Inhaltsverzeichnis

1. Das Shortest Path Problem

1.1. Ziele

☑ Algorithmen und Datenstrukturen kennen lernen.

☑ **Finde den kürzesten Pfad von A nach B.**

1.2. Fragestellung: Routenplanung

- Suche die kürzeste Fahrtzeit oder
- suche die geringsten Fahrtkosten zwischen zwei Orten.

http://fuzzy.cs.uni-magdeburg.de/studium/graph/txt/duvigneau.pdf

1.3. Definition: gerichteter Graph

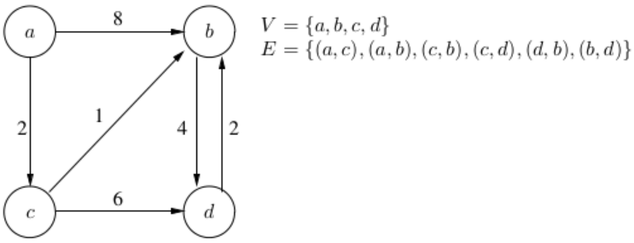
Ein **gerichteter Graph** $G=(V,E)$ ist die **Zusammensetzung**

- **einer Menge V von Knoten (Vertex) und**
- **einer Menge von Kanten (Edge) mit** $E \subset V \times V$

Beispiel:

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

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



1.4. Implementierung von Graphen

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

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

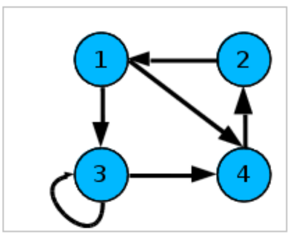
Die „richtige“ Wahl hängt von der Aufgabenstellung und davon ab, ob der Graph eher dichte oder dünne Kantenbelegung hat.

1.4.1. Adjazenzmatrix

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

$$a_{i,j}=1, \quad falls\,(v_i, v_j) \in E$$
$$a_{i,j}=0, \quad falls\,(v_i, v_j) \notin E$$

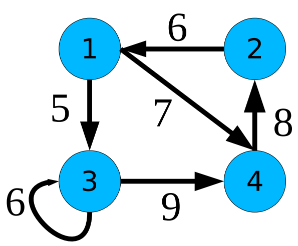
1.4.2. Beispiel: ungewichteter Graph



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

1.4.3. Beispiel: gewichteter Graph

$$a_{i,j} = c(i,j), \quad falls\,(v_i, v_j) \in E \; mit\; C: E \rightarrow \mathbb{R}$$
$$a_{i,j} = 0, \quad falls\,(v_i, v_j) \notin E$$



	1	2	3	4
1	0	0	5	7
2	6	0	0	0
3	0	0	6	9
4	0	8	0	0

Vor/Nachteile von Adjazenzmatrix :

- Platzbedarf = O(|V|^2).
- Direkter Zugriff auf Kante (i, j) in konstanter Zeit möglich.
- Kein effizientes Verarbeiten der Nachbarn eines Knotens.
- Sinnvoll bei dicht besetzten Graphen.
- Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

1.4.4. Aufgabe: AdjazenzMatrix

1. Erstellen Sie die Klasse **AdjazenzMatrix** zur Verwaltung eines **gewichteten** Graphen.
2. Testen Sie Ihr Programm, indem Sie folgende **Eingabe-Datei** (s.u: adjazenzmatrix-daten.txt) einlesen und dann die AdjazenzMatrix in Tabellenform ausgeben.
3. Achten Sie darauf, daß **innerhalb der Klasse** der Zugriff auf die Matrix-Elemente mit einem **int-Wert** als Index erfolgen soll.
   Der **Benutzer der Klasse verwendet aber Namen** (hier Städtenamen). D.h. in der Klasse muss es eine Zuordnung Name<->Index geben.

4. **Tipp:** Vielleicht bietet hier die Klasse *map* aus der STL Unterstützung?

```
// test-adjazenzmatrix.cpp
//

#include <string>
#include <iostream>
#include <fstream>

#include "adjazenzmatrix.h"

int main(){
      int dimension;
      int entfernung;
      int edges;
      string von, nach, stadt;
      fstream fin("adjazenzmatrix-daten.txt");

      fin >> dimension;
      AdjazenzMatrix<int> adjMatrix(dimension);

      // Die Knoten einlesen
      for (int i=0; i < dimension; i++){
            fin >> stadt;
            adjMatrix.addNode(stadt);
      }
```

```
      fin >> edges;
      // Die Kanten einlesen
      for (int i=0; i < edges; i++){
            fin >> von; fin >> nach; fin>>entfernung;
            adjMatrix.addEdge(von,nach,entfernung);
      }

      cout << "Die AdjazenzMatrix ..." << endl;
      cout << adjMatrix.toString() <<endl;


      fin.close();
      return 0;
}
```

Beispiel für die Ausgabe:

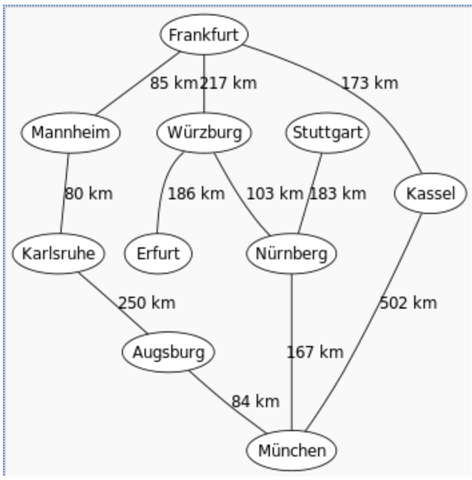
Adjazenzmatrix ...

.	Frankfurt	Mannheim	Wuerzburg	Stuttgart	Kassel	Karlsruhe	Erfurt	Nuernberg	Augsburg	Muenchen
Frankfurt	0	85	217	0	173	0	0	0	0	0
Mannheim	85	0	0	0	0	80	0	0	0	0
Wuerzburg	217	0	0	0	0	0	186	103	0	0
Stuttgart	0	0	0	0	0	0	0	183	0	0
Kassel	173	0	0	0	0	0	0	0	0	502
Karlsruhe	0	80	0	0	0	0	0	0	250	0
Erfurt	0	0	186	0	0	0	0	0	0	0
Nuernberg	0	0	103	183	0	0	0	0	0	167
Augsburg	0	0	0	0	0	250	0	0	0	84
Muenchen	0	0	0	0	502	0	0	167	84	0

Datei: adjazenzmatrix-daten.txt

```
10
Frankfurt
Mannheim
Wuerzburg
Stuttgart
Kassel
Karlsruhe
Erfurt
Nuernberg
Augsburg
Muenchen
22
Frankfurt   Mannheim    85
Frankfurt   Wuerzburg   217
Frankfurt   Kassel      173
Mannheim    Frankfurt   85
Mannheim    Karlsruhe   80
Wuerzburg   Frankfurt   217
Wuerzburg   Erfurt      186
Wuerzburg   Nuernberg   103
Stuttgart   Nuernberg   183
Kassel      Frankfurt   173
Kassel      Muenchen    502
Karlsruhe   Mannheim    80
Karlsruhe   Augsburg    250
Erfurt      Wuerzburg   186
Nuernberg   Wuerzburg   103
Nuernberg   Stuttgart   183
Nuernberg   Muenchen    167
Augsburg    Karlsruhe   250
Augsburg    Muenchen    84
Muenchen    Augsburg    84
Muenchen    Nuernberg   167
Muenchen    Kassel      502
```

Diese Datei ist folg. Graphen zugeordnet:



1.4.5. *Tipp: Template Klassen*

Das obige Beispiel verwendet sogenannte Template-Klassen. Sehen Sie hier ein kleines Beispiel.

```
template <typename T>
class CStack{
 T *pData;
 ....
 public:
 CStack(int size){
 pData = new T[size];
 ....
 }
 bool Push(const T& val);
 bool Pop(T& val);
};
```

...

```
template <typename T>
bool CStack<T>::Push(const T& val)
{....}

template <typename T>
bool CStack<T>::Pop(T& val)
{....}
```

1.5. Floyd-Algo: Der kürzeste Weg zwischen 2 beliebigen Knoten

https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_de.html

Auch **All-Pair** shortest Path **(APSP)** genannt.

Berechne in einem Graphen den kürzesten Weg zwischen bel. 2 Knoten.

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

1.5.1. Grundidee

Man verwendet zunächst eine sog. Kostenmatrix C, die in den Zellen(=Kanten des Graphen) die Kosten (zB: Entfernung) speichert.

Es gilt:

1. C[i,i]=0
2. C[i,j]=∞, falls keine Kante von i nach j existiert
3. C[i,j]=Kantengewicht von (i,j), falls eine Kante von i nach j existiert
   anders ausgedrückt:
   C[i,j]= len(i,j)

Wenn man

- 1 Kante des Graphen berücksichtigt (also nur einen Knoten weit geht), enthält die Kostenmatrix C bereits die kürzeste Entfernung von i nach j.

Wenn man

- 2 od. mehrere Kanten des Graphen berücksichtigen (also 2 od. mehrere Knoten weit geht),
  muss man die kürzeste Entfernung aller Entfernungen der Art C[i,k]+ C[k,j] mit k ist die Anzahl der Knoten suchen.
  Wir sagen: Suche den kürzesten Umweg zwischen i und j über alle k.

  Wir brechnen also für die neue Kostenmatrix D (wir wollen sie Distanzmatrix nennen):
  D[i,j]= $\min_k$(C[i,k]+ C[k,j])

Wenn man

- nun die Matrixmultiplikation betrachtet:
  D = C x C

  D[i,j]= Summe_k (C[i,k] * C[k,j]) mit k= 1...n

  dann sieht man, dass man

  - statt der Summe das Minimum und
  - statt des Produktes die Addition verwenden muss.

- Es gilt also (nach Floyd):

  D[i,j]= $\mathbf{MIN_k}$ (C[i,k] + C[k,j]) mit k= 1...n

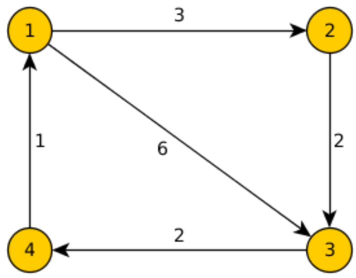
1.5.2. Beispiel

Gegeben sei:

C

	1	2	3	4
1	0	3	6	---
2	---	0	2	---
3	---	---	0	2
4	1	---		0

--- bedeutet unendlich



1. Wenn k=1 (also nur ein Knoten weit) folgt: D= C
2. Wenn k=4 (also alle 4 Knoten berücksichtigt werden)

   Man sieht aus dem Graphen: Die kürzeste Entfernung (1,3) ist 5.
   Nämlich über den Umweg Knoten 2.

Der Algorithmus: Berechne das Minimum aller Umwege k (k=1,2,3,4):

Für die Zelle (1,3) also den kürzesten Weg von Knoten 1 zu Knoten 3:

D[1,3]= MIN { (C[1,1] + C[1,3]) , (C[1,2] + C[2,3]), (C[1,3] + C[3,3]),(C[1,4] + C[4,3]) }

D[1,3]= MIN { (0 + 6) , (3 + 2), (6 + 0),(--- + ---) }

D[1,3]= MIN { (6) , (5), (6),(---) }

D[1,3]= 5

Die Zelle (1,3) erhält nun das Minimum 5

D

	1	2	3	4
1	0	3	**5**	
2		0	2	
3			0	2
4	1			0

1.5.3. Verallgemeinerung

Für einen Graph mit n Knoten berechnet man die kürzeste Entfernung zwischen jeweils allen Knoten durch:

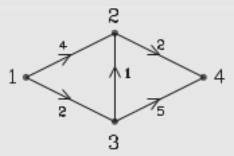
D= C^n

also: D= C x C x ..... x C (n mal)

Um auch die zugehörige Kantenfolge rekonstruieren zu können, wird parallel dazu eine Folge von (n × n)-Matrizen P_1 , P_2 , ..., P_n aufgebaut, die an Position P_k[i,j] den vorletzten Knoten auf dem kürzesten Weg von i nach j notiert, der nur über die Zwischenknoten 1, 2, ..., k-1 läuft.

Anmerkungen zur Implementierung:

Den Ortsnamen werden Indizes (beginnend bei 0) zugeordnet.



Mit dem Algorithmus von Floyd-Warshall ergeben sich die folgenden Matrizen:

$$D_0 = \begin{pmatrix} 0 & 4 & 2 & \infty \\ \infty & 0 & \infty & 2 \\ \infty & 1 & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad D_1 = \begin{pmatrix} 0 & 4 & 2 & \infty \\ \infty & 0 & \infty & 2 \\ \infty & 1 & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 4 & 2 & 6 \\ \infty & 0 & \infty & 2 \\ \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad D_3 = \begin{pmatrix} 0 & 3 & 2 & 5 \\ \infty & 0 & \infty & 2 \\ \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

$$D_4 = D_3$$

Hier der Algorithmus in Java notiert:

```
/***************************** Floyd.java *************************************/

/** berechnet alle kuerzesten Wege und ihre Kosten mit Algorithmus von Floyd  */
/*  der Graph darf keine Kreise mit negativen Kosten haben                    */

public class Floyd {

  public static void floyd (int n,          // Dimension der Matrix
                            double [][] c,  // Adjazenzmatrix mit Kosten
                            double [][] d,  // errechnete Distanzmatrix
                            int    [][] p){ // errechnete Wegematrix

    int i, j, k;                            // Laufvariablen
    for (i=0; i < n; i++) {                 // fuer jede Zeile
      for (j=0; j < n; j++) {               // fuer jede Spalte
        d[i][j] = c[i][j];                  // initialisiere mit Kantenkosten
        p[i][j] = i;                        // vorletzter Knoten
      }                                     // vorhanden ist nun D hoch -1
    }

    for (k=0; k < n; k++) {                 // fuer jede Knotenobergrenze
      for (i=0; i < n; i++) {               // fuer jede Zeile
        for (j=0; j < n; j++) {             // fuer jede Spalte
          if (d[i][k] + d[k][j] < d[i][j]){ // falls Verkuerzung moeglich
            d[i][j] = d[i][k] + d[k][j];    // notiere Verkuerzung
            p[i][j] = p[k][j];              // notiere vorletzten Knoten
          }                                 // vorhanden ist nun D hoch k
        }
      }
    }
  }
}
```

1.5.4. Aufgabe: AdjazenzMatrix mit All-Pair-Shortest-Path-Algorithmus

1. Erweitern Sie die Klasse Adjazenzmatrix um die Methode

T getShortestDistance(string von, string nach)

die nach dem All-Pair-Shortest-Path Algorithmus den kürzesten Weg zwischen 2 Knoten berechnet.

2. Erweitern Sie die Method **toString**(), sodass auch die Distanzmatrix ausgeben wird.

```
Adjazenzmatrix ...
          .  Frankfurt   Mannheim  Wuerzburg  Stuttgart     Kassel  Karlsruhe     Erfurt  Nuernberg   Augsburg   Muenchen
  Frankfurt          0         85        217          0        173          0          0          0          0          0
   Mannheim         85          0          0          0          0         80          0          0          0          0
  Wuerzburg        217          0          0          0          0          0        186        103          0          0
  Stuttgart          0          0          0          0          0          0          0        183          0          0
     Kassel        173          0          0          0          0          0          0          0          0        502
  Karlsruhe          0         80          0          0          0          0          0          0        250          0
     Erfurt          0          0        186          0          0          0          0          0          0          0
  Nuernberg          0          0        103        183          0          0          0          0          0        167
   Augsburg          0          0          0          0          0        250          0          0          0         84
   Muenchen          0          0          0          0        502          0          0        167         84          0

Distanz-Matrix ...
          .  Frankfurt   Mannheim  Wuerzburg  Stuttgart     Kassel  Karlsruhe     Erfurt  Nuernberg   Augsburg   Muenchen
  Frankfurt          0         85        217        503        173        165        403        320        415        487
   Mannheim         85          0        302        588        258         80        488        405        330        414
  Wuerzburg        217        302          0        286        390        382        186        103        354        270
  Stuttgart        503        588        286          0        676        668        472        183        434        350
     Kassel        173        258        390        676          0        338        576        493        586        502
  Karlsruhe        165         80        382        668        338          0        568        485        250        334
     Erfurt        403        488        186        472        576        568          0        289        540        456
  Nuernberg        320        405        103        183        493        485        289          0        251        167
   Augsburg        415        330        354        434        586        250        540        251          0         84
   Muenchen        487        414        270        350        502        334        456        167         84          0

Kürzester Weg: von München nach Frankfurt = 487
```

Hinweise:

Fügen Sie folgende private Members hinzu:

```
	T** distMatrix;		// DistanzMatrix
	int** pathMatrix;	// Pfadmatrix
```

Fügen Sie auch die private Methode

```
void all_pair_shortest_path (){ // nach Floyd
…
}
```

hinzu, die von getShortestDistance() benutzt wird und die beiden neuen Matrizen berechnet.