
Graphen

In diesem Kapitel behandeln wir Algorithmen auf Graphen.

Inhaltsverzeichnis

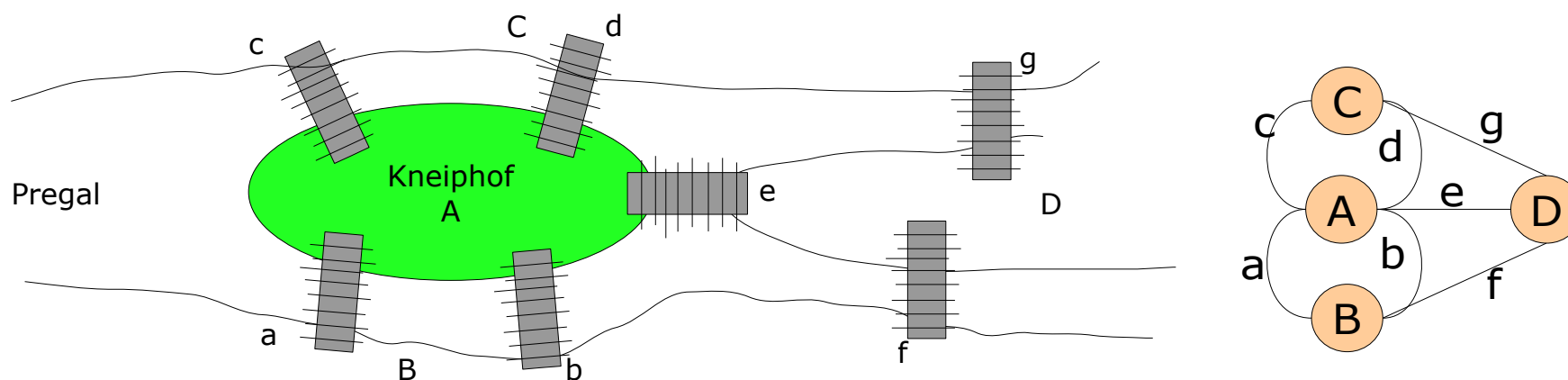
1. Grundlagen.....	2
1. Repräsentation von Graphen	8
2. Der ADT Graph.....	12
1. Durchlaufen von Graphen.....	16
1. Zyklentest und topologisches Sortieren.....	23
1. Transitive Hülle.....	29
2. Kürzeste Wege.....	33

1. Grundlagen

Graphen sind ein elegantes Mittel, zur Modellierung von „real-world“ Situationen, in denen komplexe Strukturen ausgedrückt werden müssen. Komplexe Struktur meint dabei, dass es zwischen Objekten Beziehungen gibt, die über Hierarchien, wie sie durch Bäume abbildbar sind, hinausgehen.

Das erste belegbare Graphenproblem geht auf *Euler* zurück, das **Königsberger Brückenproblem**.

In Königsberg umfließt der **Fluss Pregal** die **Insel Kneiphof** und verzweigt dann in zwei Arme. Dadurch sind 4 Landschaften gegeben die durch Brücken wie folgt verbunden sind.



Das Königsberger Brückenproblem ist:

Ist es möglich ist, von einer beliebigen Landschaft (A-D) zu starten, jede Brücke (a-g) genau einmal zu überqueren und zum Ausgangspunkt zurück zu kehren?

Versuchen Sie es !

Euler bewies, dass dies für keinen Königsberger (auch nicht für andere Menschen) möglich ist.

Er stellte die Landschaften als Knoten und die Brücken als Verbindungen zwischen Knoten dar und bewies, dass es einen (nach ihm benannten) „Euler-Pfad“ genau dann gibt, wenn der Grad jeden Knotens (Anzahl ein- und ausgehenden Kanten) gerade ist.

Um Graphen exakt definieren zu können, braucht man den **Begriff** der binären Relation.

Definition (kartesisches Produkt, Relation)

Das *kartesische Produkt* $A \times B$ zweier Mengen A und B umfasst alle geordneten Paare (a, b) mit $a \in A$ und $b \in B$.

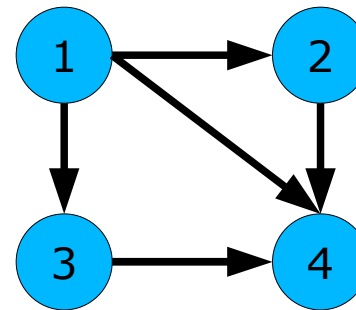
Eine Teilmenge $R \subset A \times B$ des kartesischen Produktes der Mengen A und B heißt *binäre Relation*.

Definition (gerichteter Graph, Digraph)

Ein gerichteter *Graph* $G=(V, E)$ ist die Zusammensetzung einer Menge V von Knoten und einer Relation $E \subset V \times V$ von Kanten.

Die Knoten (engl. vertex, node) eines Graphen werden als Kreise dargestellt, die Kanten (engl. edge, arc) als gerichtete Pfeile zwischen den Knoten.

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



Wenn $(v, w) \in E$ dann nennen wir das eine Kante von v nach w .

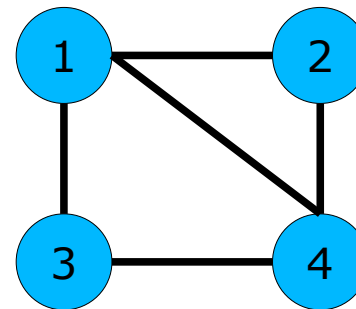
Man kann Graphen wie folgt **klassifizieren**. Ein Graph $G=(V, E)$ heißt

- **symmetrisch**, wenn mit der Kante (v,w) immer auch die Kante (w,v) zu E gehört
- **asymmetrisch**, wenn er nicht symmetrisch ist
- **reflexiv**, wenn jeder Knoten in G mit sich selbst verbunden ist
- **irreflexiv**, wenn er nicht reflexiv ist
- **transitiv**, wenn gilt $\forall (v, w), (w, z) \in E \Rightarrow (v, z) \in E$

Ein symmetrischer Graph wird auch als **ungerichteter** Graph bezeichnet und in der Darstellung werden bei den Kanten die Pfeilspitzen weggelassen.

$$V = \{1, 2, 3, 4\}$$

$$E = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3)\}$$



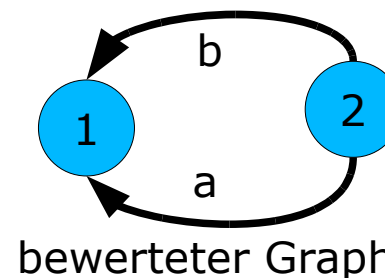
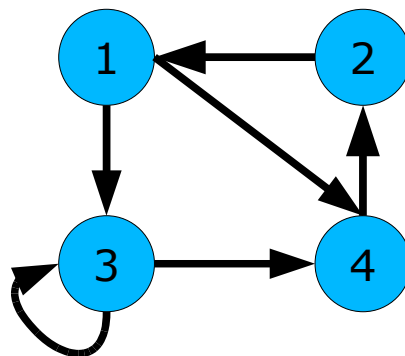
Die folgenden **Begriffe** werden immer wieder verwendet, wenn Algorithmen auf Graphen diskutiert werden (sei $G=(V, E)$ ein Graph und $(v, w) \in E$):

- v ist **Vorgänger** von w
- v und w sind **benachbart** (adjazent)

- (v,w) ist eine in den Knoten w **einlaufende Kante** und eine aus dem Knoten v **auslaufende Kante**
- ist $v=w$, dann heißt die Kante **Schlinge**
- der **Eingangsgrad** (engl. indegree) eines Knotens ist die definiert als die Anzahl der in den Knoten einlaufenden Kanten
- er **Ausgangsgrad** (engl. outdegree) eines Knotens ist die definiert als die Anzahl der aus dem Knoten auslaufenden Kanten
- der **Grad** eines Knotens ist definiert als die Summe von Eingangs- und Ausgangsgrad
- eine Folge von Knoten v_1, v_2, \dots, v_n heißt **Weg** (engl. path) von v_1 nach v_n wenn gilt:

$$\forall 1 \leq i < n : (v_i, v_{i+1}) \in E$$
- Ein **Zyklus** ist eine Weg, bei dem zumindest ein Knoten mehrfach vorkommt.
- Wenn die Kanten durch Attribute gekennzeichnet sind so nennt man den Graph einen **bewerteten** oder **gewichteten** Graph.

- 1 ist Vorgänger von 3
- $\text{outdegree}(1)=2$
- $\text{indegree}(4)=2$
- 3,4,2 ist Weg von 3 nach 2
- 1,4,2,1 ist Zyklus

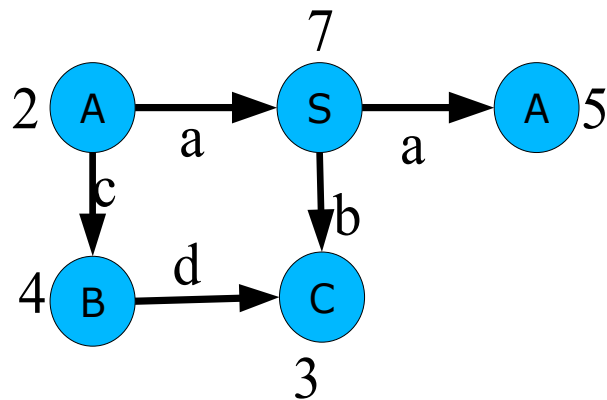


Definition (knoten- und kantenmarkierter Digraph)

Ein *knoten- und kantenmarkierter Digraph* über dem Knotenmarkierungsalphabet Σ_v und dem Kantenmarkierungsalphabet Σ_k ist definiert als Tripel $G=(V, E, L)$ mit:

1. $V \subset \mathbb{N}$ ist die Knotenmenge
2. $E \subset V \times \Sigma_k \times V$ ist die endliche Menge von Kanten.
(v, x, w) in E heisst x -Kante von v nach w .
3. $L: V \rightarrow \Sigma_v$ ist die Knotenmarkierungsfunktion.

Wenn $L(v)=A$, dann nennen wir v einen A -Knoten.



1.Repräsentation von Graphen

Zur Repräsentation von Graphen in Programmen gibt es im Wesentlichen zwei Möglichkeiten:

- Adjazenzmatrizen (Nachbarschaftsmatrizen) und
- Adjazenzlisten (Nachbarschaftslisten).

Die „richtige“ Wahl hängt von der Aufgabenstellung und davon ab, ob der Graph eher dichte oder dünne Kantenbelegung hat.

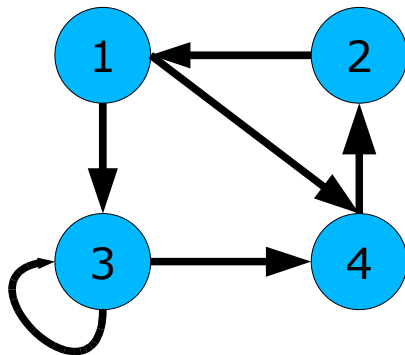
Adjazenzmatrix

Eine **Adjazenzmatrix** $A=(a_{i,j})$ eines Graphen $G=(V,E)$ mit $V=\{v_1, v_2, \dots, v_n\}$ ist eine (n,n) -Matrix mit den Elementen:

$$a_{i,j}=1, \text{ falls } (v_i, v_j) \in E$$

$$a_{i,j}=0, \text{ falls } (v_i, v_j) \notin E$$

Beispiel:



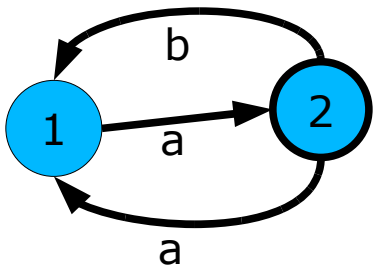
	1	2	3	4
1	0	0	1	1
2	1	0	0	0
3	0	0	1	1
4	0	1	0	0

Der **Test**, ob eine **Kante** in einem Graphen vorhanden ist, kann durch einfachen Zugriff auf das entsprechende Matrizen-Element geschehen, also in **$O(1)$** .

Viele Algorithmen auf gerichteten Graphen müssen aber auf **alle Kanten** zugreifen, dies ist teuer, es bedarf **$O(n^2)$** , wenn $n=|V|$.

Der **Speicherbedarf** ist unabhängig von der Kantenanzahl, man benötigt **$O(n^2)$** Speicherzellen. Also ist diese Form zu wählen, wenn die **Kantenbelegung relativ dicht** ist !!

Sollen bewertete Graphen gespeichert werden, kann man anstelle der booleschen Größe, die Kantenbewertung oder einen besonderen Wert für „keine Kante“ eintragen. Zu beachten ist, dass es mehrere unterschiedlich bewertete Kanten mit gleichen Start- und Endknoten geben kann:

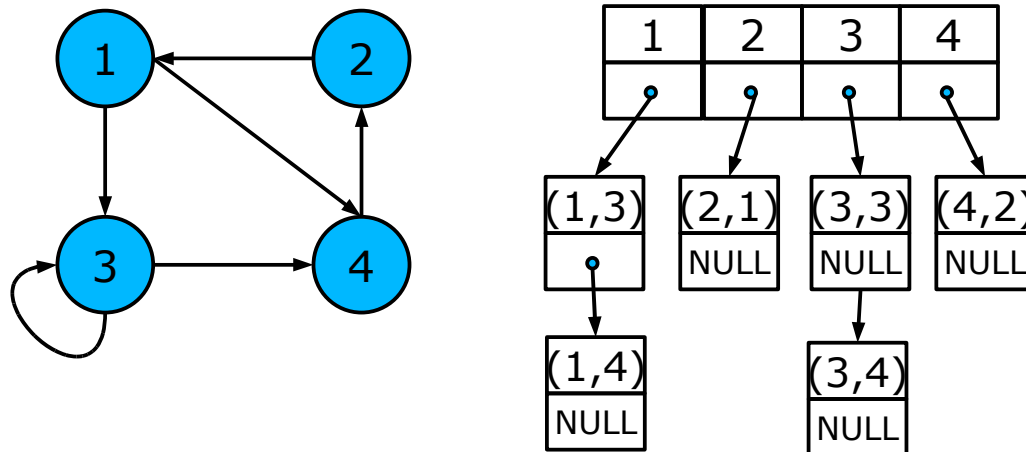


	1	2
1	-	a
2	b,a	-

Wie würden Sie *knoten- und kantenmarkierter Digraphen* mittels Adjazenzmatrizen in C++ umsetzen?

Adjazenzliste

Ein **Array** wird zur Speicherung der **Knoten** verwendet. Eine **Adjazenzliste** stellt die **Kanten** dar. Dabei gibt es für jeden Knoten v vom Array einen Zeiger auf die korrespondierende Adjazenzliste. Ein Element der Adjazenzliste für den Knoten v beinhaltet die Kante (v,w) und einen Zeiger auf die nächste Kante, die von v ausgeht.



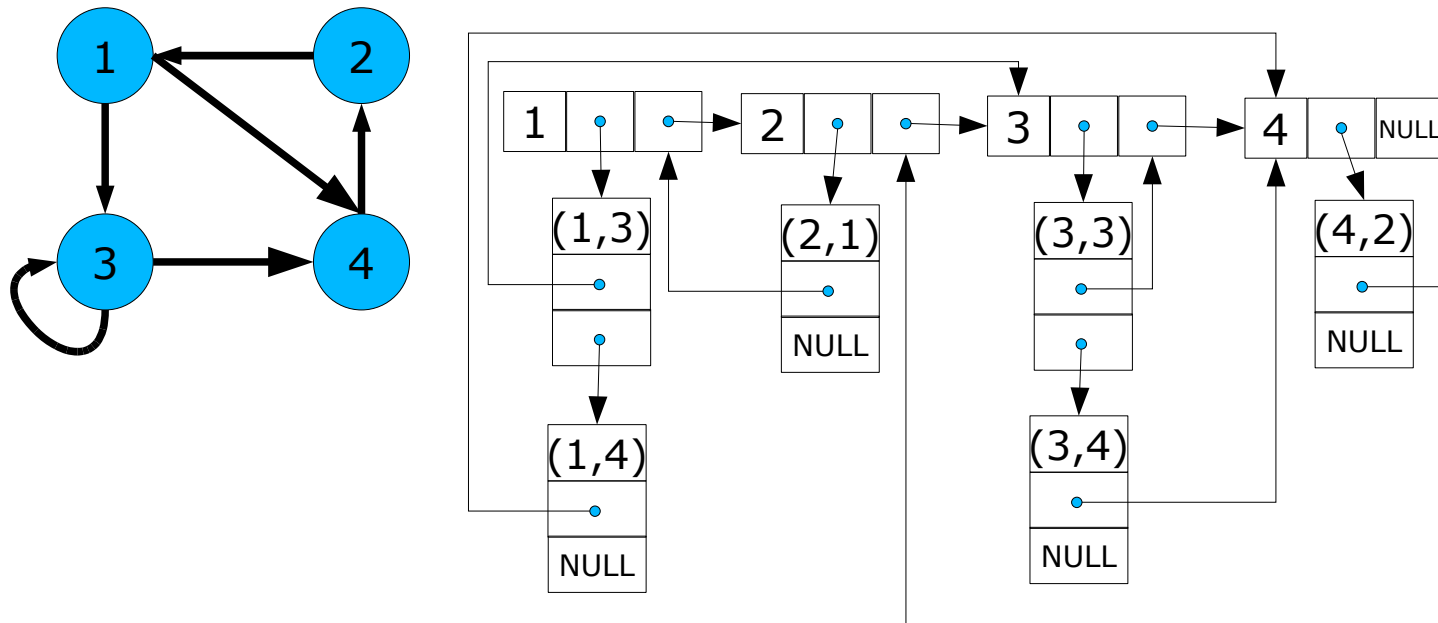
Der **Speicheraufwand** hierbei ist **proportional** zur **Summe** aus den **Knotenanzahl** n und der Zahl der **Kanten** e . Diese Darstellung ist dann von **Vorteil**, wenn die Kantenanzahl sehr viel kleiner als n^2 ist (**dünn besetzte Graphen**). Man benötigt aber $O(n)$ Schritte, um festzustellen, ob eine Kante vorhanden ist.

Zusätzlich wird die Kantenmarkierung in jedem Element der Adjazenzliste abgelegt.

Der **Nachteil** der o.a. Darstellung ist es, dass das **Array feste Grösse** hat.

Deshalb kann man anstelle des Arrays für die **Knoten eine verkettete Liste** verwenden.

Zusätzlich ist es von Vorteil (Durchlaufen des Graphen), wenn man von einer Kante der Adjazenzliste zum Knotenelement des Zielknotens verweist:



Sollten im Anwendungsumfeld häufig Knoten oder Kanten gelöscht werden, dann empfiehlt es sich, die Verkettung doppelt auszulegen.

2. Der ADT Graph

Um den ADT zu spezifizieren, muss man die von den Graph-Algorithmen erforderlichen Operationen kennen, einige sind:

- Knoten einfügen und löschen
- Kanten hinzufügen und löschen
- Verfolgen von Wegen
- Vorgänger und Nachfolger eines Knotens
- in- und outdegree eines Knotens
- Markierung eines Knotens als „schon mal besucht“

Wir realisieren einen gerichteten kantenbewerteten Graphen (directed graph, **digraph**) als **Adjazenzmatrix**, bei dem in einem Knoten (`Vertex`) neben der Knotenmarkierung (`value`) Verwaltungsinformation, wie z.B. Ein- und Ausgangsgrad abgelegt ist. Um den ADT recht allgemein verwenden zu können, ist die Klasse `Vertex` als Member Template realisiert.

```
$ cat Digraph.h
//           Klasse "Digraph" Deklarationen
```

```
template <class TV, int maxNodes>
class Digraph {
private:
    template <class T>
    class Vertex {
    public:
        Vertex() { // Konstruktor
            indegree = outdegree = ord = 0;
            visited = living = false;
            value = 0;
        }
        T value; // Markierung
        int indegree; // Eingangsgrad
        int outdegree; // Ausgangsgrad
        int ord; // Ordnungszahl des Knotens
        bool visited; // Knoten-"besucht"-Marke
        bool living; // Knoten existiert
    };

    int numvertices; // Anzahl der Knoten eines Digraphen
    int numarcs; // Anzahl der Kanten eines Digraphen
    Vertex<TV> vertex[maxNodes]; // repräsentiert Knoten des Digraphen
    double arc[maxNodes][maxNodes]; // repräsentiert bewertete Kanten des
    // Digraphen (-1=keine Kante)
```

```

// Operationen auf einem Digraphen
public:
    Digraph();           // Digraphen generieren
    ~Digraph();         // Digraphen freigeben
    void InsertVertex (int n, TV m); // fügt neuen Knoten n mit
                                     // Markierung m ein
    void DeleteVertex(int n); // entfernt Knoten n
    TV GetVertexValue(int n); // gibt Markierung des Knoten n zurück
    void SetVertexValue(int n, TV m); // Ändert Markierung des Knoten n
    int FirstVertex(); // Erster Knoten des Graphen
    int NextVertex(int n); // nächster Knoten nach n
                                     // (-1 wenn keiner existiert)
    void InsertArc(int v, int w, double weight); // fügt Kante (v,w) mit
                                     // Gewicht weight ein
    void InsertArc(int v, int w); // fügt neue Kante (v,w) mit
                                     // Gewicht 0 ein
    void DeleteArc(int v, int w); // löscht Kante (v,w)
    bool IsArc(int v, int w); // existiert Kante (v,w)?
    double GetArc(int v, int w); // gibt Kantenbewertung von (v,w) zurück
    int FirstArc(int v); // Erste Kante des Knoten v des Graphen
    int NextArc(int v, int n); // nächste Kante des Knotens v nach n
                                     // (-1 wenn keiner existiert)
    int GetNumVertices(); // Anzahl der Knoten
    int GetMaxNodes(); // Max. Anzahl der Knoten
    int GetNumArcs(); // Anzahl der Kanten
    int GetIndegree(int n); // liefert Eingangsgrad eines Knotens
    int GetOutdegree(int n); // liefert Ausgangsgrad eines Knotens

```

```

    void SetVisited(int n ,bool b); // Knoten n als besucht oder
                                   //nicht besucht markieren
    bool IsVisited(int n);         // true, wenn Knoten n besucht
    void SetOrd(int v,int n);     // Ordnungsnummer n im Knoten v setzen
    int GetOrd(int v);           // Ordnungsnummer des Knotens v liefern
    void PrintVertex(int n);      // Ausgabe des Knotens n eines Digraphen
    void PrintVerteces();        // Ausgabe aller Knotens eines Digraphen
    void PrintAdjMatrix();       // Adjazenzmatrix ausgeben
    void DeleteDigraph();        // alle Knoten und Kanten löschen
};
$

```

Diese Spezifikation soll ausreichen, um damit Graphalgorithmen beschreiben zu können. Die **Implementierung** der Klasse ist Gegenstand des **Praktikums**.

Das folgende Beispiel demonstriert die Verwendung einiger Methoden der Klasse: eine Funktion, die alle Knoten eines Digraph als besucht bzw. unbesucht markiert.

```

// Alle Knoten als besucht bzw. unbesucht markieren
template <class TV, int maxNodes>
void SetAllVisited(Digraph<TV, maxNodes> &g, bool visit) {
    int v = g.FirstVertex();
    while (v != -1) {
        g.SetVisited(v, visit);
        v = g.NextVertex(v);
    }
}

```

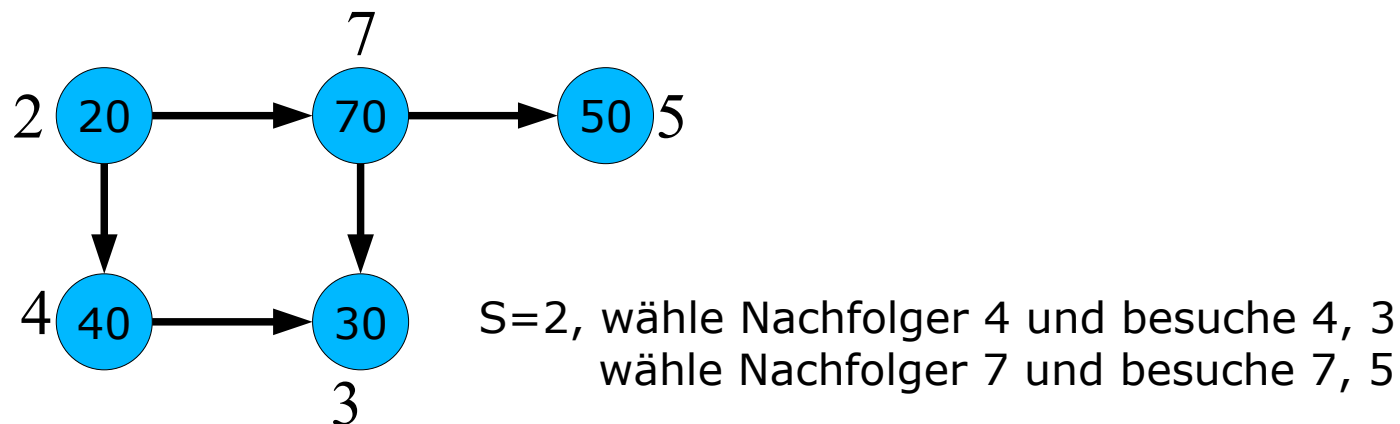
1. Durchlaufen von Graphen

In Anwendungen, bei denen Graphen als Repräsentation verwendet werden, muss oft jeder Knoten des Graphen besucht werden. Dazu kann man zwei Strategien unterscheiden:

- Tiefensuche (engl. depth first search)
- Breitensuche (engl. breadth first search)

Tiefensuche

Die Tiefensuche startet von einem Knoten s , besucht einen Nachfolger w , dann dessen Nachfolger usw. bis alle Knoten auf diesem Weg besucht sind. Dann werden alle Knoten eines anderen Nachfolgers von s in der selben Art besucht. Jetzt wird ein neuer noch unbesuchter Startknoten ausgewählt und nach der gleichen Art verfahren.



Zur Realisierung des Verfahrens ist es erforderlich, einen Knoten als **bereits besucht zu markieren**, damit **keine Unendlichschleifen** entstehen.

Das Verfahren ist realisiert durch eine Funktion `Dfs`, die einen Graph `g` ausgehend vom Startknoten `v` aus in der o.a. Reihenfolge durchläuft. Dabei wird der Knoten `v` als besucht markiert und für alle seine Nachfolger wird `Dfs` **rekursiv** aufgerufen.

Damit alle Knoten im Graph bearbeitet werden, wird `Dfs` **von einer Funktion `DepthFirstSeach` gestartet**.

```
$ cat Algorithmen.cpp
...
// Hilfsroutinen
template <class TV, int maxNodes>
void SetAllVisited(Digraph<TV, maxNodes> &g, bool visit) { // Alle Knoten als
besucht bzw unbesucht markieren
    int v = g.FirstVertex();
    while (v != -1) {
        g.SetVisited(v, visit);
        v = g.NextVertex(v);
    }
}

template <class TV, int maxNodes>
void Visit(Digraph<TV, maxNodes> &g, int n) { // Knoten n besuchen
    g.PrintVertex(n);
}
```

```
// Tiefensuche
template <class TV, int maxNodes>
void Dfs(Digraph<TV, maxNodes> &g, int v) { // Tiefensuche ab Knoten v
    if (!g.IsVisited(v)){
        g.SetVisited(v,true); // als besucht markieren
        Visit(g,v); // v besuchen
        int w = g.FirstArc(v);
        while (w != -1) { // alle Nachfolger von v bearbeiten
            if (!g.IsVisited(w)) // w noch nicht besucht
                Dfs(g,w); // Tiefensuche für w starten
            w = g.NextArc(v,w);
        }
    }
}

template <class TV, int maxNodes>
void DepthFirstSearch(Digraph<TV, maxNodes> &g) { // Tiefensuche
    SetAllVisited(g,false); // Alle Knoten als unbesucht markieren
    int v = g.FirstVertex();
    while (v != -1) { // Alle Knoten bearbeiten
        if (!g.IsVisited(v)) // v noch nicht besucht
            Dfs(g,v); // Tiefensuche für v starten
        v = g.NextVertex(v);
    }
}

...
$
```

$\text{Dfs}(g, v)$ besucht jede Kante von g , die von v aus erreichbar ist ein mal. $\text{DepthFirstSearch}(g, v)$ durchläuft die Knotenmenge ein mal und durchläuft dann (mittels Dfs) die Kanten höchstens ein mal.

Der **Aufwand**, einen Graph mit n Knoten und e Kanten zu durchlaufen ist damit **$O(n+e)$** .

Aufruf für unser o.a. Beispiel:

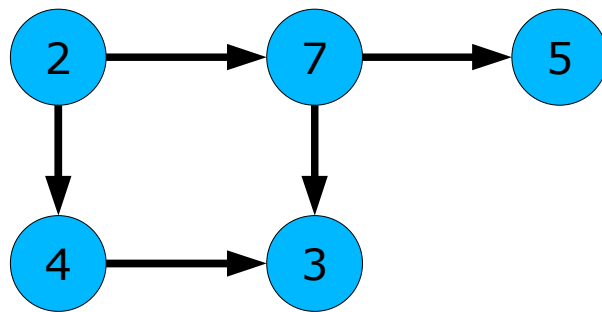
```
$ cat dfs.cpp
int main() {
    Digraph<int, 8> g;
    g.InsertVertex(2,20);
    g.InsertVertex(3,30);
    g.InsertVertex(4,40);
    g.InsertVertex(5,50);
    g.InsertVertex(7,70);
    g.PrintVerteces();
    g.InsertArc(2,4,1);
    g.InsertArc(2,7,2);
    g.InsertArc(4,3,1);
    g.InsertArc(7,3,1);
    g.InsertArc(7,5,3);
    g.PrintAdjMatrix();
    cout << "DFS: " << endl;
    DepthFirstSearch(g);
}
$
```

```
$ dfs
Knoten:
2: 20 0
3: 30 0
4: 40 0
5: 50 0
7: 70 0
Adjazenz Matrix:
      0  1  2  3  4  5  6  7
-----
0 |  -  -  -  -  -  -  -  -
1 |  -  -  -  -  -  -  -  -
2 |  -  -  -  -  1  -  -  2
3 |  -  -  -  -  -  -  -  -
4 |  -  -  -  1  -  -  -  -
5 |  -  -  -  -  -  -  -  -
6 |  -  -  -  -  -  -  -  -
7 |  -  -  -  1  -  3  -  -
DFS:
2: 20 0
4: 40 0
3: 30 0
7: 70 0
5: 50 0
$
```

Breitensuche

Bei der Breitensuche startet man von einem **Knoten s** und **besucht alle Nachfolger w** . Sind alle Nachfolge besucht, werden die Nachfolger der Nachfolger besucht usw. D.h. **man besucht zunächst die Knoten der Weglänge 1 von s besucht, dann die der Weglänge 2 usw.**

Damit Knoten nicht mehrfach besucht werden, werden sie als besucht markiert. Damit man die **Geschwister** eines gerade besuchten Knotens v findet, verwendet man eine **Schlange**, um alle **Nachfolger** von v zu merken.



$S=2$

Queue:



besuchte Knoten:

2 4 7 3 5

```

$ cat Algorithmen.cpp
...
  
```

```

// Breitensuche
template <class TV, int maxNodes>
void Bfs(Digraph<TV, maxNodes> &g, int v) { // Breitensuche ab Knoten v
    queue<int> q; // Schlange zum Merken der
                // besuchten Knoten

    if (!g.IsVisited(v)){
        g.SetVisited(v,true); // als besucht markieren
        Visit(g,v); // v besuchen
        q.push(v); // Startknoten in Schlange einfügen
        while (!q.empty()) { // Solange bis Schlange leer
            v = q.front(); // nächster besuchter Knoten aus
                          // Schlange lesen
            q.pop(); // und aus Schlange entfernen
            int w = g.FirstArc(v); // erste von v ausgehende Kante
            while (w != -1) { // für alle Nachfolger von v
                if (!g.IsVisited(w)){ // w noch nicht besucht
                    g.SetVisited(w,true); // markieren
                    Visit(g,w); // w besuchen
                    q.push(w); // w in Schlange ablegen
                }
                w = g.NextArc(v,w); // nächste Kante (v,w)
            }
        }
    }
}

```

```
template <class TV, int maxNodes>
void BreadthFirstSearch(Digraph<TV, maxNodes> &g) {           // Breitensuche
    SetAllVisited(g, false);                                // Alle Knoten als unbesucht markieren
    int v = g.FirstVertex();
    while (v != -1) {
        if (!g.IsVisited(v))                               // v noch nicht besucht
            Bfs(g, v);                                     // Breitensuche für v starten
        v = g.NextVertex(v);
    }
}
...
$
```

Der **Aufwand** für die Breitensuche ist ebenfalls **$O(n+e)$** , wenn n die Knoten- und e die Kantenanzahl ist.

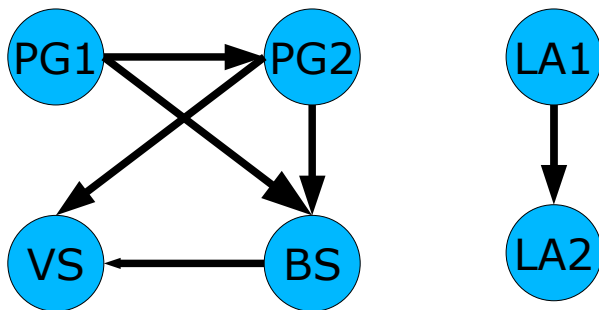
1. Zyklentest und topologisches Sortieren

In vielen Anwendungsbereichen muss man mit **partiell geordneten Mengen** umgehen (zur Erinnerung: Mengen S mit Relation $<$, die reflexiv, transitiv und antisymmetrisch ist).

Man kann solche Mengen mit ihrer Ordnungsrelation als Graph darstellen, wobei eine Kante (v,w) existiert wenn $v < w$ ist.

Beispiel (Reihenfolge, in der Veranstaltungen im Bachelor Studiengang besucht werden müssen):

- Menge der Veranstaltungen ist Knotenmenge,
- $(\text{Veranstaltung1}, \text{Veranstaltung2})$ ist eine Kante, wenn Veranstaltung1 Voraussetzung für Veranstaltung2 ist.



Aus dem o.a. Graph sieht man, dass die Relation „Voraussetzung“ keine totale Ordnung ist, da z.B. LA1 und PG2 in keinem Zusammenhang stehen.

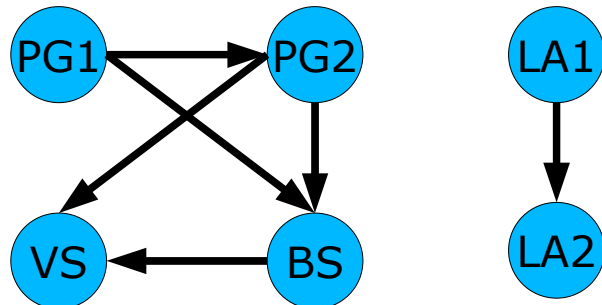
Eine **topologische Sortierung** der Knoten eines (azyklischen) gerichteten Graphen ist eine totale Ordnung der Knotenmenge, die mit der partiellen Ordnung im Graphen verträglich ist:

Folgende beiden Sortierungen sind topologische Sortierungen des o.a. Vorlesungsgraphen:

- PG1, PG2, BS, VS, LA1, LA2
- LA1, LA2, PG1, PG2, BS, VS

Folgende Sortierung ist **keine** topologische Sortierungen des o.a. Vorlesungsgraphen:

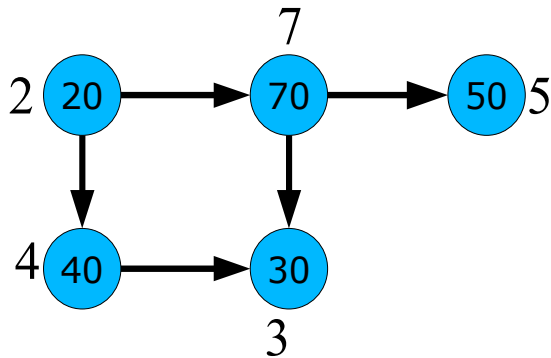
- BS, **VS**, **PG2**, **PG1**, LA1, LA2



Ein Algorithmus zur Bestimmung der topologischen Sortierung eines (azyklischen) gerichteten Graphen **startet** mit einem **Knoten**, der **keine eingehenden Kanten** hat. Ein solcher Knoten wird gewählt und er erhält die **Ordnung 1**, dann werden alle seine **ausgehenden Kanten entfernt**.

Nun wählt man erneut einen **Knoten** mit **Eingangsgrad 0**, gibt ihm die **nächst höhere Ordnungsnummer**, **entfernt** seine **ausgehenden Kanten** und wiederholt diesen Schritt, bis alle Knoten eine Ordnungszahl haben.

Wie sieht eine topologische Sortierung des folgenden Graphen aus?



Um den **Graph nicht zu verändern**, wird der **Eingangsgrad** jeden Knotens in einen **Hilfsarray** gespeichert.

```
$ cat Algorithmen.cpp
...
// Topologisches Sortieren
template <class TV, int maxNodes>
void TopSort(Digraph<TV, maxNodes> &g) { // Topologisches Sortieren;
                                        // g muss azyklisch sein
    queue<int> q;                        // Schlange für Auswahlkandidaten
    int *indegree = new int[g.GetMaxNodes()]; // Array mit Eingangsgraden
    int sortIndex = 0;                   // Sortierindex
    int v = g.FirstVertex();
    while (v != -1) {
        indegree[v] = g.GetIndegree(v); // Eingangsgrade initialisieren
        if (indegree[v] == 0)
            q.push(v); // Knoten mit idegree=0 in Schlange aufnehmen
        v = g.NextVertex(v);
    }
}
```

```

while (!q.empty()) { // Solange bis Schlange leer
    v = q.front(); // nächster Knoten aus Schlange lesen
    q.pop(); // und aus Schlange entfernen
    g.SetOrd(v,sortIndex); // Sortierindex vom Knoten v setzen
    sortIndex++; // Sortierindex inkrementieren
    int w = g.FirstArc(v); // erste von v ausgehende Kante (v,w)
    while (w != -1) { // für alle Nachfolger von v
        indegree[w]--; // Eingangsgrad von w dekrementieren
        if (indegree[w] == 0)
            q.push(w); // Knoten mit idegree=0 in Schlange aufnehmen
        w = g.NextArc(v,w); // nächste Kante (v,w)
    }
}
}

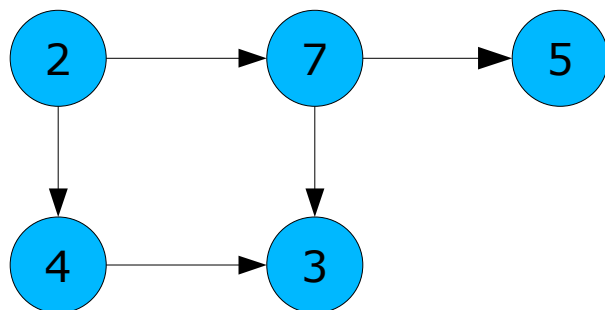
```

```

}
...
$

```

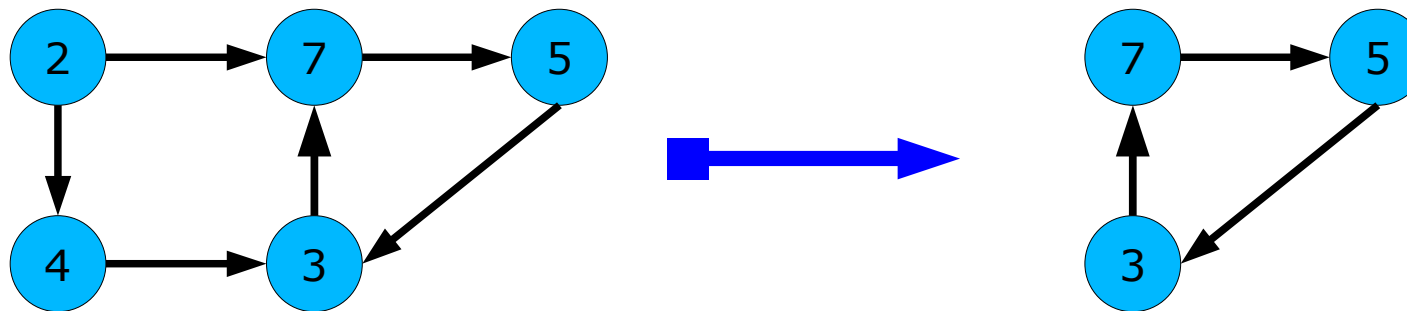
Wie sieht eine topologische Sortierung des Beispielgraphen aus:



Der **Aufwand** für das topologische Sortieren ist ebenfalls $O(n+e)$, wie man leicht aus dem o.a. Verfahren erkennen kann.

Die Analyse des o.a. Programms ergibt, dass man mit wenigen Erweiterungen **prüfen** kann, ob ein **Graph Zykel hat** oder nicht:

Wenn die **Schlange leer** ist und es **noch Knoten** gibt, die **einen von 0 verschiedenen Eingangsgrad** haben, so muss der Graph Zykel haben.



Hörsaalübung:

Erweitern Sie das Verfahren des topologischen Sortierens so, dass eine Funktion entsteht, die einen Graph auf Zyklfreiheit testet:

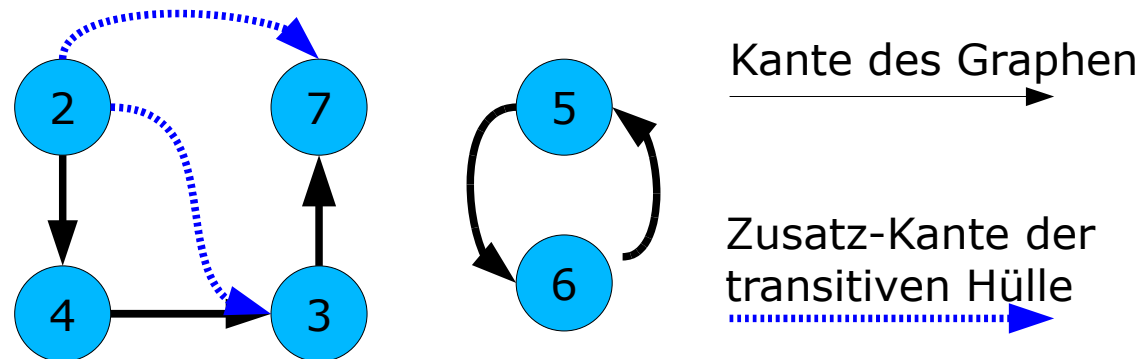
```
template <class TV, int maxNodes>
bool IsZyclic(Digraph<TV, maxNodes> g)
```

1. Transitive Hülle

In Anwendungen ist es oft interessant zu wissen, ob man überhaupt von einem Knoten v zu einem Knoten w gelangt, ganz gleich wie lange der Weg auch ist. Dieses **Erreichbarkeitsproblem** hängt mit dem transitiven Abschluss einer binären Relation zusammen, bzw. mit der **transitiven Hülle** eines **Graphen**.

Die **transitive Hülle** eines Graphen umfasst die Kantenmenge des Graphen.

Zusätzlich existiert eine Kante (v,k) , wenn es einen Knoten k gibt, der auf einem von v ausgehenden Weg liegt.

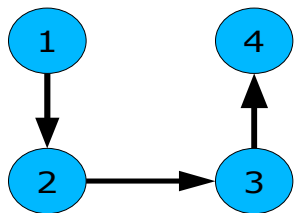


Der Algorithmus von **Warshall** basiert auf der **Adjazenzmatrix-Darstellung** und **ermittelt** für jedes **nicht adjazente Kante (i,j)** , ob es einen **Weg von i über einen Knoten k nach j** gibt.

Man geht von der Adjazenzmatrix des Graphen $H_0=A[i,j]$ aus und ermittelt eine neue Matrix H_1 , die neue Wege zwischen Knoten angibt. Das macht man solange, bis keine neuen Kanten mehr hinzugefügt werden können.

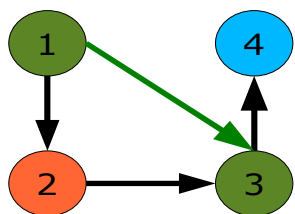
Beim Übergang von $H_{k-1}[i,j]$ nach $H_k[i,j]$ entscheidet man:

Existiert in H_{k-1} ein Weg von i nach k und von k nach j , dann wird $H_k[i,j]=1$
ansonsten wird $H_k[i,j]=0$



$$H_1$$

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



$i=1, j=3$

$$H_2$$

	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0

Dies führt zu einem Verfahren mit Komplexität $O(n^3)$.

```
$ cat Algorithmen.cpp
...
// Warshall Algorithmus zum Berechnen der transitive Hülle einer Matrix
template <class TV, int maxNodes>
bool *Warshall(Digraph<TV, maxNodes> g) { // berechnet Matrix mit
                                         //transitiver Hülle von g

    bool *H = new bool[2*maxNodes];
    for (int i=0; i<maxNodes; i++)
        for (int j=0; j<maxNodes; j++)
            H[i*maxNodes+j] = g.IsArc(i,j); // Initialisiere Feld [i,j]

    while (true) {
        bool b = false;
        for (int i=0; i<maxNodes; i++)
            for (int j=0; j<maxNodes; j++)
                if (!H[i*maxNodes+j]) // kein Weg von i nach j
                    for (int k=0; k<maxNodes; k++)
                        if (H[i*maxNodes+k] && H[k*maxNodes+j]) { // Weg von i über k nach j
                            H[i*maxNodes+j] = true;
                            b = true;
                            break;
                        }
                if (!b) break; // Änderung in Matrix H?
    } // while true
    return H;
}
```

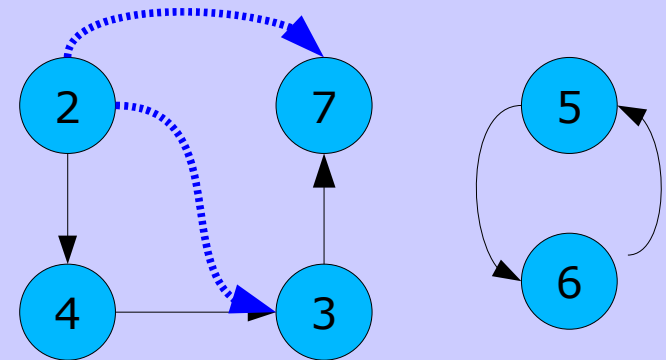
```
$ warshall
```

```
Adjazenz Matrix:
```

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	0	-	-	-
3	-	-	-	-	-	-	-	0
4	-	-	-	0	-	-	-	-
5	-	-	-	-	-	-	0	-
6	-	-	-	-	-	0	-	-
7	-	-	-	-	-	-	-	-

```
Warshall Matrix:
```

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	1	1	0	0	1
3	0	0	0	0	0	0	0	1
4	0	0	0	1	0	0	0	1
5	0	0	0	0	0	1	1	0
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	0



2. Kürzeste Wege

Bisher haben uns die Kantengewichte nicht interessiert. Nun wollen wir Fragestellungen, wie etwa folgende beantworten können:

Gegeben sei ein Graph, dessen Knoten Städte sind. Eine Kante führt von einem Knoten v nach w und ist mit x gewichtet, wenn es eine Strasse der Länge x zwischen der Stadt v und w gibt, die befahrbar ist.

Frage: Wie ist die kürzeste Verbindung zwischen Stadt A und Stadt B.

Kürzeste Wege zwischen allen Knoten

Wir wollen hier ein auf **Adjazenzmatrix** basiertes **Verfahren von Floyd** verwenden.

Grundidee:

Analog zum Warshall Algorithmus verwendet man eine Matrix, und iteriert bis sich kein Wert mehr in der Matrix ändert. Die Matrix ist eine **Kostenmatrix**, bei der die Summe der Kantengewichte zwischen Knoten gespeichert werden.

1. $C_0[i,i]=0$

2. $C_0[i,j]=\infty$, falls keine Kante von i nach j existiert

3. $C_0[i,j]=\text{Kantengewicht von } (i,j)$, falls eine Kante von i nach j existiert

4. beim Übergang von C_{k-1} nach C_k wird unterschieden:

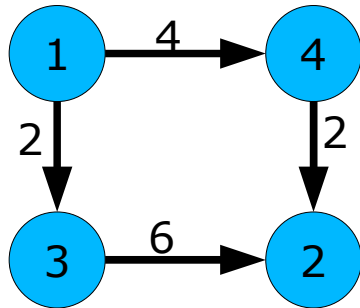
- a) $C_{k-1}[i,j] = \infty$, d.h. es wurde **noch kein Weg** von i nach j festgestellt aber es gibt einen Weg von i nach k und k nach j , dann setze

$$C_k[i,j]=C_{k-1}[i,k]+C_{k-1}[k,j]$$

- b) $C_{k-1}[i,j] \neq \infty$, d.h. es wurde **ein Weg** von i nach j festgestellt **und es gibt einen kürzeren Weg über k** (von i nach k und k nach j), dann setze

$$C_k[i,j]=C_{k-1}[i,k]+C_{k-1}[k,j]$$

Beispiel:



Kostenmatrix

Schritt 1-3: C_0

	1	2	3	4
1	0	∞	2	4
2	∞	0	∞	∞
3	∞	6	0	∞
4	∞	2	∞	0

Kostenmatrix

 $C_0[1,2]=\infty$: 1,3,2

	1	2	3	4
1	0	8	2	4
2	∞	0	∞	∞
3	∞	6	0	∞
4	∞	2	∞	0

Kostenmatrix

 $C_1[1,2]=8$: 1,4,2

	1	2	3	4
1	0	6	2	4
2	∞	0	∞	∞
3	∞	6	0	∞
4	∞	2	∞	0

```
$ cat Algorithmus.cpp
...
// Floyd Algorithmus zum Berechnen der kürzesten Wege
template <class TV, int maxNodes>
float *Floyd(Digraph<TV, maxNodes> g) {
    const float infinite = MAXFLOAT;
    float *C = new float[2*maxNodes];           // Kostenmatrix
    for (int i=0; i<maxNodes; i++)               // Kostenmatrix initialisieren
        for (int j=0; j<maxNodes; j++)
            if (g.IsArc(i,j))
                C[i*maxNodes+j] = g.GetArc(i,j); // C[i,j]=Kantengewicht
            else
                C[i*maxNodes+j] = infinite;      // C[i,j]=Unendlich

    for (int i=0; i<maxNodes; i++)               // Kostenmatrix initialisieren
        for (int j=0; j<maxNodes; j++)
            if (i == j ) C[i*maxNodes+j] = 0; // C[i,i] = 0
```

```
while (true) {
    bool b = false; // keine Änderung
    for (int i=0; i<maxNodes; i++)
        for (int j=0; j<maxNodes; j++) {
            if (i == j) continue;
            for (int k=0; k<maxNodes; k++) {
                if (k==i || k==j) continue;
                if (C[i*maxNodes+k]==infinite || C[k*maxNodes+j]==infinite)
                    continue;
                else {
                    float h = C[i*maxNodes+k] + C[k*maxNodes+j];
                    if (h < C[i*maxNodes+j]) { // kürzeren Weg gefunden
                        C[i*maxNodes+j] = h;
                        b = true; // Matrix geändert
                    } // if
                } // else
            } // for
        } // for
    if (!b) break; // Änderung in Matrix H?
} // while true
```

```
return C;
```

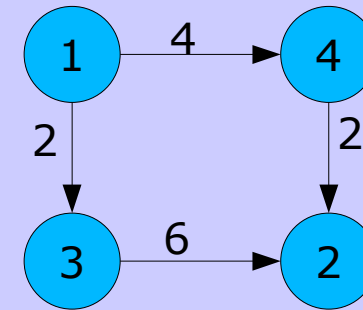
```
}
```

```
...
```

```
$
```

Adjazenz Matrix:

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	2	-	-	4
3	-	-	-	-	-	-	-	-
4	-	-	-	6	-	-	-	-
5	-	-	-	-	-	-	1	-
6	-	-	-	-	-	2	-	-
7	-	-	-	2	-	-	-	-



Cost Matrix:

	0	1	2	3	4	5	6	7
0	0	-	-	-	-	-	-	-
1	-	0	-	-	-	-	-	-
2	-	-	0	6	2	-	-	4
3	-	-	-	0	-	-	-	-
4	-	-	-	6	0	-	-	-
5	-	-	-	-	-	0	1	-
6	-	-	-	-	-	2	0	-
7	-	-	-	2	-	-	-	0

Das o.a. Verfahren hat wg. der drei geschachtelten Schleifen eine Laufzeit Komplexität von $O(n^3)$. Will man die **längsten Wege** berechnen, so ist das Floyd Verfahren einfach so abzuändern, dass anstelle der Minimumberechnung das **Maximum** ermittelt wird und als Initialwert `MINFLOAT` verwendet wird.