Graphen

- ❑ Aus AuDI kennen wir bereits Graphalgorithmen
  - Kruskals Algorithmus
  - Pfadsuche in Graphen
  - Traveling Salesman Problem
- ❑ im folgenden werden behandelt
  - Datenstrukturen für Graphen
  - Graphalgorithmen (Breiten- vs. Tiefensuche, kürzeste Pfade, ...)
- ❑ Zur Wiederholung: Wie ist ein Graph definiert?

- ❑ Standardverfahren zur Implementierung von Graphen
  - Adjazenzmatrizen
  - Adjazenzlisten

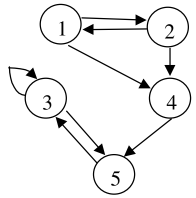
Adjazenzmatrizen

- ❑ Adjazenzmatrix
  - Sei $V = \{1, ..., n\}$, dann ist die Adjazenzmatrix für $G = (V, E)$ eine boolsche n x n Matrix A mit

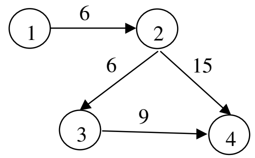
$$A_{ij} = \begin{cases} true & falls (i, j) \in E \\ false & sonst \end{cases}$$

Meist werden *true* und *false* durch 1 und 0 dargestellt.

- ❑ Beispiel
  - G1



G2



  - Wie sehen die zu den obigen Graphen gehörenden Adjazenzmatrizen aus?

Adjazenzmatrizen

- ❑ Implementierung
  - Durchnumerieren der Knoten: $V = 1 ... n$
  - Darstellung der Kanten als boolsche Matrix
    - ⇨

- ❑ Voraussetzung
  - Größe des Graphen ist bekannt und konstant
- ❑ Eigenschaften
  - Schnell. Einfache Implementierung vieler Graphalgorithmen.
    - ⇨ Zugriff auf eine Kante von v nach w in O(1).
  - Speicherintensiv
    - ⇨ Platzbedarf von $O(n^2)$ ist vor allem bei geringer Kantenzahl unbefriedigend
  - Initialisierung
    - ⇨ benötigt eine Laufzeit von $\Theta(n^2)$

Adjazenzlisten

- ❑ Für jeden Knoten wird eine Liste seiner Nachfolger verwaltet.
- ❑ Wie sieht das für das Beispiel G1 aus?

- ❑ Implementierung von Adjazenzlisten in Java

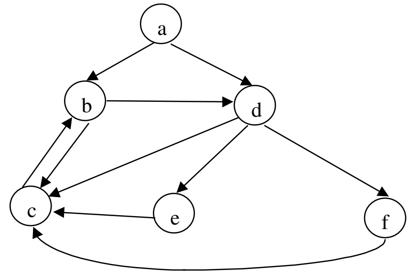
- ❑ Markierte Graphen
  - hier können die Listenelemente weitere Einträge enthalten

Adjazenzlisten

- Eigenschaften
  - geringer Speicherbedarf
    - ⇨ beträgt O(n+e), mit $n = |V|$ und $e = |E|$
  - Nachbar eines Knotens kann in linearer Zeit gefunden werden. Warum nicht in konstanter Zeit wie bei Adjazenzmatrizen?

Tiefensuche

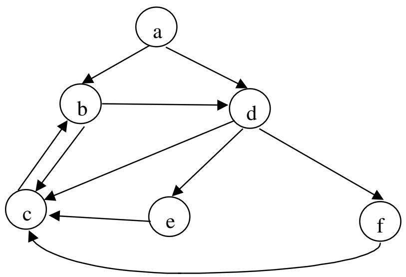
- Systematisches Durchsuchen eines Graphen
  - Suche beginnt am Startknoten
  - Dann jeweils Fortsetzung der Suche bei den Kindern jedes aufgesuchten Knotens
- Beispiel



- Wie sieht die Durchsuchungsfolge der Knoten aus?
  - Annahme: Knoten jeder Adjazenzliste sind alphabetisch geordnet.
- Wie hoch ist die Laufzeit?

Breitensuche

- Systematisches Durchsuchen eines Graphen
  - Suche beginnt an der Wurzel
  - Der Graph wird in "Schichten" durchsucht, d.h. zunächst werden alle Knoten der gleichen Ebene durchsucht, erst danach werden die Nachfolger besucht.
- Beispiel:



- Wie sieht die Durchsuchungsfolge der Knoten aus?
  - Annahme: Knoten jeder Adjazenzliste sind alphabetisch geordnet
- Wie hoch ist die Laufzeit?

Topologisches Sortieren

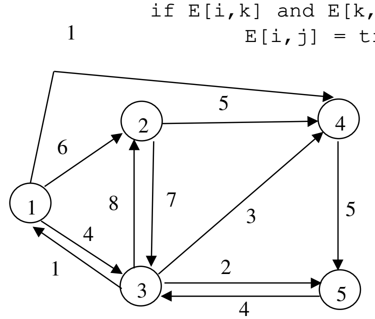
- Gegeben
  - gerichteter, azyklischer Graph
- Topologische Sortierung eines gerichteten Graphen
  - Anordnung seiner Knoten $V = \{v_1, v_2, ..., v_n\}$ so, daß für alle Kanten (v_i, v_j) gilt: $i < j$.
- Satz
  - Ein Graph G ist azyklisch genau dann, wenn er sich topologisch sortieren läßt.
- Algorithmus zur Konstruktion einer topologischen Sortierung
  - Zur topologischen Sortierung kann die Tiefensuche eingesetzt werden
  -
```
Für alle Knoten: farbe[v] = weiß
für alle Knoten
  falls farbe[v] = weiß {
          Tiefensuche(v);
  }
```
  - Die umgekehrte Reihenfolge des Schwarz-werdens stellt eine topologische Sortierung des Graphen dar.
  - Wie hoch ist die Komplexität?

Transitive Hülle

- ❑ Problemstellung
  - Welche Knoten sind von einem gegebenen Knoten aus erreichbar?
  - Sind von einem Knoten aus alle anderen erreichbar?
- ❑ Transitive Hülle
  - Graph G* = (V, E*) mit
    $(i, j) \in E^* \Leftrightarrow$ es gibt einen Pfad in G von i nach j
  - G^* ist berechenbar beispielsweise durch Breitensuche von jedem Startknoten aus.
- ❑ Grundlegendes
  - Gegeben
    - ⇨ $G = (V, E)$ mit $V = \{1, 2, ..., n\}$
  - Boolesche Variable
    - ⇨ $m_{i,j}^{k} = \begin{cases} 1, & \text{es gibt einen Pfad von i nach j, wobei alle zwischen i und j} \\ & \text{liegenden Knoten aus der Menge } \{1, ..., k\} \text{ stammen} \\ 0, & \text{sonst} \end{cases}$
  - Es gilt $(i, j) \in E^* \Leftrightarrow m_{i,j}^{k} = 1$
    - ⇨ daher kann man die transitive Hülle berechnen, indem man die Matrizen berechnet.

Algorithmus von Warshal

- ❑ Algorithmus basiert auf oben genannten Ideen
  - $A[1..n, 1..n]$ ist die Adjanzenzmatrix des Graphen G
  -
```
for k = 1 to n
  for i = 1 to n
          for j=1 to n
                  if E[i,k] and E[k,j] {
                          E[i,j] = true; }
```
- ❑ Beispiel



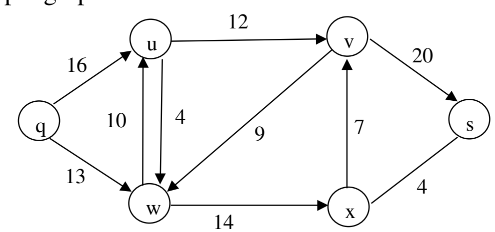
  - Wie sehen die "Momentaufnahmen" der Matrizen für dieses Beispiel aus?

Fluß in Netzwerken

- ❑ Gegeben
  - Gerichteter Graph G, in dem jede Kante (i, j) mit einer Kapazität $k(i, j)$ markiert ist. Für $(i, j) \notin E$ gilt $k(i, j) = 0$
  - G hat eine Quelle q von der man jeden Knoten erreicht und eine Senke s die von jedem Knoten erreichbar ist.
- ❑ Anschaulich
  - Man kann sich vorstellen, daß Wasser, Strom, Öl etc. fließt, wobei die Kantenkapazität das Volumen pro Kante bestimmt
- ❑ Folgende Bedingungen müssen erfüllt sein für einen zulässigen Fluß
  - Kapazitätsbedingung
    - ⇨ $f(i, j) <= k(i, j)$
  - Symmetriebedingung
    - ⇨ $f(i, j) = -f(j, i)$
  - Kirchhoffsches Gesetz
    - ⇨ $\forall i \in V - \{q, s\}: \sum_{i \in V} f(i, j) = 0$
    - ⇨ es geschehen also keine Wunder, sondern die Summe der Zu- und Abflüße ist gleich Null.
- ❑ Optimierungsaufgabe
  - Es soll ein Fluß gefunden werden, so daß $|f|$ maximal ist

Beispiel

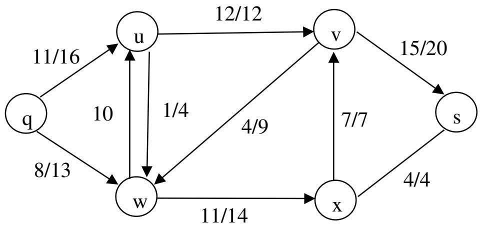
- ❑ Beispielgraph



- ❑ Wie sieht ein maximaler Fluß für dieses Beispiel aus?

Algorithmus von Ford-Fulkerson

- ❑ Ziel
  - Bestimmung des maximalen Flusses von *q* nach *s*
- ❑ Prinzipien
  - Restnetzwerk
    - ⇨ Gegeben sei ein Netzwerk *G* mit Kapazität *k* und ein zulässiger Fluß *f*. Das zugeordnete Restnetzwerk G_f mit einer zugeordneten Restkapazität $k_f(i,j) = k(i,j) - f(i,j)$ ist gegeben durch $G_f = (V, E_f)$, wobei $E_f = \{(i,j): k_f(i,j) > 0\}$.
    - ⇨ Wie sieht das Restnetzwerk dieses Beispiels aus?



  - Erweiterungspfad
    - ⇨ Einfacher (also zyklenfreier) Pfad *p* in G_f von *q* nach *s*. Für alle Kanten auf *p* gilt $k_f(i,j) > 0$. Restkapazität von *p* ist $\min\{k_f(i,j)\}$
    - ⇨ Existiert im obigen Beispiel ein Erweiterungspfad?

Algorithmus von Ford und Fulkerson

  - Min-Cut-Flow-Theorem
    - ⇨ Besagt folgendes: Wenn der bisherige Fluß nicht maximal ist, kann immer ein Erweiterungspfad gefunden werden.
- ❑ Algorithmus

```
maxflow = 0;
Berechne Restnetzwerk
while "es ist noch einen Pfad im Restnetzwerk von q nach s vorhanden"
  {
  Berechne Restnetzwerk
  Bestimme einen Erweiterungspfad im Restnetzwerk
  maxflow += Restkapazität
  }
gib f als maximalen Fluß aus
```

- ❑ Leistungsfähigkeit
  - Der Algorithmus sagt nichts über die Auswahl eines Erweiterungspfades aus.
    - ⇨ ungeschickte Auswahl kann zu recht langen Laufzeiten führen
  - Wie hoch ist der Aufwand des Algorithmus?

Erinnerung: TSP

- ❑ Travelling Salesman Problem
  - kürzeste Rundreise in einem Graphen mit gegebenen Kantenlängen (markiert)
- ❑ Lösungsmethode für exakte Lösung
  - Branch and Bound, das ist spezielles Backtracking auf Kantenmengen
- ❑ In jedem Aufruf wird eine Kante betrachtet. Zwei Möglichkeiten existieren:
  - Hinzunahme der Kante zur bisherigen Kantenmenge
  - Weglassen der Kante
- ❑ Kanten dürfen nur wann hinzugenommen werden?

- ❑ Falls Kantenmenge jeden Knoten enthält, wird mit bisher bester Lösung verglichen.
- ❑ sonst wird geprüft, ob Hinzunahme einer Kante die Kantenmenge bereits länger macht als bisher beste Rundreise
- ❑ oder ob die Kantenmenge überhaupt noch zu einer kürzeren Rundreise vervollständigt werden kann
- ❑ für letzteres addiert man für jeden Knoten, der noch weniger als zwei "Arme" hat, den bzw. die beiden kürzesten unbenutzten "Arme" und teilt die Summe durch 2
- ❑ Länge der Kantenmenge plus diese Zahl ist eine untere Schranke für jede Rundreise, die Kantemenge vervollständigt

TSP: Implementierung

```
❑ tsp(i: Kante, path_fragment: SET OF Kante); {
    if "path_fragment enthält alle Knoten" (
          if (length(path_fragment)<length(bestslution) {
                bestsolution := path_fragment; }
          else {  // probiere zuerst ohne Kante i
                if "Kante i kann weggelassen werden" {
                      if length_lowerbound(path_fragement)
                        < length(bestsolution) {
                            tsp(i+1, path_fragment)
                }}  // jetzt mit Kante i
                if "Kante i kann hinzugenommen werden" {
                      new_path_fragment :=
                      union(i+1, path_fragment); }
                if length(new_path_fragment)
                  < length(bestsolution) {
                      if
                       length_lowerbound(new_path_fragment)
                       < length(bestsolution) {
                            tsp(i+1, new_path_fragment)
    }}}}
```