8 Graphen

8.1 Überblick

In der Informatik werden (mehr oder weniger) schwierige Zusammenhänge häufig in Form von *Graphen* dargestellt. Graphen sind Strukturen, mit denen sich Beziehungen zwischen Einzelheiten ausdrücken lassen. Es gibt eine ganze Reihe von Graphvarianten, allen gemeinsam sind aber die Grundbestandteile *Knoten* und *Kanten*. Knoten sind voneinander unterscheidbare, eigenständige Objekte, Kanten verbinden Knoten. Kanten können zudem gerichtet sein, man spricht dann von gerichteten Graphen. Abbildung 8.1 zeigt eine typische Anwendung für Graphen: Verschiedene deutsche Großstädte sind durch Knoten dargestellt, die verbindenden Kanten sind mit Entfernungen beschriftet.

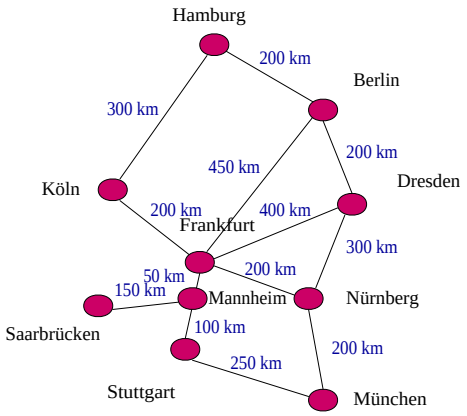


Abb. 8.1: Entfernungsgraph für verschiedene deutsche Städte

Definition 8.1. Ein *ungerichteter Graph* $G = (V, E)$ besteht aus

- einer endlichen Knotenmenge (vertices) V und
- einer endlichen Kantenmenge (edges) $E \subseteq V \times V$.

Neben diesen einfachen Graphen gibt noch eine Vielzahl anderer Graphtypen wie beispielsweise *Hypergraphen* (dessen *Hyperkanten* zwei oder mehr Knoten verbinden) oder *Multigraphen* (zwei Knoten können durch zwei oder mehr Kanten verbunden sein). Auf Multigraphen werden wir später in diesem Kapitel noch genauer eingehen, wenn wir *Eulersche Graphen* betrachten.

8.2 Begriffe

Die Knotenmenge V eines Graphen ist in der Regel *endlich* mit $n = |V|$. Somit ist es möglich, die Knoten durchzunummerieren: Der *i-te* Knoten wird dabei mit v_i bezeichnet. Ebenso können die Kanten nummeriert werden, die *j-te* Kante wird mit e_j bezeichnet. Eine

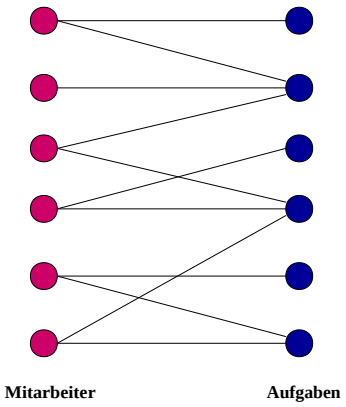


Abb. 8.2: Ein bipartiter Graph mit roten und blauen Punkten. Im Beispiel werden verschiedenen Mitarbeitern Aufgaben zugeteilt.

Kante kann auch durch die Angabe der Knoten i und j benannt werden, die die Kante verbindet, die entsprechende Bezeichnung lautet e_{ij}. Abbildung 8.3 (rechts) zeigt die beschriebene Knoten- und Kantenbenennung beispielhaft für den Graphen aus Abbildung 8.1.
Ein weiterer wichtiger Begriff im Zusammenhang mit Graphen ist der des Knotengrades:

Definition 8.2. Der *Grad* eines Knotens bezeichnet die Zahl der Kanten, die im Knoten enden.

Oft will man mit Hilfe eines Graphen ganz bestimmte Sachverhalte ausdrücken, für die weitere Eigenschaften notwendig sind. Zum Beispiel sind für die Routenplanung nicht nur Entfernungen zwischen Knoten interessant, sondern auch Bewegungsmöglichkeiten. Man möchte Kanten also zum Beispiel Richtungen zuweisen. *Richtung* bedeutet dabei, daß derartige Kanten einseitig und nicht symmetrisch sind.

Definition 8.3. Ein *gerichteter Graphen* enthält *gerichtete Kanten*, für die jeweils ein Start- sowie ein Endknoten definiert ist. Enthält ein Graph keine gerichteten Kanten, nennt man ihn *ungerichtet*.

Abbildung 8.3 (links) zeigt den Graphen aus Abbildung 8.1 mit gerichteten Kanten.

Definition 8.4. Ein Graph $G = (V, E)$ heißt *bipartit*, wenn es zwei disjunkte Knotenmengen $V_1, V_2 \subseteq V$ (also $V_1 \cap V_2 = \oslash$) gibt, so daß $E \subseteq \{\{v_1, v_2\} | v_1 \in V_1, v_2 \in V_2\}$ gilt.

Anschaulich gesprochen gibt es in einem bipartiten Graphen zwei disjunkte Knotenmengen (zum Beispiel rote und blaue Knoten). Kanten dürfen nur zwischen Knoten verschiedener Farbe gezogen werden. Ein Beispiel für einen bipartiten Graphen ist in Abbildung 8.2 (rechts) dargestellt.

Definition 8.5. In einem *vollständigen* Graphen existiert zwischen jedem Knotenpaar eine Kante.

Ein vollständiger Graph mit n Knoten hat $n(n-1)/2 = \mathcal{O}(n^2)$ Kanten. Abbildung 8.4 zeigt ein vollständigen Graphen.

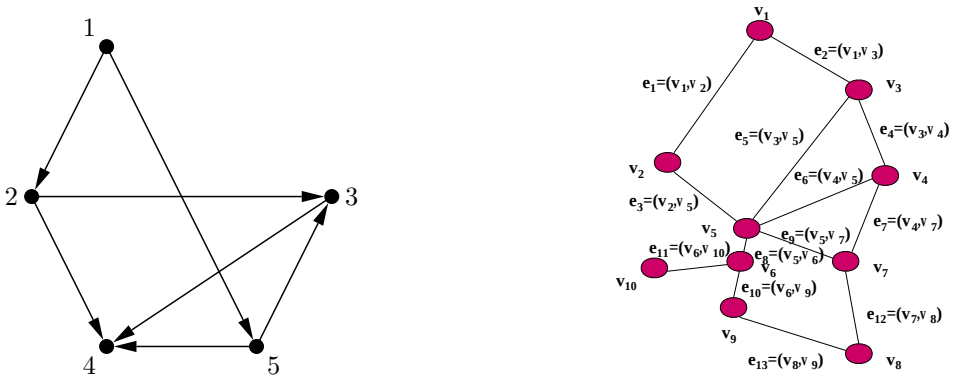


Abb. 8.3: *Links:* Noch einmal der Graph aus Abbildung 8.1, diesmal mit gerichteten Kanten ... *Rechts:* ... und mit Knoten- und Kanten-Indizes

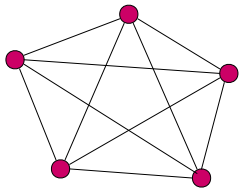


Abb. 8.4: Vollständiger Graph

8.3 Wie repräsentiert man Graphen?

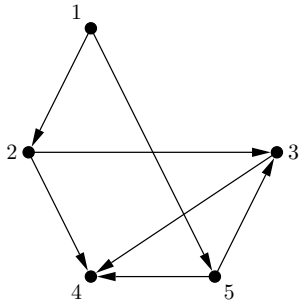
Graphen können auf verschiedene Weise repräsentiert werden. Je nach Problemstellung können ganz unterschiedliche Formen gewählt werden, von denen wir im folgenden einige darstellen. Die Repräsentationsform soll es erlauben, einfach und effizient typische Operationen auf einem Graphen auszuführen. Typische Operationen sind

- Festellen, ob zwei Knoten benachbart sind
- Bestimmung der nächsten Nachbarn eines Knotens
- Bestimmung der Nachfolger und Vorgänger eines Knotens in einem gerichteten Graphen

Effizient sollten nicht nur die genannten Operationen auszuführen sein, denn auch der Speicherplatz-Verbrauch ist ein wichtiges Kriterium. Häufig besteht allerdings eine Wechselwirkung zwischen Verbrauch an Speicherplatz und der Effizienz der Operationen: Effiziente Operationen gehen häufig zu Lasten des Speicherplatzes.

8.3.1 Adjazenzmatrix

Die Abbildung 8.5 zeigt eine naheliegende Repräsentationsform für Graphen: Eine $n \times n$-Matrix $A(G)$. Gibt es eine Kante e_{ij} in G, ist der Eintrag a_{ij} in $A(G)$ auf 1 gesetzt, anderfalls 0. Eine solche Matrix stellt Nachbarschaften der Knoten dar, sie wird daher als *Adjazenzmatrix*



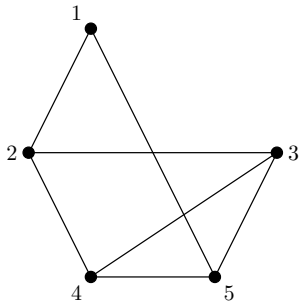
$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Abb. 8.5: Ein gerichteter Graph mit zugehöriger Adjazenzmatrix

bezeichnet. Sowohl gerichtete als auch ungerichtete Graphen lassen sich mit einer solchen Matrix repräsentieren. Im ungerichteten Fall ist die Matrix $A(G)$ symmetrisch (Abbildung 8.6).

Besonders einfach kann in einer Matrix geprüft werden, ob es eine Kante von i nach j gibt: Es muß lediglich der Eintrag a_{ij} betrachtet werden. Die Laufzeit für diese Operation ist somit $\mathcal{O}(1)$. Will man alle Nachbarn eines Knotens i in einem *ungerichteten* Graphen ermitteln, muß man hingegen alle Einträge der i-ten Zeile oder der i-ten Spalte überprüfen (n Schritte). Bei *gerichteten* Graphen finden man in der i-ten Zeile die Knoten, die von i aus erreichbar sind (*Nachfolger*), in der j-ten Spalte hingegen die Knoten, von denen aus eine Kante nach i führt (*Vorgänger*). Der Suchaufwand ist insgesamt also unabhängig von der tatsächlichen Anzahl der Nachbarn.

Der Speicherplatzbedarf für eine Adjazenzmatrix ist - unabhängig von der Anzahl der Kanten - immer n^2. Gibt es wenige Kanten im Graphen, enthält die zugehörige Adjazenzmatrix hauptsächlich Nullen. Ungerichtete Graphen können etwas effizienter gespeichert werden, da ihre Matrix symmetrisch ist; somit müssen nur die Einträge oberhalb der Diagonalen gespeichert werden. Dafür werden $n(n-1)/2$ Speicherplätze benötigt, wenn es keine *Schlingen* gibt (Kante von i zu sich selbst), $n(n+1)/2$ wenn es Schlingen gibt (in diesem Falle muß die Diagonale mitgespeichert werden).



$$A = \begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 1 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & 1 & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 0 & 1 & 1 & \mathbf{0} & \mathbf{1} \\ 1 & 0 & 1 & 1 & \mathbf{0} \end{pmatrix}$$

Abb. 8.6: Ein ungerichteter Graph mit zugehöriger Adjazenzmatrix

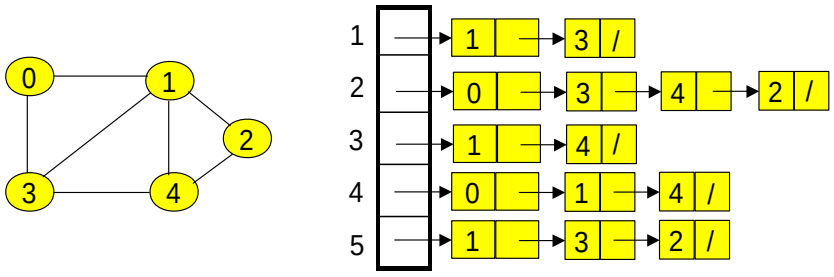


Abb. 8.7: Ungerichteter Graph mit zugehöriger Adjazenzliste

8.3.2 Adjazenzliste

In einer *Adjazenzliste* wird für jeden Knoten eine *Liste seiner Nachbarn* gespeichert. Somit hängt der Speicherplatzbedarf direkt von der Anzahl der Kanten in G ab. Sowohl gerichtete als auch ungerichtete Graphen können in Form von Adjazenzlisten gespeichert werden.

Eine Implementierungsmöglichkeit stellt ein Feld der Größe n dar mit. Pro Knoten in G gibt es einen Eintrag im Feld, dem eine verkettete Liste zugewiesen ist. In dieser verketteten Liste werden die Nachbarn des Knotens i abgelegt. Abbildung 8.7 zeigt einen ungerichteten Graphen mit zugehöriger Adjazenzliste. Will man prüfen, ob es eine Kante e_{ij} in G gibt, muß im schlimmsten Fall die gesamte Liste des Knotens i durchlaufen werden. Die Listen haben die maximale Länge $n - 1$. Die Suche nach einer Kante in einer Adjazenzliste ist also abhängig von der Größe des Graphen, im Gegensatz zur Adjazenzmatrix. Effizienter als bei der Adjazenzmatrix ist hier jedoch die Bestimmung aller Nachbarn eines Knotens i, denn diese Listen liegen in dieser Datenstruktur direkt vor.

Aufwendiger ist die Bestimmung der Vorgänger eines Knotens i, da in diesem Fall alle Nachbarlisten durchsucht werden müssen. Abhilfe schafft eine zweite Liste pro Knoten, in der zusätzlich zu den Nachfolgern die Vorgänger verwaltet werden.

Für einen gerichteten Graphen benötigt eine Adjazenzliste $n + m$ Speicherplätze, für einen ungerichteten Graphen $n + 2m$ mit n als Knotenanzahl und m als Kantenanzahl.

8.3.3 Adjazenzmatrix versus Adjazenzliste

Adjazenzlisten sind zwar aufwendiger zu implementieren und zu verwalten, bieten aber eine Reihe von Vorteilen gegenüber Adjazenzmatritzen. Zum einen verbrauchen sie stets nur linear viel Speicherplatz, was insbesondere bei dünnen Graphen (also Graphen mit wenig Kanten) von Vorteil ist, während die

Adjazenzmatrix quadratischen Platzbedarf bezüglich der Knoten-Anzahl besitzt (dafür aber kompakter bei dichten Graphen, also Graphen mit vielen Kanten ist). Zum anderen lassen sich viele graphentheoretische Probleme nur mit Adjazenzlisten in linearer Zeit lösen. In der Praxis verwendet man daher meist diese Form der Repräsentation.

8.4 Graphen und C++

In der C++-Welt existieren eine Menge fertige Bibliotheken, die Datenstrukturen und Algorithmen für Graphen zur Verfügung stellen. Zu den bekannteren Bibliotheken gehören die BOOST GRAPH LIBRARY [7] und LEDA [5]. Wir verwenden im Folgenden die Graphklassen von LEDA (Library of Efficient Data Types and Algorithms), die von Kurt Mehlhorn und Stefan Näher am Max-Planck-Institut für Informatik entwickelt wurde.
LEDA erlaubt es, sehr einfach Graphen zu definieren. Listing 8.4 zeigt, wie ein ungerichteter Graph G, zwei Knoten v, w und zwei Kanten e, f deklariert werden:

```
1 graph G;
2 node v,w;
3 edge e,f;
```

Gerichtete Graphen kann man zum Beispiel realisieren, indem jede ungerichtete Kante durch zwei gerichtete Kanten repräsentiert wird. Im Beispiel ist der Graph noch leer, hat also keine Knoten und Kanten. Ebenso sind die Knoten v, w und die Kanten e, f noch nicht initialisiert. Mit *nil* lassen sich Knoten und Kanten, die noch nicht näher spezifiziert sind, mit einem definierten Wert initialisieren.
Häufig ist es notwendig, über alle Knoten und Kanten eines Graphen zu iterieren. Zu diesem Zweck existieren in LEDA zwei Anweisungen:

```
1 forall_nodes(v,G) { };
2 forall_edges(e,G) { };
```

forall_nodes (v,G) iteriert über alle Knoten von G: Nacheinander werden alle Knoten in G dem Hilfsknoten v zugewiesen und der Anweisungsblock für jeden Knoten genau einmal ausgeführt. Genauso verhält es sich mit forall_edges (e,G), in diesem Fall für alle Kanten in G.
Sollen die Kanten durchlaufen werden, die mit einem Knoten v verbunden sind, bestehen verschiedene Möglichkeiten:

```
1 forall_out_edges(e,v) { };
2 forall_adj_edges(e,v) { };
3 forall_in_edges(e,v) { };
4 forall_inout_edges(e,v) { };
```

Die Zeilen 1 und 2 iteriert über alle Kanten, deren Quelle v ist. Zeile 3 durchläuft alle Kanten, deren Ziel v ist, Zeile 4 berücksichtigt alle Kanten von v.
Folgendes Beispiel zählt die Knoten in G:

```
1 int s = 0;
2 forall_edges(e,G) s++;
```

Die Knotenzahl für G läßt sich auch mit G.number_of_edges() abfragen.
LEDA erlaubt es, Zusatzinformation für Knoten und Kanten zu speichern. Dies kann zum Beispiel mit *node arrays* und *edge arrays* geschehen:

```
1 node_array<string> name(G);
2 edge_array<int>    length(G,1);
```

Diese beiden Deklarationen führen zwei Arrays *name* und *length* ein, die mit den Knoten bzw. Kanten von G indiziert sind. Die Einträge von *name* sind Zeichenketten. Alle Einträge von *name* werden mit dem leeren String initialisiert. *length* enthält Längenangaben für alle Kanten in G. Die Länge aller Kanten ist mit 1 initialisiert. Seien v Knoten und e Kante in G, dann ist folgendes möglich:

```
1 name[v]   = "Saarbruecken";
2 length[e] = 5;
```

Eine andere Möglichkeit, Informationen zu Knoten und Kanten zu speichern, sind *parametrisierte Graphen*:

```
1   GRAPH<string, int> H;
```

Auf diese Weise werden jedem Knoten eine Zeichenkette und jeder Kante ein Integer-Wert zugewiesen. Ähnlich wie oben ist in einem parametrisierten Graphen folgendes möglich:

```
1 H[v] = "Saarbruecken";
2 H[e] = 5;
```

Diese Zuweisungen sind natürlich nur erlaubt, wenn v und e Knoten bzw. Kante in H sind. Es besteht ein wichtiger Unterschied zwischen den beiden genannten Möglichkeiten zur Speicherung von Zusatzinformation: Während Knoten- und Kante-Arrays nur für *statische* Graphen definiert sind (nachträglich in G eingefügte Knoten oder Kanten haben keinen Eintrag im entsprechenden Array), sind parametrisierte Graphen dynamisch: Jedes neu hinzugefügte Element kann mit Zusatzinformation versehen werden. Aus dieser Sicht sind parametrisierte Graphen flexibler. Andererseits können beliebig viele Knoten- und Kanten-Arrays für einen Graphen definiert werden.

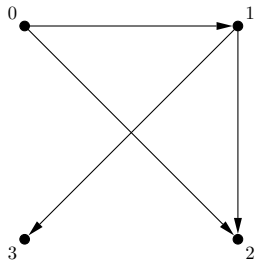


Abb. 8.8: Beispiel-Graph aus Listing 8.1

Bevor wir zu einer ersten Beispielanwendung kommen, zeigen wir im folgenden Quelltext, wie ein konkreter Graph in LEDA definiert werden kann. Der Graph entspricht dem in Abbildung 8.8 dargestellten:

Listing 8.1: Definition des Beispielgraphen aus Abb. 8.8

```
1 graph G;
2 node v0 = G.new_node();
3 node v1 = G.new_node();
4 node v2 = G.new_node();
5 node v3 = G.new_node();
6 G.new_edge(v0,v1); G.new_edge(v0,v2);
7 G.new_edge(v1,v2); G.new_edge(v1,v3);
```

Die folgende Beispielanwendung von LEDA-Graphen definiert eine Kopierfunktion, die eine Kopie H eines Graphen G erzeugt. Jeder kopierte Knoten speichert zusätzlich den entspre-

chenden Originalknoten aus G, und jede kopierte Kante speichert die entsprechende Originalkante:

```
1  void CopyGraph(GRAPH<node,edge>&H, const graph&G) {
2    H.clear();
3    node_array<node> copy_in_H(G);
4    node v;
5    forall_nodes(v,G)
6      copy_in_H[v] = H.new_node(v);
7    edge e;
8    forall_edges(e,G)
9      H.new_edge(copy_in_H[source(e)],copy_in_H[target(e)],e);
10 }
```

Wir definieren H als parametrisierten Graphen. Jeder Knoten kann einen anderen Knoten speichern, ebenso kann jede Kante eine andere Kante speichern. Das Knoten-Array *copy_in_H* für G erlaubt uns außerdem, auch jedem Knoten in G einen anderen Knoten zuzuweisen. Wir iterieren über alle Knoten in G. Für jeden Knoten v in G fügt die Anweisung *H.new_node(v)* einen neuen Knoten in H ein und assoziiert den als Parameter übergebenen Knoten v mit dem neuen Knoten. Der Rückgabewert dieser Operation ist ein neuer Knoten, den wir in *copy_in_H[v]* abspeichern. Die Schleife *forall_nodes* erzeugt genausoviele Knoten in H wie G Knoten hat und erzeugt bidirektionale Links zwischen den Knoten von G und H. Es gilt:

`H[copy_in_H[v]] = v für alle Knoten` v `aus` G
`copy_in_H[H[w]] = w für alle Knoten` w `aus` H

Die Anweisungen source(e) und target(e) geben Startknoten bzw. Zielknoten der Kante e zurück. Die Kanten in H können damit sehr einfach erzeugt werden: Für jede Kante e aus G fügen wir in H eine Kante ein, die die Knoten *copy_in_H[source(e)]* und *copy_in_H[target(e)]* verbindet. Der neuen Kante geben wir die Originalkante e als Information mit.

8.5 Pfade, Rundgänge und Eulersche Graphen

Häufig wird in Graphen nach Wegen von einem Knoten zu einem anderen gesucht, wie beispielsweise bei der Routenplanung.

Definition 8.6. Ein *Pfad* von v_i nach v_j ist eine Folge von Knoten $v_i, v_?, ..., v_j$, wobei jedes Paar von aufeinander folgenden Knoten v_k, v_l durch eine Kante (v_k, v_l) verbunden ist. Ein Pfad wird als *einfach* bezeichnet, wenn jeder Knoten in der Pfadbeschreibung genau einmal vorkommt.

Definition 8.7. Ein Pfad wird als *Circuit* (Rundgang) bezeichnet, wenn der erste und der letzte Knoten des Pfades identisch sind. Ein Circuit wird als einfach (oder als Kreis) bezeichnet, falls alle Knoten außer dem ersten und dem letzten genau einmal auftreten.

Die Abbildung 8.9 zeigt einen einfachen Pfad von v_3 nach v_8 (links), einen Rundgang (Mitte) sowie einen Kreis (rechts).

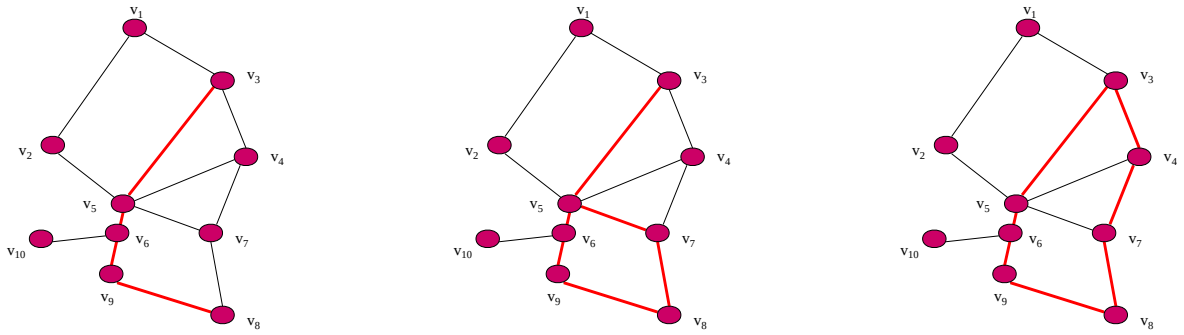


Abb. 8.9: *Links:* Die rot hervorgehobenen Kanten stellen einen *Pfad* von v_3 nach v_8 dar. *Mitte:* Ein *Circuit*: $v_3, v_5, v_6, v_9, v_8, v_7, v_5, v_3$. *Rechts:* Ein *Kreis*: $v_3, v_5, v_6, v_9, v_8, v_7, v_4, v_4$

8.5.1 Eulersche Graphen

Eine der Wurzeln der mathematischen Graphentheorie ist eine 1736 erschienene Arbeit von Euler über das Königsberger Brückenproblem. Die Lösung stellt gleichzeitig ein interessantes und typisches Beispiel für einen Algorithmus auf Graphen dar.
Dem Problem liegt ein Ausschnitt aus dem Königsberger Stadtplan zugrunde. Dort wo der alte und der neue Arm des Pregel zusammenfließen, bilden sie zusätzlich eine Insel, so daß der Fluß vier Landbereiche voneinander trennt, die durch insgesamt sieben Brücken verbunden waren, wie die Skizze 8.10 zeigt. Euler hat sich gefragt, ob es möglich ist, von irgendeinem

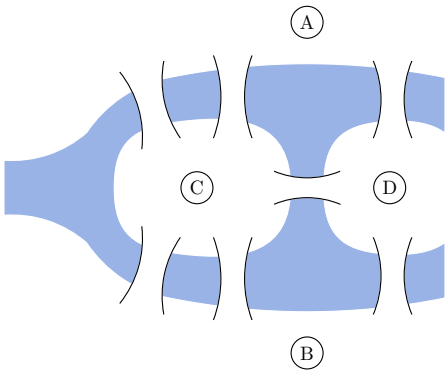


Abb. 8.10: Das Königsberger Brückenproblem

Punkt in der Stadt loszulaufen und zu diesem Punkt wieder zurückzukehren und dabei genau einmal über jede der sieben Brücken zu wandern. Eulers Antwort ist NEIN, und er hat sie nicht nur für die Königsberger Situation gegeben, sondern ein Verfahren entwickelt, mit dem sich die Frage für jeden beliebigen Wegeplan beantworten läßt. Dazu hat er die Landbereiche als Knoten aufgefaßt und die Brücken als ungerichtete Kanten zwischen den Landbereichsknoten, die sie verbinden. Bei dieser Abstraktion entsteht ein ungerichteter Graph, dessen Kanten an je zwei Knoten aufgehängt sind.

Definition 8.8. Ein *Multigraph* ist ein Graph, in dem mehrere Kanten zwischen zwei Knoten erlaubt sind.

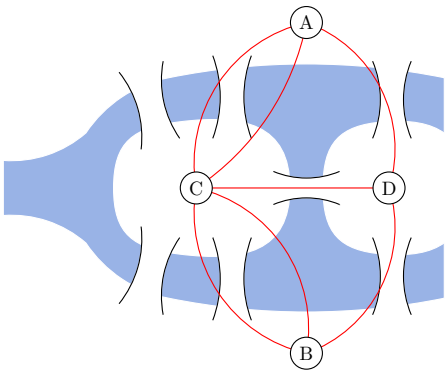


Abb. 8.11: Brückenproblem mit entsprechendem ungerichteten Graphen

Abbildung 8.11 zeigt den entsprechenden Graphen für das Königsberger Brückenproblem. Für solche Graphen lautet dann die Frage:

> Gibt es einen Rundweg (Kreis), der jede Kante genau einmal durchläuft? Wenn JA, wird der Graph Eulersch genannt.

Die Antwort lautet NEIN, da es diesem Graphen Knoten mit *ungeradem Grad* gibt (3 und 5). Ein Rundgang oder Circuit kann aber nur existieren, wenn alle Knoten geraden Grad haben und der Graph *zusammenhängend* ist.

8.5.2 Zusammenhangskomponenten

Definition 8.9. Ein Graph heißt *zusammenhängend* (connected), wenn man von jedem Knoten aus jeden anderen Knoten über einen Pfad erreichen kann. *Zusammenhängende Teilgraphen* eines Graphen bezeichnet man als *Zusammenhangskomponenten.*

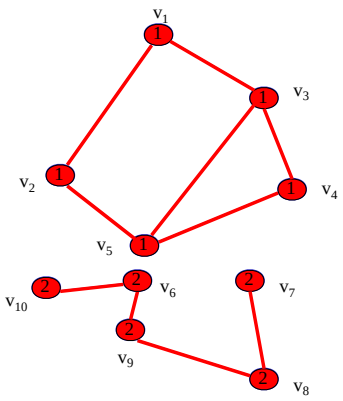


Abb. 8.12: Beispiel für Zusammenhangskomponenten in einem Graphen

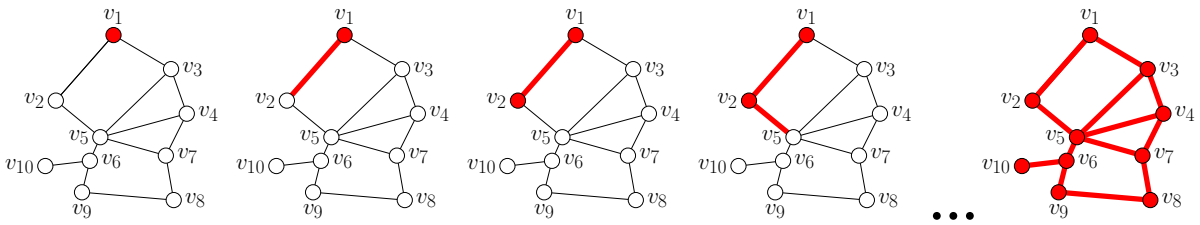


Abb. 8.13: Schritte des Algorithmus' zur Berechnung der Zusammenhangskomponente, zu der ein Knoten v gehört

Der Graph in Abb. 8.11 ist zwar zusammenhängend, aber kein Eulerscher Graph. In Abb. 8.12 wird ein Beispiel für Zusammenhangskomponenten gezeigt. Der folgende Pseudo-Code zeigt, wie man die Zusammenhangskomponente bestimmen kann, die zu einem Knoten v gehört: Gegeben ein ungerichteter Graph $G = (V, E)$ und ein Knoten v. Bestimmte die Zusammenhangskomponente (ZHK), zu der v gehört.

```
Zusammenhangskomponente(G, v):
  1. Markiere v;
  2. Für alle Kanten (v, w) führe
     die folgenden Operationen aus
     - Falls w nicht markiert ist, dann
       starte Zusammenhangskomponente G, w;
```

Der entsprechende C++-Code (Listing 8.2) definiert eine Funktion *connectedComponent*, der vier Parameter übergeben werden: v ist der Knoten, dessen Zusammenhangskomponente bestimmt werden soll, G der Graph (per konstanter Referenz), *marked* ein Knoten-Array, das zur Markierung der schon betrachteten Knoten dient (per Referenz), und schließlich eine Integer-Zahl, mit der die Knoten einer Zusammenhangskomponente markiert werden (vgl. Abb. 8.12):

Listing 8.2: connectedComponent

```
1  // Datenstruktur zur Speicherung der Nummer der ZHK:
2  // node_array<int> marked(G,0);
3  // aktuelle Nummer ist no_of_CC != 0
4  void connectedComponent(node v,
5                          const graph& G,
6                          node_array<int>& marked,
7                          int no_of_CC) {
8    marked[v]=no_of_CC;
9    edge e;
10   forall_out_edges(e,v) {
11     if(marked[target(e)]==0)
12       connectedComponent(target(e), G, marked, no_of_CC);
13   }
14 }
```

Abb. 8.14: G und G' aus Beweis Euler-Graph $\Rightarrow$ Euler-Circuit

Die Laufzeit dieser Funktion ist $\mathcal{O}$*(Zahl der Kanten der ZHK)*. Sollen *alle* Zusammenhangskomponenten eines Graphen G bestimmt werden, kann die Funktion *connectedComponent* wiederverwendet werden:

```
1  int connectedComponents(const graph& G,
2                          node_array<int>& marked) {
3    int no_of_CC = 0;
4    node v;
5    forall_nodes(v,G) {
6      if(marked[v]==0)
7        connectedComponent(v,G,marked,++no_of_CC); // Aufruf
8                                  // der oben definierten Funktion
9    }
10   return no_of_CC;
11 }
```

Diese Funktion durchläuft einfach alle Knoten v in G und ruft, falls der aktuelle Knoten noch nicht markiert ist, die Funktion *connectedComponent* auf, wobei bei jedem Aufruf die Zahl zur Markierung der Knoten einer ZHK inkrementiert wird. Der Rückgabewert der Funktion entspricht der Anzahl der gefundenen ZHKs.

Satz 8.1. *Die Zusammenhangskomponenten eines ungerichteten Graphen mit n Knoten und m Kanten können in Zeit $\mathcal{O}(n+m)$ bestimmt werden.*

Definition 8.10. Ein Graph heißt *Euler-Graph*, wenn jeder Knoten geraden Grad hat und der Graph zusammenhängend ist.

Satz 8.2. *Ein ungerichteter Graph besitzt genau dann einen Euler-Circuit, wenn der Graph ein Euler-Graph ist.*

Beweis. Wir beweisen die schwierige Richtung (Euler-Graph $\Rightarrow$ Euler-Circuit) mittels vollständiger Induktion über die Zahl m der Kanten (vgl. [6]):
Induktionsstart: m=3 (trivial).
Erklärung: Für $m = 0$ Kanten ist die Aussage wahr, da ein solcher Graph weder Kanten noch Knoten hat. Ein solcher Graph ist nicht sonderlich anschaulich, stattdessen können wir aber $m = 3$ setzen: Wenn jeder Knoten geraden Grades sein soll, kann es sich – bei $m = 3$ Kanten

– nur um den Graphen aus drei Knoten und drei Kanten handeln. Dieser Graph hat ganz sicher einen Eulerkreis. Die Fälle $m = 1$ und $m = 2$ können nicht auftreten, da die Knoten dann nicht alle geraden Grades sein könnten.
Induktionsannahme: Für alle zusammenhängenden Graphen mit $m - 1$ Kanten und ausschließlich Knoten geraden Grades gilt, daß sie einen Euler-Circuit enthalten.
Induktionsschluß: Sei $G = (V, E)$ ein Euler-Graph mit m Kanten (vgl. Abb. 8.14, links). Alle Knoten haben einen geraden Grad.

1. Bestimme in G einen Kreis K (siehe Übung). Zum Beispiel $K = ACBDA$.

2. Lösche die Kanten von K. Den resultierenden Graphen nennen wir G'. Es gilt:

   - Alle Knoten von G' haben geraden Grad, da wir einen *Zyklus* entfernt haben; in einem Zyklus hat aber jeder Knoten eine *gerade Anzahl* von Kanten, es wurde also auch eine gerade Anzahl von Kanten *pro Knoten* entfernt.
   - G' hat *weniger Kanten* als G, da wir Kanten entfernt haben.

   Ist G' zusammenhängend, sind wir fertig, da dann die Induktionsannahme gilt (für alle *zusammenhängenden* Graphen mit $m - 1$, also *weniger* als m Kanten). G' hätte dann einen Euler-Circuit, den wir nur noch mit dem Kreis K vereinigen müßten, den wir zuvor entfernt hatten.

   G' kann aber aus mehreren Zusammenhangskomponenten mit weniger als m Kanten bestehen (siehe Abb. 8.14, rechts). Auch dieser Fall ist einfach, da wieder die *Induktionsannahme* gilt: Alle Knoten der einzelnen ZHKs haben eine gerade Kantenanzahl, und ganz sicher hat jede ZHK weniger Kanten als G. Damit enthält jede verbleibende ZHK einen Euler-Circuit. Der Kreis K, den wir entfernt haben, *verbindet* diese einzelnen Euler-Circuits (anderenfalls wäre es nicht möglich, daß der Graph G zusammenhängend war). Der Euler-Circuit für den Gesamt-Graphen kann also auf einfache Weise konstruiert werden, indem K durchlaufen wird und die Euler-Circuits der ZHKs jeweils am Schnittpunkt zwischen K und ZHK betreten und durchlaufen werden.

■

Im Beispiel hat G' vier ZHKs: $ZHK(1) = \{A\}$, $ZHK(2) = \{D\}$, $ZHK(3) = \{H, I, C\}$, $ZHK(4) = \{B, E, G, F\}$ Die einzelnen Euler-Circuits können berechnet werden mit: $Circ(Z(3)) = CIHC$, $Circ(Z(4)) = BEGFB$
Aufgrund der Induktionsannahme existieren für diese ZHKs Euler-Circuits. Im letzten Schritt werden die Euler-Circuits der ZHKs mit dem Kreis K gemischt. Ergebnis: $Circuit = ACIHCBEGFBDA$
Mischprozedur: Übungsaufgabe.

Es folgen eine Reihe von Definitionen häufig verwendeter Begriffe und Eigenschaften von Graphen

Definition 8.11. Ein Knoten v eines gerichteten Graphen ist *balanciert*, wenn die Zahl der in v endenden Kanten gleich der Zahl der in v startenden Kanten ist (vgl. Abb. 8.15).

Definition 8.12. Ein gerichteter Graph ist *balanciert*, wenn jeder Knoten des Graphen balanciert ist.

Definition 8.13. Ein gerichteter Graph heißt *Euler-Graph*, wenn der Graph zusammenhängend und balanciert ist.

Satz 8.3. *Ein Graph besitzt genau dann einen Euler-Circuit, wenn der Graph ein Euler-Graph ist.*

Beweis: Analog zum konstruktiven Beweis im Falle des ungerichteten Graphen.

Definition 8.14. Ein Graph besitzt einen *Euler-Pfad*, wenn es einen Pfad im Graphen gibt, der jede kante genau einmal verwendet bzw. genau einmal über diese Kante geht.

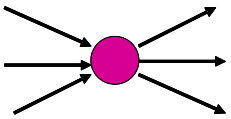


Abb. 8.15: Balancierter Knoten

Definition 8.15. Ein Knoten v eines gerichteten Graphen ist *semi-balanciert*, falls
$\mid indegree(v) - outdegree(v) \mid = 1$.

Satz 8.4. *Ein zusammenhängender Graph besitzt genau dann einen Euler-Pfad, wenn er höchstens zwei semi-balancierte Knoten besitzt und alle anderen balanciert sind.*

Beweis: Analog zum Beweis der Existenz eines Euler-Circuit (mit komplexerer Fallunterscheidung).

8.6 Suchverfahren in Graphen

Die folgenden Abschnitte beschreiben zwei Suchstrategien für Graphen. Diese Verfahren bilden die Grundlage für viele graphentheoretische Algorithmen, in denen alle Ecken oder Kanten eines Graphen systematisch durchlaufen werden müssen.
Für die meisten Suchverfahren gilt folgendes algorithmische Grundgerüst (*Markierungsalgorithmus*):

1. Markiere den Startknoten
2. Solange es noch Kanten von markierten zu unmarkierten Knoten gibt, wähle eine solche Kante und markiere deren Endknoten

Die beiden im folgenden diskutieren Verfahren, *Breitensuche* und *Tiefensuche*, unterscheiden sich in der Auswahl der Kanten in Schritt 2.

8.6.1 Breitensuche (BFS)

Die *Breitensuche* ist einer der einfachsten Algorithmen für die Suche in Graphen. Ausgehend vom Wurzelknoten v werden alle Knoten der Zusammenhangskomponente von v besucht. Zunächst werden alle Knoten besucht, die den Abstand $k = 1$ von v haben (die direkten Nachbarn von v). Diese Knoten werden in eine Liste eingefügt. Im weiteren Verlauf werden alle direkten Nachbarn der in dieser Liste gespeicherten Knoten besucht. Noch nicht entdeckte Nachbarn dieser Knoten werden wiederum in die Liste eingefügt. Der Prozeß wiederholt sich solange, bis die Liste leer ist. BFS besucht also die Knoten in der Reihenfolge ihres Abstands k zu v. Der Abstand zwischen einem Knoten u und der Wurzel v ist definiert als kleinste Zahl von Kanten zwischen v und u. Abb. 8.16 zeigt die Abarbeitungsreihenfolge der Knoten für einen Beispielbaum. BFS baut einen „breadth-first tree" auf, der nicht notwendigerweise dem Ausgangsgraphen entsprechen muß (nicht jeder Knoten in G ist notwendigerweise von v aus erreichbar, z.B. wenn es mehr als eine ZHK gibt). Der Name Breitensuche kommt daher, daß k (die Grenze zwischen besuchten und nicht besuchten Knoten) gleichförmig über die gesamte Breite des Suchbaums Schritt für Schritt erweitert wird.

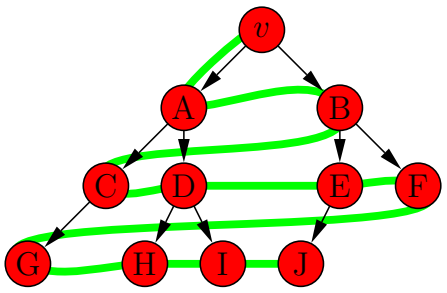


Abb. 8.16: Beispiel für BFS

8.6.2 BFS - eine Implementierung

Das folgende Listing zeigt eine BFS-Implementierung, die für jeden Knoten w in der Zusammenhangskomponente von v den Abstand zu v mitbestimmt. Die Entfernungen werden im Knoten-Array *dist* gespeichert. Alle Einträge des Knoten-Arrays werden mit Unendlich (∞) initialisiert (node_array<**int**> dist(G,∞)). Im Quellcode wird INFINITY als größte ganze Zahl definiert.

Listing 8.3: BFS

```
1  void bfs(const graph& G, node v, node_array<int>& dist) {
2    std::list<node> L;     node w,u;
3    dist[v]=0;             L.push_back(v);
4    while(!L.empty()) {
5      w=L.front(); L.pop_front();
6      forall_adj_nodes(u,w) {
7        if(dist[u]==INFINITY) {
8          dist[u]=dist[w]+1;
9          L.push_back(u);
10       }
11     }
12   }
13 }
```

Zu Beginn wird v in die Knotenliste L eingefügt, der Eintrag *dist[v]* wird auf 0 gesetzt, da v der Wurzelknoten ist. k ist damit zunächst 0. Die folgende *while*-Schleife terminiert, wenn L abgearbeitet ist. Innerhalb der Schleife wird dem temporäre Knoten w jeweils der oberste Eintrag der Liste L zugewiesen. Die Schleife *forall_adj_nodes*(u, w) durchwandert alle direkten Nachbarn des aktuellen Knoten w und prüft jeweils, ob der betreffende Knoten schon betrachtet wurde. Da alle Einträge im Array *dist* mit ∞ initialisiert wurden, kann dies als Markierung genutzt werden: Solange der Eintrag *dist[u]* auf ∞ gesetzt ist, wurde u noch nicht besucht. In diesem Fall wird *dist[u]* auf $k + 1$ gesetzt (u ist einen Schritt weiter entfernt von v als w, da u nächster Nachbar von w ist). Zuletzt wird u in die Liste L eingefügt, damit die Nachbarknoten von u im nächsten Schleifendurchlauf abgearbeitet werden können.

Satz 8.5. *Die Laufzeit von BFS für einen Graphen mit n Knoten und m Kanten ist $\mathcal{O}(n + m)$.*

Beweis. Jeder Knoten w wird genau einmal in die Liste der entdeckten Knoten aufgenommen und genau einmal aus dieser Liste entfernt. Dies wird durch die Abfrage der gespeicherten Entfernung *dist[w]==INFINITY* garantiert. Beim Abarbeiten eines Knotens werden alle seine Kanten betrachtet. Die Laufzeit der *forall_adj_nodes*-Schleife ist proportional zur Zahl der Kanten des Knotens. Da man jede Kante höchstens zweimal betrachtet (von jedem Knoten aus), ist die Gesamtlaufzeit proportional zur Zahl der Knoten und der Kanten, die betrachtet werden. ■

BFS berechnet die Distanz $d(v, u) = dist[u]$ von v zu jedem Knoten u, der von v aus erreichbar ist, d.h. der in der ZHK von v liegt.

Definition 8.16. Hierbei verstehen wir unter *Distanz* $d(v, u)$ die Länge des *kürzesten Pfades* von v nach u. Der kürzeste Pfad ist der Pfad mit der minimalen Zahl von Kanten zwischen v und u.

Knoten u, die nicht von v aus erreichbar sind, haben die Distanz *unendlich*: $d(v, u) = \infty$.

Lemma 8.1. *Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und sei $v \in V$ ein beliebiger Knoten. Dann gilt für jede Kante $(w, u) \in E$:*

$$d(v, u) \leq d(v, w) + 1$$

Beweis. Falls w von v aus erreichbar ist, dann ist es auch u. Dann kann der kürzeste Pfad von v nach u nicht länger sein als der kürzeste Pfad von v nach w gefolgt von der Kante (w, u). ■

Wir werden im folgenden zeigen, daß BFS die Distanzen $d(v, u)$ als *dist[u]* korrekt berechnet. Wir zeigen zunächst, daß *dist[u]* eine obere Schranke von $d(v, s)$ ist.

Lemma 8.2. *Für jeden Knoten u in V gilt: $dist[u] \geq d(v, u)$*

Beweis. Vollständige Induktion über die Zahl der *push_back*-Operationen, daß heißt die Zahl der Einfügungen von v in die Liste:
Induktionsstart: Ausgangspunkt für die Induktion ist die Situation direkt nach der Einfügung von v in die Liste L. Die Induktionsannahme gilt hier, denn:

$dist[v] = 0 = d(v, v)$ und $dist[u] = \infty \geq d(v, u)$

Induktionsschritt: Wir betrachten einen Knoten u, der während der Suche von einem Knoten w aus entdeckt wurde. Aus der Induktionsannahme folgt für w: $dist[w] \geq d(v, w)$. In dieser Situation gilt:

$$\begin{array}{lll} dist[u] & = dist[w] + 1 & \\ & \geq d(v, w) + 1 & \text{(aus Annahme)} \\ & \geq d(v, u) & \text{Lemma 8.1} \end{array}$$

Der Knoten u wird dann das erste und einzige Mal in die Liste L eingefügt. Der Wert von $dist[u]$ ändert sich im Laufe der Prozedur nicht mehr, und damit ist der Induktionsschluß vollzogen. ■

Lemma 8.3. *Enthält die Knotenliste während der Ausführung von BFS die Knoten $(u_1, u_2, ..., u_r)$, wobei u_1 der Kopf und u_r das Ende der Liste ist, dann gilt:*

$dist[u_r] \leq dist[u_1] + 1$ *und* $dist[u_i] \leq dist[u_{i+1}]$ *für* $i = 1, 2, ..., r - 1$

Beweis. Vollständige Induktion über die Zahl der *push_back*- und *pop_front*-Operationen:
Induktionsstart: Enthält die Liste nur v, so ist die obige Aussage trivial.
Wir müssen zeigen, daß die obige Aussage sowohl nach allen *push_back*-Einfügungen als auch nach den *pop_front*-Operationen gilt. Entfernt man den Kopf u_1 (und erhält u_2 als neuen Kopf), so folgt die Korrektheit der Aussage direkt aus der Induktionsannahme:

$dist[u_r] \leq dist[u_1] + 1 \leq dist[u_2] + 1$

Fügt man einen neuen Knoten u_{r+1} bei der Suche mit Knoten w in die Liste ein, so wurde w gerade aus der Liste entfernt und es folgt aus der Induktionsannahme: $dist[w] \leq dist[u_1]$. Daher gilt:

$dist[u_{r+1}] = dist[w] + 1 \leq dist[u_1] + 1$ und $dist[u_r] \leq dist[w] + 1 = dist[u_{r+1}]$

■

Korollar 8.1. *Wird der Knoten u_i vor dem Knoten u_j von BFS in die Knotenliste eingeführt, so gilt:*

$dist[u_i] \leq dist[u_j]$

Wir können jetzt beweisen, daß BFS korrekt funktioniert.

Satz 8.6. *Sei $G = (V, E)$ ein Graph, den man ausgehend von Knoten v mit BFS durchsucht. Dann findet BFS alle Knoten u, die von v aus erreicht werden können, und die berechnete Distanz $dist[u]$ ist gleich der Distanz $d(v, u)$ des kürzesten Pfades von v nach u im Graphen G.*

Beweis. Nehmen wir an, daß ein Knoten eine Distanz erhält, die nicht gleich der Länge des kürzesten Pfades ist. Nehmen wir ferner an, daß s der Knoten mit dem kleinsten falschen Wert $d(v, s)$ ist. Sicherlich ist s nicht gleich v. Nach Lemma 8.2 gilt:

$dist[s] \geq d(v, s)$. Dann muß folglich $dist[s] > d(v, s)$ gelten.

Knoten s muß von v aus erreichbar sein, denn sonst wäre $d(v, s) = \infty = dist[s]$, da ja nach Lemma 8.2 der Eintrag $dist[s]$ nur größer als $d(v, s)$ sein kann, wenn er auf ∞ gesetzt ist.
Sei u der Knoten, der auf dem kürzesten Pfad von v nach s liegt und von dem aus man direkt nach s gelangt, das heißt die letzte Kante des kürzesten Pfades ist (u, s). Dann gilt:

$$d(v, u) + 1 = d(v, s)$$

Wegen $d(v, u) < d(v, s)$ sowie der Art und Weise, wie wir s definiert haben, muß folgendes gelten:

$$dist[u] = d(v, u)$$

Fassen wir die letzten beiden Aussagen zusammen, so erhalten wir:

$$dist[s] > d(v, s) = d(v, u) + 1 = dist[u] + 1 \tag{8.1}$$

Nun betrachten wir den Zeitpunkt, zu dem u aus der Liste entfernt wurde. Wir unterscheiden drei Fälle und zeigen, daß wir in jedem Fall eine Widerspruch zur Annahme erhalten.

1. s **wurde von** u **aus entdeckt:** Dann wird von der Prozedur der Eintrag *dist[s]* auf *dist[u]+1* gesetzt, was zu einem Widerspruch zu Ungleichung 8.1 führt.

2. s **wurde bereits vor** u **aus der Liste** L **entfernt:** Dann wurde s bereits aus der Liste L entfernt, und aufgrund von Korollar 8.1 muß $dist[s] \leq dist[u]$ gelten, was ebenfalls der Ungleichung 8.1 widerspricht.

3. u **wird vor** s **aus der Liste** L **entfernt, aber** s **ist bereits entdeckt worden:** Dann wurde s entdeckt, als ein u vorausgehender Knoten w aus der Liste L entfernt wurde, für den gilt:

   $$dist[s] = dist[w] + 1$$

   Nach Korollar 8.1 gilt aber:

   $$dist[w] \leq dist[u]$$

   Hieraus folgt:

   $$dist[s] \leq dist[u] + 1$$

   Dies liefert wiederum einen Widerspruch zur Ungleichung 8.1.

Somit haben wir gezeigt, daß für alle $s \in V$ gilt: $dist[s] = d(v, s)$. ■

Wir ändern nun die Prozedur BFS aus Listing 8.3 so ab, daß gleichzeitig der *BFS-Baum* aufgebaut wird. Der BFS-Baum enthält alle Knoten des Graphen G und diejenigen Kanten, die von der Prozedur durchlaufen wurden. In einem ungerichteten Graphen entsprechen die Pfade im zugehörigen BFS-Baum den kürzesten Pfaden zwischen s und allen erreichbaren Knoten. Der Vater jedes Knotens im BFS-Baum wird im Knoten-Array *parent* gespeichert.

Listing 8.4: BFS mit Aufbau des BFS-Baums

```
1  void bfs(const graph& G, node v, node_array<int>& dist,
2           node_array<node>& parent) {
3    std::list<node> L;
4    node w,u;
5    dist[v]=0;
6    L.push_back(v);
7    while(!L.empty()) {
8      w=L.front(); L.pop_front();
9      forall_adj_nodes(u,w) {
10       if(dist[u]==INFINITY) {
11         dist[u]=dist[w]+1;
12         L.push_back(u);
13         parent[u]=w;     // w ist der Vater von u
14       }
15     }
16   }
17 }
```

Definition 8.17. Sei $G = (V, E)$ ein Graph mit Quelle v. Wir definieren den Predecessor-Graphen von G als $G_p = (V_p, E_p)$, wobei

$$V_p = \{s \in V | parent[s] \neq nil\} \cup \{v\}$$

und

$$E_p = \{(parent[s], s) | s \in V_p\{v\}\}$$

Lemma 8.4. *Wendet man BFS auf einen gerichteten oder ungerichteten Graphen $G = (V, E)$ an, so ist der berechnete Predecessor-Graph ein Baum. Dieser Baum heißt BFS-Baum oder BF Tree (Breadth-First Tree).*

8.6.3 Tiefensuche (DFS)

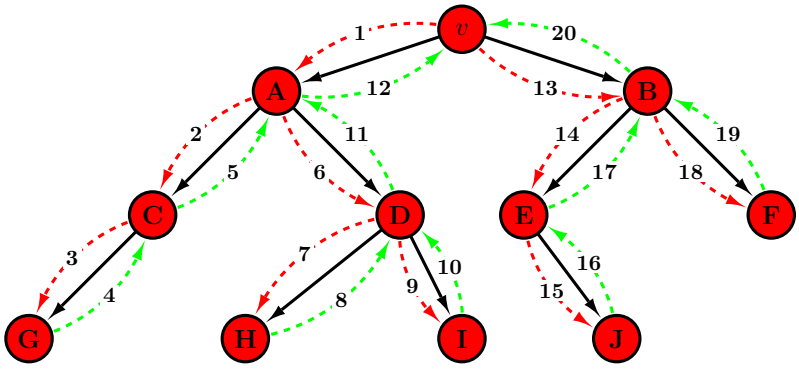


Abb. 8.17: Beispieldurchlauf für DFS: Rote Pfeile tauchen tiefer in den Graphen ein. Gibt es keine unbesuchten Kinder mehr, wandert DFS zurück (grüne Pfeile). Die Nummern an den farbigen Pfeilen geben die Ausführungsreihenfolge an.

Die *Tiefensuche* (Depth-first search) taucht immer so „tief" wie möglich in den Graphen ein. Für jeden Knoten v im Graphen werden alle Kanten ausgehend von v getestet. Der erste Nachbarknoten u, der noch nicht entdeckt wurde, wird als nächstes besucht. Von u aus verfährt die Tiefensuche analog, wieder werden alle Kanten getestet und der erste Nachbarknoten x ausgewählt, sofern er zuvor noch nicht besucht wurde. Falls ein Knoten keine unbesuchten Kinder mehr besitzt, wandert DFS zurück zum vorhergehenden Knoten und testet, ob dieser schon abgearbeitet wurde. Falls das nicht der Fall ist – falls also noch unbesuchte Nachbarknoten existieren – wird wiederum die nächsten nicht besuchten Nachbarn abgearbeitet. DFS läuft, bis alle erreichbaren Knoten in G besucht wurden. Ein Beispieldurchlauf ist in Abbildung 8.17 zu sehen.

Im folgenden wird eine DFS-Implementierung diskutiert, die die Reihenfolge, in der die Knoten der ZHK von v erreicht werden, mitberechnet und die die verwendeten Kanten des DFS-Baum in einer Liste speichert. Die Knotenreihenfolge wird im Knoten-Array *no* (für *node order*) gespeichert, die Kanten in der Kanten-Liste *T*.

Listing 8.5: DFS

```
1  static int counter = 0;
2  void dfs(const graph& G, node v, node_array<int>& no,
3           list<edge>& T) {
4    no[v] = counter++;
5    edge e;
6    forall_out_edges(e,v) {
7      node w = target(e);
8      if(no[w]==-1) {
9        T.push_back(e);     // falls erforderlich parent[w]=v;
10       dfs(G, w, no, T);   // rekursiver Aufruf von dfs
11     }
12   }
13 }
```

Für die folgenden Betrachtungen setzen wir voraus, daß die obige DFS-Prozedur so abgeändert ist, daß die Bäume aller ZHKs von G berechnet werden.

8.6.4 Eigenschaften von DFS

Satz 8.7. *Die Laufzeit von DFS für einen Graphen $G = (V, E)$ mit n Knoten und m Kanten ist $\mathcal{O}(n + m) = \mathcal{O}(|V| + |E|)$.*

Definition 8.18. Sei $G = (V, E)$ ein Graph. Wir definieren den DFS-Predecessor-Graphen von G als $G_p = (V, E_p)$, wobei

$$E_p = \{(parent[s], s) | s \in V \wedge parent[s] \neq nil\}$$

Der Predecessor-Graph von DFS bildet einen „Depth-First Forest", der sich aus mehreren „Depth-First Trees" zusammensetzen kann. Die Kanten in E_p nennt man die Baumkanten.
Wir führen nun eine Zeitmessung in die DFS-Prozedur ein:

Listing 8.6: DFS mit Zeitnahme

```
1  static int counter = 0;
2  static int dfs_time = 0;     // Unsere Uhr
3
4  void dfs(const graph& G, node v, node_array<int>& no,
5    list<edge>& T,
6    node_array<int>& detection_time,    // Entdeckungs-Zeit
7    node_array<int>& finishing_time) { // Knoten abgearbeitet
8    no[v] = counter++;
9    edge e;
10   forall_out_edges(e,v) {
11     node w = target(e);
12     if(no[w]==-1) {
13       detection_time[w]=++dfs_time; // neuer Knoten entdeckt
14       T.push_back(e);
15       dfs(G, w, no, T, detection_time, finishing_time);
16       finishing_time[w]=++dfs_time; // Knoten abgearbeitet
17     }
18   }
19 }
```

Wir verwenden im folgenden die Abkürzungen *DT* und *FT* für „detection_time" und „finishing_time".

Satz 8.8. *Nach Anwendung von DFS auf einen Graphen G gilt für zwei beliebige Knoten u und v des Graphen eine der drei folgenden Bedingungen:*

1. *Die Zeitintervalle $[DT[u], FT[u]]$ und $[DT[v], FT[v]]$ sind disjunkt, und weder v noch u sind Nachfahren des jeweils anderen Knotens im DF Forest.*

2. *Das Intervall $[DT[u], FT[u]]$ ist vollständig im Intervall $[DT[v], FT[v]]$ enthalten, und u ist ein Nachfahre von v in einem Baum des DF Forest.*

3. *Das Intervall $[DT[v], FT[v]]$ ist vollständig im Intervall $[DT[u], FT[u]]$ enthalten, und v ist ein Nachfahre von u in einem Baum des DF Forest.*

Beweis. Wir betrachten den Fall $DT[u] < DT[v]$. Im symmetrischen Fall vertausche man einfach die Rollen von v und u.
Zunächst betrachten wir den Teilfall, daß $DT[v] < FT[u]$, d.h. v wurde entdeckt, bevor u abgearbeitet ist. Dies impliziert, daß v ein Nachfahre von u ist, und daß aufgrund der rekursiven Implementierung die Kanten von v bereits abgearbeitet worden sind, bevor die Prozedur zum Knoten u zurückkehrt. In diesem Fall ist $[DT[v], FT[v]]$ vollständig in dem Intervall $[DT[u], FT[u]]$ enthalten, und v ist ein Nachfahre von u (Fall 3).
Wir betrachten nun den zweiten Teilfall $FT[u] < DT[v]$ (u wurde abgearbeitet, bevor v entdeckt wurde). In diesem Fall müssen die Zeitintervalle disjunkt sein, und daher ist weder v noch u ein Nachfahre des anderen Knotens. ■

Aus dem obigen Satz folgt direkt:

Korollar 8.2. *Knoten v ist genau dann ein Nachfahre von Knoten u im DF Forest, wenn gilt:*

$$DT[u] < DT[v] < FT[v] < FT[u]$$

Satz 8.9. *Im DF Forest eines gerichteten oder ungerichteten Graphen $G = (V, E)$ ist ein Knoten v genau dann ein Nachfahre von Knoten u, falls zu der Zeit, wenn u entdeckt wird ($DT[u]$), ein Pfad von u nach v existiert, der ausschließlich unentdeckte Knoten enthält.*

Beweis. $\Rightarrow$: Wir nehmen an, daß v ein Nachfahre von u in einem Baum des DF Forest ist. Sei w irgend ein Knoten auf dem Pfad zwischen u und v im entsprechenden DF Tree. Dann gilt nach Korollar 8.2 $DT[u] < DT[w]$, und daher sind alle Knoten w mit der obigen Eigenschaft noch nicht entdeckt.

$\Leftarrow$: Wir nehmen an, daß zum Zeitpunkt $DT[u]$ der Knoten v von u aus über einen Pfad mit unentdeckten Knoten erreicht werden kann, daß aber v nicht zu einem Nachkommen von u im DF Forest wird.
O.B.d.A. nehmen wir an, daß jeder Knoten auf dem Pfad vor v ein Nachfahre von Knoten u im DF Forest wird (andernfalls betrachten wir den Knoten, der u am nächsten ist und diese Eigenschaft besitzt).
Sei w der Vorgänger von v auf dem Pfad der unentdeckten Knoten. Nach Korollar 8.2 gilt

$$FT[w] < FT[u]$$

Man beachte, daß v nach u entdeckt wird, aber bevor w abgearbeitet ist. Daher muß gelten:

$$DT[u] < DT[v] < FT[w] < FT[u]$$

Aus Satz 8.8 folgt dann, daß das Intervall $[DT[v], FT[v]]$ in dem Intervall $[DT[u], FT[u]]$ vollständig enthalten sein muß und daß v ein Nachfahre von u im DF Forest sein muß. ■

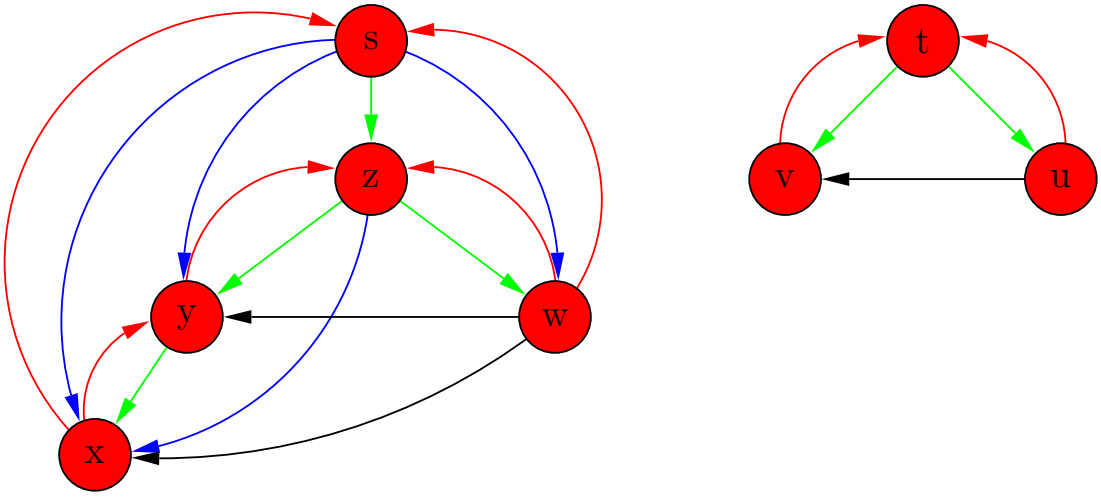


Abb. 8.18: Vier Kantentypen können mit DFS klassifiziert werden.

8.6.5 Kantenklassifikation mit DFS

Die Tiefensuche kann für eine *Kantenklassifikation* für den Graphen $G = (V, E)$ verwendet werden, mit der wichtige Informationen über G gesammelt werden können. Wir definieren vier Kantentypen, die bei einem DFS-Durchlauf für G produziert werden (siehe Abbildung 8.18):

1. Tree edges (Baumkanten) sind Kanten des DF Forest G_p. Die Kante (u, v) ist eine Baumkante, falls v über die Kante (u, v) entdeckt wurde.

2. Back edges (Rückwärtskanten) sind Kanten (u, v), die einen Knoten u mit einem Vorfahren v in einem DF Tree verbinden.

3. Forward edges (Vorwärtskanten) sind Kanten (u, v), die nicht zum DF Forest gehören und einen Knoten u mit einem Nachfahren v verbinden.

4. **Cross edges** (Querkanten) sind die anderen Kanten.

Der vorgestellte DFS-Algorithmus kann so modifiziert werden, daß die Kanten entsprechend der obigen Aufzählung klassifiziert werden. Hierzu betrachten wir den Knoten v, der von u aus erreicht wird, wenn die Kante (u, v) zum ersten Mal untersucht wird. Ist v noch nicht entdeckt, dann ist (u, v) eine Baumkante. Ist v entdeckt, aber noch nicht abgearbeitet, dann ist (u, v) eine Rückwärtskante. Ist v abgearbeitet, dann ist (u, v) eine Vorwärtskante *oder* eine **Querkante**.
Da in einem ungerichteten Graphen die Knotenpaare (u, v) und (v, u) dieselbe Kante repräsentieren, wird die Kante entsprechend der zuerst auftretenden Variante (u, v) oder (v, u) klassifiziert, je nachdem, welche Richtung zuerst während der Ausführung des Algorithmus' gefunden wird.

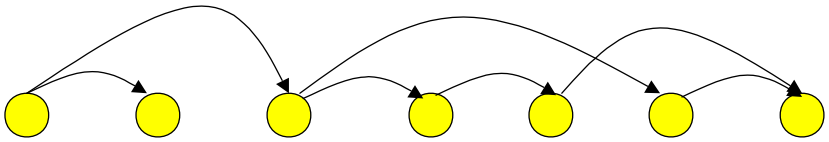


Abb. 8.19: Topologisches Sortieren mit DFS

Satz 8.10. *Die DFS-Klassifizierung eines ungerichteten Graphen $G = (V, E)$ ordnet jede Kante von G entweder in die Klasse Baumkante oder in die Klasse Rückwärtskante ein.*

Beweis. Sei (u, v) eine beliebige Kante von G und sei o.B.d.A.

$$DT[u] < DT[v]$$

Dann wird v entdeckt und abgearbeitet, bevor u abgearbeitet wird, denn v ist ein Nachbar von u, d.h. es gibt die Kante (u, v). Der Knoten v muß also ein Nachfahre von u sein. Wurde v entdeckt, als die Kante (u, v) von u aus betrachtet wurde, so ist (u, v) eine Baumkante. Betrachtete man die Kante (u, v) zum ersten Mal von v aus, dann ist (u, v) eine Rückwärtskante. Analog gilt $DT[u] > DT[v]$. ■

8.6.6 Topologisches Sortieren mittels DFS

Definition 8.19. Ein *DAG* (directed acyclic graph) ist ein gerichteter azyklischer Graph, d.h. ein gerichteter Graph, der keine Rundgänge besitzt.

Definition 8.20. Eine *topologische Sortierung* eines DAG ist eine (lineare) Ordnung aller Knoten, so daß für alle Kanten (u, v) des Graphen gilt: Der Knoten u erscheint in der Ordnung vor v.

Eine topologische Ordnung eines Graphen kann man sich als Aufreihung aller seiner Knoten entlang einer horizontalen Linie vorstellen, wobei alle gerichteten Kanten von links nach rechts führen. Abbildung 8.19 zeigt ein Beispiel. DAGs werden häufig verwendet, wenn es darum geht, die Abfolge von Ereignissen festzuhalten. Der folgende Pseudo-Code berechnet eine topologische Sortierung für einen DAG.

Topological-Sort(G):

1. Starte DFS und berechne die „finishing time” (FT) für alle Knoten
2. Wenn ein Knoten abgearbeitet ist, dann füge ihn am Anfang der (Ordnungs-)Liste ein
3. Gib die Liste zurück

Lemma 8.5. *Ein gerichteter Graph ist genau dann azyklisch, wenn DFS keine Rückwärtskanten produziert (findet).*

Beweis. $\Rightarrow$: Wir nehmen an, daß es eine Rückwärtskante (u, v) gibt. Dann ist u ein Nachfahre von v im DF Forest, und es gibt folglich einen Pfad von v nach u. Fügen wir die Kante (u, v) diesem Pfad hinzu, so erhalten wir einen Kreis. Folglich ergibt sich ein Widerspruch zur Annahme, daß wir es mit einem azyklischen Graphen zu tun haben.

$\Leftarrow$: Wir nehmen an, daß G einen Zyklus C enthält, und zeigen, daß unter DFS diesen Voraussetzungen eine Rückwärtskante findet. Sei v der erste Knoten des Zyklus C, den DFS entdeckt, und sei (u, v) die Kante im Zyklus, die zu v führt. Zur Zeit $DT[v]$ gilt: Der Teilpfad des Zyklus C von v nach u besteht nur aus noch nicht entdeckten Knoten. Daher folgt aus Satz 8.9, daß u ein Nachfahre von v im DF Forest ist. Also ist (u, v) eine Rückwärtskante. ■

Satz 8.11. Topological-Sort *produziert eine topologische Sortierung auf einem DAG.*

Beweis. Es genügt zu zeigen, daß für die Knoten jeder Kante (u, v) im DAG gilt:

$$FT[v] < FT[u]$$

Sei (u, v) eine Kante, die von DFS bearbeitet wird. Zum Zeitpunkt der Bearbeitung der Kante (u, v) kann v nicht als entdeckt, aber noch nicht abgearbeitet markiert sein, denn dann hätte DFS eine Rückwärtskante gefunden (nach vorhergehendem Lemma ist das ein Widerspruch zur Voraussetzung, daß der Graph ein DAG ist). Folglich ist v entweder noch nicht entdeckt oder schon abgearbeitet. Falls v noch nicht entdeckt ist, wird v ein Nachfahre von u und deshalb gilt:

$$FT[v] < FT[u]$$

Ist v abgearbeitet, so ist $FT[v]$ schon gesetzt worden. Da wir aber den Knoten u noch bearbeiten, wird $FT[u]$ noch gesetzt und es gilt deshalb:

$$FT[v] < FT[u]$$

■

8.6.7 Starke Zusammenhangskomponenten

Wir betrachten nun eine klassische Anwendung von DFS: das Zerlegen eines gerichteten Graphen in seine *starken Zusammenhangskomponenten* (strongly connected components). Der Abschnitt zeigt, wie eine solche Zerlegung durch zwei Anwendungen von DFS erreicht werden kann.

Definition 8.21.

1. Eine *starke Zusammenhangskomponente* eines gerichteten Graphen $G = (V, E)$ ist eine maximale Knotenmenge $C \subseteq V$, so daß für jedes Paar $u, v \in C$ gilt: Der Knoten u kann von v aus über einen Pfad, der vollständig zu C gehört, erreicht werden.

2. Ein gerichteter Graph wird als stark zusammenhängend bezeichnet, wenn er aus einer einzigen starken Zusammenhangskomponente besteht.

Der vorgestellte Algorithmus zur Auffindung starker Zusammenhangskomponenten in einem Graphen $G = (V, E)$ verwendet den *transponierten* Graphen $G^T = (V, E^T)$ von G, wobei $E^T = \{(v, u) | (u, v) \in E\}$. E^T besteht also aus den umgedrehten Kanten von G. G und G^T haben die gleichen starken Zusammenhangskomponenten (siehe Abb. 8.20).

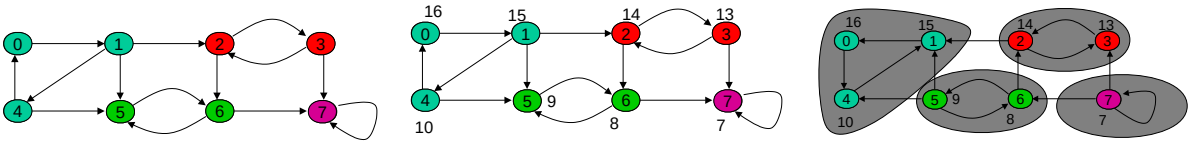


Abb. 8.20: *Links:* G. *Mitte:* G nach Schritt 2 von **SCC(G)**. *Rechts:* Transponierter Graph G^T von G mit den starken Zusammenhangskomponenten (grau).

Strongly-Connected-Components(G): SCC(G):

1. Berechne die „finishing_time" (FT) für jeden Knoten mit $DFS(G)$
2. Berechne den transponierten Graphen G^T
3. Berechne $DFS(G^T)$, wobei die Knoten in der Reihenfolge ihrer „finishing_time"-Einträge aus der $DFS(G)$-Berechnung in Schritt 1 (fallend) in der Hauptschleife von $DFS(G^T)$ abgearbeitet werden
4. Gib die Knoten eines jeden Baumes des DF Forest, der in Zeile 3 berechnet wurde, als eine starke Zusammenhangskomponente aus

Der obige Pseudo-Code umgesetzt in C++:

```
1  static int counter=0;
2  static int dfs_time=0;  // Unsere Uhr
3
4  void DFS(const graph& G, node_array<int>& no,
5           list<edge>& T,
6           node_array<int>& detection_time,  // Zeit der Entdeckung
7           node_array<int& finishing_time) { // Knoten abgearbeitet
8           node v;
9           forall_node(v,G) {
10            if(no[v]==-1) dfs(G, v, no, detection_time, finishing_time
11          }
12 }
```

Die *forall_node(v,G)*-Schleife entspricht Schritt 3 des Pseudo-Codes.

Lemma 8.6. *Seien C und C' zwei verschieden starke Zusammenhangskomponenten in einem gerichteten Graphen $G = (V, E)$. Seien ferner $u, v \in C$ und $u', v' \in C'$. Gibt es einen Pfad $u \to u'$ von u nach u' in G, dann kann es keinen Pfad $v' \to v$ von v' nach v geben.*

Beweis. Gäbe es einen Pfad von v nach v', dann wären weder C noch C' starke Zusammenhangskomponenten, da dann jeder Knoten von C von jedem Knoten von C' aus erreichbar wäre und umgekehrt. ■

Abbildung 8.21 demonstriert das obige Lemma. Da *SCC(G)* zwei DFS-Aufrufe beinhaltet, könnte es zu Verwechslungen kommen, wenn wir von „detection_time" und „finishing_time" sprechen. Im folgenden Abschnitt beziehen wir uns auf den ersten DFS-Aufruf in *SCC(G)*.

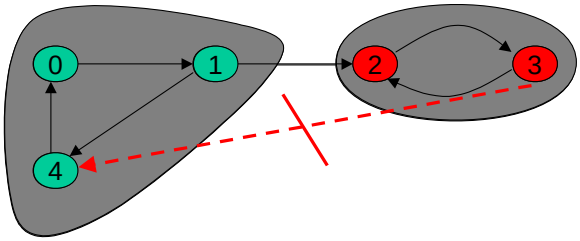


Abb. 8.21: Gäbe es die rot gestrichelte Kante, wären weder C noch C' starke Zusammenhangskomponenten.

Definition 8.22. Sei $U \subseteq V$ eine Teilmenge der Knoten. Wir definieren:

$$d(U) = min\{DT[u] | u \in U\}$$

$$f(U) = max\{FT[u] | u \in U\}$$

Lemma 8.7. *Seien C und C' zwei verschiedene starke Zusammenhangskomponenten des gerichteten Graphen $G = (V, E)$. Gibt es eine Kante mit (u, v) mit $u \in C$ und $v \in C'$, dann gilt:*

$$f(C) > f(C')$$

Beweis. Fall 1: $d(C) < d(C')$: Sei x der erste Knoten in C, der entdeckt wird: $d(C) = DT[x]$. Zu diesem Zeitpunkt sind alle Knoten von C und C' außer x noch nicht entdeckt. Zu jedem Knoten von C und C' existiert von x aus ein Pfad, der vollständig aus unentdeckten Knoten besteht. Satz 8.9 impliziert, daß alle Knoten in C und C' Nachfahren von x sind. Aus Korollar 8.2 folgt, daß $FT[x] = f(C) > f(C')$.

Fall 2: $d(C) > d(C')$: Sei y der erste Knoten in C', der entdeckt wird: $d(C') = DT[y]$. Zu diesem Zeitpunkt sind alle Knoten in $C' \backslash y$ unentdeckt. Zu jedem Knoten von C' existiert von y aus ein vollständig unentdeckter Pfad. Satz 8.9 impliziert, daß alle Knoten in C' Nachfahren von y im DF Forest sind. Aus Korollar 8.2 folgt, daß $FT[y] = f(C')$. Da es eine Kante (u, v) von C nach C' gibt, kann nach Lemma 8.6 kein Pfad von C' nach C existieren. Deshalb ist kein Knoten in C von C' aus erreichbar, und alle Knoten in C sind immer noch unentdeckt, wenn alle Knoten von C' abgearbeitet sind. Daher gilt für jeden Knoten w von C:

$$FT[w] > f(C')$$

Hieraus folgt:

$$f(C) > f(C')$$

■

Korollar 8.3. *Seien C und C' zwei verschiedene starke Zusammenhangskomponenten in einem gerichteten Graphen $G = (V, E)$. Gibt es eine Kante $(u, v) \in E^T$ im transponierten Graphen G^T, wobei $u \in C$ und $v \in C'$ ist, dann gilt $f(C) < f(C')$.*

Beweis. Da $(u, v) \in E^T$, ist $(v, u) \in E$. Da G und G^T die gleichen starken Zusammenhangskomponenten enthalten, folgt aus dem vorhergehenden Lemma, daß $f(C) < f(C')$ gilt. ■

Das obige Korollar liefert den Schlüssel zum Verständnis von *SCC(G)*. Betrachten wir hierzu den zweiten Aufruf der Prozedur *DFS(G^T)* auf dem transponierten Graphen $G^T = (V, E^T)$. Die Knoten werden beginnend mit dem Knoten mit der größten „finishing_time" in absteigender Reihenfolge abgearbeitet. Aus Korollar 8.3 folgt: Bearbeiten wir gerade eine starke Zusammenhangskomponente C', so gibt es keine Kanten aus Knoten von C', die zu nicht entdeckten Knoten in anderen starken Zusammenhangskomponenten (sZHK) führen. Betrachten wir eine Kante (u, v) mit $u \in C'$, so ist v

- entweder ein bereits entdeckter und abgearbeiteter Knoten einer sZHK C, die bereits abgearbeitet wurde, d.h. $f(C) > f(C')$
- oder ein entdeckter oder noch nicht entdeckter Knoten von C'.

Die obige Aussage besagt, daß die neuentdeckten Knoten alle zur starken Zusammenhangskomponente C' gehören.
Man findet dabei auch alle Knoten von C': Da C' eine sZHK ist, gibt es von jedem Knoten x, der als erster entdeckt wird, einen vollständig unentdeckten Pfad zu jedem anderen Knoten der sZHK C'.

Satz 8.12. *Die Prozedur SCC(G) berechnet die starken Zusammenhangskomponenten eines gerichteten Graphen G korrekt.*

Beweis. Vollständige Induktion über die Zahl k der Bäume, die in Zeile 3 der Prozedur *SCC(G)* beim Aufruf von *DFS(G^T)* berechnet und konstruiert wurden.
Die *Induktionsannahme* ist, daß die ersten k Bäume, die in Zeile 3 von *SCC(G)* berechnet wurden, sZHKn von G sind. Wir betrachten den $k+1$-ten DF Baum, der in Zeile 3 berechnet wird. Sei u die Wurzel dieses Baumes, und u gehöre zur sZHK C. Es gilt:

$$FT[u] = f(C) > f(C')$$

für alle noch nicht bearbeiteten (noch nicht entdeckten) sZHKn C'. Da u der erste entdeckte Knoten von C ist, gilt zum Zeitpunkt $DT[u]$, daß alle anderen Knoten von C zu diesem Zeitpunkt unentdeckt sind. Dann folgt aus Satz 8.9, daß alle anderen Knoten von C Nachfahren von u im $k+1$-ten DF Baum sind. Aus Korollar 8.3 und der Induktionsannahme folgt ferner, daß alle Kanten in E^T, die C verlassen, auf Knoten in bereits entdeckten und abgearbeiteten sZHKn zeigen. Deshalb wird kein Knoten einer anderen sZHK ein Nachfahre von u im $k+1$-ten DF Baum sein. Daher bilden die Knoten des $k+1$-ten DF Baumes eine starke Zusammenhangskomponente (die $k+1$-te sZHK). ■

8.6.8 Der Komponenten-Graph G^{SCC}

Der *Komponenten-Graph* $G^{SCC} = (V^{SCC}, E^{SCC})$ wird folgendermaßen aufgebaut: Für jede starke Zusammenhangskomponente C_i addiert man einen Knoten v_i zur Knotenmenge V^{SCC}.

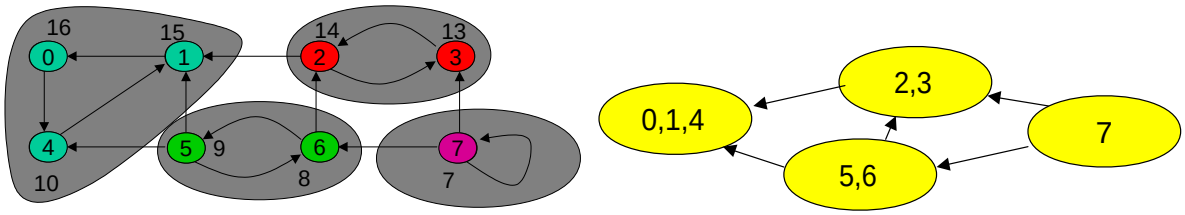


Abb. 8.22: Graph G aus Abbildung 8.20 und sein zugehöriger Komponentengraph.

Es wird genau dann eine gerichtete Kante (v_i, v_j) zur Kantenmenge E^{SCC} hinzugefügt, die die starken Zusammenhangskomponenten C_i und C_j verbindet, wenn es mindestens eine gerichtete Kante in E^T von einem Knoten $x \in C_i$ zu einem Knoten $y \in C_j$ gibt. Abbildung 8.22 zeigt den entsprechenden Komponenten-Graphen für den Graphen G aus Abbildung 8.20.

8.7 Anmerkungen

Bei der Erstellung dieses Kapitels haben wir uns die Vielzahl der bereits existierenden WWW-Seiten, Skripte und Bücher zum Thema Graphen und Graphalgorithmen zu Nutze gemacht. Die Schwierigkeit bestand weniger darin, entsprechende Ausarbeitungen zu finden, als viel mehr darin, eine geeignete Auswahl zu treffen. Die wichtigsten Quellen haben wir hier zusammengestellt:

- Algorithmen auf Graphen, Skript zur Veranstaltung Wintersemester 2003/04, Universität Bremen, Arbeitsgruppe Theoretische Informatik[1]
- Introduction to algorithms, Cormen et.al. [2]
- Algorithmische Graphentheorie von Volker Turau [11]
- Das LEDA-Buch von Mehlhorn et.al. [5]
- The Theory of Computation, Bernard M.E. Moret [6]
- Wikipedia: `http://de.wikipedia.org`

[1] Link: `http://www.informatik.uni-bremen.de/theorie/teach/aag/Skript/skript.pdf`