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

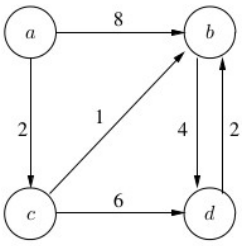
Ein **gerichteter Graph** $G=(V,E)$ ist die **Zusammensetzung**

- **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.



$V = \{a, b, c, d\}$
$E = \{(a, c), (a, b), (c, b), (c, d), (d, b), (b, d)\}$

1.4. Implementierung von Graphen

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

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

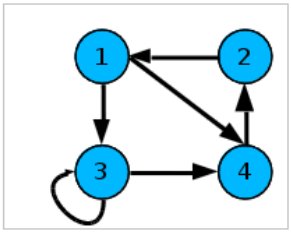
Die „richtige“ Wahl hängt von der Aufgabenstellung und davon ab, ob der Graph eher dichte oder dünne Kantenbelegung hat.

1.4.1. Adjazenzmatrix

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

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

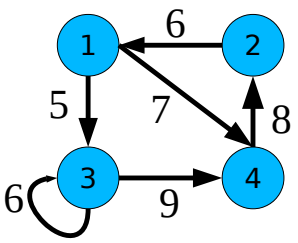
1.4.2. Beispiel: ungewichteter Graph



	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: Template Klasse Matrix

Studiere die template Klasse Matrix.

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 2 Knoten.

Wir wollen hier ein auf **Adjazenzmatrix** basiertes **Verfahren von Floyd-Warshall** 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

  $D[i,j] = \text{Summe}_k\ (C[i,k] * C[k,j])$ 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):

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

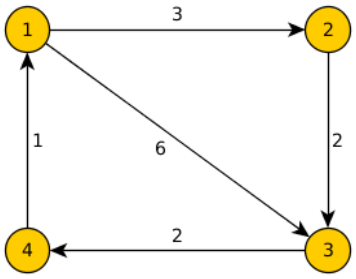
1.5.2. Beispiel

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:

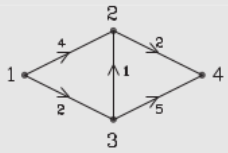
$D = C^n$

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:

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

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

$$D_4 = D_3$$

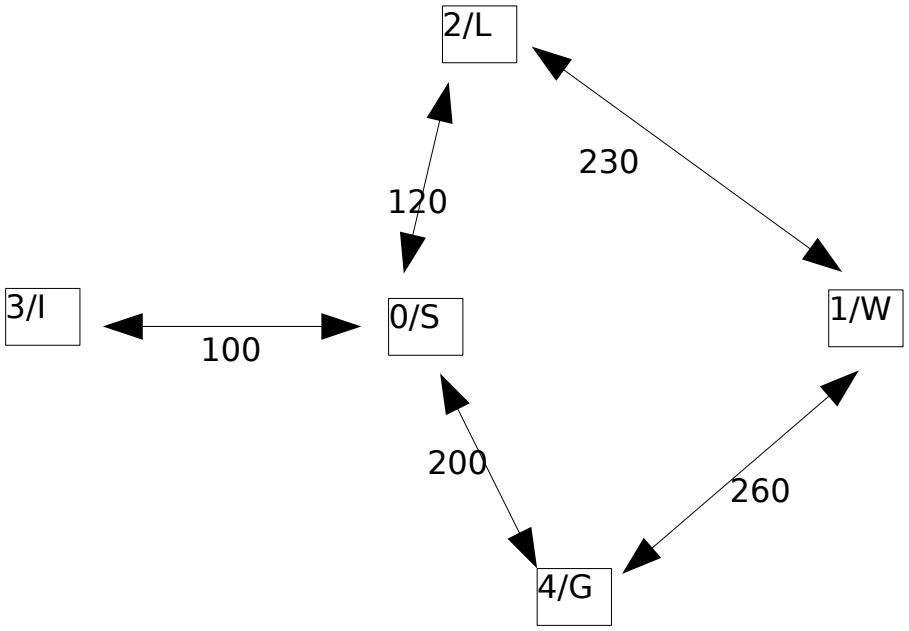
Hier der Algorithmus in Java notiert:

```
/***************************** 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. RDP-AUFGABE: FLOYD-WARSHALL-All-Pair-Shortest-Path

- Gegeben: Template Klasse: Matrix
- Gegeben: Skriptum: Floyd-Warshall all pair shortest path
- Gegeben: net.h (s.u.)
- Gegeben: main.cpp (s.u.)
- **GESUCHT: net.cpp**

- Gegeben: Folgendes Netz
    - 0(Salzburg) <-> 2(Linz) : 120
    - 0(Salzburg) <-> 3(Innsbruck) : 100
    - 0(Salzburg) <-> 4(Graz) : 200
    - 2(Linz) <-> 1(Wien) : 230
    - 4(Graz) <-> 1(Wien) : 260



- Folgende Ausgabe muss generiert werden, wenn der kürzeste Weg von Innsbruck nach Wien gesucht wird.

Kürzeste Verbindung: von (3/I) nach (1/W): 450
Route:
(3/I) nach (0/S): 100
(0/S) nach (2/L): 120
(2/L) nach (1/W): 230