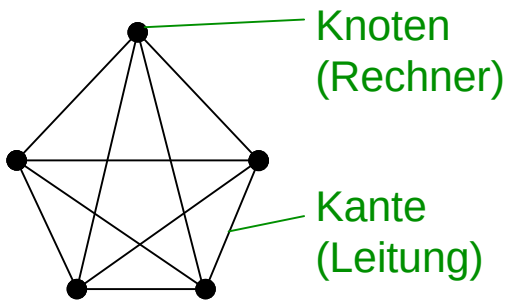
Teil 2:
Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

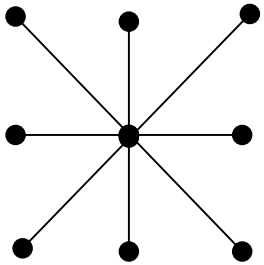
Kommunikationsnetze (1)

Menge von Rechnern, die mit Leitungen verbunden sind.

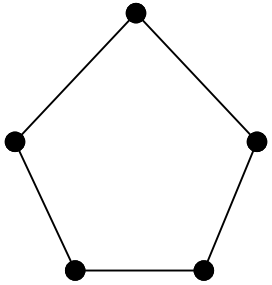
Beispiele für Kommunikationsnetze



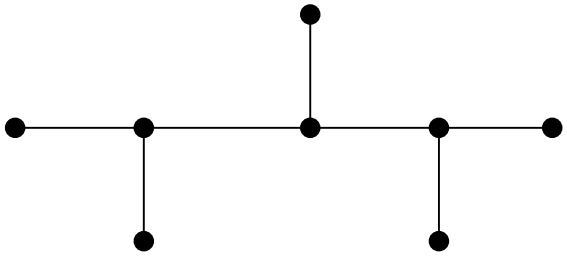
Vollständig vernetzte Struktur



Sternstruktur



Ringstruktur



Busstruktur

Kommunikationsnetze (2)

Zuverlässigkeit von Kommunikationsnetzen:

lässt sich bemessen durch:

- Verbindungszusammenhang:

  Minimale Anzahl von Verbindungen, bei deren Ausfall die Kommunikation zwischen irgendwelchen Knotenpaaren unterbrochen wird.

  Aufgabe:
  Bestimme den Verbindungszusammenhang für die Beispiele auf vorhergehender Seite.

- Knotenzusammenhang:
  Minimale Anzahl von Knoten, bei deren Ausfall die Kommunikation zwischen irgendwelchen Knotenpaaren unterbrochen wird.

  Aufgabe:

  Bestimme den Knotenzusammenhang für die Beispiele auf vorhergehender Seite.

Algorithmen → Flüsse in Netzwerken.

Routing-Problem:

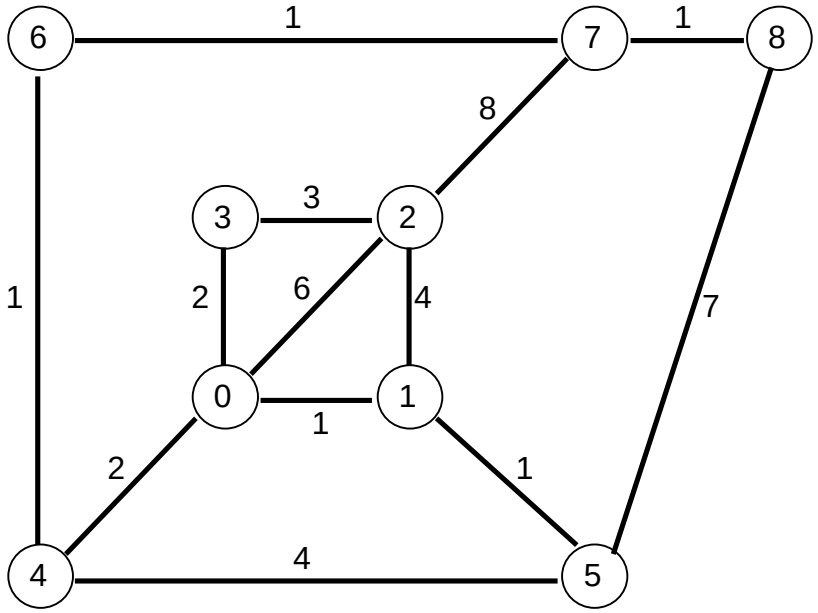
Bestimme von einem Rechnerknoten den günstigsten Weg zum Zielknoten. Die Güte einer Verbindungskante ist dabei beispielsweise durch seine Paketkapazität definiert.

Algorithmen → kürzeste Wege in Graphen

Routenplanung

Problemstellung

Finde in einem Straßennetz die kürzeste Verbindung (Länge bzw. Fahrtzeit) von einer Stadt A zu einer Stadt B.



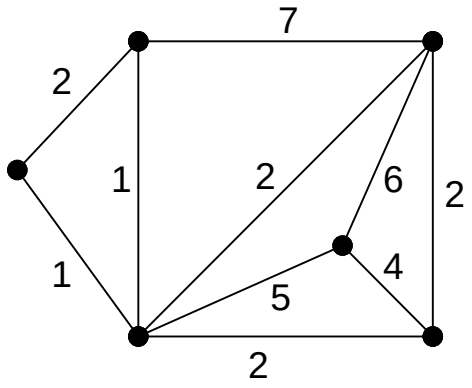
Algorithmen → kürzeste Wege in Graphen

Netzplanung

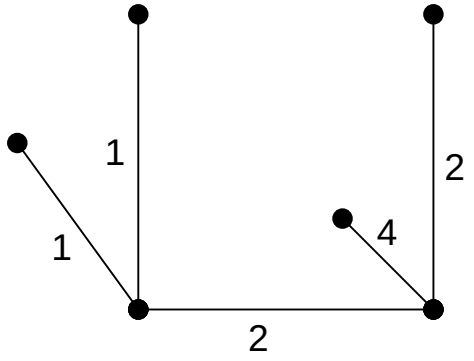
Problemstellung

Zwischen n Orten soll ein Versorgungsnetz (Kommunikation, Strom, Wasser, etc.) aufgebaut werden, so dass je 2 Orte direkt oder indirekt miteinander verbunden sind. Die Verbindungskosten zwischen 2 Orte seien bekannt.

Gesucht ist ein Versorgungsnetz mit den geringsten Kosten.



Orte mit Verbindungskosten

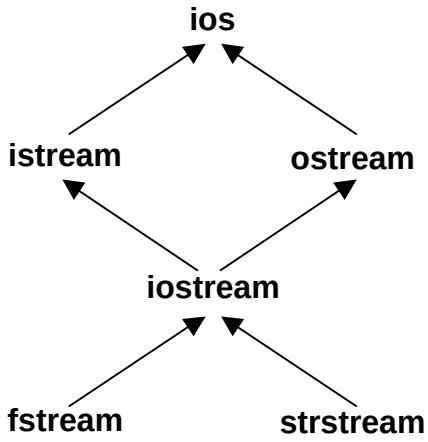


Günstigstes Versorgungsnetz

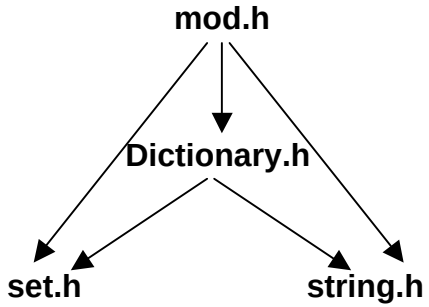
Algorithmen → minimal aufspannende Bäume

Zyklenfreiheit bei gerichteten Graphen (1)

- **Beispiel: Ausschnitt aus Vererbungsgraph für stream-Klassen**

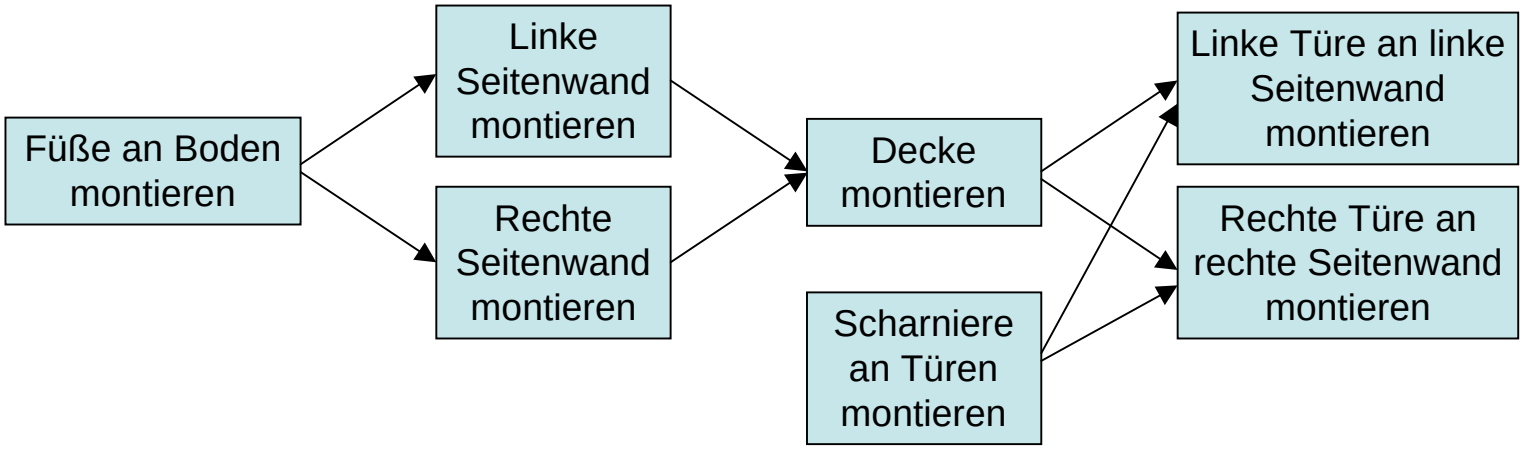


- **Beispiel: include-Graph**



Zyklenfreiheit bei gerichteten Graphen (2)

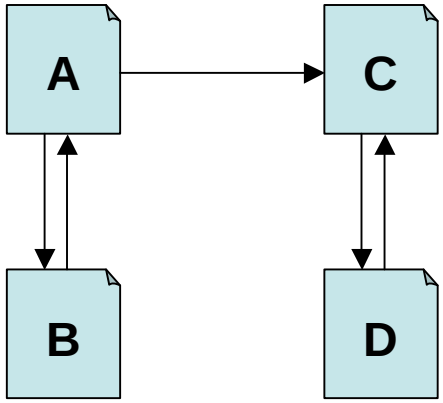
- **Beispiel: Montageanleitung für Schrank**



Algorithmen → topologische Sortierung

Internet-Suchmaschinen (1)

Web-Seiten mit ihren Verweisen



Problemstellung: Indexierung

Besuche alle Internetseiten, extrahiere Schlüsselwörter und speichere diese in einen Index.

Algorithmen → Tiefensuche

Internet-Suchmaschinen (2)

Problemstellung: Relevanz von Internetseiten

Bei Eingabe eines Suchbegriffs durchsuchen die Suchmaschinen ihren Index.
Die gefundenen Internetseiten werden üblicherweise nach Relevanz sortiert ausgegeben.

Problem: Stelle die Relevanz von Internetseiten fest.

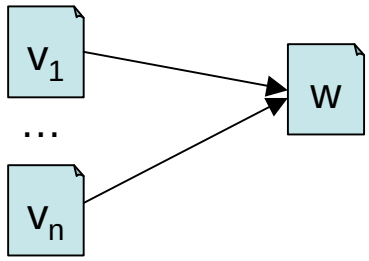
PageRank-Algorithmus von Page und Prin (s. [Turau])

1) Berechne für jede Internetseite w den PageRank $r(w)$ nach folgender Formel:

$$r(w) = (1-\alpha) + \alpha * \sum_{\substack{\text{Seite } v, \text{ wobei } v \\ \text{ein Link auf } w \text{ hat}}} \frac{r(v)}{l(v)}$$

$l(v)$ = Anzahl der ausgehenden Links von v

$\alpha \in [0,1]$ ist Dämpfungsfaktor

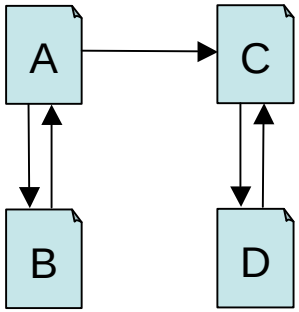


2) Wiederhole Schritt 1) solange, bis $r(w)$ für alle Seiten hinreichend konvergiert.

Internet-Suchmaschinen (3)

Beispiel:

Für das Beispiel-Netz ergeben sich
mit α = 0.5 folgende Gleichungen:



r(A) = 0.5 + 0.5*r(B)

r(B) = 0.5 + 0.5*r(A)/2

r(C) = 0.5 + 0-5*(r(A)/2 + r(D))

r(D) = 0.5 + 0.5*r(C)

Beginnt man mit den Initialisierungswerten r(w) = 1 und
wendet man die Gleichungen iterativ an,
ergeben sich bereits nach 4 Iterationen stabile Werte:

Iterationen	r(A)	r(B)	r(C)	r(D)
1	1	1	1	1
2	1	0.75	1.125	1.0625
3	0.875	0.71875	1.071875	1.0359375
4	0.859375	0.71484375	1.072265625	1.0361328125

Teil 2:
Graphenalgorithmen

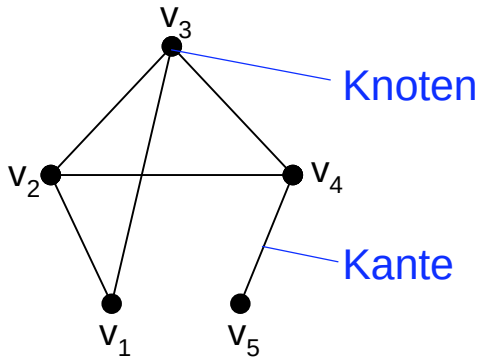
- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Definitionen (1)

Graph

Ein Graph G = (V, E) besteht aus einer Menge V von Knoten (engl. vertices) und einer Menge E von Kanten (engl. edges).

Eine Kante ist gegeben durch ein Paar von Knoten (v, w).



$V = \{v_1, v_2, v_3, v_4, v_5\}$

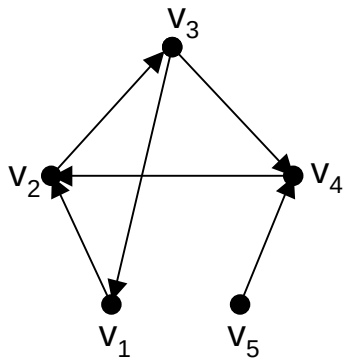
$E = \{(v_1,v_2), (v_1,v_3), (v_2,v_3), (v_2,v_4), (v_3,v_4)\}$

Definitionen (2)

Gerichteter Graph

Bei einem gerichteten Graphen (auch Digraph genannt) sind die Kanten gerichtet und werden durch ein geordnetes Paar von Knoten (v, w) bestimmt.

In den graphischen Darstellungen werden Pfeile verwendet.



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_1,v_2), (v_2,v_3), (v_3,v_1), (v_3,v_4), (v_4,v_2), (v_5,v_4)\}$

Ungerichteter Graph

Bei einem ungerichteten Graphen sind die Kanten ungerichtet. Eine Kante zwischen u und v kann als 2-elementige Menge {v, w} dargestellt werden.

Der Einfachheit wegen wollen wir jedoch auch ungerichtete Kanten als Paar (v, w) darstellen, wobei im Kontext festgelegt wird, ob Kanten gerichtet oder ungerichtet zu verstehen sind.

Definitionen (3)

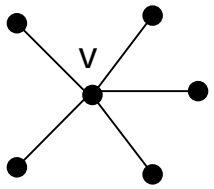
Nachbarn und Grad

Zwei Knoten v und w, die durch eine (gerichtete oder ungerichtete) Kante verbunden sind, heißen Nachbarn (v und w werden auch als adjazent bezeichnet).

Bei einem gerichteten Graphen können Nachbarn in Vorgänger und Nachfolger unterschieden werden.

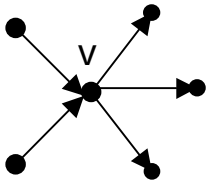
Der Grad eines Knoten ist die Anzahl seiner Nachbarn.

Bei einem gerichteten Graphen kann zwischen Eingangs- und Ausgangsgrad unterschieden werden.



Der Knoten v hat 5 Nachbarn.

Der Grad von v ist damit 5.



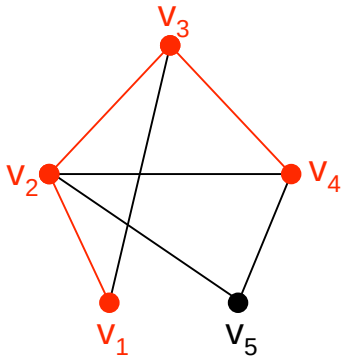
Der Knoten v hat 5 Nachbarn:
2 Vorgänger und 3 Nachfolger.

Der Eingangsgrad von v ist damit 2 und der Ausgangsgrad 3.

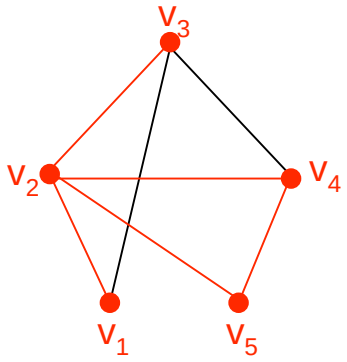
Definitionen (4)

Weg und Zyklus

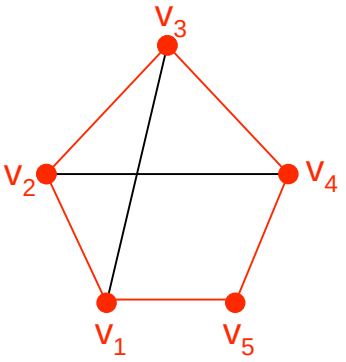
- Ein Weg (engl. path) ist eine Folge von Knoten $v_1, v_2, \ldots, v_n$, so dass $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$.
- Die Länge dieses Weges ist gleich der Anzahl der Kanten und damit n-1.
- Ein Weg heißt einfacher Weg, falls alle Knoten bis auf Anfangs- und Endknoten unterschiedlich sind.
- Ein Weg $v_1, v_2, \ldots, v_n$ heißt geschlossen oder auch Zyklus (engl. cycle), falls Anfangsknoten v_1 und Endknoten v_n identisch sind. Zyklenfreie Graphen werden auch azyklisch genannt.



Ein (einfacher) Weg der Länge 3: v_1, v_2, v_3, v_4.



Ein (nicht einfacher) Weg der Länge 5: $v_1, v_2, v_5, v_4, v_2, v_3$



Ein Zyklus und auch einfacher Weg der Länge 5: $v_1, v_2, v_3, v_4, v_5, v_1$

Definitionen (5)

Zusammenhang

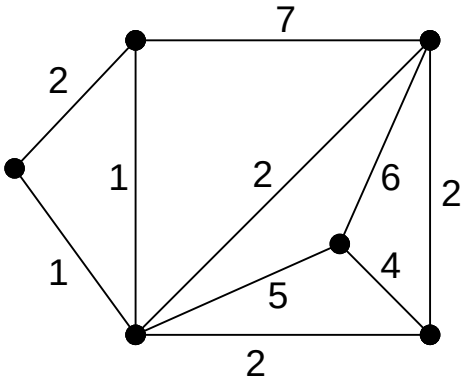
- Ein ungerichteter Graph heißt zusammenhängend (engl. connected), falls es von jedem Knoten einen Weg zu jeden anderen Knoten gibt.
- Ein gerichteter Graph mit dieser Eigenschaft heißt streng zusammenhängend (engl. strongly connected).
- Wenn ein gerichteter Graph nicht streng zusammenhängend ist, aber der entsprechende ungerichtete Graph zusammenhängend ist, dann wird G auch schwach zusammenhängend genannt. (engl. weakly connected).

Gewichteter Graph

Ist jeder Kante (v,w) eine reelle Zahl c(v,w) als Kosten oder Gewicht zugeordnet, spricht man von einem gewichteten Graphen.

Sind darüber hinaus die Gewichte ≥ 0, dann heißt der Graph auch Distanzgraph.

Beispiel:



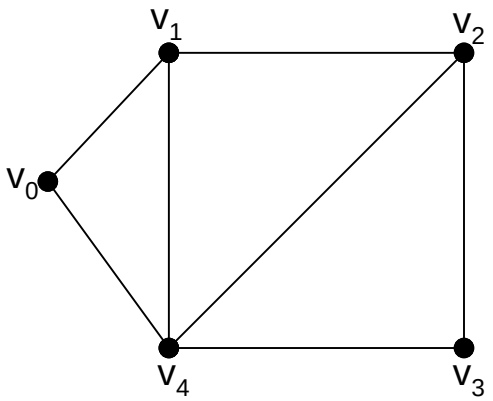
Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
  - Adjazenzmatrix
  - Adjazenzliste
  - Kantenliste
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Adjazenzmatrix

Ungewichteter Graph

$$A[i][j] = \begin{cases} 1, \text{ falls } (v_i,v_j) \in E \\ 0 \text{ sonst} \end{cases}$$

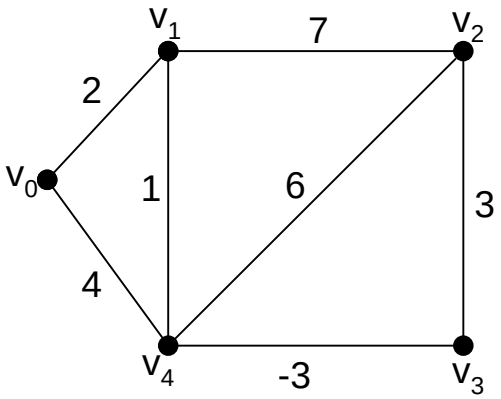


	[0]	[1]	[2]	[3]	[4]
[0]		1			1
[1]	1		1		1
[2]		1		1	1
[3]			1		1
[4]	1	1	1	1	

Einträge = 0 wurden weggelassen.

Gewichteter Graph

$$A[i][j] = \begin{cases} c(v_i, v_j), \text{ falls } (v_i,v_j) \in E \\ \infty \text{ sonst} \end{cases}$$



	[0]	[1]	[2]	[3]	[4]
[0]		2			4
[1]	2		7		1
[2]		7		3	6
[3]			3		-3
[4]	4	1	6	-3	

Einträge = ∞ wurden weggelassen.

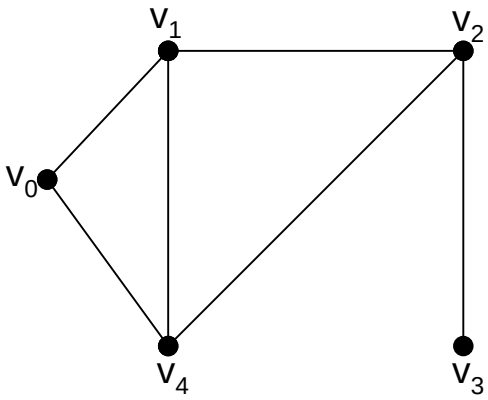
Beachte: Die Adjazenzmatrizen sind bei einem ungerichteten Graphen symmetrisch und bei einem gerichteten Graphen im allgemeinen unsymmetrisch.

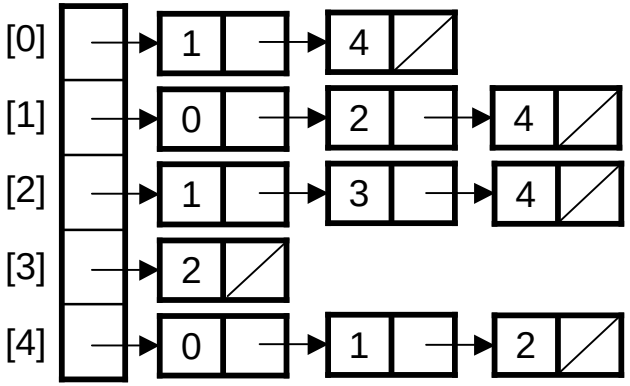
Adjazenzliste

In einer Adjazenzliste wird für jeden Knoten eine linear verkettete Liste mit allen Nachbarknoten abgespeichert.

Bei einem gerichteten Graphen kann die Adjazenzliste in Vorgänger- und Nachfolger-Liste aufgeteilt werden.

Bei einem gewichteten Graph wird noch zusätzlich das Kantengewicht eingetragen





Beachte:

- Bei Graphen mit einem geringen Verhältnis von Anzahl Kanten zu Knoten sind Adjazenzmatrizen dünn besetzt. Hier kann die Speicherung als Adjazenzliste geeigneter sein.
- Um die Dynamisierbarkeit der Datenstruktur zu verbessern, kann statt dem Zeiger-Feld auch eine linear verkettete Liste verwendet werden.

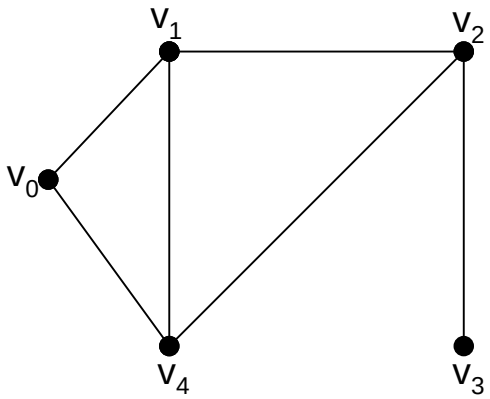
Kantenliste

Speichere Menge aller Kanten in eine Suchstruktur (s. Teil I der Vorlesung).

Als Ordnungsrelation auf Kanten ist eine lexikographische Ordnung naheliegend:

$(v, v') \leq (w, w')$ falls $v < w$ oder
$v = w$ und $v' \leq w'$

Beispiel: Sortiertes Kanten-Feld



	[0]	[1]	[2]	…							[11]
i	0	0	1	1	1	2	2	2	3	4	4	4
j	1	4	0	2	4	1	3	4	2	0	1	2

Alle Nachbarknoten zu Knoten v_2.

Beachte:

- Bei einem ungerichten Graphen wird für eine Kante von v nach w sowohl (v,w) als auch (w,v) gespeichert.
- Bei einem gerichteten Graphen kann die Kantenliste (wie bei der Adjazenzliste) in Vorgänger- und Nachfolger-Liste aufgeteilt werden.

Darstellung der Algorithmen

Um von der konkreten Datenstruktur für Graphen unabhängig zu bleiben, wird bei der Formulierung der Algorithmen folgender Pseudo-Code benutzt:

Ungerichtete und gerichtete Graphen:

```
for ( jeden Nachbarn w von v )
{
      ...
}
```

```
for ( jeden Knoten v )
{
      ...
}
```

```
for ( jede Kante (v,w) )
{
      ...
}
```

Gerichtete Graphen:

```
for ( jeden Nachfolger w von v )
{
      ...
}
```

```
for ( jeden Vorgänger w von v )
{
      ...
}
```

Beachte:

Die hier in Pseudo-Code dargestellten Schleifen lassen sich mit den besprochenen Datenstrukturen (Adjazenzmatrix, Adjazenzliste und Kantenliste) einfach realisieren.

Aufgabe

Aufgabe 2.1

Wählen Sie eine geeignete Datenstruktur, um für eine größere Menge von Städten (als Strings gegeben) eine Entfernungstabelle abzuspeichern.

Gehen Sie dabei von zwei unterschiedlichen Voraussetzungen aus:

a) Die Menge der Städte ist statisch: d.h. Anzahl und Namen ändern sich nicht.

b) Die Menge der Städte ist dynamisch.

Teil 2:
Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
  - Tiefensuche
  - Breitensuche
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Tiefensuche (1)

Rekursive Tiefensuche

Tiefensuche lässt sich für gerichtete und ungerichtete Graphen rekursiv formulieren:

```
int val[n];


void visit(Vertex v)
{
      val[v] = “besucht”;

      // Bearbeite v:
      cout << v << endl;

      for (jeden Nachbar w von v )
            if (val[w] == “nichtBesucht”)
                  visit(w);
}


for (jeden Knoten v)
      val[v] = “nichtBesucht”;
visit(s);
```

Mit val-Feld wird global gespeichert, ob ein Knoten bereits besucht worden ist.

Wichtig zur Vermeidung von Endlosschleifen.

Das val-Feld muss vor der Tiefensuche initialisiert werden.

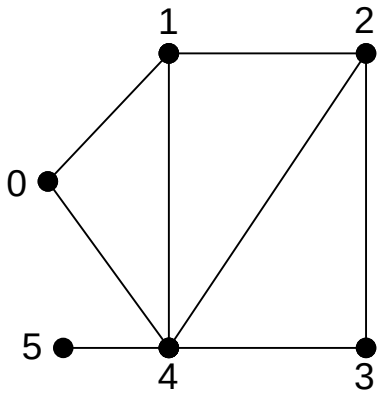
Ausschnitt aus aufrufendem Programm.

Beginne mit Startknoten s.

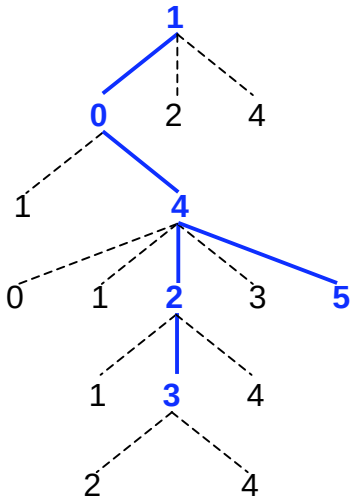
Tiefensuche (2)

Beispiel

Tiefensuche mit Start bei Knoten 1



Aufrufstruktur



Besuchsreihenfolge:
1, 0, 4, 2, 3, 5

die Nachbarn eines Knoten werden in lexikographischer Reihenfolge durchlaufen.

Bereits besuchte Nachbarknoten sind durch eine gestrichelte Kante verbunden und gehören nicht zur Aufrufstruktur.

Tiefensuche (3)

Iterative Tiefensuche (d.h. ohne Rekursion) mit einem Keller

```
void visit(Vertex v)
{
      Stack stk<Vertex>;
      stk.push(v);

      while( ! stk.empty() )
      {
            v = stk.top(); stk.pop();
            if (val[v] == “besucht”)
                  continue;

            val[v] = “besucht”;

            // Bearbeite v:
            cout << v << endl;

            for (jeden Nachbar w von v )
                  if ((val[w] == “nichtBesucht”)
                        stk.push(w);
      }
}

…
```

Im Keller stk werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Tiefensuche wird erreicht durch die LIFO-Organisation des Kellers.

Um die gleiche Besuchsreihenfolge wie bei der rekursiven Funktion zu erreichen, müssen in der for-Schleife die Nachbarn in umgekehrter Reihenfolge bearbeitet werden.

Beachte: Gleiche Knoten können eventuell mehrfach eingekellert werden.

val-Feld wird wie bei rekursiver Variante vor visit-Aufruf mit „nichtBesucht“ initialisiert.

Tiefensuche (4)

Bemerkung

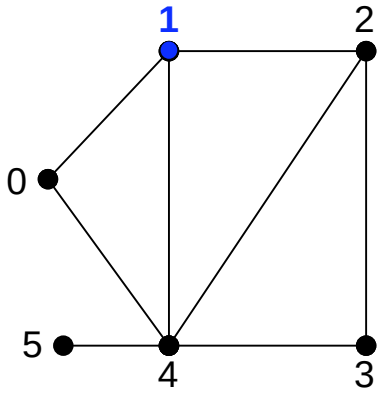
Mit dem Aufruf visit(s) lassen sich nur die vom Knoten s aus erreichbaren Knoten besuchen. (Zusammenhangskomponente).

Besteht der Graph aus mehreren Zusammenhangskomponenten und sollen alle Knoten besucht werden, dann muss visit mit allen noch nicht besuchten Knoten als Startknoten aufgerufen werden:

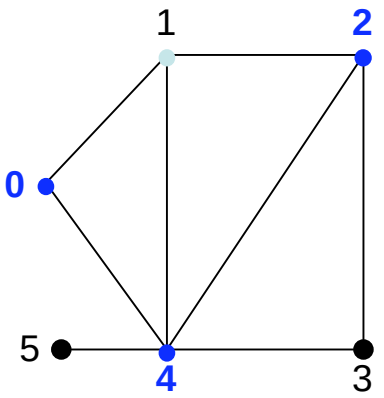
```
bool visited[n];

void visit(Vertex v)
{
        …
}

int main()
{
        for (jeden Knoten v)
                val[v] = “nichtBesucht”;

        for (jeden Knoten v)
                if (val[v] == “nichtBesucht”)
                        visit(v);
}
```

Breitensuche (1)

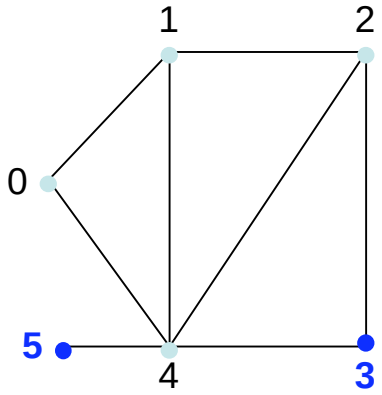
Beispiel



1) Beginne mit Knoten 1.



2) Besuche alle Knoten, die über genau eine Kante von Knoten 1 erreichbar sind.



3) Besuche alle Knoten, die über genau 2 Kanten von Knoten 1 erreichbar sind.

Besuchsreihenfolge:

1, 0, 2, 4, 3, 5

Breitensuche (2)

Breitensuche mit einer Schlange

(ähnlich wie Tiefensuche; jedoch mit Schlange statt Keller)

```
void visit(Vertex v)
{
      Queue q<Vertex>;
      q.push(v);

      while ( ! q.empty() )
      {
            v = q.top(); q.pop();

            visit[v] = “besucht”;

            // Bearbeite v:
            cout << v << endl;

            for (jeden Nachbar w von v )
                  if ((val[w] == “nichtBesucht”) {
                        val[w] = “registriert”;
                        q.push(w);
                  }
      }
}

…
```

In der Schlange q werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Breitensuche wird erreicht durch die FIFO-Organisation der Schlange

Um zu verhindern, dass gleiche Knoten mehrfach in der Schlange vorkommen, werden sie mit „registriert“ gekennzeichnet, sobald sie in die Schlange eingefügt werden.

val-Feld wird wie bei rekursiver Variante vor visit-Aufruf mit „nichtBesucht“ initialisiert.