ADS: Algorithmen und Datenstrukturen 2

Teil II

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

13. April 2011

Organisatorisches

- Aktuelle Information stets auch online unter

  `http://www.bioinf.uni-leipzig.de/`
  → Teaching → Current classes → ADS2

- Einteilung der Seminargruppen ist online.
- Für alle, die es versäumt haben, sich anzumelden:

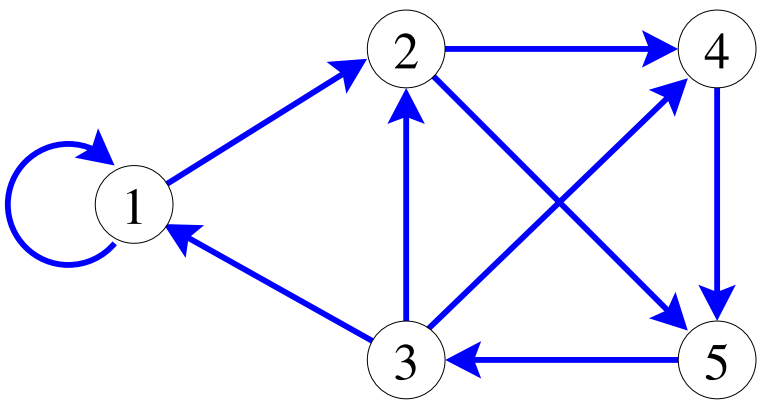
  Anmeldung nachträglich bis 14.04. (Do) 10 Uhr

- Seminare beginnen in der ersten Mai-Woche.
- Abgabe der ersten Übungsserie am 27.04.

Gerichteter Graph

Ein Tupel (V, E) heißt *gerichteter Graph* (Digraph), wenn V eine endliche Menge und E eine Menge geordneter Paare von Elementen in V ist.
V heißt *Knotenmenge*, die Elemente von V heißen *Knoten*.
E heißt *Kantenmenge*, die Elemente von E heißen *Kanten*.
Eine Kante (v, v) heißt *Schleife*.

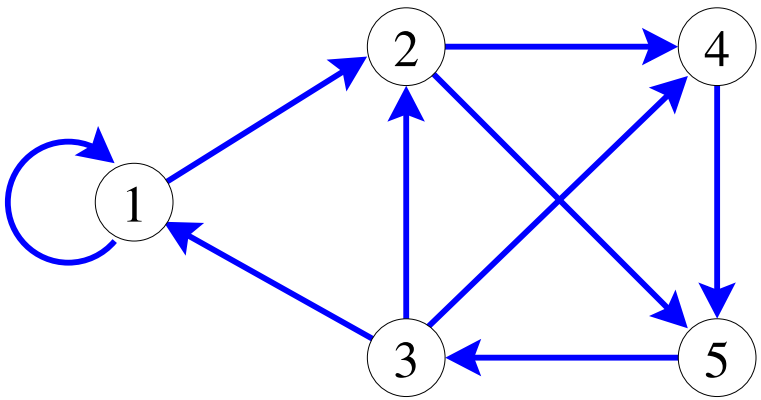
Beispiel: $V = \{1, 2, 3, 4, 5\}$, $E = \{(1, 1), (1, 2), (2, 4), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5), (4, 5)\}$



Vorgänger, Nachfolger, Grad

Sei $G = (V, E)$ ein gerichteter Graph und $v \in V$.

- $u \in V$ heißt *Vorgänger* von v, wenn $(u, v) \in E$.
- Mit $\text{pred}(v) := \{u \in V | (u, v) \in E\}$ bezeichnen wir die Menge der Vorgänger von v.
- Der *Eingangsgrad* von v ist $\text{eg}(v) = |\text{pred}(v)|$
- $w \in V$ heißt *Nachfolger* von v, wenn $(v, w) \in E$.
- Mit $\text{succ}(v) := \{w \in V | (v, w) \in E\}$ bezeichnen wir die Menge der Nachfolger von v.
- Der *Ausgangsgrad* von v ist $\text{ag}(v) = |\text{succ}(v)|$



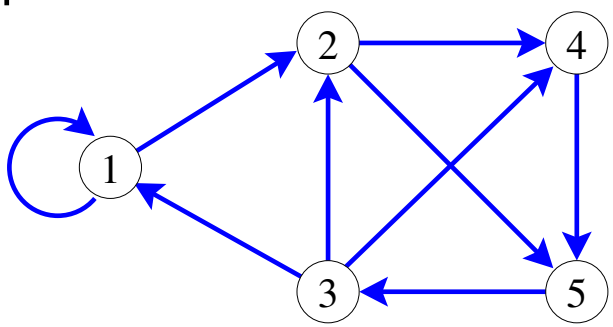
$\text{eg}(3) = 1$, $\text{pred}(3) = \{5\}$
$\text{ag}(3) = 3$, $\text{succ}(3) = \{1, 2, 4\}$

Speicherung von Graphen: Adjazenzmatrix

Ein Graph $G = (V, E)$ mit $|V| = n$ wird in einer Boole'schen $n \times n$-Matrix $A = (a_{ij})$, mit gespeichert, wobei

$$a_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

Beispiel:



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

Speicherplatzbedarf $O(n^2)$

- 1 Bit pro Position (statt Knoten/Kantennummern)
- unabhängig von Kantenmenge
- für ungerichtete Graphen ergibt sich symmetrische Belegung (Halbierung des Speicherbedarfs möglich)

Speicherung von Graphen in Listen

Knoten- und Kantenlisten

- Speicherung von Graphen als Liste von Zahlen (z.B. in Array oder verketteter Liste)
- Knoten werden von 1 bis n durchnummeriert; Kanten als Paare von Knoten

Kantenliste

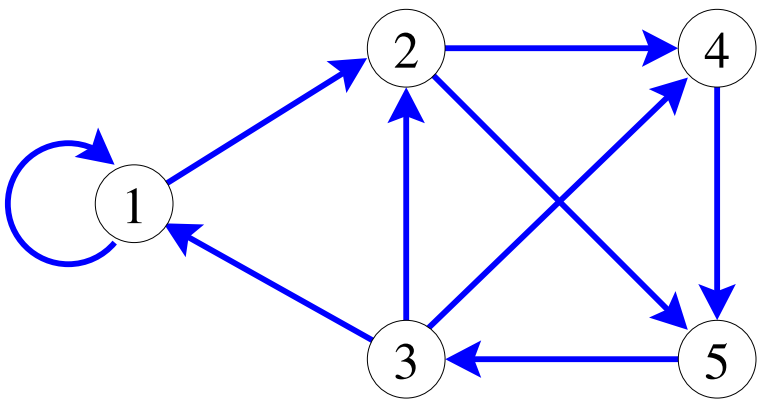
- Liste: Knotenzahl, Kantenzahl, Liste von Kanten (je als 2 Zahlen)
- Speicherbedarf: $2 + 2m$ (m = Anzahl Kanten)

Knotenliste

- Liste: Knotenzahl, Kantenzahl, Liste von Knoteninformationen
- Knoteninformation: Ausgangsgrad und Nachfolger $\mathrm{ag}(v), s_1, s_2, \ldots, s_{\mathrm{ag}(v)}$
- Speicherbedarf: $2 + n + m$

Adjazenzlisten

- verkettete Liste der n Knoten (oder Array-Realisierung)
- pro Knoten: verkettete Liste der Nachfolger (repräsentiert die von dem Knoten ausgehenden Kanten)
- Speicherbedarf: $n + m$ Listenelemente



$$
\begin{array}{l}
1 \to 1 \to 2 \\
\downarrow \\
2 \to 4 \to 5 \\
\downarrow \\
3 \to 1 \to 2 \to 4 \\
\downarrow \\
4 \to 5 \\
\downarrow \\
5 \to 3
\end{array}
$$

Speicherung von Graphen: Vergleich

Komplexitätsvergleich

Operation	Adj.-matrix	Kantenliste	Knotenliste	Adjazenzliste
Einfügen Kante	$O(1)$	$O(1)$	$O(n+m)$	$O(1)/O(n)$
Löschen Kante	$O(1)$	$O(m)$	$O(n+m)$	$O(n)$
Einfügen Knoten	$O(n^2)$	$O(1)$	$O(1)$	$O(1)$
Löschen Knoten	$O(n^2)$	$O(m)$	$O(n+m)$	$O(n+m)$

- Löschen eines Knotens löscht auch zugehörige Kanten
- Änderungsaufwand abhängig von Realisierung der Adjazenzmatrix und Adjazenzliste

Welche Repräsentation geeigneter ist, hängt vom Problem ab:

- Frage: Gibt es Kante von a nach b? $\rightarrow$ Matrix
- Durchsuchen von Knoten in durch Nachbarschaft gegebener Reihenfolge $\rightarrow$ Listen

Kantenfolgen, Pfade, Zyklen

Sei $G = (V, E)$ gerichteter Graph, $l \in \mathbb{N} \cup \{0\}$ und $k = (v_0, v_1, \ldots, v_l) \in V^{l+1}$.

- k heißt *Kantenfolge* (oder *Weg*) der Länge l von v_0 nach v_n, wenn für alle $i \in \{1, \ldots, l\}$ gilt: $(v_{i-1}, v_i) \in E$
- $v_1, \ldots, v_{l-1}$ sind die *inneren* Knoten von k. Ist $v_0 = v_n$, so ist die Kantenfolge *geschlossen*.
- k heißt *Kantenzug*, wenn k Kantenfolge ist und für alle $i, j \in \{0, \ldots, l-1\}$ mit $i \neq j$ gilt: $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$.
- k heißt *Pfad* der Länge l, wenn k eine Kantenfolge ist und für alle $i, j \in \{0, \ldots, l\}$ mit $i \neq j$ gilt: $v_i \neq v_j$.
- k heißt *Zyklus* (oder *Kreis*), wenn $(v_1, \ldots, v_l)$ ein Pfad der Länge $l-1$ ist und $v_0 = v_l$.
- k heißt *Hamiltonscher Zyklus*, wenn k Zyklus ist und $l = |V|$.

Gerichtete azyklische Graphen (DAGs)

Sei $G = (V, E)$ ein gerichteter Graph. G heißt *azyklisch* wenn für jede Kantenfolge k in G gilt: k ist kein Zyklus.

Beobachtung: G azyklisch
$\Rightarrow$ Es gibt Knoten $x, y \in V$ mit $\text{eg}(x) = \text{ag}(y) = 0$.

Beweisskizze: Annahme $\text{ag}(y) > 0$ für alle $y \in V$. Dann gibt es zu jedem $l \in \mathbb{N}$ eine Kantenfolge der Länge l, denn jede Kantenfolge der Länge $l - 1$ kann durch Hinzunahme eines Nachfolgers des letzten Knotens auf Länge l gebracht werden. In einer Kantenfolge der Länge $l > |V|$ muss mindestens ein Knoten mehrfach auftreten, so dass die Kantenfolge einen Zyklus enthält. Widerspruch, denn G ist azyklisch.
(Beweis für Eingangsgrad analog).

Topologische Sortierung

Eine *topologische Sortierung* eines Digraphen $G = (V, E)$ ist eine bijektive Abbildung

$$s : V \to \{1, \ldots, |V|\}$$

so dass für alle $(u, v) \in E$ gilt:

$$s(u) < s(v) \ .$$

Satz: Ein Digraph G besitzt eine topologische Sortierung, genau dann wenn G azyklisch ist.
Beweis: "$\Rightarrow$" klar. "$\Leftarrow$" durch Induktion über $|V|$.
Anfang: $|V| = 1$, keine Kante, bereits topologisch sortiert.
Schritt: $|V| = n$. Da G azyklisch ist, $v \in V$ mit $\mathrm{ag}(v) = 0$. Der Graph $G' = G - \{v\}$ ist ebenfalls azyklisch und hat $n - 1$ Knoten. Nach Induktionsannahme hat G' eine topologische Sortierung $s : V \setminus \{v\} \to \{1, \ldots, n-1\}$, die wir mit $s(v) = n$ zu einer topologischen Sortierung für G erweitern.

Algorithmus für topologische Sortierung

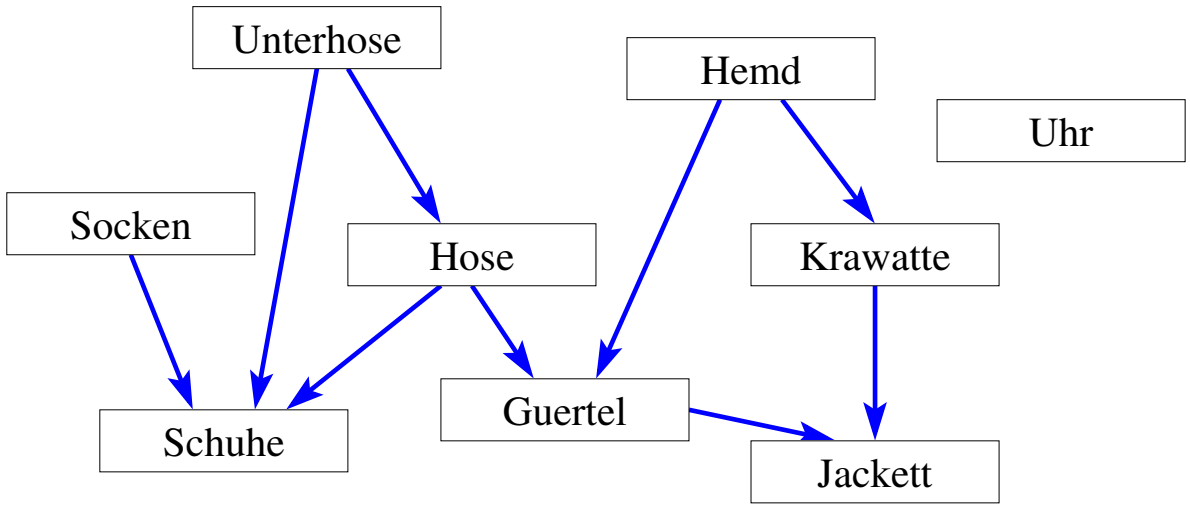
Test auf Zyklenfreiheit und ggf. Bestimmung einer topologischen Sortierung s für $G = (V, E)$

```
TS(G);
  i=|V|;
  solange (G hat einen Knoten v mit ag(v)=0)
  {
    s[v]=i;
    G=G-{v};
    i=i-1;
  }
  falls (i==0) Ausgabe("G ist zyklenfrei");
  sonst        Ausgabe("G hat Zyklus");
```

- (Neu-)Bestimmung des Ausgangsgrades aufwendig
- Effizienter: aktuellen Ausgangsgrad zu jedem Knoten speichern

Anwendungsbeispiel: Topologische Sortierung

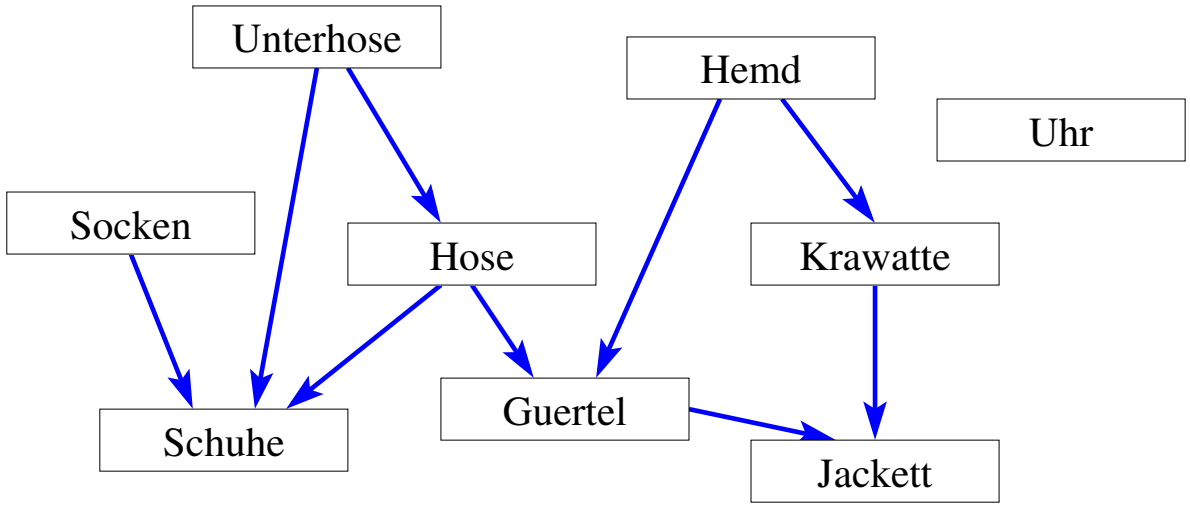
Task scheduling beim Anziehen am Morgen:



Kante (u, v) gibt zeitliche Abfolge vor: u vor v

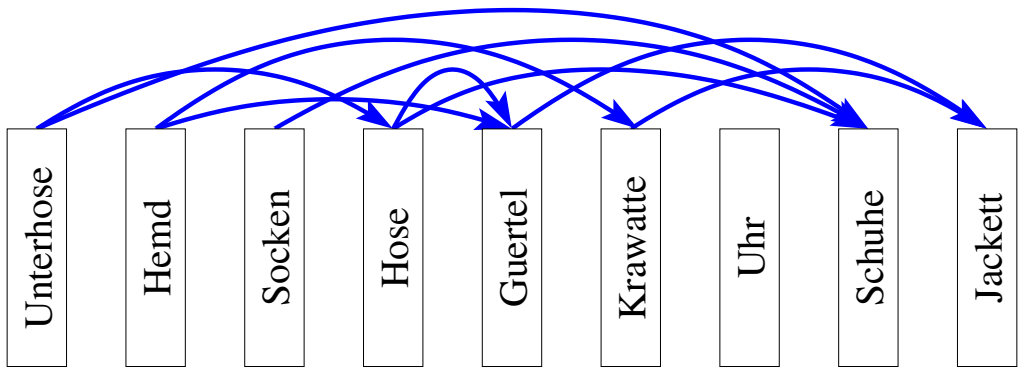
Anwendungsbeispiel: Topologische Sortierung

Task scheduling beim Anziehen am Morgen:



Kante (u, v) gibt zeitliche Abfolge vor: u vor v

Eine topologische Sortierung:



Transitive Hülle

Erreichbarkeit von Knoten

- welche Knoten sind von einem gegebenen Knoten aus erreichbar?
- gibt es Knoten, von denen aus alle anderen erreicht werden können?
- Bestimmung der transitiven Hülle ermöglicht Beantwortung solcher Fragen

Ein Digraph $G^* = (V, E^*)$ heißt *reflexive, transitive Hülle* (kurz: Hülle) eines Digraphen $G = (V, E)$, wenn für alle $u, v \in V$ gilt:

$$(u, v) \in E^* \Leftrightarrow \text{Es gibt einen Pfad von } u \text{ nach } v \text{ in } G$$

Transitive Hülle: naiver Ansatz

Naiver Algorithmus zur Berechnung von Kanten der reflexiven transitiven Hülle

```
boolean[][] A= {...};
for (int i=1; i<=A.length; i++) A[i]=true;

for (int i=1; i<=A.length; i++)
  for (int j=1; j<=A.length; j++)
    if (A[i][j])
      for (int k=1; k<=A.length; k++)
        if (A[j][k]) A[i][k]=true;
```

- Es werden i.a. nur Pfade der Länge 2 bestimmt.
- Komplexität $O(n^3)$

Transitive Hülle: Warshall-Algorithmus

Einfache Modifikation liefert vollständige transitive Hülle:

```
boolean[][] A= {...};
for (int i=1; i<=A.length; i++) A[i]=true;

for (int j=1; j<=A.length; j++)
  for (int i=1; i<=A.length; i++)
    if (A[i][j])
      for (int k=1; k<=A.length; k++)
        if (A[j][k]) A[i][k]=true;
```

Korrektheit des Warshall-Algorithmus

- Induktionshypothese $P(j)$: gibt es zu beliebigen Knoten i und k einen Pfad $(i, v_1, v_2, v_3, \ldots, v_{l-1}, j)$ mit inneren Knoten $v_1, v_2, \ldots, v_{l-1} \in \{1, ...j\}$, so ist nach dem Durchlauf j der äußeren Schleife $A[i][k] = \text{true}$.
- Induktionsanfang, $j = 1$: Falls $A[i][1]$ und $A[1][k]$ gilt, wird in der Schleife mit $j = 1$ auch $A[i][k] = \text{true}$ gesetzt
- Induktionsschluss: Wir nehmen an, daß $P(j-1)$ bereits gezeigt ist, und folgern daraus $P(j)$. Existiere ein Pfad $(i, v_1, \ldots, v_{l-1}, k)$ mit inneren Knoten $v_1, v_2, \ldots, v_{l-1} \in \{1, ...j\}$. Wenn diese inneren Knoten nicht j enthalten, so folgt mit $P(j-1)$ bereits $A[i][k] = \text{true}$. Anderenfalls gibt es genau einen Index r mit $v_r = j$. Daher sind $(i, v_1, \ldots, v_r)$ und $(v_r, v_{r+1}, \ldots, k)$ Pfade mit inneren Knoten in $\{1, \ldots, j\}$. Wegen $P(j-1)$ sind daher $A[i][j] = \text{true}$ und $A[j][k] = \text{true}$ nach Durchlauf der Schleife mit $j-1$. Im Durchlauf j wird daher $A[i][k] = \text{true}$ gesetzt, q.e.d.