# 5. Graphen - Algorithmen

# Definitionen

- Ein **Graph** besteht aus einer Menge von Knoten und einer Menge von Kanten.
- **Knoten** sind beliebige Objekte.
- **Kanten** sind ‘Verbindungen’ zwischen den Objekten. Eine Kante verbindet genau zwei Knoten.
- Kanten können **ungerichtet** oder **gerichtet** sein. Gerichtete Kanten werden durch Pfeile dargestellt.
- Ein **Weg** (**Pfad**) von einem Knoten x zu einem Knoten y ist eine lineare Liste von Knoten (Anfang: x, Ende: y), die durch Kanten verbunden sind.
- Ein Graph ist **zusammenhängend**, wenn es zwischen jedem Paar von Knoten einen Pfad gibt.
- In einem **einfachen Pfad** kommt jeder Knoten nur ein einziges Mal vor.
- Ein **Zyklus (Kreis)**, ist ein einfacher Pfad, bei dem Anfangs- und Endknoten identisch sind.
- Ein Graph, der zusammenhängend ist und keine Zyklen enthält, ist ein **Baum**.
- Ein **Wald** ist eine Menge von Bäumen, die nicht zusammenhängen.
- Ein **spannender Baum** eines Graphen ist ein Untergraph, der alle Knoten des Graphen enthält, und ein Baum ist (zusammenhängend, zyklenfrei).
- Kanten können **ungewichtet** oder **gewichtet** sein. Ein Gewicht wird durch Anschrift an die Kante dargestellt.
- Ein **vollständiger** Graph hat eine Kante zwischen jedem Paar von Knoten

  => Zahl der Kanten: $E = \frac{1}{2} V (V-1)$ für ungerichtete Graphen

  $E = V (V-1)$ für gerichtete Graphen

  E = Anzahl der Kanten, V = Anzahl der Knoten
- Ein **lichter (dünn besetzter)** Graph hat relativ wenige Knoten ($E < V \log V$)

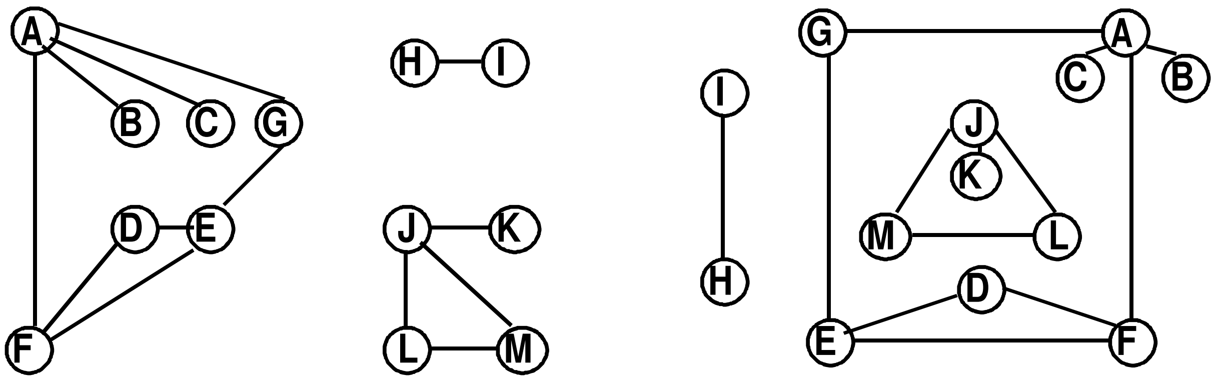
- Ein **dichter** Graph ist fast vollständig

  $E \approx \frac{1}{2} V (V-1)$ für ungerichtete Graphen

  $E \approx V (V-1)$ für gerichtete Graphen

## Repräsentation von Graphen als Diagramm

- Die Knoten sind Kreise.
- Die Kanten sind Linien bzw. Pfeile.
- Die Anordnung der Kreise und Linien im Diagramm ist irrelevant, hat aber manchmal mnemonische Bedeutung (z.B. Städte und Fluglinien: Anordnung gemäß der Geographie.



# Repräsentation von Graphen im Rechner

### 1. Nachbarschaftsmatrix (Adjazenz-Matrix)

Eine Matrix A der Größe VxV (V = Anzahl der Knoten) mit

a[i,j] = 0 : es gibt keine Kante von i nach j

a[i,j] = 1 : es gibt eine Kante von i nach j

Alle a[i,i] = 0 oder 1, je nach besserer Eignung für den Algorithmus oder Semantik des Problems.

**Eigenschaften**

- Die Adjazenzmatrix ist symmetrisch für ungerichtete Graphen und asymmetrisch für gerichtete Graphen.
- Die Matrixelemente können auch Zahlen sein, die die Gewichte der Kanten darstellen.
- Speicherbedarf : O ($V^2$)
- Zeitbedarf für die Initialisierung: O ($V^2$)

=> **für dichte Graphen geeignet**

**2. Nachbarschaftslisten (Adjazenz-Listen)**

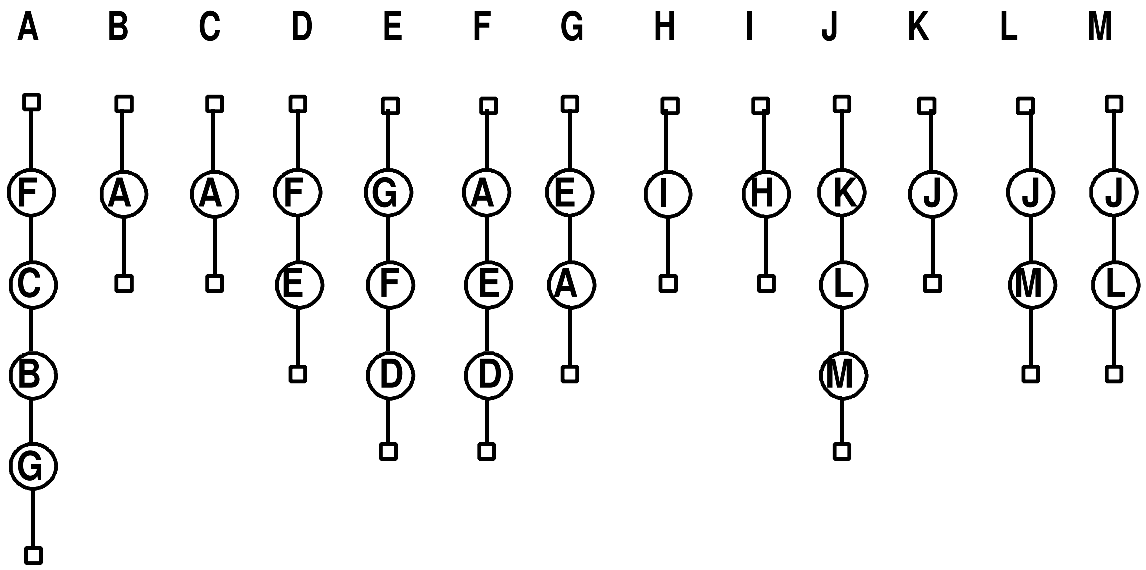
Zu jedem Knoten wird eine lineare Liste angelegt, die alle mit dem Knoten verbundenen Kanten enthält.

**Eigenschaften**

- Jede Kante ist zweifach repräsentiert.
- Gewichte können in die Liste explizit mit aufgenommen werden.
- Listenstruktur und Effizienz der Graphenalgorithmen hängen von der Eingabereihenfolge beim Aufbau der Datenstruktur ab.

=> **für lichte (dünn besetzte) Graphen geeignet**

**Beispiel für Adjazenzlisten**




**Verständnisfrage:** Warum repräsentiert man Graphen nicht in der Weise, daß jeder Knoten durch einen Datensatz dargestellt wird mit je einem Zeigerfeld pro Kante?

# 5.1 Elementare Graphenalgorithmen

Bereits ausführlich behandelt: Baum- Algorithmen

Unterschiede:

| **Baum** | **Graph** |
|---|---|
| zusammenhängend | evtl. nicht zusammenhängend |
| zyklenfrei | evtl. nicht zyklenfrei |
| | evtl. gewichtete Kanten |
| | evtl. gerichtete Kanten |

## Konventionen für die nachfolgenden Programme

Die Knoten werden mit einzelnen Buchstaben A,B,C,... bezeichnet, die in der internen Darstellung auf die Zahlen 1,2,3,... abgebildet werden.

Wir setzen voraus, daß eine Funktion *index* zur Verfügung steht, die Knotennamen in ganze Zahlen zwischen *1* und *V* umwandelt, und eine Funktion *name*, die ganze Zahlen in Knotennamen umwandelt.

Wir lesen zuerst *V* und *E* ein, dann die Knoten und die Kanten.

## Einlesen eines Graphen

```
#define maxV 50
int j, x, y, V, E;
int a[maxV][maxV];

adjmatrix()
  {
    scanf["%d %d\n", &V, &E);
    for (x = 1; x <= V; x++)
      for (y = 1; y <= V; y++) a[x][y] = 0;
    for (x = 1; x <= V; x++) a[x][x] = 1;
    for (j = 1; j <= E; j++)
        {
         scanf("%c %c\n", &v1, &v2);
         x = index(V1); y = index(V2);
         a[x][y] = 1; a[y][x] = 1;
        }
  }
```

## 5.1.1. Tiefensuche

Englisch: Depth First Search

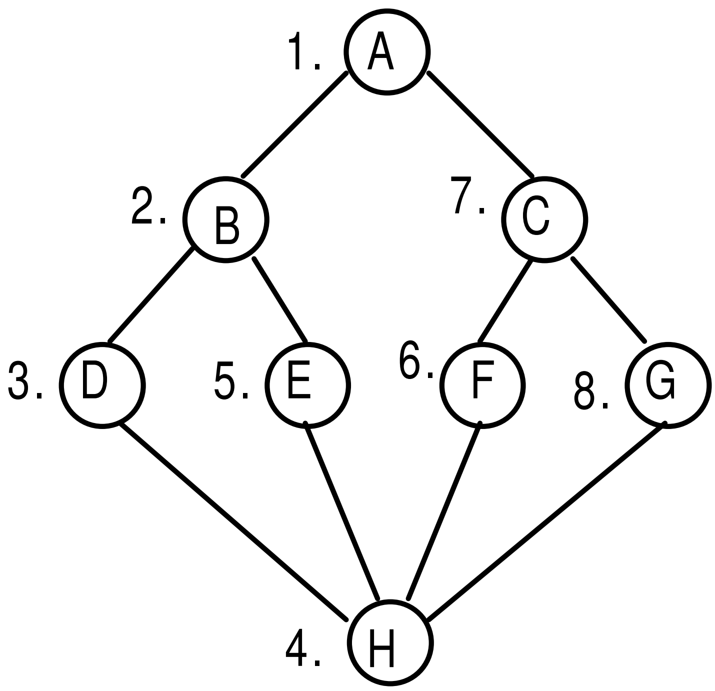
Man besuche systematisch jeden Knoten des Graphen und jede Kante.

**Algorithmus**: Tiefensuche ab Eingabeknoten

- Speichere Knoten auf einem Stack.
- Wiederhole
  - Hole jüngsten Knoten aus dem Stack.
  - Besuche ihn.
  - Speichere alle Nachbarknoten, die noch nicht besucht wurden,

  auf dem Stack

  bis Stack leer ist.



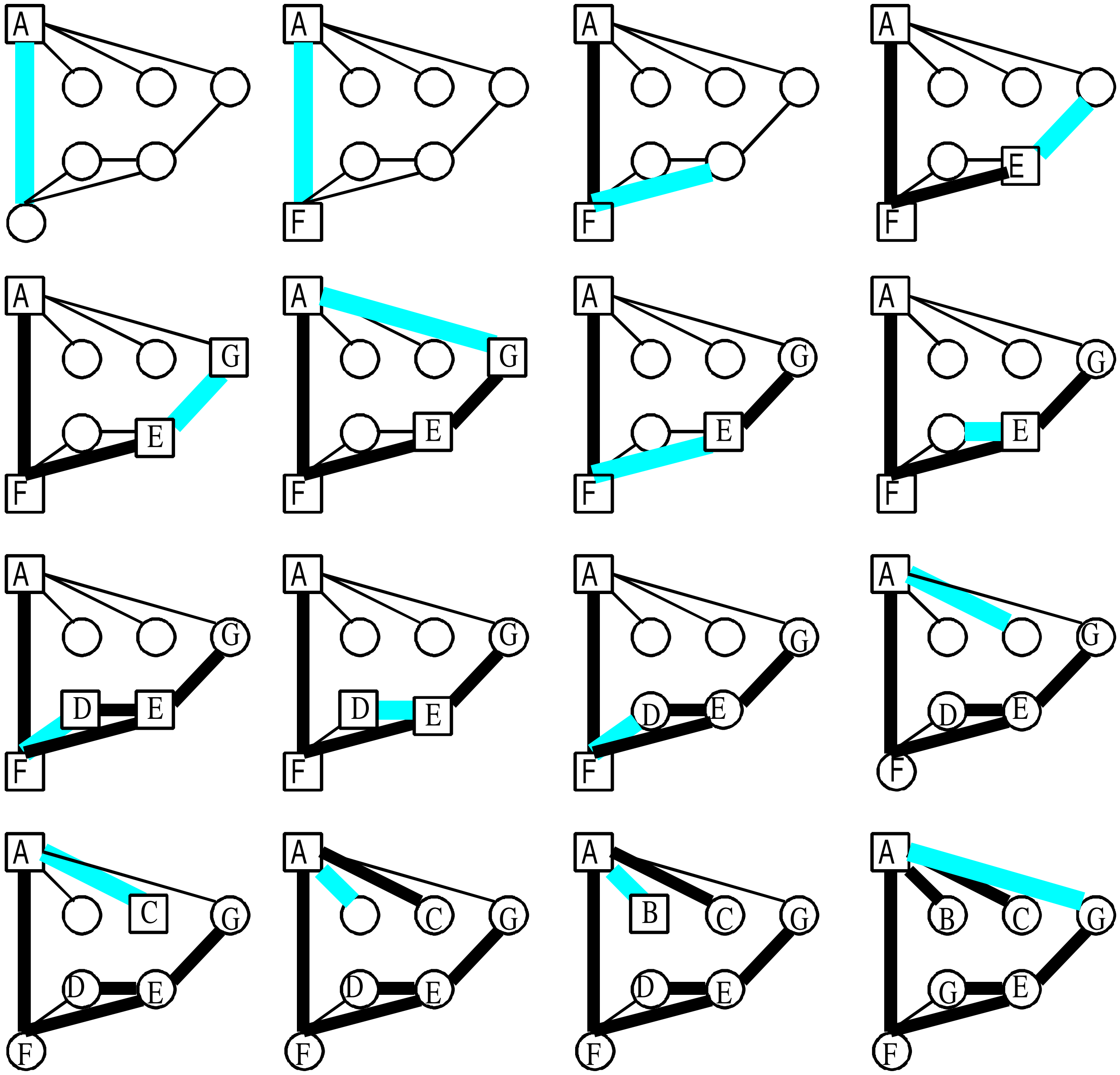
**Programm Tiefensuche (rekursiv)**

Annahme: Repräsentation des Graphen durch Adjazenzlisten

```
int val[maxV]; int id = 0;
visit[int k]
  {
    struct node *t;
    val[k] = ++id;
    for (t = adj[k]; t != z; t = t->next)
      if (val[t->v] == 0) visit (t->v);
   }
listdfs()
   {
     int k;
     for (k = 1, k <= V; k++) val[k] = 0;
     for (k = 1; k <= V; k++)
       if (val[k] == 0) visit (k);
   }
```

- Im Anweisungsteil der Prozedur ‘listdfs’ wird ‘visit (k)’ für jede zusammenhängende Komponente des Graphen einmal aufgerufen.
- Der Graph enthält einen Zyklus, wenn die ‘If’ - Bedingung in ‘visit’ mindestens einmal nicht erfüllt ist.

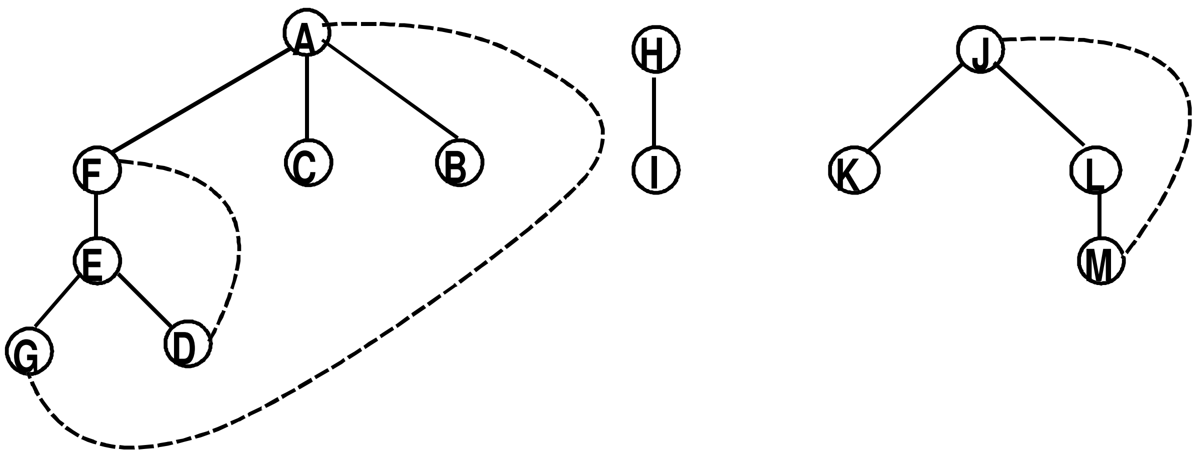
**Beispiel**




| | |
|---|---|
| aktueller Knoten: | "hohles" Quadrat mit Buchstabe |
| aktuelle Kante: | schraffiert |
| abgeschlossener Knoten: | "hohler" Kreis mit Buchstabe |

## Tiefensuchbäume

Die rekursiven Aufrufe der Prozedur *visit* lassen sich durch eine Baumstruktur veranschaulichen:




Der Name "Tiefensuche" rührt daher, daß in diesem Baum **zuerst in die Tiefe** gegangen wird, dann in die Breite.

**Die Tiefensuche ist ein klassisches Beispiel für Backtracking!**

## Komplexität der Tiefensuche auf Adjazenzlisten

- Das Feld `val` hat V Elemente, jedes wird genau einmal gesetzt (jeder Knoten wird genau einmal besucht).
- Jede Kante wird genau zweimal betrachtet.

=> Die Tiefensuche ist O(V+E).

## Tiefensuche auf einer Adjazenzmatrix

Modifikation der Prozedur visit wie folgt:

```
visit (int k)
  {
    int t;
    val[k] = ++id;
    for (t = 1; t <= V; t++)
      if (a[k][t] != 0)
        if (val[t] == 0) visit(t);
  }
```

## Komplexität der Tiefensuche auf einer Adjazenzmatrix

- Jedes Bit in der Adjazenzmatrix wird genau einmal geprüft.

=> Die Tiefensuche ist $O(V^2)$.

## Kommentare

- Die Tiefensuche ist ein einfacher Algorithmus zur Lösung elementarer Probleme, z.B.
  - Stelle fest, ob ein Graph zusammenhängend ist
  - Stelle fest, ob es einen Zyklus gibt
    (visit entdeckt einen von 0 verschiedenen Eintrag in `val`)

## 5.1.2. Breitensuche

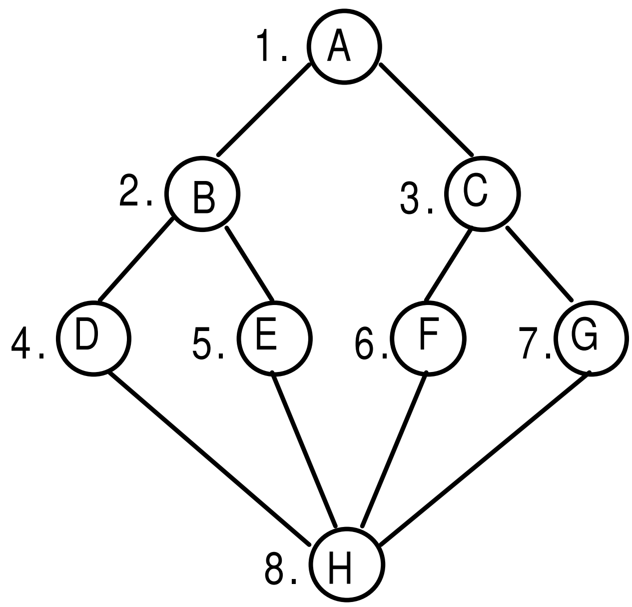
Englisch: Breadth First Search

Man besuche systematisch jeden Knoten des Graphen und jede Kante.

**Algorithmus** Breitensuche ab Eingabeknoten

- Speichere Knoten in einer Warteschlange und markiere ihn als erreicht.
- Wiederhole
  - Hole „ältesten“ Knoten aus der Warteschlange.
  - Besuche ihn.
  - Speichere alle noch nicht erreichten Nachbarknoten in der Warteschlange und markiere sie als erreicht.

bis Warteschlange leer.



**Programm Breitensuche (iterativ)**

Annahme: Repräsentation des Graphen durch Adjazenzlisten.

Iteratives Durchlaufen des Graphen. "Merken" der noch zu verarbeitenden Knoten einer Ebene in einer Warteschlange (queue).

```
int val[maxV]; int id = 0;
visit(int k)
  {
    struct node *t;
    put(k);
    while (!queueempty())
      {
        k = get(); val[k] = ++id;
        for (t = adj[k]; t != z; t= t->next)
          if (val[t->v] == 0)
            { put(t->v); val[t->v] = -1 }
       }
  }
listbfs()
  {
    int k;
    queueinit();
    for (k = 1; k <= V; k++) val[] = 0;
    for (k = 1; k <= V; k++)
      if (val[k] == 0) visit[k];
  }
```

Im Programm wird das Feld `val` wie folgt verwendet:

`val[i] = 0` : Knoten noch nicht gefunden.

`val[i] = -1` : Knoten noch zu bearbeiten (in der Queue).

`val[i] = id` : Knoten fertig bearbeitet.

## Breitensuche vs. Tiefensuche

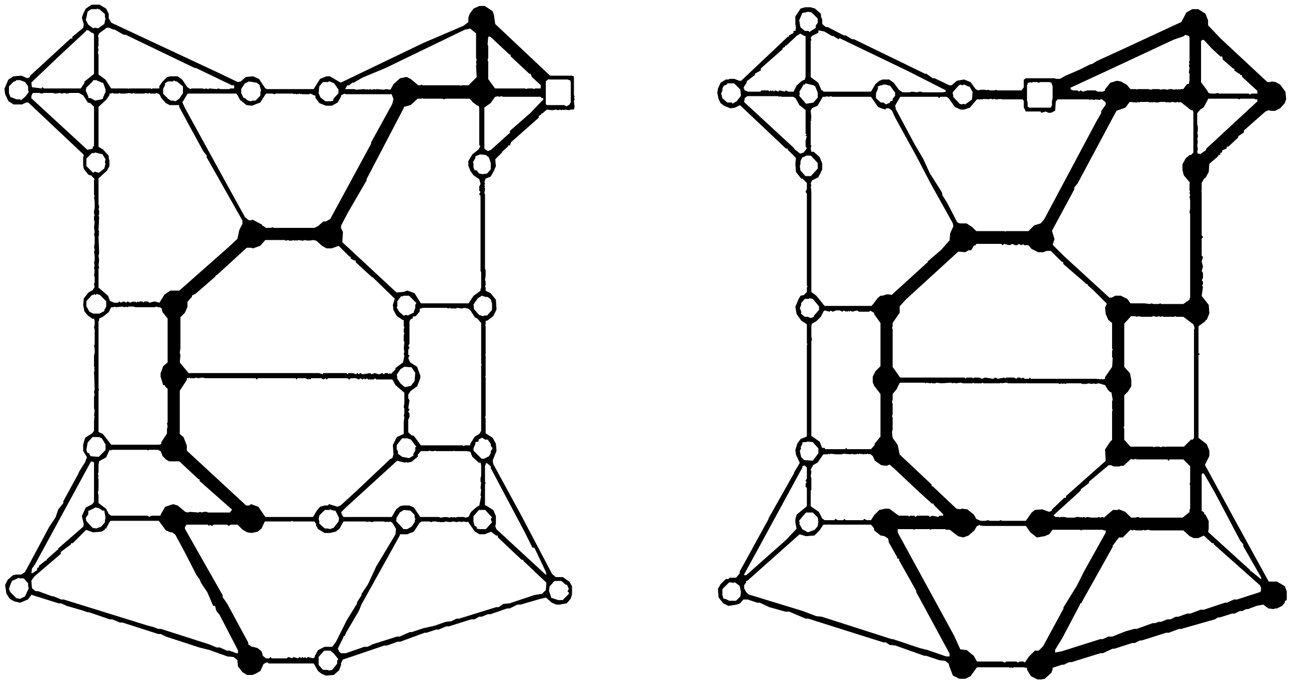
Bei der Breitensuche werden zuerst alle unmittelbaren Nachbarn eines Knoten fertig verarbeitet, bevor eine neue Ebene (mit Entfernung +1 vom Startknoten) hinzugenommen wird. Der Suchbaum wird zuerst in die Breite hinein abgearbeitet.

Bei beiden Algorithmen können wir uns die Knoten als in drei Klassen eingeteilt vorstellen: *Baum*knoten (oder *besuchte* Knoten), welche aus der Datenstruktur entnommen worden sind; Randknoten, die Baumknoten benachbart, jedoch noch nicht besucht worden sind; sowie unsichtbare Knoten, die überhaupt noch nicht angetroffen worden sind. Falls jeder Baumknoten mit der Kante verbunden wird, die bewirkt hat, daß er zur Datenstruktur hinzugefügt wurde, so bilden diese Kanten einen Baum.

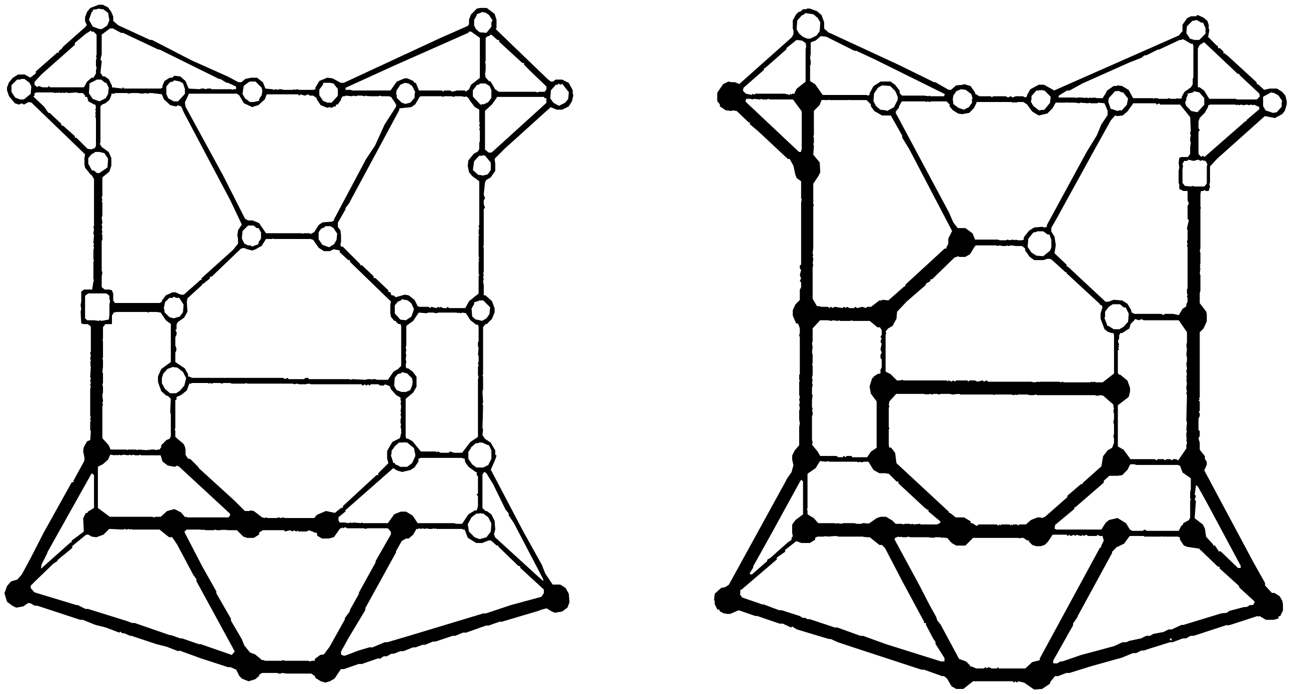
Um eine zusammenhängende Komponente eines Graphen systematisch abzusuchen (eine Prozedur *visit* zu implementieren), beginnen wir mit einem Knoten am Rand, während alle anderen unsichtbar sind, und führen den folgenden Schritt so oft aus, bis alle Knoten besucht worden sind: "Bewege einen Knoten (sagen wir *x*) vom Rand zum Baum, und nehme alle dem Knoten *x* benachbarten unsichtbaren Knoten in den Rand auf." Verfahren zur Traversierung von Graphen unterscheiden sich darin, wie entschieden wird, welcher Knoten vom Rand zum Baum bewegt werden soll. Bei der Teifensuche möchten wir den Knoten aus dem Rand auswählen, der als letzter angetroffen wurde; dies entspricht der Berwendung eines Stapels zum Speichern der Knoten, die zum Rand gehören. Bei der Breitensuche möchten wir den Knoten aus dem Rand auswöhlen, dessen Antreffen am weitesten zurückliegt; dies entspricht der Verwendung einer Warteschlange zum Speichern der zum Rand gehörenden Knoten.

# Tiefensuche vs. Breitensuche, Beispiel

## Tiefensuche



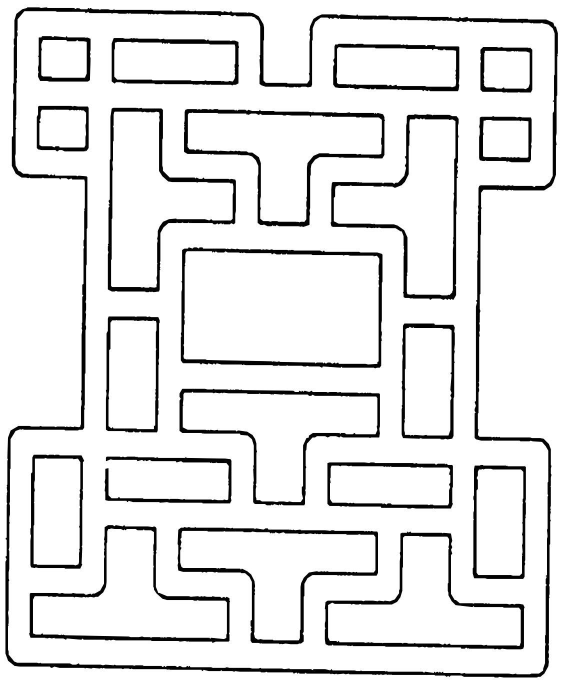
## Breitensuche

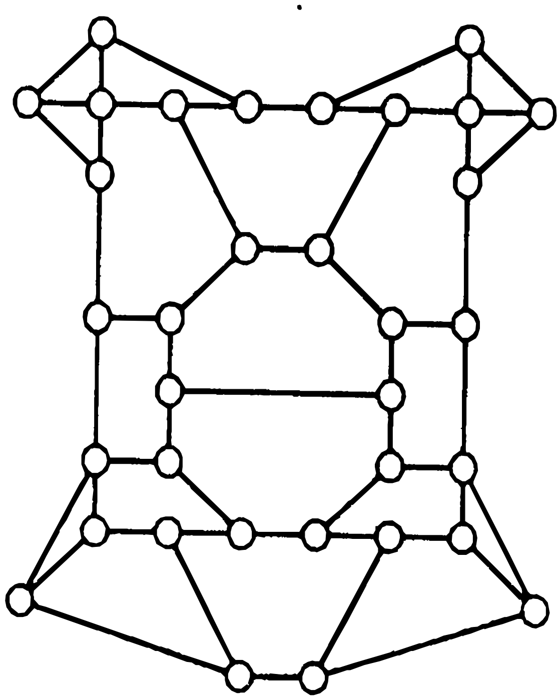


## Labyrinthe

Ein **Labyrinth** ist ein ungewichteter, ungerichteter Graph mit einem ausgezeichneten Eingangsknoten E und einem ausgezeichneten Ausgangsknoten A. Gesucht ist ein Pfad zwischen E und A.

### Beispiel





### Algorithmus

### Tiefensuche!

Dies entspricht einer Person, die immer vorwärtsgerichtet den Ausgang sucht, beim Festellen einer Sackgasse zur letzten, noch nicht fertig ausprobierten Kreuzung zurückkehrt und dort eine Alterative probiert (**Backtracking**).

# 5.2 Konnektivität und Erreichbarkeit

## 5.2.1 Mehrfach zusammenhängende Graphen

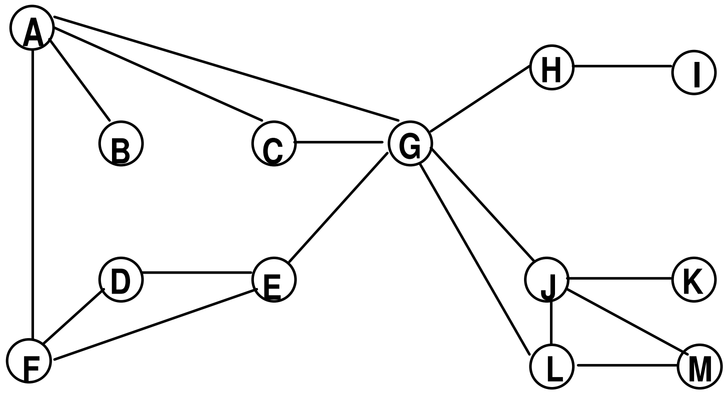
Für viele praktische Anwendungen ist es von Interesse, Graphen zu konstruieren, die **mehrfach zusammenhängend** sind. Man sagt:
sie haben eine Konnektivität > 1.

Ein Graph heißt **zweifach zusammenhängend**, wenn ein beliebiger Knoten entfernt werden kann, ohne daß der restliche Graph in unzusammenhängende Komponenten zerfällt.

Ein **Schnittpunkt** (oder auch Gelenkpunkt) eines Graphen ist ein Knoten, ohne den der restliche Graph mehr disjunkte Teilgraphen besitzt als der ursprüngliche.

Ein zusammenhängender Graph ist zweifach zusammenhängend, wenn er keinen Schnittpunkt (Gelenkpunkt) besitzt.

**Beispiel**

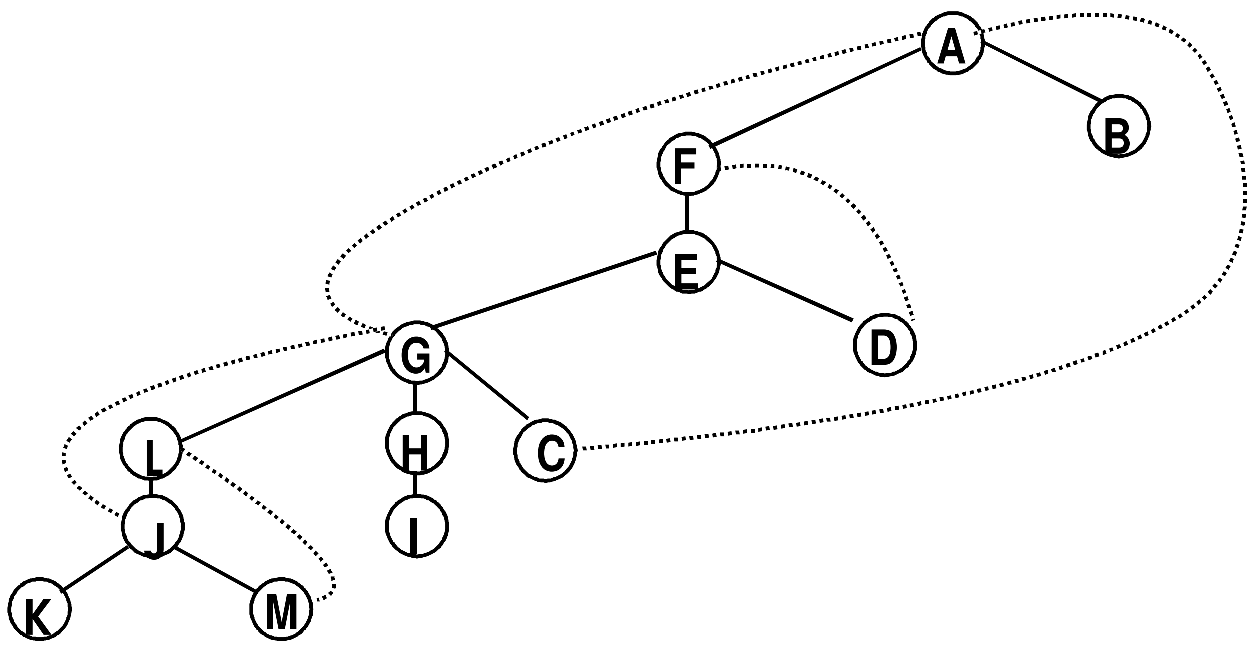


**Kriterien für Schnittpunkte (Gelenkpunkte)**

Basierend auf dem Tiefensuchbaum des Graphen

- **Wurzel**: Ist Schnittpunkt, wenn die Zahl der Kinder > 1 ist.
- **andere Knoten**: Sind Schnittpunkte, wenn mindestens ein Kind existiert, das keine Verbindung über eine gestrichelte Linie zu einem Vorfahren des Knotens weiter oben im Baum hat.

**Tiefensuchbaum zum Beispiel**



**Programm** Zweifacher Zusammenhang

Abwandlung des Programms zur Tiefensuche derart, daß der höchste Punkt im Baum (kleinste Wert von `val`) ermittelt wird, der während der Tiefensuche angetroffen wird:

```
int visit(int k)
  {
    struct node *t;
    int m, min;
    val[k] = ++id;
    min = id;
    for (t = adj[k]; t != z; t = t->next)
      if (val[t->v] == 0)
        {
          m = visit(t->v);
          if (m < min) min = m;
          if (m >= val[k])
             printf("%c ", name (k));
        }
      else if (val[t->v] < min) min = val[t->v];
    return min;
  }
```

**Zweifacher Zusammenhang, Komplexität**

Eine einfache Erweiterung der Tiefensuche, deshalb:
Der zweifache Zusammenhang kann mit Aufwand O(V+E) festgestellt werden, wenn der Graph in Form von Adjazenzlisten gespeichert ist, und mit Aufwand $O(V^2)$, wenn er als Adjazenzmatrix gespeichert ist.

**Kommentare**

Zweifacher Zusammenhang ist vor allem nützlich, wenn Fehlertoleranz gefordert ist, zum Beispiel in Rechnernetzen oder Telefonnetzen im Weitverkehr oder in elektronischen Schaltungen.

Als analoge Erweiterung ist n-facher Zusammenhang (eine Konnektivität von n) definiert.

## 5.2.2 Erreichbarkeit

Die Algorithmen zur Tiefensuche und zur Breitensuche konnten zur Feststellung des Zusammenhangs von Graphen eingesetzt werden.

Wird der Graph als Adjazenzmatrix A gespeichert, so gibt es eine analytische Methode zur Berechnung von (indirekter) Erreichbarkeit: Wir bilden das Matrix-Produkt

$$A' = A^2 = \begin{pmatrix} a_{11}a_{12}....a_{1n} \\ a_{21}a_{22}...a_{2n} \\ \\ a_{n1}a_{n2}...a_{nn} \end{pmatrix}^2$$

mit

$$a'ij = \sum_{k=1}^{n} a_{ik}a_{kj}$$

$a'ij = 1$ genau dann, wenn es einen Pfad der Länge 2 von Knoten i nach Knoten j gibt; sonst $a'ij = 0$.

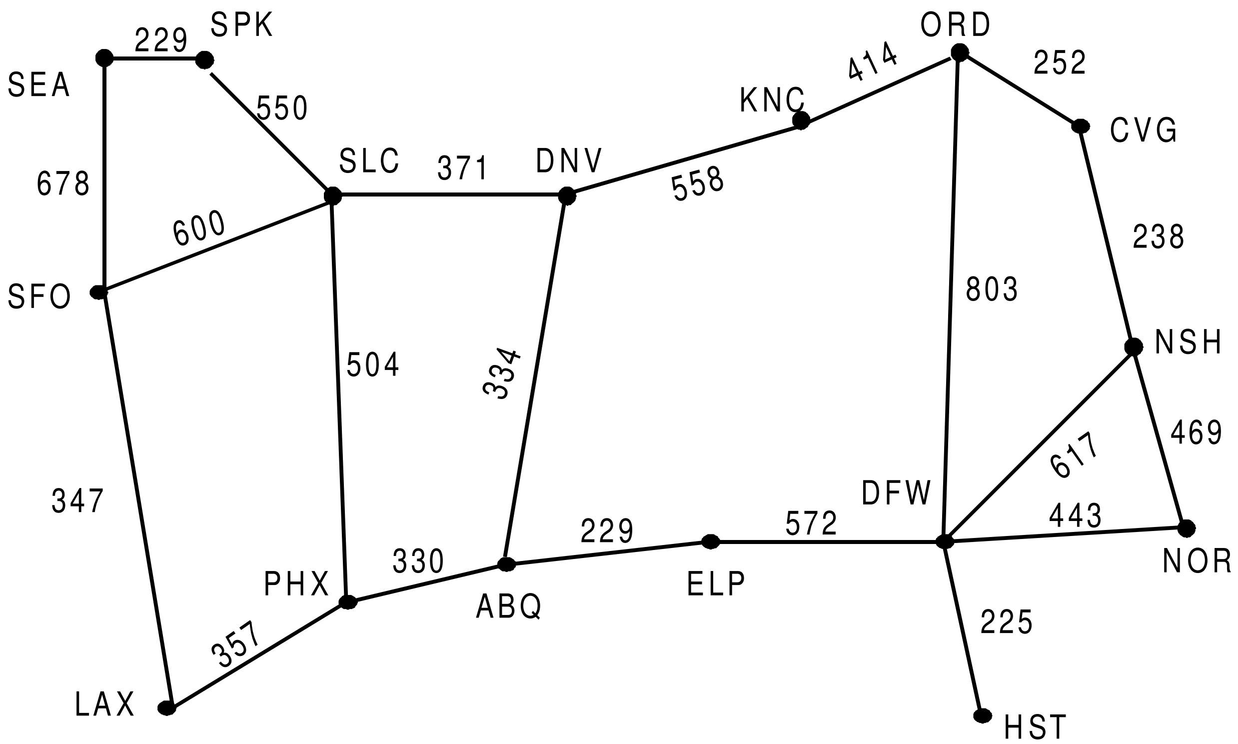
**Erreichbarkeit mit Pfadlänge m**

Generell gilt für die m-te Potenz der Adjazenzmatrix A:

$$A^m_{ij} = \begin{cases} 1, & \text{wenn es einen Pfad der Länge m von i nach j gibt} \\ 0 & \text{sonst} \end{cases}$$

**Beispiel**

Fluglinien in den USA



## Adjazenzmatrix zum Fluglinienbeispiel

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 5 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 9 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 15 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

### Übungsaufgabe

Man berechne, welche Städte von SFO aus in 4 Teilstrecken erreichbar sind.

# 5.3 Gewichtete Graphen

Zuweisung eines Gewichts an jede Kante. Repräsentiert beispielweise eine Entfernung, eine Leitungskapazität, eine Zeitdauer.

**Fragenstellungen beispielsweise:**

- Wie werden alle Knoten eines Graphen mit minimalen Kosten untereinander verbunden?
- Wie werden zwei bestimmte Knoten eines Graphen mit minimalen Kosten verbunden?

## 5.3.1 Minimale spannende Bäume

**Definition Spannender Baum (Spannbaum)**
**Definition:**

> Gegeben sei ein gewichteter, ungerichteter Graph. Dann ist ein spannender Baum (oder Spannbaum) ein Baum aus Knoten und Kanten des Graphen, der alle Knoten des Graphen enthält.
> Wie alle Bäume ist der spannende Baum zyklenfrei. Man sagt: "Der Baum spannt den Graphen auf".

**Definition:**

Ein minimaler spannender Baum (minimal spanning tree, MST) ist ein spannender Baum mit minimalem Gesamtgewicht.

# Der MST - Algorithmus von Prim (1957) oder Dijkstra

## Algorithmus MST- Prim

Beginne mit einem beliebigen **Knoten** und füge jeweils den nächsten Knoten hinzu, der mit der bisherigen Lösung durch eine Kante minimalen Gewichts verbunden ist, sofern kein Zyklus entsteht.

**Algorithmus Prioritätssuche auf einem Graphen**

- Beginne mit irgendeinem Knoten als Baum.
- Füge jeweils den Knoten zum Baum hinzu, der aus der Liste der bekannten Randknoten die nächste Priorität hat.

Auswahl des nächsten Knotens erfolgt anhand eines problemabhängigen Kriteriums; Knoten haben verschiedene Priorität,

**deshalb:**

Lösung des Problems mit dem gleichen iterativen Algorithmus wie bei Tiefen/ Breitensuche; statt Stack bzw. Warteschlange: Prioritätswarteschlangen

**Operationen auf der Prioritätswarteschlange:**

- Einsetzen eines neuen Elements
- Auslese des Elements höchster Priorität
- Änderung der Priorität eines Elements

**Motivation für Graphenalgorithmen mit Prioritätswarteschlange**

Die Suche in einem Graphen kann ganz allgemein mit Hilfe einer Einteilung der Knoten in drei Mengen beschrieben werden.

Wir erinnern daran, daß die Suche in Graphen mit Hilfe einer Einteilung der Knoten in drei Mengen beschrieben werden kann: *Baum*knoten, bei denen alle Kanten untersucht worden sind, *Rand*knoten, die sich in einer Datenstruktur befinden, die auf ihre Verarbeitung wartet, und *unsichtbare* Knoten, die noch gar nicht berührt worden sind. Das grundlegende Verfahren der Suche in Graphen, das wir verwenden, beruht auf dem Schritt "bewege einen Knoten (der mit *x* bezeichnet werde) vom Rand zum Baum, und nimm dann alle unsichtbaren Knoten, die zu *x* benachbart sind, in den Rand auf". Wir benutzen den Begriff *Prioritätssuche*, um die allgemeine Strategie zu bezeichnen, die darin besteht, eine Prioritätswarteschlange zu benutzen, um zu entscheiden, welcher Knoten vom Rand zu nehmen ist. Dies ermöglicht eine hohe Flexibilität. Wie wir sehen werden, unterscheiden sich verschiedene klassische Algorithmen (einschließliche Tiefen- und Breitensuche) nur hinsichtlich der Wahl der Priorität.
Für die Berechnung des minimalen Spannbaumes sollte die Priorität jedes zum Rand gehörenden Knoten die Länge der kürzesten Kante sein, die ihn mit dem Baum verbindet.

**Programm** Prioritätsuche auf einem Graphen

```
visit (int k)
  {
    stuct node *t;
    if (pqupdate(k, unseen) != 0) dad[k] = 0;
    while (!pqempty())
      {
        id++; k = pqremove(); val[k] = -val[k];
        if (val[k] == unseen) val[k] = 0;
        for (t = adj[k]; t != z; t = t->next)
          if (val[t->v] < 0)
            if [pqupdate(t->v, priority))
              {
                val[t->v] = -(priority);
                dad[t->v] = k
              }
      }
  }
listpfs()
  {
    int k;
    pqinitialize(),
    for (k = 1; k <= V, k++) val[k] = -unseen;
    for (k = 1; k <= V, k++)
      if (val[k] == -unseen) visit[k];
  }
```

**Erläuterungen zum Programm:**

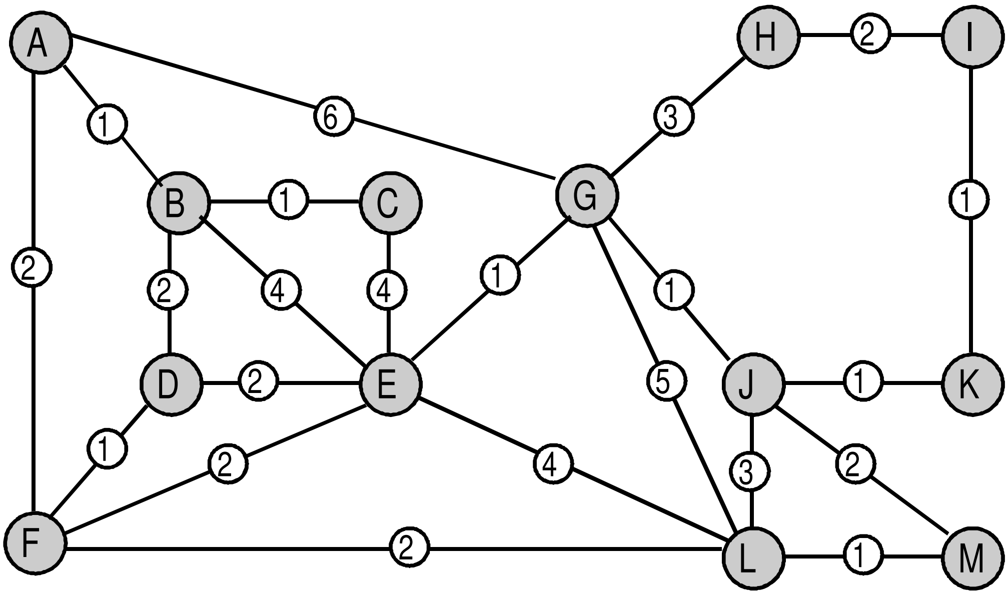
Wert eine Feldelements in `val`

- unseen < 0......... wurde noch nicht gesehen
- Prio < 0......... in der Priotitätswarteschlange
- Prio > 0......... bearbeitet
- 0 .......... bearbeitet, Wurzel

Zwei Änderungen im Vergleich zur iterativen Tiefen- oder Breitensuche:

Erstens sind zusätzliche Maßnahmen erforderlich, wenn eine Kante angetroffen wird, die bereits dem Rand angehört. Bei Tiefen- und Breitensuche werden solche Kanten ignoriert, doch im obigen Programm müssen wir prüfen, ob die neue Kante die Priorität verringert. Dadurch wird gewährleistet, daß wir als nächstes immer denjenigen Knoten des Randes besuchen, der dem Baum am nächsten liegt. Zweitens erfolgt dieses Programm explizit der Erzeugung des Baumes, indem das Feld `dad` geführt wird, in welchem der Vorgänger jedes Knotens im Prioritätssuchbaum gespeichert wird (der Name des Knotens, der bewirkt hat, daß er vom Rand zum Baum bewegt wurde.) Außerdem ist für jeden Knoten `k` im Baum `val[k]` das Gewicht der Kante zwischen `k` und `dad[k]`. Randknoten sinde wie zuvor durch negative Werte von `val` markiert; unsichtbare Knoten sind mittels der Marke `-unseen` anstelle von 0 markiert.

# Ermittlung des Spannbaums mit Prioritätssuche, Beispiel



## Prioritätssuche, Beispiel

Entwicklung der Prioritätswarteschlange für das Beispiel

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C 1 | | | | | | | | | | |
| | | D 2 | D 2 | | | M 1 | | | | | | |
| | B 1 | F 2 | F 2 | F 1 | L 2 | E 2 | E 2 | | | | | |
| | F 2 | E 4 | E 4 | E 2 | E 2 | J 3 | J 2 | G 1 | J 1 | K 1 | I 1 | |
| A · | G 6 | G 6 | G 6 | G 6 | G 6 | G 5 | G 5 | J 2 | H 3 | H 3 | H 3 | H 2 |

## Prioritätssuche, Komplexität

Bei Verwendung von Adjazenslisten entspricht der Algorithmus dem für die Tiefensuche mit der Modifikation, daß zusätzlich die Prioritätswarteschlange, also ein Heap, verwaltet werden muß. Dies erfordert bekanntlich einen Aufwand von logV für jede Operation.

Die Prioritätssuche für einen Graphen mit E Kanten und V Knoten läuft mit Aufwand

$$0 ((E+V) \log V).$$

## Algorithmus von Kruskal

- Sortiere alle Kanten nach ihrem Gewicht
- Beginne mit der Kante mit dem kleinsten Gewicht
- Wiederhole

  Wähle die nächst "schwerere" Kante

  Falls diese keinen Zyklus erzeugt,

  füge sie dem minimalen Spannbaum hinzu

  bis alle Knoten erreicht sind

Dieser Algorithmus wurde bereits in Kapitel 2 vorgestellt. Er ist ein typischer Vertreter der Klasse GREEDY.

## Algorithmus von Kruskal, Komplexität

- Die Sortierung der Kanten kann mit Aufwand 0 (E logE) erfolgen.
- Es gibt Verfahren zum Erkennen von Zyklen mit Aufwand 0 (E logE).

=> Der Algorithmus von Kruskal erfordert 0 (E logE) Operationen.

## 5.3.2 Kürzeste Pfade in gewichteten Graphen

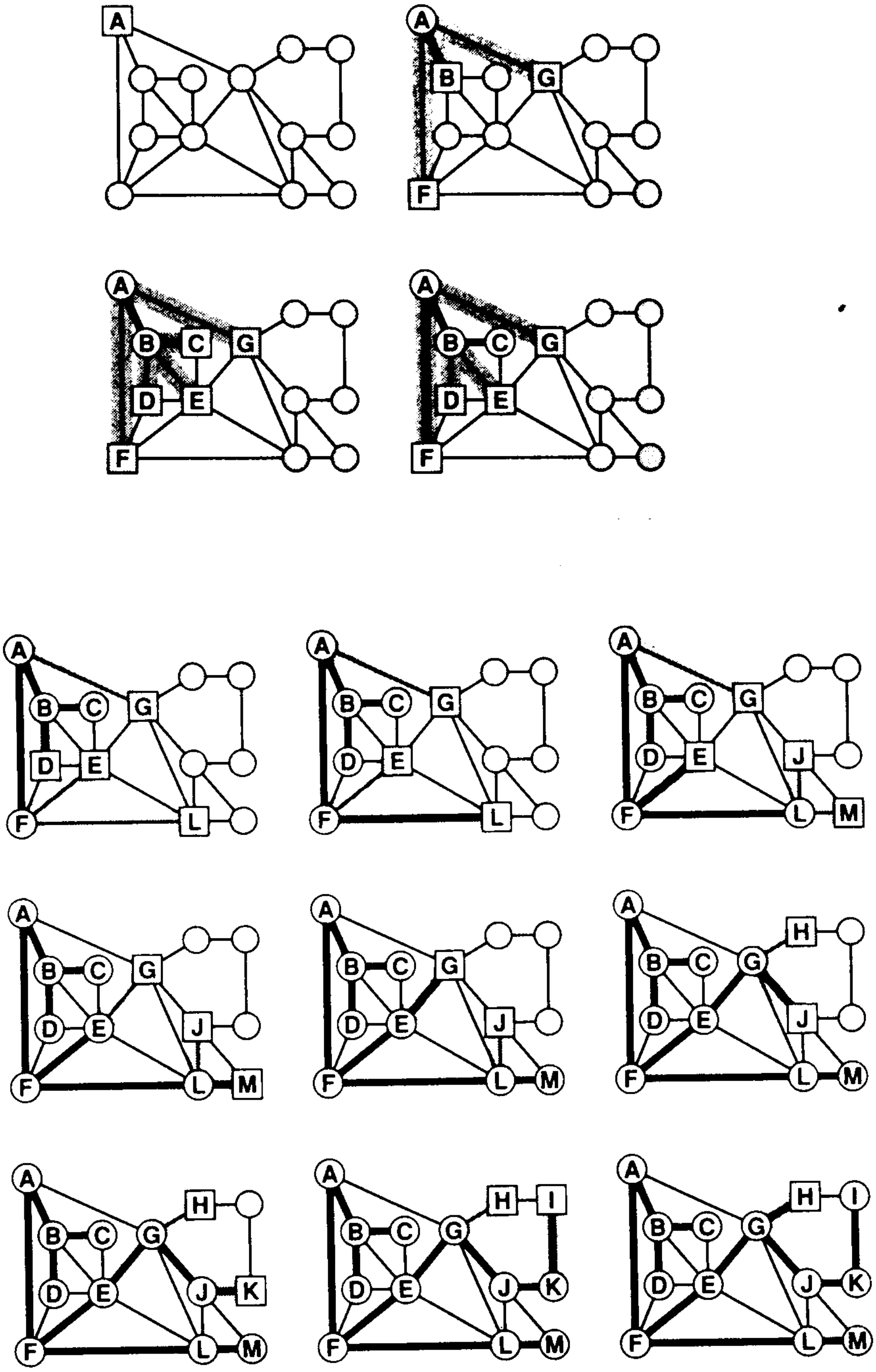
**Algorithmus Kürzester Pfad (von Dijkstra)**

- Beginne mit dem Ausgangsknoten *x*.
- Wiederhole bis der Zielknoten *y* erreicht ist.

  Füge den Knoten hinzu, der zu dem Ausgangsknoten minimalen Abstand hat.

Wenn wir für einen bestimmten Ausgangsknoten *x* die kürzesten Pfade zu **allen** Zielknoten *y* suchen, so entspricht dies der Konstruktion eines Spannbaums, beginnend mit dem Knoten *x*. Der Algorithmus von Dijkstra ist somit fast identisch mit dem Algorithmus von Prim. Man erhält aber für jeden Ausgangsknoten einen anderen Spannbaum, und dieses sind nicht notwendigerweise minimale Spannbäume des Graphen.

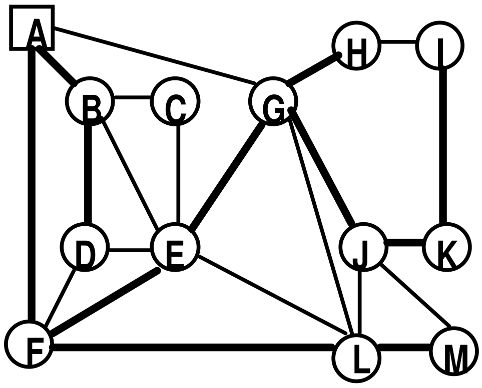
## Spannbaum der kürzesten Pfade, Beispiel

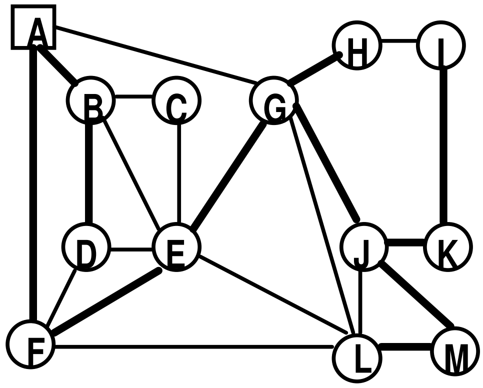


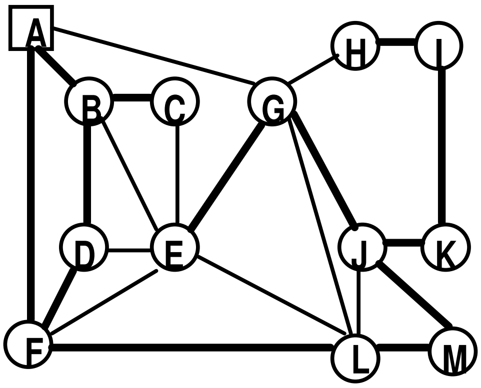
## Kürzeste Pfade, Programm

Die Lösung dieses Problems mittels Prioritätsuche ist mit der Lösung für den minimalen Spannbaum praktisch identisch: Wir erzeugen den Baum für den Knoten *x*, indem wir bei jedem Schritt den Knoten vom Rand hinzufügen, der *x* am nächsten liegt (zuvor fügten wir den Knoten hinzu, der **dem Baum** am nächsten liegt.) Um zu ermitteln, welcher Randknoten *x* am nächsten liegt, verwenden wir das Feld `val`: Für jeden Baumknoten `k` ist `val[k]` der kürzeste Abstand (welcher aus Baumknoten bestehen muß) von diesem Knoten zu *x*. Wenn `k` dem Baum hinzugefügt wird, aktualisieren wir den Rand, indem wir die Adjazenzliste von `k` durchlaufen. Für jeden Knoten `t` in dieser Liste hat der kürzeste Abstand von `t->v` über `k` zu *x* den Wert `val`[k]+t->v. Daher läßt sich der Algorithmus leicht implementieren, indem man diese Größe für `priority` in dem Programm für die Traversierung von Graphen mit Prioritätssuche verwendet.

## Drei Beispiele für Spannbäume der kürzesten Pfade




Ausgangsknoten: A, G, M

Man betrachte, daß dies nicht notwendigerweise minimale Spannbäume des Graphen sind!

## Spannbaum der kürzesten Pfade, Komplexität

Die Prioritätssuche in lichten Graphen berechnet den Baum der kürzesten Pfade in O((E+V) logV) Schritten.

## Varianten für dichte Graphen

Für dichte Graphen eignet sich die interne Darstellung mit einer Adjazenzmatrix besser als eine solche mit Adjazenzlisten. Für einen mit Hilfe einer Adjazenzmatrix dargestellten Graph ist es am besten, für die Prioritätswarteschlange eine Darstellung als ungeordnetes Feld zu benutzen, um für jeden Algorithmus für die Traversierung eines Graphen mit Prioritätssuche eine zu $V^2$ proportionale Laufzeit zu erreichen. Dies wird realisiert, indem die Schleife zur Bestimmung des Minimums kombiniert werden: Jedesmal, wenn wir einen Knoten vom Rand entfernen, durchlaufen wir alle Knoten, wobei wir bei Bedarf ihre Priorität aktualisieren und den gefundenen minimalen Wert registrieren. Dies liefert einen linearen Algorithmus für die Prioritätssuche (und folglich für die Probleme des minimalen Spannbaumes und des kürzesten Pfades) für dichte Graphen.

Wir implementieren die Operationen auf der Prioritätswarteschlange direkt; sie wird im Feld `val` gespeichert.

## Programm für dichte Graphen

```
matrixpfs()
  {
    int k, t, min = 0;
    for (k = 1; k <= V; k++
      { val[k] = -unseen; dad[k] = 0; }
    val[0] = -(unseen+1);
    for (k = 1; k !=0; k = min, min = 0)
      {
        val[k] = -val[k];
        if (val[k] == unseen) val[k] = 0;
        for (t =1; t <= V; t++)
          if (val[t] < 0)
            {
              if (a[k][t] && (val[t] < -priority))
                { val[t] = -priority; dad[t] = k; }
              if (val[t] > val[min]) min = t;
            }
      }
  }
```

Beachten Sie, daß *unseen* etwas kleiner als *maxint* sein muß, da *unseen +1* als Marke für das Auffinden des Minimums verwendet wird und dieser Wert mit negativem Vorzeichen darstellbar sein muß.
Wenn wir die Gewichte in der Adjazenzmatrix speichern und anstelle von *priority val[k]+a[k][t]* benutzen, erhalten wir den Algorithmus von Dijkstra für das Problem des kürzesten Pfades. Wie oben erhalten wir, wenn wir den Programmabschitt zur Registrierung von *id* als Anzahl der bisher durchsuchten Knoten einfügen und *V-id* anstelle von *priority* verwenden, die Tiefensuche; wenn wir *id* verwenden, erhalten wir die Breitensuche.

## Minimaler Spannbaum, kürzeste Pfade, Komplexität bei Adjazenzmatrix-Darstellung

Aus der Betrachtung des Programms ist unmittelbar ersichtlich, daß die Laufzeit im ungünstigsten Fall proportional zu $V^2$ ist. Jedesmal, wenn ein Knoten besucht wird, erfüllt ein Durchlaufen der $V$ Einträge in seiner Zeile in der Adjazenzmatrix den dualen Zweck der Prüfung aller benachbarten Kanten und der Aktualisierung und des Auffindens des nächsten minimalen Wertes in der Prioritätswarteschlange. Daher ist die Laufzeit linear, wenn $E$ zu $V^2$ proportional ist.

Dies ist bei dichten Graphen der Fall.

Die Komplexität der Algorithmen ist daher

$$O(V^2) \approx O\,(E).$$

## Spannbaum der kürzesten Pfade, Kommentare

- Viele praktische Anwendungen, zum Beispiel beim Routing in Weitverkehrsnetzen oder in endlichen Automaten
- Auch wenn die Gewichte alle 1 betragen, ist das Problem dennoch interessant: Es lautet dann, daß der $x$ und $y$ verbindende Pfad zu finden ist, der eine minimale Anzahl von Kanten enthält. Außerdem haben wir bereits einen Algorithmus betrachtet, der dieses Problem löst: Breitensuche. Mittels Induktion läßt sich leicht zeigen, daß, bei einer in $x$ beginnenden Breitensuche zuerst alle Knoten besucht werden, die von $x$ aus über eine Kante erreicht werden können, dann alle Knoten, die von $x$ aus über zwei Kanten erreicht werden können usw., so daß Knoten, die über $k$ Kanten erreicht werden können, besucht werden, bevor irgendein Knoten angetroffen wird, der $k+1$ Kanten erfordert. Wenn daher $y$ zum erstenmal angetroffen wird, ist der kürzeste Pfad von $x$ gefunden worden, da keine kürzeren Pfade $y$ erreicht haben.