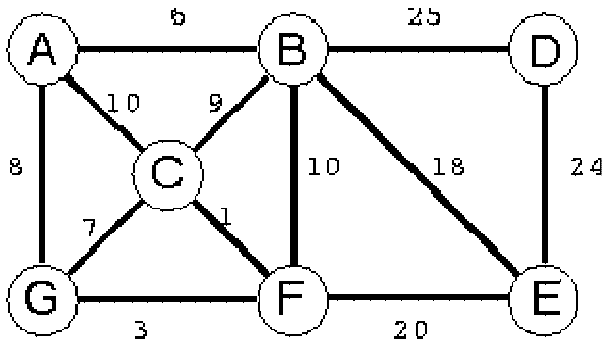
Aufg. 1:

Minimum-Spanning-Tree (MST)

Wir hatten uns im Unterricht mit dem Minimum-Spanning-tree-Problem(MST) beschäftigt.

1.1 Beschreiben Sie, wie das MST-Problem für einen Graphen lautet. Nennen Sie einen Anwendungszusammenhang für das MST-Problem. Für welche beispielhafte „alltägliche" Fragestellung kann der MST interessant sein?



1.2 Beschreiben Sie, wie der Lösungsansatz von Kruskal funktioniert (nur Angabe der entscheidenden Ideen).

1.3 Ermittlen Sie für rechts-stehenden Graphen den MST nach Kruskal und dokumentieren Sie dabei Ihr Vorgehen graphisch und mit kurzen Erklärungen.

1.4 Wir haben in der Schule noch ein weiteres Verfahren kennen gelernt den MST zu berechnen. Stellen sie dieses kurz dar.

Aufg. 2:

Kürzeste Wege in einem Graphen: Der Algorithmus von Dijkstra

2.1 Ermittlen Sie mit Hilfe des Algorithmus von Dijkstra den kürzesten Weg von A nach F in folgendem Graphen, der durch eine Adjazenzmatrix gegeben ist:

Nutzen Sie dazu folgende Tabellen:

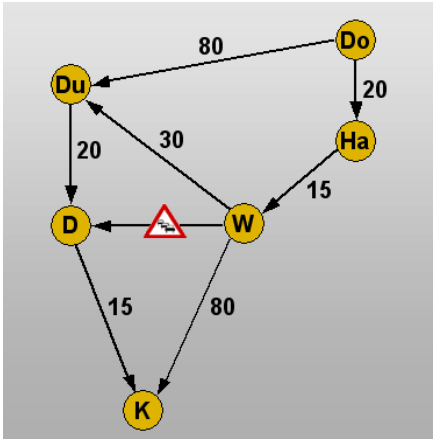
	A	**B**	**C**	**D**	**E**	**F**
A	0	2	0	1	0	0
B	2	0	3	2	0	0
C	0	3	0	3	1	5
D	1	2	3	0	6	0
E	0	0	1	6	0	2
F	0	0	5	0	2	0

(von, Entfernung zum Start, über)	**Randknoten**

Sichergestellte kürzeste Entfernung von A nach	**B**	**C**	**D**	**E**	**F**

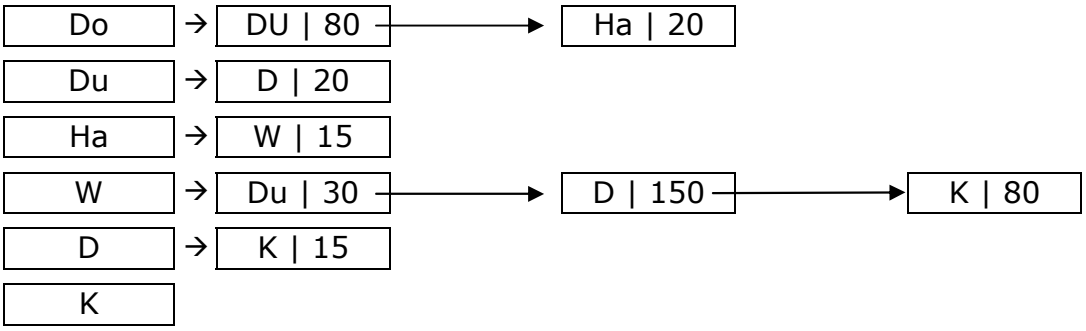
Wie man leicht feststellt, braucht man zur Abspeicherung eines Graphen mit n Knoten n² Eintragsplätze in einer Adjazenzmatrix. Viele davon erhalten aber den Eintrag 0, da im Allgemeinen zwischen zwei Knoten häufiger keine Kante gibt als dass es eine Kante gibt. Die Einträge „0" enthalten also damit keine relevanten Informationen.

2.2 Berechnen Sie die „Quote" der eingetragenen Werte in der Adjazenzmatrix aus Aufgabe 2.1, indem Sie die wirklichen Informationen zählen.



Aus obiger Erkenntnis wächst die Idee, Graphen platzsparender zu speichern, indem man zunächst nur Namen der Knoten in einem einfachen Array speichert. An jedes Feld dieses Arrays wird dann eine Liste mit den adjazenten Knoten und ihren Gewichten gehängt.

Beispiel: Es wird das Beispiel aus dem Unterricht (s. rechts oben) aufgegriffen (wobei die Strecke W ⇔ D mit dem Gewicht 150 besetzt wird). Es entsteht dann folgende **Adjazenzliste**:



Achtung : Die Grafik sagt nicht aus, dass eine Kante von DU nach HA geht, sondern dass DU und HA beide von DO aus direkt erreichbar sind!

2.3 Erstellen Sie für den Graphen, der in Aufgabe 2.1 als Adjazenzmatrix gegeben ist, die Adjazenzliste.

2.4 Wir hatten im Unterricht ausführlich Listenstrukturen angewendet. Dabei hatten wir in die Listenstruktur allgemeine Objekte eingefügt. In diese Listenstruktur werden offensichtlich Objekte vom Typ `Edge` eingefügt, die sowohl den Namen (`pNeighbour`) als auch eine Gewichtung der Kante (`pWeight`) beinhalten.
In der Anlage finden Sie die Klassendokumentationen der Klassen `Edge` und `List` (die Listendefinition stammt aus dem kommenden Zentralabitur), mit deren Hilfe hier gearbeitet werden soll.
Erstellen Sie eine umgangssprachliche Methode, die aus einer gegebenen Adjanzenzmatrix (ein 2-Dimensionales Array mit Namen `matrix`) der Größe 5x5 die Adjazenzliste aufbaut. Dazu sei die Adjazenzliste durch `List []aliste = new List[5];` gegeben.

Aufg. 3:

Der Algorithmus von Warshall

Gegeben sei unten stehender [gerichteter] Graph (Abb. 3.1). Bei der Ermittlung der sogenannten „**transitiven Hülle**" des Graphen werden genau dann zwei Knoten des Graphen direkt miteinander verbunden, wenn es irgendeinen Weg zwischen diesen Knoten durch den Graphen gibt. Wie Abb. 3.2 zeigt, wird dann also zum Beispiel er Knoten 4 mit dem Knoten 3 verbunden, da ja eine Verbindung von 4 nach 3 (über Knoten 1) besteht. Genauso wird ein Knoten durch einer Kante mit sich selbst verbunden, falls es einen Rundweg von diesem Knoten zu sich selbst gibt (z.B. Knoten 2: Rundweg 2->3->1->2 oder Knoten 3: 3->1->3). Es entsteht auf diese Art und Weise ein neuer Graph.

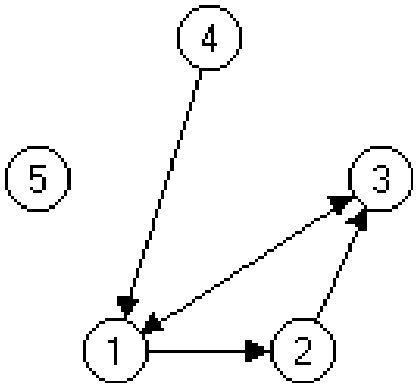


Abb. 3.1

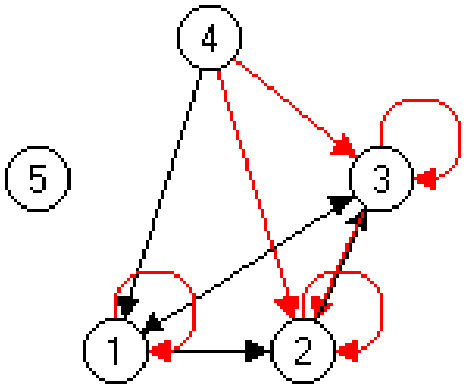


Abb. 3.2: transitive Hülle von Graph 3.1

3.1 *Überlegen Sie sich eine Anwendung/Situation, in der die Suche nach der transitiven Hülle eines Graphen von Interesse sein kann und geben sie dieses an.*

3.2 *Füllen Sie nachfolgende Adjazenzmatrix für die Graphen aus. Markieren Sie dabei farbig, welche Kanten vom Ausgangsgraph 3.1 zum Graph 3.2 hinzu gefügt wurden:*

		nach				
		1	**2**	**3**	**4**	**5**
von	**1**					
	2					
	3					
	4					
	5					

Der **Algorithmus von Warshall** ermittelt die transitive Hülle eines Graphen, indem Kanten schrittweise zum bestehenden Graphen hinzugefügt werden:

- Im ersten Schritt kommt eine Kante (i,j) [= Kante von Knoten i nach Knoten j] hinzu, falls sich aus **zwei** Kanten ein Weg von i nach j bilden lässt, der über den Knoten 1 führt (also (i,1) und (1,j) existieren).
- Im zweiten Schritt kommt eine Kante (i,j) [= Kante von Knoten i nach Knoten j] hinzu, falls sich aus **zwei** Kanten ein Weg von i nach j bilden lässt, der über den Knoten 2 führt (also (i,2) und (2,j) existieren). Dabei werden die neuen Kanten, die im vorherigen Schritt hinzugekommen sind, mit berücksichtigt!
- Im dritten Schritt kommt eine Kante (i,j) [= Kante von Knoten i nach Knoten j] hinzu, falls sich aus **zwei** Kanten ein Weg von i nach j bilden lässt, der über den Knoten 3 führt (also (i,3) und (3,j) existieren). Dabei werden die neuen Kanten, die in den vorherigen Schritten hinzugekommen sind, mit berücksichtigt!
- ...
- Dieses Verfahren wird bis zum n-ten Knoten (n = Anzahl Knoten, hier also n=5) fortgesetzt.

3.3 *Im Graph 3.2 sind folgende Kanten neu hinzu gewonnen worden. Überlegen Sie sich, in welcher Reihenfolge und in welchem Schritt sie vom Warshalls-Algorithmus hinzugefügt wurden:*

Kante (i,j)	**Hinzugefügt als __ . Kante**	**in Schritt ....**
(1,1)		
(2,1)		
(2,2)		
(3,2)		
(3,3)		
(4,2)		
(4,3)		

3.4 *Die Adjazenzmatrix des Ausgangsgraphen 3.1 sei im Array a gespeichert (wir lassen hier die Array-Nummerierung bei 1 beginnen [siehe Aufg. 3.2]!!!). Schreiben Sie eine Java-Methode* `void warshall()`*, die die transitive Hülle des Graphen nach dem Algorithmus von Warshall ermittelt. Wichtig:* ***Schreiben sie zunächst einen Algorithmus damit ich verstehe was sie meinen.***

Tipp: Sie brauchen sicherlich mehrfach verschachtelte for-Schleifen.

Anlage zu Aufgabe 2:

Die Klasse List

Objekte der Klasse *List* verwalten beliebige Objekte nach einem Listenprinzip. Ein interner Positionszeiger wird durch die Listenstruktur bewegt, seine Position markiert ein aktuelles Objekt. Die Lage des Positionszeigers kann abgefragt, verändert und die Objektinhalte an den Positionen können gelesen oder verändert werden.

Konstruktor `List()`
Nachher Eine leere Liste ist angelegt. Der interne Positionszeiger steht vor der leeren Liste

Anfrage `isEmpty(): boolean`
Nachher Die Anfrage liefert den Wert ***true***, wenn die Liste keine Elemente enthält, sonst liefert sie den Wert ***false***.

Anfrage `isBefore(): boolean`
Nachher Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger vor dem ersten Listenelement oder vor der leeren Liste steht, sonst liefert sie den Wert ***false***.
Anfrage `isBehind(): boolean`
Nachher Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger hinter dem letzten Listenelement oder hinter der leeren Liste steht, sonst liefert sie den Wert ***false***.

Auftrag `next()`
Nachher Der Positionszeiger ist um eine Position in Richtung Listenende weitergerückt, d.h. wenn er vor der Liste stand, wird das Element am Listenanfang zum aktuellen Element, ansonsten das jeweils nachfolgende Listenelement. Stand der Positionszeiger auf dem letzten Listenelement, befindet er sich jetzt hinter der Liste. Befand er sich hinter der Liste, hat er sich nicht verändert.

Auftrag `previous()`
Nachher Der Positionszeiger ist um eine Position in Richtung Listenanfang weitergerückt, d.h. wenn er hinter der Liste stand, wird das Element am Listenende zum aktuellen Element, ansonsten das jeweils vorhergehende Listenelement. Stand der Positionszeiger auf dem ersten Listenelement, befindet er sich jetzt vor der Liste. Befand er sich vor der Liste, hat er sich nicht verändert.

Auftrag `toFirst()`
Nachher Der Positionszeiger steht auf dem ersten Listenelement. Falls die Liste leer ist befindet er sich jetzt hinter der Liste.

Auftrag `toLast()`
Nachher Der Positionszeiger steht auf dem letzten Listenelement. Falls die Liste leer ist, befindet er sich jetzt vor der Liste.

Anfrage `getItem(): Object`
Nachher Die Anfrage liefert den Wert des aktuellen Listenelements bzw. *null*, wenn die Liste keine Elemente enthält, bzw. der Positionszeiger vor oder hinter der Liste steht.

Auftrag `update (Object pObject)`
Vorher Die Liste ist nicht leer. Der Positionszeiger steht nicht vor oder hinter der Liste.
Nachher Der Wert des Listenelements an der aktuellen Position ist durch *pObject* ersetzt.

Auftrag **`insertBefore (Object pObject)`**
Vorher Der Positionszeiger steht nicht vor der Liste.
Nachher Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und vor der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht hinter dem eingefügten Element.

Auftrag **`insertBehind (Object pObject)`**
Vorher Der Positionszeiger steht nicht hinter der Liste.
Nachher Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und hinter der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht vor dem eingefügten Element.

Auftrag **`delete()`**
Vorher Der Positionszeiger steht nicht vor oder hinter der Liste.
Nachher Das aktuelle Listenelement ist gelöscht. Der Positionszeiger steht auf dem Element hinter dem gelöschten Element, bzw. hinter der Liste, wenn das gelöschte Element das letzte Listenelement war.

Die Klasse Edge

Objekte der Klasse *Edge* sind Kanten eines Graphen. Eine Kante hat ein Gewicht und den Namen des zu ihr gehörenden Nachbarknotens.

Konstruktor **`Edge (String pNeighbour, double pWeight)`**
nachher Eine Kante mit dem Nachbarknoten *pNeighbour* und dem Gewicht *pWeight* wurde erzeugt.

Anfrage **`neighbour(): String`**
nachher Die Anfrage liefert den Namen Nachbarknotens des Knotens, zu dem die Kante gehört.

Anfrage **`weight(): double`**
nachher Die Anfrage liefert das Gewicht der Kante.