Navigation: Zugriff auf Kanten in $O(m)$ – nicht sehr effizient, daher ist diese Darstellung für viele Algorithmen auf Graphen ungeeignet
- Aktualisieren: Kanten lassen sich in $O(1)$ Zeit hinzufügen (ohne auf Vorhandensein zu prüfen, sonst $O(m)$)
- Speicherbedarf: $2m = O(m)$

8.2 Adjanzenzarrays – Statische Graphen

Ziel: schneller Zugriff auf Kanten in einem Knoten. Speichere dazu für jeden Knoten die von ihm ausgehenden Kanten in einem Array. Falls G statisch können diese zu einem großen Array konkateniert werden, mit Dummyknoten am Ende.

Beispiel:





Dummyeintrag:
$V[n+1]$ zeigt auf
$E[m+1]$

Sind mit den Kanten keine weiteren Informationen verknüpft, speichere nur Zielknoten.

Eigenschaften:

- Navigation: Zugriff auf ausgehende Kanten in $O(\deg(v))$
- Aktualisieren: Einfügen/Löschen in Array (relativ teuer)
- Speicherbedarf: $n + m + 1 = O(m + n)$

8.3 Adjazenzlisten – Dynamische Graphen

RUB

Wenn Kanten hinzugefügt oder gelöscht werden sollen, eignen sich Listen besser zum Abspeichern der ausgehenden Kanten.
Konkret: einfach oder doppelt verkettete Listen ohne Dummyknoten.

Eigenschaften:

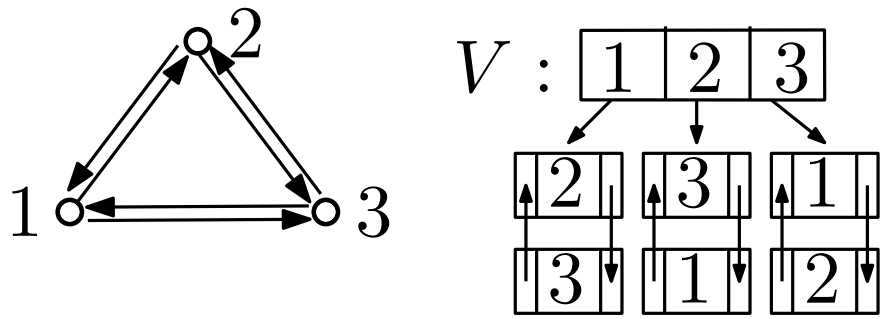
- Navigation: Zugriff auf ausgehende Kanten in $O(\deg(v))$
- Aktualisieren: Einfügen/Löschen in Listen (recht effizient)
- Speicherbedarf: $O(m+n)$

8.3 Adjazenzlisten – Dynamische Graphen

RUB

Wenn Kanten hinzugefügt oder gelöscht werden sollen, eignen sich Listen besser zum Abspeichern der ausgehenden Kanten.
Konkret: einfach oder doppelt verkettete Listen ohne Dummyknoten.

Beispiel:



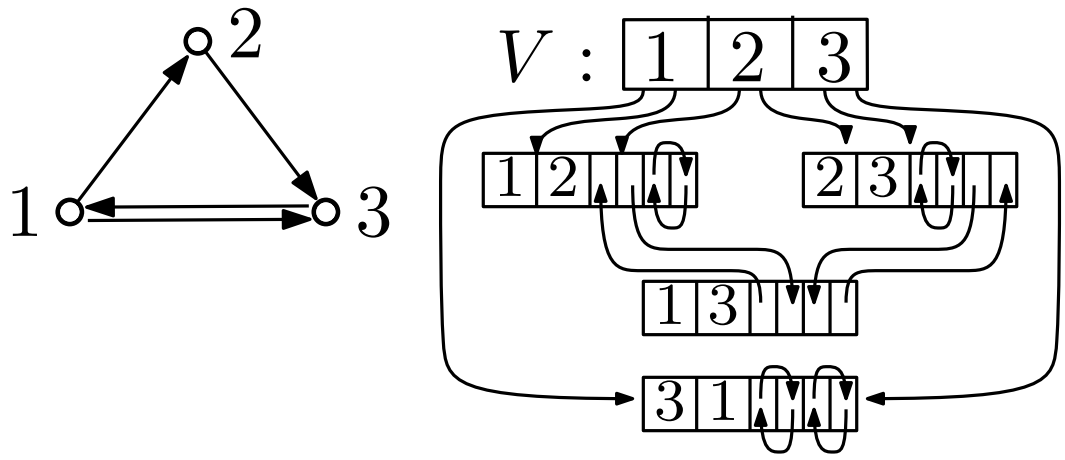
Im doppelt gerichteten Graphen kann Zugriff auf Gegenkante oder Kanteninformation durch weitere Zeiger realisiert werden.

8.3 Adjazenzlisten – Dynamische Graphen

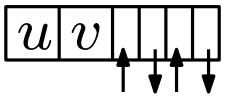
RUB

Wenn Kanten hinzugefügt oder gelöscht werden sollen, eignen sich Listen besser zum Abspeichern der ausgehenden Kanten.
Konkret: einfach oder doppelt verkettete Listen ohne Dummyknoten.

Beispiel:



Verzeigertes Kantenobjekt:



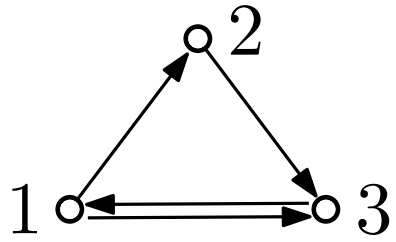
Zeiger auf ein-/ausgehende Kanten

Möchte man im (gerichteten) Graphen ebenfalls Zugriff auf Eingangskanten haben, eignen sich *verzeigerte Kantenobjekte*.

8.4 Adjazenzmatrizen

Speichere zu G die $n \times n$ Matrix A mit $A_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$

Beispiel:



$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Eigenschaften:

- Navigation: Zugriff auf ein-/ausgehende Kanten in $O(n)$
- Aktualisieren: $O(1)$ (sehr effizient)
- Speicherbedarf: $\Theta(n^2)$ (hoch für dünn besetzte Graphen)

8.5 Implizite Darstellungen

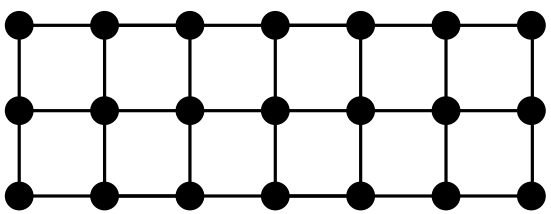
RUB

Manche Graphen haben eine spezielle Struktur, die eine einfachere Darstellung erlaubt.

Z.B. **Gittergraph** $G_{k,\ell}$ mit Knoten $V = [1 \dots k] \times [1 \dots \ell]$ und Kanten $E = \{((i,j),(i,j')) \in V^2 : |j - j'| = 1\}$
$\cup \{((i,j),(i',j)) \in V^2 : |i - i'| = 1\}$

Dieser ist durch Angabe der Parameter k, ℓ bereits vollständig bestimmt.

Beispiel: $G_{7,3}$



2.9.1 Test auf Azyklizität

Beobachtung: Ein Knoten v mit outdeg$(v) = 0$ kann nicht auf einem Kreis liegen.

Algorithmus: Streiche sukzessive alle Knoten $v \in V$ mit outdeg$(v) = 0$. Entweder:

- leerer Graph $\rightarrow$ azyklisch
- nicht-leer; jeder Knoten hat outdeg $> 0 \rightarrow$ enthält Zykel

Korrektheit:

- folgt aus Beobachtung
- Zykel lässt sich leicht finden: starte beliebig bis ein bereits besuchter Knoten betreten wird

Laufzeit: abhängig von der Darstellung des Graphen; bei geeigneter Darstellung in $O(n + m)$ Zeit.