3

Graphen

In der Einleitung wurde ein spezielles Graphenproblem anhand eines Alltagsbeispiels vorgestellt, das Färbbarkeitsproblem, mit dessen Lösung ein Lehrer versucht, seine Schüler so auf die Zimmer einer Jugendherberge zu verteilen, dass ihm Ärger während der Klassenfahrt möglichst erspart bleibt (siehe Abb. 1.1 und Abb. 1.2). In diesem Kapitel werden weitere Graphenprobleme eingeführt, die wir in diesem Buch untersuchen wollen. Zunächst benötigen wir einige grundlegende Begriffe und Definitionen der Graphentheorie.

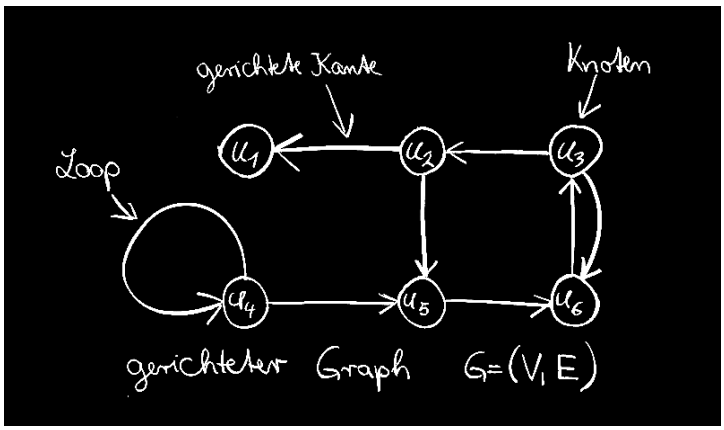

Abb. 3.1. Beispiel für einen gezeichneten Graphen

3.1 Grundbegriffe

Eine der wichtigsten Datenstrukturen in der Informatik ist der Graph. Mit Hilfe von Graphen lassen sich oft sehr einfach die strukturellen Gegebenheiten algorithmischer

F. Gurski et al., *Exakte Algorithmen für schwere Graphenprobleme*, eXamen.press,
DOI 10.1007/978-3-642-04500-4_3,

Graph, Knoten
Kante

Probleme beschreiben. Ein *Graph* G ist ein Paar (V,E), wobei V eine Menge von *Knoten* (englisch: *vertices*) und E eine Menge von *Kanten* (englisch: *edges*) ist. Wir betrachten in diesem Buch ausschließlich Graphen mit endlichen Knoten- und endlichen Kantenmengen. Es gibt in der Literatur unterschiedliche Graphenmodelle, die sich hauptsächlich in der Definition der Kantenmenge unterscheiden. Die beiden folgenden Varianten werden jedoch am häufigsten für algorithmische Analysen verwendet.

gerichteter Graph

1. In einem *gerichteten Graphen* (englisch: *digraph*, als Abkürzung von *directed graph*) $G=(V,E)$ ist die Kantenmenge eine Teilmenge der Menge aller Knotenpaare:
$$E \subseteq V \times V.$$
gerichtete Kante
Startknoten
Zielknoten
Jede *gerichtete Kante* (u,v) ist ein geordnetes Paar von Knoten, wobei u der *Startknoten* und v der *Zielknoten* der Kante (u,v) ist.
ungerichteter Graph
2. In einem *ungerichteten Graphen* $G=(V,E)$ ist die Kantenmenge eine Teilmenge der Menge aller zwei-elementigen Knotenmengen:
$$E \subseteq \{\{u,v\} \mid u,v \in V, u \neq v\}.$$
ungerichtete Kante
Endknoten
Jede *ungerichtete Kante* $\{u,v\}$ ist eine Menge von zwei Knoten u und v, die die *Endknoten* der Kante $\{u,v\}$ sind.

Grundsätzlich kann ein ungerichteter Graph auch als ein gerichteter Graph dargestellt werden, indem für jede ungerichtete Kante $\{u,v\}$ zwei gerichtete Kanten (u,v) und (v,u) verwendet werden. In der Regel ist es jedoch bequemer und anschaulicher, für ungerichtete Graphen auch das ungerichtete Graphenmodell zu verwenden.

Graphen lassen sich sehr einfach zeichnerisch veranschaulichen. Dabei werden die Knoten als Punkte oder Kreise und die Kanten als verbindende Pfeile oder Linien gezeichnet. Graphalgorithmen lesen die Knotenmenge und Kantenmenge jedoch normalerweise sequenziell als Folgen von Knoten und Kanten ein.

Kardinalität einer Menge
Größe eines Graphen

Die *Kardinalität einer Menge* S (also die Anzahl ihrer Elemente) bezeichnen wir mit $|S|$. Die *Größe eines Graphen* G ist
$$\text{size}(G) \;=\; |V|+|E|.$$

Die „Gleichheit“ von Graphen wird wie bei allen komplexen Strukturen über einen Isomorphismus definiert, da in der Regel eine *strukturelle Gleichheit* und keine mengentheoretische Gleichheit gemeint ist. Zwei gerichtete bzw. zwei ungerichtete

Graphisomorphie

Graphen $G=(V,E)$ und $G'=(V',E')$ sind *isomorph*, falls eine bijektive Abbildung $f: V \to V'$ existiert mit
$$(\forall u,v \in V)\ \ [(u,v) \in E \quad \Leftrightarrow \quad (f(u),f(v)) \in E'] \tag{3.1}$$
bzw.
$$(\forall u,v \in V)\ \ [\{u,v\} \in E \quad \Leftrightarrow \quad \{f(u),f(v)\} \in E']. \tag{3.2}$$

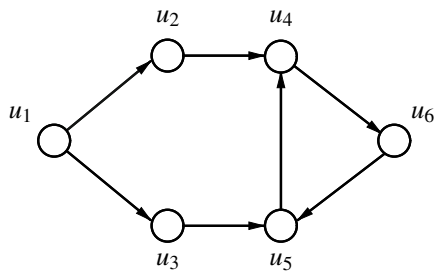

Abb. 3.2. Ein gerichteter Graph $G = (V,E)$ mit den sechs Knoten $u_1,\ldots,u_6$ und den sieben gerichteten Kanten (u_1,u_2), (u_1,u_3), (u_2,u_4), (u_3,u_5), (u_4,u_6), (u_6,u_5) und (u_5,u_4)

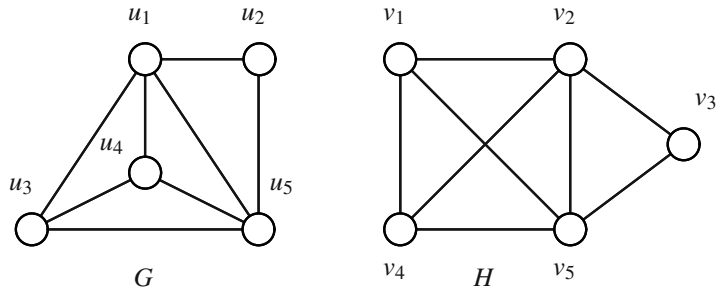

Abb. 3.3. Zwei unterschiedlich dargestellte isomorphe Graphen

Beispiel 3.1 (Graphisomorphie). Abbildung 3.3 zeigt zwei isomorphe Graphen $G = (V,E)$ und $G' = (V',E')$. Die Isomorphie wird zum Beispiel durch die bijektive Abbildung $f : V \to V'$ mit $f(u_1) = v_2$, $f(u_2) = v_3$, $f(u_3) = v_1$, $f(u_4) = v_4$ und $f(u_5) = v_5$ belegt, da

$$(\forall u,v \in V)\ \ [\{u,v\} \in E) \ \ \Leftrightarrow \ \ \{f(u),f(v)\} \in E'].$$

Teilgraph

Ein Graph $G' = (V',E')$ heißt *Teilgraph* eines Graphen $G = (V,E)$, falls $V' \subseteq V$ und $E' \subseteq E$. Ist $G' = (V',E')$ ein Teilgraph von G und gilt zusätzlich, dass E' alle Kanten aus E enthält, deren Endknoten in V' sind (d. h., $E' = \{\{u,v\} \in E \mid u,v \in V'\}$), so heißt G' *induzierter Teilgraph* von G. Jeder induzierte Teilgraph ist somit auch ein gewöhnlicher Teilgraph, jedoch gilt dies im Allgemeinen nicht umgekehrt. Die Kantenmenge in einem induzierten Teilgraphen G' von G ist eindeutig durch die Knotenmenge V' bestimmt. Wir schreiben deshalb auch $G[V']$ für den durch $V' \subseteq V$ in G induzierten Teilgraphen.

induzierter Teilgraph

Beispiel 3.2 (Teilgraph). Abbildung 3.4 zeigt einen Graphen G mit sieben Knoten und zehn Kanten. Der Graph J enthält sechs Knoten und sechs Kanten und ist kein induzierter Teilgraph von G. Der Graph H mit seinen fünf Knoten und sechs Kanten ist dagegen ein induzierter Teilgraph von G und damit natürlich (wie J auch) ein „ganz normaler“ Teilgraph von G.

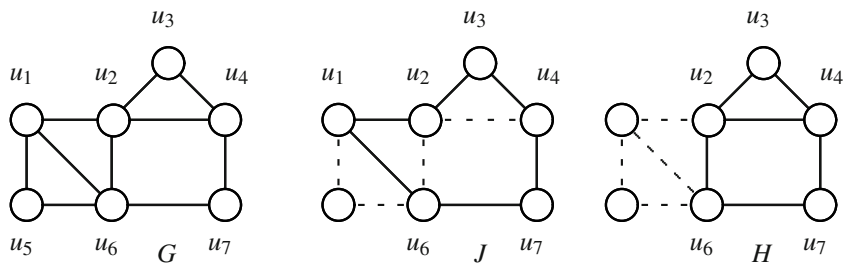

Abb. 3.4. Ein Graph, ein Teilgraph und ein induzierter Teilgraph

Die folgende Definition fasst einige der elementaren Begriffe über Graphen zusammen, die sowohl für gerichtete als auch für ungerichtete Graphen relevant sind. Die entsprechenden Definitionen unterscheiden sich gelegentlich in nur unbedeutenden formalen Details. Die formale Abänderung der jeweiligen Definition für ungerichtete Graphen geben wir in Klammern an, falls nötig.

Definition 3.3. *Sei $G = (V,E)$ ein gerichteter (bzw. ein ungerichteter) Graph.*

adjazent
inzident
Nachbarschaft

1. *Zwei Knoten u,v sind in G miteinander* verbunden *oder* adjazent *oder* benachbart*, wenn es eine gerichtete Kante $e = (u,v)$ oder $e = (v,u)$ gibt (bzw. wenn es eine ungerichtete Kante $e = \{u,v\}$ gibt); dann nennt man u und v mit e* inzident*. Zwei Kanten mit einem gemeinsamen Knoten heißen* inzident*. Die* Nachbarschaft *$N(v)$* eines Knotens *v ist die Menge aller mit v adjazenten Knoten.*

Eingangsgrad
Ausgangsgrad

2. *Der* Eingangsgrad *$indeg_G(u)$ eines Knotens u in G ist die Anzahl der gerichteten Kanten mit u als Zielknoten. Der* Ausgangsgrad *$outdeg_G(u)$ von u in G ist die Anzahl der gerichteten Kanten mit u als Startknoten.*

Knotengrad

3. *Der* Knotengrad *$deg_G(u)$ eines Knotens u in G ist die Anzahl der mit u inzidenten Kanten. (Für ungerichtete Graphen ist nur der Knotengrad von Belang, d. h., man unterscheidet nicht zwischen Eingangs- und Ausgangsgrad.)*
maximaler Knotengrad
Der maximale Knotengrad *eines Graphen G ist definiert als*

$$\Delta(G) = \max_{v \in V} deg_G(v).$$

minimaler Knotengrad
Analog ist der minimale Knotengrad *von G definiert als*

$$\delta(G) = \min_{v \in V} deg_G(v).$$

Schleife

4. *Eine* Schleife *(englisch:* loop*) ist eine gerichtete Kante (u,u) von einem Knoten zu sich selbst.*

Weg P_k
Länge eines Wegs

5. *Ein* gerichteter *(bzw.* ungerichteter*)* Weg *(englisch:* path*) in G ist eine alternierende Folge $P_k = (u_1, e_1, u_2, \ldots, u_{k-1}, e_{k-1}, u_k)$ von Knoten $u_1, \ldots, u_k \in V$ und Kanten $e_1, \ldots, e_{k-1} \in E$ mit $e_i = (u_i, u_{i+1})$ (bzw. $e_i = \{u_i, u_{i+1}\}$) für alle i, $1 \leq i \leq k-1$. Die* Länge eines Wegs *ist in der Regel die Anzahl seiner Kanten,*[10] *und der Weg P_k mit k Knoten hat dementsprechend die Länge $k-1$.*

[10] Gelegentlich wird aber auch die Anzahl der Knoten als Weglänge verwendet.

Ein Weg P_k wird oft vereinfacht als Knotenfolge

$$P_k = (u_1, \ldots, u_k)$$

mit der Eigenschaft $(u_i, u_{i+1}) \in E$ (bzw. $\{u_i, u_{i+1}\} \in E$) für $i \in \{1, \ldots, k-1\}$ definiert. Es ist aber auch möglich, einen Weg P als Teilgraphen $G_P = (V_P, E_P)$ von G zu definieren. In einem gerichteten Weg $P_k = (u_1, \ldots, u_k)$ wird der erste Knoten, u_1, als Startknoten *und der letzte Knoten, u_k, als* Zielknoten *bezeichnet. In einem ungerichteten Weg P_k heißen die beiden Knoten u_1 und u_k* Endknoten *des Wegs.*

Startknoten
Zielknoten
Endknoten

Die Distanz *zwischen zwei Knoten u und v im Graphen G, bezeichnet mit $\mathrm{dist}_G(u, v)$, ist die minimale Anzahl der Kanten in einem Weg von u nach v (bzw. zwischen u und v).*

Distanz

Beispiel 3.4 (Weg und Schleife). *Abbildung 3.5 zeigt zwei Wege der Länge vier: einen gerichteten Weg P und einen ungerichteten Weg P'. Der dritte Graph in Abb. 3.5, S, besteht aus genau einem Knoten, u, und einer Schleife.*

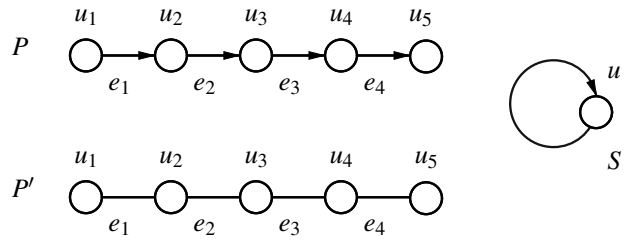

Abb. 3.5. Zwei Wege, P und P', und eine Schleife S

6. *Ein Weg $C_k = (u_1, \ldots, u_k)$ in G ist ein* Kreis in G, *falls zusätzlich $(u_k, u_1) \in E$ (bzw. $\{u_k, u_1\} \in E$) gilt. Alternativ können Kreise C auch als Folgen*

Kreis C_k

$$C = (u_1, e_1, u_2, \ldots, u_{k-1}, e_{k-1}, u_k, e_k)$$

von alternierenden Knoten bzw. Kanten oder auch als Teilgraphen

$$G_C = (\{u_1, \ldots, u_k\}, \{e_1, \ldots, e_k\})$$

von G definiert werden.

7. *Ein Weg bzw. Kreis P ist* einfach, *falls alle Knoten in P paarweise verschieden sind.*

einfacher Weg
einfacher Kreis

8. *Zwei Wege, $P = (u_0, \ldots, u_k)$ und $P' = (v_0, \ldots, v_{k'})$, sind* knotendisjunkt, *falls $u_i \neq v_j$ für alle i und j mit $0 < i < k$ und $0 < j < k'$ gilt. Die Endknoten der Wege werden bei dieser Ungleichheit in der Regel nicht mit einbezogen.*

knotendisjunkte Wege

Die folgenden Begriffe beziehen sich ausschließlich auf gerichtete Graphen.

Definition 3.5 (schwacher und starker Zusammenhang). *Sei $G = (V,E)$ ein gerichteter Graph.*

schwach zusammenhängend

1. *G ist* schwach zusammenhängend*, falls*

$$(\forall U \subseteq V)\ (\exists u \in U)\ (\exists v \in V - U)\ \ [(u,v) \in E \vee (v,u) \in E].$$

bilateral

2. *G ist* bilateral*, falls es in G zwischen je zwei Knoten $u,v \in V$ einen Weg von u nach v oder einen Weg von v nach u gibt.*

stark zusammenhängend

3. *G ist* stark zusammenhängend*, falls es in G zwischen je zwei Knoten $u,v \in V$ einen Weg von u nach v gibt.*

Die folgenden Begriffe beziehen sich ausschließlich auf ungerichtete Graphen.

Definition 3.6 (*k*-facher Zusammenhang). *Sei $G = (V,E)$ ein ungerichteter Graph.*

zusammenhängend

1. *G ist* zusammenhängend*, falls es in G zwischen je zwei Knoten $u,v \in V$ einen Weg gibt.*

k-fach zusammenhängend

2. *G ist* k-fach zusammenhängend*, $k \geq 2$, falls es in G zwischen je zwei Knoten $u,v \in V$, $u \neq v$, k einfache, paarweise knotendisjunkte Wege gibt.*

Satz 3.7 (Menger [Men27]). *Ein Graph mit mehr als $k \geq 0$ Knoten ist genau dann k-fach zusammenhängend, wenn er nicht durch Herausnahme von höchstens $k-1$ Knoten und ihren inzidenten Kanten unzusammenhängend werden kann.* **ohne Beweis**

Für den Entwurf effizienter Graphalgorithmen ist es wichtig, den zu untersuchenden Graphen geeignet intern zu speichern. Am einfachsten kann ein gerichteter Graph $G = (V,E)$ mit n Knoten in einem zweidimensionalen Array A als *Adjazenzmatrix* gespeichert werden, also als die $(n \times n)$-Matrix mit den folgenden n^2 Einträgen:

Adjazenzmatrix

$$A[u][v] \ = \ \begin{cases} 0, & \text{falls } (u,v) \notin E, \\ 1, & \text{falls } (u,v) \in E. \end{cases}$$

Die Speicherung eines Graphen mit n Knoten als Adjazenzmatrix benötigt $\Theta(n^2)$ Platz, unabhängig davon, ob der Graph sehr viele oder nur sehr wenige Kanten hat. Algorithmen, die als interne Datenstruktur eine Adjazenzmatrix verwenden, benötigen deshalb wegen der Initialisierung der Adjazenzmatrix immer mindestens $\Omega(n^2)$ Rechenschritte. Die Verwendung einer Adjazenzmatrix ist somit nur dann sinnvoll, wenn die Laufzeit der besten Algorithmen im besten Fall (also bezüglich der Best-case-Zeitkomplexität) $\Omega(n^2)$ nicht unterschreitet. Selbst wenn der Graph bereits in Form einer Adjazenzmatrix als Eingabe gegeben ist und die Adjazenzmatrix nicht erst aufgebaut werden muss, kann die Laufzeit der Algorithmen für viele Graphenprobleme $\Omega(n^2)$ nicht unterschreiten. Dies zeigt der folgende Satz von Rivest und Vuillemin [RV76], den wir hier jedoch nicht beweisen möchten.

Satz 3.8 (Rivest und Vuillemin [RV76]). *Sei $\mathscr{E}$ eine Grapheigenschaft, für die gilt:*

nicht triviale Grapheigenschaft

1. *$\mathscr{E}$ ist* nicht trivial*, d. h., es gibt mindestens einen Graphen, der die Eigenschaft $\mathscr{E}$ hat, und es gibt mindestens einen Graphen, der die Eigenschaft $\mathscr{E}$ nicht hat.*

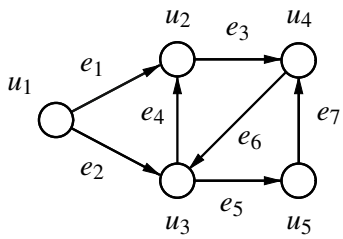

Abb. 3.6. Beispielgraph $G = (V,E)$ mit $V = \{u_1,\ldots,u_5\}$ und $E = \{e_1,\ldots,e_7\}$, wobei $e_1 = (u_1,u_2)$, $e_2 = (u_1,u_3)$, $e_3 = (u_2,u_4)$, $e_4 = (u_3,u_2)$, $e_5 = (u_3,u_5)$, $e_6 = (u_4,u_3)$ und $e_7 = (u_5,u_4)$

2. *$\mathscr{E}$ ist* monoton*, d. h., wenn ein Graph G die Eigenschaft $\mathscr{E}$ hat, dann haben auch alle Teilgraphen von G die Eigenschaft $\mathscr{E}$.* monotone Grapheigenschaft
3. *$\mathscr{E}$ ist unabhängig von der Anordnung der Knoten der Graphen mit Eigenschaft $\mathscr{E}$, d. h., für alle Graphen G und alle zu G isomorphen Graphen G' gilt $\mathscr{E}(G) = \mathscr{E}(G')$.*

Dann benötigt jeder Algorithmus, der die Eigenschaft $\mathscr{E}$ auf der Basis einer Adjazenzmatrix entscheidet, mindestens $\Omega(n^2)$ Rechenschritte. **ohne Beweis**

Satz 3.8 gilt zum Beispiel für die Eigenschaft, ob ein Graph kreisfrei ist. Daraus folgt, dass jeder Algorithmus, der auf der Basis einer Adjazenzmatrix entscheidet, ob ein gegebener Graph $G = (V,E)$ einen Kreis enthält, mindestens $\Omega(n^2)$ Rechenschritte benötigt. Adjazenzmatrizen sind besonders dann ungeeignet, wenn alle mit einem Knoten $u \in V$ inzidenten Kanten inspiziert werden müssen. Um die inzidenten Kanten zu finden, müssen alle n Einträge einer Zeile und einer Spalte untersucht werden, auch wenn ein Knoten u nur eine einzige inzidente Kante besitzt.

Eine geeignetere Datenstruktur für die effiziente Analyse von Graphen sind *Adjazenzlisten*. Hier werden an jedem Knoten u in einer linear verketteten Liste (siehe zum Beispiel Cormen et al. [CLRS09]) die mit u inzidenten Kanten gespeichert. Die in u einlaufenden Kanten können im Fall gerichteter Graphen getrennt von den aus u auslaufenden Kanten gespeichert werden. Adjazenzliste

Beispiel 3.9 (Adjazenzmatrix und Adjazenzliste). Die Adjazenzmatrix für den Graphen G aus Abb. 3.6 ist definiert durch

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

In der Listendarstellung werden die Knoten und Kanten als Objekte verwaltet. An jedem Knoten u werden in doppelt verketteten Listen die von u auslaufenden und die in u einlaufenden Kanten gespeichert. Die Knoten des Graphen werden ebenfalls

in einer doppelt verketteten Knotenliste gespeichert. Für den Beispielgraphen aus Abb. 3.6 ergibt sich:

Knotenliste:	$u_1 \leftrightarrow u_2 \leftrightarrow u_3 \leftrightarrow u_4 \leftrightarrow u_5$		
Kantenlisten:	$u_1.out : e_1 \leftrightarrow e_2$	$u_2.out : e_3$	$u_3.out : e_4 \leftrightarrow e_5$
	$u_1.in :$	$u_2.in : \quad e_1 \leftrightarrow e_3$	$u_3.in : \quad e_2 \leftrightarrow e_6$
	$u_4.out : e_6$	$u_5.out : e_7$	
	$u_4.in : \quad e_3 \leftrightarrow e_7$	$u_5.in : \quad e_5$	

Für einige Anwendungen ist es notwendig, mehrere gleich gerichtete Kanten zwischen denselben zwei Knoten zuzulassen. Daher werden gerichtete Graphen gelegentlich auch wie folgt definiert. Ein *gerichteter Graph* ist ein System $G = (V, E, \text{source}, \text{target})$ mit den folgenden Eigenschaften:

1. V ist eine endliche Menge von *Knoten*,
2. E ist eine endliche Menge von *Kanten*,
3. $\text{source} : E \to V$ ist eine Abbildung, die jeder Kante einen *Startknoten* zuordnet, und
4. $\text{target} : E \to V$ ist eine Abbildung, die jeder Kante einen *Zielknoten* zuordnet.

multiple Kante

In diesem Graphenmodel sind *multiple Kanten* möglich. Es kann also mehrere Kanten mit gleichem Start- und Zielknoten geben. Multiple Kanten können unterschiedliche Markierungen oder Gewichte mit Hilfe zusätzlicher Funktionen $f : E \to \mathbb{R}$ erhalten.

Im ungerichteten Fall werden die beiden Abbildungen source und target durch eine Abbildung

$$\text{vertices} : E \to \{\{u, v\} \mid u, v \in V, u \neq v\}$$

ersetzt.

Ebenfalls gebräuchlich sind Abbildungen der Art

$$\text{vertices} : E \to \{V' \mid V' \subseteq V, V' \neq \emptyset\}.$$

Hyperkante
Hypergraph

Kanten, die mit mehr als zwei Knoten inzident sind, werden *Hyperkanten* genannt. Graphen mit Hyperkanten heißen *Hypergraphen*.

einfacher Graph

Wir betrachten in diesem Buch jedoch vorwiegend *einfache* Graphen, also Graphen ohne multiple Kanten und ohne Schleifen, und auch keine Hypergraphen.

3.2 Spezielle Graphen und Grapheigenschaften

Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik.

Definition 3.10 (Wald, Baum und Wurzelbaum).

Wald
Baum, Blatt
innerer Knoten

1. *Ein ungerichteter Graph ohne Kreise ist ein* Wald. *Ein zusammenhängender Wald ist ein* Baum. *In einem Wald werden die Knoten vom Grad* 1 Blätter *genannt; die übrigen Knoten heißen* innere Knoten.

2. *Ein gerichteter Graph ohne Kreise, in dem jeder Knoten höchstens eine einlaufende Kante besitzt, ist ein* gerichteter Wald. *In einem gerichteten Wald werden die Knoten vom Ausgangsgrad* 0 *ebenfalls* Blätter *genannt. Knoten vom Eingangsgrad* 0 *werden* Wurzeln *genannt. Ein gerichteter Wald mit genau einer Wurzel ist ein* gerichteter Baum *oder auch* Wurzelbaum. *Ist* (u,v) *eine Kante in einem gerichteten Wald, so ist u der* Vorgänger von *v und v ein* Nachfolger *bzw. ein* Kind von *u.*

gerichteter Wald
Blatt, Wurzel
gerichteter Baum
Wurzelbaum
Vorgänger
Nachfolger, Kind

Beispiel 3.11 (Baum). Abbildung 3.7 zeigt einen Baum mit 17 Knoten.

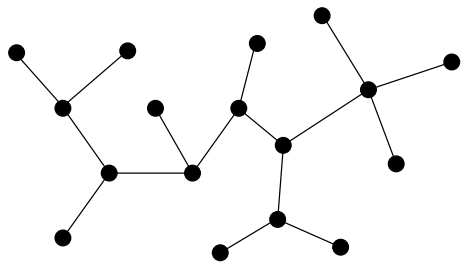

Abb. 3.7. Ein Baum

Die meisten Algorithmen für baumstrukturierte Graphen verwenden so genannte Bottom-up-Strategien. Diese bearbeiten einen baumstrukturierten Graphen, indem die Knoten in der zugehörigen Baumstruktur in Bottom-up-Reihenfolge inspiziert werden.

Definition 3.12 (Bottom-up-Reihenfolge). *Sei* $G = (V,E)$ *ein gerichteter Baum. Eine* Bottom-up-Reihenfolge *ist eine Anordnung der Knoten von G, in der für jede Kante* (u,v) *in E der Knoten v vor dem Knoten u kommt.*

Bottom-up-Reihenfolge

Beispiel 3.13 (Bottom-up-Reihenfolge). Die Nummerierung der Knoten von 1 bis 30 des Wurzelbaums in Abb. 3.8 ist eine Bottom-up-Reihenfolge.

Wir betrachten im Folgenden Knotenmengen eines Graphen, die bestimmte Eigenschaften haben. Diese Knotenmengen sind insbesondere für die Definition der Probleme nötig, die wir später algorithmisch lösen wollen. Informal lassen sich diese Mengen von Knoten folgendermaßen beschreiben. In einer *Clique* ist der Zusammenhalt der entsprechenden Knoten bezüglich der durch Kanten ausgedrückten Beziehung am größten: Jeder Knoten einer Clique ist mit jedem anderen Knoten der Clique verbunden. Im Gegensatz dazu ist dieser Zusammenhalt in einer *unabhängigen Menge* am kleinsten bzw. nicht vorhanden: Kein Knoten einer unabhängigen Menge U ist mit irgendeinem anderen Knoten aus U verbunden. Eine *Knotenüberdeckung* eines Graphen ist eine Knotenmenge, die von jeder Kante des Graphen mindestens einen Endpunkt enthält. Ein Graph wird von einer Teilmenge seiner Knoten *dominiert*, wenn jeder Knoten des Graphen entweder zu dieser Teilmenge gehört oder

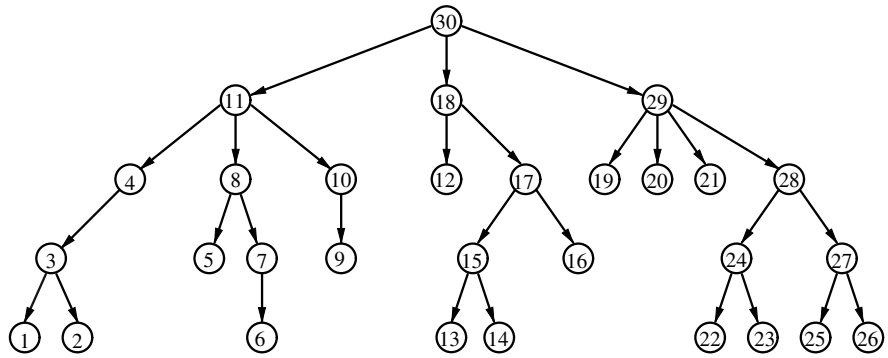

Abb. 3.8. Ein gerichteter Baum mit Bottom-up-Reihenfolge der Knoten

einen Nachbarn in ihr hat. Diese graphentheoretischen Begriffe werden nun formal definiert.

Definition 3.14 (Clique, unabhängige Menge, Knotenüberdeckung und dominierende Menge). *Seien $G = (V,E)$ ein ungerichteter Graph und $U \subseteq V$ eine Teilmenge der Knotenmenge von G.*

Clique
1. *U ist eine* Clique *in G, falls $\{u,v\} \in E$ für alle $u,v \in U$ mit $u \neq v$ gilt.*

vollständiger Graph K_n
2. *G ist* vollständig*, wenn V eine Clique in G ist. Der vollständige Graph mit $n \geq 0$ Knoten wird mit K_n bezeichnet.*

unabhängige Menge
3. *U ist eine* unabhängige Menge *(englisch:* independent set*) in G, falls $\{u,v\} \notin E$ für alle $u,v \in V$ mit $u \neq v$ gilt.*

Knotenüberdeckung
4. *U ist eine* Knotenüberdeckung *(englisch:* vertex cover*) in G, falls $\{u,v\} \cap U \neq \emptyset$ für alle Kanten $\{u,v\} \in E$ gilt.*

dominierende Menge
5. *U ist eine* dominierende Menge *(englisch:* dominating set*) in G, falls es für jeden Knoten $u \in V - U$ einen Knoten $v \in U$ mit $\{u,v\} \in E$ gibt.*

Die Größe *einer unabhängigen Menge (bzw. einer Clique, einer Knotenüberdeckung oder einer dominierenden Menge) $U \subseteq V$ ist die Anzahl der Knoten in U.*

Beispiel 3.15 (Clique, unabhängige Menge, Knotenüberdeckung und dominierende Menge). In Abb. 3.9 ist ein Graph mit einer (jeweils durch graue Knoten repräsentierten) Clique der Größe 3 und einer unabhängigen Menge der Größe 3 gezeigt, beide in demselben Graphen dargestellt, und es gibt keine größere Clique bzw. unabhängige Menge in diesem Graphen. Bei den entsprechenden Problemen (siehe Abschnitt 3.4) ist man an Cliquen bzw. an unabhängigen Mengen eines gegebenen Graphen interessiert, die *mindestens* eine vorgegebene Größe haben, oder sogar an *größten* Cliquen bzw. an *größten* unabhängigen Mengen.

größte Clique

größte unabhängige Menge

Bei anderen Problemen fragt man dagegen nach *maximalen* Cliquen bzw. nach *maximalen* unabhängigen Mengen in Graphen. Eine unabhängige Menge zum Beispiel heißt *maximal*, falls sie die Eigenschaft der Unabhängigkeit durch Hinzunahme eines beliebigen Knotens verlieren würde, d. h., falls sie keine echte Teilmenge einer anderen unabhängigen Menge des Graphen ist.

maximale Clique

maximale unabhängige Menge

Weiterhin zeigt Abb. 3.9 eine (wieder jeweils durch graue Knoten repräsentierte) Knotenüberdeckung der Größe 3 und eine dominierende Menge der Größe 2, beide in demselben Graphen dargestellt, und es gibt in diesem Graphen keine kleinere Knotenüberdeckung bzw. dominierende Menge. Hier geht es bei den entsprechenden Problemen (siehe Abschnitt 3.4) um Knotenüberdeckungen bzw. dominierende Mengen eines gegebenen Graphen, die *höchstens* eine vorgegebene Größe besitzen, oder sogar um *kleinste* bzw. *minimale* Knotenüberdeckungen und *kleinste* bzw. *minimale* dominierende Mengen in diesem Graphen.

kleinste Knotenüberdeckung
minimale Knotenüberdeckung
kleinste dominierende Menge
minimale dominierende Menge

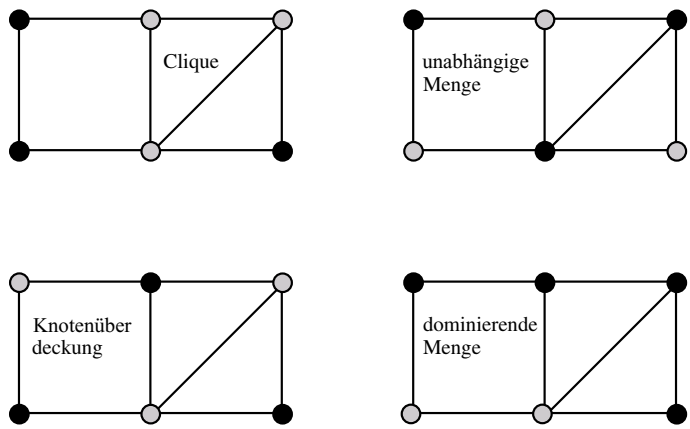

Abb. 3.9. Clique, Knotenüberdeckung, unabhängige Menge und dominierende Menge

Beispiel 3.16 (vollständiger Graph). In Abb. 3.10 sind ein vollständiger Graph mit vier Knoten und ein vollständiger Graph mit sechs Knoten dargestellt.

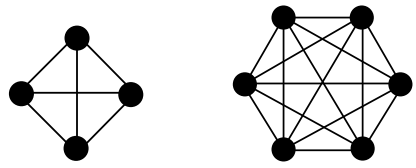

Abb. 3.10. Zwei vollständige Graphen, K_4 und K_6

Der Begriff der k-Färbbarkeit für Graphen wurde bereits in der Einleitung an einem Beispiel vorgestellt (siehe Abb. 1.1 und Abb. 1.2) und im zweiten Kapitel wurde in Beispiel 2.2 ein Algorithmus präsentiert, der testet, ob ein gegebener Graph zweifärbbar ist (siehe auch die Übungen 2.3 und 2.9). Nun definieren wir diesen Begriff formal.

Definition 3.17 (*k*-Färbbarkeit). *Sei $k \in \mathbb{N}^+$. Eine* k-Färbung *eines ungerichteten Graphen $G = (V,E)$ ist eine Abbildung $\psi : V \to \{1,\ldots,k\}$, wobei $1,\ldots,k$ die Farben repräsentieren. Eine k-Färbung ψ von G heißt* legal, *falls $\psi(u) \neq \psi(v)$ für jede Kante $\{u,v\}$ in E gilt. G heißt* k-färbbar, *falls es eine legale k-Färbung von G gibt. Die Mengen $\{v \in V \mid \psi(v) = i\}$, $1 \leq i \leq k$, heißen die* Farbklassen *von G.*

k-färbbar
Farbklasse

Alternativ zu Definition 3.17 kann man den Begriff der k-Färbbarkeit wie folgt charakterisieren: Ein Graph $G = (V,E)$ ist genau dann k-färbbar, wenn V in k unabhängige Mengen $U_1,\ldots,U_k$ partitioniert werden kann, also genau dann, wenn gilt:

1. $\bigcup_{i=1}^{k} U_i = V$,
2. $(\forall i,j \in \{1,\ldots,k\})\ \ [i \neq j \Longrightarrow U_i \cap U_j = \emptyset]$ und
3. für jedes $i \in \{1,\ldots,k\}$ ist U_i eine unabhängige Menge in G. Jedes U_i entspricht einer Farbklasse gemäß Definition 3.17.

Beispiel 3.18 (dreifärbbarer Graph). Abbildung 3.11 zeigt einen dreifärbbaren Graphen, der (statt mit 1, 2 und 3) mit den Farben weiß, schwarz und grau legal gefärbt ist. Eine legale Färbung dieses Graphen mit weniger Farben ist nicht möglich. Bei Graphfärbbarkeitsproblemen möchte man stets mit möglichst wenigen Farben auskommen.

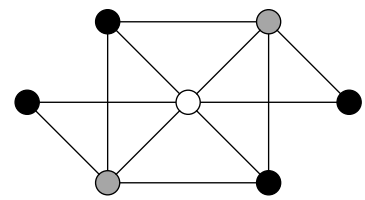

Abb. 3.11. Ein dreifärbbarer Graph

Übung 3.19. Zeigen Sie, dass unter allen minimalen legalen Färbungen φ eines gegebenen Graphen G (das heißt, φ benutzt genau $\chi(G)$ Farben) wenigstens eine sein muss, die eine *maximale* unabhängige Menge als Farbklasse enthält.

Komplementgraph

Definition 3.20 (Komplementgraph). *Der* Komplementgraph *zu einem ungerichteten Graphen $G = (V,E)$ ist definiert durch*

$$\overline{G} = (V, \{\{u,v\} \mid u,v \in V, u \neq v, \{u,v\} \notin E\})$$

und enthält somit genau die Kanten, die in G nicht enthalten sind.

Beispiel 3.21 (Komplementgraph). Abbildung 3.12 zeigt einen Graphen G und den zugehörigen Komplementgraphen $\overline{G}$.

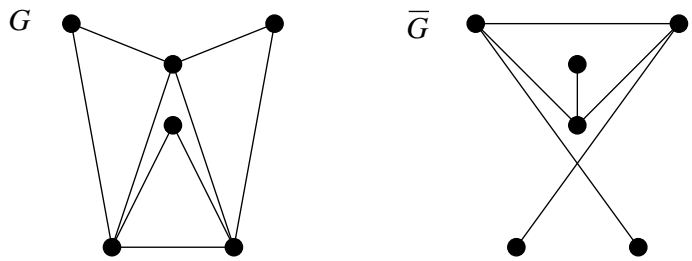

Abb. 3.12. Ein Graph G und der zugehörige Komplementgraph $\overline{G}$

Definition 3.22 (k-partiter und vollständig k-partiter Graph). *Ein Graph wird als* k-partit *bezeichnet, falls sich seine Knotenmenge in k Teilmengen einteilen lässt, sodass für jede Kante aus G die beiden Endknoten in verschiedenen Teilmengen liegen. Besonders häufig wird der Fall $k = 2$ betrachtet. 2-partite bzw. 3-partite Graphen werden auch* bipartite *bzw.* tripartite Graphen *genannt.*

k-partiter Graph

bipartiter Graph

tripartiter Graph

Sind in einem k-partiten Graphen je zwei Knoten aus verschiedenen der k Teilknotenmengen adjazent, so heißt der Graph auch vollständig k-partit. *Falls die k Knotenmengen in einem vollständig k-partiten Graphen die Mächtigkeiten $n_1, \ldots, n_k$ besitzen, so kürzt man den Graphen auch mit $K_{n_1,\ldots,n_k}$ ab. Vollständig bipartite Graphen $K_{1,n}$ werden auch als* Sterne *bezeichnet.*

vollständig k-partiter Graph

Stern

Beispiel 3.23 (vollständig k-partiter Graph). Abbildung 3.13 zeigt die vollständig bipartiten Graphen $K_{1,6}$, $K_{2,3}$ und $K_{4,4}$ und den vollständig tripartiten Graphen $K_{1,2,2}$.

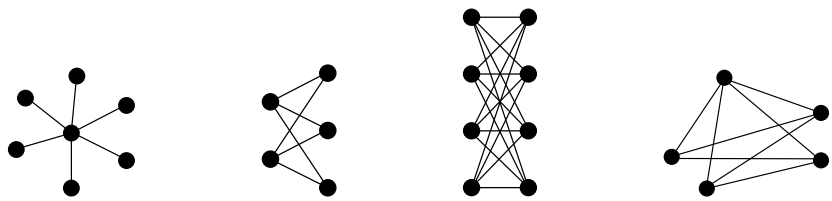

Abb. 3.13. Drei vollständig bipartite Graphen und ein vollständig tripartiter Graph

Ein Graph ist genau dann zweifärbbar, wenn er bipartit ist. Im Grunde testet der Zweifärbbarkeitsalgorithmus in Beispiel 2.2 also, ob der gegebene Graph bipartit ist.

Übung 3.24. Zeigen Sie, dass ein ungerichteter Graph genau dann bipartit ist, wenn er keinen Kreis ungerader Länge enthält.

Der Begriff des Kreises in einem Graphen wurde in Definition 3.3 definiert. Ein Kreis der Länge m wird üblicherweise mit

$$C_m = (\{v_1, \ldots, v_m\}, \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{m-1}, v_m\}, \{v_m, v_1\}\})$$

bezeichnet.

Beispiel 3.25 (Kreis). Abbildung 3.14 zeigt drei Kreise.

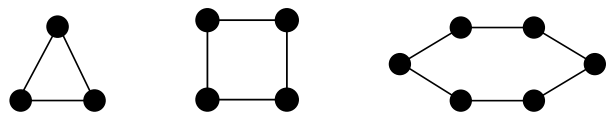

Abb. 3.14. Die Kreise C_3, C_4 und C_6

Man sagt, ein Graph enthält einen Kreis, falls er einen Kreis C_m für ein $m \geq 3$ als Teilgraphen enthält. Enthält ein Graph $G = (V,E)$ einen Kreis (bzw. Weg) mit $|V|$ Knoten, so heißt dieser auch *Hamilton-Kreis* (bzw. *Hamilton-Weg*) in G.

Hamilton-Kreis
Hamilton-Weg

Wie oben erwähnt, ist die Eigenschaft der Zweifärbbarkeit für einen Graphen äquivalent zur Frage, ob er bipartit ist. Ebenfalls äquivalent dazu ist die Eigenschaft des Graphen, keine Kreise ungerader Länge zu enthalten, siehe Übung 3.24.

Gittergraph

Definition 3.26 (Gittergraph). *Für $n,m \geq 0$ ist der* Gittergraph *$G_{n,m}$ definiert durch*

$$G_{n,m} = (\{1,\ldots,n\}\times\{1,\ldots,m\}, \{\{(i,j),(i',j')\} \mid |i-i'|+|j-j'|=1\}),$$

Absolutbetrag einer Zahl

wobei $|x|$ den Absolutbetrag einer Zahl $x \in \mathbb{Z}$ bezeichnet.

Beispiel 3.27 (Gittergraph). Abbildung 3.15 zeigt zwei Gittergraphen.

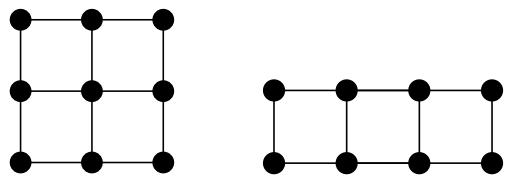

Abb. 3.15. Zwei Gittergraphen, $G_{3,3}$ und $G_{2,4}$

Graphparameter

Ein *Graphparameter* ist eine Funktion, die jedem Graphen eine natürliche Zahl zuordnet. Beispiele für Graphparameter sind die oben definierten Funktionen Δ und δ, die den maximalen bzw. minimalen Knotengrad von Graphen angeben.

Graphklasse

Eine Menge von Graphen, die alle eine gemeinsame Eigenschaft besitzen, bezeichnet man als *Graphklasse*. Es gibt verschiedene Möglichkeiten, Graphklassen zu definieren. Häufig werden Graphklassen betrachtet, die sich durch den Ausschluss spezieller Graphen als induzierte Teilgraphen beschreiben lassen (siehe dazu auch Definition 6.26 in Abschnitt 6.3). Oben haben wir bereits die Graphklasse der Wälder kennen gelernt, zu der alle Graphen gehören, die keine Kreise enthalten. Ebenfalls durch Ausschluss bestimmter induzierter Teilgraphen lässt sich die Klasse der *Co-Graphen* definieren, in die alle Graphen fallen, die keinen P_4 – also keinen Weg

Co-Graph

mit vier Knoten – als induzierten Teilgraphen enthalten. Eine alternative Definition von Co-Graphen über den Abschluss unter bestimmten Operationen auf Graphen wird in Abschnitt 9.2.1 angegeben, siehe Definition 9.14. Eine Graphklasse $\mathscr{G}$ heißt *abgeschlossen bezüglich einer Operation f auf Graphen* (wie z. B. der Teilgraphenbildung, Komplementbildung usw.), falls für alle Graphen $G \in \mathscr{G}$ auch der Graph $f(G)$ in $\mathscr{G}$ liegt. Wir werden noch häufig Graphklassen betrachten, zu denen alle Graphen G mit $\beta(G) \leq k$ gehören, wobei β ein Graphparameter und k eine Konstante ist. So erhält man etwa für den oben definierten Graphparameter Δ (den maximalen Knotengrad von Graphen) und $k = 3$ die so genannten *kubischen Graphen*.

Abschluss einer Graphklasse

kubischer Graph

3.3 Einige Algorithmen für Graphen

In diesem Abschnitt werden einige elementare Strategien und Entwurfstechniken für Algorithmen auf Graphen beschrieben, die später benötigt werden. Graphen spielen dabei nicht nur als Teil der Eingabe der zu lösenden Probleme eine Rolle, sondern auch zum Beispiel hinsichtlich der Struktur des Lösungsraums, in dem wir nach einer Lösung des betrachteten Problems suchen. Somit sind diese Strategien und Algorithmen nicht nur für Graphenprobleme an sich nützlich, sondern auch für andere Probleme (z. B. das Erfüllbarkeitsproblem der Aussagenlogik, siehe Kapitel 4). Ebenso kann man mit Graphen die Ablaufstruktur eines Algorithmus beschreiben, etwa mittels Flussdiagrammen (die in diesem Buch allerdings nicht betrachtet werden) oder als Rekursionsbäume, die die Verschachtelung der rekursiven Aufrufe eines rekursiven Algorithmus darstellen. Abschnitt 3.5 behandelt grundlegende Algorithmenentwurfstechniken, die bei der Lösung von Graphenproblemen Anwendung finden. Weder die Liste der hier vorgestellten Durchlaufstrategien noch die der Algorithmenentwurfstechniken ist als eine vollständige Abhandlung zu verstehen.

3.3.1 Topologische Anordnungen

Eine topologische Knotenordnung für einen gerichteten Graphen $G = (V, E)$ ist eine Nummerierung der Knoten von G, sodass alle Kanten von Knoten mit kleineren Nummern zu Knoten mit größeren Nummern gerichtet sind. Formal ist eine *topologische Knotenordnung* für G eine bijektive Abbildung $h : V \to \{1, \ldots, |V|\}$, sodass $h(u) < h(v)$ für alle Kanten (u, v) in E gilt.

topologische Knotenordnung

Satz 3.28. *Ein gerichteter Graph besitzt genau dann eine topologische Knotenordnung, wenn er kreisfrei ist.*

Beweis. Aus der Existenz einer topologischen Knotenordnung folgt unmittelbar, dass G kreisfrei ist. Dass es für jeden kreisfreien Graphen tatsächlich eine topologische Knotenordnung gibt, folgt aus der folgenden induktiven Überlegung über die Anzahl n der Knoten von G.

Induktionsanfang: Für kreisfreie Graphen mit genau einem Knoten u ist $h(u) = 1$ eine topologische Knotenordnung.

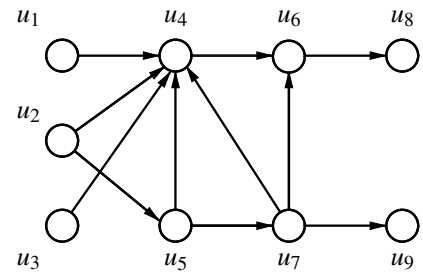

Abb. 3.16. Eine topologische Knotenordnung für G ist $u_1, u_2, u_5, u_3, u_7, u_4, u_6, u_9, u_8$.

Induktionsschritt: Seien nun $n > 1$ und u ein Knoten mit $\text{outdeg}(u) = 0$. Ein solcher Knoten existiert immer, wenn G kreisfrei ist. Er kann wie folgt gefunden werden. Starte bei einem beliebigen Knoten u und verfolge die auslaufenden Kanten bis zu einem Knoten, der keine auslaufenden Kanten hat.

Sei $G' = G[V - \{u\}]$ der Graph G ohne den Knoten u. Jede topologische Knotenordnung h für G' mit $n-1$ Knoten, erweitert durch $h(u) = n$, ist eine topologische Knotenordnung für G. ❑

In dem induktiven Beweis von Satz 3.28 ist bereits eine algorithmische Idee für die effiziente Konstruktion einer topologischen Knotenordnung enthalten. Diese Idee kann einfach mit Hilfe einer Datenstruktur D umgesetzt werden, in der Kanten gespeichert werden. Und zwar werden nur Kanten (u,v) in D gespeichert, deren Startknoten u bereits eine Nummer erhalten hat, deren Zielknoten v jedoch noch nicht. Wie diese Datenstruktur im Detail realisiert ist, spielt keine Rolle. Es muss lediglich möglich sein, eine Kante effizient mit einer Methode insert$(D,(u,v))$ einzufügen und eine beliebige Kante effizient mit einer Methode extract(D) zu entfernen. Zu Beginn werden alle Knoten ohne einlaufende Kanten nummeriert und ihre auslaufenden Kanten in D eingefügt. Solange die Datenstruktur D nicht leer ist, wird eine beliebige Kante (u,v) aus D entnommen. Haben bereits alle Vorgänger von v eine Nummer erhalten, so bekommt v die nächste noch nicht vergebene Knotennummer und seine auslaufenden Kanten werden zu D hinzugefügt. Um einfach zu überprüfen, ob alle Vorgänger von v bereits eine Nummer erhalten haben, speichern wir an jedem Knoten u eine Variable $I[u]$, die initial dem Eingangsgrad von u entspricht und nach jeder Ausgabe eines Vorgängers um 1 erniedrigt wird. Wird die Variable $I[v]$ irgendwann 0, so wurde die letzte einlaufende Kante von u betrachtet und der Knoten v kann seine Nummer erhalten.

```
1: i := 1;
2: for all u ∈ V {
3:     I[u] := indeg(u);
4:     if (I[u] = 0) then {
5:         h(u) := i;
```

```
 6:        i := i + 1;
 7:        for all (u,v) ∈ E {
 8:           insert(D,(u,v)); } } }
 9:  while (D ≠ ∅) {
10:     (u,v) := extract(D);
11:     I[v] := I[v] − 1;
12:     if (I[v] = 0) then {
13:        h(v) := i;
14:        i := i + 1;
15:        for all (v,w) ∈ E {
16:           insert(D,(v,w)); } } }
```

Der Algorithmus fügt jede Kante höchstens ein Mal in die Datenstruktur D ein. Jede Kante wird höchstens ein Mal inspiziert. Die Anzahl der Schritte in dem obigen Algorithmus ist somit proportional zur Anzahl der Knoten und Kanten von G, also aus $\mathcal{O}(|V|+|E|)$. Enthält der Graph Kreise, so werden die Kanten, die sich auf Kreisen befinden, nie in die Datenstruktur D eingefügt. Es muss also zum Schluss noch überprüft werden, ob tatsächlich alle Knoten nummeriert wurden. Das ist genau dann der Fall, wenn der Graph keinen Kreis enthält. Der obige Algorithmus entscheidet also ebenfalls in linearer Zeit, ob ein gerichteter Graph einen Kreis enthält.

Übung 3.29. Eine *topologische Kantenordnung* für einen gerichteten Graphen $G = (V,E)$ ist eine bijektive Abbildung $h : E \to \{1,\ldots,|E|\}$, sodass $h((u,v)) < h((v,w))$ für je zwei Kanten (u,v) und (v,w) in E gilt.

topologische Kantenordnung

(a) Zeigen Sie, dass eine topologische Kantenordnung genau dann existiert, wenn der gerichtete Graph G kreisfrei ist.
(b) Modifizieren Sie den Algorithmus für die Konstruktion einer topologischen Knotenordnung so, dass er in linearer Zeit eine topologische Kantenordnung konstruiert.

3.3.2 Durchlaufordnungen für Graphen

Angenommen, ein oder mehrere Knoten eines gegebenen Graphen enthalten Schätze, die wir finden wollen. Stellt der Graph einen Lösungsraum für ein Problem dar, so ist mit „Schatz“ einfach eine Lösung der gegebenen Probleminstanz gemeint. Im Folgenden beschreiben wir verschiedene einfache Strategien, mit denen der Graph auf der Suche nach einem Schatz durchlaufen werden kann.

Diese Strategien sind „blind“ in dem Sinn, dass sie keine zusätzliche Information über das betrachtete Problem zur Beschleunigung der Suche verwenden, wie es beispielsweise Heuristiken oft tun. Stattdessen gehen sie „stur“ immer gleich vor, egal, welche Struktur der Graph besitzt und welches Problem zu lösen ist. Das Ziel der Suche ist es, in einem gerichteten Graphen $G = (V,E)$ alle von einem Startknoten s erreichbaren Knoten zu finden.

Für jeden Knoten u verwenden wir eine Variable $b[u]$, die am Anfang 0 ist. Das bedeutet, dass diese Knoten bei der Suche bis jetzt noch nicht gefunden wurden. Für den Startknoten s setzen wir $b[s]$ auf 1 und nehmen alle aus s auslaufenden Kanten mit insert$(D,(s,v))$ in eine Datenstruktur D auf. Solange die Datenstruktur D nicht leer ist, wird eine beliebige Kante (u,v) mit extract(D) aus D entnommen. Wurde der Zielknoten v der Kante (u,v) noch nicht besucht, ist also $b[v]=0$, so werden alle aus v auslaufenden Kanten ebenfalls zu D hinzugefügt, und es wird $b[v]$ auf 1 gesetzt. Wenn die Datenstruktur D keine weiteren Kanten mehr enthält, ist in dem Graphen G ein Knoten u genau dann vom Startknoten s erreichbar, wenn die Variable $b[u]$ auf 1 gesetzt ist.

```
 1: for all u ∈ V {
 2:    b[u] := 0; }
 3: b[s] := 1;
 4: for all (s,v) ∈ E {
 5:    insert(D,(s,v)); }
 6: while (D ≠ ∅) {
 7:    (u,v) := extract(D);
 8:    if (b[v] = 0) then {
 9:       for all (v,w) ∈ E {
10:          insert(D,(v,w)); }
11:       b[w] := 1; } }
```

In der Datenstruktur D speichern wir diejenigen Kanten, über die möglicherweise bisher noch unbesuchte Knoten erreicht werden können. Jede Kante wird höchstens ein Mal in D eingefügt. Jeder Knoten wird höchstens ein Mal inspiziert. Die Laufzeit ist daher proportional zur Anzahl der Knoten plus der Anzahl der vom Startknoten s erreichbaren Kanten, also aus $\mathcal{O}(|V|+|E|)$, falls das Einfügen in D und das Entfernen aus D in konstanter Zeit möglich ist.

Durchlaufordnung

Der Typ der Datenstruktur D bestimmt die *Durchlaufordnung*. Ist D ein *Stack* (first in, last out) (deutsch: *Kellerspeicher*), dann werden die Knoten in einer *Tiefensuche* durchlaufen. Ist D eine *Queue* (first in, first out) (deutsch: *Liste*), dann werden die Knoten in einer *Breitensuche* durchlaufen.

Tiefensuche

Breitensuche

DFS-Nummer

BFS-Nummer

Werden bei der Tiefen- bzw. bei der Breitensuche die Knoten in der Reihenfolge nummeriert, in der sie besucht werden, so erhält man eine *DFS-Nummerierung* (englisch: *depth-first-search*) bzw. *BFS-Nummerierung* (englisch: *breadth-first-search*). Neben den Anfangsnummerierungen ist insbesondere bei der Tiefensuche auch die *DFS-End-Nummerierung* oft sehr nützlich. Sie nummeriert die Knoten in der Reihenfolge, in der ihre auslaufenden Kanten vollständig abgearbeitet wurden.

DFS-End-Nummer

Da die Tiefensuche einen Stack als Datenstruktur verwendet, kann sie relativ einfach rekursiv implementiert werden. Der Stack wird dabei durch das Betriebssystem realisiert, das bei jedem rekursiven Aufruf alle Variablenwerte sichert.

```
1: for all u ∈ V {
2:    b[u] := 0; }
```

```
3:  c1 := 1;
4:  c2 := 1;
5:  Tiefensuche(s);

Tiefensuche(u)
1:  b[u] := 1;
2:  dfs[u] := c1;
3:  c1 := c1 + 1;
4:  for all (u,v) ∈ E {
5:    if (b[v] = 0) then {
6:      Tiefensuche(v); } }
7:  dfs-end[u] := c2;
8:  c2 := c2 + 1;
```

Abbildung 3.17 zeigt eine Reihenfolge, in der die Knoten eines Graphen bei der Tiefensuche durchlaufen werden. Diese Reihenfolge ist nicht immer eindeutig.

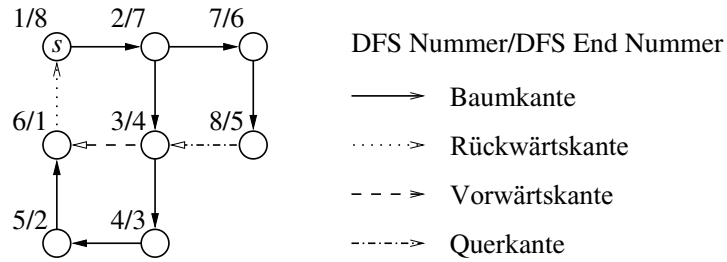

Abb. 3.17. Eine Tiefensuche, gestartet bei Knoten s. Die Knoten sind mit der DFS-Nummer und der DFS-End-Nummer beschriftet.

Die Kanten in einem gerichteten Graphen können entsprechend ihrer Rolle, die sie bei der Tiefensuche spielen, wie folgt aufgeteilt werden:

1. Kanten, denen die Tiefensuche folgt, sind *Baumkanten*. Die Baumkanten bilden den *Tiefensuche-Wald*. — Baumkanten, Tiefensuche-Wald
2. Kanten (u,v) mit $\mathrm{DFS}[v] > \mathrm{DFS}[u]$, die keine Baumkanten sind, sind *Vorwärtskanten*. Die Vorwärtskanten kürzen Wege im Tiefensuche-Baum ab. — Vorwärtskanten
3. Kanten (u,v) mit $\mathrm{DFS}[v] < \mathrm{DFS}[u]$ und $\mathrm{DFE}[v] < \mathrm{DFE}[u]$ sind *Querkanten*. — Querkanten
4. Kanten (u,v) mit $\mathrm{DFS}[v] < \mathrm{DFS}[u]$ und $\mathrm{DFE}[v] > \mathrm{DFE}[u]$ sind *Rückwärtskanten*. Rückwärtskanten bilden zusammen mit den Wegen im Tiefensuche-Baum Kreise. — Rückwärtskanten

Im Gegensatz zur Tiefensuche betrachtet die Breitensuche alle auslaufenden Kanten eines Knotens unmittelbar hintereinander. Die Knoten werden somit stufenweise in Abhängigkeit von der Entfernung zum Startknoten als besucht markiert. Ein

Knoten, der über k Kanten vom Startknoten erreichbar ist, aber nicht über $k-1$ Kanten, wird erst dann gefunden, wenn alle Knoten besucht wurden, die über weniger als k Kanten vom Startknoten s erreichbar sind.

Abbildung 3.18 zeigt eine Reihenfolge, in der die Knoten des schon in Abb. 3.17 betrachteten Graphen bei einer Breitensuche durchlaufen werden. Auch hier ist die Reihenfolge nicht eindeutig. Wie man sieht, durchläuft man dabei den Graphen bezüglich eines Startknotens s Stufe für Stufe, bis alle vom Startknoten erreichbaren Knoten besucht wurden.

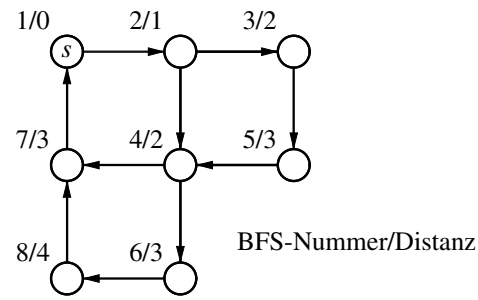

Abb. 3.18. Eine Breitensuche, gestartet bei Knoten s. Die Knoten sind mit der BFS-Nummer und der Distanz zum Startknoten s beschriftet. Die Distanz zum Knoten s ist die kleinste Anzahl von Kanten, über die ein Knoten von s erreichbar ist.

Die Tiefensuche wie auch die Breitensuche hat eine Worst-case-Laufzeit aus $\mathcal{O}(|V|+|E|)$. Beide Durchlaufstrategien können natürlich auch für ungerichtete Graphen implementiert werden. Für viele Anwendungen ist es einfacher, in der Datenstruktur D Knoten anstatt Kanten zu verwalten. In diesen Fällen ist der benötigte zusätzliche Speicherplatzbedarf oft nur $\mathcal{O}(|V|)$ und nicht $\mathcal{O}(|E|)$. Dies trifft übrigens auch für die oben angegebene rekursive Tiefensuche zu.

Übung 3.30.

1. Entwerfen Sie einen rekursiven Algorithmus für die Tiefensuche, sodass die verwendete Datenstruktur D Knoten anstatt Kanten speichert. Die Worst-case-Laufzeit der Lösung soll linear bleiben.
2. Entwerfen Sie einen iterativen Algorithmus für die Breitensuche, sodass die verwendete Datenstruktur D Knoten anstatt Kanten speichert. Die Worst-case-Laufzeit der Lösung soll linear bleiben.
3. Erweitern Sie die beiden obigen Algorithmen für ungerichtete Graphen. Hier ist zu beachten, dass eine Kante von jedem der beiden Endknoten betrachtet werden kann.

Übung 3.31. In einem gerichteten Graphen G sei e eine *potenzielle Vorwärtskante*, wenn es einen Knoten s gibt, sodass Kante e bei einer Tiefensuche gestartet bei s eine Vorwärtskante ist. Analog seien die Begriffe *potenzielle Rückwärtskante* und *potenzielle Querkante* definiert. Gesucht ist

1. ein Graph G_1, in dem es eine Kante gibt, die eine potenzielle Vorwärtskante, eine potenzielle Rückwärtskante, aber keine potenzielle Querkante ist,
2. ein Graph G_2, in dem es eine Kante gibt, die eine potenzielle Vorwärtskante, eine potenzielle Querkante, aber keine potenzielle Rückwärtskante ist,
3. ein Graph G_3, in dem es eine Kante gibt, die eine potenzielle Rückwärtskante, eine potenzielle Querkante, aber keine potenzielle Vorwärtskante ist, und
4. ein Graph G_4, in dem es eine Kante gibt, die eine potenzielle Vorwärtskante, eine potenzielle Rückwärtskante und eine potenzielle Querkante ist.

Übung 3.32. Entwerfen Sie einen Algorithmus, der mit Hilfe einer Tiefensuche (nicht mit einer Breitensuche) in einem gerichteten Graphen die Distanz von einem Startknoten s zu allen übrigen Knoten berechnet.

3.3.3 Zusammenhangsprobleme

Komponente

Eine *Komponente* in einem Graphen ist ein maximaler induzierter Teilgraph, der eine gegebene Zusammenhangseigenschaft erfüllt. Zum Beispiel ist eine Zusammenhangskomponente ein maximaler induzierter Teilgraph von G, der zusammenhängend ist, und eine starke Zusammenhangskomponente ist ein maximaler induzierter Teilgraph von G, der stark zusammenhängend ist.

Die Eigenschaft, ob ein ungerichteter Graph zusammenhängend oder ein gerichteter Graph schwach zusammenhängend ist, kann einfach mit einer Tiefen- oder Breitensuche in linearer Zeit überprüft werden. Die Tiefensuche kann aber auch für den Test auf starken Zusammenhang benutzt werden. Da die starken Zusammenhangskomponenten knotendisjunkt sind, können sie einfach durch ihre Knotenmengen spezifiziert werden. Die folgende Tiefensuche berechnet die Knotenmengen der starken Zusammenhangskomponenten. Dazu werden die besuchten Knoten auf einem Stack abgelegt. Über Rückwärtskanten wird die kleinste DFS-Nummer der Knoten gesucht, die von einem gegebenen Knoten u erreichbar sind. Dieser Wert wird in der Variablen min-dfs$[u]$ gespeichert. Sind alle Kanten an einem Knoten u abgearbeitet und ist danach min-dfs$[u]$ = DFS$[u]$, so werden alle Knoten auf dem Stack bis einschließlich u ausgegeben. Diese Knoten induzieren eine starke Zusammenhangskomponente von G.

```
1:  for all u ∈ V {
2:      b[u] := 0; }
3:  c := 1;
4:  initialisiere einen Stack S;
5:  for all u ∈ V {
6:      if (b[u] = 0) then {
7:          Starker-Zusammenhang(u); } }
```

```
Starker-Zusammenhang(u)
1:  b[u] := 1;
```

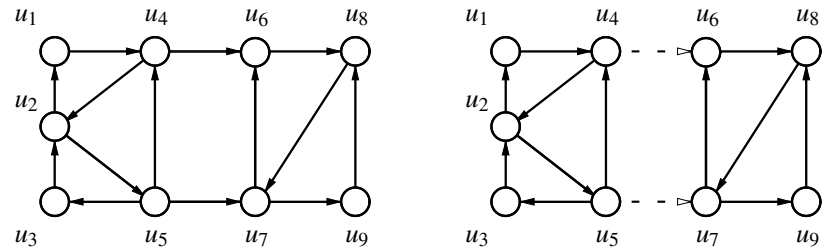

Abb. 3.19. Ein Graph mit zwei starken Zusammenhangskomponenten, induziert durch die Knotenmengen $\{u_1,\ldots,u_5\}$ und $\{u_6,\ldots,u_9\}$

```
 2:  dfs[u] := c;
 3:  c := c + 1;
 4:  min-dfs[u] = dfs[u];
 5:  lege Knoten u auf Stack S;
 6:  for all (u,v) ∈ E {
 7:    if (b[v] = 0) then {
 8:      Starker-Zusammenhang(v);
 9:      min-dfs[u] := min{min-dfs[u], min-dfs[v]}; }
10:    else {
11:      if (v auf Stack S) then {
12:        min-dfs[u] := min{min-dfs[u], dfs[v]}; } } }
13:  if (min-dfs[u] = DFS[u]) then {
14:    nimm alle Knoten bis einschließlich u von Stack S; }
```

In Zeile 11 wird überprüft, ob sich der Zielknoten v auf dem Stack befindet. Dies ist bei Rückwärtskanten immer der Fall, aber bei Vorwärts- und Querkanten genau dann, wenn sie zu einer noch nicht ausgegebenen starken Zusammenhangskomponente führen. Der obige Algorithmus zur Berechnung der starken Zusammenhangskomponenten hat offensichtlich eine Worst-case-Laufzeit in $\mathcal{O}(|V|+|E|)$.

Die starken Zusammenhangskomponenten eines gerichteten Graphen $G=(V,E)$ können aber auch mit zwei entgegengesetzten Tiefensuche-Durchläufen in linearer Zeit bestimmt werden. Dazu werden zuerst mit einer „normalen“ Tiefensuche für alle Knoten DFS-End-Nummern bestimmt. Dann werden alle Kanten umgedreht,

invertierter Graph

d. h., die neue Kantenmenge im *invertierten Graphen* ist $E' = \{(v,u) \mid (u,v) \in E\}$. Sei s_1 der Knoten mit größter DFS-End-Nummer. Dann wird eine Tiefensuche auf (V,E') mit Startknoten s_1 durchgeführt. Die von s_1 erreichbaren Knoten sind genau die Knoten der starken Zusammenhangskomponente von s_1. Die nächste Tiefensuche startet am nächsten Knoten mit größter DFS-End-Nummer, der durch die vorherigen Durchläufe noch nicht besucht wurde. Die von s_2 erreichbaren Knoten sind genau die Knoten der starken Zusammenhangskomponente von s_2 usw.

Warum berechnet dieser Algorithmus die starken Zusammenhangskomponenten? Sei max-dfe(u) die größte DFS-End-Nummer der Knoten in der starken Zusammenhangskomponente eines Knotens u. Wenn es einen Weg von einem Knoten

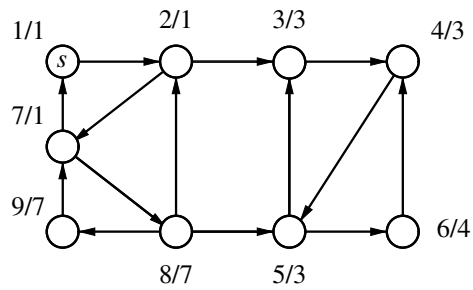

Abb. 3.20. Eine Tiefensuche, gestartet bei s, in der zwei starke Zusammenhangskomponenten bestimmt werden. Die Knoten sind mit der DFS-Nummer und der minimalen DFS-Nummer beschriftet, die nach dem vollständigen Durchlauf an den Knoten zurückbleiben.

u zu einem Knoten v gibt und v nicht in der gleichen starken Zusammenhangskomponente wie u ist, dann ist max-dfe$(u) >$ max-dfe(v). Dies folgt aus der Definition der DFS-End-Nummerierung. Sei s_1 nun der Knoten mit größter DFS-End-Nummer, dann kann es natürlich keinen Knoten u mit max-dfe$(u) >$ max-dfe(s_1) geben. Also müssen alle Knoten, die einen Weg zum Knoten s_1 haben, in der starken Zusammenhangskomponente von s_1 sein. Andererseits sind die Knoten, die keinen Weg zu s_1 haben, selbstverständlich auch nicht in der starken Zusammenhangskomponente von s_1. Somit ist ein Knoten genau dann in der starken Zusammenhangskomponente von s_1, wenn er in G einen Weg zu s_1 hat. Die Knoten, die in G einen Weg zu s_1 haben, können einfach mit einer Tiefensuche von s_1 im invertierten Graphen $G = (V, E')$ bestimmt werden.

Übung 3.33. Zeigen Sie, dass der obige Algorithmus nicht korrekt die starken Zusammenhangskomponenten berechnet, wenn die **if**-Abfrage in Zeile 11 durch den Test auf eine „Rückwärtskante" ersetzt wird:

11: **if** $((u,v)$ Rückwärtskante) **then** $\{$

Die Tiefensuche eignet sich auch für die Berechnung der zweifachen Zusammenhangskomponenten in einem ungerichteten Graphen $G = (V, E)$. Ein Knoten u ist ein *Trennungspunkt* bzw. *Artikulationspunkt* in G, wenn der Graph G ohne u mehr zusammenhängende Komponenten besitzt als G. Eine ungerichtete Kante $\{u, v\}$ ist eine *Brücke*, wenn beide Endknoten u und v Trennungspunkte in G sind. Jede Kante ist entweder in genau einer zweifachen Zusammenhangskomponente von G oder eine Brücke. Da die zweifachen Zusammenhangskomponenten keine disjunkten Knotenmengen besitzen, spezifizieren wir sie durch ihre Kantenmengen. Wir ordnen jeder Kante eine zweifache Zusammenhangskomponente oder die Eigenschaft „Brücke" zu.

Trennungspunkt
Artikulationspunkt
Brücke

Die zweifachen Zusammenhangskomponenten können mit Hilfe der folgenden Beobachtung ermittelt werden. Seien $G = (V, E)$ ein zusammenhängender ungerichteter Graph und $E' = (V', E')$ eine zweifache Zusammenhangskomponente von

G. Eine Tiefensuche auf G partitioniert die Kanten in Baumkanten, Kanten, denen die Tiefensuche folgt, und Nicht-Baumkanten. Die Einteilung in Vorwärtskanten, Rückwärtskanten und Querkanten sind für ungerichtete Graphen nicht definiert. Die Baumkanten aus E' bilden einen zusammenhängenden Teilbaum T' des „ungerichteten" Tiefensuche-Baums. Die Knoten aus V' sind die Knoten des Teilbaums T'. Sei $u'_{\text{min-dfs}} \in V'$ der Knoten aus V' mit der kleinsten DFS-Nummer. Es gibt genau eine Baumkante in E', die $u'_{\text{min-dfs}}$ mit einem weiteren Knoten aus V' verbindet. Besteht E' lediglich aus dieser einen Kante, so ist dies eine Brücke. Ansonsten gibt es mindestens eine weitere Kante in E', die keine Baumkante ist, aber einen Knoten aus V' mit $u'_{\text{min-dfs}}$ verbindet. Über diese Kante kann die kleinste DFS-Nummer der Knoten aus V' ermittelt werden.

Die folgende Suche bestimmt die Kantenmengen der zweifachen Zusammenhangskomponenten:

```
1:  for all u ∈ V {
2:     b[u] := 0; }
3:  c := 1;
4:  initialisiere einen Stack S;
5:  for all u ∈ V {
6:     if (b[u] = 0) then {
7:        Zweifacher-Zusammenhang(u); } }
```

```
Zweifacher-Zusammenhang(u)
 1:  b[u] := 1;
 2:  dfs[u] := c;
 3:  c := c + 1;
 4:  min-dfs[u] = dfs[u];
 5:  for all {u,v} ∈ E {
 6:     if (b[v] = 0) then {
 7:        lege Kante {u,v}) auf Stack S;
 8:        Zweifacher-Zusammenhang(v);
 9:        if (min-dfs[v] ≥ dfs[u]) then {
10:           nimm alle Kanten bis einschließlich {u,v} von Stack S; }
11:        else {
12:           min-dfs[u] := min(min-dfs[u], min-dfs[v]); } }
13:     else if ({u,v} wurde noch nie auf Stack S gelegt) then {
14:        lege Kante {u,v} auf Stack S;
15:        min-dfs[u] := min(min-dfs[u], dfs[v]); } }
```

In Zeile 11 wird min-dfs$[v]$ mit dfs$[u]$ verglichen. Sind die beiden Werte gleich, so werden alle Kanten der zweifachen Zusammenhangskomponente bis zur Kante $\{u,v\}$ vom Stack genommen und als eine zweifache Zusammenhangskomponente ausgegeben. Ist min-dfs$[v]$ größer als dfs$[u]$), dann wird lediglich eine Brücke ausgegeben. Der Algorithmus hat ebenfalls eine Worst-case-Laufzeit in $\mathcal{O}(|V|+|E|)$.

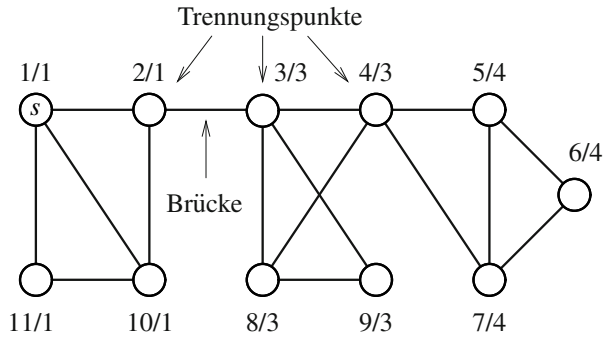

Abb. 3.21. Eine Tiefensuche, gestartet bei s, in der drei zweifache Zusammenhangskomponenten, drei Trennungspunkte und eine Brücke bestimmt werden. Die Knoten sind mit der DFS-Nummer und der kleinsten erreichbaren DFS-Nummer beschriftet.

Übung 3.34. Zeigen Sie, dass der obige Algorithmus nicht korrekt die zweifachen Zusammenhangskomponenten berechnet, wenn in Zeile 11 lediglich überprüft wird, ob die Kante $\{u,v\}$ auf dem Stack liegt:

13: else **if** ($\{u,v\}$ liegt nicht auf Stack S) **then** {

3.3.4 Transitiver Abschluss

Der *transitive Abschluss eines gerichteten Graphen* $G=(V,E)$ ist die Kantenmenge transitiver Abschluss

$$E^* \subseteq V^2$$

mit

$$(u,v) \in E^* \quad \Leftrightarrow \quad \text{es gibt in } G \text{ einen gerichteten Weg von } u \text{ nach } v.$$

Der transitive Abschluss kann einfach mit Hilfe der folgenden Überlegung berechnet werden. Sei $V=\{u_1,\ldots,u_n\}$ die Knotenmenge von $G=(V,E)$. Jeder einfache Weg von einem Knoten u_i zu einem Knoten u_j mit mehr als zwei Knoten kann in zwei einfache Wege von u_i nach u_k und u_k nach u_j zerlegt werden, wobei u_k der Knoten mit dem größten Index k auf dem Weg zwischen u_i und u_j ist. Dabei ist u_k immer einer der inneren Knoten des Wegs, niemals einer der Knoten u_i oder u_j.

Der folgende Algorithmus berechnet den transitiven Abschluss eines Graphen $G=(V,E)$. Dieser Algorithmus baut eine $(n\times n)$-Matrix A auf, die initial die Adjazenzmatrix für den Graphen G ist. Nach der Ausführung des Algorithmus ist $A[i][j]$ genau dann 1, wenn es in G einen Weg vom Knoten u_i zum Knoten u_j gibt.

1: **for** ($i:=1, i\le n; i$++) {
2: $A[i][i]:=1;$ }
3: **for** ($k:=1; k\le n; k$++) {
4: **for** ($i:=1; i\le n; i$++) {

```
5:        for (j := 1; j ≤ n; j++) {
6:          if ((A[i][k] = 1) und (A[k][j] = 1)) then {
7:            A[i][j] := 1; } } } }
```

Die Korrektheit der obigen Lösung kann mit der folgenden Invariante leicht gezeigt werden. Sie gilt nach jeder Ausführung der beiden inneren **for**-Schleifen.

> *$A[i][j]$ ist genau dann* 1*, wenn es einen Weg von u_i nach u_j gibt, in dem alle inneren Knoten einen Index kleiner oder gleich k haben.*

Diese Invariante gilt offensichtlich auch für $k = 0$ vor dem Ausführen der **for**-Schleife mit der Variablen k in Zeile 3, da A initial die Adjazenzmatrix von G ist und in den Anweisungen 1 und 2 alle Schleifen in A aufgenommen wurden.

Die Laufzeit des obigen Algorithmus ist für Graphen G mit n Knoten in $\Theta(n^3)$ und unabhängig von der Anzahl der Kanten in G. Sie kann in einigen Fällen etwas verbessert werden, indem der erste Teil der Abfrage „**if** $(A[i][k] = 1$ und $A[k][j] = 1)$“ vor die letzte **for**-Schleife gezogen wird. Dies ist möglich, da der Ausdruck $(A[i][k] = 1)$ den Index j nicht enthält. Dann hat der Algorithmus die folgende Gestalt:

```
1:   for (i := 1, i ≤ n; i++) {
2:     A[i][i] := 1; }
3:   for (k := 1; k ≤ n; k++) {
4:     for (i := 1; i ≤ n; i++) {
6:       if (A[i][k] = 1) then {
5:         for (j := 1; j ≤ n; j++) {
6:           if (A[k][j] = 1) then {
7:             A[i][j] := 1; } } } } }
```

Nun wird die innere **for**-Schleife mit dem Index j nicht mehr für jedes (k,i)-Paar aufgerufen, sondern nur dann, wenn $A[i][k] = 1$ ist. Dies kann aber nicht häufiger vorkommen, als es Kanten im transitiven Abschluss von G gibt. Wenn m^* die Anzahl der Kanten im transitiven Abschluss von G ist, dann hat der obige Algorithmus eine Laufzeit aus $\mathcal{O}(n^2 + m^* \cdot n)$.

Für einen stark zusammenhängenden Graphen G enthält der transitive Abschluss alle in G möglichen Kanten. Dies sind zusammen mit den Schleifen genau $|V|^2$ Kanten. Es macht also nur dann einen Sinn, den transitiven Abschluss mit dem obigen Algorithmus zu berechnen, wenn der Graph nicht stark zusammenhängend ist.

Definition 3.35 (verdichteter Graph). *Für einen gerichteten Graphen $G = (V,E)$ mit k starken Zusammenhangskomponenten $G_1 = (V_1,E_1),\ldots,G_k = (V_k,E_k)$ ist der gerichtete Graph $G' = (V',E')$ mit $V' = \{u_1,\ldots,u_k\}$ und*

$$E' = \{(u_i,u_j) \mid (\exists v \in V_i)\ (\exists w \in V_j)\ \ [(v,w) \in E]\}$$

verdichteter Graph

der verdichtete Graph von *G.*

Der transitive Abschluss für einen gerichteten Graphen $G = (V_G, E_G)$ kann nun einfach mit Hilfe des transitiven Abschlusses des verdichteten Graphen G' bestimmt werden. Seien v ein Knoten in G und $\mathrm{sz}(w)$ die Nummer der starken Zusammenhangskomponente, in der sich v befindet. Dann hat der transitive Abschluss von G genau dann eine Kante (v,w), wenn es im transitiven Abschluss des verdichteten Graphen eine Kante $(u_{\mathrm{sz}(v)}, u_{\mathrm{sz}(w)})$ gibt. Hier sei erwähnt, dass der transitive Abschluss für jeden Knoten u immer die Kante (u,u) enthält. Daraus folgt, dass der transitive Abschluss eines Graphen in der Zeit $\mathcal{O}(m^* + (n')^2 + m'^* \cdot n')$ berechnet werden kann, wobei m^* die Anzahl der Kanten im transitiven Abschluss von G, n' die Anzahl der starken Zusammenhangskomponenten von G und m'^* die Anzahl der Kanten im transitiven Abschluss des verdichteten Graphen sind.

3.3.5 Matching-Probleme

Das Bilden einer möglichst großen Auswahl von erlaubten paarweisen Zuordnungen wird in der Informatik als das *Matching-Problem* bezeichnet. Angenommen, eine Gruppe von Personen unternimmt eine mehrtägige Reise und muss zur Übernachtung in Doppelzimmer untergebracht werden. Die Bereitschaft von zwei Personnen, in einem gemeinsamen Zimmer zu übernachten, entspricht hier einer möglichen paarweisen Zuordnung. Für gewöhnlich ist jedoch nicht jede Person bereit, mit jeder anderen Person ein Zimmer zu teilen. Das Matching-Problem ist in diesem Beispiel die Frage nach der Anzahl der notwendigen Zimmer, um alle Personen zu beherbergen. Dabei kann es selbst in der besten Lösung vorkommen, dass nicht jedes Zimmer tatsächlich mit zwei Personen belegt ist.

Matching-Problem

Definition 3.36 (Matching). *Sei $G = (V,E)$ ein ungerichteter Graph. Ein* Matching *ist eine Teilmenge $M \subseteq E$, sodass keine zwei Kanten aus M einen gemeinsamen Knoten haben. Ist jeder Knoten aus V in mindestens einer Kante aus M (und somit in genau einer Kante aus M), dann ist M ein* perfektes Matching. *Die Größe des Matchings M ist die Anzahl der Kanten in M.*

Matching

perfektes Matching

Die Kanten in einem Matching M werden gebundene Kanten *genannt. Die übrigen Kanten, also die Kanten aus der Menge $E - M$, werden* freie Kanten *genannt. Die Endknoten der gebundenen Kanten sind* gebundene Knoten*, alle übrigen Knoten sind* freie Knoten.

gebundene Kante
freie Kante
gebundener Knoten
freier Knoten

Das Matching-Problem ist wie folgt als Graphenproblem definiert.

MAXIMUM MATCHING

MAXIMUM MATCHING	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$.
Ausgabe:	Eine größte Kantenmenge $M \subseteq E$, die keine zwei inzidente Kanten enthält.

Da es ein perfektes Matching nicht immer geben wird, interessieren wir uns für die größtmögliche Zuordnung. Dabei unterscheiden wir zwischen einem größten Matching und einem maximalen Matching (ein nicht vergrößerbares Matching). Ein Matching $M \subseteq E$ ist maximal, wenn es nicht durch Hinzunahme einer weiteren Kante

größtes Matching
maximales Matching

vergrößert werden kann. Ein maximales Matching muss kein größtes Matching sein. Es kann sehr einfach in linearer Zeit konstruiert werden, indem Schritt für Schritt eine Kante $\{u,v\}$ genau dann in M aufgenommen wird, wenn keiner der beiden Endknoten u,v zu einer bereits aufgenommenen Kante gehört.

Beispiel 3.37 (maximales und größtes Matching). In Abb. 3.22 ist zwei Mal ein ungerichteter Graph $G=(V,E)$ mit zehn Knoten abgebildet. Auf der linken Seite ist ein maximales Matching und auf der rechten Seite ist ein größtes Matching für G eingezeichnet. Das maximale Matching mit vier Kanten kann nicht durch Hinzunahme einer weiteren Kante vergrößert werden. Ein größtes Matching hat jedoch fünf Kanten. Dies ist sogar ein perfektes Matching, da jeder Knoten des Graphen G mit einer Matching-Kante inzident ist.

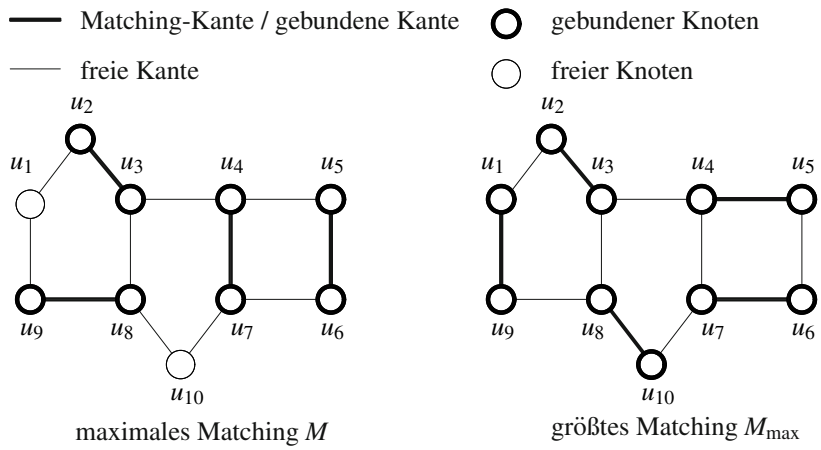

Abb. 3.22. Ein maximales Matching und ein größtes Matching

Um ein möglichst großes Matching zu finden, betrachten wir Wege, die bei einem freien Knoten starten, abwechselnd freie und gebundene Kanten verwenden und bei einem freien Knoten enden.

Definition 3.38 (alternierender Weg). *Es seien $G=(V,E)$ ein ungerichteter Graph und $M \subseteq E$ ein Matching. Ein einfacher Weg*

$$P=(u_1,e_1,u_2,e_2,\ldots,e_{2k-1},u_{2k}),\quad k\geq 1,$$

alternierender Weg

mit einer ungeraden Anzahl von Kanten ist ein alternierender Weg *bezüglich M in G, wenn*

1. *$e_2,e_4,e_6,\ldots,e_{2k-2}$ gebundene Kanten,*
2. *$e_1,e_3,e_5,\ldots,e_{2k-1}$ freie Kanten und*
3. *u_1 und u_{2k} freie Knoten sind.*

In einem alternierenden Weg sind alle inneren Knoten gebundene Knoten, da jede zweite Kante eine gebundene Kante ist. Besitzt ein Graph $G = (V,E)$ einen alternierenden Weg

$$P = (u_1, e_1, u_2, e_2, \ldots, e_{2k-1}, u_{2k}), \quad k \geq 1,$$

bezüglich M, dann gibt es ein weiteres Matching, das größer ist als M. Dies zeigt die folgende Beobachtung:

> Werden die Kanten $e_2, e_4, \ldots, e_{2k-2}$ aus M entfernt und dafür die Kanten e_1, e_3, …, e_{2k-1} hinzugefügt, dann ist die resultierende Kantenmenge ein Matching, das genau eine Kante mehr enthält als M.

Ist das Matching M ein größtes Matching, dann kann es offensichtlich keinen alternierenden Weg bezüglich M geben, da es ja ansonsten ein größeres Matching gäbe. Dass es immer einen alternierenden Weg bezüglich M gibt, wenn M kein größtes Matching ist, zeigt das folgende Lemma.

Lemma 3.39. *Ist M ein Matching für einen ungerichteten Graphen $G = (V,E)$ und ist M kein größtes Matching für G, dann gibt es einen alternierenden Weg bezüglich M in G.*

Beweis. Sei M ein beliebiges Matching für G und $M_{\max}$ ein größtes Matching für G. Sei $k = |M_{\max}| - |M|$ der Größenunterschied der beiden Zuordnungen. Betrachte nun die Menge aller Kanten aus $M_{\max}$, die nicht in M sind, zusammen mit der Menge aller Kanten aus M, die nicht in $M_{\max}$ sind. Diese Menge

$$M_{\text{sym}} \;=\; (M_{\max} - M) \cup (M - M_{\max})$$

ist die *symmetrische Differenz* der beiden Mengen $M_{\max}$ und M. symmetrische Differenz

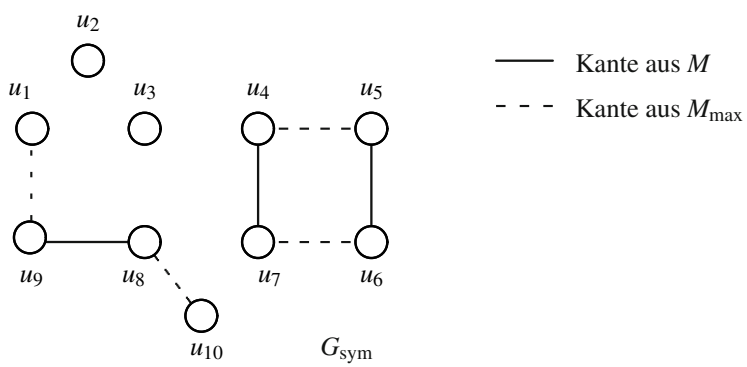

Abb. 3.23. Der Graph $G_{\text{sym}} = (V, M_{\text{sym}})$, gebildet aus dem Matching M und dem größten Matching $M_{\max}$ aus Abb. 3.22

Jeder Knoten des Graphen G ist mit höchstens zwei Kanten aus M_{sym} inzident, da er mit höchstens einer Kante aus $M_{\max}$ und mit höchstens einer Kante aus M inzident ist. Der Graph $G_{\text{sym}} = (V, M_{\text{sym}})$ mit Knotenmenge V und Kantenmenge M_{sym} besteht also aus knotendisjunkten Wegen bzw. Kreisen, siehe Abb. 3.23. Jeder Weg und jeder Kreis verwendet abwechselnd Kanten aus $M_{\max}$ und M. Jeder Kreis, falls einer existiert, hat somit genau so viele Kanten aus $M_{\max}$ wie aus M. Da $M_{\max}$ jedoch k Kanten mehr enthält als M, muss es mindestens k Wege in G_{sym} geben, die eine Kante mehr aus $M_{\max}$ als aus M enthalten. Diese Wege in G_{sym} sind offensichtlich alternierende Wege bezüglich M in G, da alle Kanten aus $M_{\max}$, die sich in M_{sym} befinden, nicht in M sind. ❑

Ein größtes Matching kann also durch wiederholtes Finden alternierender Wege aus einem beliebigen Matching konstruiert werden. Einen alternierenden Weg zu finden ist jedoch nicht ganz so einfach. Betrachte zum Beispiel den Graphen G aus Abb. 3.24 mit dem eingezeichneten Matching $\{\{u_2,u_3\},\{u_4,u_9\},\{u_5,u_6\},\{u_8,u_{10}\}\}$. Eine Tiefensuche, die an dem freien Knoten u_1 beginnt und abwechselnd freie und gebundene Kanten verwendet, könnte sich über u_2, u_3, u_4, u_9 und u_{10} bis zum Knoten u_8 fortsetzen. Da der Knoten u_3 bereits besucht ist, bricht die Suche hier ab. Der Knoten u_5 wird nicht besucht, weil abwechselnd gebundene und freie Kanten verwendet werden müssen. Die Tiefensuche hat also keinen alternierenden Weg gefunden. Hätte die Suche bei dem Knoten u_3 jedoch zuerst den Nachfolger u_8 gewählt anstatt u_4, dann hätte sie den alternierenden Weg $(u_1,u_2,u_3,u_8,u_{10},u_9,u_4,u_5,u_6,u_7)$ gefunden.

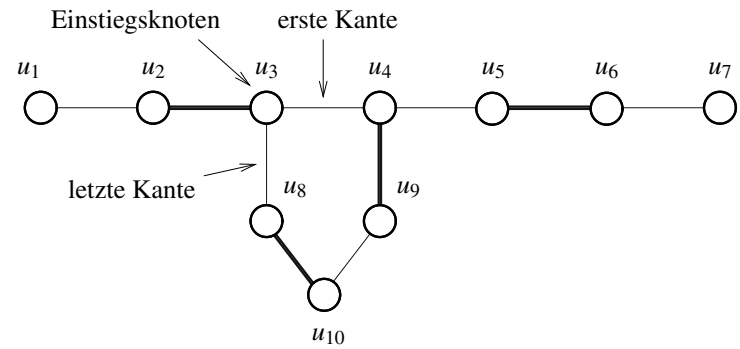

Abb. 3.24. Eine Suche nach einem alternierenden Weg

Dieses Problem tritt ausschließlich dann auf, wenn die Tiefensuche einen Kreis ungerader Länge durchläuft und dabei zu einem bereits besuchten Knoten zurückkommt. Die erste und letzte Kante in dem Kreis ist hier immer eine freie Kante. Der gemeinsame Knoten dieser beiden freien Kanten ist der Einstiegsknoten. Einen solchen gefundenen Kreis nennen wir ein *Blüte*. Das Problem mit den Blüten kann beseitigt werden, indem die Knoten der Blüte B zu einem Knoten u_B zusammengefasst werden, sobald ein solcher Kreis als Blüte identifiziert wurde. Alle Kanten, die

Blüte

mit einem Knoten aus der Blüte inzident waren, sind anschließend mit dem Knoten u_B inzident. Wird mit dem Zusammenfassen von Blüten ein alternierender Weg gefunden, dann besitzt der ursprüngliche Graph G offensichtlich ebenfalls einen alternierenden Weg. Dieser kann einfach durch das Expandieren der Blüten beginnend am Einstiegsknoten konstruiert werden.

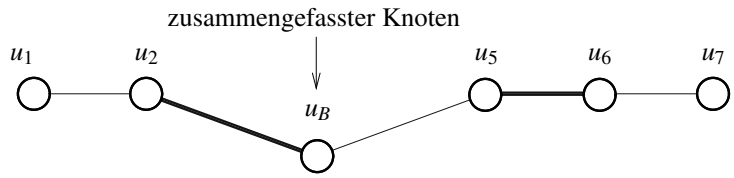

Abb. 3.25. Eine zusammengefasste Blüte

Dass dadurch immer alternierende Wege gefunden werden, wenn sie existieren, ist nicht so offensichtlich. Um dies zu erkennen, betrachten wir den folgenden Algorithmus, der an den freien Knoten eine Suche startet und gefundene Blüten zusammenfasst. Um einen alternierenden Weg schließlich auszugeben, speichern wir an jedem besuchten Knoten u den Vorgänger $p[u]$, über den der Knoten erstmalig in der Suche erreicht wurde. Der freie Knoten, bei dem die Suche gestartet wurde, bezeichnen wir mit $r[u]$. Der nächste gemeinsame Vorgänger von zwei Knoten u und v sei $p[u,v]$, falls es einen solchen Vorgänger gibt. Ansonsten sei $p[u,v] = -1$. Um den nachfolgenden Algorithmus übersichtlich zu gestalten, verzichten wir auf die explizite Berechnung der Werte $p[u,v]$ und $r[u]$.

In einem Zustand $z[u]$ speichern wir, ob der Knoten u über eine gerade Anzahl von Kanten ($z[u] = 0$) oder über eine ungerade Anzahl von Kanten ($z[u] = 1$) von einem freien Knoten erreicht wurde. Der Wert ($z[u] = -1$) kennzeichnet die noch nicht besuchten Knoten. Ist $\{u,v\}$ eine gebundene Kante, also eine Kante aus dem Matching M, dann seien $m[u] = v$ und $m[v] = u$. Die Suche verwendet eine Datenstruktur D zur Verwaltung der noch zu betrachtenden Kanten, siehe Abschnitt 3.3.2.

```
 1: for all u ∈ V {
 2:   if (u ist frei bezüglich M) {
 3:     z[u] := 0;  p[u] := u;
 4:     for all (u,v) ∈ E {
 5:       insert(D,(u,v)); } }
 6:   else {
 7:     z[v] := −1; } }
 8: while (D ≠ ∅) {
 9:   (u,v) := extract(D);
10:   if (z[u] = 0) and (z[v] = −1) {
11:     z[v] := 1;  p[v] := u;
12:     z[m[v]] := 0;  p[m[v]] := v;
13:     for all (m[v],w) ∈ E {
```

```
14:          insert(D,(m[v],w)); } }
15:    if (z[u] = 0) and (z(v) = 0) {
16:       if ((w := p[u,v])! = −1){
17:          fasse die Knoten u, p[u],...,w,...,p[v],v zu w_B zusammen; }
18:       else {
19:          r[u],...,p[u],u,v,p[u]...,r[v] ist ein alternierender Weg;
20:          halt; } } }
```

In Zeile 11 wird ein Weg von einem Startknoten zum Knoten u über eine freie Kante $\{u,v\}$ und eine gebundene Kante $\{v,m[v]\}$ nach $m[v]$ verlängert. Der Knoten v ist ein gebundener Knoten, da alle freien Knoten in Zeile 3 den Zustand 0 erhalten haben. Da die Zustände der Endknoten einer gebundenen Kante immer gleichzeitig vergeben werden, muss der andere Knoten $m[v]$ ebenfalls noch unbesucht sein. In Zeile 15 wurde eine freie Kante gefunden, die zwei Wege verbindet. Haben die beiden Wege einen gemeinsamen Vorgänger $p[u,v]$, so bilden die Knoten auf den beiden Wegen von $p[u,v]$ nach u und von $p[u,v]$ nach v eine Blüte. Haben die beiden Wege keinen gemeinsamen Vorgänger, dann bilden sie zusammen mit der verbindenden Kante $\{u,v\}$ einen alternierenden Weg. Dieser Weg muss eventuell noch nachbearbeitet werden, indem zusammengefasste Blüten wieder expandiert werden.

Edmonds Algorithmus

Der obige Algorithmus ist unter dem Namen *Edmonds Algorithmus* bekannt, siehe [Edm65]. Es gibt Implementierungen für diesen Algorithmus, die einen zunehmenden Weg in linearer Zeit finden können, siehe hierzu auch [GT83]. Da ein Matching höchstens $|V|/2$ Kanten enthalten kann, ist es daher möglich, ein größtes Matching in der Zeit $\mathcal{O}(|V| \cdot |E|)$ zu konstruieren. Die schnellsten Algorithmen für die Berechnung größter Zuordnungen haben sogar eine Laufzeit von $\mathcal{O}(\sqrt{|V|} \cdot |E|)$, siehe zum Beispiel [MV80].

Übung 3.40. Entwerfen Sie einen Algorithmus, der in linearer Zeit einen alternierenden Weg bezüglich eines Matchings M findet, wenn der Graph keine Kreise ungerader Länge enthält.

3.4 Ausgewählte Probleme auf Graphen

In den folgenden Abschnitten werden einige klassische Entscheidungsprobleme für ungerichtete Graphen definiert.

3.4.1 Unabhängige Mengen, Cliquen und Knotenüberdeckungen

UNABHÄNGIGE MENGE

UNABHÄNGIGE MENGE	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $k \leq \|V\|$.
Frage:	Gibt es in G eine unabhängige Menge $V' \subseteq V$ der Größe $\|V'\| \geq k$?

Das Problem UNABHÄNGIGE MENGE wird im Englischen als INDEPENDENT SET bezeichnet. Die Größe einer größten unabhängigen Menge in einem Graphen G ist die *Stabilitäts-* oder *Unabhängigkeitszahl von G*. Sie wird mit $\alpha(G)$ bezeichnet.

INDEPENDENT SET

Stabilitäts- bzw. Unabhängigkeitszahl $\alpha(G)$

CLIQUE	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $k \leq \|V\|$.
Frage:	Gibt es in G eine Clique $V' \subseteq V$ der Größe $\|V'\| \geq k$?

CLIQUE

Die Größe einer größten Clique in einem Graphen G ist die *Cliquenzahl von G*. Sie wird mit $\omega(G)$ bezeichnet.

Cliquenzahl $\omega(G)$

Offensichtlich ist eine unabhängige Menge in einem Graphen $G = (V,E)$ eine Clique im Komplementgraphen $\overline{G}$ und umgekehrt. Somit gilt:

$$\omega(G) = \alpha(\overline{G}). \tag{3.3}$$

KNOTENÜBERDECKUNG

KNOTENÜBERDECKUNG	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $k \leq \|V\|$.
Frage:	Gibt es in G eine Knotenüberdeckung $V' \subseteq V$ der Größe $\|V'\| \leq k$?

Gebräuchlich ist auch der englische Name VERTEX COVER für das Problem KNOTENÜBERDECKUNG. Die Größe einer kleinsten Knotenüberdeckung in einem Graphen G ist die *Knotenüberdeckungszahl von G*. Sie wird mit $\tau(G)$ bezeichnet.

VERTEX COVER

Knotenüberdeckungszahl $\tau(G)$

Es gilt die folgende Beziehung zwischen unabhängigen Mengen, Cliquen und Knotenüberdeckungen in ungerichteten Graphen.

Lemma 3.41. *Für jeden Graphen $G = (V,E)$ und alle Teilmengen $U \subseteq V$ sind die folgenden drei Aussagen äquivalent:*

1. *U ist eine unabhängige Menge von G.*
2. *U ist eine Clique im Komplementgraphen $\overline{G}$.*
3. *$V - U$ ist eine Knotenüberdeckung von G.*

Übung 3.42. Beweisen Sie Lemma 3.41.

Die nicht ganz triviale Äquivalenz der ersten und der dritten Aussage in Lemma 3.41 wurde schon 1959 von Gallai [Gal59] gezeigt und impliziert für einen Graphen $G = (V,E)$ die folgende Beziehung zwischen $\tau(G)$ und $\alpha(G)$:

$$\tau(G) + \alpha(G) = |V|. \tag{3.4}$$

Mit der algorithmischen Komplexität der Probleme UNABHÄNGIGE MENGE, CLIQUE und KNOTENÜBERDECKUNG befassen wir uns in Kapitel 5.

3.4.2 Partition in unabhängige Mengen und Cliquen

Partition

Eine *Partition* einer Menge M ist eine Aufteilung der Elemente aus M in nicht leere, disjunkte Mengen, deren Vereinigung die Menge M ergibt.

Beispiel 3.43 (Partition). Eine Partition der Menge

$$M = \{m_1, m_2, m_3, m_4, m_5, m_6\}$$

ist zum Beispiel gegeben durch

$$\{m_1, m_2\},\ \{m_4\},\ \{m_3, m_5\},\ \{m_6\}.$$

Keine Partitionen von M sind hingegen die Aufteilungen

$$\{m_1, m_2, m_3\},\ \{m_4\},\ \{m_3, m_5\},\ \{m_6\} \tag{3.5}$$

und

$$\{m_1, m_2, m_4\},\ \{m_3\},\ \{m_6\}. \tag{3.6}$$

In der ersten Aufteilung (3.5) ist das Element m_3 in mehr als einer Menge, in der zweiten Aufteilung (3.6) fehlt das Element m_5.

Bei Partitionsproblemen für Graphen sucht man nach einer Aufteilung der Knoten oder Kanten in Teilmengen, die eine bestimmte Bedingung erfüllen, beispielsweise die Bedingung, dass alle Teilmengen unabhängig sind.

PARTITION IN UNABHÄNGIGE MENGEN

	PARTITION IN UNABHÄNGIGE MENGEN
Gegeben:	Ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq \lvert V \rvert$.
Frage:	Gibt es eine Partition von V in k unabhängige Mengen?

Färbungszahl $\chi(G)$
chromatische Zahl

Der Begriff der k-Färbbarkeit wurde im Anschluss an Definition 3.17 durch eine Partitionierung der Knotenmenge eines Graphen in k unabhängige Mengen charakterisiert. Dementsprechend bezeichnet man die minimale Anzahl von unabhängigen Mengen, die zur Überdeckung eines Graphen G notwendig sind, als die *Färbungszahl von G*, kurz $\chi(G)$. Synonym nennt man die Färbungszahl eines Graphen auch seine *chromatische Zahl*. Das Problem PARTITION IN UNABHÄNGIGE MENGEN ist daher auch unter dem Namen FÄRBBARKEIT bekannt. Ist der Parameter k nicht Teil der Probleminstanz, so nennt man das entsprechende Problem k-FÄRBBARKEIT.

k-FÄRBBARKEIT

Beispiel 3.44 (Partition in unabhängige Mengen). In Abb. 3.26 ist ein Graph gezeigt, dessen Knotenmenge in drei unabhängige Mengen aufgeteilt ist. Dieser Graph ist somit dreifärbbar. Eine Partition des Graphen in zwei unabhängige Mengen ist offenbar nicht möglich. Daher ist seine Färbungszahl drei.

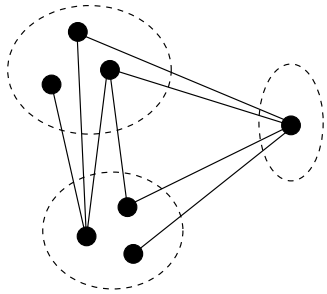

Abb. 3.26. Partition in drei unabhängige Mengen

Da die Färbungszahl eines vollständigen Graphen K_n genau n beträgt, bildet $\omega(G)$ stets eine untere Schranke für den Wert von $\chi(G)$. Für jeden Graphen G gilt also:

$$\omega(G) \leq \chi(G). \tag{3.7}$$

Hier stellt sich die natürliche Frage, für welche Graphen die Gleichheit gilt. Dazu verweisen wir auf Satz 3.53. Graphen mit Färbungszahl höchstens eins oder zwei lassen sich einfach wie folgt charakterisieren.

Lemma 3.45. *1. Ein Graph G hat genau dann die Färbungszahl eins, wenn G keine Kante besitzt.*
2. Ein Graph G hat genau dann die Färbungszahl ≤ 2, wenn G bipartit ist.

Analog zum eben definierten Problem betrachtet man auch das folgende:

PARTITION IN CLIQUEN

PARTITION IN CLIQUEN	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $s \leq \lvert V \rvert$.
Frage:	Gibt es eine Partition von V in s Cliquen?

Die *Cliquenüberdeckungszahl eines Graphen G* ist die minimale Anzahl von Cliquen, die zur Überdeckung von G notwendig sind. Sie wird mit $\theta(G)$ bezeichnet.

Cliquenüberdeckungszahl $\theta(G)$

Beispiel 3.46 (Partition in Cliquen). In Abb. 3.27 ist ein Graph G gezeigt, dessen Knotenmenge in drei Cliquen aufgeteilt ist. Eine Partition des Graphen in zwei Cliquen ist offenbar nicht möglich. Daher ist $\theta(G) = 3$.

Da in einer Überdeckung eines Graphen G mit Cliquen jede Clique nur höchstens einen Knoten aus einer unabhängigen Menge in G enthalten kann, bildet $\alpha(G)$ stets eine untere Schranke für $\theta(G)$. Für jeden Graphen G gilt also:

$$\alpha(G) \leq \theta(G). \tag{3.8}$$

Offensichtlich ist jede Partition der Knotenmenge eines ungerichteten Graphen G in unabhängige Mengen eine Partition der Knotenmenge des Komplementgraphen $\overline{G}$ in Cliquen, siehe Lemma 3.41. Damit gilt für jeden Graphen G die Beziehung:

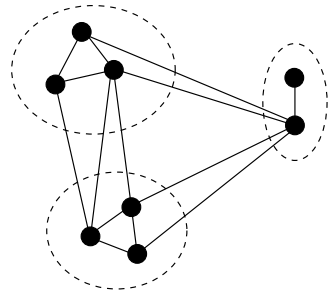

Abb. 3.27. Partition in drei Cliquen

$$\chi(G) = \theta(\overline{G}). \tag{3.9}$$

unabhängige Kantenmenge

Als Nächstes betrachten wir Partitionierungen von Kantenmengen. Sei $G = (V,E)$ ein ungerichteter Graph. Eine Kantenmenge $E' \subseteq E$ heißt *unabhängige Kantenmenge von G*, falls gilt:

$$(\forall e_1, e_2 \in U)\ [e_1 \neq e_2 \ \Rightarrow\ e_1 \cap e_2 = \emptyset].$$

PARTITION IN UNABHÄNGIGE KANTENMENGEN

PARTITION IN UNABHÄNGIGE KANTENMENGEN	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $k \leq \lvert V \rvert$.
Frage:	Gibt es eine Partition von E in k unabhängige Kantenmengen?

Kantenfärbungszahl $\chi'(G)$ chromatischer Index KANTENFÄRBBARKEIT

Die kleinste Anzahl von disjunkten unabhängigen Kantenmengen eines Graphen G nennt man die *Kantenfärbungszahl* von G oder auch den *chromatischen Index* von G, kurz $\chi'(G)$. Das Problem PARTITION IN UNABHÄNGIGE KANTENMENGEN ist auch unter dem Namen KANTENFÄRBBARKEIT bekannt. Die Kantenfärbungszahl eines ungerichteten Graphen G kann wie folgt mit Hilfe des maximalen Knotengrads $\Delta(G)$ abgeschätzt werden.

Satz 3.47 (Vizing [Viz64] und Gupta [Gup66]). *Für jeden Graphen G gilt:*

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1.$$

ohne Beweis

Da man für einen gegebenen Graphen G den maximalen Knotengrad sehr leicht bestimmen kann, ist die effiziente Berechnung der Kantenfärbungszahl offensichtlich bis auf einen additiven Fehler von eins leicht möglich. Das Finden des exakten Wertes für $\chi'(G)$ ist jedoch selbst für kubische Graphen sehr schwer.

Kantengraph

Der *Kantengraph* (englisch: *line graph*) eines ungerichteten Graphen G, kurz mit $L(G)$ bezeichnet, hat einen Knoten für jede Kante in G. Zwei Knoten sind genau dann adjazent in $L(G)$, wenn die entsprechenden Kanten in G inzident sind, siehe auch [Whi32].

Beispiel 3.48 (Kantengraph). Abbildung 3.28 zeigt einen Graphen G mit sechs Kanten e_i, $1 \leq i \leq 6$. Der zugehörige Kantengraph $L(G)$ hat sechs Knoten v_i, $1 \leq i \leq 6$.

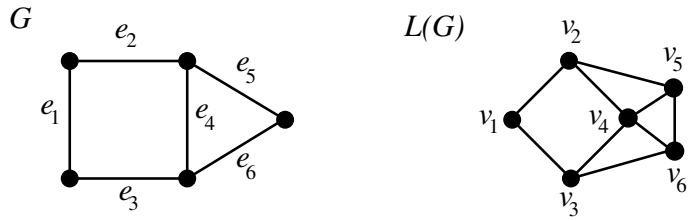

Abb. 3.28. Ein Graph G und der zugehörige Kantengraph $L(G)$

Der Kantengraph beschreibt die folgende einfache Beziehung zwischen den Parametern χ' und χ.

Satz 3.49. *Für jeden Graphen G gilt:*

$$\chi'(G) = \chi(L(G)).$$

ohne Beweis

Die vier Graphparameter α, ω, χ und θ spielen eine wichtige Rolle bei der Untersuchung *perfekter* Graphen.

Definition 3.50. *Ein Graph G ist* perfekt, *falls für jeden induzierten Teilgraphen H von G die Bedingung* perfekt

$$\omega(H) = \chi(H)$$

erfüllt ist.

Beispiel 3.51 (perfekter Graph).

1. Für Graphen $G = (V,E)$ ohne Kanten ist
$$\omega(W) = \chi(W) = 1.$$
2. Für Graphen $G = (V,E)$ mit Kanten zwischen je zwei verschiedenen Knoten ist
$$\omega(G) = \chi(G) = |V|.$$
3. Für Wälder und bipartite Graphen G mit mindestens einer Kante ist
$$\omega(G) = \chi(G) = 2.$$

Da total unzusammenhängende Graphen, Cliquen, Wälder und bipartite Graphen abgeschlossen bezüglich der Bildung induzierter Teilgraphen sind, sind sie perfekt.

Übung 3.52. Zeigen Sie, dass nicht jeder 3-färbbare Graph perfekt ist.

Die wichtigsten Charakterisierungen für perfekte Graphen fassen wir in dem folgenden Satz zusammen. Die Äquivalenz der ersten und letzten Aussage war lange Zeit unter der Bezeichung „Strong Perfect Graph Conjecture“ bekannt. Sie wurde jedoch vor wenigen Jahren bewiesen.

Satz 3.53 (Chudnovsky, Robertson, Seymour und Thomas [CRST06]). *Für jeden ungerichteten Graphen G sind die folgenden Eigenschaften äquivalent:*

1. *G ist perfekt.*
2. *Für jeden induzierten Teilgraphen H von G gilt: $\alpha(H) = \theta(H)$.*
3. *Für kein $n \geq 2$ enthält G den ungerichteten Kreis C_{2n+1} mit $2n+1$ Knoten oder dessen Komplementgraphen $\overline{C_{2n+1}}$ als induzierten Teilgraphen.*

ohne Beweis

Weitere Beispiele für perfekte Graphen sind Co-Graphen, distanzerhaltende Graphen und chordale Graphen, die wir später in den Kapiteln 6, 9 und 11 näher kennen lernen werden.

3.4.3 Dominierende Mengen und domatische Zahl

In Definition 3.14 wurde der Begriff der dominierenden Menge eingeführt. Sehen wir uns zur Erinnerung ein Beispiel an.

Beispiel 3.54 (dominierende Menge). Abbildung 3.29 zeigt den Sitzplan der Klasse 8c. Jeder Knoten entspricht einem Schüler, wobei die sehr guten Schüler durch volle Kreise dargestellt sind. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn die entsprechenden Schüler so dicht beieinander sitzen, dass sie voneinander abschreiben können.

Dass die Klassenarbeit von letzter Woche so ausgesprochen gut ausgefallen war (jeder einzelne Schüler hatte diesmal eine glatte Eins geschrieben!), machte den Mathematiklehrer der 8c sehr glücklich und stolz, dann aber misstrauisch. Er sah sich den Sitzplan etwas genauer an, und tatsächlich: Die acht wirklich sehr guten Schüler hatten sich so platziert, dass sie eine dominierende Menge der Klasse bildeten.

DOMINIERENDE MENGE

DOMINIERENDE MENGE	
Gegeben:	Ein ungerichteter Graph $G = (V,E)$ und eine natürliche Zahl $k \leq \lvert V \rvert$.
Frage:	Gibt es in G eine dominierende Menge der Größe höchstens k?

Im Sinne von Beispiel 3.54 fragt dieses Problem also für einen gegebenen Graphen und eine Zahl k, ob sich nicht mehr als k sehr gute Schüler so auf die Knoten des Graphen verteilen können, dass jeder nicht so gute Schüler einen sehr guten Nachbarn hat, von dem er abschreiben kann. Für den Graphen G in Abb. 3.29 ist zum Beispiel $(G,8)$ eine „Ja“-Instanz des Problems DOMINIERENDE MENGE.

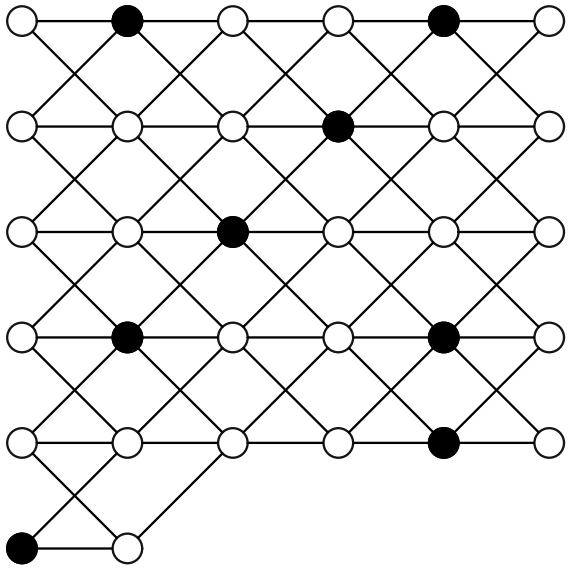

Abb. 3.29. Sitzplan der Klasse 8c: eine dominierende Menge für Beispiel 3.54

Definition 3.55. *Die Größe einer kleinsten dominierenden Menge eines Graphen G heißt* Dominierungszahl von G, *kurz mit $\gamma(G)$ bezeichnet. Die* domatische Zahl von G, *kurz $d(G)$, ist die Größe einer größten domatischen Partition von G, d. h. die größte Zahl k, sodass G in k dominierende Mengen partitioniert werden kann.*

Dominierungszahl $\gamma(G)$
domatische Zahl $d(G)$

Dominierende Mengen spielen nicht nur in Klassenräumen (wie in Beispiel 3.54), sondern auch in vielen praktischen Anwendungsbereichen eine wichtige Rolle, so etwa im Zusammenhang mit Computer- oder Kommunikationsnetzwerken. Eine dominierende Menge in einem Kommunikationsnetzwerk, die wir eine *Sendergruppe* nennen wollen, ist eine Teilmenge der Knoten, die jeden anderen Knoten in einem Schritt erreicht (so ähnlich, wie die nicht so guten Schüler in Beispiel 3.54 „Sendungen“ von den sehr guten Schülern erhalten konnten, sofern sie in deren unmittelbarer Nachbarschaft saßen).

Nun kann es sein, dass eine Sendergruppe ganz oder teilweise ausfällt, sodass nicht mehr alle Knoten des Netzwerks Sendungen empfangen können. Es wäre dann sinnvoll, eine alternative Sendergruppe im Netzwerk zu installieren, die zweckmäßigerweise zu der anderen disjunkt sein sollte. Fällt auch diese aus, möchte man sie vielleicht durch eine dritte, zu den beiden ersten disjunkte Sendergruppe ersetzen können, jedenfalls wenn es wichtig ist, dass wirklich alle Knoten des Netzwerks erreicht werden. Die domatische Zahl gibt die maximale Anzahl disjunkter Sendergruppen an, die das gegebene Netzwerk in alternative dominierende Mengen partitionieren.

DOMATISCHE ZAHL	
Gegeben:	Ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq \|V\|$.
Frage:	Ist die domatische Zahl von G mindestens k?

DOMATISCHE ZAHL

Beispiel 3.56 (domatische Zahl). Abbildung 3.30 zeigt einen Graphen G mit domatischer Zahl $d(G) = 3$. Eine der drei dominierenden Mengen enthält nur den grauen, eine andere nur den schwarzen Knoten, und die dritte dominierende Menge besteht aus den (übrigen drei) weißen Knoten. Offenbar ist es nicht möglich, G in mehr als drei disjunkte dominierende Mengen zu zerlegen.

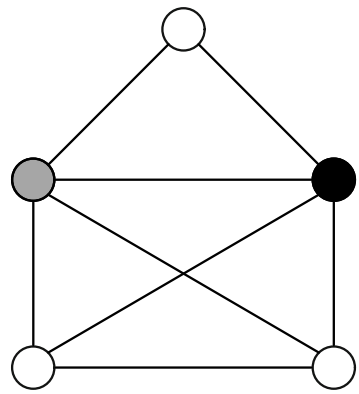

Abb. 3.30. Ein Graph mit domatischer Zahl drei für Beispiel 3.56

Die Entscheidungsprobleme DOMINIERENDE MENGE und DOMATISCHE ZAHL sind ebenfalls schwere Probleme, siehe Kapitel 5.

Übung 3.57. Angenommen, es ist ein verregneter Sonntagnachmittag, alle Ihre Freunde sind verreist, es ist langweilig und gibt nichts zu tun. Dann haben Sie vielleicht Lust, für den Graphen in Abb. 3.29 die Dominierungszahl und die domatische Zahl zu bestimmen.

3.4.4 Das Problem des Handelsreisenden

Das Problem des Handelsreisenden ist eines der klassischen kombinatorischen Optimierungsprobleme. Wir verwenden auch die englische Bezeichnung TRAVELING SALESPERSON PROBLEM (oder kurz TSP) für dieses Problem, da diese auch im Deutschen gebräuchlich ist. Die Aufgabe besteht darin, eine Reihenfolge für den Besuch mehrerer Orte so zu bestimmen, dass die gesamte Reisestrecke nach der Rückkehr zum Ausgangsort möglichst kurz ist. Die Länge einer Rundreise, die also jeden

der gegebenen Orte genau einmal besucht und dann zum Ausgangsort zurückkehrt, messen wir in der Summe der vorkommenden Entfernungen (die wir auch Kantenkosten oder -gewichte nennen). Als Entscheidungsproblem lässt sich das Problem des Handelsreisenden wie folgt formulieren:

TRAVELING SALESPERSON PROBLEM (TSP)

TRAVELING SALESPERSON PROBLEM (TSP)	
Gegeben:	Ein vollständiger Graph $K_n = (V,E)$, eine Kostenfunktion $c : E \to \mathbb{N}$ und eine Zahl $k \in \mathbb{N}$.
Frage:	Gibt es in K_n eine Rundreise der Länge höchstens k?

Beispiel 3.58 (TSP). Stellen wir uns vor, ein Händler muss fünf Städte in Deutschland bereisen. Da er so schnell wie möglich damit fertig sein will, interessiert er sich für einen kürzesten Weg zwischen den Städten. Abbildung 3.31 zeigt den Graphen, auf dem der Händler seine Rundreise unternimmt. Die Zahlen an den Kanten des Graphen geben jeweils Vielfache von 100 km an und stellen die Kostenfunktion dar.

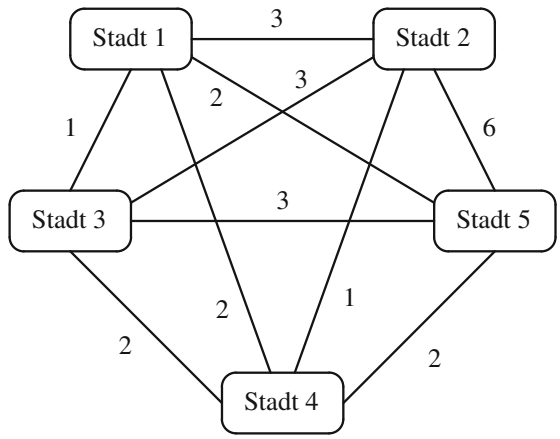

Abb. 3.31. Ein Graph für die Rundreise eines Händlers

Die entsprechende Kosten- oder Entfernungsmatrix sieht so aus:

$$D = \begin{pmatrix} 0 & 3 & 1 & 2 & 2 \\ 3 & 0 & 3 & 1 & 6 \\ 1 & 3 & 0 & 2 & 3 \\ 2 & 1 & 2 & 0 & 2 \\ 2 & 6 & 3 & 2 & 0 \end{pmatrix}.$$

Ein Eintrag $d(i,j)$ in dieser Matrix gibt die Entfernung zwischen den Städten i und j an. In unserem Beispiel handelt es sich um eine symmetrische Entfernungsmatrix

(d. h., $d(i,j) = d(j,i)$ für alle i und j, $1 \leq i,j \leq 5$). Im Allgemeinen müssen Matrizen zur Darstellung der Kostenfunktion natürlich nicht symmetrisch sein. Misst man etwa den Benzinverbrauch zwischen zwei Städten und liegt die Stadt i im Tal und die Stadt j auf dem Berg, so wird der Spritverbrauch von i nach j höher sein als in der umgekehrten Richtung; $d(i,j) > d(j,i)$ ist dann eine realistische Annahme. In diesem Fall müsste man TSP auf gerichteten Graphen definieren. Wir werden auf das Beispiel des TSP in Abb. 3.31 später noch zurückkommen (siehe Abschnitt 8.1).

3.5 Ausgewählte Algorithmenentwurfstechniken

3.5.1 Backtracking

Lisa ist gerade sieben Jahre alt geworden und feiert ihren Kindergeburtstag. Sie hat verbundene Augen und einen Kochlöffel in der Hand, mit dem sie den Boden vor sich erkundet, während sie durchs Wohnzimmer kriecht. Eine Schar aufgeregter Kinder feuert sie beim „Topfschlagen“ an („Wärmer, wärmer!“), während Lisa dem Topf, unter dem die Süßigkeiten liegen, immer näher kommt. Sie verpasst ihn jedoch knapp und hört plötzlich: „Kalt, immer kälter!“ Sie hält inne und krabbelt zurück, bis es wieder „wärmer“ wird, verfehlt den Topf jetzt auf der anderen Seite, wird wieder von den anderen Kindern zur Umkehr bewegt und schlägt schließlich laut auf den Topf. Lisa hat so nicht nur viele leckere Süßigkeiten gefunden, sondern auch das Prinzip des *Backtracking* entdeckt.

Backtracking

Backtracking, das man auf deutsch die „Rücksetzmethode“ nennen könnte (wenn sich nicht der englische Begriff auch bei uns durchgesetzt hätte), ist ein algorithmisches Verfahren, das Probleme nach dem Prinzip „Versuch und Irrtum“ zu lösen versucht. Nehmen wir an, dass wir ein Problem lösen wollen, dessen Lösungsraum eine Baumstruktur hat. Die Wurzel enthält die „leere Lösung“, jedes Kind eines inneren Knotens im Baum erweitert die (potenzielle) Teillösung seines Vorgängerknotens, und die Blätter enthalten Kandidaten für vollständige Lösungen (d. h., einige Blätter stellen Lösungen der gegebenen Probleminstanz dar, andere nicht). Ausgehend von der Wurzel durchläuft man den Baum mittels Tiefensuche, erweitert also eine anfangs leere Lösung Schritt für Schritt zu einer immer größeren (potenziellen) Teillösung in der Hoffnung, schließlich zu einem Lösungsblatt zu gelangen. Im Unterschied zur Tiefensuche durchläuft man jedoch nicht den gesamten Baum, sondern versucht, „hoffnungslose“ Teilbäume abzuschneiden, um Zeit zu sparen. Das heißt, wenn man einen Knoten erreicht, dessen potenzielle Teillösung unmöglich zu einer Lösung der Probleminstanz erweitert werden kann,[11] dann kann man sich das Durchsuchen des gesamten darunter befindlichen Teilbaums ersparen. Man setzt also einen Schritt zurück und versucht, den Vorgänger dieser hoffnungslosen potenziellen Teillösung zu einer vollständigen Lösung auszubauen. Wie die Tiefensuche lässt sich dieses Verfahren am besten rekursiv realisieren. In diesem Buch werden uns einige Beispiele für Anwendungen von Backtracking begegnen (siehe Kapitel 7).

[11] Das ist der Punkt, an dem man „Kalt, immer kälter!“ hört.

3.5.2 Teile und herrsche

Hier haben wir es wieder mit einer rekursiven Herangehensweise zu tun. Eine Probleminstanz, die nur wegen ihrer Größe sehr schwierig ist, wird in kleinere Probleminstanzen derselben Art aufgeteilt. Sind diese kleineren Instanzen immer noch schwer zu lösen, so unterteilt man sie (rekursiv) wieder, bis schließlich die entstehenden kleinen Probleminstanzen einfach zu lösen sind. Entweder erhält man so im letzten Rekursionsschritt unmittelbar eine Lösung der großen Ausgangsinstanz des Problems oder aber man setzt die Lösungen der kleineren Probleminstanzen zu einer Lösung der Ausgangsinstanz zusammen (wieder Schritt für Schritt, indem man von einer Rekursionsstufe zur nächsthöheren auftaucht). Ein typisches Anwendungsbeispiel für die Teile-und-herrsche-Strategie ist der Sortieralgorithmus *Quicksort*, ein anderes findet man zum Beispiel im Beweis von Satz 5.48. Man beachte, dass die Laufzeit solcher Algorithmen stark von der Teilung der Probleme selbst abhängt.

Teile-und-herrsche-Strategie

3.5.3 Dynamische Programmierung

Dynamische Programmierung ist eine Technik, mit der man beispielsweise schwere Optimierungsprobleme lösen kann. Sie wurde in den 1940er Jahren von Richard Bellman, der ihr auch diesen Namen gab, für Probleme der Regelungstheorie eingesetzt; das Prinzip selbst war aber schon vorher (unter anderem Namen) in der Physik gebräuchlich.

Das gegebene Problem wird wieder in Teilprobleme aufgeteilt, die man entweder direkt löst oder wiederum in mehrere Teilprobleme zerlegt. Die Lösungen der kleinsten Teilprobleme setzt man dann zusammen zu Lösungen der nächstgrößeren Teilprobleme und so weiter, bis man schließlich eine (optimale) Lösung des Ausgangsproblems erhält. Der Witz des Verfahrens besteht darin, dass alle bereits gefundenen Lösungen aufgehoben (also gespeichert) und für die Lösungen der größeren Teilprobleme verwendet werden, sodass man sie nicht jedes Mal neu berechnen muss. Auf diese Art kann man beträchtlich Zeit sparen.

Das Kernstück eines Algorithmus, der auf dem Prinzip der dynamischen Programmierung beruht, ist immer eine (rekursive) Formel, die angibt, wie man die Lösungen der kleinen Teilprobleme zu einer Lösung eines nächstgrößeren Teilproblems zusammensetzen kann.

Ein Beispiel, das uns auch später noch beschäftigen wird, ist der Algorithmus von Lawler [Law76], der mittels dynamischer Programmierung die chromatische Zahl eines Graphen berechnet. Die Idee ist simpel: Unter allen minimalen legalen Färbungen φ des gegebenen Graphen $G = (V,E)$ (das heißt, φ benutzt genau $\chi(G)$ Farben) muss wenigstens eine sein, die eine *maximale* unabhängige Menge als Farbklasse enthält (siehe Übung 3.59). Betrachte eine nicht leere Teilmenge $U \subseteq V$. Hat man nun die chromatische Zahl aller induzierten Teilgraphen $G[U']$ für jede echte Teilmenge $U' \subset U$ bereits bestimmt, so lässt sich die chromatische Zahl des induzierten Teilgraphen $G[U]$ mit der Formel

$$\chi(G[U]) = 1 + \min\{\chi(G[U - W])\} \tag{3.10}$$

berechnen, wobei das Minimum über alle maximalen unabhängigen Mengen $W \subseteq U$ zu nehmen ist. Der Algorithmus von Lawler wird ausführlich in Abschnitt 7.2 behandelt.

Übung 3.59. Zeigen Sie, dass unter allen minimalen legalen Färbungen φ eines gegebenen Graphen G (das heißt, φ benutzt genau $\chi(G)$ Farben) wenigstens eine sein muss, die eine *maximale* unabhängige Menge als Farbklasse enthält.

3.6 Entscheidungs-, Optimierungs- und Suchprobleme

Die Antwort auf eine algorithmische Fragestellung ist nicht immer „ja" oder „nein". Manchmal soll auch etwas explizit ausgerechnet oder konstruiert werden. Wir betrachten noch einmal das TSP. Wenn die Frage nach der Existenz einer Rundreise mit Kosten von höchstens k mit „ja" beantwortet wird, möchte man in den meisten Fällen als Nächstes eine solche Rundreise finden. Es wäre ebenfalls interessant zu wissen, welche Kosten eine kürzeste Rundreise besitzt. Wird der Wert einer optimalen Lösung gesucht, so sprechen wir von Optimierungsproblemen. Sollen explizit eine oder mehrere Lösungen angegeben werden, so sprechen wir von Such- oder Konstruktionsproblemen. Wir werden für Optimierungsprobleme die gleichen Problembezeichner verwenden, diese jedoch mit dem Zusatz „MIN" bzw. „MAX" ergänzen.

Optimierungsproblem
Suchproblem

Die *Optimierungsvariante von* TSP ist somit die Aufgabe, die Länge einer möglichst kurzen Rundreise für den Besuch aller Orte zu bestimmen. Als Graphenproblem kann man dies wie folgt formulieren:

MIN TSP

MIN TSP	
Gegeben:	Ein vollständiger Graph $K_n = (V,E)$ und eine Kostenfunktion $c : E \to \mathbb{N}^+$.
Ausgabe:	Die bezüglich c minimalen Kosten eines Hamilton-Kreises in K_n.

Das *Suchproblem* besteht für TSP in der Aufgabe, eine Rundreise mit minimalen Kosten zu finden (oder, als eine weitere Variante des Suchproblems, eine Rundreise zu finden, die einen vorgegebenen Kostenwert nicht überschreitet, sofern es eine solche gibt). Die erste dieser beiden Varianten des Suchproblems für TSP kann also folgendermaßen formuliert werden:

SEARCH TSP

SEARCH TSP	
Gegeben:	Ein vollständiger Graph $K_n = (V,E)$ und eine Kostenfunktion $c : E \to \mathbb{N}^+$.
Ausgabe:	Ein Hamilton-Kreis in K_n mit minimalen Kosten bezüglich c.

Es stellt sich die Frage, welche dieser Problemvarianten – TSP, MIN TSP oder SEARCH TSP – allgemeiner als die anderen ist. Die folgenden Überlegungen zeigen,

wie man aus einer Lösung einer jeden dieser Varianten Lösungen für die jeweils anderen beiden Varianten erhalten kann – man sagt dann, dass sich das eine Probleme auf das andere reduzieren lässt.[12]

Ist etwa eine optimale Lösung gegeben, so kann in der Regel relativ einfach der Wert der optimalen Lösung ermittelt werden; beim Problem des Handelsreisenden zum Beispiel muss man lediglich die Kosten entlang der gegebenen optimalen Route aufaddieren. Also lässt sich MIN TSP leicht auf SEARCH TSP reduzieren, d. h., wenn man das Suchproblem effizient lösen könnte, so könnte man auch das entsprechende Optimierungsproblem effizient lösen.

Ist der Wert einer optimalen Lösung gegeben, so lässt sich unmittelbar entscheiden, ob dieser Wert eine vorgegebene Schranke über- oder unterschreitet. Also lässt sich TSP direkt auf MIN TSP reduzieren.

Die umgekehrten Betrachtungen sind in der Regel schon etwas komplizierter, aber trotzdem häufig möglich.

Angenommen, es steht ein Algorithmus A_{Dec} zur Verfügung, der entscheiden kann, ob es eine optimale Rundreise mit Kosten von höchstens k gibt. Da eine Rundreise jede Kante höchstens ein Mal verwendet, sind die Kosten jeder Rundreise höchstens so groß wie die Summe C aller Kantenbewertungen. Der Algorithmus A_{Dec} für das Entscheidungsproblem TSP würde für den Kostenwert C immer „ja" antworten. Der Wert einer optimalen Rundreise kann nun wie folgt durch die wiederholte Anwendung von A_{Dec} ermittelt werden. Wenn es eine Rundreise mit Kosten von höchstens $C/2$ gibt, wird die nächste Frage mit $C/4$ gestellt. Gibt es keine Rundreise mit Kosten von höchstens $C/2$, so wird nach der Existenz einer Rundreise mit Kosten von höchstens $(C/2) + (C/4) = (3/4) \cdot C$ gefragt. Die nächsten Veränderungen sind $\pm C/8$, $\pm C/16$, $\pm C/32$ usw. Der Wert einer optimalen Rundreise kann also mit einer binären Aufteilungsstrategie in $\mathcal{O}(\log(C))$ Iterationen ermittelt werden, und mit diesem Aufwand lässt sich MIN TSP auf TSP reduzieren.

Angenommen, es steht ein Algorithmus A_{Opt} zur Verfügung, der den Wert einer optimalen Rundreise berechnen kann. Um nun eine optimale Rundreise zu finden, wird der Wert einer beliebigen Kante e um einen (beliebigen) positiven Wert k vergrößert. Ändert sich dadurch der Wert der optimalen Lösungen nicht, was mit dem Algorithmus A_{Opt} überprüft werden kann, so gibt es offensichtlich mindestens eine optimale Lösung, die diese Kante e nicht benötigt. Verändert sich der Wert der Lösung jedoch um k, so gehört diese Kante zu jeder optimalen Lösung. Die Erhöhung um k wird in diesem Fall wieder rückgängig gemacht. Sind alle Kanten analysiert worden, bilden die Kanten ohne Werterhöhung um k eine optimale Rundreise. Mit dieser Strategie ist es also möglich, eine optimale Rundreise für n Städte in $\mathcal{O}(n^2)$ Iterationen tatsächlich zu konstruieren, und mit diesem Aufwand lässt sich SEARCH TSP auf MIN TSP reduzieren.

[12] In Kapitel 5 wird der Begriff der *Turing-Reduzierbarkeit* eingeführt, mit dem man solche Transformationen zwischen Problemen formal beschreiben kann (siehe Übung 5.29 – die dort angegebene Definition bezieht sich zwar nur auf Entscheidungsprobleme, kann aber leicht so abgeändert werden, dass sie auch auf Such- oder Optimierungsprobleme anwendbar ist). Mit Hilfe von Reduzierbarkeiten kann man zwei gegebene Probleme hinsichtlich ihrer Komplexität vergleichen.

Konstruktionsproblem
Zählproblem

Es gibt noch weitere Problemvarianten, zum Beispiel *Konstruktionsprobleme*, bei denen alle Lösungen einer gegebenen Probleminstanz konstruiert werden, oder *Zählprobleme*, bei denen die Anzahl der Lösungen einer gegebenen Probleminstanz berechnet werden.

3.7 Literaturhinweise

Einführungen zu Graphen findet man in den Büchern von Diestel [Die06] und Jungnickel [Jun08]. Gerichtete Graphen werden von Bang-Jensen und Gutin [BJG09] ausführlich beschrieben. Ein Buch über viele spezielle Graphklassen ist das von Brandstädt, Le und Spinrad [BLS99]. Die Ungleichung von Gallai stammt aus dem Jahr 1959, siehe [Gal59]. Die Abschätzung des chromatischen Index über den maximalen Knotengrad findet man zum Beispiel in [Viz64] oder [Gup66]. Kantengraphen wurden bereits 1932 von Whitney [Whi32] eingeführt. Das Prinzip der dynamischen Programmierung geht (zumindest in der Mathematik und Informatik) auf Bellman [Bel57] zurück. Lawler [Law76] erkannte, wie man dieses Prinzip auf das Problem der Bestimmung der chromatischen Zahl eines gegebenen Graphen anwenden kann.