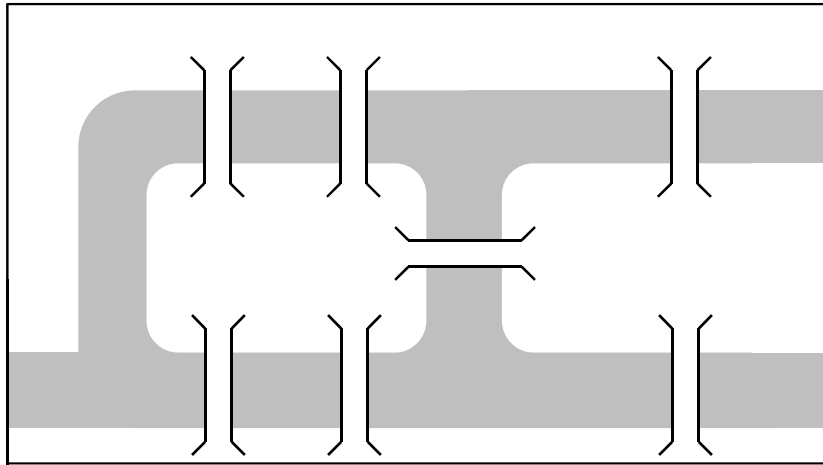


## Kapitel 17

# Graphentheoretische Algorithmen

## Das Königsberger Brückenproblem - Aufgabenstellung

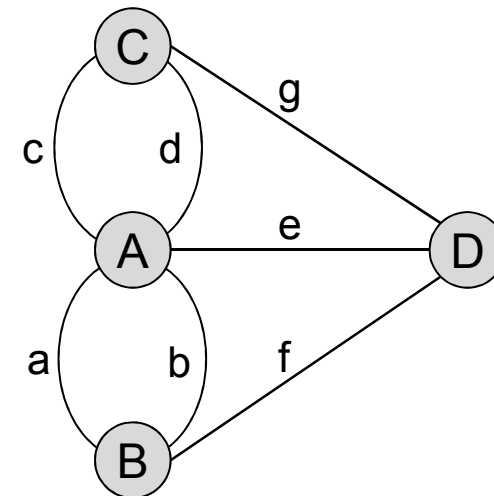
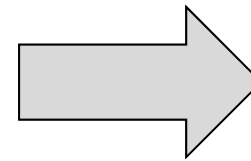
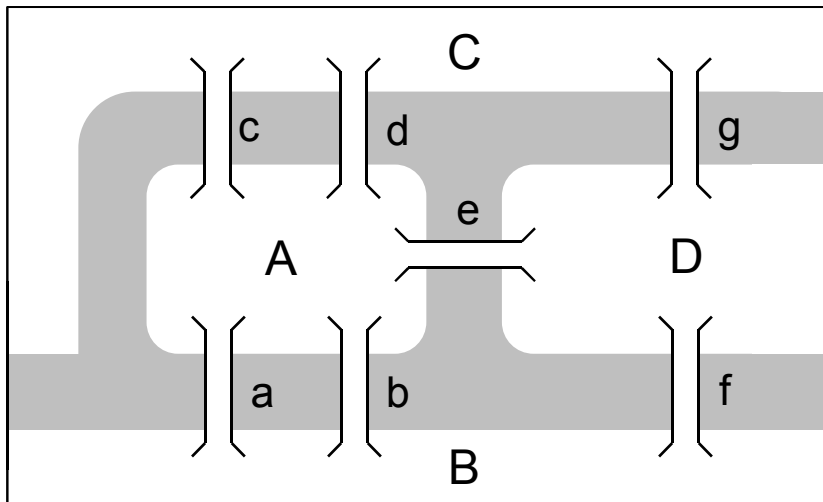


Die geografische Lage von Königsberg am Pregel ist gekennzeichnet durch vier Landgebiete (Festland oder Inseln), die durch sieben Brücken miteinander verbunden sind.

Problemstellung:

Finde einen Rundweg, der jede Brücke genau einmal enthält.

Formalisierung als "Graph" mit "Knoten" (A-D) und "Kanten" (a-g):



Das Königsberger Brückenproblem ist ein klassisches Problem der "Graphentheorie", das auf den berühmten Mathematiker Leonhard Euler (1707-1783) zurückgeht. Den gesuchten Rundweg bezeichnet man auch als **Eulerschen Weg**.

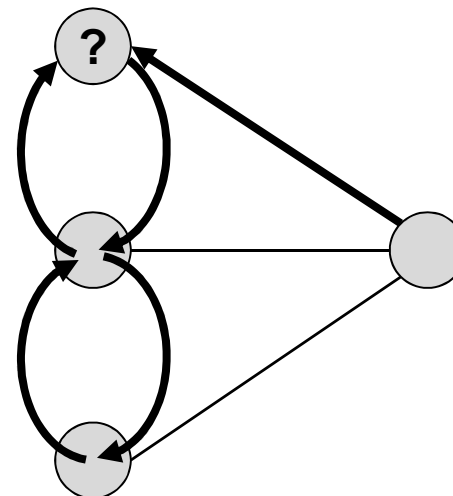
## Die Unlösbarkeit des Königsberger Brückenproblems

Beim Versuch, das Königsberger Brückenproblem zu lösen, stößt man auf drei, für die Existenz einer Lösung notwendige Bedingungen:

1. Der Graph muss **zusammenhängend** sein. Das heißt, man muss jeden Knoten von jedem anderen Knoten aus über einen Weg erreichen können.
2. Zu dem Startknoten muss es neben der Kante, über die man ihn verlässt, eine weitere Kante geben, über die man ihn am Ende des Wegs wieder erreicht.
3. Wenn man einen Knoten auf dem gesuchten Rundweg über eine Kante erreicht und der Weg noch nicht beendet ist, muss es eine weitere, noch nicht benutzte Kante geben, über den man ihn wieder verlassen kann.

Die Bedingungen 2 und 3 besagen, dass die Kanten an jedem Knoten "paarig" auftreten müssen, damit ein Eulerscher Weg überhaupt existieren kann.

Der Königsberger Graph erfüllt diese Bedingungen nicht. Er ist zwar zusammenhängend, aber es gibt sogar an keinem Knoten eine gerade Anzahl von Kanten. Es kann den gesuchten Rundweg nicht geben. Jeder Versuch wird zwangsläufig scheitern:



## Das allgemeine Brückenproblem

Wir denken uns einen zusammenhängenden Graphen, bei dem es an jedem Knoten eine gerade Anzahl Kanten gibt.

Wir starten an einem beliebigen Knoten zu einer Wanderung. Die dabei benutzten Kanten markieren wir, damit wir sie nicht noch einmal verwenden. Wenn wir zu einem Knoten kommen, versuchen wir den Knoten über eine beliebige, noch nicht benutzte Kante wieder zu verlassen. Irgendwann wird die Wanderung an einem Knoten enden, den wir nicht mehr verlassen können, da alle Kanten an dem Knoten markiert sind. Dieser Knoten kann nur unser Startknoten sein, da wir beim Durchwandern eines Knotens immer zwei Kanten streichen und immer keine oder eine gerade Anzahl von Kanten übrig bleibt. Das heißt, entweder kommen wir zu dem Knoten nicht mehr hin oder wenn wir hinkommen, können wir ihn auch wieder verlassen.

Wir haben also eine Rundwanderung gemacht, wobei wir unter Umständen noch nicht alle Kanten verwendet haben. Wir laufen daher unseren Weg noch einmal ab, bis wir auf einen Knoten kommen, an dem es eine noch nicht verwendete Kante gibt. Dort starten wir wieder eine Rundwanderung über noch ungenutzte Kanten, die uns zwangsläufig wieder zu diesem Knoten zurückführt. Die so gelaufene "Schleife" fügen wir zu unserem Weg hinzu.

Diesen Prozess setzen wir fort, bis es an unserem Weg keine unbenutzten Kanten mehr gibt.

Wir haben jetzt aber einen Eulerschen Weg gefunden, denn gäbe es noch irgendwo eine ungenutzte Kante, dann gäbe es ja einen Weg von dieser Kante zum Startknoten unseres Weges. Irgendwo würde dieser Weg auf unseren Rundwanderweg treffen. Dort gäbe es dann aber eine noch ungenutzte Brücke an unserem Weg.

## Das Brückenproblem - Zusammenfassung der Ergebnisse

Wir haben zwei Erkenntnisse gewonnen:

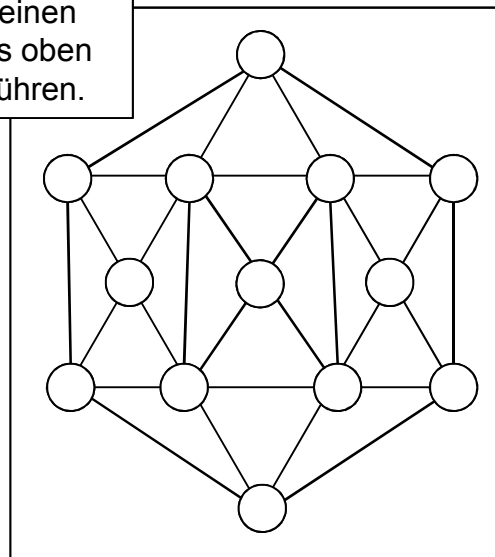
1. Einen Eulerschen Weg kann es in einem Graphen nur geben, wenn der Graph zusammenhängend ist und alle Knoten eine gerade Anzahl von Kanten haben.
2. Wenn ein Graph zusammenhängend ist und alle Knoten eine gerade Anzahl von Kanten haben, dann gibt es einen Eulerschen Weg.

Damit können wir den folgenden Satz formulieren:

**In einem Graphen gibt es genau dann einen Eulerschen Weg, wenn der Graph zusammenhängend ist und alle Knoten eine gerade Anzahl von Kanten haben.**

Leonard Euler hat mit diesem Satz 1736 den Grundstein für die Graphentheorie gelegt. Heute ist die Graphentheorie eine unerschöpfliche Quelle für Datenstrukturen und Algorithmen.

Finden Sie in diesem Graphen einen Eulerschen Weg, indem Sie das oben beschriebene Verfahren durchführen.

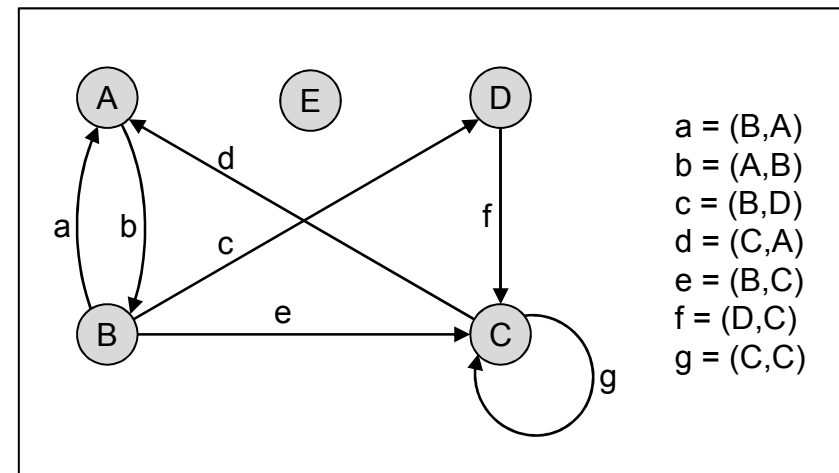


Unter einem **Graphen** verstehen wir eine Struktur, die aus endlich vielen **Knoten** und **Kanten** besteht. Einer Kante ist jeweils ein **Anfangsknoten** und ein **Endknoten** zugeordnet.

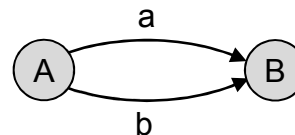
Typischerweise bezeichnen wir Knoten mit Großbuchstaben (A, B, C, ...) und Kanten mit Kleinbuchstaben (a, b, c, ...). Wenn eine Kante k den Anfangsknoten A und den Endknoten E hat, sagen wir, dass die Kante von A nach E führt und schreiben  $k = (A, E)$ . Es ist nicht ausgeschlossen, dass Anfangs- und Endknoten einer Kante gleich sind. Es ist auch nicht ausgeschlossen, dass es zu einem Knoten keine Kante gibt.

Wir visualisieren einen Graphen, indem wir die Knoten als Kreise und die Kanten als Pfeile von ihrem Anfangsknoten zu ihrem Endknoten zeichnen.

Was Knoten und Kanten konkret sind oder sein könnten (z.B. Landgebiete und Brücken), interessiert uns nicht. Diese Abstraktion ermöglicht die universelle Verwendbarkeit von Graphen für unterschiedlichste Aufgaben.



Grundsätzlich ist nicht ausgeschlossen, dass es in einem Graphen verschiedene Kanten mit gleichem Anfangs- und gleichem Endknoten (Parallelkanten) gibt.

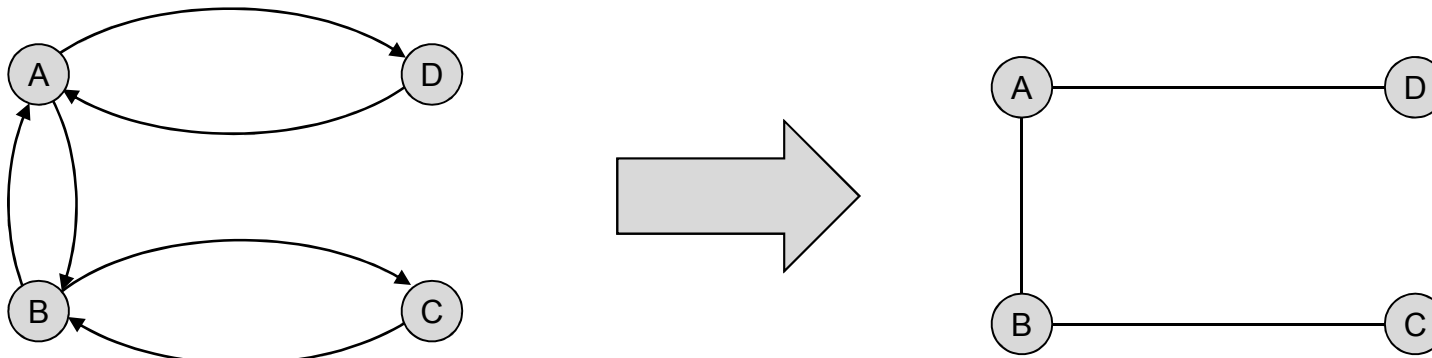


Wir wollen hier aber nur Graphen ohne Parallelkanten betrachten.

## Ungerichtete Graphen

Wenn in einem Graphen zu jeder Kante  $k = (A,B)$  auch die Kante  $k' = (B,A)$  vorkommt, so bezeichnen wir den Graph als **ungerichtet** oder **symmetrisch**.

Da in einem symmetrischen Graphen zu jeder Kante auch die in umgekehrter Richtung verlaufende Kante vorhanden ist, identifizieren wir die beiden Kanten miteinander und zeichnen für Kante und Umkehrkante jeweils nur eine Linie. Die Pfeile lassen wir in solchen Graphen weg:



## Die Adjazenzmatrix

Zur Speicherung eines Graphen in einem Programm dient häufig die sog. Adjazenzmatrix:

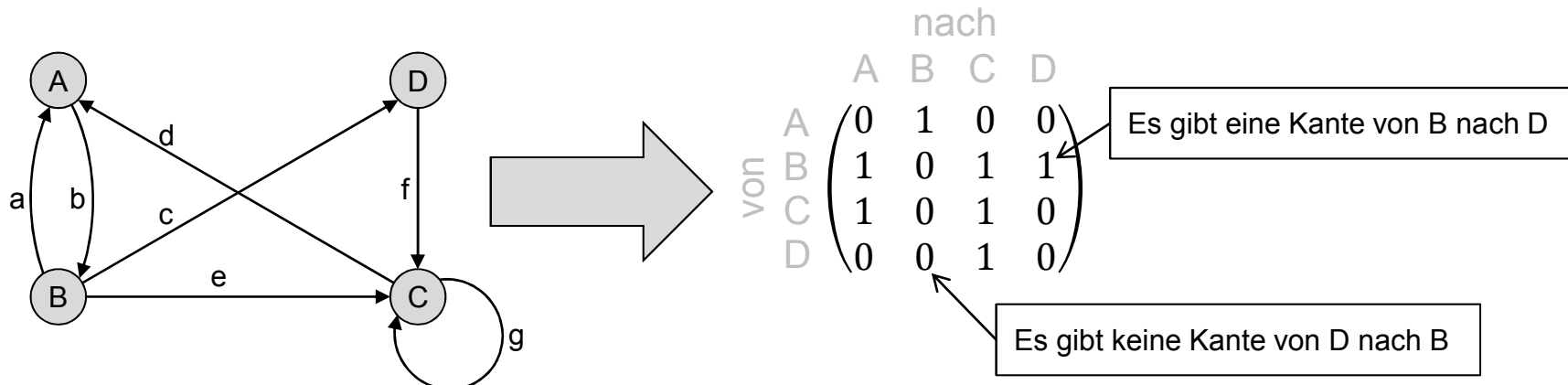
Gegeben sei ein Graph mit fortlaufend nummerierten Knoten  $(E_1, E_2, E_3, \dots, E_n)$ . Die Matrix

$$A = (a_{i,j}) = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$$

mit

$$a_{i,j} = \begin{cases} 1 & \text{Es gibt eine Kante von } E_i \text{ nach } E_j \\ 0 & \text{Es gibt keine Kante von } E_i \text{ nach } E_j \end{cases}$$

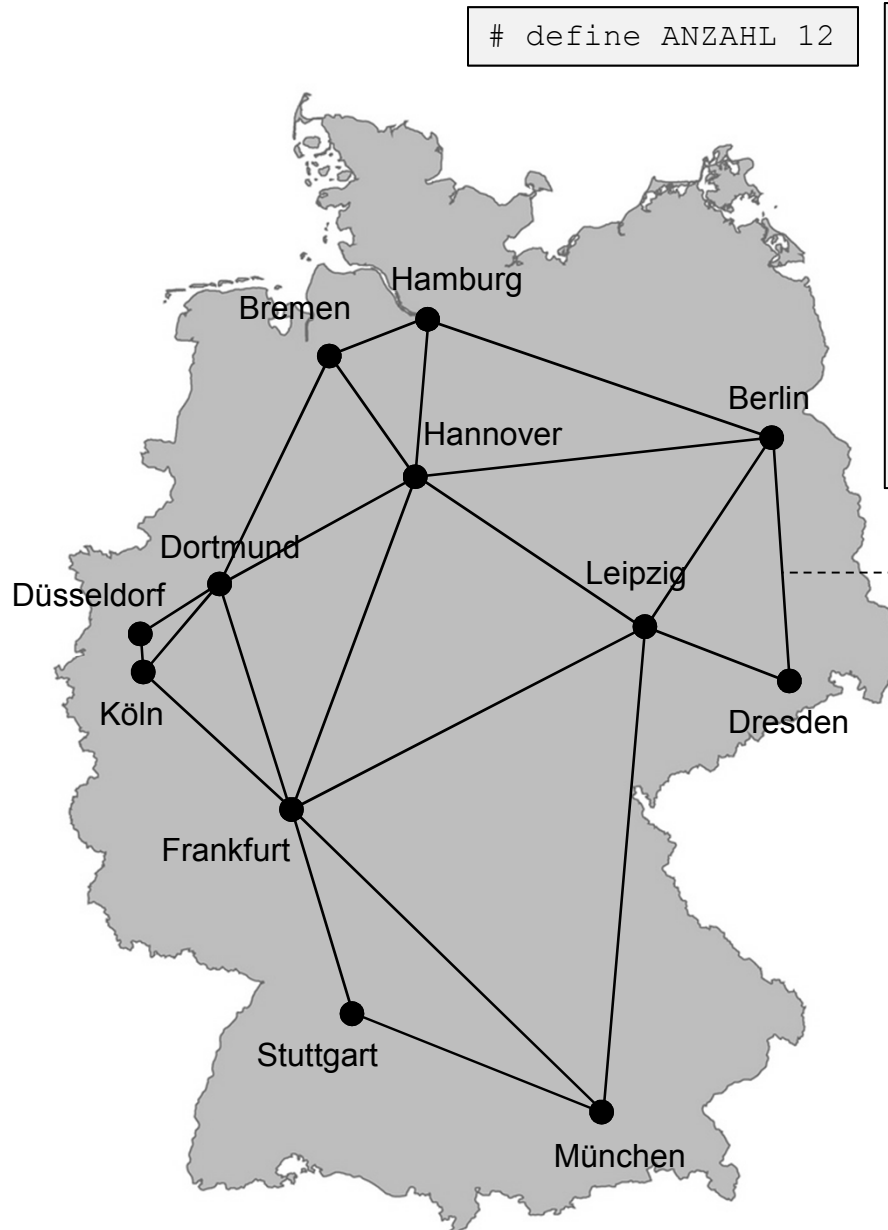
heißt die **Adjazenzmatrix** des Graphen.



Symmetrische Graphen haben eine symmetrische Adjazenzmatrix ( $a_{i,j} = a_{j,i}$ ). Das heißt, die Matrix ist spiegelsymmetrisch zur Hauptdiagonalen (von links oben nach rechts unten).



## Das Hauptbeispiel dieses Abschnitts



```
# define ANZAHL 12
```

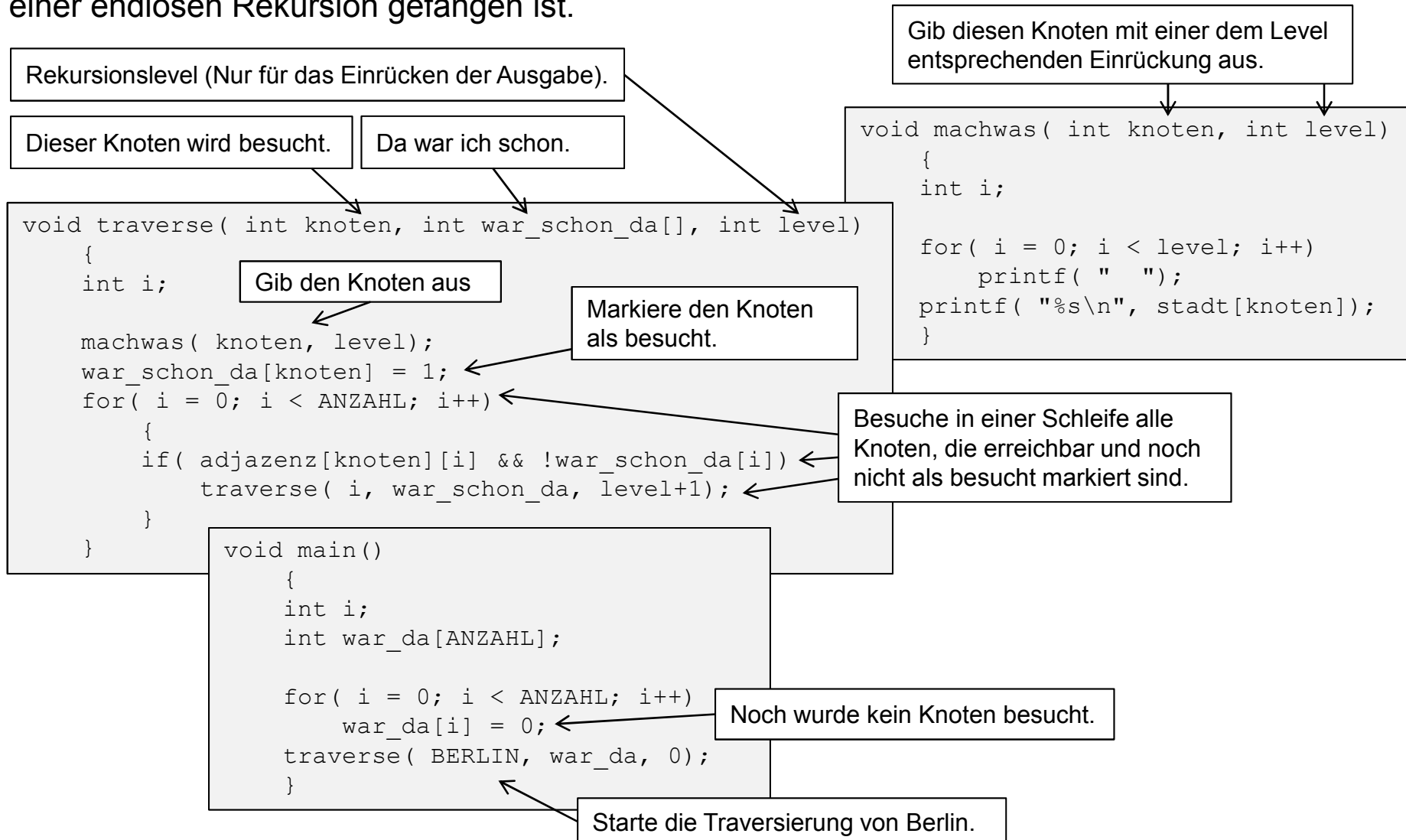
```
# define BERLIN      0
# define BREMEN      1
# define DORTMUND    2
# define DRESDEN     3
# define DUESSELDORF 4
# define FRANKFURT   5
# define HAMBURG     6
# define HANNOVER    7
# define KOELN       8
# define LEIPZIG     9
# define MUENCHEN    10
# define STUTTGART   11
```

```
char *stadt[ANZAHL] =
{
    "Berlin",
    "Bremen",
    "Dortmund",
    "Dresden",
    "Duesseldorf",
    "Frankfurt",
    "Hamburg",
    "Hannover",
    "Koeln",
    "Leipzig",
    "Muenchen",
    "Stuttgart"
};
```

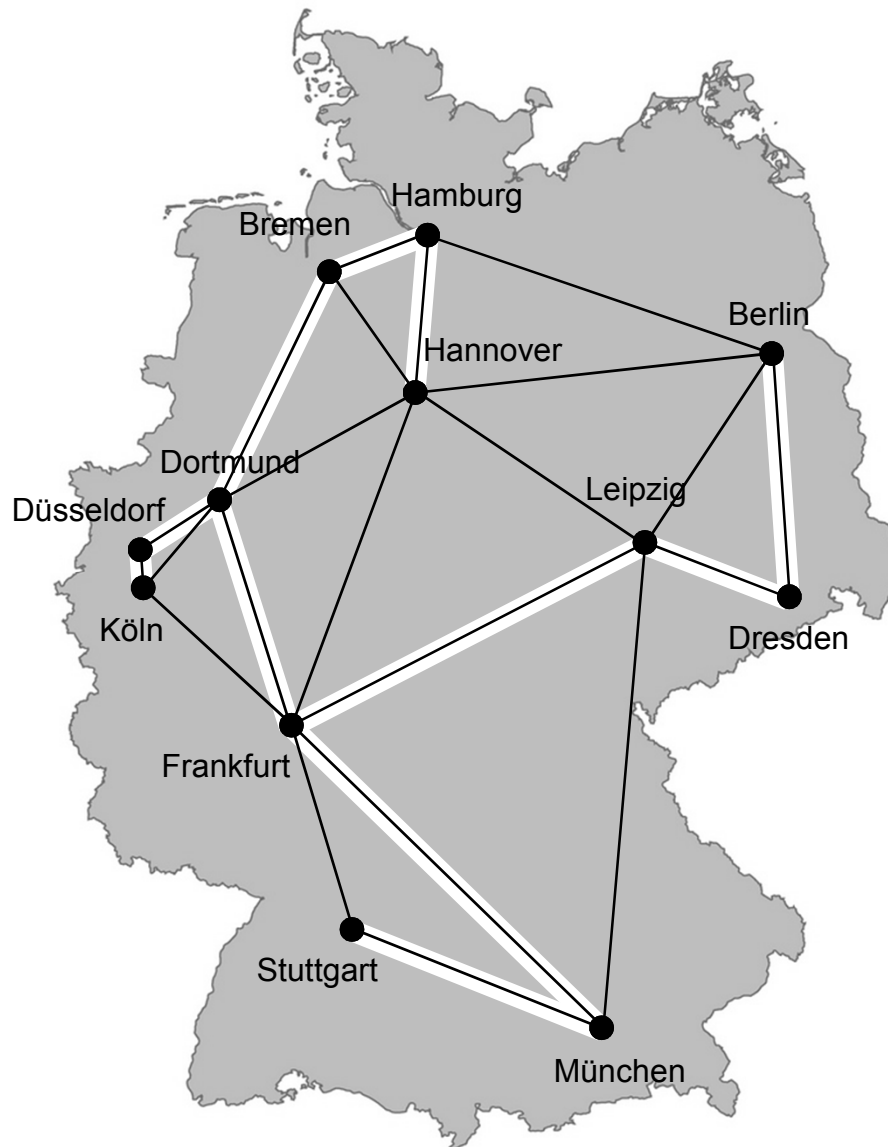
```
unsigned int adjazenz[ ANZAHL][ ANZAHL] =
{
    {0,0,0,1,0,0,1,1,0,1,0,0},
    {0,0,1,0,0,0,1,1,0,0,0,0},
    {0,1,0,0,1,1,0,1,1,0,0,0},
    {1,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,1,0,0,0,0,0,1,0,0,0},
    {0,0,1,0,0,0,0,1,1,1,1,1},
    {1,1,0,0,0,0,0,1,0,0,0,0},
    {1,1,1,0,0,1,1,0,0,1,0,0},
    {0,0,1,0,1,1,0,0,0,0,0,0},
    {1,0,0,1,0,1,0,1,0,0,1,0},
    {0,0,0,0,0,1,0,0,0,1,0,1},
    {0,0,0,0,0,1,0,0,0,0,1,0},
};
```

## Traversierung von Graphen

Einen Graphen kann man ähnlich einem Baum rekursiv traversieren. Man muss nur darauf achten, dass man Knoten, die man bereits besucht hat, nicht erneut besucht, weil man sonst in einer endlosen Rekursion gefangen ist.



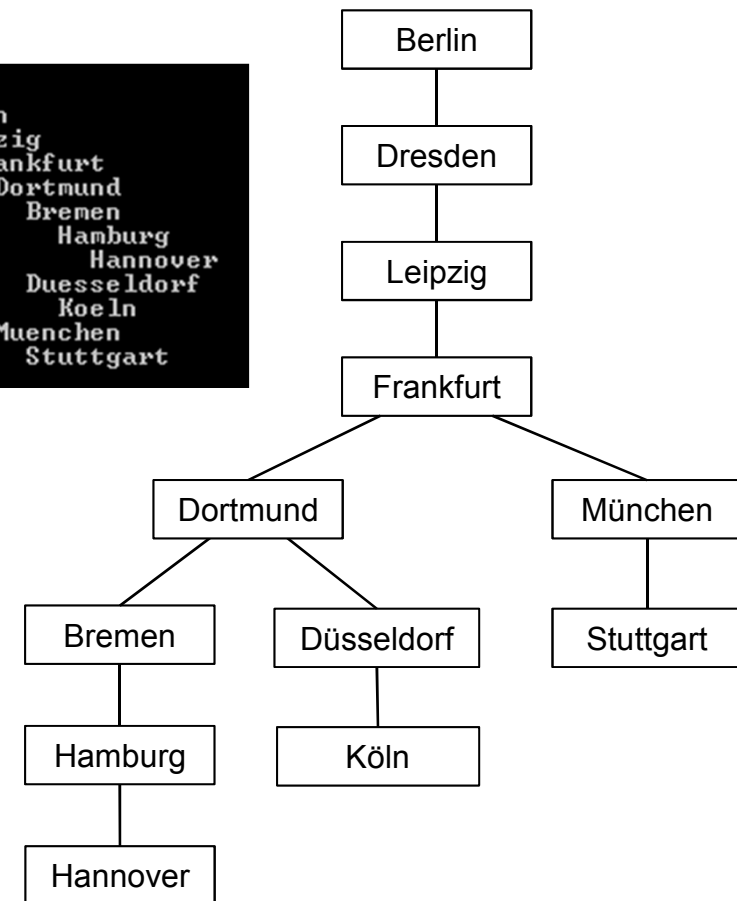
## Traversierung - Ergebnis



Der Algorithmus geht, in Berlin startend, immer zu der (alphabetisch) ersten Stadt, die direkt erreichbar ist und in der er noch nicht war. Gibt es keine solche Stadt mehr, erfolgt der Rücksprung auf die nächst höhere Aufrufebene. Auf diese Weise wird in dem Graphen ein Baum aller von Berlin aus erreichbaren Städte konstruiert.



Berlin  
Dresden  
Leipzig  
Frankfurt  
Dortmund  
Bremen  
Hamburg  
Hannover  
Düsseldorf  
Köln  
München  
Stuttgart



## Wege in Graphen

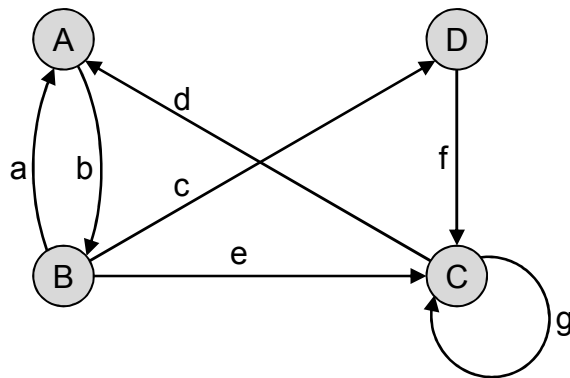
Eine endliche Folge  $A_1, A_2, \dots, A_n$  von Knoten eines Graphen heißt **Weg**, wenn je zwei aufeinander folgende Knoten durch eine Kante miteinander verbunden sind.

$A_1$  wird als der Anfangs-,  $A_n$  als der Endknoten des Weges bezeichnet und man spricht von einem **Weg von  $A_1$  nach  $A_n$** . Sind Anfangs- und Endknoten eines Weges gleich, so sprechen wir von einem **geschlossenen Weg** oder einer **Schleife**.

Ein Weg heißt **schleifenfrei**, wenn alle vorkommenden Knoten voneinander verschieden sind.

Ein Weg heißt **Kantenzug**, wenn alle im Weg vorkommenden Kanten voneinander verschieden sind. Ein geschlossener Kantenzug heißt **Kreis**. Ein Graph heißt **kreisfrei**, wenn er keine Kreise enthält.

Die Anzahl der Kanten in einem Weg wird auch als die **Länge** des Weges bezeichnet.



Die Folge (B, A, B, D, C) ist ein Weg der Länge 4.

Die Folge (B, D, C, A, B) ist ein geschlossener Weg.

Der Weg (A, B, D, C) ist schleifenfrei.

Der Weg (B, A, B, D) ist ein Kantenzug, aber nicht schleifenfrei.

Der Weg (A, B, A) ist ein Kreis.

## Die Wegematrix

Die Adjazenzmatrix eines Graphen liefert nur die Information, welche Knoten durch eine Kante, also durch einen Weg der Länge 1, miteinander verbunden sind. Wir wollen jetzt die allgemeinere Frage, welche Knoten durch einen beliebigen Weg miteinander verbindbar sind, beantworten. Dazu definieren wir die Wegematrix eines Graphen:

Gegeben sei ein Graph mit fortlaufend nummerierten Knoten  $(E_1, E_2, E_3, \dots, E_n)$ . Die Matrix

$$W = (w_{i,j}) = \begin{pmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,n} \end{pmatrix}$$

mit

$$w_{i,j} = \begin{cases} 1 & \text{Es gibt einen Weg von } E_i \text{ nach } E_j \\ 0 & \text{Es gibt keinen Weg von } E_i \text{ nach } E_j \end{cases}$$

heißt die **Wegematrix** des Graphen.

Die Wegematrix eines Graphen ist in der Regel nicht bekannt, sie muss aus der Adjazenzmatrix berechnet werden. Dazu verwenden wir das Verfahren von Warshall.

## Das Verfahren von Warshall – Verfahrensidee

Der betrachtete Graph hat die Knoten  $E_1, E_2, E_3, \dots, E_n$  und die Adjazenzmatrix  $A$ . Wir bilden eine Folge von Mengen, die anfänglich leer ist und nach und nach alle Knoten aufnimmt:

$$M_0 = \emptyset$$

$$M_1 = \{E_1\}$$

$$M_2 = \{E_1, E_2\}$$

$$\vdots$$

$$M_n = \{E_1, E_2, \dots, E_n\}$$

Dann berechnen wir eine Folge von Matrizen  $W_0, W_1, \dots, W_n$ , die wir aus der Adjazenzmatrix ableiten:

$$\begin{array}{ccccccccc} M_0 & & M_1 & & M_2 & & M_3 & & M_n \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ A = W_0 \rightarrow W_1 \rightarrow W_2 \rightarrow W_3 \cdots W_n \end{array}$$

Wir versuchen dabei die folgende Eigenschaft zu realisieren:

- (\*) Die Matrix  $W_k$  soll in der Zeile  $i$  und Spalte  $j$  genau dann den Wert 1 haben, wenn es einen Weg von  $E_i$  nach  $E_j$  gibt, dessen Zwischenpunkte sämtlich in  $M_k$  liegen.

Die Matrix  $W_0$  hat diese Eigenschaft, weil  $W_0$  die Adjazenzmatrix ist, die ja die Verbindungen ohne Zwischenpunkte enthält.

Wenn es jetzt gelingt, die Eigenschaft (\*) durch ein Konstruktionsverfahren (das wir noch nicht kennen) von Matrix zu Matrix ( $W_k \rightarrow W_{k+1}$ ) zu übertragen, haben wir am Ende in  $W_n$  die gesuchte Wegematrix, da die Eigenschaft (\*) für  $k = n$  die Wegematrix charakterisiert.

## Das Verfahren von Warshall – Verfahrensschritt

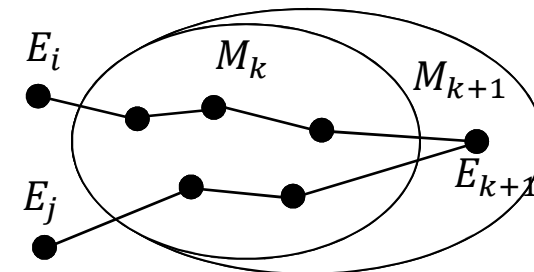
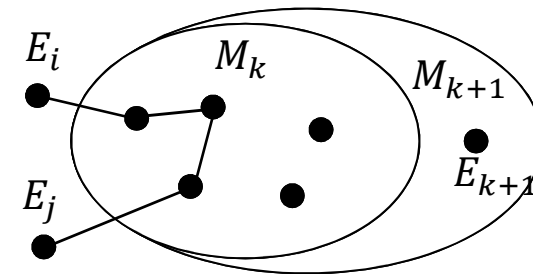
Wir gehen davon aus, dass wir Matrix  $W_k$  erfolgreich konstruiert haben und wollen jetzt die Matrix  $W_{k+1}$  konstruieren. Dazu bilden wir die Menge  $M_{k+1}$ , indem wir zur Menge  $M_k$  den Knoten  $E_{k+1}$  hinzunehmen. Wir betrachten jetzt zwei beliebige Knoten  $E_i$  und  $E_j$ .

Die Matrix  $W_k$  hat dann in der Zeile  $i$  und Spalte  $j$  genau dann den Wert 1, wenn es einen Weg von  $E_i$  nach  $E_j$  gibt, dessen Zwischenpunkte sämtlich in  $M_k$  liegen.

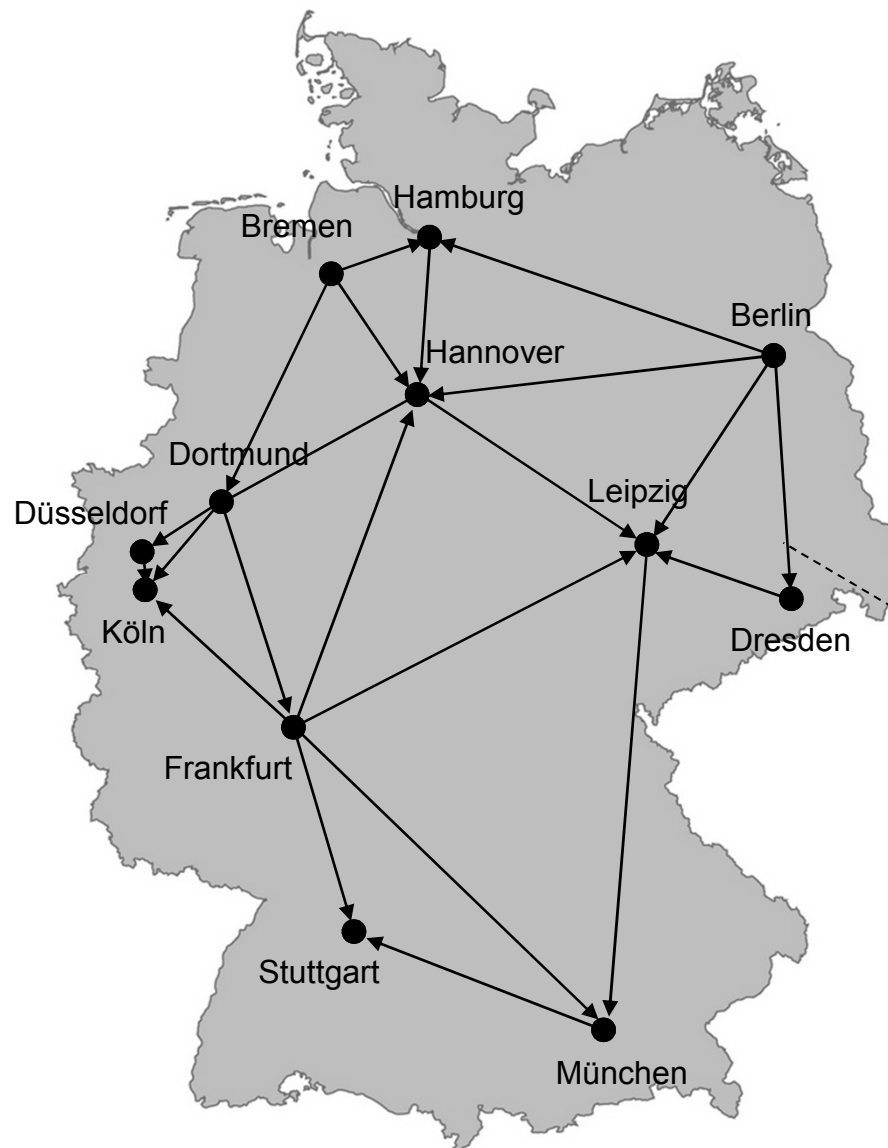
Wenn die beiden Knoten bereits durch einen Weg in  $M_k$  verbunden sind, dann steht in  $W_k$  in der entsprechenden Zeile und Spalte bereits eine 1 und diese 1 wird dann in  $W_{k+1}$  übernommen.

Wenn die Knoten in  $M_k$  noch nicht verbunden sind, können sie in  $M_{k+1}$  nur über den Zwischenpunkt  $E_{k+1}$  verbunden werden. Dazu muss es in  $M_k$  aber bereits Wege von  $E_i$  nach  $E_{k+1}$  und von  $E_{k+1}$  nach  $E_j$  geben. Das können wir in den entsprechenden Zeilen und Spalten der Matrix  $W_k$  überprüfen. Wenn beide Prüfungen positiv ausfallen, können wir  $E_i$  und  $E_j$  in  $W_{k+1}$  als verbindbar markieren.

Wenn wir dieses Verfahren für alle Knotenpaare  $E_i, E_j$  durchgeführt haben, hat  $W_{k+1}$  die gewünschte Eigenschaft und zeigt die Verbindbarkeit von Knoten über  $M_{k+1}$  an.



## Beispiel



Die Wegematrix im deutschen Autobahnnetz zu berechnen, ist wenig ergiebig, da die Matrix in allen Feldern 1 enthalten wird.

Wir machen daher die Autobahnen zu Einbahnstraßen. Jetzt ist nicht mehr jede Stadt von jeder anderen aus erreichbar. Es ergibt sich folgende Adjazenzmatrix, die ich bereits `weg` genannt habe, weil sie in die Wegematrix umgerechnet werden soll:

```
# define ANZAHL 12
```

```

unsigned int weg[ ANZAHL][ ANZAHL] =
{
    {0,0,0,1,0,0,1,1,0,1,0,0},
    {0,0,1,0,0,0,1,1,0,0,0,0},
    {0,0,0,0,1,1,0,1,1,0,0,0},
    {0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,0,0,0,1,1,1,1,1},
    {0,0,0,0,0,0,0,1,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,1,0},
    {0,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0},
} i;

```



## Das Verfahren von Warshall – Implementierung

Bei der Implementierung des Verfahrens arbeiten wir "in place". Das heißt, wir erzeugen nicht ständig neue Matrizen, sondern modifizieren die Adjazenzmatrix Schritt für Schritt, bis aus ihr die Wegematrix entstanden ist.

```
void warshall()
{
    int von, nach, zpkt;

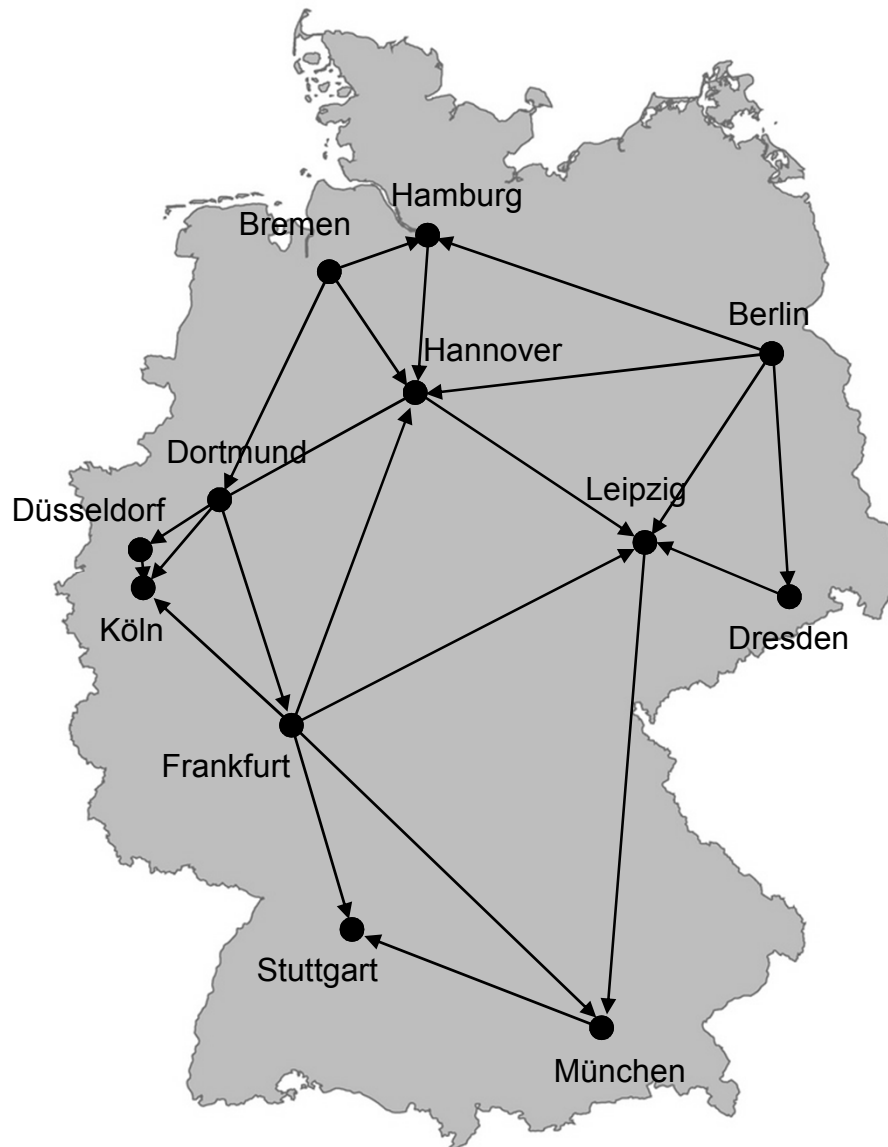
    for( zpkt = 0; zpkt < ANZAHL; zpkt++)
    {
        for( von = 0; von < ANZAHL; von++)
        {
            if( weg[von][zpkt])
            {
                for( nach = 0; nach < ANZAHL; nach++)
                {
                    if( weg[zpkt][nach])
                        weg[von][nach] = 1;
                }
            }
        }
    }
}
```

Dies ist der Zwischenpunkt, der jeweils neu zur Menge der Zwischenpunkte hinzugenommen wird. Die Schleife erzeugt also gedanklich die Mengenfolge  $M_1, M_2, \dots, M_n$

In dieser Doppelschleife werden alle Knotenpaare betrachtet und es wird untersucht, ob eine Verbindung über den Zwischenpunkt möglich ist.

Der Fall, einer Verbindbarkeit ohne Verwendung des Zwischenpunkts muss nicht geprüft werden, da diese Information bereits aus der vorherigen Iteration in der Matrix vorhanden ist und durch die "in place"-Strategie übernommen wird.

## Das Verfahren von Warshall - Ausgabe



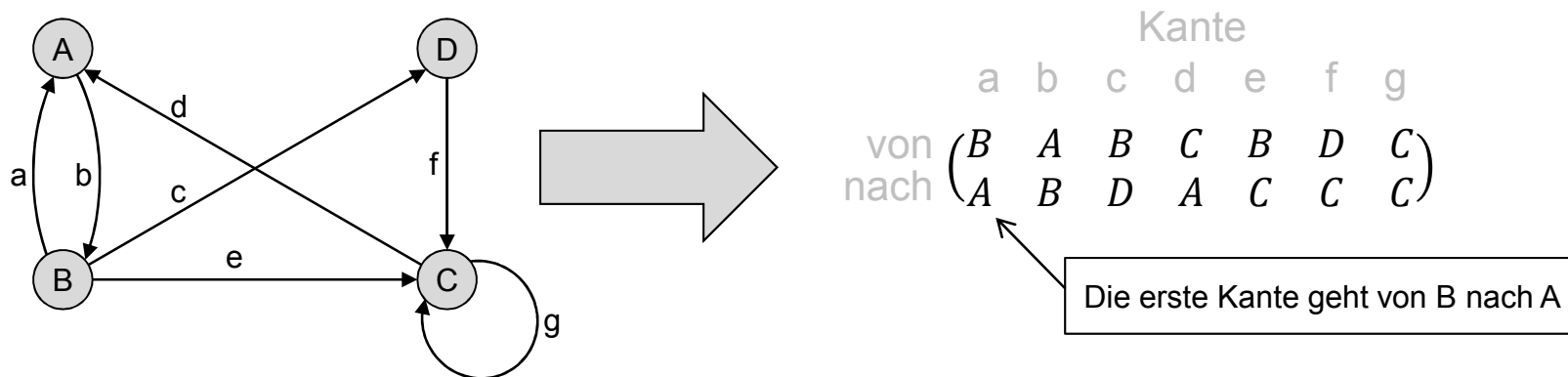
	Ber	Bre	Dor	Dre	Due	Fra	Ham	Han	Koe	Lei	Mue	Stu
Ber	0	0	0	1	0	0	1	1	0	1	1	1
Bre	0	0	1	0	1	1	1	1	1	1	1	1
Dor	0	0	0	0	1	1	0	1	1	1	1	1
Dre	0	0	0	0	0	0	0	0	0	1	1	1
Due	0	0	0	0	0	0	0	0	1	0	0	0
Fra	0	0	0	0	0	0	0	1	1	1	1	1
Ham	0	0	0	0	0	0	0	1	0	1	1	1
Han	0	0	0	0	0	0	0	0	0	1	1	1
Koe	0	0	0	0	0	0	0	0	0	0	0	0
Lei	0	0	0	0	0	0	0	0	0	0	1	1
Mue	0	0	0	0	0	0	0	0	0	0	0	1
Stu	0	0	0	0	0	0	0	0	0	0	0	0

Die Ergebnismatrix zeigt, von welcher Stadt aus welche Städte erreichbar sind. Das Erreichbarkeitsproblem ist damit vollständig gelöst. Die Matrix zeigt allerdings nicht, welchen Weg man im Falle der Erreichbarkeit einschlagen sollte. Mit dieser Frage werden wir uns später beschäftigen.

## Kantentabellen

Eine Adjazenzmatrix ist eine sinnvolle Repräsentation für einen Graphen, wenn man eine knotenorientierte Verarbeitung des Graphen plant. Die bisher betrachteten Algorithmen waren knotenorientiert. Manchmal ist es aber sinnvoll, in einem Algorithmus kantenorientiert vorzugehen. Das heißt, man möchte der Reihe nach alle Kanten eines Graphen betrachten, um gewisse Berechnungen durchzuführen.

In dieser Situation bietet es sich an, eine **Kantentabelle** statt einer Adjazenzmatrix zu verwenden. Eine Kantentabelle ist ein Array (oder eine Liste) in der alle Kanten des Graphen mit Anfangs- und Endpunkt aufgeführt sind.



Ein Graph mit  $n$  Knoten kann  $n^2$  Kanten haben, wenn alle Knoten paarweise miteinander verbunden sind. In der Regel werden es aber deutlich weniger Kanten sein. Verwendet man bei einem kantenorientierten Verfahren eine Adjazenzmatrix, so muss man alle  $n^2$  Knotenpaare betrachten und wird quadratische Laufzeit haben. Bei Verwendung einer Kantentabelle kann man die Laufzeit reduzieren, wenn es relativ wenig Kanten im Vergleich zum Quadrat der Knotenzahl gibt.

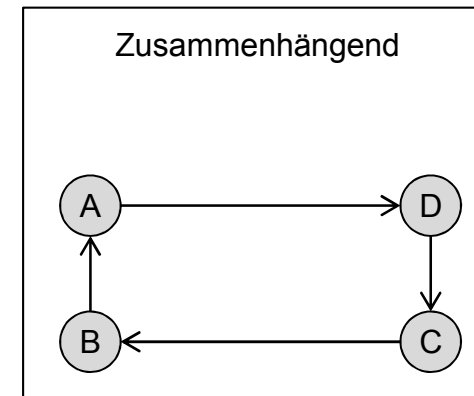
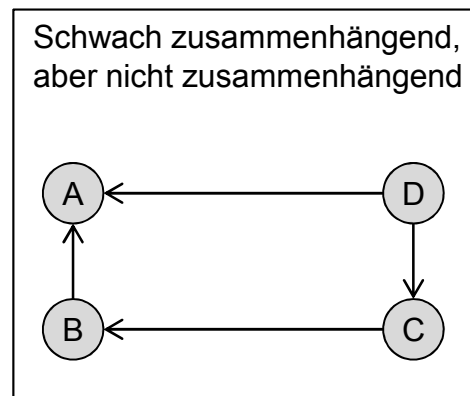
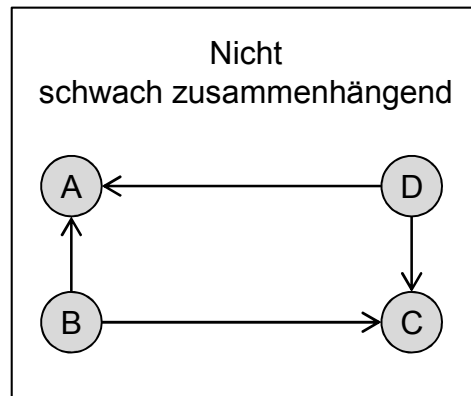
## Zusammenhang

Zur Anwendung von Kantentabellen wollen wir uns mit den sog. Zusammenhangskomponenten eines symmetrischen Graphen beschäftigen. Dazu führen wir zunächst den Begriff des Zusammenhangs in beliebigen Graphen ein:

Ein Graph heißt **schwach zusammenhängend**, wenn es für je zwei Knoten X und Y einen Weg von X nach Y oder einen Weg von Y nach X gibt.

Ein Graph heißt **stark zusammenhängend** oder einfach **zusammenhängend**, wenn es für je zwei Knoten X und Y einen Weg von X nach Y gibt.

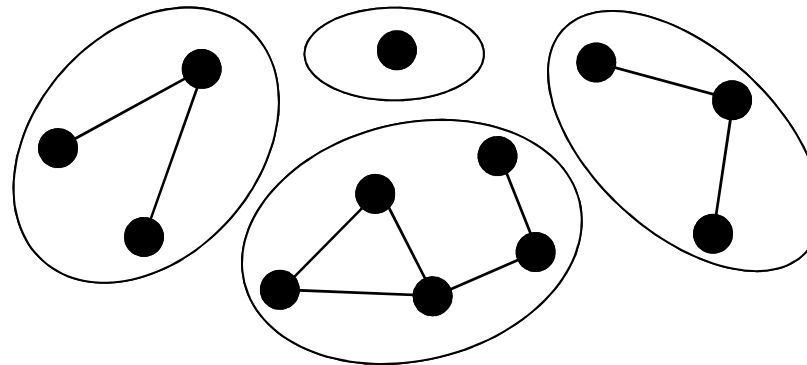
Ein zusammenhängender Graph ist immer schwach zusammenhängend. In symmetrischen Graphen fallen die beiden Begriffe zusammen.



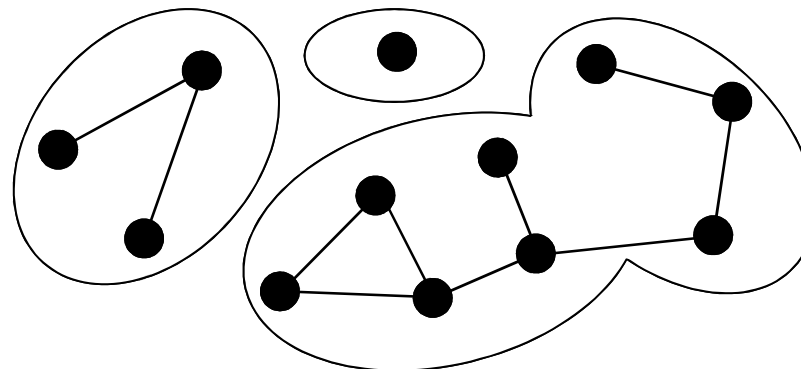
Ein ungerichteter, zusammenhängender kreisfreier Graph wird auch als **Baum** bezeichnet.

## Zusammenhangskomponenten

In einem ungerichteten Graphen ergeben sich immer "Cluster" von paarweise untereinander zusammenhängenden Knoten. Diese Cluster heißen **Zusammenhangskomponenten**. Im folgenden Beispiel gibt es vier Zusammenhangskomponenten:



Die Zusammenhangskomponenten bilden immer eine disjunkte Zerlegung der Knotenmenge. Das bedeutet, dass jeder Knoten genau einer Zusammenhangskomponente zugeordnet ist. Würde man in obigem Beispiel eine zusätzliche Kante von einem Knoten eines Clusters zu einem Knoten eines anderen Clusters ziehen, würden die beiden Cluster sofort verschmelzen.



Ist der Graph zusammenhängend, dann gibt es nur eine Zusammenhangskomponente.

## Äquivalenzrelationen

Die Cluster bilden sich, weil die Verbindungsbeziehung in symmetrischen Graphen die folgenden drei Eigenschaften hat:

1. Jeder Knoten ist mit sich selbst verbindbar
2. Wenn A mit B verbindbar ist, dann ist auch B mit A verbindbar
3. Wenn A mit B und B mit C verbindbar ist, dann ist auch A mit C verbindbar

Diese drei Eigenschaften heißen **Reflexivität**, **Symmetrie** und **Transitivität**. Eine Beziehung, die diese drei Eigenschaften hat, nennt sich **Äquivalenzrelation**.

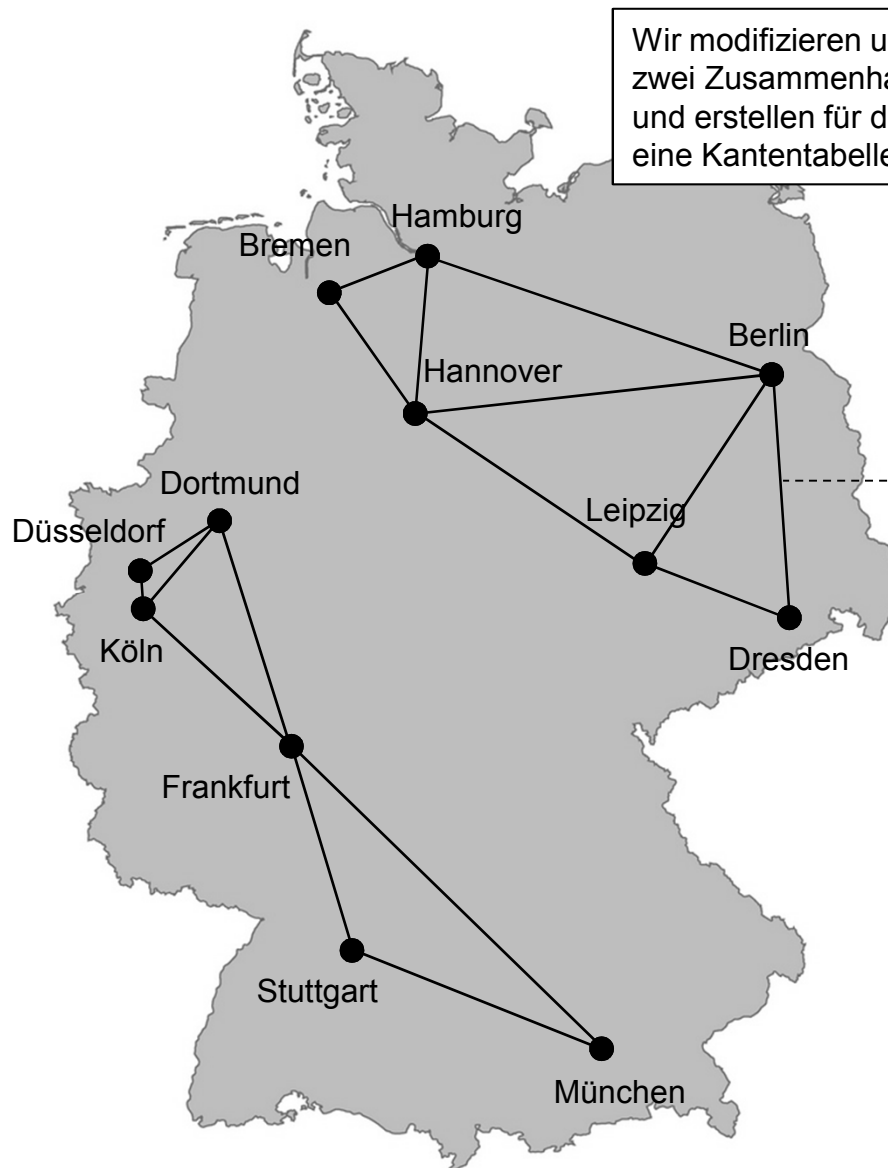
Äquivalenzrelationen haben immer die Eigenschaft, die Grundmenge vollständig in paarweise elementfremde Teilmengen (sog. **Äquivalenzklassen**) zu zerlegen.

Äquivalenzrelationen sind eine ganz wesentliche Grundlage unseres Denkens. Immer wenn wir abstrahieren, verwenden wir (bewusst oder unbewusst) eine Äquivalenzrelation.

Betrachten Sie zum Beispiel die Menge aller Autos und auf dieser Menge die Relation "vom gleichen Hersteller sein". Diese Relation ist eine Äquivalenzrelation (Bedingungen 1-3 sind erfüllt) und zerlegt die Menge der Autos in elementfremde Klassen von Autos, die jeweils vom gleichen Hersteller kommen. Diese Klassen heißen dann Audi, BMW, Mercedes, oder VW. In diesem Sinne bilden Äquivalenzrelationen auch das Fundament der objektorientierten Programmierung (siehe später).

Die Zusammenhangskomponenten sind die Äquivalenzklassen bezüglich der Äquivalenzrelation "durch einen Weg verbindbar" über der Knotenmenge eines Graphen. Wir wollen einen Algorithmus entwickeln, der die Zusammenhangskomponenten für einen Graphen berechnet und folgen dabei der Idee von der Verschmelzung der Cluster.

## Zusammenhangskomponenten - Beispiel



Wir modifizieren unser Standardbeispiel, damit zwei Zusammenhangskomponenten entstehen und erstellen für den modifizierten Graphen eine Kantentabelle

```
# define ANZ_KNOTEN 12
# define ANZ_KANTEN 17
```

```
struct kante
{
    int von;
    int nach;
};
```

```
struct kante kantentabelle[ANZ_KANTEN] =
{
```

```
{0,3},
{1,6},
{0,7},
{1,7},
{6,7},
{2,8},
{4,8},
{5,8},
{0,9},
{3,9},
{2,4},
{2,5},
{0,6},
{7,9},
{5,10},
{5,11},
{10,11}
```

```
};
```

```
# define BERLIN      0
# define BREMEN      1
# define DORTMUND    2
# define DRESDEN     3
# define DUESSELDORF 4
# define FRANKFURT   5
# define HAMBURG     6
# define HANNOVER    7
# define KOELN       8
# define LEIPZIG     9
# define MUENCHEN    10
# define STUTTGART   11
```

## **Zusammenhangskomponenten – Lösungsidee**

Um die Zusammenhangskomponenten zu bestimmen, bilden wir Mengen von Knoten.

Am Anfang liegt jeder Knoten für sich allein in einer eigenen Menge.

Dann betrachten wir der Reihe nach alle Kanten des Graphen. Wenn Anfangs und Endpunkt der Kante bereits in der gleichen Menge liegen, ist nichts zu tun. Wenn aber der Anfangs und der Endpunkt in verschiedenen Mengen liegen, müssen die beiden Mengen vereinigt werden.

Die Mengen, die dann nach Betrachtung aller Kanten noch übrig sind, sind die Zusammenhangskomponenten.

Es bleibt die Frage: Wie kann man möglichst einfach eine Datenstruktur für eine Menge von Zahlen (Knotenindices) aus einem festen Bereich implementieren, die die folgenden Operationen unterstützt:

- Einfügen eines Elements (Knotenindex) in eine Menge
- Vereinigen von zwei Mengen

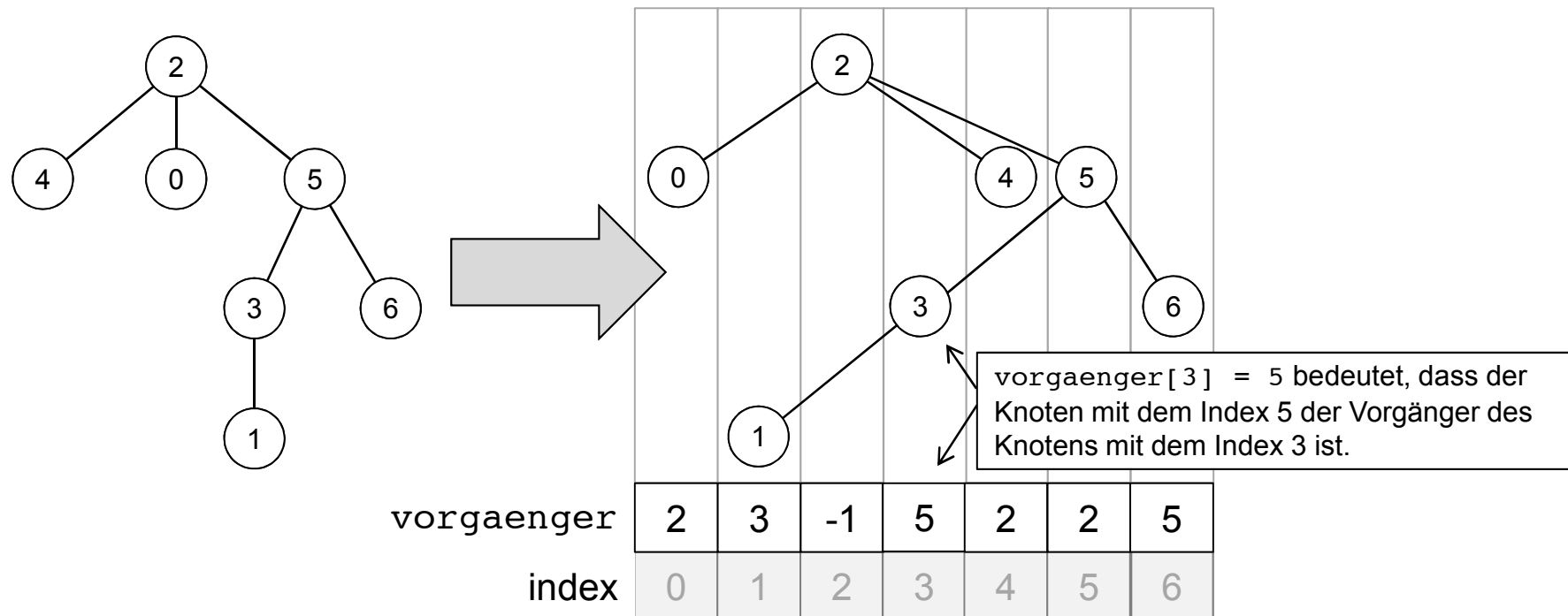


## Zusammenhangskomponenten - Datenstruktur

Die benötigten Mengen werden als logische Baumstruktur in einem Array gespeichert.

```
int vorgaenger[ANZ_KNOTEN];
```

Im Array speichern wir für jeden Knoten den Index seines Vorgängers (Rückverweis) im Baum:



Im Array können mehrere elementfremde Bäume liegen. Die Wurzel eines Baums erkennt man am Index -1. Im Grunde genommen interessiert der genaue Aufbau des Baumes nicht. Wichtig ist nur, dass jeder Baum im Array eine Menge beschreibt. Alles, was im selben Baum ist, ist in derselben Menge.

## Zusammenhangskomponenten – Mengenoperationen

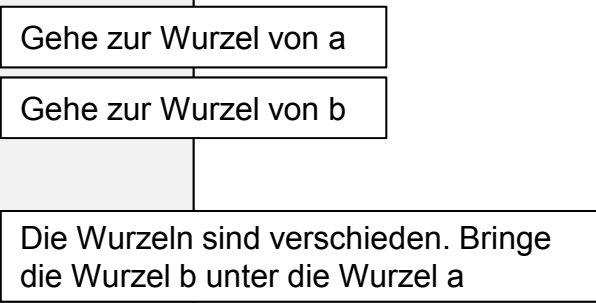
Anfänglich liegt jeder Knoten allein in einer Menge. Jeder Knoten ist also die Wurzel in einem ansonsten leeren Baum. Um dies zu erreichen, muss man alle Werte im Rückverweisarray (`vorgaenger`) auf `-1` setzen:

```
void init()
{
    int i;

    for( i= 0; i < ANZ_KNOTEN; i++)
        vorgaenger[i] = -1;
}
```

Die Funktion `join` zur Vereinigung von zwei Mengen erhält zwei Knotenindices und geht im Baum zu den zu diesen Knoten gehörenden Wurzeln. Sind die Wurzeln gleich, so sind die beiden Knoten bereits im selben Baum. Sind die Wurzeln verschieden, so werden die beiden Mengen vereinigt, indem man nur Wurzel der einen Menge (egal welche von beiden) unter die Wurzel der anderen bringen:

```
void join( int a, int b)
{
    while( vorgaenger[a] != -1)
        a = vorgaenger[a];
    while( vorgaenger[b] != -1)
        b = vorgaenger[b];
    if( a != b)
        vorgaenger[b] = a;
}
```



Nach dem Aufruf dieser Funktion sind die Menge, die den Knoten `a` enthält und die Menge, die den Knoten `b` enthält miteinander verschmolzen.

## Zusammenhangskomponenten – Berechnung der Komponenten

Um die Zusammenhangskomponenten zu berechnen, wird nach der Initialisierung über die Kanten der Kantentabelle iteriert. Für jede Kante wird die Menge in der der Anfangspunkt liegt, mit der Menge, in der der Endpunkt liegt, verschmolzen:

```
void bilde_komponenten()
{
    int k;

    init();
    for( k = 0; k < ANZ_KANTEN; k++)
        join( kantentabelle[k].von, kantentabelle[k].nach);
}
```

Nach Aufruf dieser Funktion liegen die Zusammenhangskomponenten im Vorgänger-Array vor. Sie müssen nur noch ausgegeben werden.

## Zusammenhangskomponenten – Hauptprogramm und Ausgabefunktion

Die hier gezeigten Funktionen haben mit der eigentlichen Aufgabe nichts zu tun. Sie dienen nur zur Ausgabe des Ergebnisses:

```
void main()
```

```
{  
    bilde_komponenten();  
    ausgabe();  
}
```

Das Hauptprogramm berechnet die Komponenten und gibt sie auf dem Bildschirm aus.

```
void ausgabe()
```

```
{  
    int i, k, z;  
    for( i = 0, z = 0; i < ANZ_KNOTEN; i++)  
    {  
        if( vorgaenger[i] == -1)  
        {  
            printf( "%d-te Zusammenhangskomponente:\n", ++z);  
            for( k = 0; k < ANZ_KNOTEN; k++)  
            {  
                if( wurzel( k) == i)  
                    printf( "    %2d %s\n", k, stadt[k]);  
            }  
            printf( "\n");  
        }  
    }  
}
```

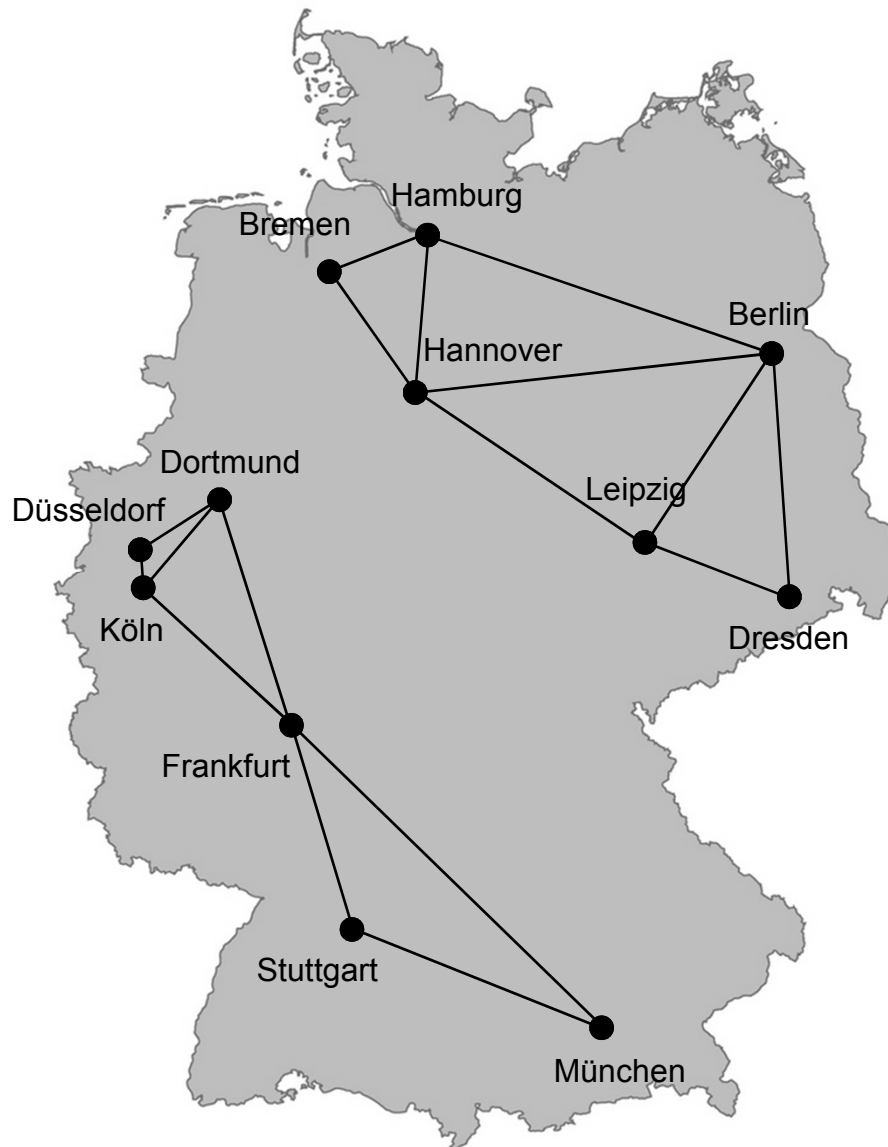
```
int wurzel( int a)
```

```
{  
    while( vorgaenger[a] != -1)  
        a = vorgaenger[a];  
    return a;  
}
```

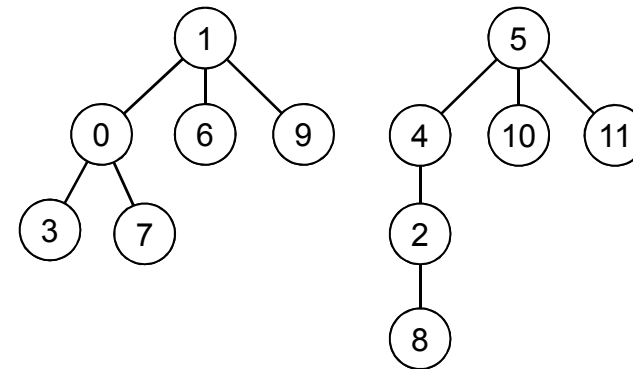
Diese Hilfsfunktion ermittelt zu einem Knoten den Index seiner Wurzel.

In der Ausgabefunktion werden alle Knoten gesucht, die Wurzel eines Baumes sind. Jeder dieser Knoten repräsentiert eine Zusammenhangskomponente. In der inneren Schleife werden dann alle Knoten gesucht, die den in der äußeren Schleife gefundenen Knoten als Wurzel haben und ausgegeben.

## Zusammenhangskomponenten – Ausgabe



Angesetzt auf den Graphen mit dem reduzierten deutschen Autobahnnetz ergeben sich zwei Mengen:



Diese werden ausgegeben:

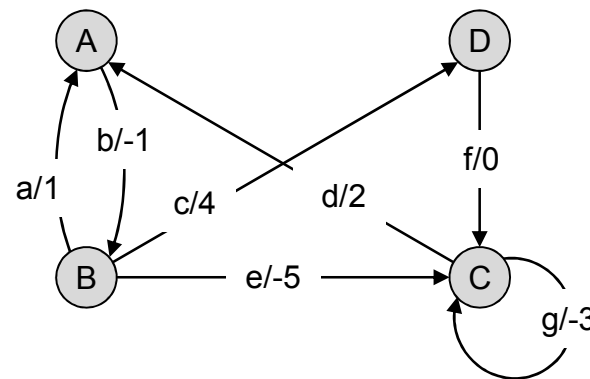
```
1-te Zusammenhangskomponente:  
0 Berlin  
1 Bremen  
3 Dresden  
6 Hamburg  
7 Hannover  
9 Leipzig  
  
2-te Zusammenhangskomponente:  
2 Dortmund  
4 Duesseldorf  
5 Frankfurt  
8 Koeln  
10 Muenchen  
11 Stuttgart
```

## Gewichtete oder bewertete Graphen

Wenn in einem Graphen jeder Kante ein Zahlenwert zugeordnet ist, so sprechen wir von einem **gewichteten** oder **bewerteten Graphen**. Den Zahlenwert einer Kante bezeichnen wir als das **Kantengewicht**.

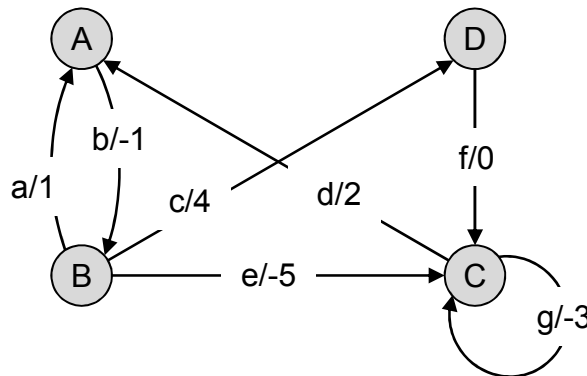
Kantengewichte können in konkreten Beispielen unter anderem Entfernungskilometer, Reise- oder Produktionskosten, Reise- oder Produktionszeiten, Gewinne oder Verluste beziehungsweise Leitungs- oder Transportkapazitäten bedeuten.

In der Darstellung schreiben wir die Kantengewichte zusätzlich an die einzelnen Kanten:



## Gewichtete Wege in Graphen

In einem gewichteten Graphen wird die Summe der Kantengewichte aller Kanten eines Weges als das **Gewicht** oder die **Bewertung des Weges** bezeichnet.



Der Weg (a, b, a, b, c) hat das Gewicht 4.  
Der Weg (a, b, c, f, d, b) hat das Gewicht 5.  
Der Weg (f, g, g, d) hat das Gewicht -4.

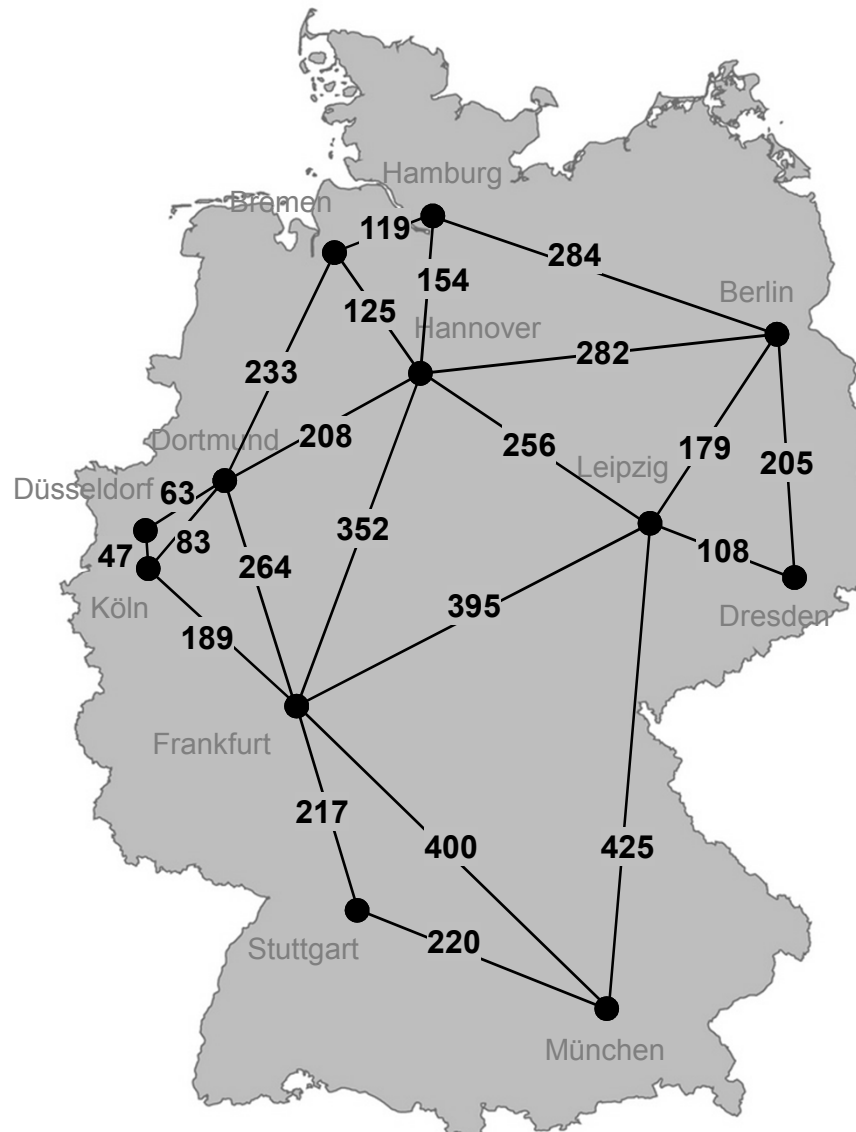
Je nach Bedeutung (Entfernung/Dauer/Preis) der Kantengewichte können wir uns dann zum Beispiel fragen:

**Was ist der kürzeste/schnellste/kostengünstigste Weg, also der Weg mit dem niedrigsten Gewicht, von einem Knoten zu einem anderen?**

Auf diese Frage gibt es nur eine Antwort, wenn es keine negativ bewerteten Schleifen in einem Graph gibt.

Wir wollen im Folgenden nur Graphen mit nicht negativen Kantengewichten betrachten. Dann kann es keine negativ bewerteten Schleifen geben und wir sind sicher, dass es immer Wege mit minimalem Gewicht gibt, sofern es überhaupt Wege gibt. Ausgangspunkt der folgenden Betrachtungen ist eine "Adjazenzmatrix", in die wir, statt 0 oder 1 für die Existenz einer Kante, das Kantengewicht eintragen. In unserem Beispiel (Autobahnnetz) sprechen wir dann auch von einer **Distanzenmatrix**.

## Distanzenmatrix - Beispiel



```
# define ANZAHL 12
# define xxx 10000
```

```
unsigned int distanz[ANZAHL][ANZAHL] =
{
    { 0,xxx,xxx,205,xxx,xxx,284,282,xxx,179,xxx,xxx},
    {xxx, 0,233,xxx,xxx,xxx,119,125,xxx,xxx,xxx,xxx},
    {xxx,233, 0,xxx, 63,264,xxx,208, 83,xxx,xxx,xxx},
    {205,xxx,xxx, 0,xxx,xxx,xxx,xxx,xxx,108,xxx,xxx},
    {xxx,xxx, 63,xxx, 0,xxx,xxx,xxx, 47,xxx,xxx,xxx},
    {xxx,xxx,264,xxx,xxx, 0,xxx,352,189,395,400,217},
    {284,119,xxx,xxx,xxx,xxx, 0,154,xxx,xxx,xxx,xxx},
    {282,125,208,xxx,xxx,352,154, 0,xxx,256,xxx,xxx},
    {xxx,xxx, 83,xxx, 47,189,xxx,xxx, 0,xxx,xxx,xxx},
    {179,xxx,xxx,108,xxx,395,xxx,256,xxx, 0,425,xxx},
    {xxx,xxx,xxx,xxx,xxx,400,xxx,xxx,xxx,425, 0,220},
    {xxx,xxx,xxx,xxx,xxx,217,xxx,xxx,xxx,xxx,220, 0},
};
```

In der Distanzenmatrix stehen die Entfernungen zwischen Städten, die durch eine Kante verbunden sind. Bei Städten, die nicht durch eine Kante verbunden sind, steht dort ein "großer" Wert (xxx, 10000), der erkennbar keine gültige Entfernungsangabe darstellt.

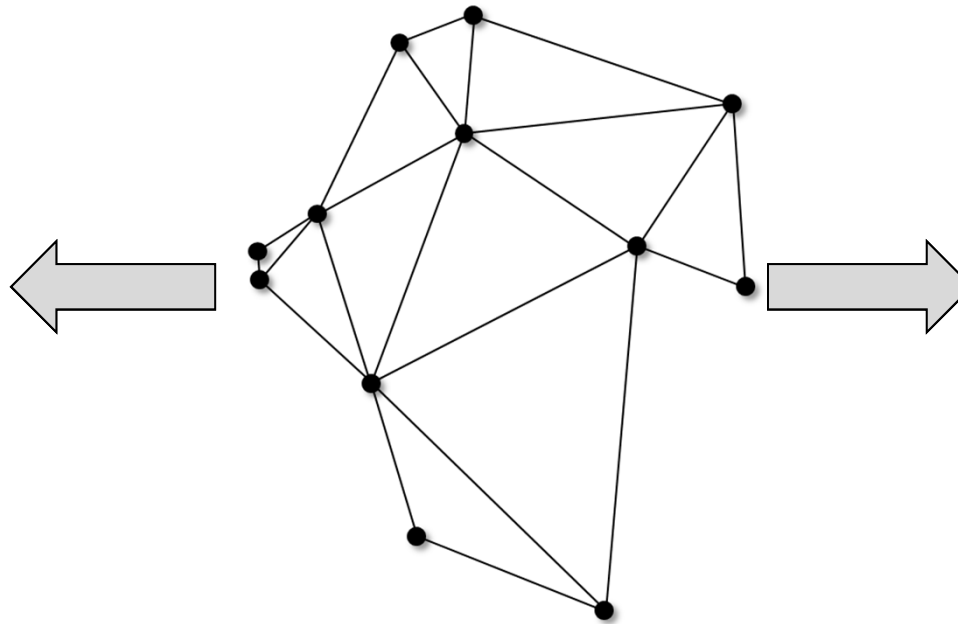
Mit dieser Matrix als Ausgangspunkt kann man versuchen, kürzeste Wege zu finden.

Die Distanzenmatrix muss im Allgemeinen nicht symmetrisch sein. Theoretisch könnte der Weg von Berlin nach Dresden weiter sein als der Weg von Dresden nach Berlin. Die im Folgenden betrachteten Algorithmen liefern auch für nicht symmetrische Graphen korrekte Ergebnisse.



## Kürzeste Wege

In einem analogen Modell ist das Problem, einen kürzesten Weg zu finden, einfach zu lösen. Man baut den Graphen als Drahtmodell, wobei die Länge der Drähte dem Kantengewicht entspricht, fasst an den beiden Knoten an und zieht sie so weit es geht auseinander.



Die Folge der unter Spannung stehenden Drähte bildet dann den gesuchten Weg.

In einem digitalen Modell, etwa unter Verwendung der Distanzenmatrix, wird dieser Weg nicht so einfach zu finden sein.

## Kürzeste Wege – Aufgabenstellungen

Gegeben sei ein Graph mit nicht negativen Kantengewichten. Die Kantengewichte werden dabei als Entfernungen interpretiert. Dann gibt es drei verschiedene Aufgabenstellungen mit offensichtlich wachsendem Lösungsaufwand:

1. Finde den kürzesten Weg von einem Knoten A zu einem Knoten B
2. Finde die kürzesten Wege von einem Knoten A zu allen anderen Knoten des Graphen
3. Finde die kürzesten Wege zwischen allen Knoten des Graphen

Wenn man die Aufgabe 1 für zwei Knoten A und B lösen will, so muss man den kürzesten Weg von A zu jedem Knoten C ermitteln, um zu entscheiden zu können, ob es sich lohnt den Weg über diesen Knoten zu nehmen. Dies bedeutet, dass man die Aufgabe 1 nicht lösen kann, ohne zugleich die Aufgabe 2 zu lösen. Wir haben es also de facto nur mit zwei Aufgaben zu tun

Aufgabe I: Finde die kürzesten Wege zwischen allen Knoten des Graphen

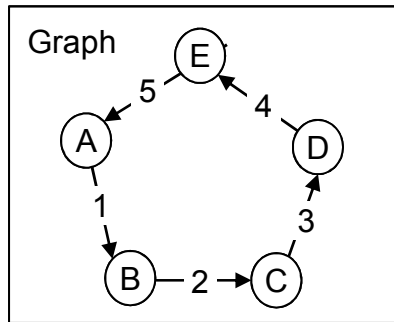
Aufgabe II: Finde die kürzesten Wege von einem Knoten A zu allen anderen Knoten des Graphen

Wir werden im Folgenden drei Algorithmen betrachten:

1. Algorithmus von Floyd (Aufgabe I)
2. Algorithmus von Dijkstra (Aufgabe II)
3. Algorithmus von Ford (Aufgabe II)

## Der Algorithmus von Floyd – Datenstruktur

Gesucht ist eine Datenstruktur zur Speicherung kürzester Wege.



Distanzenmatrix

A	0	1	—	—	—
B	—	0	2	—	—
C	—	—	0	3	—
D	—	—	—	0	4
E	5	—	—	—	0
	A	B	C	D	E

Algorithmus von Floyd

Distanzmatrix

A	0	1	3	6	10
B	14	0	2	5	9
C	12	13	0	3	7
D	9	10	12	0	4
E	5	6	8	11	0
	A	B	C	D	E

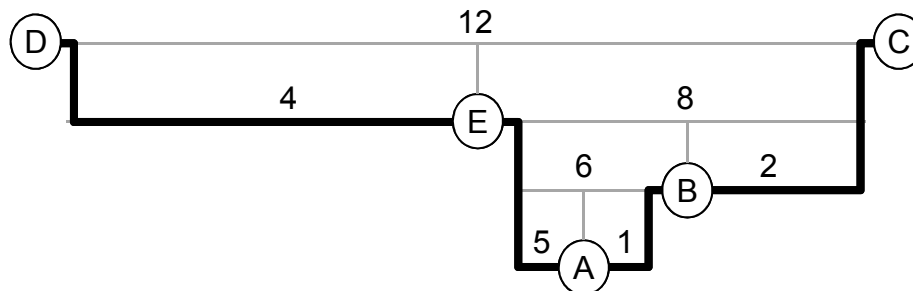
Die Distanz von D nach C beträgt 12 Einheiten.

Da alle Teilstrecken optimaler Wege ihrerseits optimal sind, reicht es aus, für je zwei Knoten X und Y einen Zwischenpunkt Z in einer Zwischenpunktmatrix zu speichern. Die weiteren Zwischenpunkte findet man dann, indem man in der Matrix Zwischenpunkte zu X und Z bzw. Z und Y sucht und dieses Verfahren (rekursiv) fortsetzt, bis keine Zwischenpunkte mehr gefunden werden. Im folgenden Beispiel wird der kürzeste Weg von D nach C aus der rechts stehenden Zwischenpunktmatrix gelesen:

Zwischenpunktmatrix

	A	B	C	D	E
A	—	—	B	C	D
B	E	—	—	C	D
C	E	E	—	—	D
D	E	E	E	—	—
E	—	A	B	C	—

Der kürzeste Weg von D nach C geht über den Zwischenpunkt E.



## Der Algorithmus von Floyd – Ausgabe der Matrizen

Ausgabe der Distanzen- und der Zwischenpunktmatrix:

```
int distanz[ANZAHL][ANZAHL];

void print_distanzen()
{
    int z, s;

    printf( "Distanzen:\n");
    for( z = 0; z < ANZAHL; z++)
    {
        for( s = 0; s < ANZAHL; s++)
            printf( "%3d ", distanz[z][s]);
        printf( "\n");
    }
}
```

```
int zwischenpunkt[ANZAHL][ANZAHL];

void print_zwischenpunkte()
{
    int z, s;

    printf( "Zwischenpunkte:\n");
    for( z = 0; z < ANZAHL; z++)
    {
        for( s = 0; s < ANZAHL; s++)
            printf( "%3d ", zwischenpunkt[z][s]);
        printf( "\n");
    }
}
```

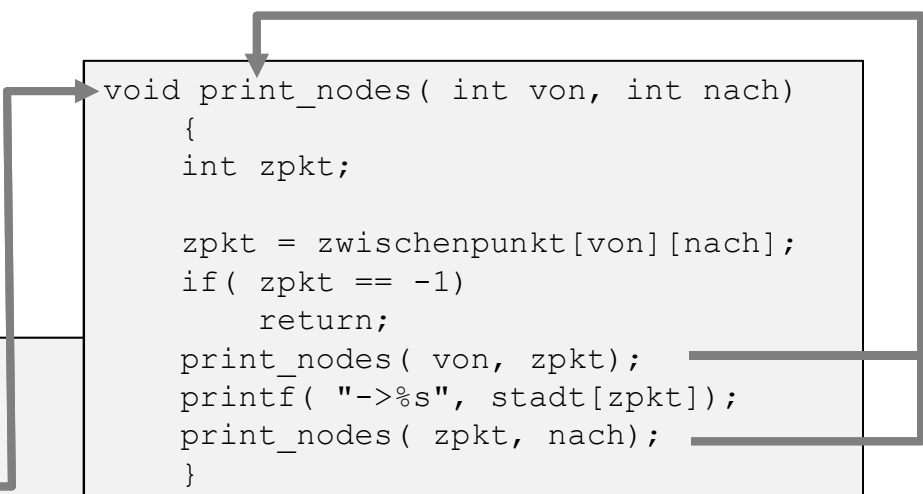
## Der Algorithmus von Floyd – Ausgabe eines optimalen Weges

Zur Ausgabe eines kürzesten Wegs werden in der Funktion `print_path` Start- und Zielknoten sowie die Entfernung ausgegeben. Dazwischen werden mit der Funktion `print_nodes` rekursiv alle Zwischenpunkte aus der Zwischenpunktmatrix gelesen und ausgegeben:

```
void print_path( int von, int nach)
{
    printf( "%s", stadt[von]);
    print_nodes( von, nach);
    printf( "->%s", stadt[nach]);
    printf( " (%d km)\n", distanz[von][nach]);
}
```

```
void print_nodes( int von, int nach)
{
    int zpkt;

    zpkt = zwischenpunkt[von][nach];
    if( zpkt == -1)
        return;
    print_nodes( von, zpkt);
    printf( "->%s", stadt[zpkt]);
    print_nodes( zpkt, nach);
}
```



## Der Algorithmus von Floyd – Initialisierung der Zwischenpunktmatrix

Der Wert  $-1$  in Zeile `von` und Spalte `nach` der Zwischenpunktmatrix zeigt an, dass für den Weg vom Knoten `von` zum Knoten `nach` noch kein Zwischenpunkt berechnet wurde. Die Zwischenpunktmatrix wird dementsprechend initialisiert:

```
void init()
{
    int von, nach;

    for( von = 0; von < ANZAHL; von++)
    {
        for( nach = 0; nach < ANZAHL; nach++)
            zwischenpunkt[von][nach] = -1;
    }
}
```

## Der Algorithmus von Floyd – Berechnung der kürzesten Wege

Von der Idee her ist der Algorithmus von Floyd identisch mit dem Algorithmus von Warshall (Folie 14). Auch hier wird Schritt für Schritt eine Menge von bereits bearbeiteten Knoten aufgebaut:

```
void floyd()
{
    int von, nach, zpkt;
    unsigned int d;

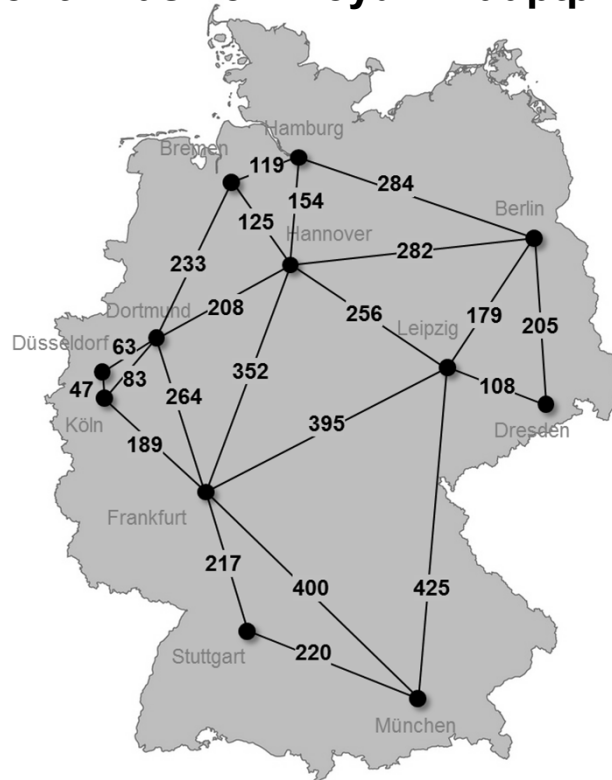
    for( zpkt = 0; zpkt < ANZAHL; zpkt++)
    {
        for( von = 0; von < ANZAHL; von++)
        {
            for( nach = 0; nach < ANZAHL; nach++)
            {
                d = distanz[von][zpkt] + distanz[zpkt][nach];
                if( d < distanz[von][nach])
                {
                    distanz[von][nach] = d;
                    zwischenpunkt[von][nach] = zpkt;
                }
            }
        }
    }
}
```

Es wird geprüft, ob man über den Zwischenpunkt zpkt den Weg vom Knoten von zum Knoten nach verkürzen kann.

Ist eine Verkürzung möglich, hat man eine neue Distanz und einen neuen Zwischenpunkt.

Hier wird jedoch nicht nur nach der Existenz eines Weges über den jeweils neu hinzugekommenen Zwischenpunkt gefragt, sondern es wird geprüft, ob der Weg über den Zwischenpunkt kürzer ist als der bisher kürzeste Weg. Ist das der Fall, wird die neue Distanz in der Distanzenmatrix und der Zwischenpunkt in der Zwischenpunktmatrix gespeichert.

## Der Algorithmus von Floyd – Hauptprogramm und Ausgabe



```
# define BERLIN      0
# define BREMEN      1
# define DORTMUND    2
# define DRESDEN     3
# define DUESSELDORF 4
# define FRANKFURT    5
# define HAMBURG      6
# define HANNOVER     7
# define KOELN        8
# define LEIPZIG      9
# define MUENCHEN    10
# define STUTTGART   11
```

```
void main()
{
    init();
    floyd();

    print_distanzen();
    print_zwischenpunkte();

    print_path( BERLIN, STUTTGART);
    print_path( MUENCHEN, HAMBURG);
}
```

Distanzen:

0	403	490	205	553	574	284	282	573	179	604	791
403	0	233	489	296	477	119	125	316	381	806	694
490	233	0	572	63	264	352	208	83	464	664	481
205	489	572	0	635	503	489	364	655	108	533	720
553	296	63	635	0	236	415	271	47	527	636	453
574	477	264	503	236	0	506	352	189	395	400	217
284	119	352	489	415	506	0	154	435	410	835	723
282	125	208	364	271	352	154	0	291	256	681	569
573	316	83	655	47	189	435	291	0	547	589	406
179	381	464	108	527	395	410	256	547	0	425	612
604	806	664	533	636	400	835	681	589	425	0	220
791	694	481	720	453	217	723	569	406	612	220	0

Zwischenpunkte:

-1	6	7	-1	7	9	-1	-1	7	-1	9	9
6	-1	-1	9	2	7	-1	-1	2	7	9	7
7	-1	-1	9	-1	-1	1	-1	-1	7	5	5
-1	9	9	-1	9	9	0	9	9	-1	9	9
7	2	-1	9	-1	8	2	2	-1	7	8	8
9	7	-1	9	8	-1	7	-1	-1	-1	-1	-1
-1	-1	1	0	2	7	-1	-1	2	7	9	7
-1	-1	-1	9	2	-1	-1	-1	2	-1	9	5
7	2	-1	9	-1	-1	2	2	-1	7	5	5
-1	7	7	-1	7	-1	7	-1	7	-1	-1	5
9	9	5	9	8	-1	9	9	5	-1	-1	-1
9	7	5	9	8	-1	7	5	5	5	-1	-1

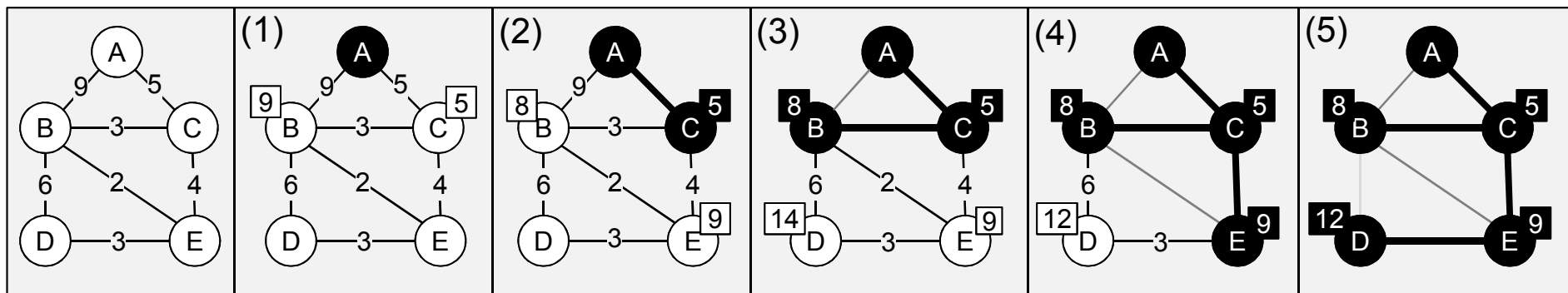
Berlin->Leipzig->Frankfurt->Stuttgart (791 km)  
Muenchen->Leipzig->Hannover->Hamburg (835 km)



## Der Algorithmus von Dijkstra – Verfahrensidee

Suche im unten stehenden Graphen alle günstigsten, von A ausgehenden Wege.

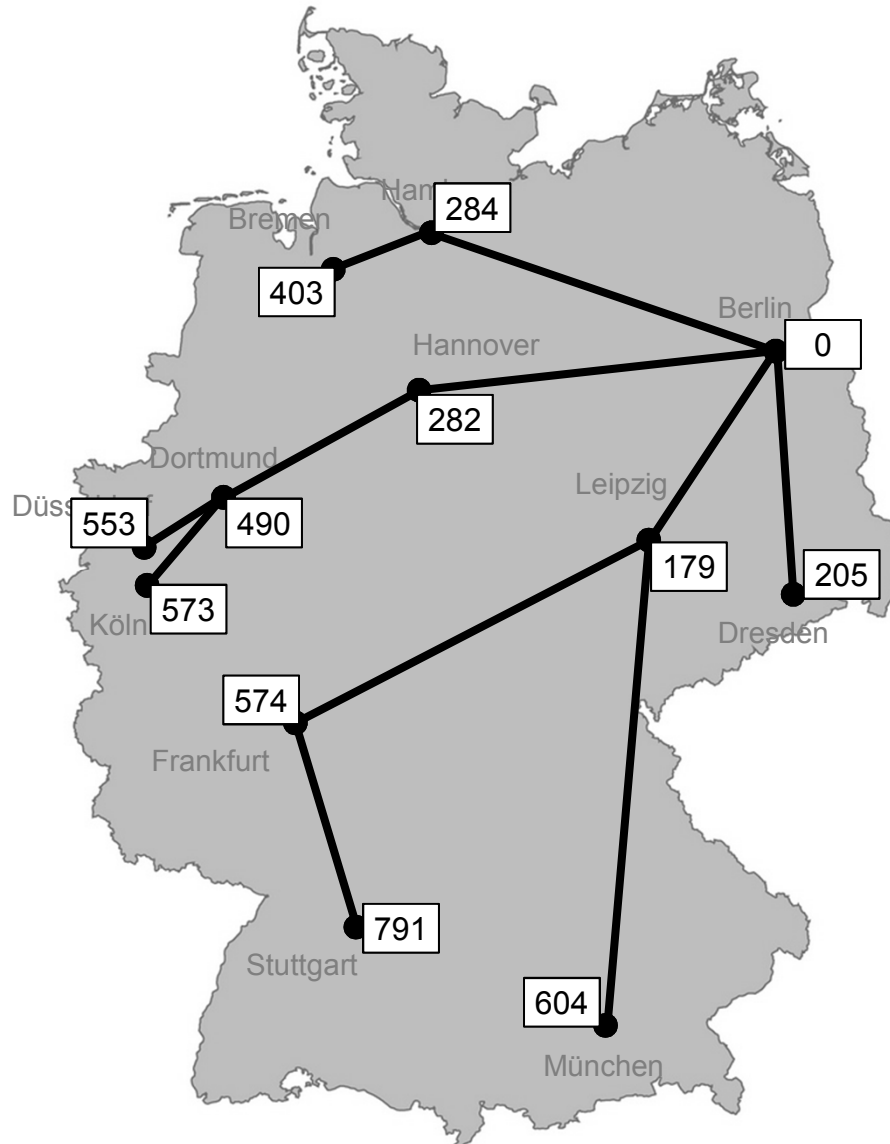
- (1) Starte am Knoten A und bewerte die Knoten, die von dort aus direkt erreichbar sind.
- (2) Wähle den am günstigsten bewerteten Knoten (das ist C) und markiere den Weg, der zu dieser Bewertung geführt hat. Danach bewerte alle von A oder C direkt erreichbaren Knoten. Dabei ergeben ggf. sich neue Bewertungen oder Verbesserungen bisheriger Bewertungen.
- (3) Wähle den am günstigsten bewerteten, noch nicht erledigten, Knoten (B) und markiere den Weg, der zu dieser Bewertung geführt hat. Danach bewerte alle von A, C oder B direkt erreichbaren Knoten.
- (4) Wähle den am günstigsten bewerteten, noch nicht erledigten, Knoten (E) und markiere den Weg, der zu dieser Bewertung geführt hat. Danach bewerte alle von A, C, B oder E direkt erreichbaren Knoten.
- (5) Wähle den am günstigsten bewerteten, noch nicht erledigten, Knoten (D) und markiere den Weg, der zu diesem Knoten geführt hat. Beende das Verfahren, da keine Knoten mehr zu bewerten sind.



Beachten Sie, dass die insgesamt günstigste Kante nicht ausgewählt wurde.

## Der Algorithmus von Dijkstra – Datenstruktur

Die kürzesten Wege von einem festen Startknoten zu allen anderen Knoten bilden einen Baum innerhalb des Graphen, da Teilstrecken kürzester Wege ebenfalls kürzeste Wege sind.



Datenstruktur zur  
Speicherung des Baums.

```
# define ANZAHL 12
struct knoteninfo
{
    unsigned int distanz;
    int vorgaenger;
    char erledigt;
};
struct knoteninfo info[ANZAHL];
```

```
# define BERLIN 0
# define BREMEN 1
# define DORTMUND 2
# define DRESDEN 3
# define DUESSELDORF 4
# define FRANKFURT 5
# define HAMBURG 6
# define HANNOVER 7
# define KOELN 8
# define LEIPZIG 9
# define MUENCHEN 10
# define STUTTGART 11
```

Von Berlin nach  
Dortmund sind 490 km.

	0	1	2	3	4	5	6	7	8	9	10	11
Distanz	0	403	490	205	553	574	284	282	573	179	604	791
Vorgänger	-1	6	7	0	2	9	0	0	2	0	9	5
Erledigt	1	1	1	1	1	1	1	1	1	1	1	1

Der Knoten Dortmund  
ist bearbeitet.

Der Vorgänger des Knotens 2  
(Dortmund) ist der Knoten 7 (Hannover).

## Der Algorithmus von Dijkstra – Initialisierung

Zur Initialisierung werden die Entfernungen aus der zum Startknoten gehörenden Zeile der Distanzenmatrix übernommen. Wenn es keine direkte Verbindung durch eine Kante gibt, ist dieser Wert ist zunächst noch "sehr" groß (xxx = 10000). Nur der Ausgangspunkt wird als "erledigt" markiert. Alle anderen Knoten müssen noch bearbeitet werden.

```
void init( int ausgangspkt)
{
    int i;

    for( i = 0; i < ANZAHL; i++)
    {
        info[i].erledigt = 0;
        info[i].distanz = distanz[ausgangspkt][i];
        info[i].vorgaenger = ausgangspkt;
    }
    info[ausgangspkt].erledigt = 1;
    info[ausgangspkt].vorgaenger = -1;
}
```

```
# define xxx 10000

unsigned int distanz[ ANZAHL][ ANZAHL] =
{
    { 0,xxx,xxx,205,xxx,xxx,284,282,xxx,179,xxx,xxx},
    {xxx, 0,233,xxx,xxx,xxx,119,125,xxx,xxx,xxx,xxx},
    {xxx,233, 0,xxx, 63,264,xxx,208, 83,xxx,xxx,xxx},
    {205,xxx,xxx, 0,xxx,xxx,xxx,xxx,xxx,108,xxx,xxx},
    {xxx,xxx, 63,xxx, 0,xxx,xxx,xxx, 47,xxx,xxx,xxx},
    {xxx,xxx,264,xxx,xxx, 0,xxx,352,189,395,400,217},
    {284,119,xxx,xxx,xxx,xxx, 0,154,xxx,xxx,xxx,xxx},
    {282,125,208,xxx,xxx,352,154, 0,xxx,256,xxx,xxx},
    {xxx,xxx, 83,xxx, 47,189,xxx,xxx, 0,xxx,xxx,xxx},
    {179,xxx,xxx,108,xxx,395,xxx,256,xxx, 0,425,xxx},
    {xxx,xxx,xxx,xxx,xxx,400,xxx,xxx,xxx,425, 0,220},
    {xxx,xxx,xxx,xxx,xxx,217,xxx,xxx,xxx,xxx,220, 0},
};
```

## Der Algorithmus von Dijkstra – Knotenauswahl

In dieser Hilfsfunktion wird unter allen noch nicht erledigten Knoten derjenige ermittelt, der den geringsten Abstand zum Startknoten hat.

```
int knoten_auswahl()
{
    int i, minpos;
    unsigned int min;

    min = xxx;
    minpos = -1;
    for( i = 0; i < ANZAHL; i++)
    {
        if( info[i].distanz < min && !info[i].erledigt)
        {
            min = info[i].distanz;
            minpos = i;
        }
    }
    return minpos;
}
```

Die Funktion gibt den Index des Knotens (oder -1 falls alle Knoten bereits erledigt sind) zurück.

Die Effizienz der Knotensuche kann gesteigert werden, wenn man eine Datenstruktur zur Zwischenspeicherung von Knoten verwendet, die eine effiziente Entnahme des jeweils am nächsten liegenden Knotens ermöglicht, wobei die Struktur nach Einbau eines neuen Knotens in die Menge der erledigten Knoten reorganisiert werden müsste, da sich die Abstände vermindern. Eine geeignete Struktur wäre ein sog. **Fibonacci-Heap**, den wir hier aber nicht behandeln.

## Der Algorithmus von Dijkstra – Kernalgorithmus

Im Kern des Algorithmus wird genau das gemacht, was wir in der Verfahrensidee (Folie 40) bereits diskutiert haben:

```
void dijkstra( int ausgangspkt)
{
    int i, knoten, k;
    unsigned int d;

    init( ausgangspkt);
    for( i = 0; i < ANZAHL-2; i++)
    {
        knoten = knoten_auswahl();
        info[knoten].erledigt = 1;
        for( k = 0; k < ANZAHL; k++)
        {
            if( info[k].erledigt)
                continue;
            d = info[knoten].distanz + distanz[knoten][k];
            if( d < info[k].distanz)
            {
                info[k].distanz = d;
                info[k].vorgaenger = knoten;
            }
        }
    }
}
```

Der Ausgangsknoten ist bereits erledigt und der letzte, am Ende übrig bleibende Knoten muss nicht mehr eigens behandelt werden. Darum wird die Schleife ANZAHL-2 mal durchlaufen.

Wähle den nächsten (= nächstliegenden) Knoten. Der Knoten ist dann erledigt

Iteriere über alle noch nicht erledigten Knoten k.

Wenn der Weg zum Knoten k über den Knoten knoten verkürzt werden kann, dann ergibt sich eine kürzere Distanz und ein neuer Vorgänger. Ansonsten bleibt alles beim Alten.

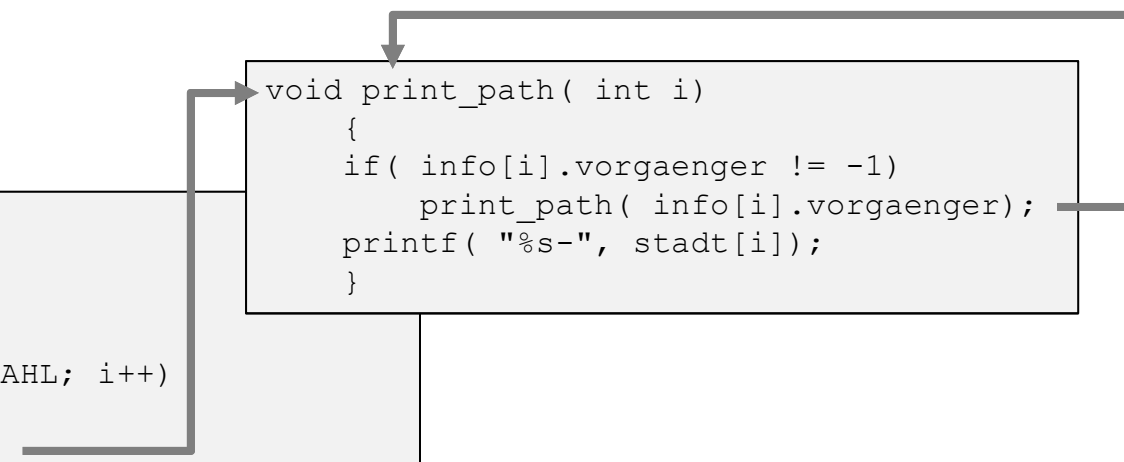
## Der Algorithmus von Dijkstra – Ausgabefunktionen

Da wir einen rückwärts verketteten Baum erzeugt haben, drehen wir bei der Ausgabe die Ausgabereihenfolge der Knoten durch Rekursion um:

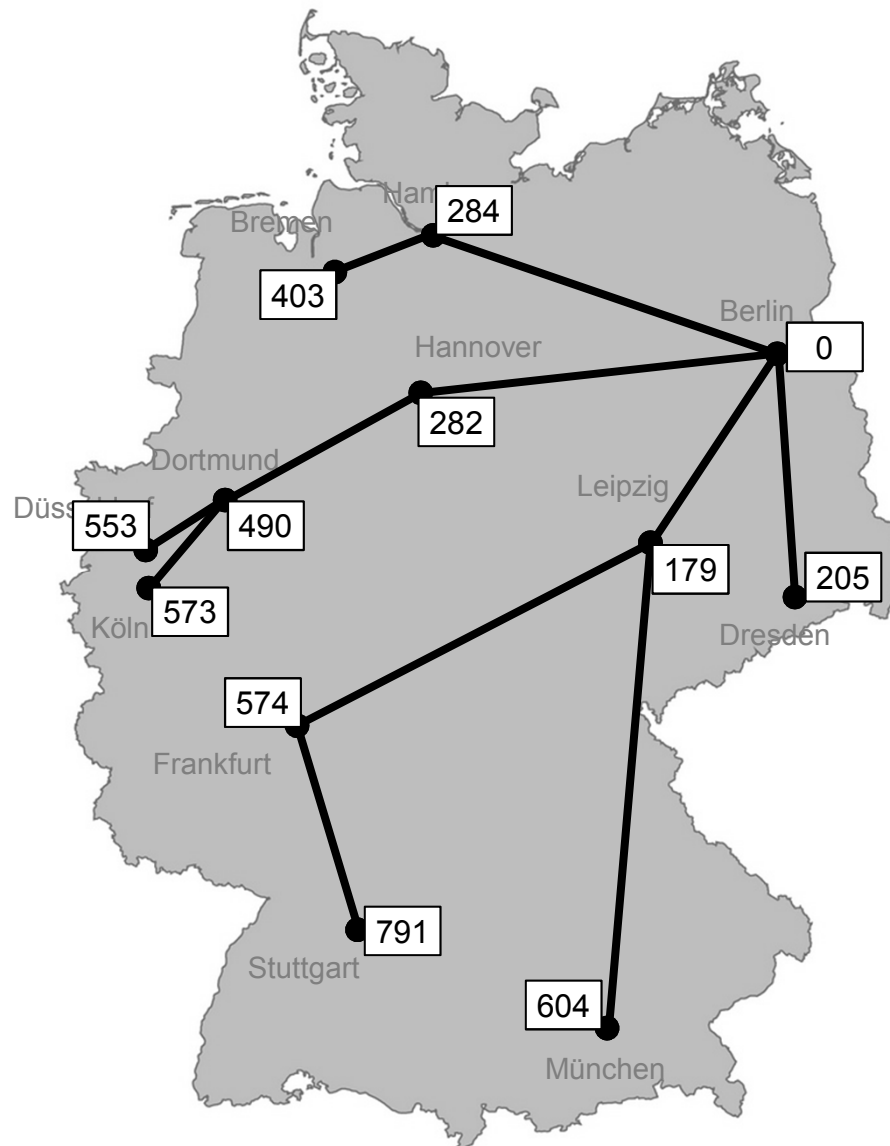
```
void print_all()
{
    int i;

    for( i = 0; i < ANZAHL; i++)
    {
        print_path( i);
        printf( "%dkm\n", info[i].distanz);
    }
}
```

```
void print_path( int i)
{
    if( info[i].vorgaenger != -1)
        print_path( info[i].vorgaenger);
    printf( "%s-", stadt[i]);
}
```



## Der Algorithmus von Dijkstra – Hauptprogramm und Ausgabe

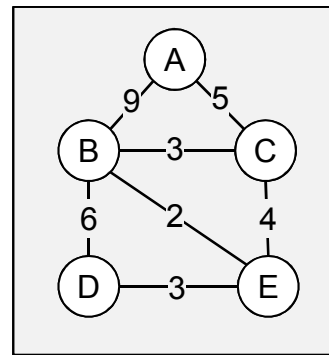


```
void main()  
{  
    dijkstra( BERLIN);  
    print_all();  
}
```

```
Berlin-0km  
Berlin-Hamburg-Bremen-403km  
Berlin-Hannover-Dortmund-490km  
Berlin-Dresden-205km  
Berlin-Hannover-Dortmund-Duesseldorf-553km  
Berlin-Leipzig-Frankfurt-574km  
Berlin-Hamburg-284km  
Berlin-Hannover-282km  
Berlin-Hannover-Dortmund-Koeln-573km  
Berlin-Leipzig-179km  
Berlin-Leipzig-Muenchen-604km  
Berlin-Leipzig-Frankfurt-Stuttgart-791km
```

## Erzeugung einer Kantentabelle

Wenn ein Graph nur relativ wenig Kanten im Verhältnis zur Zahl der Knotenpaare enthält, kann ein kantenorientiertes Vorgehen bei der Suche nach kürzesten Wegen sinnvoll sein.



### Kantentabelle

Kante 1: A → B	9	Kante 8: C → B	3
Kante 2: A → C	5	Kante 9: C → E	4
Kante 3: B → A	9	Kante 10: D → B	6
Kante 4: B → C	3	Kante 11: D → E	3
Kante 5: B → D	6	Kante 12: E → B	2
Kante 6: B → E	2	Kante 13: E → C	4
Kante 7: C → A	5	Kante 14: E → D	3

Erzeuge aus der Distanzenmatrix eines Graphen eine Kantentabelle, die für jede Kante deren Anfangs- und Endpunkt sowie das Kantengewicht enthält:

```
# define ANZAHL 5
# define xxx 10000

int distanz[ ANZAHL][ ANZAHL];

struct kante
{
    int von;
    int nach;
    int distanz;
};

int anzahl_kanten;
struct kante kantentabelle[ANZAHL*ANZAHL];
```

```
void setup_kantentabelle()
{
    int i, j, k, d;

    for( i = k = 0; i < ANZAHL; i++)
    {
        for( j = 0; j < ANZAHL; j++)
        {
            d = distanz[i][j];
            if((d > 0) && (d < xxx))
            {
                kantentabelle[k].distanz = d;
                kantentabelle[k].von = i;
                kantentabelle[k].nach = j;
                k++;
            }
        }
        anzahl_kanten = k;
    }
}
```



## Berechnung der Kantentabelle im deutschen Autobahnnetz

```
# define xxx 10000
```

```
int distanz[ ANZAHL][ ANZAHL] =
{
    { 0,xxx,xxx,205,xxx,xxx,284,282,xxx,179,xxx,xxx},
    {xxx, 0,233,xxx,xxx,xxx,119,125,xxx,xxx,xxx,xxx},
    {xxx,233, 0,xxx, 63,264,xxx,208, 83,xxx,xxx,xxx},
    {205,xxx,xxx, 0,xxx,xxx,xxx,xxx,xxx,108,xxx,xxx},
    {xxx,xxx, 63,xxx, 0,xxx,xxx,xxx, 47,xxx,xxx,xxx},
    {xxx,xxx,264,xxx,xxx, 0,xxx,352,189,395,400,217},
    {284,119,xxx,xxx,xxx,xxx, 0,154,xxx,xxx,xxx,xxx},
    {282,125,208,xxx,xxx,352,154, 0,xxx,256,xxx,xxx},
    {xxx,xxx, 83,xxx, 47,189,xxx,xxx, 0,xxx,xxx,xxx},
    {179,xxx,xxx,108,xxx,395,xxx,256,xxx, 0,425,xxx},
    {xxx,xxx,xxx,xxx,xxx,400,xxx,xxx,xxx,425, 0,220},
    {xxx,xxx,xxx,xxx,xxx,217,xxx,xxx,xxx,xxx,220, 0},
};
```

```
void print_kantentabelle()
{
    int i;
```

```
    for( i = 0; i < anzahl_kanten; i++)
        printf( "Kante %2d: %s->%s %d\n", i+1,
                stadt[kantentabelle[i].von],
                stadt[kantentabelle[i].nach],
                kantentabelle[i].distanz);
}
```

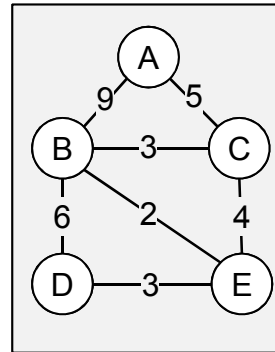
```
void main()
{
    setup_kantentabelle();
    print_kantentabelle();
}
```

```
Kante 1: Berlin->Dresden 205
Kante 2: Berlin->Hamburg 284
Kante 3: Berlin->Hannover 282
Kante 4: Berlin->Leipzig 179
Kante 5: Bremen->Dortmund 233
Kante 6: Bremen->Hamburg 119
Kante 7: Bremen->Hannover 125
Kante 8: Dortmund->Bremen 233
Kante 9: Dortmund->Duesseldorf 63
Kante 10: Dortmund->Frankfurt 264
Kante 11: Dortmund->Hannover 208
Kante 12: Dortmund->Koeln 83
Kante 13: Dresden->Berlin 205
Kante 14: Dresden->Leipzig 108
Kante 15: Duesseldorf->Dortmund 63
Kante 16: Duesseldorf->Koeln 47
Kante 17: Frankfurt->Dortmund 264
Kante 18: Frankfurt->Hannover 352
Kante 19: Frankfurt->Koeln 189
Kante 20: Frankfurt->Leipzig 395
Kante 21: Frankfurt->Muenchen 400
Kante 22: Frankfurt->Stuttgart 217
Kante 23: Hamburg->Berlin 284
Kante 24: Hamburg->Bremen 119
Kante 25: Hamburg->Hannover 154
Kante 26: Hannover->Berlin 282
Kante 27: Hannover->Bremen 125
Kante 28: Hannover->Dortmund 208
Kante 29: Hannover->Frankfurt 352
Kante 30: Hannover->Hamburg 154
Kante 31: Hannover->Leipzig 256
Kante 32: Koeln->Dortmund 83
Kante 33: Koeln->Duesseldorf 47
Kante 34: Koeln->Frankfurt 189
Kante 35: Leipzig->Berlin 179
Kante 36: Leipzig->Dresden 108
Kante 37: Leipzig->Frankfurt 395
Kante 38: Leipzig->Hannover 256
Kante 39: Leipzig->Muenchen 425
Kante 40: Muenchen->Frankfurt 400
Kante 41: Muenchen->Leipzig 425
Kante 42: Muenchen->Stuttgart 220
Kante 43: Stuttgart->Frankfurt 217
Kante 44: Stuttgart->Muenchen 220
```

## Algorithmus von Ford – Verfahrensidee

Gesucht sind alle kürzesten Wege ausgehend vom Knoten D.

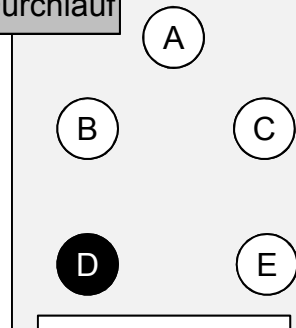
Das Verfahren besteht aus mehreren Durchläufen. In jedem Durchlauf werden der Reihe nach alle Kanten betrachtet und, sofern sie eine Verkürzung ermöglichen, in den Ergebnisbaum eingebaut.



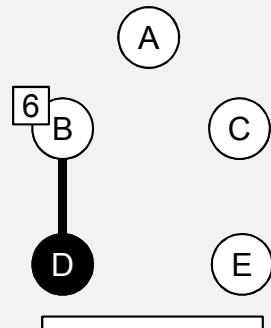
### Kantentabelle

Kante 1: A → B	9	Kante 8: C → B	3
Kante 2: A → C	5	Kante 9: C → E	4
Kante 3: B → A	9	Kante 10: D → B	6
Kante 4: B → C	3	Kante 11: D → E	3
Kante 5: B → D	6	Kante 12: E → B	2
Kante 6: B → E	2	Kante 13: E → C	4
Kante 7: C → A	5	Kante 14: E → D	3

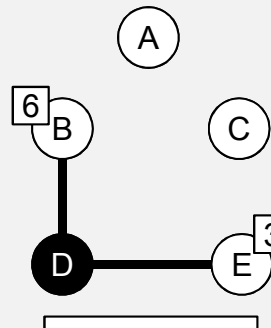
### 1. Durchlauf



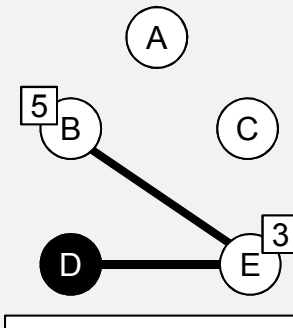
Kante 1 – Kante 9 bringen nichts.



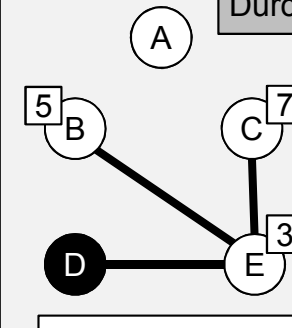
Kante 10 wird eingebaut



Kante 11 wird eingebaut



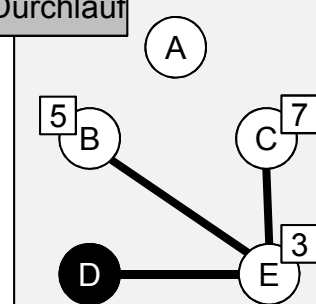
Kante 12 wird statt Kante 11 eingebaut.



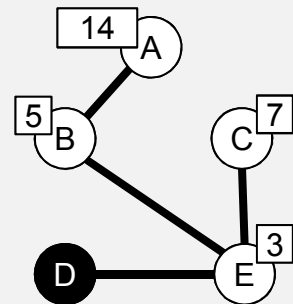
Kante 13 wird eingebaut, Kante 14 bringt nichts

Durchlauf beendet

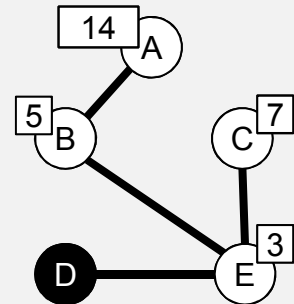
### 2. Durchlauf



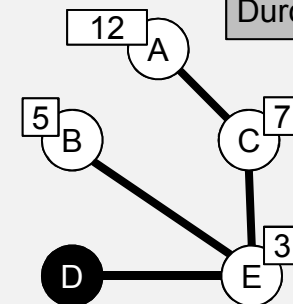
Kante 1 und Kante 2 bringen nichts.



Kante 3 wird eingebaut.



Kante 4 - Kante 6 bringen nichts



Kante 7 wird statt Kante 3 eingebaut. Kanten 8-14 bringen nichts

Durchlauf beendet

Das Verfahren wird so oft durchgeführt, wie sich innerhalb eines Durchlaufs noch Verbesserungen ergeben. Weitere Durchläufe ergeben hier keine Verbesserungen.

## Algorithmus von Ford – Datenstruktur

Die im Algorithmus von Ford zur Speicherung des Ergebnisbaums verwendete Datenstruktur ist bis auf eine Kleinigkeit (das Feld `erledigt` in der Datenstruktur `knoteninfo` wird nicht benötigt) identisch mit der beim Algorithmus von Dijkstra verwendeten Struktur:

```
struct knoteninfo  
{  
    unsigned int distanz;  
    int vorgaenger;  
};  
struct knoteninfo info[ANZAHL];
```

Dementsprechend gleichen sich auch die Funktionen zur Initialisierung und zur Ausgabe dieser Struktur:

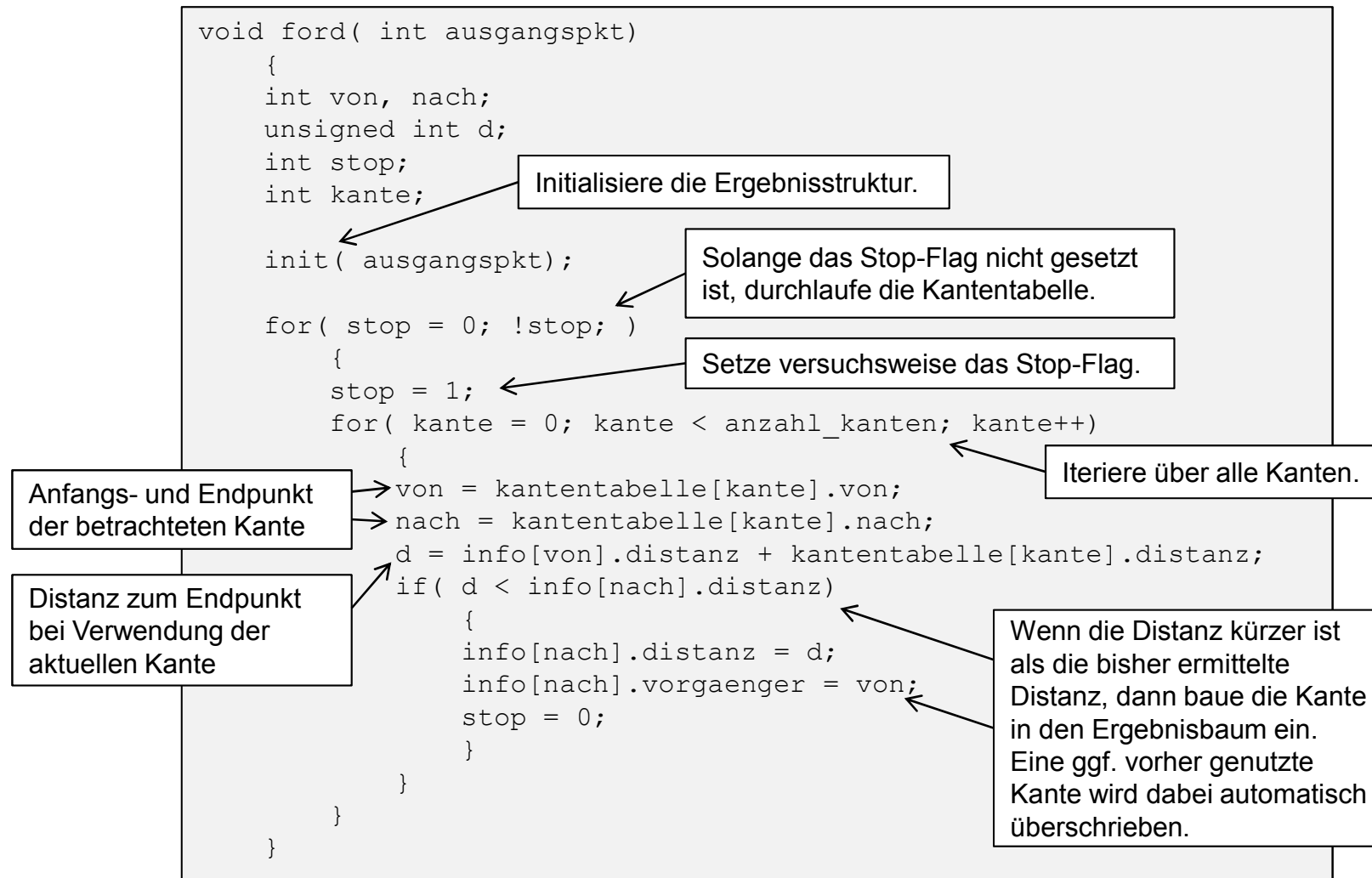
```
void init( int ausgangspkt)  
{  
    int i;  
  
    for( i = 0; i < ANZAHL; i++)  
    {  
        info[i].distanz = xxx;  
        info[i].vorgaenger = ausgangspkt;  
    }  
    info[ausgangspkt].distanz = 0;  
    info[ausgangspkt].vorgaenger = -1;  
}
```

```
void print_all()  
{  
    int i;  
  
    for( i = 0; i < ANZAHL; i++)  
    {  
        print_path( i);  
        printf( "%d km\n", info[i].distanz);  
    }  
}
```

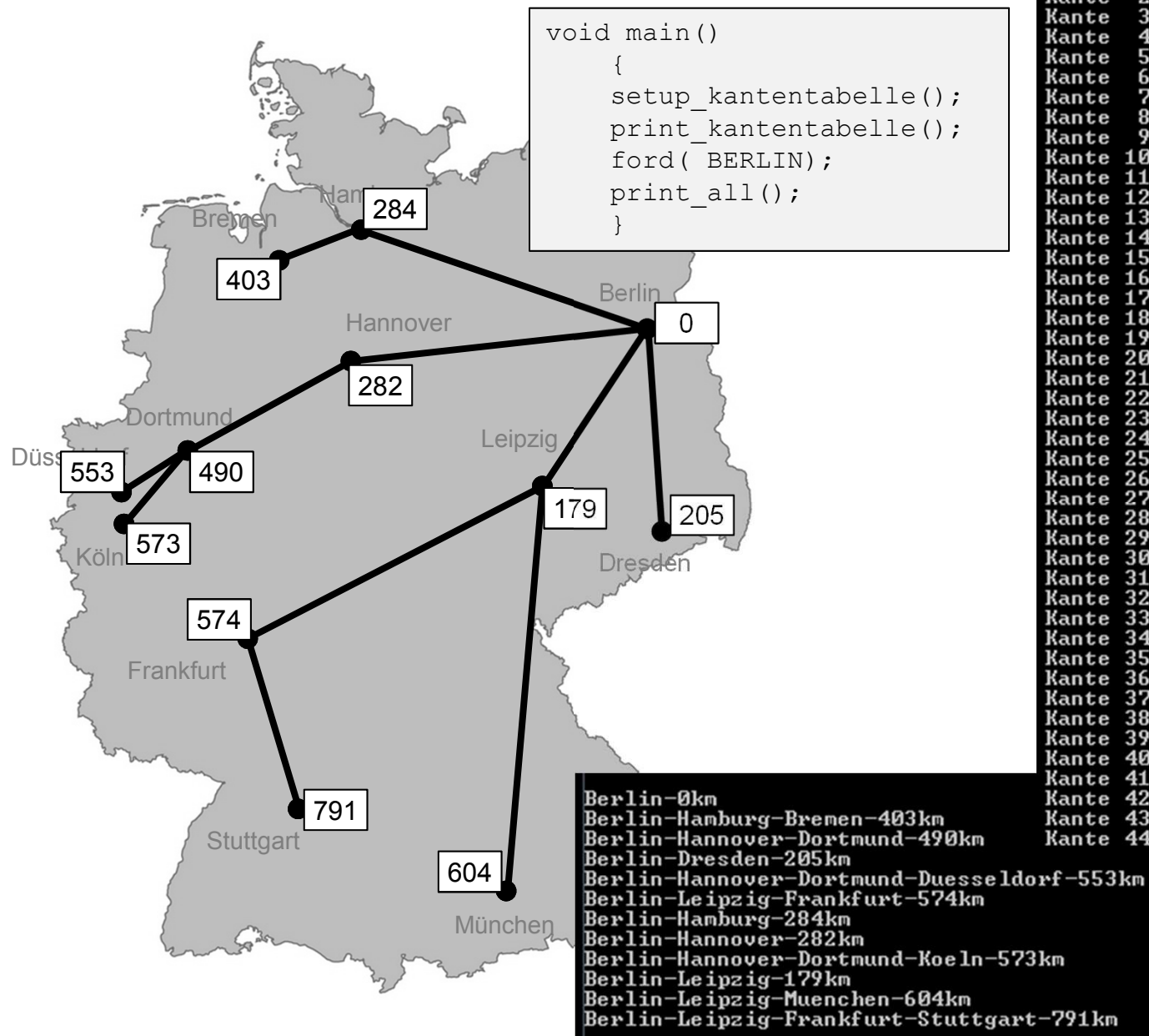
```
void print_path( int i)  
{  
    if( info[i].vorgaenger != -1)  
        print_path( info[i].vorgaenger);  
    printf( "%s-", stadt[i]);  
}
```

## Der Algorithmus von Ford – Kernalgorithmus

Der Kernalgorithmus implementiert die Verfahrensidee (Folie 49).



## Der Algorithmus von Ford – Hauptprogramm und Ausgabe



```

Kante 1: Berlin->Dresden 205
Kante 2: Berlin->Hamburg 284
Kante 3: Berlin->Hannover 282
Kante 4: Berlin->Leipzig 179
Kante 5: Bremen->Dortmund 233
Kante 6: Bremen->Hamburg 119
Kante 7: Bremen->Hannover 125
Kante 8: Dortmund->Bremen 233
Kante 9: Dortmund->Duesseldorf 63
Kante 10: Dortmund->Frankfurt 264
Kante 11: Dortmund->Hannover 208
Kante 12: Dortmund->Koeln 83
Kante 13: Dresden->Berlin 205
Kante 14: Dresden->Leipzig 108
Kante 15: Duesseldorf->Dortmund 63
Kante 16: Duesseldorf->Koeln 47
Kante 17: Frankfurt->Dortmund 264
Kante 18: Frankfurt->Hannover 352
Kante 19: Frankfurt->Koeln 189
Kante 20: Frankfurt->Leipzig 395
Kante 21: Frankfurt->Muenchen 400
Kante 22: Frankfurt->Stuttgart 217
Kante 23: Hamburg->Berlin 284
Kante 24: Hamburg->Bremen 119
Kante 25: Hamburg->Hannover 154
Kante 26: Hannover->Berlin 282
Kante 27: Hannover->Bremen 125
Kante 28: Hannover->Dortmund 208
Kante 29: Hannover->Frankfurt 352
Kante 30: Hannover->Hamburg 154
Kante 31: Hannover->Leipzig 256
Kante 32: Koeln->Dortmund 83
Kante 33: Koeln->Duesseldorf 47
Kante 34: Koeln->Frankfurt 189
Kante 35: Leipzig->Berlin 179
Kante 36: Leipzig->Dresden 108
Kante 37: Leipzig->Frankfurt 395
Kante 38: Leipzig->Hannover 256
Kante 39: Leipzig->Muenchen 425
Kante 40: Muenchen->Frankfurt 400
Kante 41: Muenchen->Leipzig 425
Kante 42: Muenchen->Stuttgart 220
Kante 43: Stuttgart->Frankfurt 217
Kante 44: Stuttgart->Muenchen 220

```

## Minimale Spannbäume

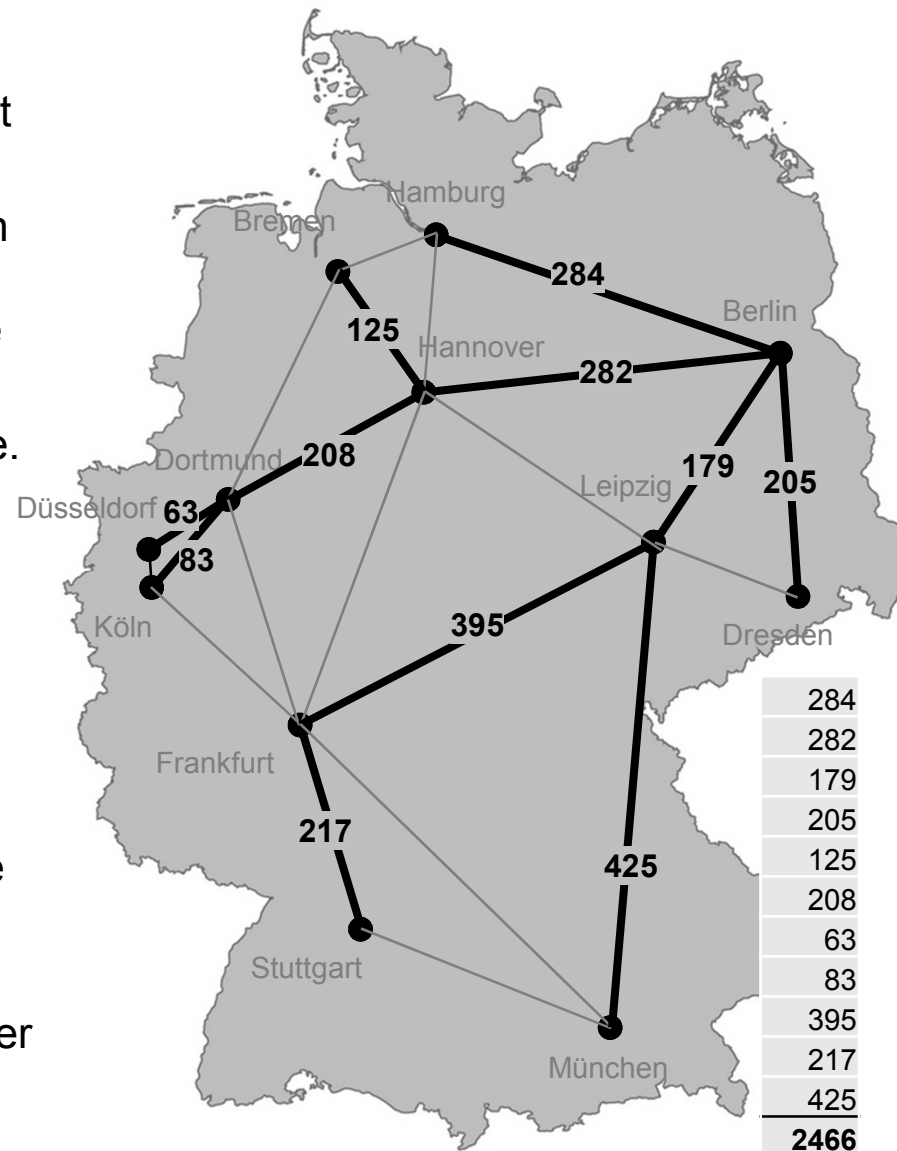
Im Folgenden betrachten wir ungerichtete, zusammenhängende, gewichtete Graphen mit nicht-negativen Kantengewichten.

Unter einem **Spannbaum** verstehen wir einen Teilgraphen eines Graphen, der ein Baum (zusammenhängend und kreisfrei) ist und alle Knoten des Graphen enthält.

Ein Graph hat in der Regel viele Spannbäume. Einen Spannbaum (siehe rechts) erhält man, wenn man aus dem Graphen so lange wie möglich Kanten entfernt, ohne den Zusammenhang zu zerstören.

Als **minimalen Spannbaum** eines Graphen bezeichnen wir den Spannbaum, der unter allen Spannbäumen die niedrigste Kantengewichtssumme hat (z.B. das kürzeste Glasfasernetz längs der Autobahn, das alle Städte miteinander verbindet).

Im nebenstehenden Beispiel ist die Summe der Kantengewichte 2466. Das ist sicher nicht minimal.

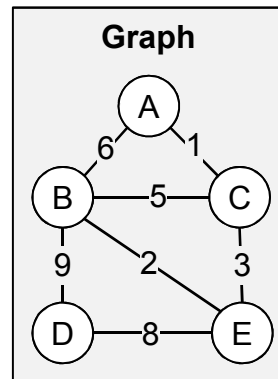


**Gesucht ist ein Algorithmus, der den minimalen Spannbaum eines Graphen berechnet.**



## Der Algorithmus von Kruskal – Verfahrensidee

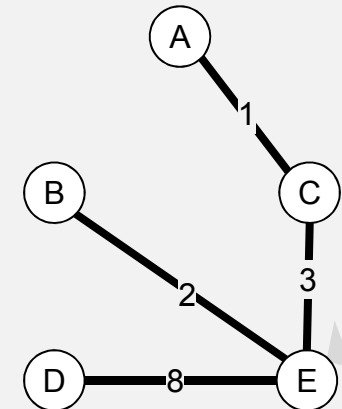
Erzeuge eine Kanten-tabelle, die nach Kantenlänge sortiert ist und berechne mit Hilfe dieser Tabelle den minimalen Spannbaum.



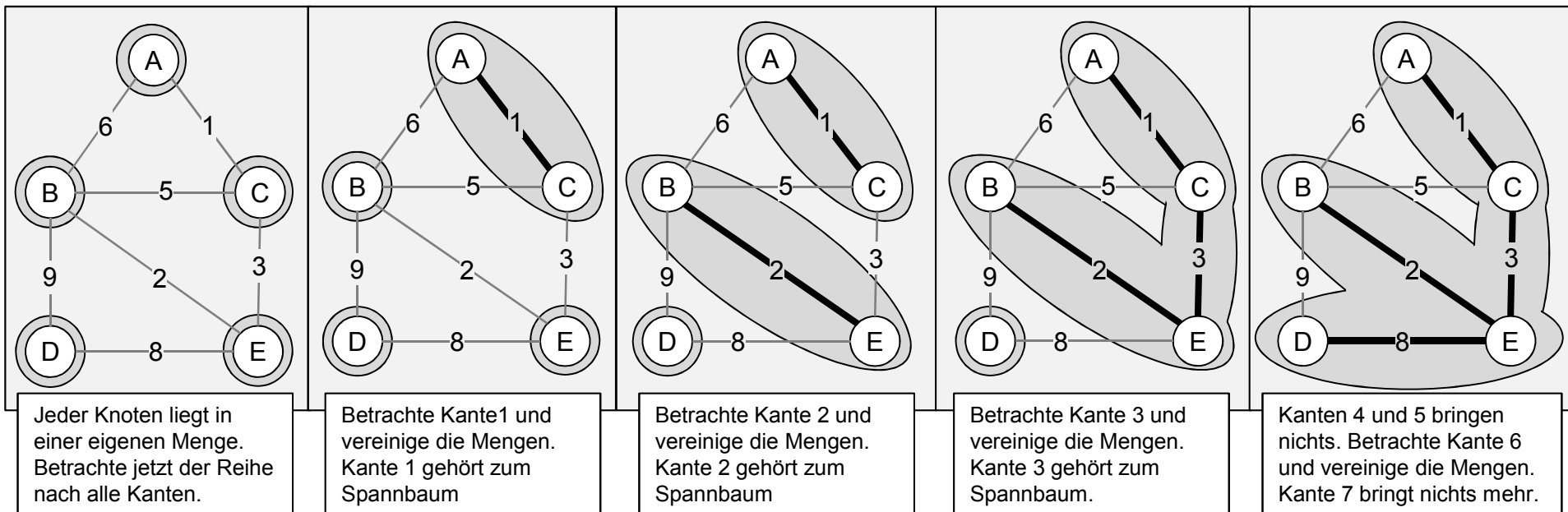
### Sortierte Kanten-tabelle

Kante 1: A ↔ C 1  
 Kante 2: B ↔ E 2  
 Kante 3: C ↔ E 3  
 Kante 4: B ↔ C 5  
 Kante 5: A ↔ B 6  
 Kante 6: D ↔ E 8  
 Kante 7: B ↔ D 9

### Minimaler Spannbaum



Bilde für jeden Knoten eine Menge. Betrachte dann der Reihe nach alle Kanten. Wenn Anfangs- und Endpunkt der Kante in verschiedenen Mengen liegen, dann nimm die Kante hinzu und vereinige die beiden Mengen.



## Der Algorithmus von Kruskal – Datenstruktur

Die Mengen werden als rückwärts verkettete Bäume in einem Vorgängerarray konstruiert:

```
# define ANZAHL 12
int vorgaenger[ANZAHL];
```

Die ausgewählten Kanten werden in einem Array markiert:

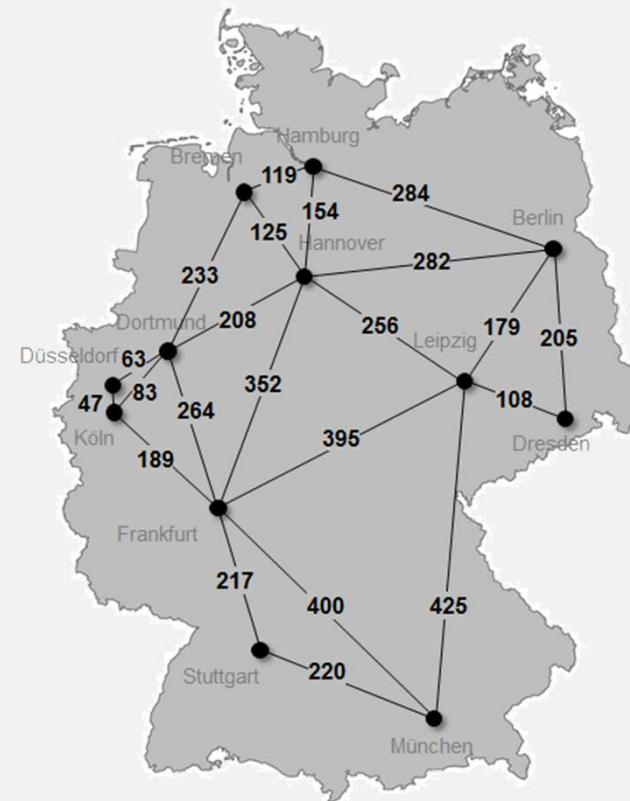
```
# define ANZ_KANTEN 22
int ausgewaehlt[ANZ_KANTEN];
```

Für die Kantentabelle wird eine Datenstruktur deklariert:

```
struct kante
{
    int distanz;
    int von;
    int nach;
};
```

Die Kantentabelle ist ein Array von Kanten (struct kante):

```
struct kante kantentabelle[ANZ_KANTEN] =
{
    { 47, 4, 8},
    { 63, 2, 4},
    { 83, 2, 8},
    { 108, 3, 9},
    { 119, 1, 6},
    { 125, 1, 7},
    { 154, 6, 7},
    { 179, 0, 9},
    { 189, 5, 8},
    { 205, 0, 3},
    { 208, 2, 7},
    { 217, 5, 11},
    { 220, 10, 11},
    { 233, 1, 2},
    { 256, 7, 9},
    { 264, 2, 5},
    { 282, 0, 7},
    { 284, 0, 6},
    { 352, 5, 7},
    { 395, 5, 9},
    { 400, 5, 10},
    { 425, 9, 10},
};
```



Die Kantentabelle ist hier fest vorgegeben. Sie kann aber auch aus der Adjazenzmatrix berechnet werden.



## Der Algorithmus von Kruskal – Implementierung

```
void init()
{
    int i;

    for( i= 0; i < ANZAHL; i++)
        vorgaenger[i] = -1;
}
```

Zur Initialisierung erhält jeder Knoten eine eigene Menge, indem er zur Wurzel (-1) eines rückwärts verketteten Baums gemacht wird.

```
int join( int a, int b)
{
    while( vorgaenger[a] != -1)
        a = vorgaenger[a];
    while( vorgaenger[b] != -1)
        b = vorgaenger[b];
    if( a == b)
        return 0;
    vorgaenger[b] = a;
    return 1;
}
```

Zur Vereinigung der zu den Knoten *a* und *b* gehörenden Mengen werden zunächst die Wurzeln zu *a* und *b* gesucht. Sind die Wurzeln gleich, dann sind die beiden Knoten schon in der gleichen Menge und es muss nichts gemacht werden (`return 0`). Sind die Knoten ungleich, werden die Mengen vereinigt, indem die eine Wurzel (*b*) unter die andere (*a*) gebracht wird (`return 1`).

In der Funktion `kruskal` werden die Kanten der Reihe nach betrachtet und Kanten die zur Vereinigung von zwei Mengen führen, werden im Array `ausgewaehlt` markiert:

```
void kruskal()
{
    int kante;

    init();
    for( kante = 0; kante < ANZ_KANTEN; kante++)
        ausgewaehlt[kante] = join( kantentabelle[kante].von, kantentabelle[kante].nach);
}
```

## Der Algorithmus von Kruskal – Ausgabefunktion

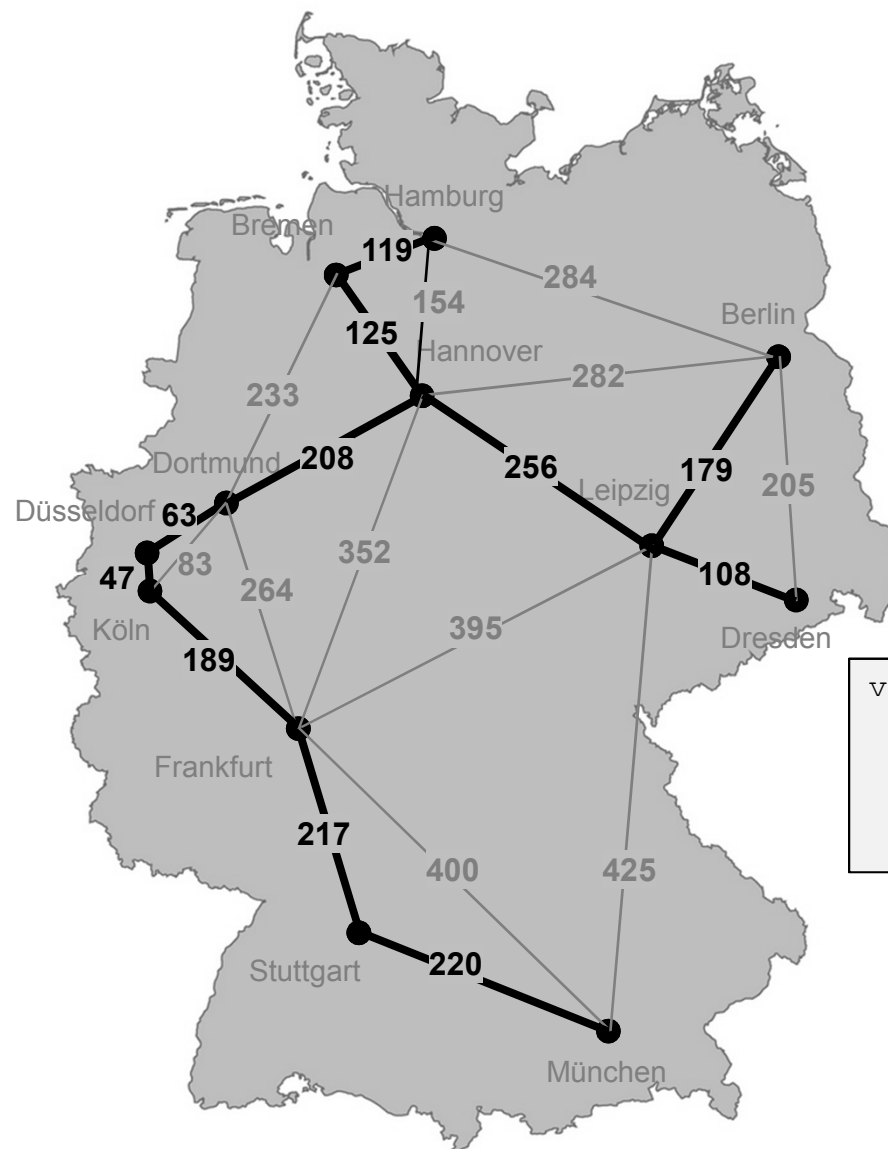
Die ausgewählten Kanten werden ausgegeben:

```
void ausgabe()
{
    int kante;
    unsigned int summe;

    for( kante = 0, summe = 0; kante < ANZ_KANTEN; kante++)
    {
        if( ausgewaehlt[kante])
        {
            summe += kantentabelle[kante].distanz;
            printf( "%4d %s-%s\n", kantentabelle[kante].distanz,
                    stadt[kantentabelle[kante].von],
                    stadt[kantentabelle[kante].nach]);
        }
    }
    printf( "----\n%4d\n", summe);
}
```

Zusätzlich werden die Gewichte der ausgewählten Kanten summiert und am Ende ebenfalls ausgegeben.

## Der Algorithmus von Kruskal – Hauptprogramm und Ausgabe

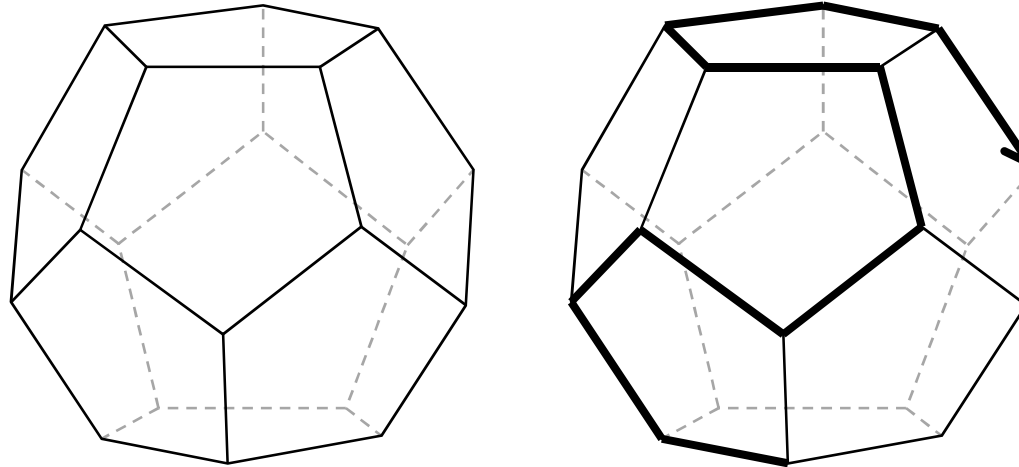


```
void main()  
{  
    kruskal();  
    ausgabe();  
}
```

```
47 Duesseldorf-Koeln  
63 Dortmund-Duesseldorf  
108 Dresden-Leipzig  
119 Bremen-Hamburg  
125 Bremen-Hannover  
179 Berlin-Leipzig  
189 Frankfurt-Koeln  
208 Dortmund-Hannover  
217 Frankfurt-Stuttgart  
220 Muenchen-Stuttgart  
256 Hannover-Leipzig  
-----  
1731
```

## Die Reise um die Welt

Im Jahre 1859 stellte der irische Mathematiker W.R. Hamilton eine Knobelaufgabe vor, bei der es darum ging, auf einem Dodekaeder eine »Reise um die Welt« zu machen.



Ausgehend von einem beliebigen Eckpunkt des Dodekaeders sollte man, immer an den Kanten entlang fahrend, alle anderen Eckpunkte besuchen, um schließlich zum Ausgangspunkt zurückzukehren, ohne einen Eckpunkt zweimal besucht zu haben.

Auf den ersten Blick ähnelt dieses Problem dem Königsberger Brückenproblem. Bei genauerem Hinsehen sind die beiden Probleme jedoch grundverschieden. Bei dem Hamiltonschen Problem geht es darum, alle Knoten eines Graphen genau einmal zu besuchen, während es bei dem Eulerschen Problem darum geht, alle Kanten eines Graphen genau einmal zu benutzen. Dieser Unterschied wirkt unbedeutend, doch erstaunlicherweise sind die Probleme von extrem verschiedener Berechnungskomplexität. Während sich das Problem des Eulerschen Weges in einem Graphen in polynomialer Zeitkomplexität lösen lässt, sind für das Hamiltonsche Problem nur Algorithmen exponentieller Laufzeit bekannt.

## Hamiltonsche Wege

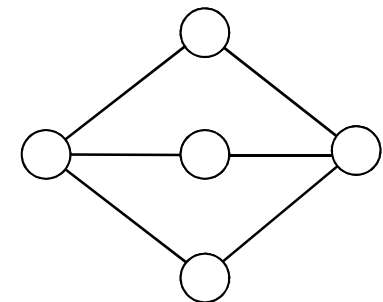
Ein Weg in einem ungerichteten Graphen heißt **Hamiltonscher Weg**, wenn die folgenden drei Bedingungen erfüllt sind:

1. Der Weg ist geschlossen
2. Alle Knoten des Weges, außer Anfangs und Endpunkt, sind voneinander verschieden.
3. Jeder Knoten des Graphen kommt in dem Weg vor.

Ein Hamiltonscher Weg erfüllt immer zwei notwendige Kriterien:

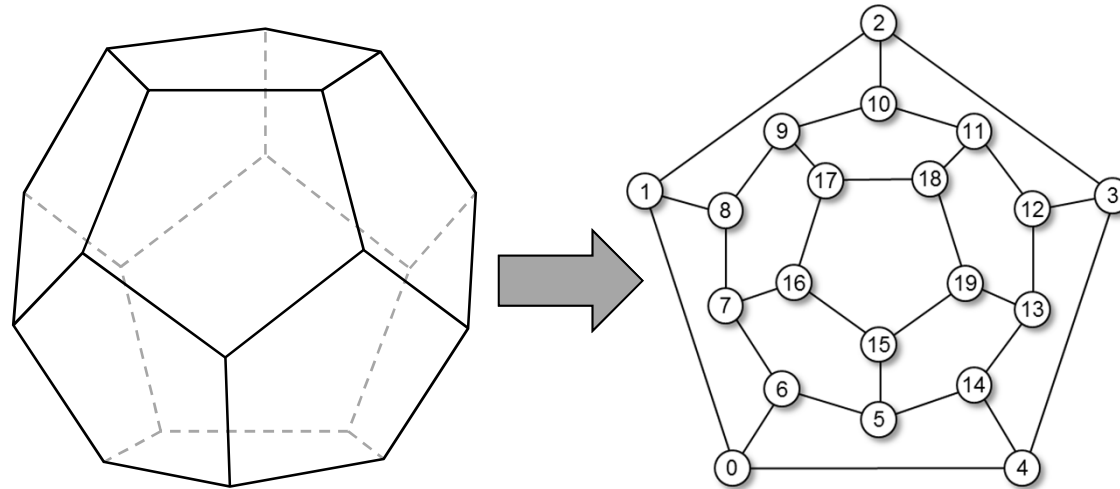
1. Der Weg muss genau so viele Kanten haben, wie der Graph Knoten hat
2. In jedem Knoten des Graphen muss genau eine Kante des Hamiltonschen Weges einlaufen und genau eine Kante auslaufen.

Um im nebenstehenden Graphen einen Hamiltonschen Weg zu finden, müsste man nach (1) eine Kante außer Betracht lassen. Das würde aber immer an einem der drei mittleren Knoten dazu führen, dass Bedingung (2) verletzt ist. **In dem rechts stehenden Graphen gibt es daher keinen Hamiltonschen Weg.**



## Die Reise um die Welt – Lösungsidee

Der Dodekaeder kann durch einen Graphen modelliert werden, in dem ein Hamiltonscher Weg zu finden ist:



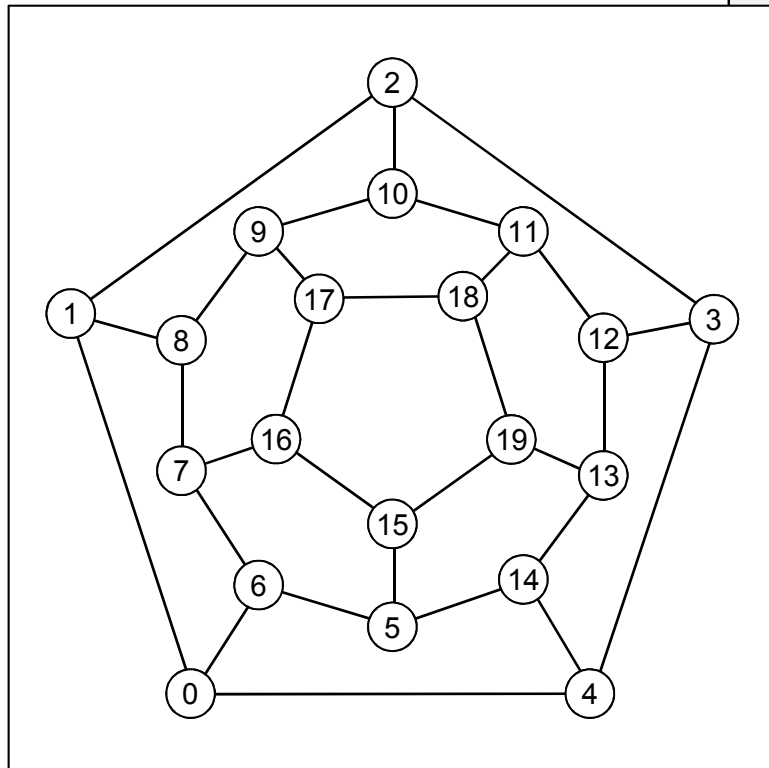
Ein Hamiltonscher Weg ist eine Permutation der Knotenmenge, die zusätzlich die folgenden Bedingungen erfüllt:

1. Jeder Knoten, außer dem letzten, der Permutation muss mit seinem Nachfolger durch eine Kante verbunden sein.
2. Der letzte Knoten der Permutation muss mit dem ersten durch eine Kante verbunden sein.

Um einen Hamiltonschen Weg zu finden, kann man alle Permutationen der Knotenmenge erzeugen und für jede Permutation prüfen, ob sie einen Hamiltonschen Weg beschreibt. Auf diese Weise erhält man nicht nur einen sondern alle Hamiltonschen Wege.

## Die Reise um die Welt – Datenstruktur

Der Graph des Dodekaeders wird durch die Adjazenzmatrix beschrieben:

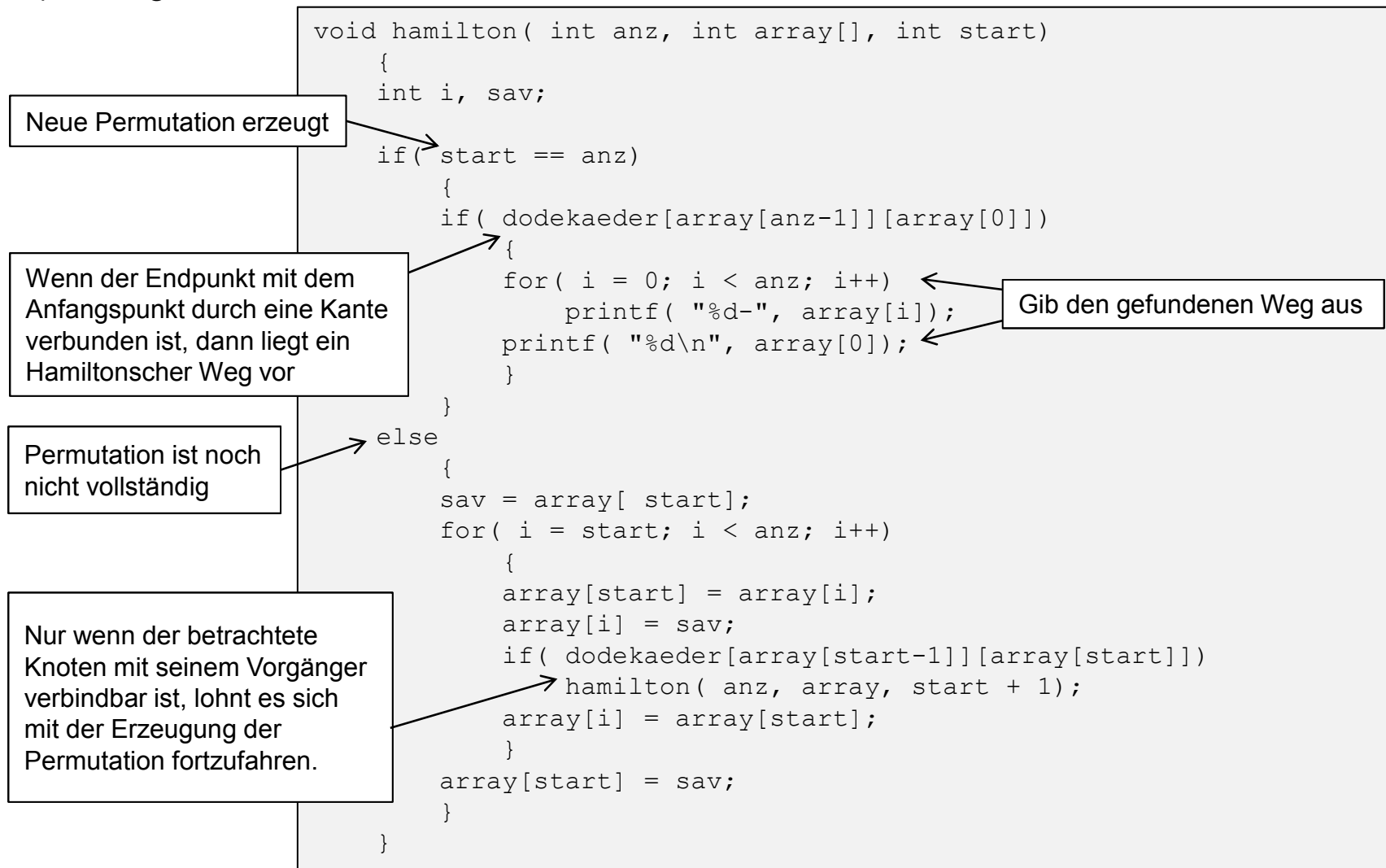


```
# define ANZAHL 20

unsigned int dodekaeder[ ANZAHL][ ANZAHL] =
{
    {0,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0},
    {0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0},
    {0,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
    {1,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0},
    {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0},
    {1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0},
    {0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0},
    {0,0,1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0},
    {0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,1},
    {0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0},
    {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1,0},
    {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1,0},
    {0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,0}
};
```

## Die Reise um die Welt – Implementierung

Die benötigten Permutationen werden mit einer Abwandlung der Funktion `perm` (Kapitel 7, Folie 30) erzeugt:





## Die Reise um die Welt – Hauptprogramm

```
void main()
```

```
{  
  int pfa[ANZAHL];  
  int i;
```

```
  for( i = 0; i < ANZAHL; i++)  
    pfa[i] = i;
```

```
  hamilton(ANZAHL, pfa, 1);  
}
```

Array für die Permutationen

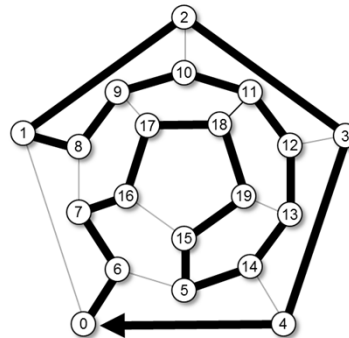
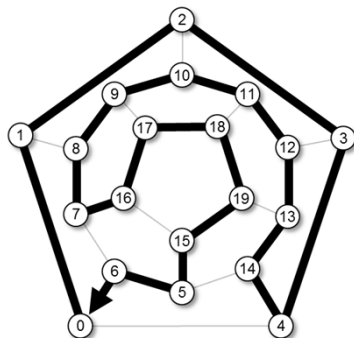
Initialisieren des zu permutierenden Arrays

Suche nach Hamiltonschen Wegen.  
Starte bei dem zweiten Element des  
Pfa, da das erste Element als Startpunkt  
festgehalten wird.

Das Programm findet 60 Lösungen:

```
1: 0-1-2-3-4-14-13-12-11-10-9-8-7-16-17-18-19-15-5-6-0  
2: 0-1-2-3-4-14-5-15-16-17-18-19-13-12-11-10-9-8-7-6-0  
...  
59: 0-6-7-16-17-18-11-12-13-19-15-5-14-4-3-2-10-9-8-1-0  
60: 0-6-7-16-17-18-19-15-5-14-13-12-11-10-9-8-1-2-3-4-0
```

Die erste und die letzte Lösung:



## Das Travelling Salesman Problem (TSP)

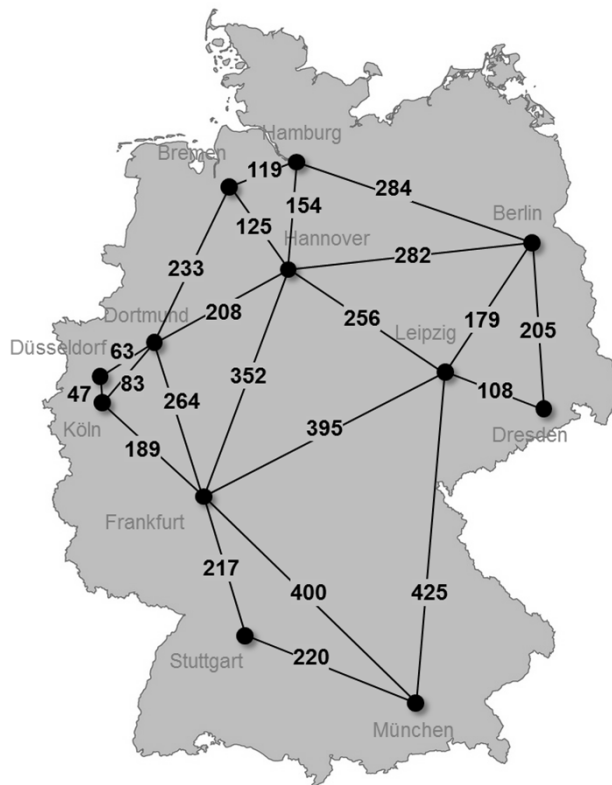
Das Problem, einen möglichst kurzen Hamiltonschen Weg in einem nicht negativ bewerteten Graphen zu finden, wird auch als **Problem des Handlungsreisenden** (engl. **Travelling Salesman Problem**, kurz **TSP**) bezeichnet.

Ein Handlungsreisender will alle seine Kunden besuchen. Er startet mit der Rundreise von seinem Büro und möchte am Ende der Rundreise wieder an seinem Schreibtisch sitzen. Unter allen möglichen Reiserouten möchte er natürlich die mit der kürzesten Gesamtstrecke wählen.

Mit der Lösungsstrategie der "Reise um die Welt" können wir dieses Problem lösen, wenn wir zusätzlich die Weglängen berechnen und uns den jeweils kürzesten Weg speichern.

## Die kürzeste Rundreise in Deutschland – Datenstruktur

Zusätzlich zur Distanzenmatrix (`distanz`) benötigen wir die Länge der kürzesten Rundreise (`mindist`) und einen Array (`minpfad`), in dem wir den Pfad der kürzesten Rundreise ablegen.



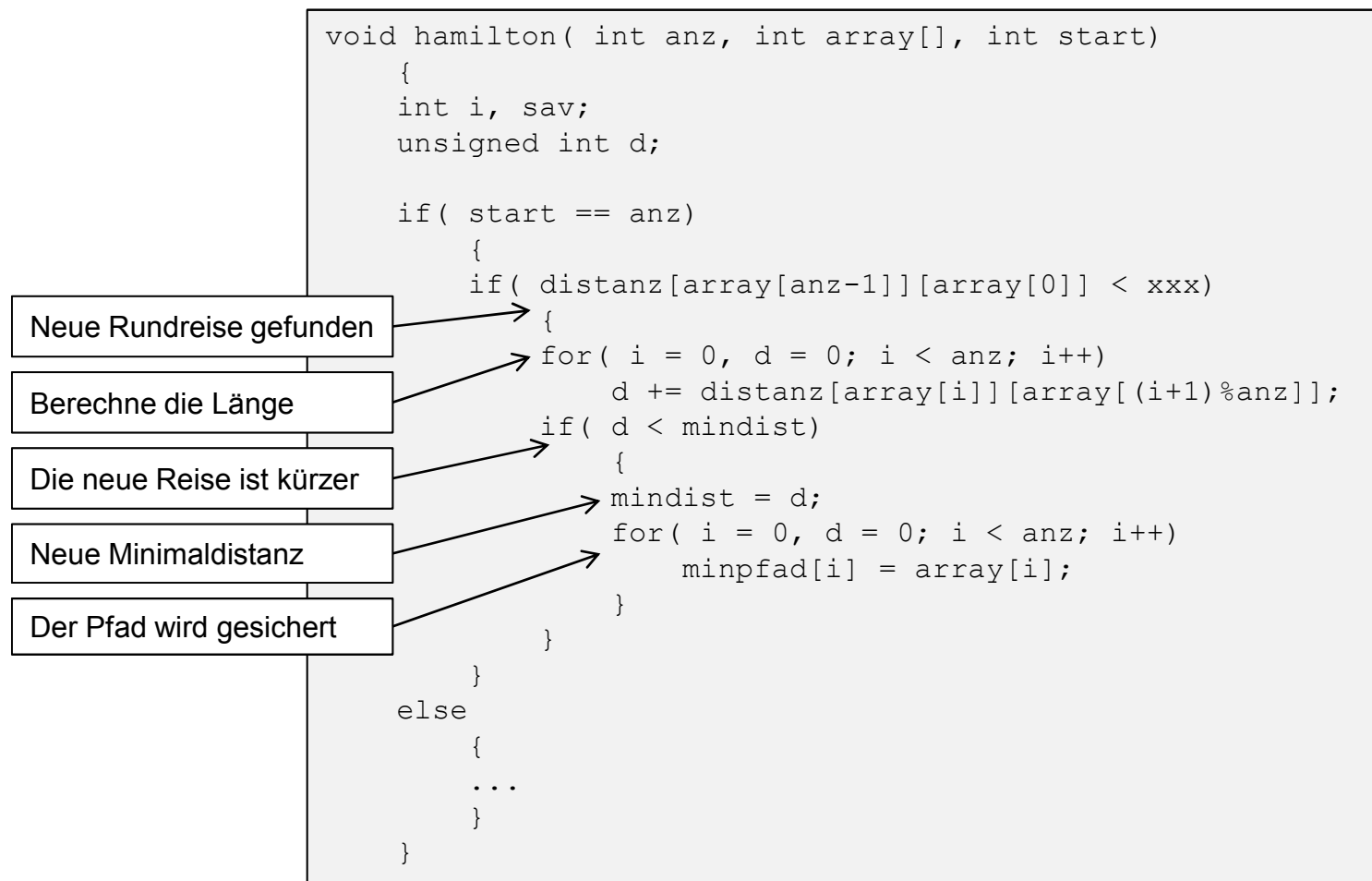
```
# define ANZAHL 12
# define xxx 10000
```

```
int distanz[ ANZAHL][ ANZAHL] =
{
    { 0,xxx,xxx,205,xxx,xxx,284,282,xxx,179,xxx,xxx},
    {xxx, 0,233,xxx,xxx,xxx,119,125,xxx,xxx,xxx,xxx},
    {xxx,233, 0,xxx, 63,264,xxx,208, 83,xxx,xxx,xxx},
    {205,xxx,xxx, 0,xxx,xxx,xxx,xxx,xxx,108,xxx,xxx},
    {xxx,xxx, 63,xxx, 0,xxx,xxx,xxx, 47,xxx,xxx,xxx},
    {xxx,xxx,264,xxx,xxx, 0,xxx,352,189,395,400,217},
    {284,119,xxx,xxx,xxx,xxx, 0,154,xxx,xxx,xxx,xxx},
    {282,125,208,xxx,xxx,352,154, 0,xxx,256,xxx,xxx},
    {xxx,xxx, 83,xxx, 47,189,xxx,xxx, 0,xxx,xxx,xxx},
    {179,xxx,xxx,108,xxx,395,xxx,256,xxx, 0,425,xxx},
    {xxx,xxx,xxx,xxx,xxx,400,xxx,xxx,xxx,425, 0,220},
    {xxx,xxx,xxx,xxx,xxx,217,xxx,xxx,xxx,xxx,220, 0},
};
```

```
int mindist = xxx;
int minpfad[ANZAHL];
```

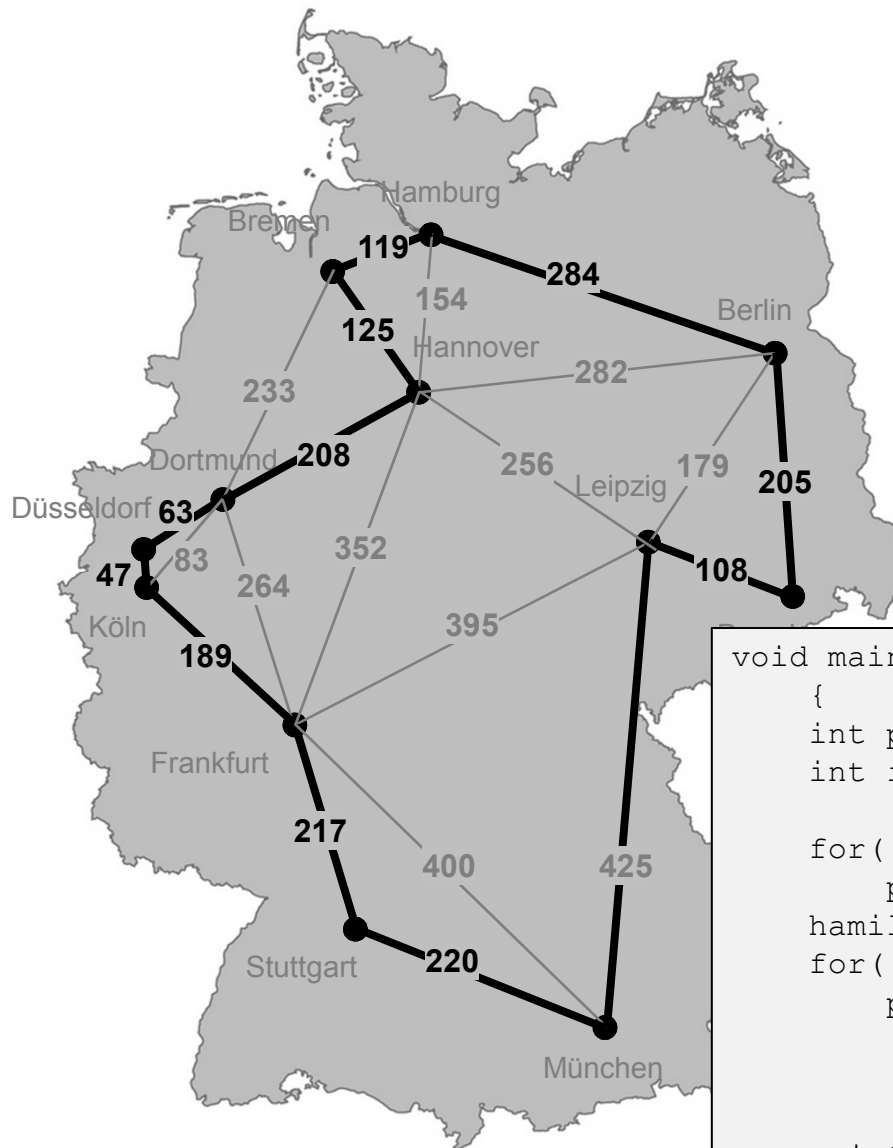
## Die kürzeste Rundreise in Deutschland – Implementierung

Wir erzeugen, wie in der "Reise um die Welt", alle möglichen Rundreisen im deutschen Autobahnnetz. Immer, wenn eine neue Rundreise erzeugt ist, berechnen wir deren Länge. Wenn die Rundreise kürzer als die bisher kürzeste Rundreise ist, kopieren wir den Pfad der Rundreise in den Array `minpfad` um und wir haben eine neue minimale Distanz (`mindist`).



## Die kürzeste Rundreise in Deutschland – Hauptprogramm und Ausgabe

Das Programm findet 6 Hamiltonsche Wege, von denen der folgende der kürzeste ist:



```

205 Berlin-Dresden
108 Dresden-Leipzig
425 Leipzig-Muenchen
220 Muenchen-Stuttgart
217 Stuttgart-Frankfurt
189 Frankfurt-Koeln
47 Koeln-Duesseldorf
63 Duesseldorf-Dortmund
208 Dortmund-Hannover
125 Hannover-Bremen
119 Bremen-Hamburg
284 Hamburg-Berlin
2210

```

```

char *stadt[ANZAHL] =
{
    "Berlin",
    "Bremen",
    "Dortmund",
    "Dresden",
    "Duesseldorf",
    "Frankfurt",
    "Hamburg",
    "Hannover",
    "Koeln",
    "Leipzig",
    "Muenchen",
    "Stuttgart"
};

```

```

void main()
{
    int pfad[ANZAHL];
    int i;

    for( i = 0; i < ANZAHL; i++)
        pfad[i] = i;
    hamilton(12, pfad, 1);
    for( i = 1; i <= ANZAHL; i++)
        printf( "%5d %s-%s\n",
                distanz[minpfad[i-1]][minpfad[i%ANZAHL]],
                stadt[minpfad[i-1]],
                stadt[minpfad[i%ANZAHL]] );
    printf( "%5d\n", mindist);
}

```

## Die kürzeste Rundreise in Deutschland – Die Komplexität der Aufgabe

Ob das Rundreiseprogramm wirklich die absolut kürzeste Rundreise durch Deutschland geliefert hat, wissen wir nicht. Das Programm hat nur die kürzeste Rundreise innerhalb des ihm zur Verfügung stehenden Graphen berechnet. Der Graph enthielt aber nur ausgewählte Städteverbindungen. Zum Beispiel gab es keine Direktverbindung von Berlin nach Frankfurt und diese Direktverbindung wird kürzer sein, als der Weg über Hannover oder Leipzig. Die kürzeste Reiseroute könnte mit einer Direktfahrt von Berlin nach Frankfurt beginnen. Leipzig und Hannover würden dann später angefahren. Wenn wir wirklich die kürzeste Reiseroute zwischen unseren 12 Städten finden wollen, müssen wir einen Graphen betrachten, der alle paarweisen Städteverbindungen enthält. In solch einem Graphen steht jede Permutation der Knoten für einen Hamiltonschen Weg. Da der erste Knoten fest gewählt werden kann, müssen noch  $11! = 39916800$ , also knapp 40 Millionen Hamiltonsche Wege untersucht werden. In unserem Programm wurden dagegen nur 6 Wege verglichen.

Wenn man die vollständige Untersuchung macht, stellt man fest, dass ein kürzerer Weg, als der von uns bereits gefundene, trotz des erheblich größeren Suchraums nicht gefunden werden kann. Irgendwie ist es uns mit Intuition oder Glück gelungen, durch eine geschickte Vorauswahl von Kanten den Umfang der Aufgabe drastisch zu verkleinern, ohne die optimale Lösung zu verlieren. Das liegt natürlich an der geometrischen Anschaulichkeit des Problems. Bei vielen Optimierungsaufgaben fehlt diese Anschauung und die kombinatorische Zahl der zu betrachtenden Permutationen ist noch um ein Vielfaches größer.

Ein Computer hat nicht die Intuition, eine solche Vorauswahl zu treffen und es ist bisher nicht gelungen, eine allgemeine Lösung des Travelling Salesman Problems zu finden, die die kombinatorische Explosion vermeidet. Beim Versuch, die Explosion zu vermeiden, hat man aber Überraschendes und Tieflegendes entdeckt.

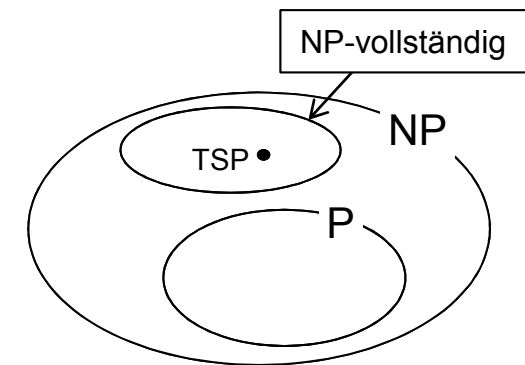
## Ein Exkurs in die theoretische Informatik

Für die Menge aller, mit polynomialer Berechnungskomplexität lösbaren Probleme verwenden wir die Bezeichnung  $P$ .

Die Menge aller Probleme die man algorithmisch lösen kann und für die man eine Lösung mit polynomialer Komplexität überprüfen kann, nennen wir  $NP$ .  $P$  ist eine Teilmenge von  $NP$ .

Das Travelling Salesmann Problem ist zum Beispiel ein Problem aus  $NP$ .

Es gibt in  $NP$  – und das kann man beweisen – sogenannte  $NP$ -vollständige Probleme. Dies sind Probleme, auf die man alle Probleme in  $NP$  mit einem maximal polynomialen Zusatzaufwand reduzieren kann. Das TSP ist ein solches Problem. Hätte man also einen Algorithmus polynomialer Laufzeit für ein  $NP$ -vollständiges Problem (z.B. für das TSP), so könnte man alle Probleme aus  $NP$  mit polynomialen Aufwand lösen. Es wäre  $NP=P$ . In diesem Fall müssten alle Bücher über Algorithmen neu geschrieben werden.



Die Frage, ob es für ein  $NP$ -vollständiges Problem einen polynomialen Lösungsalgorithmus gibt, ist vielleicht die bedeutendste Frage der theoretischen Informatik und sie ist bis heute unbeantwortet. Auf die Beantwortung dieser Frage ist eine Belohnung von 1 Millionen Dollar ausgesetzt. Sollten Sie die Antwort auf diese Frage finden, können sie sich hier

<http://www.claymath.org>

ihr Preisgeld abholen.

## Näherungslösung für das TSP

Für die allgemeine Lösung TSP sind nur Algorithmen exponentieller Laufzeit verfügbar. Dies bedeutet, dass man das TSP für "große" Graphen nicht in akzeptabler Zeit lösen kann. Da man aber für viele technische und betriebswirtschaftliche Fragestellungen an einer Lösung des TSP interessiert ist muss man einen Kompromiss zwischen zwei gegensätzlichen Anforderungen suchen:

1. Der Suchraum sollte so klein sein, dass man zu einem Algorithmus von polynomialer Laufzeit kommt.
2. Der Suchraum sollte so groß sein, dass sich die optimale Lösung oder zumindest eine halbwegs optimale Lösung noch darin befindet.

Gesucht ist ein Algorithmus polynomialer Laufzeitkomplexität der eine Näherungslösung für das Problem des Handlungsreisenden findet, die eine garantierte Maximalabweichung von der unbekannten optimalen Lösung hat.

Wir treffen zwei zusätzliche Annahmen über den Graphen:

1. Für je zwei Knoten gibt es eine direkte Verbindung (Kante) im Graphen.
2. Direkte Verbindungen sind nie länger als Umwegstrecken über einen dritten Ort.

Die allgemeinen Rahmenbedingungen Symmetrie, Zusammenhang und keine negativ bewerteten Kanten bleiben natürlich weiterhin bestehen.

Unter diesen Annahmen, die in wichtigen Anwendungsfällen zutreffen, ist es möglich, eine Näherungslösung für das TSP zu finden. Der Algorithmus dazu basiert auf minimalen Spannbäumen, die wir ja in polynomialer Laufzeit berechnen können.



## Näherungslösung für das TSP durch einen minimalen Spannbaum 1

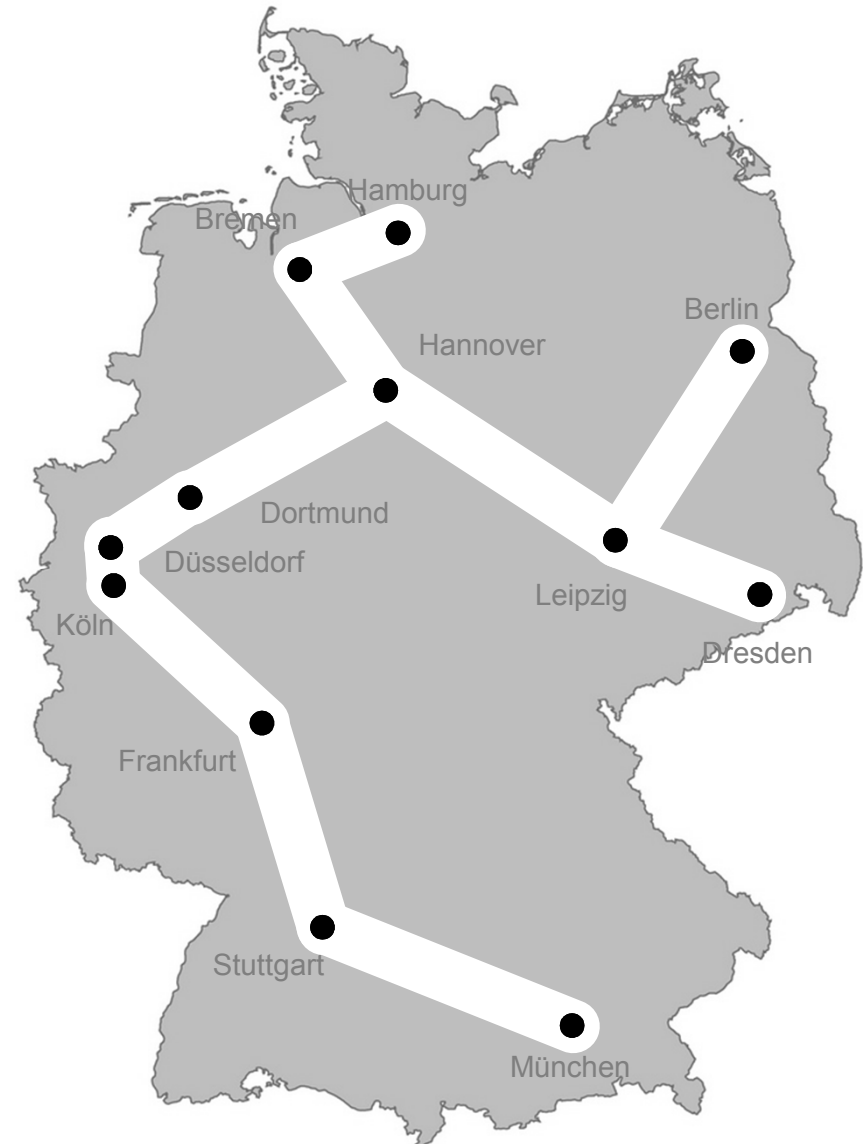
Wenn man aus einem Hamiltonschen Weg eine Kante entfernt, erhält man einen Spannbaum.

Umgekehrt kann man versuchen, aus einem Spannbaum einen Hamiltonschen Weg zu konstruieren.

Dazu betrachten wir den bereits berechneten Spannbaum des deutschen Autobahnnetzes und durchlaufen diesen Baum, von Berlin aus startend, in Tiefensuche, wobei wir die Nachfolger eines Knotens in alphabetisch aufsteigender Reihenfolge besuchen.

Es ergibt sich die folgende Besuchsreihenfolge:

Start: Berlin  
Vor: Leipzig-Dresden  
Zurück: Leipzig  
Vor: Hannover-Bremen-Hamburg  
Zurück: Bremen-Hannover  
Vor: Dortmund-Düsseldorf-Köln-Frankfurt-Stuttgart-München  
Zurück: Stuttgart-Frankfurt-Köln-Düsseldorf-Dortmund-Hannover-Leipzig  
Ziel: Berlin



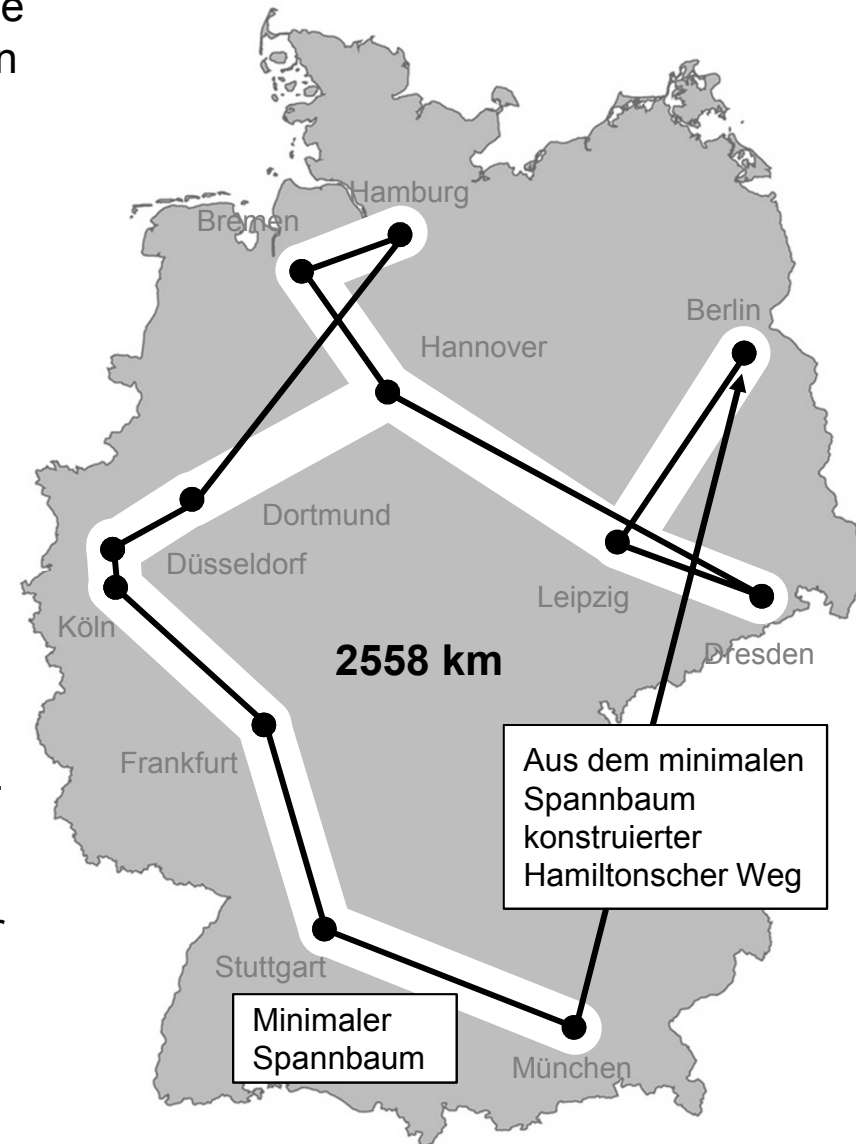
## Näherungslösung für das TSP durch einen minimalen Spannbaum 2

Überspringt man beim Rückzug aus der Tiefensuche die Knoten, an denen man schon war, so erhält man die folgende Besuchsfolge:

Start: Berlin  
Vor: Leipzig-Dresden  
Zurück: Leipzig  
Vor: Hannover-Bremen-Hamburg  
Zurück: Bremen-Hannover  
Vor: Dortmund-Düsseldorf-Köln-Frankfurt-Stuttgart-München  
Zurück: Stuttgart-Frankfurt-Köln-Düsseldorf-Dortmund-Hannover-Leipzig  
Ziel: Berlin

Das ist ein Hamiltonscher Weg der Länge 2558 km.

Je nach Startknoten erhält man unter Umständen einen anderen Hamiltonschen Weg. Man könnte für jeden Startknoten diesen Hamiltonschen Weg berechnen und den kürzesten dieser Wege als Näherungslösung für das TSP verwenden.



## Näherungslösung für das TSP – Distanzenmatrix

Für den Algorithmus wird eine Distanzenmatrix verwendet, die die Entfernungen zwischen allen Städtepaaren enthält:

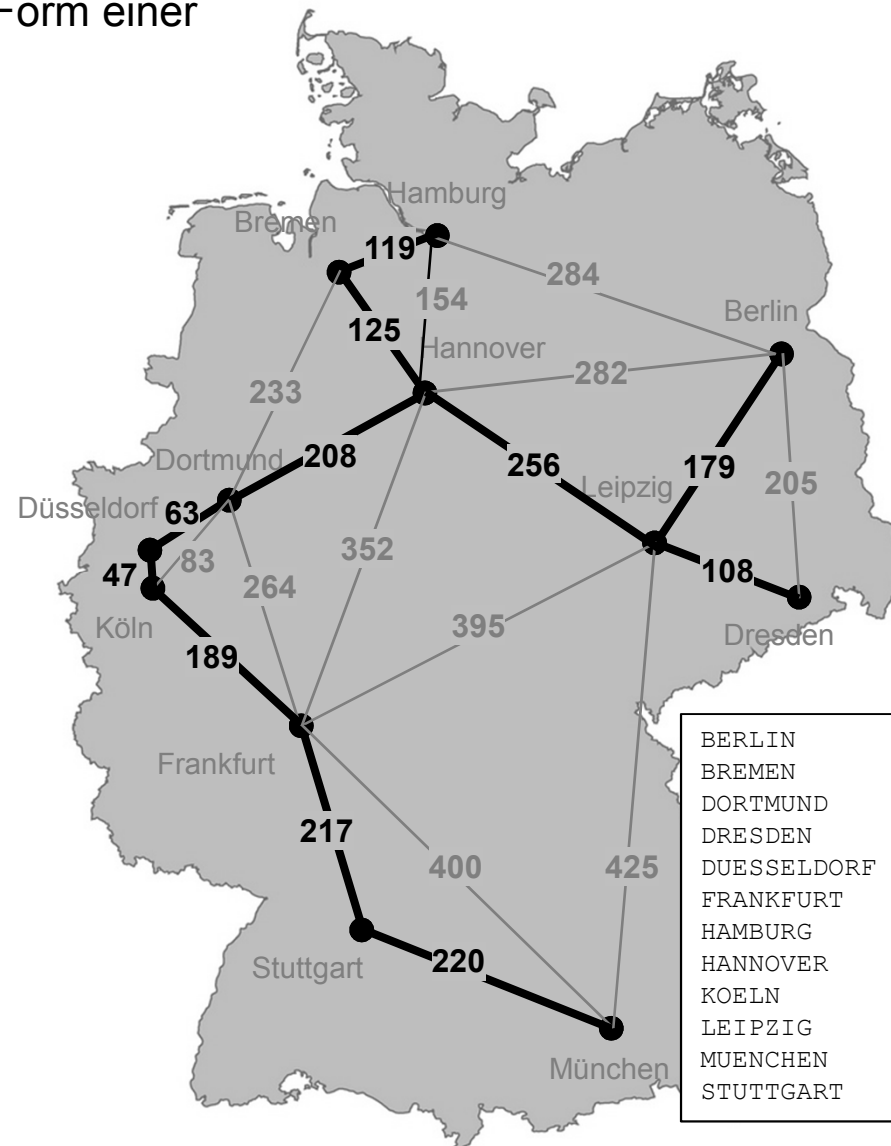
Berlin	Bremen	Dortmund	Dresden	Düsseldorf	Frankfurt	Hamburg	Hannover	Köln	Leipzig	Stuttgart	München	
0	412	488	205	572	555	284	282	569	179	584	634	Berlin
	0	233	470	317	466	119	125	312	362	753	640	Bremen
		0	607	63	264	343	208	83	532	653	451	Dortmund
			0	629	469	502	364	589	108	484	524	Dresden
				0	232	427	292	47	558	621	419	Düsseldorf
					0	495	352	189	395	400	217	Frankfurt
						0	154	422	391	782	668	Hamburg
							0	287	256	639	526	Hannover
								0	515	578	376	Köln
									0	425	465	Leipzig
										0	220	Stuttgart
											0	München

```
int distanz[ ANZAHL][ ANZAHL] =
{
    { 0,412,488,205,572,555,284,282,569,179,584,634},
    {412, 0,233,470,317,466,119,125,312,362,753,640},
    {488,233, 0,607, 63,264,343,208, 83,532,653,451},
    {205,470,607, 0,629,469,502,364,589,108,484,524},
    {572,317, 63,629, 0,232,427,292, 47,558,621,419},
    {555,466,264,469,232, 0,495,352,189,395,400,217},
    {284,119,343,502,427,495, 0,154,422,391,782,668},
    {282,125,208,364,292,352,154, 0,287,256,639,526},
    {569,312, 83,589, 47,189,422,287, 0,515,578,376},
    {179,362,532,108,558,395,391,256,515, 0,425,465},
    {584,753,653,484,621,400,782,639,578,425, 0,220},
    {634,640,451,524,419,217,668,526,376,465,220, 0},
};
```

## Näherungslösung für das TSP – Spannbaum

Einen minimalen Spannbaum haben wir zuvor bereits berechnet. Wir übernehmen das Ergebnis in Form einer Adjazenzmatrix:

```
int spannbaum[ ANZAHL][ ANZAHL] =
{
    {0,0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,0,0,1,1,0,0,0,0},
    {0,0,0,0,1,0,0,1,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,1,0,0,0,0,0,1,0,0,0},
    {0,0,0,0,0,0,0,0,1,0,0,1},
    {0,1,0,0,0,0,0,0,0,0,0,0},
    {0,1,1,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,1,1,0,0,0,0,0,0},
    {1,0,0,1,0,0,0,1,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,1,0,0,0,0,1,0},
};
```



## Näherungslösung für das TSP – Implementierung

Wir erstellen einen Array (`pfad`), um den bei der Tiefensuche konstruierten Weg aufzunehmen:

```
static int pfad[ANZAHL];  
int position;
```

Die globale Variable `position` legt dabei die aktuelle Schreibposition in diesem Array fest.

Die Tiefensuche wird rekursiv implementiert. Als Parameter wird der aktuelle Knoten (`knoten`) und der Knoten, über den man zu diesem Knoten gelangt ist (`herkunft`) mitgegeben. Der aktuelle Knoten wird an der nächsten Schreibposition in den Pfad geschrieben. Danach wird zu allen Folgeknoten im Baum gegangen. Über den Parameter `herkunft` wird dabei verhindert, dass man dabei zu dem Knoten zurück geht, von dem man gekommen ist.

```
void tiefensuche( int knoten, int herkunft)  
{  
    int i;  
  
    pfad[position++] = knoten;  
    for( i = 0; i < ANZAHL; i++)  
    {  
        if( spannbaum[knoten][i] && (herkunft != i))  
            tiefensuche( i, knoten);  
    }  
}
```

Trage den aktuellen Knoten in den Pfad ein und erhöhe die Schreibposition

Gehe zu allen Folgeknoten im Spannbaum, die nicht der Herkunftsknoten sind und fahre dort mit der Tiefensuche fort.

Da immer nur neu erreichte Knoten in den Pfad eingetragen werden, entstehen keine Doubletten in dem Pfad.

## Näherungslösung für das TSP – Hauptprogramm und Ausgabefunktion

Die Funktion `ausgabe` gibt den aktuell im Array `pfad` vorliegenden Weg auf dem Bildschirm aus.

Bei der Ausgabe wird die Länge des Weges berechnet und abschließend ebenfalls ausgegeben.

```
void ausgabe()
{
    int i;
    int d;

    for( i = 0, d = 0; i < ANZAHL; i++)
    {
        printf( "%d-", pfad[i]);
        d += distanz[pfad[i]][pfad[(i+1)%ANZAHL]];
    }
    printf( "%d (%d km)\n", pfad[0], d);
}
```

Im Hauptprogramm werden in einer Schleife alle Knoten einmal als Startpunkt gewählt.

Für jeden Knoten wird, nachdem die Schreibposition zurückgesetzt wurde, die Tiefensuche gestartet und das Ergebnis ausgegeben.

```
void main()
{
    int start;

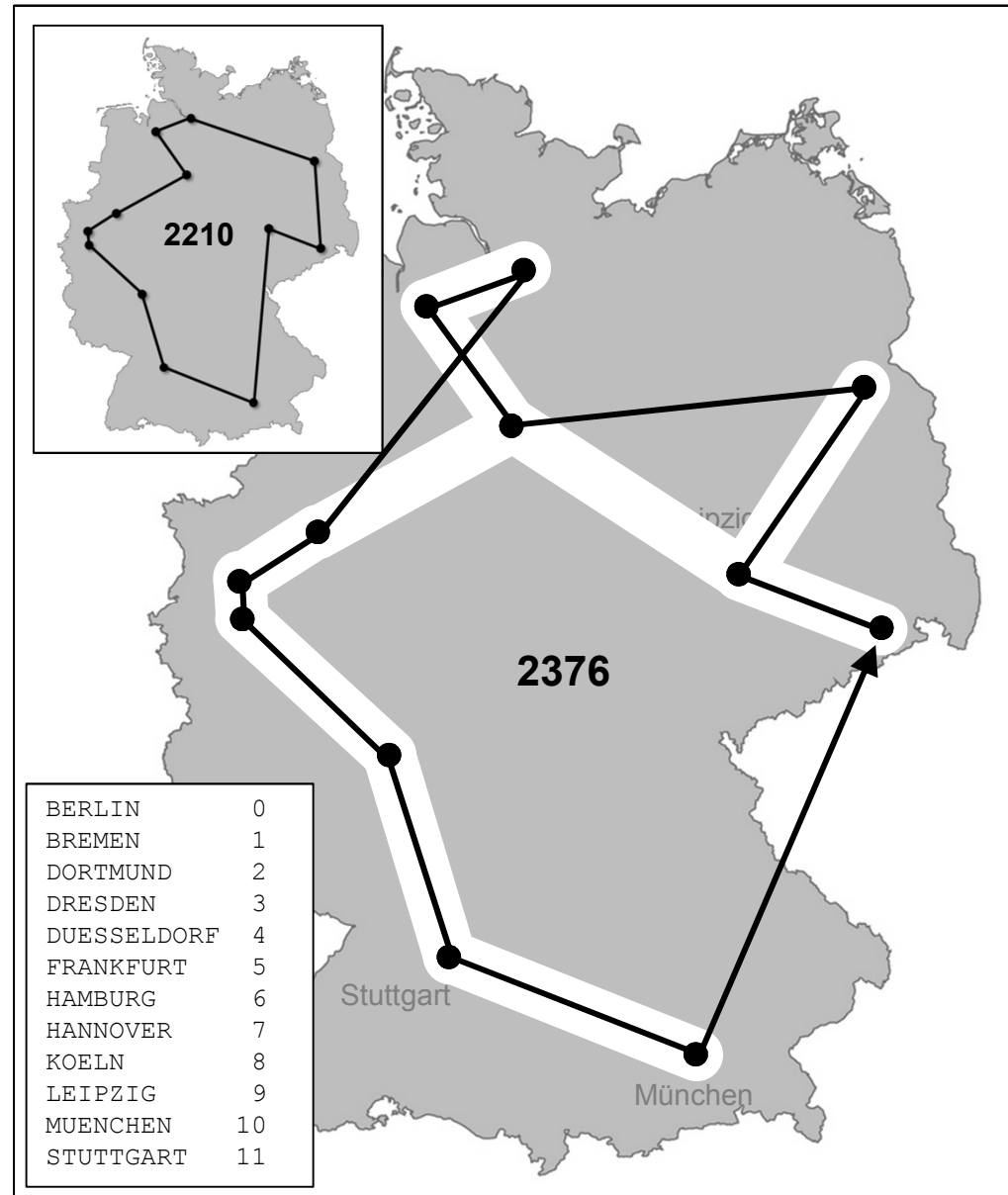
    for( start = 0; start < ANZAHL; start++)
    {
        position = 0;
        tiefensuche( start, -1);
        ausgabe();
    }
}
```

Das Programm berechnet so viele Hamiltonsche Wege, wie der Graph Knoten hat. Unter diesen Wegen wäre dann noch der kürzeste auszuwählen.

## Näherungslösung für das TSP – Ergebnis

Das Programm gibt 12 Hamiltonsche Wege aus, von denen der kürzeste mit 2376 km eine gute Approximation des optimalen Weges (2210 km) ist.

0-9-3-7-1-6-2-4-8-5-11-10-0	(2558 km)
1-6-7-2-4-8-5-11-10-9-0-3-1	(2496 km)
2-4-8-5-11-10-7-1-6-9-0-3-2	(3001 km)
3-9-0-7-1-6-2-4-8-5-11-10-3	(2376 km)
4-2-7-1-6-9-0-3-8-5-11-10-4	(3126 km)
5-8-4-2-7-1-6-9-0-3-11-10-5	(2670 km)
6-1-7-2-4-8-5-11-10-9-0-3-6	(2499 km)
7-1-6-2-4-8-5-11-10-9-0-3-7	(2496 km)
8-4-2-7-1-6-9-0-3-5-11-10-8	(2821 km)
9-0-3-7-1-6-2-4-8-5-11-10-9	(2496 km)
10-11-5-8-4-2-7-1-6-9-0-3-10	(2447 km)
11-5-8-4-2-7-1-6-9-0-3-10-11	(2447 km)



## Näherungslösung für das TSP – Fehlerabschätzung

Im Folgenden sei

- m die Länge des minimalen Hamiltonschen Weges
- s die Länge des minimalen Spannbaums
- h die Länge des durch dieses Verfahren ermittelten Hamiltonschen Weges

Wenn man aus dem minimalen Hamiltonschen Weg eine Kante entfernt, erhält man einen Spannbaum, der kürzer ist als der minimale Hamiltonsche Weg. Daraus folgt, dass der minimale Spannbaum kürzer ist als der minimale Hamiltonsche Weg. Es ist also:  $s \leq m$ .

Wenn man den minimalen Spannbaum in Tiefensuche durchläuft, wird jede Kante maximal zweimal abgefahren. Beim Entfernen der doppelt vorkommenden Knoten wird diese Länge nicht vergrößert, da wir ja vorausgesetzt haben, dass direkte Verbindungen nie länger als Umwege sind. Es gilt also für den mit unserem Verfahren ermittelten Hamiltonschen Weg:  $h \leq 2s$ .

Insgesamt folgt:  $h \leq 2s \leq 2m$

Der aus dem Spannbaum gewonnene Hamiltonsche Weg ist also maximal doppelt so lang wie der kürzeste Hamiltonsche Weg. Damit haben wir eine Route für den Handlungsreisenden gefunden, die maximal doppelt so lang ist, wie die optimale Route. Das mag unbefriedigend sein, aber das Approximationsverfahren hat polynomiale Laufzeit, während die vollständige Lösungssuche exponentielle Laufzeit hat.

Für das TSP gibt es hunderte von Verfahren, die versuchen die Lösungssuche unter speziellen Randbedingungen zu verbessern oder zu beschleunigen, aber keines dieser Verfahren löst das allgemeine Problem in polynomialer Laufzeit.

Das TSP ist vielleicht das am intensivsten untersuchte und am meisten diskutierte Problem der Informatik. Ein Ende dieser Diskussion ist nicht in Sicht.