Kapitel 8

Graphenalgorithmen

Wie komme ich am schnellsten von Freiburg nach Königsberg, dem heutigen Kaliningrad? Wie komme ich am billigsten von Freiburg nach Königsberg? Wie transportiere ich ein Gut am billigsten von mehreren Anbietern zu mehreren Nachfragern? Wie ordne ich die Arbeitskräfte meiner Firma am besten denjenigen Tätigkeiten zu, für die sie geeignet sind? Wann kann ich frühestens mit meinem Hausbau fertig sein, wenn die einzelnen Arbeiten in der richtigen Reihenfolge ausgeführt werden? Wie besuche ich alle meine Kunden mit einer kürzestmöglichen Rundreise? Welche Wassermenge kann die Kanalisation in Freiburg höchstens verkraften? Wie muß ein Rundweg durch Königsberg aussehen, auf dem ich jede Brücke über den Pregel genau einmal überquere und am Schluß zum Ausgangspunkt zurückkomme? Diese und viele andere Probleme lassen sich als Probleme in Graphen formulieren und mit Hilfe von Graphenalgorithmen lösen. In einem Graphen wird dabei die wesentliche Struktur des Problems, befreit von unbedeutenden Nebenaspekten, repräsentiert.

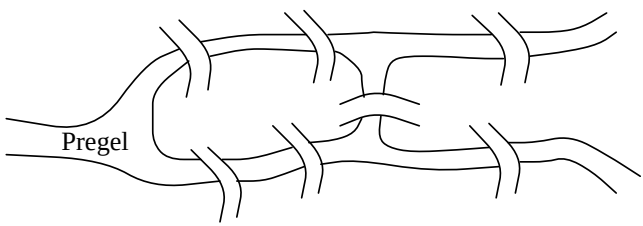


Abbildung 8.1

Abbildung 8.1 zeigt einen (verzerrten) Ausschnitt aus dem Stadtplan von Königsberg, Abbildung 8.2 zeigt den dazugehörigen Graphen. Das Wesentliche am Königsberger Brückenproblem ist die Verbindungsstruktur der einzelnen Stadtteile gemäß den sieben Brücken. Jeder Stadtteil ist im Graphen durch einen Punkt, genannt Knoten, wiedergegeben; eine Verbindung ist eine Linie von einem Knoten zu einem anderen Knoten,

genannt Kante. In unserem Beispiel entspricht eine Verbindung gerade einer Brücke. Bereits 1736 löste Euler [48] das Königsberger Brückenproblem: Er stellte fest, daß der gewünschte Rundweg nicht möglich ist.

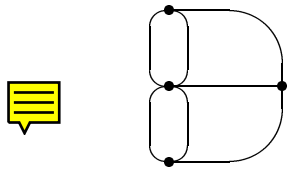


Abbildung 8.2

Im Laufe dieses Kapitels werden wir Beispiele für andere Graphenprobleme und entsprechende Lösungsalgorithmen kennenlernen. Insbesondere kann man sich vorstellen, daß Verbindungen — anders als beim Königsberger Brückenproblem — mit einer Richtung ausgezeichnet sind und in Gegenrichtung nicht benutzt werden dürfen, wie etwa Einbahnstraßen in einer Stadt. Ähnliches gilt bei der Kanalisation oder beim Hausbau (vgl. Abbildung 8.3, bei der ein Pfeil einem Vorgang entspricht). Betrachten wir zunächst solche Graphen.

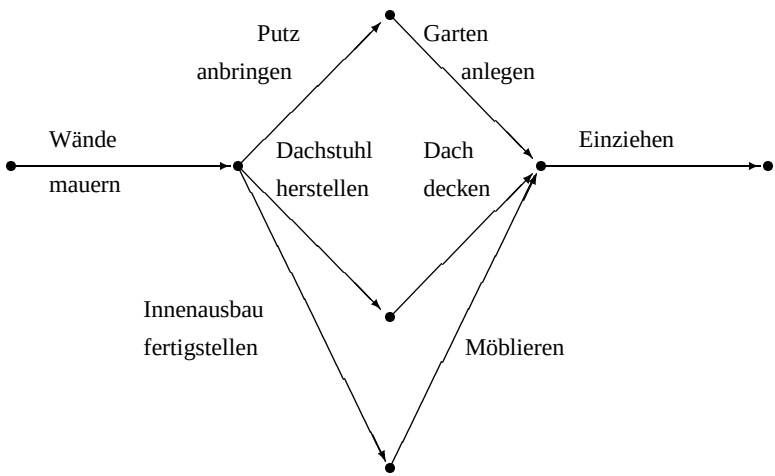


Abbildung 8.3

Ein *gerichteter Graph* $G = (V, E)$ (englisch: *digraph*) besteht aus einer Menge $V = \{1, 2, \ldots, |V|\}$ von *Knoten* (englisch: *vertices*) und einer Menge $E \subseteq V \times V$ von *Pfeilen* (englisch: *edges, arcs*). Ein Paar $(v, v') \in E$ heißt *Pfeil von v nach v'*. Wir nennen v den

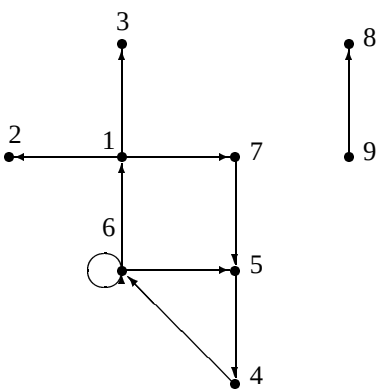
Anfangs- und v' den *Endknoten* des Pfeils (v,v'); v und v' heißen auch *adjazent*; v (und ebenso v') heißt mit *e inzident*; ebenso nennen wir e inzident mit v und v'. Wir werden Knoten eines Graphen stets als Punkte, Pfeile als Verbindungslinien mit einer auf den Endknoten gerichteten Pfeilspitze darstellen. Wir beschränken uns auf endliche Mengen von Knoten und Pfeilen, also auf *endliche Graphen*; weil E eine Menge ist, kann in diesen Graphen jeder Pfeil höchstens einmal auftreten (wir erlauben keine *parallelen* Pfeile).

Für die Effizienz von Graphenalgorithmen, sowohl im Hinblick auf Speicherplatz als auch im Hinblick auf Laufzeit, ist es wichtig, Graphen geeignet zu speichern. Wir betrachten drei naheliegende Möglichkeiten der Speicherung eines Graphen $G = (V,E)$.

Speicherung in einer Adjazenzmatrix

Ein Graph $G = (V,E)$ wird in einer Boole'schen $|V| \times |V|$-Matrix $A_G = (a_{ij})$, mit $1 \leq i \leq |V|$, $1 \leq j \leq |V|$ gespeichert, wobei

$$a_{ij} = \begin{cases} 0 & \text{falls } (i,j) \notin E; \\ 1 & \text{falls } (i,j) \in E. \end{cases}$$



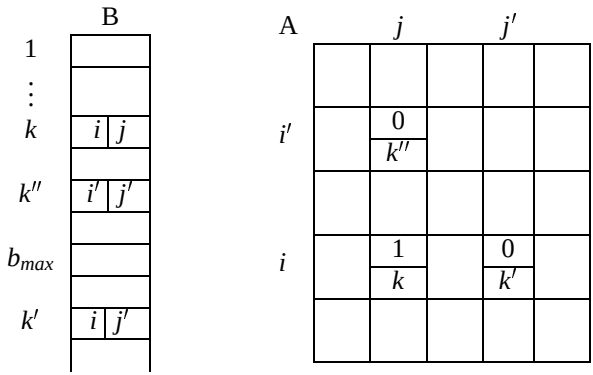
(a)

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

(b)

Abbildung 8.4

Abbildung 8.4 (b) ist die Adjazenzmatrix zum Graphen aus Abbildung 8.4 (a).



$A[i, j]$ ist bedeutsam, $A[i, j']$ und $A[i', j]$ sind es nicht.

Abbildung 8.5

Bei der Speicherung eines Graphen mit Knotenmenge V in einer Adjazenzmatrix ergibt sich ein Speicherbedarf von $\Theta(|V|^2)$. Dieser Speicherbedarf ist nicht abhängig von der Anzahl der Pfeile im Graphen; enthält der Graph vergleichsweise wenige Pfeile, so ist der Speicherplatzbedarf vergleichsweise hoch. Verwendet man die Adjazenzmatrix *ohne Zusatzinformation*, so benötigen die meisten Algorithmen wegen der erforderlichen Initialisierung der Matrix oder der Berücksichtigung aller Einträge der Matrix $\Omega(|V|^2)$ Rechenschritte.

Dem läßt sich aber mit Zusatzinformationen abhelfen, die den Platzbedarf nicht über $O(|V|^2)$ hinaus erhöhen. Dies gelingt mit einem zusätzlichen Feld B, das für jeden in der Adjazenzmatrix benutzten Eintrag einen Feldeintrag enthält; für in der Adjazenzmatrix zwar vorhandene, aber nicht mit einer Bedeutung belegte Einträge gibt es im Feld keinen Eintrag (vgl. Abbildung 8.5).

Nun geht es darum, für gegebenen Zeilenindex i und Spaltenindex j der Matrix A festzustellen, ob $A[i, j]$ eine Bedeutung besitzt, also einen bereits benutzten Eintrag bezeichnet. Dazu speichern wir mit $A[i, j]$ neben dem gewünschten Bit für die Adjazenz von Knoten i mit Knoten j einen Index k des Feldes B. Im Feld B werden an Stelle k die Matrixindizes i und j gespeichert, wenn der Matrixeintrag Bedeutung besitzt. Im Feld B sind stets die Einträge mit Indizes 1 bis *bmax* bedeutsam. Setzen wir die Definitionen

```
const    knotenzahl = {Anzahl |V| der Knoten};
         pfeilzahl = {Anzahl |E| der Pfeile};
type     knotentyp = 1 .. knotenzahl;
         pfeiltyp = 1 .. pfeilzahl;
         bit = 0 .. 1;
         matrixeintrag = record
                           adjazent : bit;
                           index : pfeiltyp
                         end;
```

```
        feldeintrag = record
                          zeile, spalte : knotentyp
                      end;
        matrix = array [knotentyp, knotentyp] of matrixeintrag;
        feld = array [pfeiltyp] of feldeintrag;
var     A : matrix;
        B : feld;
        i,j : knotentyp;
        bmax : pfeiltyp
```

voraus, so ist ein Eintrag $A[i,j]$ genau dann *bedeutsam* (*echt, gültig*), wenn $1 \leq A[i,j].index \leq bmax$, $B[A[i,j].index].zeile = i$ und $B[A[i,j].index].spalte = j$ gelten. Damit ist es gelungen, die Initalisierung der Matrix A durch die Initialisierung des Feldes B zu ersetzen:

```
{Initialisiere A:}
    {Initialisiere B:}
        bmax := 0
```

Die Laufzeit von Graphenalgorithmen bei Verwendung einer Adjazenzmatrix ist also nicht unbedingt durch $\Omega(|V|^2)$ nach unten beschränkt. Trotzdem bleiben typische Operationen, wie etwa das Inspizieren aller von einem gegebenen Knoten ausgehenden Pfeile, für Graphen mit wenigen Pfeilen ineffizient. Betrachten wir nun eine hierfür besser geeignete Speicherungsform.

Speicherung in Adjazenzlisten

Hier wird für jeden Knoten eine lineare, verkettete Liste der von diesem Knoten ausgehenden Pfeile gespeichert. Die Knoten werden als lineares Feld von $|V|$ Anfangszeigern auf je eine solche Liste verwaltet. Abbildung 8.6 zeigt Adjazenzlisten für den Graphen aus Abbildung 8.4 (a).

Die i-te Liste enthält ein Listenelement mit Eintrag j für jeden Endknoten eines Pfeils $(i,j) \in E$. In pascalähnlicher Notation läßt sich diese Struktur wie folgt definieren:

```
const   knotenzahl = {Anzahl |V| der Knoten};
type    knotentyp = 1 .. knotenzahl;
        pfeilzeiger = ↑pfeilelement;
        pfeilelement = record
                           endknoten : knotentyp;
                           next : pfeilzeiger
                       end;
        feld = array [knotentyp] of pfeilzeiger;
var     adjazenzlisten : feld
```

Für einen Graphen $G = (V,E)$ benötigen Adjazenzlisten $\Theta(|V| + |E|)$ Speicherplätze. Adjazenzlisten unterstützen viele Operationen, z.B. das Verfolgen von Pfeilen in Graphen, sehr gut. Andere Operationen dagegen werden nur schlecht unterstützt, insbesondere das Hinzufügen und Entfernen von Knoten.

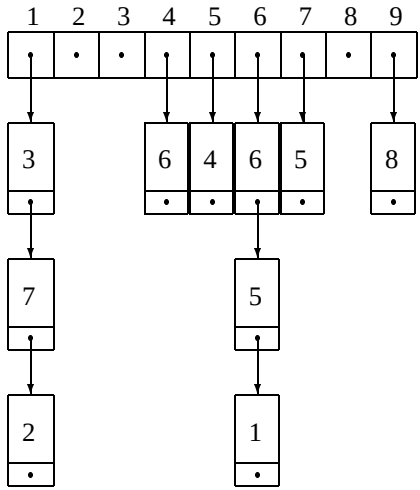


Abbildung 8.6

Speicherung in einer doppelt verketteten Pfeilliste

Die bei Adjazenzlisten fehlende Dynamik kann erreicht werden, indem man die Knoten in einer doppelt verketteten Liste speichert, anstatt sie in einem Feld fester Größe zu verwalten. Jedes Listenelement dieser doppelt verketteten Liste enthält drei Verweise, zwei davon auf benachbarte Listenelemente und einen auf eine Pfeilliste, wie bei Adjazenzlisten. Jede Pfeilliste ist doppelt verkettet; statt einer Knotennummer besitzt jedes Pfeillistenelement einen Verweis auf ein Element der Knotenliste.

Abbildung 8.7 zeigt eine solche *doppelt verkettete Pfeilliste* (englisch: *doubly connected arc list; DCAL*) für das Beispiel aus Abbildung 8.4 (a).

Natürlich kann man bei den Listenelementen weitere Informationen speichern. In Abbildung 8.7 haben wir bei den Listenelementen für Knoten die Knotennummer explizit gespeichert; ebensogut könnte man Pfeilnummern oder ähnliches in der DCAL verwalten. Ohne diese Verwaltungsinformation kann eine DCAL in pascalähnlicher Notation wie folgt beschrieben werden:

```
type  knotenzeiger = ↑knotenelement;
      pfeilzeiger = ↑pfeilelement;
      knotenelement =   record
                        {diverse Informationen, wie z.B. Knotennummer}
                          pre, next : knotenzeiger;
                          pfeilliste : pfeilzeiger
                        end;
      pfeilelement  =   record
                          next : pfeilzeiger;
                          endknoten : knotenzeiger;
```

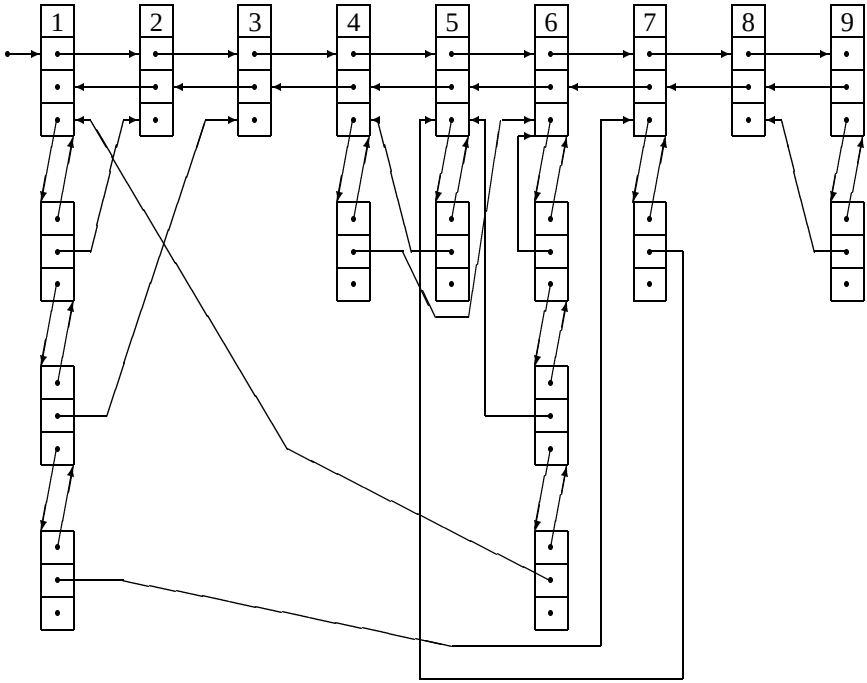


Abbildung 8.7

```
                    case pfeillistenanfang : boolean of
                        true : (kno : knotenzeiger);
                        false : (pre : pfeilzeiger)
                    end;
var     dcal : knotenzeiger;
```

Wegen der etwas einfacheren Struktur werden wir die Adjazenzlistenrepräsentation von Graphen überall dort der DCAL vorziehen, wo sich dies nicht negativ auf die Effizienz von Algorithmen auswirkt.

Bevor wir uns nun die algorithmische Lösung einiger Graphenprobleme genauer ansehen, wollen wir wichtige Grundbegriffe der Graphentheorie kurz rekapitulieren. Weitergehende Definitionen findet man in Standardlehrbüchern zur Graphentheorie und zu Graphenalgorithmen [18, 30, 49, 65, 66, 75, 83, 104, 121, 144] und teilweise auch in Lehrbüchern über Algorithmen und Datenstrukturen.

Sei $G = (V,E)$ ein gerichteter Graph (englisch: *directed graph; Digraph*). Der *Eingangsgrad* (englisch: *indegree*) $indeg(v)$ eines Knotens v ist die Anzahl der in v einmündenden Pfeile, also $indeg(v) = |\{v' | (v',v) \in E\}|$. Im Digraphen des Beispiels der Abbildung 8.8 ist $indeg(0) = 1$, $indeg(2) = 2$. Der *Ausgangsgrad* (englisch: *outdegree*) $outdeg(v)$ ist die Anzahl der von v ausgehenden Pfeile, also $outdeg(v) = |\{v' | (v,v') \in E\}|$. Ein Digraph $G' = (V',E')$ ist ein *Teilgraph* von G, geschrieben als $G' \subseteq G$, falls

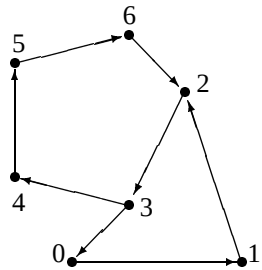


Abbildung 8.8

$V' \subseteq V$ und $E' \subseteq E$ ist. Für $V' \subseteq V$ *induziert* V' den Teilgraphen $(V', E \cap (V' \times V'))$, auch *Untergraph* genannt. Im durch V' induzierten Teilgraphen findet man also *alle* Pfeile aus E wieder, die lediglich mit Knoten aus V' inzidieren. Der durch $V - V'$ induzierte Teilgraph von G wird als $G - V'$ notiert; für einelementiges $V' = \{v'\}$ schreiben wir auch $G - v'$. Für den Digraphen der Abbildung 8.8 ist mit $V' = \{0,3,4,5\}$ der Graph $(V', \{(3,0),(4,5)\})$ ein Teilgraph; der Graph $G' = (V', \{(3,0),(3,4),(4,5)\})$ ist der durch V' induzierte Teilgraph.

Ein *Weg* (englisch: *path*) von v nach v', wobei $v, v' \in V$, ist der durch eine Folge $(v_0, v_1, \ldots, v_k)$ von Knoten mit $v_0 = v$, $v_k = v'$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i < k$ beschriebene Teilgraph $G' = (V', E')$ von G, für den $V' = \{v_0, v_1, \ldots, v_k\}$ und $E' = \{(v_i, v_{i+1}) | \, 0 \leq i < k\}$; k ist die *Länge* des Weges. Für jedes $v \in V$ gibt es also den trivialen Weg von v nach v mit Länge 0. In dem in Abbildung 8.8 gezeigten Digraphen ist beispielsweise die Knotenfolge (2, 3, 4, 5, 6, 2, 3, 0) ein Weg von Knoten 2 nach Knoten 0.

Ein Weg heißt *einfach*, wenn kein Knoten mehrfach besucht wird, d.h., wenn für alle i, j mit $0 \leq i < j \leq k$ gilt, daß $v_i \neq v_j$ ist. Der im Beispiel genannte Weg im Digraph der Abbildung 8.8 ist also nicht einfach; Weg (0, 1, 2, 3, 4) dagegen ist einfach.

Ein *Zyklus* ist ein Weg, der am Ausgangsknoten endet, also ein Weg von einem Knoten v nach v. Wir wollen im folgenden der Einfachheit halber triviale Wege und triviale Zyklen, also Wege und Zyklen, die nur aus einem Knoten und keinem Pfeil bestehen, aus unseren Betrachtungen ausschließen. Ein Digraph heißt *zyklenfrei* oder *azyklisch*, wenn er keinen Zyklus enthält. Der Digraph aus Abbildung 8.8 ist also nicht zyklenfrei: Er enthält die beiden (einfachen) Zyklen (2, 3, 4, 5, 6, 2) und (0, 1, 2, 3, 0).

Manchmal interessieren wir uns für Wege, die nur einen Teil aller Pfeile benutzen. Für $F \subseteq E$ schreiben wir $v \longrightarrow_F^* v'$ genau dann, wenn es einen Weg von v nach v' gibt, der nur Pfeile aus F benutzt. Wenn $v \longrightarrow_E^* v'$ gilt, so bezeichnen wir v' als von v aus *erreichbar*.

Wir haben Bäume und Ansammlungen von Bäumen bereits in anderen Kapiteln als Datenstrukturen kennengelernt. Auch als Graphen haben sie eine besondere Bedeutung. Ein Digraph $G = (V, E)$ heißt *gerichteter Wald*, wenn E zyklenfrei ist und $indeg(v) \leq 1$ für alle $v \in V$. Jeder Knoten v mit $indeg(v) = 0$ ist eine *Wurzel* des Waldes. Ein gerichteter Wald mit genau einer Wurzel ist ein *gerichteter Baum* (*Wurzelbaum*). Wie wir schon von der Datenstruktur Baum wissen, gibt es in einem gerichteten Baum von der Wurzel zu jedem Knoten genau einen Weg. Im Beispiel der Abbildung 8.8 ist der

oben beschriebene Teilgraph $(\{0,3,4,5\}, \{(3,0)\}, \{(4,5)\}$ ein gerichteter Wald mit Wurzeln 3 und 4; der von $\{0,3,4,5\}$ induzierte Untergraph ist ein Baum mit Wurzel 3.

Für einen Knoten v eines gerichteten Baums ist der *Teilbaum mit Wurzel v* der von den *Nachfolgern* $\{v' | v \longrightarrow_E^* v'\}$ von v induzierte Teilgraph. Für manche Berechnungen benötigen wir einen Wald, der alle Knoten eines gegebenen Digraphen enthält. Für einen Digraphen $G = (V,E)$ ist ein gerichteter Wald $W = (V,F)$ mit $F \subseteq E$ ein *spannender Wald* von G. Falls W ein Baum ist, heißt W *spannender Baum* von G.

In vielen Fällen kommt es uns auf die Richung von Verbindungen zwischen Knoten nicht an. Dann vernachlässigen wir die Richtung von Pfeilen, beispielsweise indem wir erzwingen, daß zwischen zwei Knoten entweder kein Pfeil oder in jeder der beiden Richtungen ein Pfeil verläuft. Ein solcher Digraph $G = (V,E)$, für den $(v,v') \in E \iff (v',v) \in E$, heißt *ungerichteter Graph* oder einfach *Graph*. Ein Paar $((v,v'),(v',v))$ von Pfeilen heißt *Kante*. Abhängig vom modellierten Problem repräsentiert eine Kante eine in beiden Richtungen gleichzeitig benutzbare Verbindung, wie etwa eine Straße, oder eine wahlweise in jeder der beiden Richtungen — aber nicht gleichzeitig — benutzbare Verbindung, wie etwa ein Eisenbahngleis. Der Grad $deg(v)$ eines Knotens v ist gerade gleich $indeg(v)$ (und ebenfalls $outdeg(v)$), also die Anzahl der mit v inzidenten Kanten. Ein ungerichteter Graph heißt zyklenfrei oder azyklisch, falls er keinen einfachen Zyklus mit wenigstens mit drei Pfeilen enthält (natürlich enthält jeder Graph mit einer Kante bereits einen Zyklus aus zwei Pfeilen). Die übrigen Definitionen im Zusammenhang mit gerichteten Graphen gelten entsprechend.

Wir werden eine Kante der Übersicht wegen stets als *eine* Verbindungslinie ohne Pfeilspitze zeichnen und als (v,v') notieren, wobei die Reihenfolge der Knoten ohne Bedeutung ist (manche Autoren verwenden auch $[v,v']$). Davon machen wir beispielsweise im Algorithmus zur Berechnung der zweifachen Zusammenhangskomponenten Gebrauch (siehe Abschnitt 8.4.1). Beide Knoten v und v' der Kante $(v,v') = (v',v)$ werden als Endknoten bezeichnet.

8.1 Topologische Sortierung

Ein Digraph kann stets als eine binäre Relation angesehen werden; ein zyklenfreier Digraph beschreibt also eine Halbordnung. Liest man etwa einen Pfeil als „ist teurer als“, so stößt man beim Betrachten des in Abbildung 8.4 (a) dargestellten Digraphen auf einen Widerspruch. Eine topologische Sortierung eines Digraphen ist nun eine vollständige Ordnung über den Knoten des Graphen, die mit der durch die Pfeile ausgedrückten partiellen Ordnung verträglich ist. Genauer: Eine topologische Sortierung eines Digraphen $G = (V,E)$ ist eine Abbildung $ord\colon V \longrightarrow \{1,\ldots,n\}$ mit $n = |V|$, so daß mit $(v,w) \in E$ auch $ord(v) < ord(w)$ gilt.

Nun ist G genau dann zyklenfrei, wenn es für G eine topologische Sortierung gibt. Dies überlegt man sich wie folgt. Es ist klar, daß aus der Existenz einer topologischen Sortierung die Zyklenfreiheit von G folgt. Daß es zu jedem zyklenfreien Digraphen $G = (V,E)$ auch eine topologische Sortierung gibt, kann man durch Induktion über die Knotenzahl zeigen. Falls $|V| = 1$, dann gibt es natürlich eine topologische Sortierung:

Man definiert einfach $ord(1) = 1$. Falls $|V| > 1$, so betrachtet man einen Knoten v mit $indeg(v) = 0$. Wegen der Zyklenfreiheit von G muß es einen solchen Knoten geben. Durch Entfernen von v entsteht ein um einen Knoten verkleinerter Digraph. An dessen topologische Sortierung wird v vorne angefügt. Hieraus ergibt sich unmittelbar ein Algorithmus für die topologische Sortierung:

Algorithmus *Topologische Sortierung (Grobentwurf)*
{*liefert zu einem Digraphen* $G = (V,E)$ *eine topologische Sortierung* *ord*[*knotentyp*]}
begin
  lfd.Nr. := 0;
  while *G hat wenigstens einen Knoten v mit Eingangsgrad* 0 **do**
    begin
      erhöhe lfd.Nr. um 1;
      ord[*v*] := *lfd.Nr.*;
      $G := G - v$
    end;
  if $G = \emptyset$
    then *G ist zyklenfrei*
    else *G hat Zyklen*
end {*Topologische Sortierung*}

Es ist noch zu klären, wie man einen Knoten mit Eingangsgrad 0 findet. Hier ist es naheliegend, an einem beliebigen Knoten zu beginnen und Pfeile rückwärts zu verfolgen. Da der Digraph G zyklenfrei ist, trifft man nicht mehrmals auf einen Knoten. Also endet das Zurückverfolgen von Pfeilen spätestens, wenn alle Knoten besucht worden sind. Das Zurückverfolgen von Pfeilen kann aber nur in einem Knoten mit Eingangsgrad 0 enden. Damit hat man einen solchen Knoten gefunden.

Wenn man dazu jedoch stets den ganzen Digraphen durchläuft, so benötigt man pro Knoten wenigstens $\Omega(n)$ Schritte, insgesamt also wenigstens $\Omega(n^2)$ Schritte.

Es ist sicherlich effizienter, den jeweils aktuellen Eingangsgrad zu jedem Knoten zu speichern und auf dem neuesten Stand zu halten. Dann genügt es, statt einen Knoten aus G zu entfernen, die Eingangsgrade seiner direkten Nachfolger zu verringern. Um einen Knoten mit Eingangsgrad 0 schnell zu finden, verwalten wir die Menge aller Knoten mit aktuellem Eingangsgrad 0. Diese Menge ändert sich höchstens bei der Wahl eines Knotens für die topologische Sortierung und beim Verringern der Eingangsgrade direkter Nachfolger eines gewählten Knotens. Damit ergibt sich die folgende Präzisierung des Algorithmus für die topologische Sortierung eines Digraphen:

Algorithmus *Topologische Sortierung (Präzisierung)*
{*liefert zu einem Digraphen* $G = (V,E)$ *eine topologische Sortierung* *ord*[*knotentyp*]}
var *lfd.Nr.* : 0 .. *knotenzahl*;
  Gradnull : **stack of** *knotentyp*;
  Eingrad : **array** [*knotentyp*] **of** 0 .. *knotenzahl* − 1
begin
  1. *setze Eingrad*[*v*] *auf den Eingangsgrad von v in G,*

```
        für alle v ∈ V;
   2.   übernimm alle Knoten v ∈ V mit Eingangsgrad 0
        nach Gradnull;
   3.   lfd.Nr. := 0;
   4.   while Gradnull ≠ ∅ do
          begin
            wähle v ∈ Gradnull;
            entferne v aus Gradnull;
            erhöhe lfd.Nr. um 1;
            ord[v] := lfd.Nr.;
   {*1*}    for all (v,w) ∈ E do
   {*2*}      begin
   {*3*}        erniedrige Eingrad[w] um 1;
   {*4*}          if Eingrad[w] = 0
   {*5*}            then füge w zu Gradnull hinzu
   {*6*}      end
          end;
   5.   if lfd.Nr. = knotenzahl
          then G ist zyklenfrei
          else G hat Zyklus
end {Topologische Sortierung}
```

Die einzelnen Schritte des Algorithmus lassen sich leicht präzisieren, wenn wir die Speicherung des gegebenen Digraphen in Adjazenzlistenform annehmen, wie eingangs angegeben:

```
{1. setze Eingrad ... }
  for v := 1 to knotenzahl do Eingrad[v] := 0;
  for v := 1 to knotenzahl do
    begin
      p := adjazenzliste[v];
      while p ≠ nil do
        begin
          erhöhe Eingrad[p↑.endknoten] um 1;
          p:= p↑.next
        end
    end


{2. übernimm ... }
  Gradnull := leerer Stapel;
  for v := 1 to knotenzahl do
    if Eingrad[v] = 0
      then füge v zu Gradnull hinzu;


{Die Zeilen {*1*} bis {*6*} in 4. while Gradnull ... }
  p := adjazenzliste[v];
  while p ≠ nil do
```

```
begin
  w := p↑.endknoten;
  {*3*};
  {*4*};
  {*5*};
  p := p↑.next
end
```

Damit benötigt Schritt 1 des Verfahrens eine Laufzeit von $O(|V| + |E|)$; Schritt 2 kommt wegen der konstanten Zeit für jede einzelne Stapeloperation mit einer Laufzeit von $O(|V|)$ aus, und Schritt 3 kann in konstanter Zeit ausgeführt werden. Die **while**-Schleife in Schritt 4 wird gerade $|V|$-mal durchlaufen; in der inneren **while**-Schleife wird jeder Pfeil im Digraphen gerade einmal inspiziert. Damit benötigt Schritt 4 eine Laufzeit von $O(|V| + |E|)$. Mit der konstanten Laufzeit von Schritt 5 ergibt sich in der Summe eine Laufzeit von $O(|V| + |E|)$ für die Berechnung einer topologischen Sortierung für einen Digraphen $G = (V, E)$. Ebenfalls in Zeit $O(|V| + |E|)$ kann somit ein Digraph $G = (V, E)$ auf Zyklenfreiheit getestet werden.

8.2 Transitive Hülle

Beschäftigen wir uns nun mit der Erreichbarkeit von Knoten in einem Graphen, ausgehend von anderen Knoten. So kann man sich etwa fragen, welche Knoten von einem gegebenen Knoten aus erreichbar sind, oder ob es womöglich einen Knoten gibt, von dem aus jeder andere erreicht werden kann. In einem Zyklus beispielsweise kann jeder Knoten von jedem anderen aus erreicht werden. Um solche Fragen zu beantworten, kann es sinnvoll sein, von vornherein alle Erreichbarkeiten explizit zu berechnen. Sind die Knoten eines Digraphen beispielsweise Straßenkreuzungen und die Pfeile verbindende Einbahnstraßen, so ist Kreuzung Z von Kreuzung X aus gerade dann erreichbar, wenn es entweder einen Pfeil von X nach Z gibt oder eine Kreuzung Y, die von X aus erreichbar ist und von der aus Z erreichbar ist. Natürlich ist auch jede Kreuzung von sich selbst aus erreichbar. Dies führt zur Definition der reflexiven transitiven Hülle.

Ein Digraph $G^* = (V, E^*)$ ist die *reflexive, transitive Hülle* eines Digraphen $G = (V, E)$, wenn genau dann $(v, v') \in E^*$ ist, wenn es einen Weg von v nach v' in G gibt. Die reflexive, transitive Hülle (kurz: *Hülle*) des Digraphen aus Abbildung 8.8 enthält alle Pfeile zwischen Knoten, weil jeder Knoten von jedem aus erreicht werden kann. Für den speziellen Fall, daß der gegebene Digraph azyklisch ist, ist die Berechnung der transitiven Hülle einfacher als im allgemeinen Fall. Betrachten wir jedoch zunächst den allgemeinen Fall.

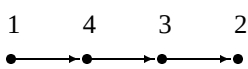
8.2.1 Transitive Hülle allgemein

Erinnern wir uns daran, daß wir die Existenz eines Weges von v nach v' in $G = (V,E)$ mit $v \to_E^* v'$ notieren. Wenn wir nun schon wissen, daß $v \to_E^* v'$ und $v' \to_E^* v''$ gelten, so können wir auf die Gültigkeit von $v \to_E^* v''$ schließen. Damit ergibt sich unmittelbar ein erster Ansatz eines Algorithmus zur Berechnung der transitiven Hülle. Beginnend mit der Adjazenzmatrix A für den gegebenen Digraphen suchen wir zu allen Pfeilen (i,j) alle Pfeile (j,k) und vermerken die daraus entstehenden Pfeile (i,k) in der Adjazenzmatrix:

Algorithmus *Berechnung von Pfeilen der reflexiven transitiven Hülle*
1. **for** $i := 1$ **to** *knotenzahl* **do** $A[i,i] := 1$;
2. **for** $i := 1$ **to** *knotenzahl* **do**
   for $j := 1$ **to** *knotenzahl* **do**
   if $A[i,j] = 1$ **then**
   for $k := 1$ **to** *knotenzahl* **do**
   if $A[j,k] = 1$ **then** $A[i,k] := 1$

end *{Berechnung von Pfeilen}*

Es ist klar, daß mit diesem Algorithmus tatsächlich einige Wege berechnet werden; man sieht aber auch leicht, daß nicht alle Wege gefunden werden. Abbildung 8.9 (a) zeigt ein Beispiel für einen Graphen, 8.9 (b) dessen Adjazenzmatrix, und 8.9 (c) das Resultat der Anwendung des Algorithmus zum Finden von Pfeilen der reflexiven transitiven Hülle. Man erkennt, daß alle aus bis zu zwei Pfeilen bestehenden Wege gefunden worden sind. Der aus drei Pfeilen bestehende Weg vom Knoten 1 zum Knoten 2 wurde aber nicht entdeckt. Wege größerer Länge werden gefunden, wenn man den Algorithmus wiederholt solange anwendet, bis sich keine neuen Pfeile ergeben. Dies ist aber nicht besonders effizient: Bereits die einfache Anwendung des Algorithmus benötigt wegen der drei im Schritt 2 geschachtelten **for**-Schleifen eine Laufzeit von $\Theta(|V|^3)$. Folgende Überlegung zeigt, daß es auch schneller geht.



(a)

A	1	2	3	4
1	0	0	0	1
2	0	0	0	0
3	0	1	0	0
4	0	0	1	0

(b)

A	1	2	3	4
1	1	0	1	1
2	0	1	0	0
3	0	1	1	0
4	0	1	1	1

(c)

Abbildung 8.9

Zum Auffinden eines Weges vom Knoten i zum Knoten k betrachten wir nicht jede mögliche Zusammensetzung von Teilwegen, sondern nur eine spezielle. Ein Weg von einem Knoten i zu einem Knoten k ist entweder ein Pfeil von i nach k oder kann so in

einen Weg von i nach j und einen Weg von j nach k zerlegt werden, daß j die größte Nummer eines Knotens auf dem Weg zwischen i und k ist (ohne i und k selbst). Die Knotennummern sind dabei die mit dem Graphen willkürlich festgelegten, also nicht etwa die durch topologische Sortierung ermittelten. Wir ermitteln nun Wege in einer Reihenfolge, die sicherstellt, daß beim Zusammensetzen der beiden Wege von i nach j und von j nach k beide nur Zwischenknoten mit einer Nummer kleiner als j benutzen. Dies ist der Fall, wenn unser Algorithmus für aufsteigende Werte von j die folgende *Invariante* erfüllt: Für das aktuelle j sind alle Wege bereits bekannt, die nur Zwischenknoten mit Nummer kleiner als j benutzen. Es ist klar, daß die Invariante anfangs gilt. Beim Zusammenfügen bereits bekannter Wege benutzt jeder resultierende Weg nur Knoten, deren Nummer höchstens j ist, also nur Knoten mit Nummer kleiner als $j+1$. Da alle beim Erhöhen von j neu gefundenen Wege den Knoten j benutzen müssen, wird auch jeder solche Weg tatsächlich gefunden. Damit ergibt sich der folgende Algorithmus für die Berechnung der reflexiven, transitiven Hülle eines Digraphen, der sich von dem zuvor angegebenen Algorithmus für das Finden von Pfeilen der Hülle nur durch das Vertauschen der beiden äußeren **for**-Schleifen in Schritt 2 unterscheidet [191]:

Algorithmus *Reflexive transitive Hülle*
1. **for** $i := 1$ **to** *knotenzahl* **do** $A[i,i] := 1$;
2. **for** $j := 1$ **to** *knotenzahl* **do**
   for $i := 1$ **to** *knotenzahl* **do**
   if $A[i,j] = 1$ **then**
   for $k := 1$ **to** *knotenzahl* **do**
   if $A[j,k] = 1$ **then** $A[i,k] := 1$

end *{Reflexive transitive Hülle}*

Die Laufzeit dieses Algorithmus ist offensichtlich beschränkt durch $O(|V|^3)$. Bei näherem Hinsehen zeigt sich, daß die innerste der drei **for**-Schleifen nur durchlaufen wird, wenn ein Pfeil von i nach j vorhanden ist. Dieser Pfeil kann aus dem gegebenen Digraphen G stammen; er kann aber auch im Verlauf der Berechnung der Hülle G^* ermittelt worden sein. Die innerste **for**-Schleife wird also nicht unbedingt $\Theta(|V|^2)$-mal, sondern nur $O(|E^*|)$-mal durchlaufen. Da jeder Durchlauf in $O(|V|)$ Schritten erledigt werden kann, ergibt sich die Gesamtlaufzeit zu $O(|V|^2 + |E^*| \cdot |V|)$.

8.2.2 Transitive Hülle für azyklische Digraphen

Betrachten wir nun das Problem der Berechnung der reflexiven, transitiven Hülle für azyklische Digraphen. Wir wollen uns die topologische Sortierung zunutze machen, indem wir die dort vergebenen Ordnungsnummern gerade als Knotennummern wählen. Wie man eine topologische Sortierung in linearer Zeit berechnen kann, wurde bereits in Abschnitt 8.1 erläutert. Wir nehmen an, daß der Digraph in Adjazenzlistenform, mit Knoten in topologischer Sortierung, gegeben ist.

Die Grundidee beim Berechnen der reflexiven, transitiven Hülle besteht darin, die Knoten in der Reihenfolge absteigender Nummern zu betrachten. Für einen betrachteten Knoten i mit Pfeil (i, j) kennen wir wegen der topologischen Sortierung bereits alle von j aus erreichbaren Knoten (vgl. Abbildung 8.10 (a)). Die Menge der von i aus erreichbaren Knoten besteht also aus i selbst und allen von j aus erreichbaren Knoten, vereinigt über alle Pfeile (i, j).

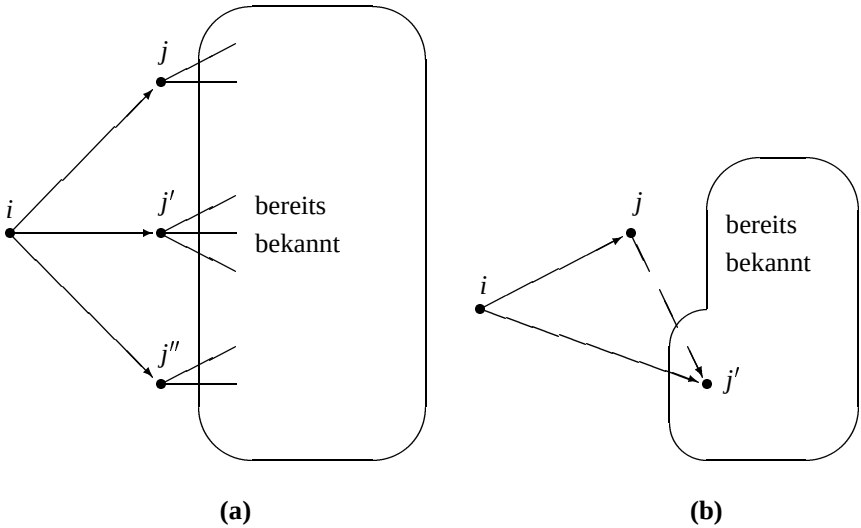


Abbildung 8.10

Für das aktuelle i betrachten wir die Endknoten j der Pfeile (i, j) aus Effizienzgründen in aufsteigender Reihenfolge ihrer Nummern. Falls nämlich bei Pfeilen (i, j) und (i, j') mit $j' > j$ Knoten j' bereits über Knoten j erreicht werden kann, so ist die Menge der über j' erreichbaren Knoten bereits in der Menge der über j erreichbaren Knoten enthalten, und j' muß zu diesem Zweck nicht weiter untersucht werden (siehe Abbildung 8.10 (b)).

Das skizzierte Verfahren läßt sich wie folgt präzisieren:

Algorithmus *Reflexive transitive Hülle für azyklischen Digraphen*
{*liefert zu einem in Adjazenzlistenrepräsentation gegebenen,*
topologisch sortierten, azyklischen Digraphen $G = (V, E)$ *die reflexive,*
transitive Hülle von G im Feld erreichbar_ab[*knotentyp*]}
var i, j, k : *knotentyp*;
erreichbar : **set of** *knotentyp*;
erreichbar_ab : **array** [*knotentyp*] **of list of** *knotentyp*;
begin
erreichbar := $\emptyset$; {*ab Knoten i als erreichbar bekannt*}
for i := *knotenzahl* **downto** 1 **do**

```
    begin
      erreichbar_ab[i] := {i};
      erreichbar := {i};
      for all (i, j) ∈ E mit aufsteigendem j do
        if j ∉ erreichbar then
          for all k ∈ erreichbar_ab[j] do
            if k ∉ erreichbar then
              begin
                füge k zu erreichbar hinzu;
                füge k zu erreichbar_ab[i] hinzu
              end;
      {setze erreichbar := ∅ :}
      for all k ∈ erreichbar_ab[i] do
        entferne k aus erreichbar
    end
end {Reflexive, transitive Hülle für azyklischen Digraphen}
```

Daß dieser Algorithmus gerade G^* berechnet, zeigen folgende Überlegungen. Es sollte klar sein, daß der Algorithmus nur Pfeile aus E^* findet. Durch ein Widerspruchsargument kann man sich davon überzeugen, daß er alle Pfeile aus E^* auch tatsächlich findet. Nehmen wir dazu an, daß es einen Pfeil in der Hülle gibt, den der Algorithmus nicht findet. Wählen wir dann i als die größte Nummer, für die der Algorithmus den Pfeil (i,h) der Hülle nicht findet. Wenn (i,h) nicht gefunden wird, muß $(i,h) \notin E$ gelten. Betrachten wir jetzt den längsten Weg $i, j, \ldots, h$ von i nach h. Weil i die größte solche Nummer ist, befindet sich h in der Liste *erreichbar_ab*$[j]$. Bei der Betrachtung des Pfeils (i,j) ist die Bedingung $j \notin$ *erreichbar* erfüllt, weil der Weg von i über j nach h der längste ist. Also wird h zur Liste *erreichbar_ab*$[i]$ hinzugefügt. Damit hat aber der Algorithmus den Pfeil (i,h) gefunden, ein Widerspruch zur Annahme.

Für jeden Knoten i sind die ab Knoten i erreichbaren Knoten einerseits als lineare Liste *erreichbar_ab*$[i]$ gespeichert. Alle Listenelemente können der Reihe nach besucht werden, in konstanter Laufzeit pro Listenelement. Außerdem kann jede Liste um weitere Elemente ergänzt werden, ebenfalls in konstanter Laufzeit pro Listenelement. Andererseits sind die ab dem aktuellen Knoten erreichbaren Knoten als Menge (Bitvektor) *erreichbar* gespeichert, damit das Enthaltensein eines Knotens in dieser Menge in konstanter Zeit geprüft werden kann; das Hinzufügen eines Elements zur Menge und das Entfernen eines Elements aus der Menge ist ebenfalls in konstanter Zeit möglich. Damit benötigt eine Abarbeitung der innersten der drei geschachtelten **for**-Schleifen eine Schrittzahl, die proportional ist zur Anzahl der ab j erreichbaren Knoten. Das für jeden weiteren Durchlauf der äußersten **for**-Schleife erforderliche Zurücksetzen der Menge der von i aus erreichbaren Knoten auf die leere Menge wird mit der entsprechenden **for**-Schleife in einer Schrittzahl erledigt, die proportional ist zur Anzahl der ab i erreichbaren Knoten. Die mittlere der drei geschachtelten **for**-Schleifen wird gerade einmal für jeden Pfeil ausgeführt. Wir müssen uns noch fragen, wie oft die innerste der drei geschachtelten **for**-Schleifen zur Ausführung kommt. Dazu betrachten wir für einen gegebenen azyklischen Digraphen $G = (V,E)$ den *reduzierten Graphen* $G_{red} = (V, E_{red})$, der durch $E_{red} = \{(i,j) | (i,j) \in E, \nexists\, k,\ i \neq k \neq j,$ mit $(i,k) \in E^*,\ (k,j) \in E^*\}$ definiert

ist. G_{red} ist also gerade G ohne transitive Pfeile. Die Definition des reduzierten Graphen ist so gewählt, daß $G^* = G^*_{red}$ gilt.

Daß die innerste der drei geschachtelten **for**-Schleifen im Algorithmus nur für Pfeile des reduzierten Graphen ausgeführt wird, sieht man wie folgt. Betrachten wir einen Pfeil (i, j), der nicht zum reduzierten Graphen gehört. Dann gibt es im reduzierten Graphen Pfeile (i,k) und (k, j), wobei wegen der topologischen Sortierung $k < j$ gilt; demnach wird Pfeil (i,k) vor (i, j) betrachtet. Weil j von k aus erreichbar ist, wird j bereits bei der Betrachtung des Pfeiles (i,k) zur Menge *erreichbar* hinzugefügt. Beim Betrachten des Pfeils (i, j) ist dann die für die Ausführung der innersten **for**-Schleife geforderte Bedingung nicht erfüllt.

Bringen wir nun unsere Überlegungen zur Laufzeit des Algorithmus zur Berechnung der Hülle zum Ende. Die letzte **for**-Schleife kostet einen Rechenschritt für jeden Pfeil der Hülle, also insgesamt Zeit $O(|E^*|)$. Die innerste der drei geschachtelten **for**-Schleifen wird für jeden Pfeil des reduzierten Graphen ausgeführt. Schlimmstenfalls sind jedes Mal größenordnungsmäßig alle Knoten erreichbar; dann ergibt sich hierfür insgesamt eine Laufzeit von $O(|E_{red}| \cdot |V|)$. Alle anderen Schritte zusammen können in Laufzeit $O(|V|)$ ausgeführt werden. Somit kann die reflexive, transitive Hülle eines azyklischen Digraphen $G = (V,E)$ in Zeit $O(|V| \cdot |E_{red}|) = O(|V| \cdot |E|) = O(|V|^3)$ ermittelt werden.

8.3 Durchlaufen von Graphen

Für manche Probleme ist es wichtig, alle Knoten eines Graphen zu betrachten. So kann man es etwa einer in einem Labyrinth eingeschlossenen Person nachfühlen, daß sie gerne sämtliche Kreuzungen von Gängen des Labyrinths in Augenschein nehmen will. Die Gänge des Labyrinths sind hier die Kanten des Graphen, und Kreuzungen von Gängen sind Knoten. Das Betrachten oder Inspizieren eines Knotens in einem Graphen nennt man auch oft *Besuchen* des Knotens. Manchmal ist es wichtig, die Knoten nach einer gewissen Systematik zu besuchen. So kann man sich leicht vorstellen, daß eine einzelne Person im Labyrinth einem Gang zunächst eine ganze Weile folgt, bevor sie vielleicht schließlich kehrt macht, also mit der Suche zunächst „in die Tiefe“ des Labyrinths geht; suchen dagegen mehrere Personen gleichzeitig, so werden sie eher vom Startpunkt aus ausschwärmen, also „in die Breite“ gehen.

Wir werden im folgenden die *Tiefensuche* und die *Breitensuche* als zwei Spezialfälle eines allgemeinen Knotenbesuchsalgorithmus kennenlernen. Es ist ganz erstaunlich, wieviel Information über die Struktur eines Graphen man alleine durch systematisches Besuchen der Knoten erhalten kann. Stellt etwa ein Graph ein Computernetz dar, wobei die Knoten des Graphen Computer und die Kanten des Graphen Verbindungsleitungen zwischen Computern sind, so kann man die Frage, ob nach dem Ausfall eines beliebigen Computers die anderen noch miteinander kommunizieren können, durch systematisches Besuchen aller Knoten lösen. Mittels spezialisierter Knotenbesuchsalgorithmen kann man aber nicht nur entscheiden, ob ein gegebener Graph *zweifach zusammenhängend* — wie für das Computernetz gefordert — ist, sondern man kann auch die größten

zweifach zusammenhängenden Teilgraphen (die zweifachen Zusammenhangskomponenten des Graphen) berechnen. Das Gerüst der Knotenbesuchsalgorithmen ist dabei stets dasselbe:

```
Algorithmus-Gerüst Besuche Knoten
{besucht in einem gegebenen Graphen oder Digraphen G = (V,E) der
 Reihe nach alle Knoten}
var B : set of knotentyp;
   {Menge der bereits besuchten Knoten}
begin
   B := {b}, wobei b ein erster besuchter Knoten ist;
   for all e ∈ E do
      markiere e als unbenutzt;
   while es gibt unbenutzte Kante/Pfeil (v,v') ∈ E mit v ∈ B do
      begin
         markiere (v,v') als benutzt;
         B := B ∪ {v'}
      end
end {Besuche Knoten}
```

Man überlegt sich leicht, daß B am Ende der Ausführung des Algorithmus *Besuche Knoten* die Menge aller von b aus erreichbaren Knoten enthält. Wir müssen noch präzisieren, wie die Menge B implementiert werden soll und welche unbenutzte Kante/Pfeil in der **while**-Schleife als jeweils nächste gewählt werden soll. Damit die die **while**-Schleife kontrollierende Bedingung schnell überprüft werden kann, speichern wir neben der Menge B noch eine weitere Knotenmenge $R \subseteq B$ derjenigen Knoten in B, von denen noch unbenutzte Kanten oder Pfeile ausgehen können — den *Rand* von B. Dann können wir den Knotenbesuchsalgorithmus wie folgt formulieren:

```
procedure Durchlaufe G = (V,E) ab Knoten b;
begin
   B := {b}; R := {b};
   while R ≠ ∅ do
      begin
         wähle Knoten v ∈ R;
         if es gibt keine unbenutzte Kante/Pfeil (v,v') ∈ E
            then lösche v aus R;
            else
               begin
                  sei (v,v') die nächste unbenutzte Kante/Pfeil ∈ E;
                  if v' ∉ B then
                     begin
                        B := B ∪ {v'};
                        R := R ∪ {v'}
                     end
               end
      end {while}
end {Durchlaufe}
```

Um zu entscheiden, welche Datenstrukturen für B und R am besten gewählt werden sollten, betrachten wir die mit B und R auszuführenden Operationen. Wir müssen B als

leere Menge initialisieren, ein Element zu B hinzufügen und prüfen können, ob ein gegebener Knoten in B enthalten ist. Für R müssen wir neben der Initialisierung als leere Menge ein Element hinzufügen können, prüfen können, ob R leer ist, ein beliebiges Element wählen können und ein gewähltes Element aus R entfernen können. Dabei ist das Initialisieren von B und R die einzige Operation, die beim Durchlaufen eines Graphen nur einmal ausgeführt wird; alle anderen Operationen werden wiederholt ausgeführt.

Wählen wir für B ein Boole'sches Array mit einem Element pro Knoten und für R eine Schlange oder einen Stapel, so benötigt jede Operation außer dem Initialisieren von B nur eine konstante Schrittzahl; das Initialisieren von B kann in $O(|V|)$ Schritten ausgeführt werden. Um für jeden Knoten $v \in V$ schnell entscheiden zu können, ob es noch eine unbenutzte Kante oder einen unbenutzten Pfeil $(v,v') \in E$ gibt, und um gegebenenfalls die nächste solche Kante zu wählen, speichern wir zusätzlich für jeden Knoten v einen Zeiger $p[v]$, der auf die nächste ungenutzte Kante in der Adjazenzliste des Knoten v zeigt. Mit den zusätzlichen Definitionen

var B : **array** [*knotentyp*] **of** *boolean*;
R : **stack of** *knotentyp*;
p : **array** [*knotentyp*] **of** *pfeilzeiger*

können wir die Prozedur für das Durchlaufen an zwei Stellen wie folgt präzisieren:

1. *es gibt keine unbenutzte Kante/Pfeil* $(v,v') \in E$:
   $p[v] =$ **nil**

2. *sei* (v,v') *die nächste unbenutzte Kante/Pfeil* $\in E$:
   $v' := p[v]\uparrow$.*endknoten*;
   $p[v] := p[v]\uparrow$.*next*

Die von der Prozedur *Durchlaufe* benötigte Zeit ist proportional zur Summe der Anzahlen der vom Startknoten b aus erreichbaren Knoten und Kanten/Pfeile, weil jeder Schleifendurchlauf nur konstant viele Schritte benötigt und einen Knoten oder eine Kante betrachtet, die danach nicht mehr betrachtet werden. Damit können alle Knoten eines Graphen in höchstens $O(|V|+|E|)$ Schritten besucht werden.

8.3.1 Einfache Zusammenhangskomponenten

Betrachten wir zunächst eine der einfachsten Anwendungen des linearen Knotenbesuchsalgorithmus. Hier geht es darum, zu einer gegebenen Menge V mit einer symmetrischen, binären Relation $E \subseteq V \times V$, deren reflexive, transitive Hülle eine Äquivalenzrelation ist, die Äquivalenzklassen zu bestimmen. Ist V die Menge der Knoten und E die Menge der Kanten eines ungerichteten Graphen, so sind dies gerade die größten zusammenhängenden Teilgraphen von $G = (V,E)$.

Genauer: Ein ungerichteter Graph G heißt genau dann *zusammenhängend*, wenn es für jedes Knotenpaar $(v,v') \in V$ einen Weg von v nach v' gibt. Eine *Zusammenhangskomponente* von G ist ein (bezüglich Mengeninklusion) maximaler zusammenhängender Untergraph von G. Ersetzen wir nun in der Prozedur *Durchlaufe* die Anweisung $B := \{b\}$ durch $B := B \cup \{b\}$, so berechnet der folgende Algorithmus gerade die Zusammenhangskomponenten eines ungerichteten Graphen $G = (V,E)$:

```
Algorithmus Zusammenhangskomponenten
  for v := 1 to knotenzahl do p[v] := adjazenzliste[v];
  B := ∅;
  for v := 1 to knotenzahl do
    if v ∉ B
      then Durchlaufe G ab Knoten v
end {Zusammenhangskomponenten}
```

Jeder Aufruf der Prozedur *Durchlaufe* im Algorithmus *Zusammenhangskomponenten* besucht die Knoten der Zusammenhangskomponente, die den Startknoten v enthält, und fügt diese zur Menge B hinzu. Die Laufzeit des Algorithmus *Zusammenhangskomponenten* ergibt sich damit zu $O(|V| + |E|)$.

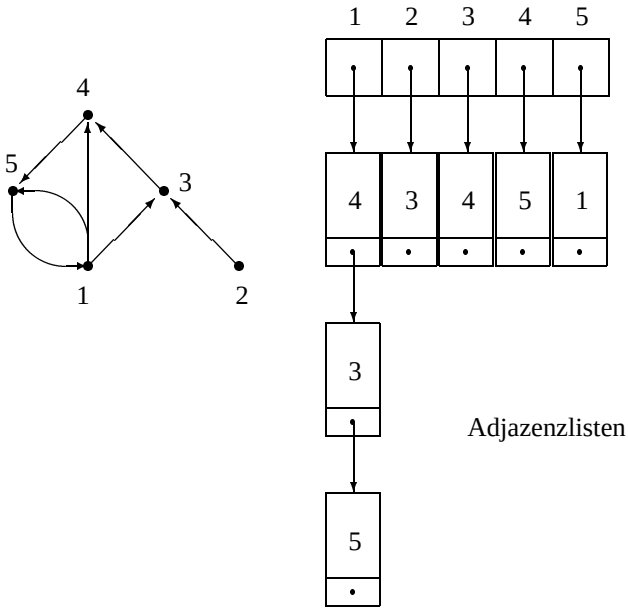
8.3.2 Strukturinformation durch Tiefensuche

Um beim systematischen Durchlaufen eines Graphen mehr über dessen Struktur zu erfahren, wollen wir dieses nun näher in Augenschein nehmen. Betrachten wir zunächst anhand eines Beispiels den Unterschied, der sich ergibt, wenn wir zum einen die noch unbenutzten Kanten als Stapel (*last in first out*), zum anderen als Schlange (*first in first out*) verwalten. Abbildung 8.11 (a) zeigt einen Digraphen und eine Adjazenzlistenrepräsentation; Abbildung 8.11 (b) und 8.11 (c) zeigen die Entwicklung von R als Stapel und als Schlange.

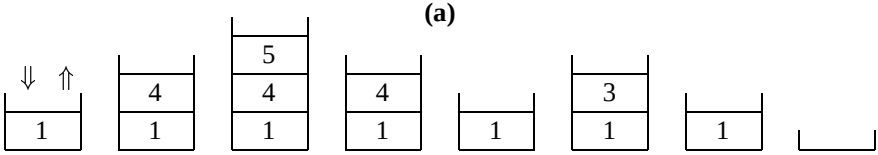
Ist R als Stapel realisiert, so trifft man die Knoten in der Reihenfolge 1, 4, 5, 3 erstmals an; Knoten 2 ist von Knoten 1 aus nicht erreichbar. Ist dagegen R als Schlange realisiert, so ergibt sich die Reihenfolge 1, 4, 3, 5.

Bei Verwendung eines Stapels für R reden wir von *Tiefensuche* (englisch: *depth first search; DFS*), bei einer Schlange von *Breitensuche* (englisch: *breadth first search; BFS*). Die Tiefensuche bietet sich oft für Zusammenhangsprobleme an, die Breitensuche dagegen für Distanzprobleme, wie wir später noch sehen werden. Für manche Algorithmen, in denen es um Aussagen über die Struktur des gegebenen Graphen geht, ist die Tiefensuche von besonderer Bedeutung. Dabei betrachtet man nicht nur die Reihenfolge, in der man Knoten erstmals antrifft, sondern beispielsweise auch die Reihenfolge, in der man Knoten vom Stapel R wieder entfernt. In unserem Beispiel ist dies die Reihenfolge 5, 4, 3, 1. Die relative Position eines Knotens in der Reihenfolge, in der die Knoten auf den Stapel R abgelegt worden sind, nennen wir den *depth-first-begin-Index* (DFBI) eines Knotens. Im Beispiel der Abbildung 8.11 sind die DFBIndizes der Knoten 1, 4, 5 und 3 gerade 1, 2, 3 und 4. Entsprechend bezeichen wir als *depth-first-end-Index* (DFEI) eines Knotens seine relative Position in der Reihenfolge, in der die Knoten vom Stapel R entfernt werden. Im Beispiel der Abbildung 8.11 sind also die DFEIndizes der Knoten 5, 4, 3 und 1 gerade 1, 2, 3 und 4.

Formuliert man die Prozedur für das Durchlaufen eines Graphen ab einem Startknoten b rekursiv, anstatt explizit einen Stapel für R zu benutzen, so entspricht der DFB-Index gerade einer beim Prozeduraufruf vergebenen laufenden Nummer, der DFEIndex einer beim Beenden des Prozeduraufrufs vergebenen Nummer. Wenden wir die bei Bäumen übliche Terminologie (vgl. Kapitel 5) auf den Baum der rekursiven Aufrufe

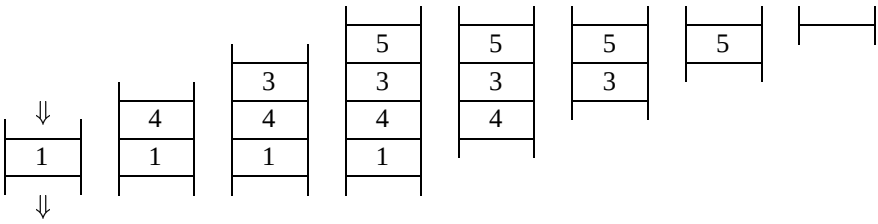


(a)



Durchlaufe mit Stapel *R* ab Startknoten 1

(b)



Durchlaufe mit Schlange *R* ab Startknoten 1

(c)

Abbildung 8.11

an, so ist der DFBIndex gerade die Knotennummer in *Hauptreihenfolge* (*preorder*), der DFEIndex diejenige in *Nebenreihenfolge* (*postorder*).

Wir unterscheiden außerdem bei einem Digraphen die Pfeile nach der Rolle, die sie bei einer Tiefensuche spielen. Dazu teilen wir die Menge aller Pfeile in vier Klassen ein. Die Pfeile, denen die Tiefensuche folgt, die also als unbenutzte Pfeile gewählt werden, heißen *Baumpfeile*; die Menge *BP* der Baumpfeile bildet den *Tiefensuchbaum* (DFS-Baum) vom Startknoten der Tiefensuche aus. Im Beispiel der Abbildung 8.11 ist $BP = \{(1,4),(4,5),(1,3)\}$; die Pfeile in *BP* können an den obersten beiden Elementen des Stapels *R* abgelesen werden, wenn ein neuer Knoten auf den Stapel abgelegt wird. Pfeile, die zu einem bereits erreichten Nachfolgerknoten im DFS-Baum führen, heißen *Vorwärtspfeile*. Jeder Pfeil in der Menge *VP* der Vorwärtspfeile gehört zur transitiven Hülle der Baumpfeile und kürzt einen Weg der Länge mindestens 2 im DFS-Baum ab. Im Beispiel der Abbildung 8.11 ist bei einer Tiefensuche ab Knoten 1 gerade der Pfeil $(1,5)$ ein Vorwärtspfeil. *Rückwärtspfeile* sind all diejenigen Pfeile, die von einem Knoten im DFS-Baum zu einem Vorgänger dieses Knotens im DFS-Baum weisen. Jeder Pfeil in der Menge *RP* der Rückwärtspfeile bildet also mit dem DFS-Baum einen Zyklus. Im Beispiel der Abbildung 8.11 ist der Pfeil $(5,1)$ der einzige Rückwärtspfeil für die Tiefensuche ab Knoten 1. Alle anderen Pfeile heißen *Seitwärtspfeile*; *SP* ist die Menge aller Seitwärtspfeile.

Die folgende rekursiv formulierte Prozedur für die Tiefensuche illustriert die Berechnung der Knotenindizes und die Klassifikation der Pfeile mit Hilfe eines kleinen Programmstücks:

```
procedure DFS für G ab Knoten v, kommend von w;
begin
  if v ∉ B
    then {v noch nicht besucht}
      begin
        B := B ∪ {v};
        BP := BP ∪ {(w,v)};
        erhöhe dfbi um 1; {aktueller DFBIndex}
        DFBI[v] := dfbi;
        for all (v,v') ∈ E do
          DFS für G ab v', kommend von v;
        erhöhe dfei um 1; {aktueller DFEIndex}
        DFEI[v] := dfei
      end
    else {v bereits besucht : klassifiziere Pfeil}
      begin
        if w →*_BP v
          then VP := VP ∪ {(w,v)}
          else if v →*_BP w
            then RP := RP ∪ {(w,v)}
            else SP := SP ∪ {(w,v)}
      end
end {DFS}
```

begin
  $B := \emptyset$;
  dfbi := *dfei* := 0;
  $BP := VP := RP := SP := \emptyset$;
  DFS für G ab v, kommend von nirgends
end

Wir haben noch nicht klargestellt, wie man denn die Bedingungen $w \longrightarrow^*_{BP} v$ und $v \longrightarrow^*_{BP} w$ für das Klassifizieren eines Pfeils als Vorwärtspfeil, Rückwärtspfeil oder Seitwärtspfeil effizient überprüfen kann. Hier helfen uns der DFBIndex und der DFEIndex. Von einem Knoten *w* kommt man im Tiefensuchbaum genau dann zu einem Knoten *v*, wenn der Aufruf der Prozedur *DFS* für *w* vor dem Aufruf von *DFS* für *v* liegt und *DFS* für *v* früher abgeschlossen ist als für *w*. Anders ausgedrückt heißt das, daß $w \longrightarrow^*_{BP} v$ genau dann gilt, wenn $DFBI[w] \leq DFBI[v]$ und $DFEI[w] \geq DFEI[v]$ gelten. Ein Pfeil (w,v) ist genau dann ein Baumpfeil oder ein Vorwärtspfeil, wenn $DFBI[w] \leq DFBI[v]$ gilt. Andernfalls ist ein Pfeil ein Rückwärts- oder Seitwärtspfeil. Damit ergibt sich für die Tiefensuche eine Laufzeit von $O(|V| + |E|)$.

Bei ungerichteten Graphen sind die Verhältnisse einfacher. Zunächst kann es keine Seitwärtskanten geben, weil eine Tiefensuche einer solchen Kante folgen würde. Natürlich bilden die Baumkanten einen Baum der durch die Tiefensuche erreichten Knoten. Alle anderen Kanten werden durch die Tiefensuche zu Rückwärtskanten. Mit diesen Überlegungen genügt es also, bei der Tiefensuche für jeden Knoten bzw. Pfeil eine konstante Anzahl von Schritten aufzuwenden.

Wir wollen im folgenden Abschnitt ein Beispiel für die Anwendung der Tiefensuche betrachten; weitere Beispiele findet man etwa in [121].

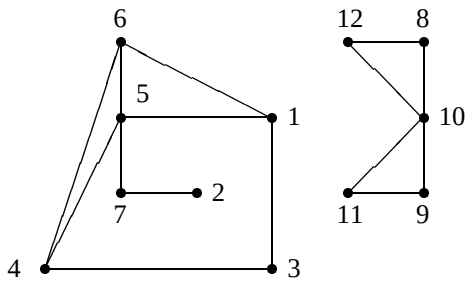
8.4 Zusammenhangskomponenten

Die Bestimmung einfacher Zusammenhangskomponenten ungerichteter Graphen haben wir im letzten Abschnitt, beim Durchlaufen von Graphen, bereits behandelt. Bei der Definition des Zusammenhangs in gerichteten Graphen ist es sinnvoll, die Richtung von Pfeilen zu berücksichtigen. So kann man sich etwa fragen, ob man in einem Netz von Einbahnstraßen einer Stadt überhaupt von jeder Kreuzung zu jeder anderen Kreuzung gelangen kann.

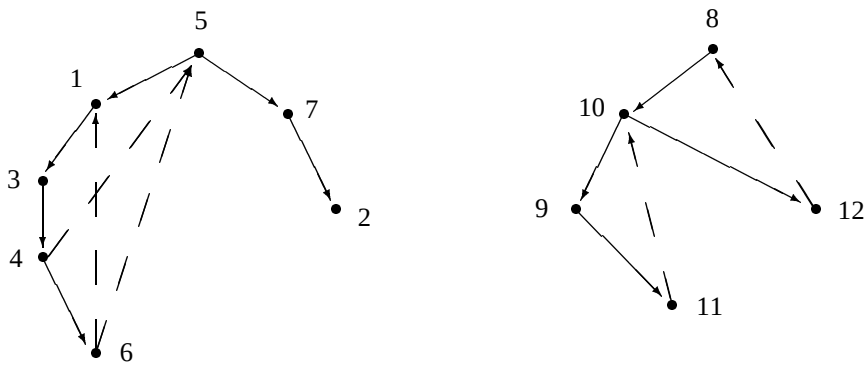
Wir bezeichnen einen Digraphen $G = (V,E)$ als *stark zusammenhängend*, wenn es einen Weg von jedem Knoten zu jedem anderen Knoten im Graphen gibt. Eine *starke Zusammenhangskomponente* (englisch: *strongly connected component; scc*) eines Digraphen *G* ist ein (bezüglich Mengeninklusion) maximaler, stark zusammenhängender Untergraph von *G*. Einen ungerichteten Graphen $G = (V,E)$ nennen wir *zweifach zusammenhängend* (englisch: *biconnected*), wenn nach dem Entfernen eines beliebigen Knotens *v* aus *G* der verbleibende Graph $G - v$ zusammenhängend ist. Eine *zweifache Zusammenhangskomponente* (englisch: *biconnected component; bcc*) eines ungerichteten Graphen ist ein (bezüglich Mengeninklusion) maximaler, zweifach zusammenhän-

gender Untergraph. In einem zweifach zusammenhängenden Graphen kann man einen beliebigen Knoten samt allen inzidenten Kanten entfernen, ohne daß der Graph zerfällt. Ein Knoten *v* ist ein *Schnittpunkt* (englisch: *cut point, articulation point*) eines Graphen *G*, wenn $G - v$ mehr Zusammenhangskomponenten hat als *G*. Durch Wegnahme eines Schnittpunkts zerfällt also eine Zusammenhangskomponente des Graphen.

Betrachten wir als Beispiel den in Abbildung 8.12 (a) gezeigten Graphen.



(a)



Knoten	1	2	3	4	5	6	7	8	9	10	11	12
DFBI	2	7	3	4	1	5	6	8	10	9	11	12
DFEI	4	5	3	2	7	1	6	12	9	11	8	10

(b)

Abbildung 8.12

Er besteht aus zwei einfachen Zusammenhangskomponenten; keine von beiden ist zweifach zusammenhängend. Die Schnittpunkte des Graphen sind die Knoten 5, 7 und

10. Die zweifachen Zusammenhangskomponenten sind die durch die Knotenmengen $\{1,3,4,5,6\}$, $\{2,7\}$, $\{5,7\}$, $\{8,10,12\}$ und $\{9,10,11\}$ induzierten Untergraphen.

8.4.1 Zweifache Zusammenhangskomponenten

Zur Berechnung der zweifachen Zusammenhangskomponenten ermitteln wir die Schnittpunkte eines Graphen mit folgenden Überlegungen. Ein Schnittpunkt ist die ausschließliche Verbindung von wenigstens zwei zweifachen Zusammenhangskomponenten. Wenn also ein Schnittpunkt v Wurzel eines Tiefensuchbaums ist, so hat v im Tiefensuchbaum mehr als einen Sohn, weil die Tiefensuche nicht anders als über v von der einen in die andere zweifache Zusammenhangskomponente gelangen kann. In Abbildung 8.12 (b) ist der mögliche Verlauf einer Tiefensuche, beginnend bei Knoten 5 und bei Knoten 8, mit dem sich ergebenden DFBIndex und DFEIndex gezeigt. Knoten 5 als Schnittpunkt und Wurzel eines Tiefensuchbaums hat einen Sohn für jede einfache Zusammenhangskomponente, die sich durch Entfernen des Knotens 5 ergibt.

Trifft man während der Tiefensuche auf einen Schnittpunkt v, d.h., ist v nicht Wurzel eines Tiefensuchbaums, so muß sich wenigstens eine zweifache Zusammenhangskomponente im Tiefensuchbaum in einem Teilbaum ab v befinden; aus einem solchen Teilbaum heraus darf also keine Kante zu einem Vorgänger von v führen. Anders ausgedrückt: Ist ein Schnittpunkt v nicht Wurzel eines Tiefensuchbaums, dann hat v einen Sohn v', so daß kein Nachfolger von v' im Tiefensuchbaum, inklusive v' selbst, über eine Rückwärtskante mit einem Vorgänger von v verbunden ist. Im Beispiel der Abbildung 8.12 ist Knoten 10 ein solcher Schnittpunkt; von Knoten 9 und Knoten 11 führt keine Rückwärtskante über Knoten 10 hinaus. Das ist auch intuitiv plausibel, weil die Tiefensuche in der anderen der beiden zweifachen Zusammenhangskomponenten begonnen hat, die durch Knoten 10 verbunden sind.

Dies legt nahe, sich während der Tiefensuche für jeden Knoten zu merken, wie weit man über Rückwärtskanten höchstens im DFBIndex zurückgelangen kann. Dies leistet ein für jeden Knoten v während der Tiefensuche zu berechnender Wert $P[v]$, der durch $P[v] := \min(\{DFBI[v]\} \cup \{DFBI[v'] \mid v' \text{ ist Vorgänger von } v \text{ im DFS-Baum und ist mit Rückwärtskante mit Nachfolger von } v \text{ verbunden}\})$ definiert ist. Wenn nun ein Schnittpunkt v nicht Wurzel eines DFS-Baumes ist, dann hat v einen Sohn v' mit $P[v'] \geq DFBI[v]$. Um die Berechnung von $P[v]$ in den rekursiv formulierten Tiefensuchalgorithmus einzubetten, formulieren wir $P[v]$ zunächst noch rekursiv, und zwar als $P[v] := \min(\{DFBI[v]\} \cup \{P[v'] \mid v' \text{ ist Sohn von } v\} \cup \{DFBI[v'] \mid (v,v') \text{ ist Rückwärtskante}\})$. Ein Programmstück, das nach diesem Verfahren die zweifachen Zusammenhangskomponenten zu einem Graphen mittels einer rekursiv formulierten Tiefensuche berechnet, ist dann das folgende:

```
procedure DFSBCC für G ab Knoten v;
begin
  B := B ∪ {v};
  erhöhe dfbi um 1;
  DFBI[v] := dfbi;
  P[v] := dfbi;
  for all (v,v') ∈ E do
```

```
    {beachte, daß (v,v') = (v',v) die Kante identifiziert, daß also in
     der Schleife jede Kante genau einmal bearbeitet wird}
      begin
        lege (v,v') auf Stapel BCC;
          {Stapel BCC speichert begonnene bcc's}
        if v' ∉ B
          then {(v,v') ist eine Baum-Kante}
            begin
              Vater[v'] := v;
              DFSBCC für G ab Knoten v';
              if P[v'] ≥ DFBI[v]
                then {v ist Schnittpunkt oder letzter Knoten dieser
                      Komponente}
                  nimm jede Kante bis inkl. (v,v') vom Stapel BCC und
                    berichte sie als bcc;
              {jetzt ist Sohn v' behandelt}
              P[v] := min(P[v],P[v'])
            end
          else if v' ≠ Vater[v]
            then {(v,v') ist Rückwärtskante}
              P[v] := min(P[v], DFBI[v'])
      end
end {DFSBCC}

begin
  B := ∅; {bereits besuchte Knoten}
  dfbi:= 0;
  BCC := leerer Stapel;
  for all v ∈ V do
    if v ∉ B
      then DFSBCC für G ab Knoten v
end
```

Abbildung 8.13 zeigt die Berechnung der zweifachen Zusammenhangskomponenten mit Hilfe von *DFSBCC* für den in 8.12 (a) gezeigten Graphen ab Knoten 5, wenn die Tiefensuche verläuft wie in 8.12 (b) skizziert. Momentaufnahmen des Stapels *BCC* sind unmittelbar vor und nach jeder Entnahme der Kanten einer zweifachen Zusammenhangskomponente wiedergegeben.

Aus der Effizienz der Tiefensuche und den zusätzlich erforderlichen Operationen mit Stapel *BCC*, auf dem jede Kante des Graphen gerade einmal abgelegt wird, ergibt sich als Laufzeit für die Berechnung der zweifachen Zusammenhangskomponenten eines ungerichteten Graphen $G = (V,E)$ unmittelbar $O(|V| + |E|)$.

Knoten	1	2	3	4	5	6	7	8	9	10	11	12
P	~~2~~ 1	7	~~3~~ 1	~~4~~ 1	1	~~5~~ ~~2~~ 1	6	8	~~10~~ 9	~~9~~ 8	~~11~~ 9	~~12~~ 8

Stapel *BCC*:

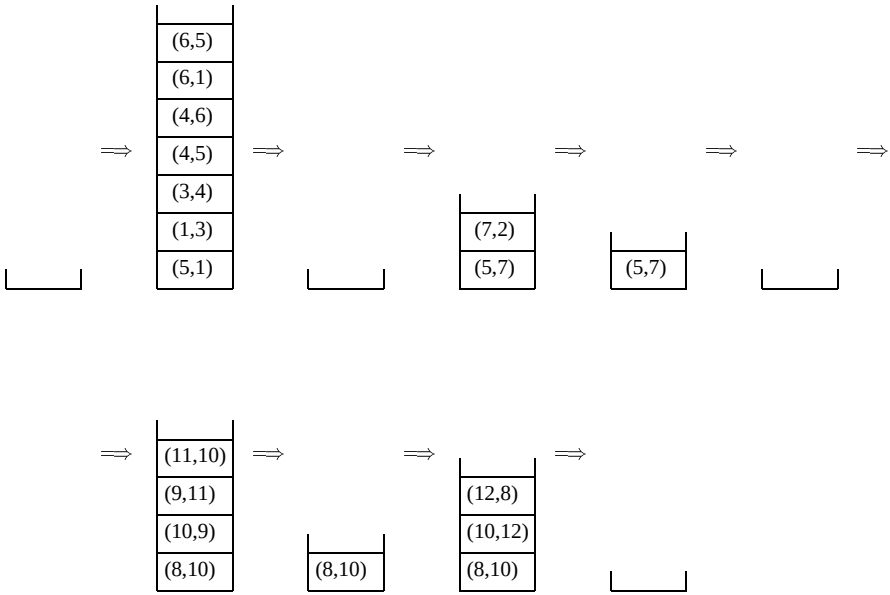
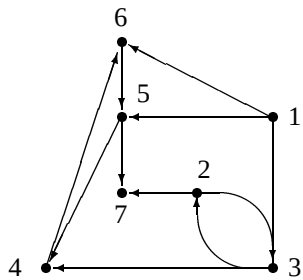


Abbildung 8.13

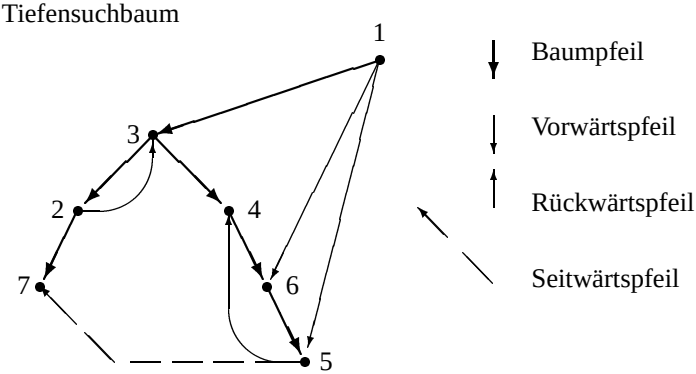
8.4.2 Starke Zusammenhangskomponenten

Betrachten wir nun das Problem, zu einem gegebenen Digraphen die starken Zusammenhangskomponenten zu berechnen. Im Beispiel der Abbildung 8.14 (a) sind dies die durch die vier Knotenmengen {1}, {2,3}, {4,5,6} und {7} induzierten Untergraphen. Abbildung 8.14 (b) zeigt den Verlauf und das Resultat einer beim Knoten 1 beginnenden Tiefensuche. Wir wollen uns nun überlegen, in welcher Reihenfolge die Tiefensuche die Knoten starker Zusammenhangskomponenten komplett besucht hat, also wieder verläßt. Im Beispiel der Abbildung 8.14 ist die erste komplett besuchte starke Zusammenhangskomponente diejenige mit Knotenmenge {7}; kein Pfeil verläßt diese Komponente, und der größte DFEIndex eines Knotens dieser Komponente ist 1. Die nächste durch die Tiefensuche komplett besuchte starke Zusammenhangskomponente ist diejenige mit Knotenmenge {4,5,6}. Der einzige Pfeil, der diese Komponente verläßt, führt zu einem Knoten einer bereits berechneten starken Zusammenhangskomponente (Pfeil (5,7) führt zu Knoten 7).



(a)

Tiefensuchbaum



Tiefensuche

Knoten	1	2	3	4	5	6	7
DFBI	1	3	2	5	7	6	4
DFEI	7	2	6	5	3	4	1

(b)

Abbildung 8.14

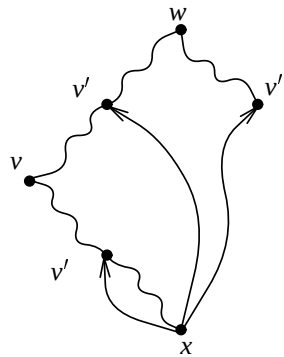


Abbildung 8.15

Natürlich kann eine starke Zusammenhangskomponente bei der Tiefensuche nicht in mehrere Tiefensuchbäume zerfallen, weil ja jeder Knoten der starken Zusammenhangskomponente von jedem anderen aus erreichbar ist. Diejenigen Pfeile einer starken Zusammenhangskomponente, die in einem DFS-Wald Baumpfeile sind, bilden zusammen *einen* DFS-Baum. Die Wurzel des DFS-Baums für eine starke Zusammenhangskomponente nennen wir *Wurzel der Zusammenhangskomponente*. Im Beispiel der Abbildung 8.14 sind die Knoten 7, 4, 3 und 1 Wurzeln von starken Zusammenhangskomponenten. Wir wollen starke Zusammenhangskomponenten berechnen, indem wir ihre Wurzeln in einem Tiefensuchwald bestimmen. Weil der DFEIndex der Wurzel eines Teilbaums der größte DFEIndex der Knoten dieses Teilbaums ist, betrachten wir die Wurzeln von starken Zusammenhangskomponenten in der Reihenfolge aufsteigender DFEIndizes. Seien dies die Wurzeln $w_1, w_2, \ldots, w_k$. Haben wir eine Wurzel w_i einer starken Zusammenhangskomponente in einem Tiefensuchbaum gefunden, so gehören zu dieser Komponente all diejenigen Knoten, die im Teilbaum des Tiefensuchbaums mit Wurzel w_i stehen, aber nicht auch in bereits identifizierten Teilbäumen mit Wurzeln $w_1, \ldots, w_{i-1}$. Im Beispiel der Abbildung 8.14 sind dies etwa die Knoten des Teilbaums mit Wurzel 3, die nicht auch im Teilbaum mit Wurzel 4 oder im Teilbaum mit Wurzel 7 liegen, also die Knoten 2 und 3.

Während der Tiefensuche berechnen wir für jeden Knoten v einen Wert $Q[v]$, der uns darüber Auskunft gibt, ob v Wurzel einer starken Zusammenhangskomponente ist. Dazu definieren wir $Q[v]$ als $Q[v] := \min(\{DFBI[v]\} \cup \{DFBI[v'] |$ für einen Nachfolger x von v ist $(x,v') \in RP \cup SP$, und die Wurzel w der starken Zusammenhangskomponente von v' ist Vorgänger von $v\}$). Die Begriffe Nachfolger und Vorgänger beziehen sich dabei auf den betrachteten Tiefensuchbaum. Dann ist die Wurzel einer starken Zusammenhangskomponente der Knoten v mit $Q[v] = DFBI[v]$. Abbildung 8.15 illustriert, auf welche Arten ein Zyklus von Knoten v über die Knoten x, v' und w zum Knoten v möglich ist. Man beachte, daß dabei ein Knoten v', der Nachfolger von v ist, wegen $DFBI[v'] > DFBI[v]$ nichts zu $Q[v]$ beiträgt. Zur Einbettung der Berechnung von Q in die rekursiv formulierte Tiefensuche läßt sich Q auch rekursiv formulieren:

$Q[v] := \min(\{DFBI[v]\} \cup \{Q[v'] \mid v' \text{ ist Sohn von } v\} \cup \{DFBI[v'] \mid (v,v') \in RP \cup SP, \text{ und die Wurzel } w \text{ der starken Zusammenhangskomponente von } v' \text{ ist Vorgänger von } v\})$.

Das folgende Programmstück berechnet zu einem gegebenen Digraphen die starken Zusammenhangskomponenten nach diesem Verfahren, wobei die Vereinbarung eines Feldes

```
var gestapelt: array [knotentyp] of boolean
```

vorausgesetzt wird:

```
procedure DFSSCC für G ab Knoten v;
begin
  B := B ∪ {v};   {Menge bereits besuchter Knoten}
  erhöhe dfbi um 1;
  DFBI[v] := dfbi;
  Q[v] := dfbi;
  lege v auf Stapel SCC;
    {Stapel SCC speichert Knoten, die noch keiner scc zugeordnet sind}
  gestapelt[v] := true;
  for all (v,v') ∈ E do
    if v' ∉ B
      then {v' noch nicht besucht}
        begin
          DFSSCC für G ab Knoten v';
          Q[v] := min(Q[v], Q[v'])
        end
      else if DFBI[v'] < DFBI[v] and gestapelt [v']
        then Q[v] := min(Q[v], DFBI[v']);
  if Q[v] = DFBI[v]    {Wurzel einer scc}
    then nimm jeden Knoten u bis incl. v vom Stapel SCC und berichte
      scc, und setze jeweils gestapelt[u] := false
end; {DFSSCC}

begin
  B := ∅;    {anfangs noch kein Knoten besucht}
  dfbi := 0;
  SCC := leerer Stapel;
  for all v ∈ V do gestapelt[v] := false;
  for all v ∈ V do
    if v ∉ B
      then DFSSCC für G ab Knoten v
end
```

Abbildung 8.16 zeigt die Berechnung der starken Zusammenhangskomponenten mit Hilfe von *DFSSCC* für den in Abbildung 8.14 (a) gezeigten Graphen ab Knoten 1, wenn die Tiefensuche verläuft wie in Abbildung 8.14 (b) skizziert. Momentaufnahmen des Stapels *SCC* sind unmittelbar vor und nach jeder Entnahme der Pfeile einer starken Zusammenhangskomponente wiedergegeben.

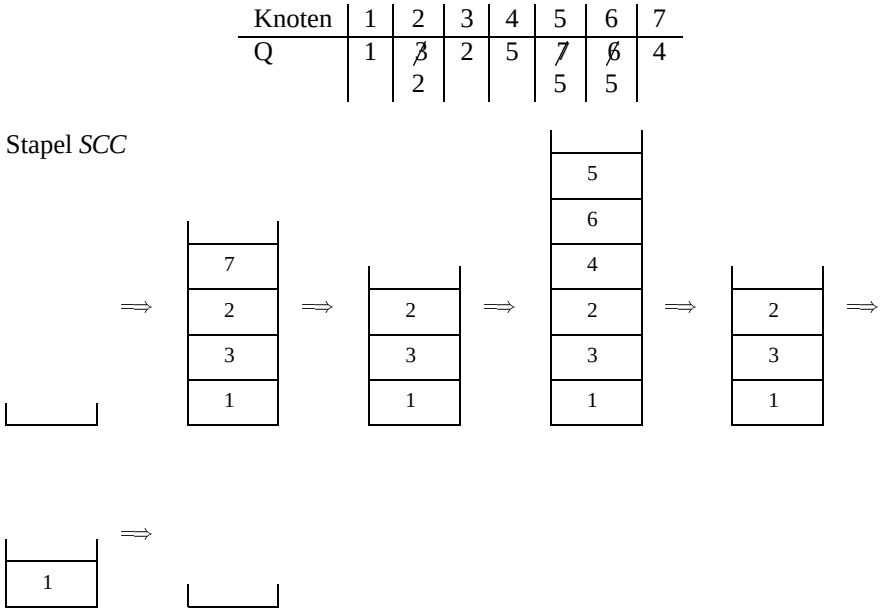


Abbildung 8.16

Aus der Effizienz der Tiefensuche und der zusätzlich erforderlichen Operationen mit Stapel *SCC*, auf dem jeder Knoten des Graphen gerade einmal abgelegt wird, sowie der Überprüfung, ob ein Knoten gestapelt ist, die mit Hilfe des Feldes *gestapelt* in konstanter Zeit stattfindet, ergibt sich für die Berechnung der starken Zusammenhangskomponenten eines Digraphen $G = (V,E)$ als Laufzeit unmittelbar $O(|V| + |E|)$.

Interpretiert man die Menge der Pfeile als eine Relation über der Menge der Knoten, so definieren die starken Zusammenhangskomponenten gerade die Äquivalenzklassen der Relation. Wenn man den gegebenen Digraphen verdichtet, indem man jede starke Zusammenhangskomponente durch einen Knoten ersetzt und Pfeile zwischen Knoten derselben Zusammenhangskomponente wegläßt, so stellt der entstehende zyklenfreie, verdichtete Digraph gerade die partielle Ordnung über den Äquivalenzklassen der Relation dar. Für den in Abbildung 8.14 angegebenen Beispielgraphen zeigt Abbildung 8.17 den verdichteten Graphen.

Genauer: Für einen gegebenen Digraphen $G = (V,E)$ mit Knotenmengen $V_1, \ldots, V_k$ für k starke Zusammenhangskomponenten heißt der Digraph $G' = (V',E')$ mit $V' = \{1, \ldots, k\}$ und $E' = \{(i,j) | \exists\ v \in V_i, v' \in V_j, (v,v') \in E\}$ *verdichteter Digraph*. G' ist azyklisch. Für die Graphen mit wenigen starken Zusammenhangskomponenten führt der Umweg über den verdichteten Digraphen zu einem schnelleren Algorithmus zur Berechnung der reflexiven, transitiven Hülle, gemäß folgender Beobachtung. Ein Pfeil (i,j) in der reflexiven, transitiven Hülle des verdichteten Digraphen impliziert Pfeile von allen Knoten der starken Zusammenhangskomponente V_i zu allen Knoten der starken Zusammenhangskomponente V_j in der reflexiven, transitiven Hülle des gegebenen

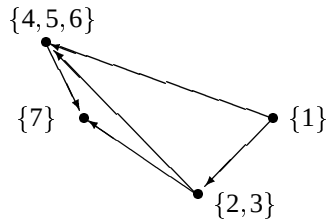


Abbildung 8.17

Graphen. Außerdem gibt es in der reflexiven, transitiven Hülle des gegebenen Graphen G Pfeile zwischen allen Knoten innerhalb jeder starken Zusammenhangskomponente. Damit läßt sich die reflexive, transitive Hülle eines gegebenen Digraphen $G = (V,E)$ wie folgt berechnen:

1. Berechne die starken Zusammenhangskomponenten $V_1, \ldots, V_k$.
2. Berechne den verdichteten Digraphen $G' = (V', E')$.
3. Berechne die reflexive, transitive Hülle $G'^* = (V', E'^*)$ von G'.
4. Berechne die reflexive, transitive Hülle $G^* = (V, E^*)$ von G.

Die ersten beiden Teile dieses Algorithmus benötigen jeweils $O(|V| + |E|)$ Schritte; Teil 3 kann gemäß Abschnitt 8.2 schlimmstenfalls in $O(k^3)$ Schritten gelöst werden, und Teil 4 benötigt offenbar höchstens $O(|E^*|)$ Schritte. Damit kann für einen gegebenen Digraphen $G = (V,E)$ mit k starken Zusammenhangskomponenten die reflexive, transitive Hülle in Zeit $O(|V| + |E^*| + k^3)$ berechnet werden.

8.5 Kürzeste Wege

Bei der Modellierung realer Probleme durch Graphen ist es oft wichtig, nicht nur das Vorhandensein oder Fehlen von Knoten und Kanten zu unterscheiden. Vielmehr müssen Knoten und Kanten Eigenschaften zugeordnet werden, die für die Lösung des Problems wesentlich sind. Beispielsweise haben Kanalisationsrohre eine gewisse maximale Transportkapazität, Arbeiten an einem Haus eine minimale, eine maximale und eine erwartete Dauer und Bahnstrecken eine Länge und (je nach Tarif) einen Preis. In diesem Abschnitt interessieren wir uns für kostengünstigste Wege in Graphen, wenn jeder Kante/jedem Pfeil ein Kostenwert zugeordnet ist. Meist redet man dabei, stellvertretend für allerlei Interpretationen der Kosten, von der *Länge* von Kanten/Pfeilen; ein kostengünstigster Weg ist dann ein *kürzester* Weg. Wir wollen auch zulassen, daß Kanten/Pfeile eine negative Länge haben. Dann können wir Gewinne und Verluste modellieren, aber auch längste Wege durch kürzeste Wege ausdrücken, nämlich mit negativ gemachten Längen der einzelnen Kanten/Pfeile.

Ein ungerichteter Graph $G = (V,E)$ mit einer reellwertigen Bewertungsfunktion $c : E \to \mathbb{R}$ (englisch: cost) heißt *bewerteter Graph*. Für eine Kante $e \in E$ heißt $c(e)$ *Bewertung* (*Länge, Gewicht, Kosten*) der Kante e. Die Länge $c(G)$ des Graphen G ist die Summe der Längen aller Kanten, also $c(G) = \sum_{e \in E} c(e)$. Damit ist für einen Weg $p = (v_0, v_1, \ldots, v_k)$ die Länge dieses Wegs gerade $c(p) = \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$. Für Graphen ohne Bewertung, die wir bisher betrachtet haben, haben wir die Länge von Wegen so definiert, als sei $c \equiv 1$. Die *Entfernung d* (Distanz; englisch: *distance*) von einem Knoten v zu einem Knoten v' ist definiert als $d(v,v') = min\{c(p) \mid p \text{ ist Weg von } v \text{ nach } v'\}$, falls es überhaupt einen Weg von v nach v' gibt; sonst ist $d(v,v') = \infty$. Ein Weg p zwischen v und v' mit $c(p) = d(v,v')$ heißt *kürzester Weg* (englisch: *shortest path*) zwischen v und v'; wir bezeichnen ihn mit $sp(v,v')$.

Ganz entsprechend heißt ein Digraph $G = (V,E)$ mit Bewertungsfunktion $c : E \to \mathbb{R}$ *bewerteter Digraph*; wenn er keine Knoten ohne inzidente Pfeile hat, heißt er *Netzwerk*. Die übrigen Begriffe sind entsprechend definiert.

Ist die Länge jeder Kante nicht negativ, also $c : E \to \mathbb{R}_0^+$, so heißt $G = (V,E)$ mit c *Distanzgraph* (in Abschnitt 6.1.1 haben wir Distanzgraphen betrachtet und die Kosten einer Kante entsprechend als Länge bezeichnet). Die Berechnung kürzester Wege in Distanzgraphen ist einfacher und kann schneller ausgeführt werden als in beliebigen bewerteten Graphen, weil sich in Distanzgraphen Wege durch Hinzunahme weiterer Kanten nicht verkürzen können. Algorithmen für das Finden kürzester Wege zwischen gegebenen Knoten in ungerichteten Distanzgraphen operieren nach dem Grundmuster der Breitensuche. Als Folge davon werden beim Berechnen eines kürzesten Weges von einem gegebenen *Anfangsknoten* zu einem gegebenen *Endknoten* auch kürzeste Wege vom Anfangsknoten zu vielen anderen Knoten des Graphen ermittelt. Das Verfahren zur Berechnung eines kürzesten Weges zwischen Anfangs- und Endknoten (*one-to-one shortest path, single pair shortest path*) unterscheidet sich vom Verfahren zur Berechnung der kürzesten Wege von einem Anfangsknoten zu allen anderen Knoten des Graphen (*one-to-all shortest paths, single source shortest paths*) nur durch das Abbruchkriterium. Im schlimmsten Fall haben beide Verfahren dieselbe Laufzeit; wir werden daher im folgenden das Problem kürzester Wege von einem zu allen anderen Knoten zunächst für Distanzgraphen und dann für beliebige bewertete Graphen betrachten. Dem Problem, zu jedem Paar von Knoten einen kürzesten Weg zu finden, werden wir uns am Schluß dieses Abschnitts zuwenden.

8.5.1 Kürzeste Wege in Distanzgraphen

Wir betrachten das Problem, zu einem gegebenen Distanzgraphen $G = (V,E)$ mit $c : E \to \mathbb{R}_0^+$ je einen kürzesten Weg von einem gegebenen Anfangsknoten s (englisch: source) zu jedem anderen Knoten des Graphen zu finden. Abbildung 8.18 zeigt ein Beispiel für einen ungerichteten Distanzgraphen; neben jeder Kante ist deren Länge vermerkt. Man sieht leicht, daß ein kürzester Weg, beispielsweise von Knoten 1 zu Knoten 8, gefunden werden kann, indem man eine Art äquidistanter Welle um den Knoten 1 solange wachsen läßt, bis sie den Knoten 8 erreicht.

Wichtig für das dieser Idee zugrunde liegende Verlängern eines Wegs durch Hinzunahme einer weiteren Kante ist das *Optimalitätsprinzip*: Für jeden kürzesten Weg

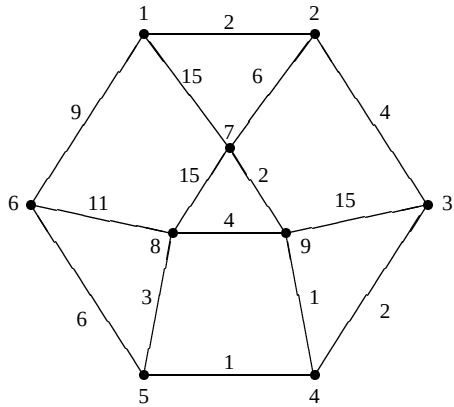


Abbildung 8.18

$p = (v_0, v_1, \ldots, v_k)$ von v_0 nach v_k ist jeder Teilweg $p' = (v_i, \ldots, v_j)$, $0 \leq i < j \leq k$, ein kürzester Weg von v_i nach v_j. Wäre dies nicht so, gäbe es also einen kürzeren Weg p'' von v_i nach v_j, so könnte auch in p der Teilweg p' durch p'' ersetzt werden, und der entstehende Weg von v_0 nach v_k wäre kürzer als p; dies ist aber ein Widerspruch zu der Annahme, daß p ein kürzester Weg von v_0 nach v_k ist. Damit können wir länger werdende kürzeste Wege durch Hinzunahme einzelner Kanten zu bereits bekannten kürzesten Wegen mit folgender *Invariante* berechnen:

1. Für alle kürzesten Wege $sp(s,v)$ und Kanten (v,v') gilt:
   $c(sp(s,v)) + c((v,v')) \geq c(sp(s,v'))$.
2. Für wenigstens einen kürzesten Weg $sp(s,v)$ und eine Kante (v,v') gilt:
   $c(sp(s,v)) + c((v,v')) = c(sp(s,v'))$.

Abbildung 8.19 zeigt, wie die entsprechende Berechnung kürzester Wege realisiert werden kann. Jeder Knoten gehört zu einer von drei Klassen: Er ist entweder *gewählter Knoten*, *Randknoten* oder *unerreichter Knoten*. Zu jedem gewählten Knoten ist ein kürzester Weg vom Anfangsknoten s bereits bekannt; zu jedem Randknoten kennt man einen Weg von s, und für jeden unerreichten Knoten kennt man noch keinen solchen Weg.

Wir merken uns für jeden Knoten v die bisher berechnete, vorläufige *Entfernung* zum Anfangsknoten s, den *Vorgänger* von v auf dem bisher berechneten, vorläufig kürzesten Weg von s nach v und eine Markierung, die darüber Auskunft gibt, ob der Knoten bereits *gewählt* ist oder nicht. Außerdem speichern wir die Menge R der Randknoten. Dann realisiert der folgende, von Dijkstra [35] bereits 1959 vorgeschlagene Algorithmus die Berechnung kürzester Wege von einem Knoten zu allen anderen in der skizzierten Weise:

Algorithmus *kürzeste Wege in* $G = (V,E)$ *mit* $c : E \to \mathbb{R}_0^+$ *von einem Knoten* $s \in V$ *zu allen anderen*

1. {*Initialisierung*:}
1.1 {*anfangs sind alle Knoten außer* s *unerreicht*:}

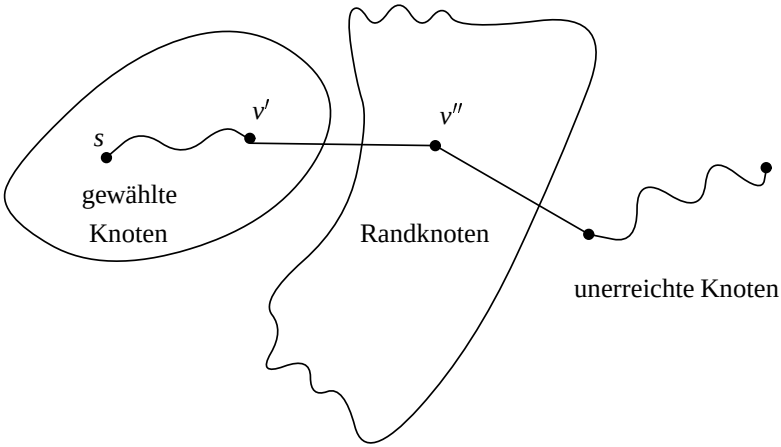


Abbildung 8.19

```
      for all v ∈ V − {s} do
        begin
          v.Vorgänger := undefiniert;
          v.Entfernung := ∞;
          v.gewählt := false
        end;
1.2   {s ist gewählt:}
      s.Vorgänger := s;
      s.Entfernung := 0;
      s.gewählt := true;
1.3   {alle zu s adjazenten Knoten gehören zum Rand R:}
      R := ∅;
      ergänze R bei s;
2.    {berechne Wege ab s:}
      while R ≠ ∅ do
        begin
          {wähle nächstgelegenen Randknoten:}
2.1       wähle v ∈ R mit v.Entfernung minimal, und entferne v aus R;
2.2       v.gewählt := true;
2.3       ergänze R bei v
        end
end {kürzeste Wege}
```

Das Ergänzen des Randes R bei einem gewählten Knoten v besteht in der Hinzunahme aller unerreichten Knoten zum Rand R und im Anpassen der möglicherweise kürzer gewordenen Entfernungen zu Randknoten:

```
ergänze Rand R bei v:
  for all (v,v') ∈ E do
    if not v'.gewählt and (v.Entfernung +c((v,v')) < v'.Entfernung)
      then {v' ist (kürzer) über v erreichbar}
        begin
          v'.Vorgänger := v;
          v'.Entfernung := v.Entfernung +c((v,v'));
          vermerke v' in R
        end
```

Abbildung 8.20 zeigt, wie für den in Abbildung 8.18 gezeigten Graphen eine Suche nach allen kürzesten Wegen vom Knoten 1 aus nach diesem Algorithmus verläuft. Der jeweils gewählte Knoten und die aktuelle Menge R der Randknoten sind im Zeitablauf angegeben. Man sieht, wie sich vorläufige Distanzen von Randknoten ändern können, etwa am Beispiel des Knotens 8. Wenn die von Knoten 1 ausgesandte äquidistante Welle mit aktueller Distanz 8 den Knoten 7 erreicht hat, wird Knoten 8 als mit 7 adjazenter Knoten zu einem Randknoten; seine vorläufige Distanz zu Knoten 1, die er über den Vorgängerknoten 7 realisiert, beträgt 23. Nach der Wahl des Knotens 6 verringert sich diese Distanz auf 20, nach Wahl von Knoten 9 auf 13 und nach Wahl von Knoten 5 schließlich auf 12. Anhand der Liste der gewählten Knoten und der dazugehörigen Vorgängerinformation läßt sich ein kürzester Weg von Knoten 1 zu jedem anderen Knoten rekonstruieren. Für Knoten 8 beispielsweise findet man den Vorgänger 5, für 5 den Vorgänger 4, für 4 den Knoten 3, für 3 den Knoten 2 und für 2 schließlich den Knoten 1 als Vorgänger.

Wir haben in der Illustration des Verlaufs der Berechnung kürzester Wege keine bestimmte Implementierung für die Menge der Randknoten unterstellt; schon dieses Beispiel macht aber klar, daß die geeignete Verwaltung der Randknotenmenge für die Effizienz des Verfahrens wesentlich ist. Rekapitulieren wir vor der Diskussion der verschiedenen Möglichkeiten hierfür die auf dem Rand auszuführenden Operationen:

(a) Rand R als leer initialisieren;
(b) prüfen, ob Rand R leer ist;
(c) wählen und entfernen des Knotens mit minimaler Entfernung aus dem Rand R;
(d) neuen oder geänderten Eintrag im Rand R vermerken.

Wir betrachten die folgenden drei Implementierungsvorschläge, mit denen die angegebenen Operationen unterschiedlich gut unterstützt werden.

Keine explizite Speicherung des Randes

Der Rand wird nicht explizit gespeichert, sondern genauso behandelt wie die unerreichten Knoten. Für jeden Knoten ist also nur an der *gewählt*-Markierung erkennbar, ob er gewählt ist oder nicht. Mit der angegebenen Initialisierung der Entfernungswerte aller Knoten führt dies zum richtigen Ergebnis; diese Tatsache haben wir bereits beim Ergänzen des Randes ausgenutzt. Die angegebenen Operationen können dann wie folgt realisiert werden:

(a) diese Operation ist implizit, kann also entfallen;
(b) für alle Knoten v wird **not** *v.gewählt* überprüft;

Knoten $\widehat{=}$ (Nr., Entfernung, Vorgänger)	
gewählt	Randknoten
(1,0,1)	(2,2,1), (6,9,1), (7,15,1)
(2,2,1)	(6,9,1), (7,8,2), (3,6,2)
(3,6,2)	(6,9,1), (7,8,2), (4,8,3), (9,21,3)
(7,8,2)	(6,9,1), (4,8,3), (9,10,7), (8,23,7)
(4,8,3)	(6,9,1), (8,23,7), (9,9,4), (5,9,4)
(6,9,1)	(9,9,4), (5,9,4), (8,20,6)
(9,9,4)	(5,9,4), (8,13,9)
(5,9,4)	(8,12,5)
(8,12,5)	$\emptyset$

Abbildung 8.20

(c) unter allen Knoten v mit **not** *v.gewählt* wird das Minimum von *v.Entfernung* berechnet; das Entfernen des Minimums aus dem Rand ist implizit, kann also entfallen;
(d) diese Operation ist implizit, kann also entfallen.

Damit benötigt Schritt 1 des Algorithmus eine Laufzeit von $O(|V|)$, und in Schritt 2 werden $\Theta(|V|)$ Schleifendurchläufe mit Laufzeit jeweils $O(|V|)$ ausgeführt. Die Gesamtlaufzeit ist also $O(|V|^2)$. Diese von Dijkstra [35] vorgeschlagene Implementierung ist sehr effizient für Graphen mit vielen Kanten. Bei $\Omega(|V|^2)$ Kanten ist die Laufzeit linear in der Größe der Eingabe, also größenordnungsmäßig optimal. Für Graphen mit weniger Kanten (*dünnere* Graphen) lohnt es sich, über andere Implementierungen des Randes nachzudenken.

Verwaltung der Randknoten in einem Heap

Da die für den Rand R benötigten Operationen (a) bis (c) gerade Heap-Operationen sind, können diese in konstanter Zeit für Operationen (a) und (b) und in logarithmischer Zeit für Operation (c) ausgeführt werden. Ist der in Operation (d) im Rand zu vermerkende Eintrag neu, so kann er gerade als Einfügeoperation im Heap in logarithmischer Zeit realisiert werden. Wenn der Heap — wie üblich — die Suche nach einem beliebigen Eintrag und das Löschen dieses Eintrags nicht unterstützt, so kann ein Knoten mit geänderter Entfernung einfach zusätzlich in den Heap eingefügt werden. Dann ist ein und derselbe Knoten unter Umständen mit mehreren verschiedenen Entfernungen im Rand gespeichert. Der Algorithmus arbeitet trotzdem korrekt, wenn man für jeden Knoten nur die erste Entnahme dieses Knotens aus dem Heap beachtet und alle weiteren ignoriert.

Da bei dieser Implementierung für jede Kante gerade ein Eintrag in den Heap vorgenommen wird, enthält dieser nie mehr als $O(|E|)$ Knoten. Weil mit $|E| \leq |V|^2$ auch $\log|E| \leq 2 \cdot \log|V|$ und damit $O(\log|E|) = O(\log|V|)$ gilt, kostet sowohl das Eintragen aller Knoten in den Heap als auch das Entfernen aller Knoten aus dem Heap jeweils $O(|E|\log|V|)$ Rechenschritte. Das ist sehr effizient für dünne Graphen, aber schlechter als Dijkstras einfache Implementierung für sehr dichte Graphen, also insbesondere

wenn $|E| = \Omega(|V|^2)$. Eine bessere Laufzeit erhält man mit einer anderen Heapstruktur, die sich für verschiedene Graphenprobleme sehr gut eignet.

Verwaltung der Randknoten in einem Fibonacci-Heap

Fibonacci-Heaps [60] (vgl. Kapitel 6) unterstützen die Operationen (a), (b) und (d) in konstanter amortisierter Laufzeit; lediglich Operation (c) benötigt logarithmische Zeit. Operation (d) wird für unerreichte Knoten als Einfügeoperation im Fibonacci-Heap realisiert und für Randknoten, deren Entfernung sich vermindert, als Decrease-Key-Operation. Die maximale Größe des Fibonacci-Heaps ist somit $O(|V|)$. Die $|E|$ Neueinträge und Änderungen von Knoten im Fibonacci-Heap können in Zeit $O(|E|)$ ausgeführt werden. Mit der $(|V|-1)$-maligen Ausführung der Operation (c), die jeweils in Zeit $O(\log|V|)$ erledigt werden kann, ergibt sich eine Gesamtlaufzeit von $O(|E| + |V|\log|V|)$ für das Finden der kürzesten Wege von einem zu allen anderen Knoten in einem Distanzgraphen, für ungerichtete ebenso wie für gerichtete Graphen [60].

Wählt man diese Implementierung, so kann man den Algorithmus *kürzeste Wege* spezieller als Prozedur *shortestpath* wie in Abschnitt 6.1.1 formulieren. Dort ist $v.Entfernung$ mit $d(v)$ bezeichnet, gewählte Knoten sind diejenigen in S, und auf die Berechnung der Wege (also von Vorgängerknoten auf kürzesten Wegen) wurde verzichtet.

8.5.2 Kürzeste Wege in beliebig bewerteten Graphen

Die Berechnung kürzester Wege ändert sich erheblich, wenn wir auch negative Kantenbewertungen zulassen, also eine Längenfunktion $c : E \to \mathbb{R}$ voraussetzen. Ändern wir beispielsweise in dem in Abbildung 8.18 gezeigten Graphen die Bewertung der Kante $(2,7)$ auf -6 und die der Kante $(2,3)$ auf -4, so sind nicht nur die zuvor gefundenen kürzesten Wege nun keine kürzesten mehr, sondern es gibt plötzlich gar keinen kürzesten Weg mehr im Graphen. Der Grund dafür ist die Existenz eines Zyklus negativer Länge, nämlich des Zyklus (2, 3, 4, 9, 7, 2) mit Länge -5. Zu jedem denkbaren Weg zwischen zwei Knoten kann man nun einen kürzeren Weg finden, indem man einen Abstecher zu diesem *negativen Zyklus* macht und ihn — unter Umständen mehrfach — durchläuft. In einem bewerteten, ungerichteten oder gerichteten Graphen, der einen Weg von einem Knoten s zu einem Knoten t enthält, gibt es einen kürzesten Weg von s nach t genau dann, wenn kein Weg von s nach t einen Zyklus negativer Länge enthält. Wenn es einen kürzesten Weg von s nach t gibt, dann gibt es natürlich auch einen einfachen kürzesten Weg von s nach t.

Selbst im Falle negativer Kantenbewertungen lassen sich alle kürzesten Wege von einem Anfangsknoten s mit Hilfe einer Breitensuche bestimmen: Man berechnet die Länge von Wegen für zunehmende Kantenzahl. Man kann aber nicht, wie im vorangehenden Abschnitt, die Länge eines Weges zu einem gewählten Knoten als endgültig kürzest ansehen, weil das Hinzunehmen von Kanten die Länge eines Weges verkürzen kann. Die Länge eines Weges vom Anfangsknoten s aus läßt sich genau dann verkürzen, wenn der folgende *Auswahlschritt* von Ford [56] angewandt werden kann:

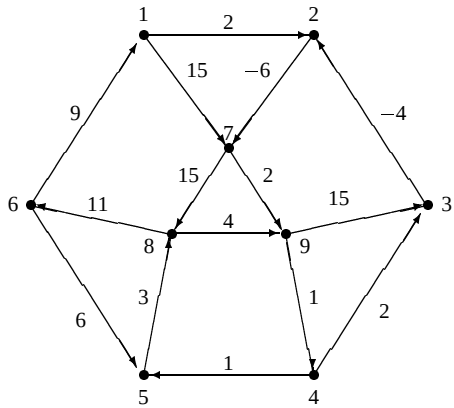


Abbildung 8.21

Auswahlschritt *von Ford*:
wähle eine Kante $(v,v') \in E$ *mit*
$v.Entfernung + c((v,v')) < v'.Entfernung$;
$v'.Vorgänger := v$;
$v'.Entfernung := v.Entfernung + c((v,v'))$;

Wählen wir als vorläufige Entfernung *v.Entfernung* anfangs 0 für $v = s$ und ∞ für $v \neq s$, dann bewahrt Fords Auswahlschritt die folgende *Invariante*: Wenn *v.Entfernung* einen endlichen Wert hat, dann gibt es einen Weg von s nach v mit Länge *v.Entfernung*.

Ein *Auswahlverfahren nach Ford* sei nun jedes Verfahren, das den Auswahlschritt von Ford wiederholt solange anwendet, bis dies nicht mehr möglich ist. Wenn ein Auswahlverfahren nach Ford anhält, dann ist *v.Entfernung* für jeden von s aus erreichbaren Knoten v die Länge eines kürzesten Wegs von s nach v und für alle anderen Knoten ∞. Ein Auswahlverfahren nach Ford hält nicht an, wenn es einen von s aus erreichbaren negativen Zyklus im Graphen gibt.

Eine Implementierung eines Auswahlverfahrens nach Ford muß noch spezifizieren, wie denn eine Kante (v,v') im Auswahlschritt gewählt werden soll. Hierfür eignet sich eine Breitensuche, ähnlich wie bei Distanzgraphen, wobei aber lediglich Randknoten von Bedeutung sind. Zwischen gewählten und unerreichten Knoten wird nicht unterschieden. Durch Abänderung des Algorithmus für kürzeste Wege in Distanzgraphen ergibt sich damit das folgende Auswahlverfahren nach Ford:

Algorithmus *kürzeste Wege in* $G = (V,E)$ *mit* $c : E \to \mathbb{R}$ *von einem Knoten* $s \in V$ *zu allen anderen*
1. {*Initialisierung*:}
1.1 {*anfangs kennt man für alle Knoten außer s keinen Weg*:}
for all $v \in V - \{s\}$ **do**
begin

```
            v.Vorgänger := undefiniert;
            v.Entfernung := ∞
          end;
1.2   {für s ist ein Weg bekannt:}
        s.Vorgänger := s;
        s.Entfernung := 0;
1.3   {alle zu s adjazenten Knoten gehören zum Rand R:}
        R := ∅;
        verschiebe R bei s;
2.    {berechne Wege ab s:}
        while R ≠ ∅ do
          begin
            wähle v ∈ R und entferne v aus R;
            verschiebe R bei v
          end
end {kürzeste Wege}
```

Beim Verschieben des Randes bei einem Knoten v werden alle mit v inzidenten Kanten auf ihre Eignung für den Auswahlschritt von Ford überprüft, und für die geeigneten Kanten wird der Auswahlschritt durchgeführt:

```
verschiebe R bei v :
  for all (v,v') ∈ E do
    if v.Entfernung +c((v,v')) < v'.Entfernung
      then {v' ist (kürzer) über v erreichbar}
        begin
          v'.Vorgänger := v;
          v'.Entfernung := v.Entfernung +c((v,v'));
          vermerke v' in R, falls v' dort nicht bereits vermerkt ist
        end
```

Die Prüfung, ob ein Knoten v' bereits im Rand vermerkt ist, kann mit Hilfe eines Bits pro Knoten leicht in konstanter Zeit erfolgen. Da es unerheblich ist, welcher Knoten aus dem Rand gewählt wird, kann als an der Breitensuche orientierte Datenstruktur für den Rand beispielsweise eine Schlange gewählt werden.

Man sieht, daß dieser Algorithmus demjenigen für die Berechnung kürzester Wege mit positiven Kantenbewertungen stark ähnelt; es ist instruktiv, sich die Unterschiede durch vergleichende Betrachtung beider Algorithmen deutlich zu machen.

Für den in Abbildung 8.21 gezeigten Digraphen haben wir in Abbildung 8.22 den Verlauf des Algorithmus bis zu den ersten zehn Randverschiebeoperationen angegeben. Die verschiedenen Inhalte der Schlange der Randknoten sind in zeitlicher Abfolge waagerecht nebeneinander dargestellt. Bei jedem Randknoten sind die aktuelle Entfernung zu Knoten 1 und der zugehörige Vorgänger mit angegeben, obwohl sie natürlich nicht in der Schlange verwaltet werden (sonst müßte man beispielsweise die Position eines Knotens in der Schlange kennen, um seinen Entfernungswert zu ändern). Man sieht beispielsweise, wie zunächst für Knoten 8 ein Weg der Länge 11 von Knoten 1 über Knoten 2 und Knoten 7 gefunden wird; erst später findet man den kürzeren Weg mit Länge 3 von Knoten 1 über Knoten 2, 7, 9, 4 und 5.

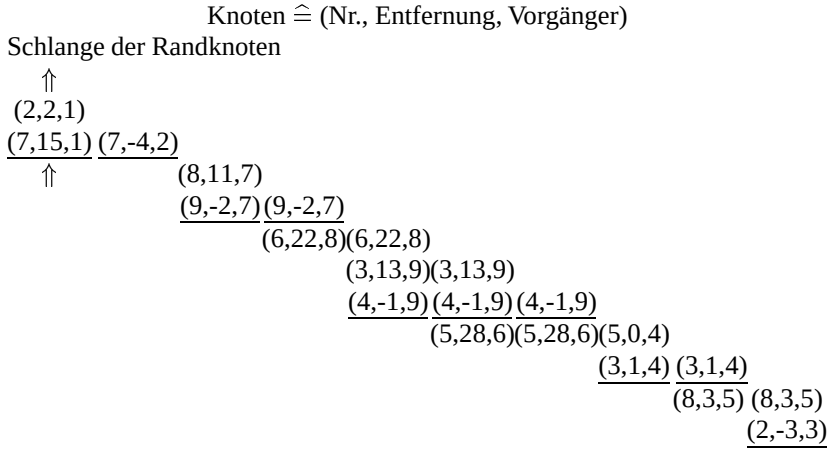


Abbildung 8.22

Der waagerechte Querstrich ist ein spezieller Markierungseintrag in der Schlange, der einfach ans Schlangenende angehängt wird, sobald er am Schlangenkopf angekommen ist. Er dient der Illustration der *Phasen* des Algorithmus. In Phase 1 werden alle Knoten erreicht, die mit *einem* Pfeil vom Anfangsknoten aus erreichbar sind; im Beispiel sind dies die Knoten 2 und 7. In Phase $j+1$ werden alle Knoten erreicht, die mit *einem* Pfeil von den in Phase j erreichten Knoten aus erreichbar sind. Natürlich müssen die in Phase j erreichten Knoten dort nicht unbedingt erstmals erreicht worden sein. In der Schlange der Randknoten befinden sich zwischen Schlangenkopf und Phasen-Ende-Markierung und zwischen Phasen-Ende-Markierung und Schlangenende jeweils höchstens die Knoten der entsprechenden Phase, unter Umständen auch weniger. So werden etwa im gezeigten Beispiel in Phase 2 die Knoten 7, 8 und 9 erreicht, aber Knoten 7 befindet sich aus Phase 1 zu dem Zeitpunkt noch in der Schlange, zu dem er über Knoten 2 in Phase 2 erreicht wird.

Die Laufzeit des Algorithmus läßt sich abschätzen, wenn man die einzelnen Phasen betrachtet. In jeder Phase wird jeder Knoten höchstens einmal betrachtet, zusammen mit seinen inzidenten Kanten. Dies ergibt wegen der in konstanter Zeit ausführbaren einzelnen Schlangenoperationen eine Laufzeit von $O(|E|)$ für jede Phase. Weil es stets einen einfachen, also zyklenfreien, kürzesten Weg gibt — es sei denn, ein Weg über einen negativen Zyklus ist möglich —, genügt es, Wege mit höchstens $|V|$ Knoten zu betrachten. Damit kann die Berechnung kürzester Wege nach höchstens $|V|$ Phasen abgebrochen werden. Man kann also ein Auswahlverfahren nach Ford für einen bewerteten Digraphen $G=(V,E)$ so implementieren, daß es in $O(|V|\cdot|E|)$ Schritten kürzeste Wege von einem zu allen anderen Knoten berechnet, falls diese existieren. Andernfalls hält dieses Verfahren nicht an. Zählt man allerdings die Anzahl der Phasen mit, dann kann man in jedem Fall nach dem Ende der $|V|$-ten Phase anhalten. Wenn nämlich nach dem Ende der $|V|$-ten Phase der Rand R nicht leer ist, gibt es einen vom Anfangsknoten s aus erreichbaren, negativen Zyklus in G.

Natürlich ist man bei der Reihenfolge, in der man im Schritt 2 des Algorithmus Randknoten auswählt, nicht auf die durch die Implementierung des Randes als Schlange festgelegte Reihenfolge angewiesen. Entscheidet man sich bei einem azyklischen Graphen etwa für die Reihenfolge einer topologischen Sortierung, so ist für jeden aus dem Rand gewählten Knoten die Berechnung der Entfernung endgültig. Die Laufzeit des Algorithmus verkürzt sich damit zu $O(|E|)$. Dies ist ein für die Netzplantechnik wichtiges Ergebnis, weil man damit auch längste Wege in azyklischen Graphen schnell berechnen kann: Man multipliziert einfach die Längen der Pfeile mit -1 und berechnet danach kürzeste Wege.

8.5.3 Alle kürzesten Wege

Wir betrachten nun das Problem, für jedes Paar v und v' von Knoten einen kürzesten Weg von v nach v' zu berechnen. Dieses Problem läßt sich einfach dadurch lösen, daß wir einen Algorithmus zum Finden kürzester Wege von einem zu allen anderen Knoten für jeden Knoten anwenden. Für einen Distanzgraphen ergibt sich bei dieser Vorgehensweise eine Laufzeit von $O(|V| \cdot (|E| + |V| \log |V|))$, für einen beliebigen, bewerteten Graphen ohne negative Zyklen eine Laufzeit von $O(|E| \cdot |V|^2)$. Daß es auch schneller geht, wollen wir uns für beliebige, bewertete Graphen ohne negative Zyklen überlegen. Das Verfahren, das wir hierfür verwenden wollen, hat folgende Grobstruktur:

Algorithmus *alle kürzesten Wege in* $G = (V,E)$ *mit* $c : E \to \mathbb{R}$
1. *Transformiere G in einen Distanzgraphen* G' *so, daß kürzeste Wege erhalten bleiben*;
2. *wende Algorithmus kürzeste Wege für jeden Knoten in* G' *an*

end *{alle kürzesten Wege}*

Dabei kann der kritische Schritt, die Transformation von G in einen Distanzgraphen G', wie folgt realisiert werden [45]. Zunächst nimmt man einen neuen Knoten s zum Graphen hinzu und verbindet s mit je einem Pfeil mit jedem anderen Knoten des Graphen (siehe Abbildung 8.23). Wir wählen der Einfachheit halber Pfeillänge 0 für jeden dieser Pfeile, obgleich man interessanterweise jeden einzelnen Pfeil beliebig bewerten könnte.

Damit ist die Länge eines kürzesten Weges von s zu einem beliebigen anderen Knoten des Graphen stets höchstens 0. Betrachten wir nun einen Pfeil (v,v') aus G. Einer der Wege von s nach v' führt über v. Da ein kürzester Weg $sp(s,v')$ von s nach v' nicht länger sein kann als der Umweg über v, gilt offenbar $c(sp(s,v')) \leq c(sp(s,v)) + c((v,v'))$. Damit gilt für die durch $c'((v,v')) := c((v,v')) + c(sp(s,v)) - c(sp(s,v'))$ definierte Länge c' im transformierten Graphen unmittelbar $c'((v,v')) \geq 0$. Der transformierte Graph ist also ein Distanzgraph. In dem in Abbildung 8.23 gezeigten Beispiel ergibt die Transformation für den Pfeil (v,v') eine Länge von 4 und für den Pfeil (v'',v') eine Länge von 0. Beim Aufsummieren der transformierten Längen entlang eines Weges von einem Knoten v zu einem Knoten w neutralisieren sich die Längen kürzester Wege von s zu Zwischenknoten auf dem Weg von v nach w; lediglich die Längen kürzester Wege von s nach v und nach w bleiben übrig. Für *jeden* Weg p von einem Knoten v zu einem Knoten w gilt also $c'(p) = c(p) + c(sp(s,v)) - c(sp(s,w))$. Damit bleibt die relative

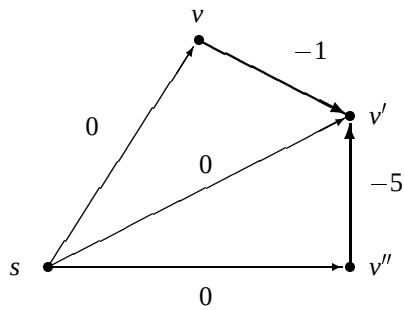


Abbildung 8.23

Ordnung der Längen aller Wege von v nach w bei der Transformation erhalten. Insbesondere bleibt also ein kürzester Weg in G auch ein kürzester Weg in G'. Algorithmisch kann die Transformation wie folgt realisiert werden:

Algorithmus *transformiere* $G = (V,E)$ *mit* $c : E \to \mathbb{R}$ *in* $G' = (V',E')$ *mit* $c' : E' \to \mathbb{R}_0^+$:

1. $V' := V \cup \{s\}$;
   $E' := E \cup \{(s,v) \mid v \in V\}$;
   for all $v \in V$ **do**
   $c((s,v)) := 0$;
2. *berechne kürzeste Wege in* G' *von s zu allen anderen Knoten* $v \in V$ *und vermerke die Länge jeweils in v.Entfernung*;

3. **for all** $(v,v') \in E$ **do**
   $c'((v,v')) := c((v,v')) + v.Entfernung - v'.Entfernung$

end *{transformiere}*

Schritt 1 der Transformation kann in Laufzeit $O(|V|)$ bewältigt werden; für Schritt 2 genügt eine Laufzeit von $O(|V| \cdot |E|)$, wie im vorangehenden Abschnitt gezeigt wurde. Schritt 3 kann in Zeit $O(|E|)$ erledigt werden, so daß die gesamte Transformation in Zeit $O(|V| \cdot |E|)$ durchgeführt werden kann. Die $|V|$-malige Anwendung des Algorithmus für kürzeste Wege in einem Distanzgraphen mit einer Laufzeit von jeweils $O(|E| + |V| \log |V|)$ führt zu einer Gesamtlaufzeit des Verfahrens von $O(|V| \cdot (|E| + |V| \log |V|))$. Damit können alle kürzesten Pfade in einem beliebigen, bewerteten Graphen ebenso schnell berechnet werden wie in einem Distanzgraphen.

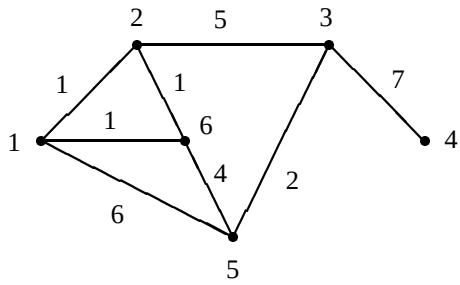


Abbildung 8.24

8.6 Minimale spannende Bäume

Ein *minimaler spannender Baum* (englisch: *minimum spanning tree; MST*) eines Graphen G ist ein spannender Baum von G von minimaler Gesamtlänge unter allen spannenden Bäumen von G. Minimale spannende Bäume sind oft dann von Interesse, wenn es darum geht, aus einer Vielzahl möglicher Kanten diejenigen auszuwählen, die alle Knoten mit kürzester Gesamtlänge verbinden. So kann man sich etwa vorstellen, daß in dem in Abbildung 8.24 gezeigten Graphen die Knoten hausinterne Telefonanschlüsse einer großen Firma repräsentieren und die Kantenlängen Kosten für das Legen einer entsprechenden Direktleitung sind. Telefongespräche von einer Sprechstelle zur anderen sollen auch über Zwischenstationen, also indirekt, geschaltet werden können.

In der Tat hat die amerikanische Telefonfirma AT&T die Gebühren für hausinterne Netze von Firmenkunden nach der Länge eines minimalen spannenden Baumes aller denkbaren Direktleitungen — und nicht nach der Länge der tatsächlich verlegten Leitungen — berechnet. Bei diesem Berechnungsverfahren kann es natürlich vorkommen, daß durch das Hinzunehmen weiterer Telefonanschlüsse die Gesamtkosten gesenkt werden. Dies ist leicht am Beispiel der Abbildung 8.24 einzusehen: Würde man in dem von Knotenmenge {2,3,4,5} induzierten Untergraphen Knoten 3 entfernen und dafür Knoten 2 und 4 mit einer Kante der Länge 12, Knoten 2 und 5 mit einer Kante der Länge 7 und Knoten 4 und 5 mit einer Kante der Länge 9 direkt verbinden, so würde die Länge eines minimalen spannenden Baumes von 14 auf 16 wachsen. Das Problem, einen kürzesten Baum in einem Graphen von Telefondirektleitungen zu finden, der neben den in der Firma wirklich benötigten Telefonsprechstellen auch *optionale* Sprechstellen enthält, die nur in das Telefonnetz einbezogen werden sollen, wenn dadurch dessen Gesamtlänge verkürzt wird, ist ungleich aufwendiger zu lösen als das Problem des Findens eines minimalen spannenden Baumes; wir werden es in diesem Buch nicht weiter betrachten.

Zur Berechnung eines minimalen spannenden Baumes in einem zusammenhängenden, ungerichteten Graphen wollen wir ein *gieriges* (englisch: *greedy*) Verfahren verwenden. Bei gierigen Verfahren werden Entscheidungen, die den Rechenprozeß der

Lösung näher bringen, auf der Basis der vom Rechenprozeß bis dahin gesammelten Informationen gefällt und nicht mehr revidiert. Im Unterschied zu Verfahren, die Lösungsschritte ausprobieren und gegebenenfalls revidieren müssen, sind gierige Verfahren stets vergleichsweise effizient. Wir wählen das folgende Verfahren:

Algorithmus-Gerüst *Minimaler spannender Baum*
{*liefert zu einem zusammenhängenden, ungerichteten, bewerteten Graphen* $G = (V,E)$ *mit* $c : E \to \mathbb{R}$ *einen minimalen spannenden Baum* $T' = (V,E')$ *von* G}
begin
  $E' := \emptyset$;
  while *noch nicht fertig* **do**
    begin
      wähle geeignete Kante $e \in E$;
      $E' := E' \cup \{e\}$
    end
end *{Minimaler spannender Baum}*

Es bleibt hier im wesentlichen offen, welches geeignete Kanten sind und wie man sie wählt. Wir präzisieren das Verfahren als *Auswahlprozeß* für Kanten von G. Dabei hat eine Kante stets einen von drei Zuständen: Sie ist entweder *gewählt*, *verworfen* oder *unentschieden*. Anfangs ist jede Kante unentschieden. Es soll stets die *Auswahlinvariante* gelten, daß es einen minimalen spannenden Baum von G gibt, der alle gewählten und keine verworfenen Kanten enthält. Zu Beginn ist dies natürlich erfüllt, da alle Kanten unentschieden sind. Am Ende des Verfahrens sollen alle Kanten gewählt oder verworfen sein. Dann gilt mit der Invariante offenbar, daß gerade die gewählten Kanten einen minimalen spannenden Baum bilden.

Im Laufe der Jahre sind verschiedene effiziente Algorithmen vorgeschlagen worden, die nach diesem Verfahren operieren und für Kanten gemäß einer von zwei Regeln entscheiden, ob sie gewählt oder verworfen werden. Eine dieser Regeln betrachtet Schnitte im Graphen. Ein *Schnitt* (englisch: *cut*) in einem Graphen $G = (V,E)$ ist eine Zerlegung von V in S und $\overline{S} = V - S$. Eine Kante *kreuzt* den Schnitt, wenn sie mit einem Knoten aus S und einem aus $\overline{S}$ inzident ist. Im Beispiel der Abbildung 8.24 ist $S = \{2,4,5\}$ und $\overline{S} = \{1,3,6\}$ ein Schnitt, den alle Kanten außer $(1,6)$ kreuzen. Die folgenden beiden Regeln dienen der Entscheidung darüber, ob eine unentschiedene Kante gewählt oder verworfen wird:

Regel 1: *Wähle eine Kante*
Wähle einen Schnitt, den keine gewählte Kante kreuzt.
Wähle eine kürzeste unter den unentschiedenen Kanten, die den Schnitt kreuzen.

Regel 2: *Verwirf eine Kante*
Wähle einen einfachen Zyklus, der keine verworfene Kante enthält.
Verwirf eine längste unter den unentschiedenen Kanten im Zyklus.

Verschiedene effiziente Algorithmen für das Berechnen minimaler spannender Bäume unterscheiden sich nun zum einen in der Reihenfolge, in der diese beiden Regeln angewandt werden, und zum anderen in der Art, wie ein Schnitt oder ein Zyklus gewählt wird. Allen gemeinsam sind die folgenden beiden Präzisierungen des Algorithmusgerüsts *Minimaler spannender Baum*:

Wähle geeignete Kante $e \in E$:
repeat
wende eine anwendbare Auswahlregel an und
until *Kante $e \in E$ mit Regel 1 gewählt oder es gibt keine unentschiedene Kante mehr*

noch nicht fertig:
es gibt noch unentschiedene Kanten

Jedes so operierende Verfahren ist ein korrektes Verfahren zum Berechnen eines minimalen spannenden Baumes. Weil das Algorithmusgerüst *Minimaler spannender Baum* mit den beiden angegebenen Präzisierungen Grundlage aller von uns behandelten Verfahren zum Berechnen minimaler spannender Bäume ist, wollen wir Überlegungen zu seiner Korrektheit etwas ausführlicher anstellen.

Satz 8.1 *Jedes nach dem Algorithmusgerüst Minimaler spannender Baum mit den beiden angegebenen Präzisierungen operierende Verfahren wählt oder verwirft jede Kante eines zusammenhängenden, ungerichteten, bewerteten Graphen und bewahrt die Auswahlinvariante.*

Beweis: Wir zeigen zunächst, daß die Auswahlinvariante bewahrt wird. Wir wissen bereits, daß die Invariante anfangs erfüllt ist, denn jeder zusammenhängende, ungerichtete, bewertete Graph besitzt einen minimalen spannenden Baum, und jeder minimale spannende Baum erfüllt die Invariante. Wir betrachten jetzt den Effekt der Anwendung jeder der beiden Regeln auf die Invariante.

Betrachten wir zunächst *Regel* 1: *Wähle eine Kante.* Sei e die mit Regel 1 gewählte Kante und sei T ein minimaler spannender Baum, der die Invariante erfüllt, bevor e gewählt wird. T enthält also alle vor der Wahl von e gewählten und keine der vor der Wahl von e verworfenen Kanten. Gehört nun e zu den Kanten von T, so wird offensichtlich die Invariante bewahrt. Gehört andererseits e nicht zu den Kanten von T, so betrachten wir den in Regel 1 gewählten Schnitt $S, \overline{S}$ (vgl. Abbildung 8.25). Wenigstens eine Kante des Wegs in T, der die beiden Endknoten von e verbindet, kreuzt diesen Schnitt; nennen wir eine solche Kante e'. Weil T die Invariante erfüllt, kann e' nicht verworfen sein. Weil Regel 1 auf den Schnitt angewandt wurde, kann e' nicht gewählt sein. Also ist e' unentschieden und wegen Regel 1 nicht kürzer als e. Dann erhalten wir aus T durch Entfernen von e' und Hinzufügen von e einen Baum $T' = (T - \{e'\}) \cup \{e\}$, der die Invariante nach Anwendung von Regel 1 erfüllt und ein minimaler spannender Baum ist.

Betrachten wir nun *Regel* 2: *Verwirf eine Kante.* Sei e die durch Regel 2 verworfene Kante und T ein minimaler spannender Baum, der die Invariante vor der Anwendung von Regel 2 erfüllt. Falls e nicht zu T gehört, so wird die Invariante bewahrt. Falls aber e zu T gehört, so wird T durch das Entfernen von e in zwei Teile geteilt, die einen Schnitt für G bilden; e kreuzt diesen Schnitt. Weil Regel 2 angewandt werden konnte, liegt e in einem einfachen Zyklus, der keine verworfene Kante enthält; dieser Zyklus enthält wenigstens eine andere Kante, wir nennen sie e', die den Schnitt kreuzt (siehe Abbildung 8.26). Weil e' nicht zu T gehört, ist e' unentschieden; weil mit Regel 2 Kante e verworfen wird, ist e' nicht länger als e. Dann erhalten wir aus T durch Entfernen von e

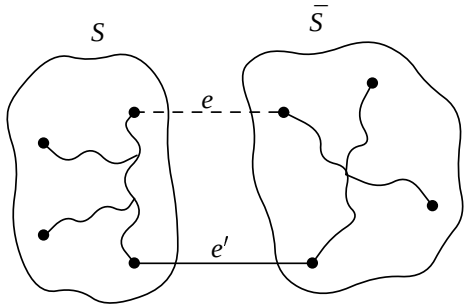


Abbildung 8.25

und Hinzunehmen von e' einen minimalen spannenden Baum $T' = (T - \{e\}) \cup \{e'\}$, der die Invariante nach Anwendung von Regel 2 erfüllt. Also wird die Auswahlinvariante im Algorithmus bewahrt.

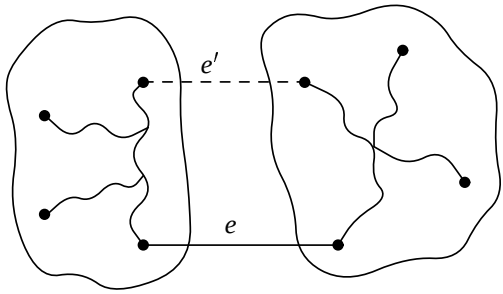


Abbildung 8.26

Wir zeigen, daß keine Kante unentschieden bleibt, indem wir aus der gegenteiligen Annahme einen Widerspruch herleiten. Nehmen wir also an, *e* sei eine Kante, die unentschieden bleibt. Zu jedem Zeitpunkt im Verlauf der Rechnung bilden die bereits gewählten Kanten eine Menge *gewählter Bäume*. Falls beide Endknoten von *e* im selben gewählten Baum liegen, ist Regel 2 anwendbar. Es kann also eine Kante verworfen werden (nicht unbedingt *e*). Falls beide Endknoten von *e* in verschiedenen gewählten Bäumen liegen, ist Regel 1 anwendbar. Es kann also eine Kante gewählt werden (nicht unbedingt *e*). Damit sichert die Existenz einer unentschiedenen Kante die Anwendbarkeit einer Auswahlregel; mit der Anwendung einer Auswahlregel verringert sich aber die Anzahl unentschiedener Kanten um 1. Damit kann keine Kante unentschieden bleiben. □

Betrachten wir nun im Einzelnen einige Algorithmen zur Berechnung eines minimalen spannenden Baumes. Wir werden zur Beschreibung der Algorithmen stets nur angeben, auf welche Weise Kanten gewählt oder verworfen werden.

Der Algorithmus von Boruvka [21]
Dies ist der historisch erste Algorithmus zur Berechnung minimaler spannender Bäume; wir wollen ihn hier nur kurz skizzieren; eine parallelisierte Version hiervon, Sollins Algorithmus, wird in Kapitel 9 genauer behandelt. Für einen Graphen $G = (V,E)$ ist am Anfang jeder einzelne Knoten ein gewählter Baum. In einem *Auswahlschritt* wird für jeden gewählten Baum eine kürzeste Kante zu einem anderen Baum gewählt. Gibt es für einen Baum mehr als eine kürzeste Kante zu einem anderen Baum, so wird diejenige gewählt, die mit einem Knoten kleinster Nummer inzidiert. Auf diese Weise wird vermieden, daß in einem Auswahlschritt durch ungeschickte Entscheidung für eine von mehreren kürzesten Kanten ein Zyklus entsteht. Die wiederholte Anwendung des Auswahlschritts liefert einen minimalen spannenden Baum. Im Beispiel der Abbildung 8.24 werden im ersten Auswahlschritt die Kanten $(1,2),(2,1),(3,5),(4,3),(5,3),(6,1)$ gewählt, wobei wir die Kanten $(1,2)$ und $(3,5)$ jeweils zweimal aufgeführt haben, weil sie einmal von Baum 1 bzw. Baum 3 aus und einmal von Baum 2 bzw. Baum 5 aus gewählt werden. Im zweiten Auswahlschritt wird der minimale spannende Baum durch Auswahl von Kanten $(5,6),(6,5)$ vervollständigt.

Der Algorithmus von Kruskal [96]
Anfangs ist jeder einzelne Knoten des Graphen ein gewählter Baum. Dann wird auf jede Kante *e* in aufsteigender Reihenfolge der Kantenlängen folgender *Auswahlschritt* angewandt: Falls *e* beide Endknoten im selben gewählten Baum hat, verwirf *e*, sonst wähle *e*. Abbildung 8.27 zeigt die gewählten Bäume und die gewählten Kanten für das Beispiel in Abbildung 8.24.

Bei einer effizienten Implementierung des Verfahrens von Kruskal muß man außer der Sortierung von Kanten nach ihrer Länge die bereits gewählten Bäume so verwalten, daß zwei gewählte Bäume zu einem gewählten Baum verbunden werden können, und daß geprüft werden kann, in welchem Baum der Endknoten einer Kante liegt. Dies gelingt gerade mit Hilfe einer Union-find-Struktur, wie sie in Kapitel 6 beschrieben ist. Eine solche Struktur bietet die folgenden Operationen an:
– *Find*(v) ist der Name des gewählten Baumes, zu dem Knoten *v* gehört;
– *Union*(v,w) vereinigt Bäume mit Namen *v* und *w* zu einem Baum mit Namen *v*;
– *Make-set*(v) kreiert den Baum, dessen einziger Knoten *v* ist.

Damit kann Kruskals Verfahren im Algorithmusgerüst *Minimaler spannender Baum* wie folgt präzisiert werden:

```
begin {Kruskal}
  E' := ∅;
  sortiere E nach aufsteigender Länge;
  for all v ∈ V do
    Make-set(v);
  for all (v,w) ∈ E, aufsteigend, do
    if Find(v) ≠ Find(w) then {wähle Kante (v,w)}
      begin
        Union(Find(v),Find(w));
        E' := E' ∪ {(v,w)}
      end
end {Kruskal}
```

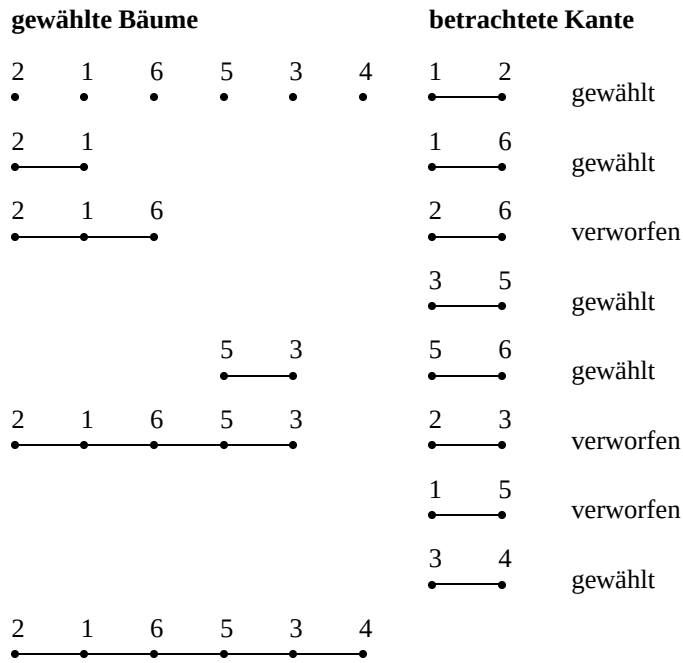


Abbildung 8.27

Das Verfahren von Kruskal ist auch schon in Abschnitt 8.6 beschrieben. Dort ist die Kollektion von Mengen explizit angesprochen, auf die hier nur über die Operationen der Union-Find-Struktur zugegriffen wird. Überdies ist in Abschnitt 6.2.1 eine Alternative zum Sortieren angegeben, das Verwalten der Kanten nach ihrer Länge in einer Prioritätswarteschlange. Beide Varianten sind asymptotisch gleich effizient. Das Sortieren der Kanten des Graphen kann in Zeit $O(|E|\log|E|) = O(|E|\log|V|)$ ausgeführt werden; für $O(|V|)$ *Make-set*-, $O(|E|)$ *Find*- und $O(|V|)$ *Union*-Operationen benötigt man nicht mehr als $O(|E|\alpha(|E|,|V|)) = O(|E|\log|V|)$ Schritte. Damit ergibt sich die gesamte Laufzeit des Verfahrens für einen Graphen $G = (V,E)$ zu $O(|E|\log|V|)$. Aber es geht noch schneller.

Der Algorithmus von Jarník, Prim, Dijkstra [35, 82, 150]

Dieses Verfahren ähnelt Dijkstras Verfahren zur Berechnung kürzester Wege. Zu jedem Zeitpunkt bilden die gewählten Kanten *einen* gewählten Baum. Wir beginnen mit einem beliebigen Anfangsknoten s des Graphen und führen den folgenden *Auswahlschritt* $(|V|-1)$-mal aus: Wähle eine Kante mit minimaler Länge, für die genau ein Endknoten zum gewählten Baum gehört. Zu Beginn besteht der gewählte Baum aus dem Anfangsknoten s; später bilden alle gewählten Kanten und deren inzidente Knoten den gewählten Baum. Abbildung 8.28 zeigt den Verlauf des Algorithmus, angewandt auf den Graphen der Abbildung 8.24, beginnend mit Anfangsknoten 1.

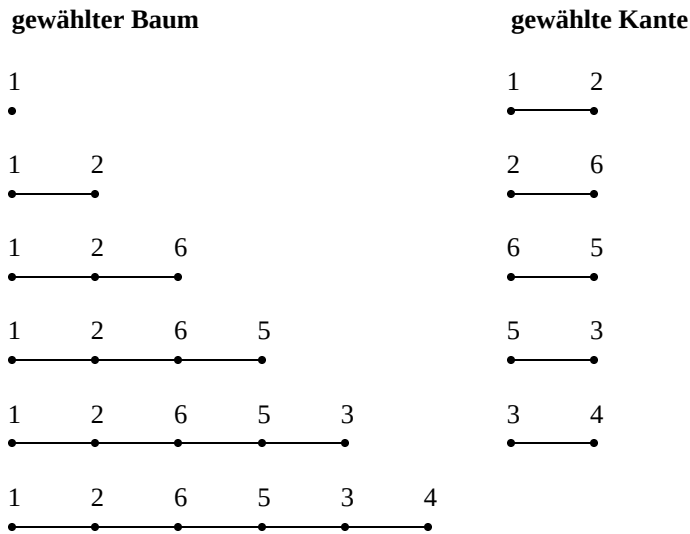


Abbildung 8.28

Da hierbei nur *ein* Baum wächst, benötigen wir im Unterschied zu Kruskals Algorithmus keine Union-find-Struktur; statt dessen genügt eine Priority Queue. Wie bei Dijkstras Algorithmus für kürzeste Wege hängt die Effizienz der Implementierung des Verfahrens ab von der Wahl einer Datenstruktur für die Priority Queue. Die beste Wahl ist hier der Fibonacci-Heap [60]. Dann unterscheidet sich der Algorithmus für minimale spannende Bäume von dem für kürzeste Wege nur dadurch, daß anstelle der Entfernung zum Anfangsknoten für die kürzesten Wege nunmehr die Entfernung zum nächsten Knoten im gewählten Baum verwaltet werden muß. Dies kann aber auf die gleiche Weise geschehen wie beim Algorithmus zum Finden kürzester Wege. Damit läßt sich dieser Algorithmus zum Finden eines minimalen spannenden Baumes für einen zusammenhängenden, ungerichteten, bewerteten Graphen $G = (V, E)$ so implementieren, daß er mit einer Laufzeit von $O(|E| + |V| \log |V|)$ auskommt. In [60] ist ein noch schnellerer Algorithmus beschrieben, bei dem mehrere Bäume ein wenig wachsen und dann zu Superknoten kollabieren; dasselbe Verfahren wird auf den so kondensierten Graphen angewandt, bis schließlich ein minimaler spannender Baum erreicht ist.

8.7 Flüsse in Netzwerken

Welchen Verkehrsfluß (in Fahrzeugen pro Minute) kann ich höchstens durch eine Stadt leiten, deren Straßennetz gegeben ist? Welche Wassermenge kann ich durch die Kanalisation höchstens abtransportieren? Solche und andere Flußprobleme in Netzwer-

ken sind in vielen Varianten und Verkleidungen ausgiebig untersucht worden. Obgleich schon 1962 ein inzwischen klassisches Buch zu diesem Thema [58] erschien, werden auch heute noch immer wieder neue und bessere Algorithmen für Flußprobleme gefunden. Wir betrachten hier das Problem, maximale Flüsse in Netzwerken zu finden, bei denen Pfeile Verbindungen repräsentieren, durch die Güter fließen können. Dabei hat jeder Pfeil nur eine beschränkte Kapazität; beispielsweise verträgt ein Straßenstück nur einen Durchsatz von 10 Fahrzeugen je Minute, oder ein Kanalisationsrohr verkraftet nicht mehr als 20 Liter pro Sekunde.

Sei im folgenden $G = (V,E)$ ein gerichteter Graph mit einer Kapazitätsfunktion $c : E \to \mathbb{R}^+$ (englisch: capacity) und zwei ausgezeichneten Knoten, einer *Quelle* q und einer *Senke* s. Unser Ziel ist es, einen maximalen Fluß von q nach s zu ermitteln. Ein Fluß durch einen Pfeil muß die Kapazitätsbeschränkung dieses Pfeils einhalten; an jedem Knoten muß der Fluß erhalten bleiben, also gleichviel hinein- wie herausfließen (außer an der Quelle und an der Senke). Wir definieren daher einen *Fluß* als eine Funktion $f : E \to \mathbb{R}_0^+$, wobei gilt:

- *Kapazitätsbeschränkung*: für alle $e \in E$ ist $f(e) \leq c(e)$;
- *Flußerhaltung*: für alle $v \in V - \{q,s\}$ ist $\sum_{(v',v)\in E} f((v',v)) - \sum_{(v,v'')\in E} f((v,v'')) = 0$.

Der Einfachheit halber wird oft angenommen, daß kein Pfeil in q mündet und kein Pfeil s verläßt; wir wollen hier im allgemeinen auf diese Annahme verzichten, aber unsere Beispiele manchmal so beschränken.

Betrachten wir das in Abbildung 8.29 gezeigte Beispiel. An jedem Pfeil e ist dort

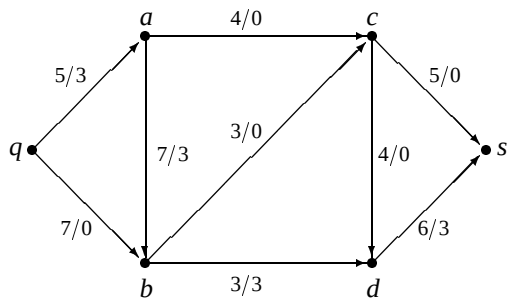


Abbildung 8.29

$c(e)/f(e)$ angegeben. Es fließt also gerade ein Fluß von Knoten q über Knoten a, b, d zu Knoten s. Der *Wert* $w(f)$ eines Flusses f ist die Summe der Flußwerte aller q verlassenden Pfeile, also $w(f) = \sum_{(q,v)\in E} f((q,v)) - \sum_{(v',q)\in E} f((v',q))$. In unserem Beispiel ist $w(f) = 3$. Ein *maximaler Fluß* in G ist ein Fluß f in G mit maximalem Wert $w(f)$ unter allen Flüssen in G. Für das Problem, einen maximalen Fluß in einem gegebenen

Digraphen zu ermitteln, sind im Laufe der Zeit zahlreiche, verschiedene Algorithmen vorgeschlagen worden. Wir werden im folgenden einige der wichtigsten vorstellen.

Überlegen wir uns aber zunächst, wie groß ein maximaler Fluß überhaupt sein kann. Es ist intuitiv plausibel, daß nicht mehr im Netzwerk fließen kann, als aus der Quelle herausfließt oder in die Senke hineinfließt. In unserem Beispiel verlassen höchstens 12 Einheiten die Quelle, und höchstens 11 fließen in die Senke. Aber nicht nur Quelle und Senke begrenzen den Wert eines maximalen Flusses, sondern jeder Schnitt durch den Graphen, der q von s trennt. Ein (*q von s trennender*) *Schnitt* ist eine Zerlegung der Knotenmenge V in zwei Teilmengen Q und S, so daß q zu Q und s zu S gehört. Die *Kapazität* $c(Q,S)$ eines Schnittes Q,S ist die Summe der Kapazitäten von Pfeilen, die von Q nach S führen, also $c(Q,S) = \sum_{v\in Q, v'\in S, (v,v')\in E} c((v,v'))$. Ein Schnitt mit kleinster Kapazität unter allen möglichen Schnitten heißt *minimaler Schnitt*. In dem in Abbildung 8.29 gezeigten Beispiel ist etwa $Q = \{q,b\}, S = \{a,c,d,s\}$ ein Schnitt; die Kapazität $c(Q,S)$ dieses Schnitts ist $c((q,a)) + c((b,c)) + c((b,d)) = 11$. Für einen Fluß f und einen Schnitt Q,S ist der (*Netto-*) *Fluß über den Schnitt* $f(Q,S) = \sum_{v\in Q, v'\in S, (v,v')\in E} f((v,v')) - \sum_{v\in Q, v'\in S, (v',v)\in E} f((v',v))$. In unserem Beispiel ist also der Fluß $f(\{q,b\},\{a,c,d,s\}) = f((q,a)) + f((b,c)) + f((b,d)) - f((a,b)) = 3+0+3-3 = 3$. Daß dies gerade dem Wert des Flusses $w(f)$ entspricht, ist kein Zufall.

Ganz allgemein gilt für jeden Fluß f und jeden Schnitt Q,S, daß der Fluß $f(Q,S) = w(f)$ ist. Dies sieht man wie folgt ein. Nach Definition ist

$$f(Q,S) \;=\; \sum_{\substack{v\in Q,\\ v'\in S,\\ (v,v')\in E}} f((v,v')) \;-\; \sum_{\substack{v\in Q,\\ v'\in S,\\ (v',v)\in E}} f((v',v)).$$

Addieren wir zur rechten Seite dieser Gleichung

$$\sum_{\substack{v\in Q,\\ v'\in Q,\\ (v,v')\in E}} f((v,v')) \;-\; \sum_{\substack{v\in Q,\\ v'\in Q,\\ (v,v')\in E}} f((v,v')) \;+\; \sum_{\substack{v\in Q,\\ v'\in Q,\\ (v',v)\in E}} f((v',v)) \;-\; \sum_{\substack{v\in Q,\\ v'\in Q,\\ (v',v)\in E}} f((v',v)),$$

so können wir die Summanden neu zusammenfassen zu

$$f(Q,S) \;=\; \sum_{\substack{v\in Q,\\ v'\in V,\\ (v,v')\in E}} f((v,v')) \;-\; \sum_{\substack{v\in Q,\\ v'\in V,\\ (v',v)\in E}} f((v',v)) \;+\; \sum_{\substack{v\in Q,\\ v'\in Q,\\ (v',v)\in E}} f((v',v)) \;-\; \sum_{\substack{v\in Q,\\ v'\in Q,\\ (v,v')\in E}} f((v,v')).$$

Wegen der Flußerhaltung ergeben die ersten beiden Summanden zusammen gerade $w(f)$; die letzten beiden Summanden ergeben 0, und somit ist die Behauptung nachgewiesen. Für das in Abbildung 8.29 angegebene Beispiel kann man leicht überprüfen, daß der Fluß für jeden Schnitt 3 beträgt.

Wegen der Kapazitätsbeschränkung kann man sofort schließen, daß der Fluß über einen beliebigen Schnitt dessen Kapazität nicht übersteigen kann. Damit ist der Wert eines maximalen Flusses sicher nicht größer als die Kapazität eines minimalen Schnittes; wir werden noch sehen, daß in der Tat beide Werte gleich sind.

Maximaler Fluß durch zunehmende Wege

Für das in Abbildung 8.29 gezeigte Beispiel hat der Fluß seinen maximalen Wert offenbar noch nicht erreicht. Zwar können wir den Fluß entlang des Weges q,a,b,d,s nicht mehr erhöhen, weil Pfeil (b,d) bereits die maximal mögliche Menge transportiert. Aber es gibt noch andere Wege, bei denen die Kapazitäten nicht voll ausgenutzt sind. So lassen sich zum Beispiel entlang des Weges q,a,c,s zwei weitere Einheiten transportieren. Erhöhen wir außerdem den Fluß auf dem Weg q,b,c,s um 3 Einheiten, so erhalten wir die in Abbildung 8.30 gezeigte Situation.

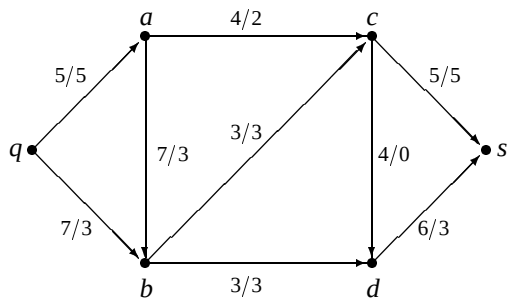


Abbildung 8.30

Jetzt ist auf jedem Weg von q nach s wenigstens ein Pfeil *gesättigt*, d.h., der Fluß auf diesem Pfeil entspricht gerade der Kapazität des Pfeils. Trotzdem ist der Wert des Flusses nur 8, obgleich die Kapazität des minimalen Schnitts $\{q,a,b\},\{c,d,s\}$ 10 beträgt. Der Fluß ist also nicht maximal. Dies haben wir einer unglücklichen Entscheidung im Knoten a zu verdanken: Dort werden drei Flußeinheiten über Knoten b weitergeleitet, wodurch auf dem Pfeil (a,c) nur noch zwei Einheiten transportiert werden müssen. Im Knoten b ergibt sich aber ein Engpaß, weil von ihm aus nur sechs Einheiten weitergeleitet werden können. Es wäre also besser gewesen, zwei Einheiten vom Knoten a über den Knoten c weiterzuleiten und damit Platz zu schaffen für zwei Einheiten, die vom Knoten q über den Knoten b geleitet werden könnten. Von Knoten c aus könnten die zwei Einheiten über Knoten d zum Knoten s gelangen.

Wir können dieses Abändern von Flüssen in Wegen von q nach s ausdrücken, wenn wir nicht nur das Erhöhen eines Flusses entlang eines Pfeiles mit noch freier *Restkapazität* $rest(e) = c(e) - f(e)$ in Betracht ziehen, sondern auch das Verringern eines Flusses entlang eines Pfeiles, also gewissermaßen das Erhöhen eines Flusses entgegen der Pfeilrichtung. Einen Fluß $f(e)$ kann man natürlich höchstens um $f(e)$ Einheiten verringern; dann ergibt sich für $f(e)$ der Wert 0. In unserem Beispiel bedeutet dies gerade, daß wir den Weg q,b,a,c,d,s betrachten und feststellen, daß wir den Fluß durch Pfeil (q,b) um 4 erhöhen, durch (a,b) um 3 senken, durch (a,c) um 2 erhöhen, durch (c,d) um 4 erhöhen und durch (d,s) um 3 Einheiten erhöhen können. Also läßt sich der

Fluß um das Minimum dieser Werte, nämlich 2 erhöhen. Ein solcher Weg ohne Rücksicht auf die Pfeilrichtungen (ein *ungerichteter* Weg) von q nach s, auf dem man den Fluß erhöhen kann, wird *zunehmender Weg* genannt. Für jeden Pfeil e auf einem zunehmenden Weg, der in Pfeilrichtung durchlaufen wird (ein *Vorwärtspfeil*), ist $f(e) < c(e)$, also $rest(e) > 0$; für jeden Pfeil, der in Gegenrichtung durchlaufen wird (ein *Rückwärtspfeil*), gilt $f(e) > 0$. Der *Restgraph* zu einem Fluß f beschreibt gerade alle Flußvergrößerungsmöglichkeiten: Er enthält einen Pfeil e, wenn $rest(e) > 0$ gilt; er enthält den zu e entgegengesetzten Pfeil, wenn $f(e) > 0$ gilt.

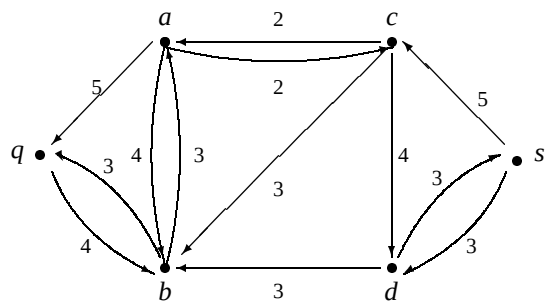


Abbildung 8.31

Abbildung 8.31 zeigt den Restgraphen zu dem in Abbildung 8.30 gezeigten Fluß. Jeder Weg im Restgraphen von q nach s ist ein zunehmender Weg für den gegebenen Fluß. In unserem Beispiel ist der einzige zunehmende Weg der einzige einfache Weg von q nach s im Restgraphen, also der Weg q,b,a,c,d,s. Nach der Flußvergrößerung um 2 Einheiten auf diesem Weg ergibt sich der in Abbildung 8.32 gezeigte Fluß; im zugehörigen Restgraphen führt kein Weg mehr von q nach s. Der Fluß hat den Wert 10, ist also maximal. Wir haben im Restgraphen nur solche Pfeile e eingezeichnet, für die $rest(e) > 0$ gilt, wobei $rest(e)$ genauer wie folgt definiert ist:

$$rest((v,v')) = \begin{cases} c((v,v')) - f((v,v')), & \text{falls } (v,v') \in E \\ f((v',v)) & \text{falls } (v',v) \in E. \end{cases}$$

Bereits 1956 wurde gezeigt [46, 57], daß ein Fluß f genau dann maximal ist, wenn es für f keinen zunehmenden Weg gibt, und daß genau dann der Wert des Flusses f der Kapaziät eines minimalen Schnitts entspricht. Dies sieht man wie folgt ein. Wenn es einen zunehmenden Weg für einen Fluß f gibt, dann können wir den Fluß entlang dieses Wegs vergrößern. Damit ist klar, daß es für einen maximalen Fluß f keinen zunehmenden Weg geben kann. Nehmen wir jetzt also an, daß es für f keinen zunehmenden Weg gibt. Sei X die Menge aller im Restgraphen von q aus erreichbaren Knoten, und sei $\overline{X} = V - X$. Weil es für f keinen zunehmenden Weg gibt, gehört q zu X und s zu $\overline{X}$. Also ist $X, \overline{X}$ ein Schnitt. Nach Definition gibt es im Restgraphen keinen Pfeil von

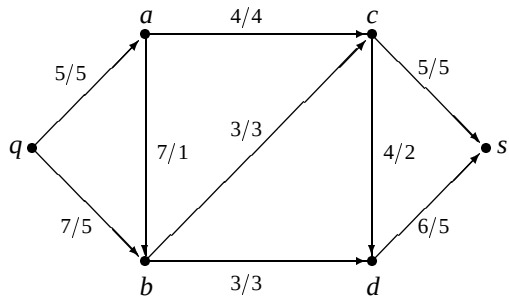


Abbildung 8.32

einem Knoten in X zu einem Knoten in $\overline{X}$. Also gilt $f(e) = c(e)$ für jeden Pfeil e im gegebenen Graphen G, der von einem Knoten in X zu einem Knoten in $\overline{X}$ führt. Damit ist $w(f) = c(X, \overline{X})$; der Wert des Flusses f entspricht also der Kapazität *eines* Schnitts. Der Wert eines jeden Flusses, also auch $w(f)$, ist durch die Kapazität c_{min} eines minimalen Schnitts beschränkt. Wegen $w(f) \leq c_{min}$ und $c_{min} \leq c(X, \overline{X})$ folgt mit $w(f) = c(X, \overline{X})$ auch $w(f) = c_{min} = c(X, \overline{X})$, d.h., $X, \overline{X}$ muß ein minimaler Schnitt und f ein maximaler Fluß sein.

Beliebige zunehmende Wege

Hieraus ergibt sich unmittelbar die in [57] vorgestellte Methode zur Konstruktion eines maximalen Flusses durch wiederholtes Einbeziehen zunehmender Wege:

Algorithmus *Maximaler Fluß durch zunehmende Wege* [57]
{*berechnet zu einem Digraphen* $G = (V, E)$ *mit Kapazität* $c : E \to \mathbb{R}^+$
einen maximalen Fluß $f : E \to \mathbb{R}_0^+$ *für* G}
begin
1. {*Initialisiere mit Nullflu*
for all $e \in E$ **do**
$f(e) := 0$;
2. {*iterierte Flußvergrößerung*:}
while *es gibt einen zunehmenden Weg* p **do**
begin
$r := \min\{rest(e) \mid e$ *liegt auf Weg* p *im Restgraphen*$\}$;
erhöhe f *entlang* p *um* r
end
end {*Maximaler Fluß*}

Hierbei ist es sinnvoll, neben der Kapazität c für jede Kante auch einen aktuellen Flußwert f zu speichern. Das Erhöhen des Flusses f entlang eines Weges p um einen Betrag r wird für Vorwärtspfeile durch Erhöhen von f um r und für Rückwärtspfeile durch Erniedrigen von f um r realisiert.

Genau genommen arbeitet der vorgestellte Algorithmus aber noch nicht einmal korrekt: Man kann sich überlegen, daß er für irrationale Kapazitäten nicht unbedingt terminieren muß und daß aufeinanderfolgende Flußwerte zwar konvergieren, aber nicht unbedingt zum Wert des maximalen Flusses. Beschränken wir jedoch die Kapazitäten auf ganze (oder rationale) Zahlen, so ist der vorgeschlagene Algorithmus korrekt. Bei ganzzahligen Kapazitäten ist auch ein maximaler Fluß ganzzahlig, und bei jedem Durchlauf der **while**-Schleife wird der gefundene Fluß wenigstens um 1 erhöht. Ein maximaler Fluß f_{max} wird also mit höchstens $w(f_{max})$ Durchläufen der **while**-Schleife gefunden. Damit hängt aber die Laufzeit des Algorithmus nicht nur von der Anzahl der Knoten und Kanten des gegebenen Graphen ab. Abbildung 8.33 zeigt einen Beispielgraphen mit vier Knoten und fünf Kanten, bei dem der Algorithmus $2 \cdot c_1$ Durchläufe der **while**-Schleife benötigt, wenn abwechselnd die Wege q, a, b, s und q, b, a, s als zunehmende Wege gewählt werden.

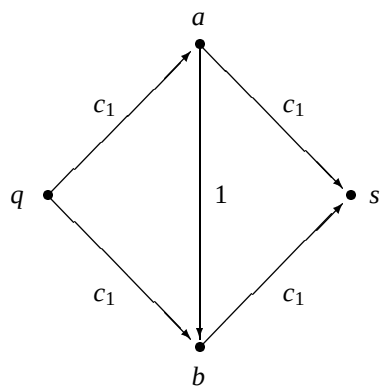


Abbildung 8.33

Kürzeste zunehmende Wege

Eine Laufzeitschranke, die lediglich von der Größe des Graphen abhängt, erhält man, wenn man als zunehmenden Weg immer einen mit möglichst wenigen Pfeilen wählt [45]. Bestimmt man solche kürzesten zunehmenden Wege für die einzelnen Flußvergrößerungsschritte, so vergrößert sich die Anzahl der Pfeile auf einem kürzesten Weg von q nach s nach höchstens $|E|$ Schleifendurchläufen wenigstens um 1. Damit ist die Anzahl der erforderlichen Iterationen beschränkt durch $(|V|-1) \cdot |E|$. Weil man einen einzelnen zunehmenden Weg mittels Breitensuche in $O(|E|)$ Schritten finden kann, ergibt sich eine Laufzeit von insgesamt $O(|V| \cdot |E|^2)$ Schritten für das Berechnen eines maximalen Flusses. Es geht aber noch schneller.

Alle kürzesten zunehmenden Wege

Wir betrachten wiederholt Flüsse, die sich nicht entlang eines Weges im gegebenen Graphen vergrößern lassen. Für den in Abbildung 8.29 gezeigten Graphen ist dies bei dem in Abbildung 8.30 gezeigten Fluß der Fall. Ein solcher Fluß enthält auf jedem Weg von q nach s einen gesättigten Pfeil; wir bezeichnen ihn als *blockierenden Fluß*. Abbildung 8.34 zeigt einen Fluß für den Graphen aus Abbildung 8.29; Abbildung 8.35 zeigt den dazugehörigen Restgraphen. Zur Bestimmung eines kürzesten zunehmenden Weges von q nach s sind nicht alle Pfeile im Restgraphen von Interesse. Vielmehr genügt es, für jeden von q aus erreichbaren Knoten v im Restgraphen einen kürzesten Weg von q nach v zu kennen.

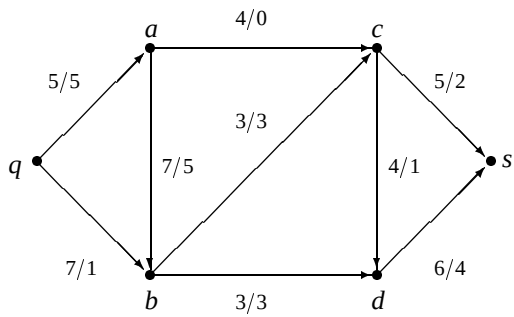


Abbildung 8.34

Für Knoten v bezeichnen wir die Länge (das ist die Anzahl der Pfeile) eines kürzesten Weges von q nach v im Restgraphen als *Niveau* von v.

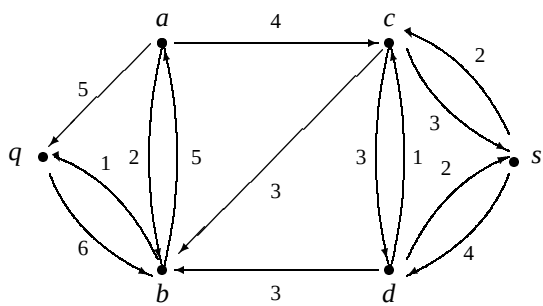


Abbildung 8.35

Für einen Fluß f ist der *Niveaugraph* derjenige Teilgraph des Restgraphen, der nur die von q aus erreichbaren Knoten enthält und nur solche Pfeile, die auf einem kürzesten Weg liegen. Ein Pfeil (v, v') des Restgraphen gehört also genau dann zum Niveaugraphen, wenn $Niveau(v') = Niveau(v) + 1$ gilt. Abbildung 8.36 zeigt den Niveaugraphen zu dem in Abbildung 8.35 gezeigten Fluß.

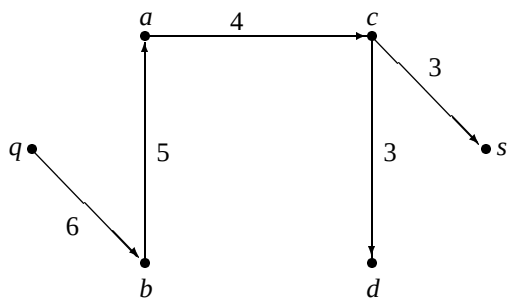


Abbildung 8.36

Der Niveaugraph enthält jeden kürzesten vergrößernden Weg, aber nicht unbedingt jeden vergrößernden Weg. Mit einer Breitensuche kann der Niveaugraph in Zeit $O(|E|)$ konstruiert werden. Damit ergibt sich die folgende Variante des Schritts 2 zur iterierten Flußvergrößerung im Algorithmus *Maximaler Fluß durch zunehmende Wege*:

2. {*iterierte Flußvergrößerung nach Dinic* [37]:}
while *s gehört zum Niveaugraphen für* f **do**
begin
$f_b :=$ *ein blockierender Fluß im Niveaugraphen für* f;
$f := f \oplus f_b$
end

Dabei bezeichnet $\oplus$ das bereits erläuterte Addieren zweier Flüsse unter Berücksichtigung der Pfeilrichtung. Für den in Abbildung 8.36 gezeigten Niveaugraphen ist ein blockierender Fluß der Fluß der Stärke 3 entlang des Weges q, b, a, c, s. Die Addition dieses Flusses zu dem in Abbildung 8.34 gezeigten ergibt den in Abbildung 8.37 gezeigten Fluß. Abbildungen 8.38 bis 8.40 setzen das Beispiel bis zu einem maximalen Fluß und einem Niveaugraphen fort, der s nicht enthält.

Die Anzahl der im Verlauf der Berechnung erforderlichen iterierten Flußvergrößerungen ist vergleichsweise gering. Weil bei jeder Flußvergrößerung ein blockierender Fluß im Niveaugraphen zum aktuellen Fluß hinzugefügt wird, wächst das Niveau der Senke s bei jeder Iteration wenigstens um 1, sofern s von q aus im Niveaugraphen überhaupt erreichbar bleibt. Bei jeder Iteration werden also gleichzeitig alle kürzesten Wege zu einer Flußvergrößerung herangezogen. Damit berechnet der Algorithmus mit

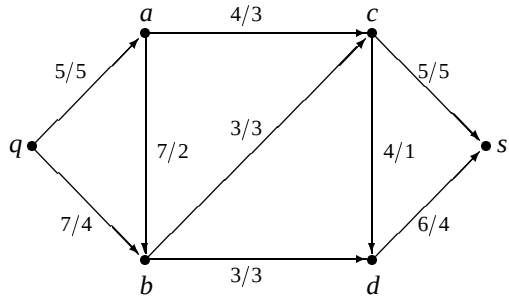


Abbildung 8.37

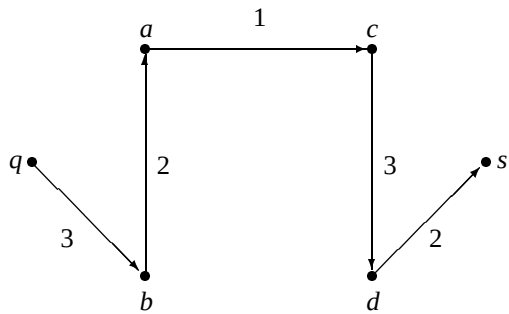


Abbildung 8.38

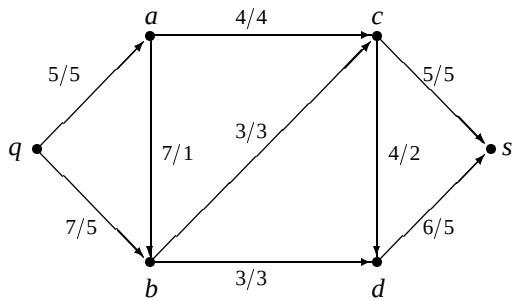


Abbildung 8.39

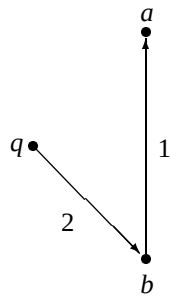


Abbildung 8.40

iterierter Flußvergrößerung nach [37] einen maximalen Fluß mit höchstens $|V|-1$ Iterationen.

In speziellen Fällen kommt dieser Algorithmus sogar mit weniger Iterationen aus. Man kann sich überlegen [50], daß für ein Netzwerk mit ganzzahligen Kapazitäten, in dem außer der Quelle und der Senke jeder Knoten genau einen einmündenden Pfeil mit Kapazität 1 (und beliebig viele ausgehende Pfeile) oder einen ausgehenden Pfeil mit Kapazität 1 (und beliebig viele einmündende Pfeile) hat, $2\left\lceil\sqrt{|V|-2}\right\rceil$ Iterationen genügen. Wir werden dieses spezielle Ergebnis im nächsten Abschnitt zu einer Laufzeitabschätzung einsetzen.

Wir müssen uns jetzt noch überlegen, wie man einen blockierenden Fluß schnell findet. Beim einfachsten Verfahren [37] wählt man einen Weg von q nach s und erhöht auf diesem Weg den Fluß so, daß einer der Pfeile gesättigt wird. Dann entfernt man alle gesättigten Pfeile. Dies wird solange wiederholt, bis s nicht mehr von q aus erreichbar ist. Sobald dies der Fall ist, ist auf jedem Weg von q nach s ein Pfeil gesättigt, also ein blockierender Fluß erreicht.

Das Finden eines Weges von q nach s kann man als Tiefensuche organisieren. Inspizierte Pfeile, die schließlich nicht zu einem Weg zu s gehören, werden gelöscht. Wenn man einen Pfeil betrachtet hat, so gehört dieser also entweder zu einem Weg von q nach s oder er wird gelöscht. Für einen gefundenen Weg, der höchstens aus $|V|-1$ Pfeilen bestehen kann, wird der Wert der Flußvergrößerung als kleinste Restkapazität von Pfeilen auf diesem Weg ermittelt. Beim Durchführen der Flußvergrößerung müssen alle Restkapazitäten von Pfeilen auf dem gefundenen Weg angepaßt und wenigstens ein Pfeil entfernt werden. Weil bei jeder Flußvergrößerung wenigstens ein Pfeil aus dem verbleibenden Graphen entfernt wird, entsteht nach höchstens $|E|$ Flußvergrößerungen ein blockierender Fluß. Da insgesamt höchstens jede Kante einmal gelöscht wird und jede Flußvergrößerung in $O(|V|)$ Schritten durchgeführt werden kann, findet der Algorithmus von Dinic [37] einen blockierenden Fluß in höchstens $O(|V|\cdot|E|)$ Schritten und damit einen maximalen Fluß in höchstens $O(|V|^2\cdot|E|)$ Schritten. Im oben erwähnten

Spezialfall [50] findet der Algorithmus von Dinic [37] einen blockierenden Fluß in Zeit $O(|E|)$ und einen maximalen Fluß in Zeit $O(\sqrt{|V|} \cdot |E|)$.

In letzter Zeit sind einige weitere Methoden vorgeschlagen worden, einen blockierenden Fluß zu berechnen. Ein Verfahren, bei dem man einen Knoten nach dem anderen sättigt — und nicht, wie bei Dinic, einen Pfeil nach dem anderen — ist in [85] erstmals vorgestellt worden. Später wurde diese Methode in [180] vereinfacht. Man kann hierbei einen Knoten in Zeit $O(|V|)$ sättigen; die Konstruktion eines blockierenden Flusses kostet also nu[illegible]h $O(|V|^2)$ Schritte, und ein maxi[illegible] Fluß kann in $O(|V|^3)$ Schritten ermittelt werden.

Eine andere Realisierung der Grundidee, Knoten zu sättigen, ist in [115] vorgeschlagen worden. Hier merkt man sich für jeden Knoten v den maximal zusätzlich noch möglichen *Durchsatz* durch Knoten v. So kann man etwa im Beispiel der Ab[illegible]ng 8.34 den Durchsatz nur für die Knoten c und d erhöhen, bei Knoten a alle einmündenden Pfeile und bei Knoten b alle ausgehenden Pfeile gesättigt sind. Der Durchsatz bei Knoten c kann um 4 Einheiten erhöht werden, weil sowohl vier zusätzliche Einheiten von a nach c als auch von c weg, nach d und s, fließen können, wenn man den Rest des Netzwerks außer Betracht läßt. Einen blockierenden Fluß findet man [illegible] indem man wiederholt über einen Knoten mit kleinstem maximal möglichen zusätzlichen Durchsatz gerade soviele Einheiten von der Quelle q zur Senke s schickt, wie dieser Durchsatz angibt. Bei geeigneter Implementierung kommt dieses Verfahren ebenfalls mit $O(|V|^3)$ Schritten aus.

In anderen Verfahren [64, 169] wurde versucht, einen Pfeil nach dem anderen zu sättigen und die Laufzeit des Verfahrens durch Verwendung einer geeigneten Datenstruktur zu reduzieren. Der schnellste dieser Philosophie folgende Algorithmus [171] verwendet eine Datenstruktur für dynamische Bäume. Jeder Baumknoten speichert eine reelle Zahl, die *Kosten* des Knotens. Die vorgeschlagene Datenstruktur bietet für eine Menge knotendisjunkter Bäume die folgenden Operationen an:

– *maketree*(v) : stellt einen neuen Baum her, dessen einziger Knoten v mit Kosten 0 ist.
– *findroot*(v) : liefert die Wurzel des Baumes, der Knoten v enthält.
– *findcost*(v) : liefert den Knoten v' und seine Kosten c, wobei c das M[illegible]num der Kosten aller Knoten auf dem Pfad von v zur Wurzel *findroot*(v) ist und v' auf diesem Pfad der am nächsten bei der Wurzel liegende Knoten mit Kosten c ist.
– *addcost*(v,c) : addiere c zu den Kosten jedes Knotens auf dem Pfad von v zur Wurzel *findroot*(v).
– *link*(v,v') : verbinde die beiden Bäume mit Knoten v und v' durch einen Pfeil (v',v). Hier wird angenommen, daß v die Wurzel des einen Baumes ist, und daß v und v' nicht im selben Baum liegen.
– *cut*(v) : teile den Baum, der Knoten v enthält, durch Entfernen der Kante, die v mit dem Vater von v verbindet, in zwei Bäume. Hier wird angenommen, daß v keine Wurzel ist.

Um einen blockierenden Fluß zu finden, speichert man für jeden Knoten einen inzidenten Pfeil, auf dem man möglicherweise den Fluß vergrößern kann. Diese Pfeile zusammen ergeben im Graphen eine Menge von Bäumen. Für $|V|$ insgesamt verwaltete Knoten kann jede der sechs angebotenen dynamischen Baumoperationen in einer amortisierten Laufzeit von $O(\log|V|)$ ausgeführt werden, wobei sich die Folge der aus-

zuführenden Operationen durch eine Umformulierung von Dinics Algorithmus ergibt. Mit $O(|E|)$ Baumoperationen kostet das Berechnen eines blockierenden Flusses dann $O(|E|\log|V|)$ Schritte; ein maximaler Fluß kann also in Zeit $O(|V|\cdot|E|\log|V|)$ berechnet werden.

8.8 Zuordnungsprobleme

Zuordnungsprobleme, bei denen es um eine insgesamt bestmögliche Bildung von Paaren von Elementen über einer Grundmenge geht, lassen sich oft günstig durch Graphen repräsentieren. Die Elemente der Grundmenge sind die Knoten des Graphen und die Kanten beschreiben alle möglichen Paarbildungen. Repräsentiert beispielsweise jeder Knoten einen Teilnehmer an einer Gruppenreise und jede Kante die Bereitschaft der beiden Teilnehmer, in einem gemeinsamen Doppelzimmer zu übernachten, so kann man sich fragen, wieviele Zimmer unter dieser Voraussetzung mindestens benötigt werden. Weil jeder Teilnehmer nur in *einem* Doppelzimmer übernachten soll, ist dies im Graphen die Frage nach einer größtmöglichen Teilmenge der Kanten, bei der jeder Knoten des Graphen mit höchstens einer Kante inzidiert. In dem in Abbildung 8.41 gezeigten Fall sieht man, daß für die sechs Reiseteilnehmer drei Doppelzimmer genügen.

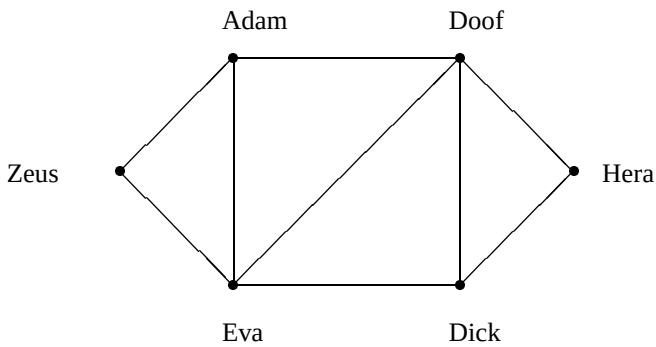


Abbildung 8.41

Für einen ungerichteten Graphen $G=(V,E)$ ist eine *Zuordnung* Z (englisch: *matching*) eine Teilmenge der Kanten von G, so daß keine zwei Kanten in Z denselben Endknoten haben. Die Anzahl $|Z|$ der Kanten in Z heißt *Größe* der Zuordnung. Ein Knoten ist bezüglich einer Zuordnung Z *alleine* (englisch: *unmatched*), wenn er nicht Endknoten einer Kante in Z ist. Z ist eine *perfekte Zuordnung* (englisch: *perfect matching*), wenn mit Z kein Knoten alleine bleibt. In dem in Abbildung 8.41 gezeigten Beispiel gibt es gleich mehrere perfekte Zuordnungen, darunter beispielsweise {(Zeus,

Eva), (Adam, Doof), (Dick, Hera)}. Da es eine perfekte Zuordnung für einen gegebenen Graphen nicht unbedingt geben muß, interessiert man sich für bestmögliche Zuordnungen. Eine Zuordnung Z für einen Graphen $G = (V,E)$ ist *nicht erweiterbar* (englisch: *maximal*), wenn es keine Kante $e \in E$ gibt, die man noch zu Z hinzunehmen könnte, für die also $Z \cup \{e\}$ eine Zuordnung für G bleibt. In unserem Beispiel ist etwa die Zuordnung {(Adam, Eva), (Dick, Doof)} nicht erweiterbar; trotzdem gibt es im Graphen eine Zuordnung, die mehr Kanten enthält. Eine Zuordnung Z mit maximaler Größe $|Z|$ ist eine *maximale Zuordnung* (englisch: *maximum matching*).

Beim Versuch, die Realität etwas genauer zu modellieren, wird man im Beispiel der Abbildung 8.41 vielleicht feststellen, daß Adam zwar bereit ist, ein Doppelzimmer mit Zeus, Eva oder Doof zu teilen, daß ihm aber nicht jede dieser Möglichkeiten gleich lieb ist. Ordnet man nun jeder Kante im Graphen eine Maßzahl für die Zufriedenheit der beiden Reiseteilnehmer bei einer gemeinsamen Übernachtung zu, so ergibt sich etwa die in Abbildung 8.42 gezeigte Situation. Hier können wir nach einer Zuordnung fragen, die die Summe der Zufriedenheiten maximiert. Das ist offenbar die Zuordnung {(Adam, Eva), (Dick, Doof)}, auch wenn dabei Zeus und Hera alleine bleiben.

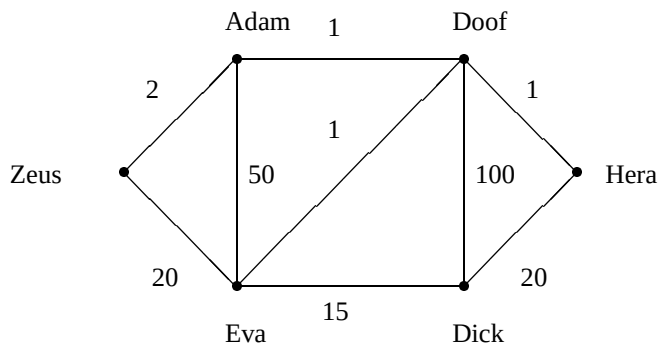


Abbildung 8.42

Für einen ungerichteten, bewerteten Graphen $G = (V,E)$ mit Kantenbewertung $w : E \to \mathbb{R}$ ist das *Gewicht* (englisch: *weight*) einer Zuordnung Z die Summe der Gewichte der Kanten in Z. Wir interessieren uns hier für eine *maximale gewichtete Zuordnung* (englisch: *maximum weight matching*), also eine Zuordnung mit maximalem m Gewicht. Wenn beispielsweise in einer Firma mit k Mitarbeitern $m_1, \ldots, m_k$ die k Tätigkeiten $t_1, \ldots, t_k$ auszuführen sind und eine Maßzahl $w(m_i, t_j)$ für die Eignung des Mitarbeiters m_i für Tätigkeit t_j bekannt ist, sofern Mitarbeiter m_i Tätigkeit t_j überhaupt ausführen kann, so kann eine maximale gewichtete Zuordnung von Mitarbeitern und Tätigkeiten erwünscht sein. Abbildung 8.43 zeigt eine Situation, in der die Zuordnung $\{(m_1,t_1),(m_2,t_3),(m_3,t_2),(m_4,t_5),(m_5,t_4),(m_6,t_6)\}$ maximales Gewicht hat.

Wie in diesem Beispiel lassen sich auch in vielen anderen Fällen die Knoten des Graphen so in zwei Gruppen teilen, daß es nur Kanten zwischen Knoten verschiedener Gruppen gibt. In unserem Beispiel ist es etwa unsinnig, von der Eignung eines

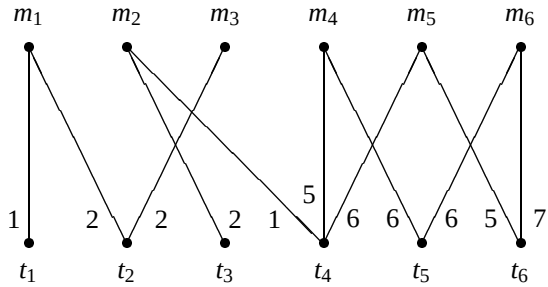


Abbildung 8.43

Mitarbeiters für einen anderen Mitarbeiter oder einer Tätigkeit für eine andere Tätigkeit zu reden. Das Entsprechende gilt beispielsweise, wenn es um die Zuordnung von Studienanfängern zu Studienplätzen oder von Männern zu Frauen bei einem Eheanbahnungsinstitut geht. Weil man in solchen Situationen eine maximale Zuordnung oder eine maximale gewichtete Zuordnung schneller und einfacher finden kann, wollen wir diese separat betrachten. Wir nennen einen Graphen $G = (V,E)$ *bipartit* (englisch: *bipartite*), wenn sich die Knotenmenge V so in zwei Teilmengen X und Y zerlegen läßt (also $V = X \cup Y$ und $X \cap Y = \emptyset$ gilt), daß $E \subseteq X \times Y$, also keine Kante zwei Knoten in X oder zwei Knoten in Y verbindet.

8.8.1 Maximale Zuordnungen in bipartiten Graphen

Betrachten wir zunächst bipartite Graphen ohne Gewichtsfunktion. Abbildung 8.44 zeigt einen solchen Graphen $G = (X \cup Y, E)$ mit $X = \{x_1, \ldots, x_6\}$ und $Y = \{y_1, \ldots, y_6\}$ sowie eine Zuordnung, ausgedrückt durch dicker gezeichnete Kanten.

Man sieht leicht, daß diese Zuordnung nicht maximal ist: Eine Zuordnung mit mehr Kanten erhält man beispielsweise, indem man die Paare (x_1,y_1) und (x_3,y_2) anstatt (x_1,y_2) in die Zuordnung aufnimmt. Um aus einer gegebenen Zuordnung eine maximale Zuordnung zu ermitteln, kann es also nötig sein, eine für die Zuordnung bereits gewählte Kante wieder aus der Zuordnung zu entfernen. Das Entfernen von Kanten aus einer bereits gefundenen Zuordnung läßt sich aber nicht auf einzelne Kanten beschränken. So kann man die in Abbildung 8.44 dargestellte Zuordnung nicht vergrößern, indem man eine einzelne der Kanten (x_2,y_4) oder (x_4,y_5) entfernt und danach möglichst viele Kanten zur Zuordnung hinzunimmt, aber man kann die Zuordnung vergrößern, wenn man diese beiden Kanten aus der Zuordnung entfernt und statt dessen die Kanten $(x_2,y_3), (x_4,y_4)$ und (x_6,y_5) in die Zuordnung aufnimmt. Dies erinnert an das Konzept der *zunehmenden Wege* bei den im Abschnitt 8.7 vorgestellten Algorithmen zum Finden maximaler Flüsse.

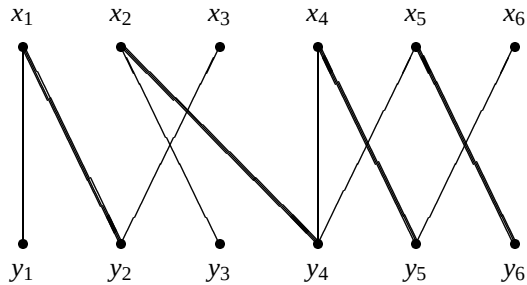


Abbildung 8.44

In der Tat kann man das Zuordnungsproblem für bipartite Graphen als Flußproblem formulieren. Dazu statten wir die Knotenmenge mit zwei zusätzlichen Knoten aus, einer Quelle q und einer Senke s. Jede Kante (x_i,y_j) des Graphen $G = (X \cup Y, E)$ wird im Graphen $G' = (X \cup Y \cup \{q,s\}, E')$ zu einem Pfeil von x_i nach y_j. Außerdem gibt es von q einen Pfeil zu jedem Knoten $x_i \in X$ und von jedem Knoten $y_j \in Y$ einen Pfeil nach s. Es ist also $E' = E \cup \{(q,x) |\ x \in X\} \cup \{(y,s) |\ y \in Y\}$, wobei die Kanten aus E wie beschrieben zu Pfeilen werden.

Abbildung 8.45 zeigt den zum Graphen G in Abbildung 8.44 gehörenden Flußgraphen G' und den Fluß für die dort gezeigte Zuordnung. Als Kapazitätsfunktion wählen wir hierbei $c : E' \to \{1\}$. Man sieht in diesem Beispiel sofort, daß der Ablösung von (x_1,y_2) in der dargestellten Zuordnung durch (x_1,y_1) und (x_3,y_2) ein zunehmender Weg von q nach s entspricht, nämlich der Weg q,x_3,y_2,x_1,y_1,s. Jedem Fluß f in G' entspricht eine Zuordnung $Z = \{(x_i,y_j) | f((x_i,y_j)) = 1\}$ in G, wobei $|Z| = w(f)$ gilt. Ebenso entspricht jede Zuordnung Z in G durch Hinzunahme der Pfeile (q,x) und (y,s) für alle $(x,y) \in Z$ einem Fluß f in G', für den $|Z| = w(f)$ gilt. Eine maximale Zuordnung in G entspricht also einem maximalen Fluß in G'. Somit können wir im bipartiten Graphen G eine maximale Zuordnung berechnen, indem wir in G' einen maximalen Fluß bestimmen. Dies ist, wie wir in Abschnitt 8.7 bereits gesehen haben, für Graphen der speziellen Art von G' in Zeit $O(\sqrt{|V|} \cdot |E|)$ möglich [37].

Wir können das Konzept zunehmender Wege in G' in ein entsprechendes Konzept für G übertragen. Dazu genügt die Feststellung, daß auf einem zunehmenden Weg in G' jeder Vorwärtspfeil e den aktuellen Fluß $f(e) = 0$ und jeder Rückwärtspfeil e' den aktuellen Fluß $f(e') = 1$ transportiert. Einem zunehmenden Weg $q,x_i,\ldots,y_j,s$ in G' entspricht in G ein Weg $x_i,\ldots,y_j$. Weil dieser Weg in G' mit einem Vorwärtspfeil beginnt und mit einem Vorwärtspfeil endet und sich Vorwärtspfeile und Rückwärtspfeile stets abwechseln, ist die Anzahl der Vorwärtspfeile auf diesem Weg um 1 größer als die Anzahl der Rückwärtspfeile. So enthält im Beispiel der Abbildung 8.45 der Weg x_6,y_5,x_4,y_4,x_2,y_3 die Vorwärtspfeile $(x_6,y_5),(x_4,y_4)$ und (x_2,y_3) und die Rückwärtspfeile (x_4,y_5) und (x_2,y_4). Einem solchen Weg in G' entspricht in G ein Weg, der abwechselnd aus Kanten besteht, die zur Zuordnung gehören bzw. nicht zur Zuordnung

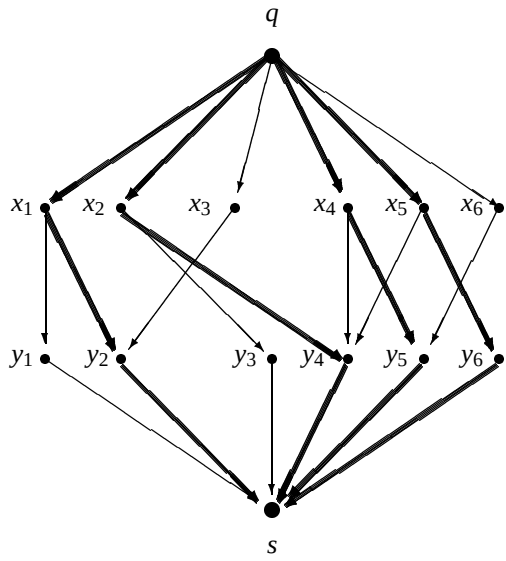


Abbildung 8.45

gehören. Solche Wege spielen bei Zuordnungen die Rolle, die zunehmende Wege bei Flüssen spielen.

Für eine gegebene Zuordnung Z nennen wir jede für die Zuordnung verwendete Kante $e \in Z$ *gebunden*; jede Kante $e' \in E - Z$ ist *frei*. Jeder Knoten, der mit einer gebundenen Kante inzidiert, ist ein *gebundener Knoten*, jeder andere Knoten ist *frei*. Ein Weg in G, dessen Kanten abwechselnd gebunden und frei sind, heißt *alternierender Weg*. Die *Länge* eines alternierenden Wegs ist die Anzahl der Kanten auf diesem Weg. Natürlich kann nicht jeder alternierende Weg zur Vergrößerung einer Zuordnung benützt werden. Dies geht nur dann, wenn die beiden Knoten an den beiden Enden des Wegs frei sind. Ein alternierender Weg mit zwei freien Knoten an den beiden Enden heißt deshalb *vergrößernd*. So sind im Beispiel der Abbildung 8.44 die Wege x_6, y_5 und x_2, y_4, x_5, y_6 zwar alternierend, aber nicht vergrößernd; der alternierende Weg $y_3, x_2, y_4, x_4, y_5, x_6$ ist dagegen vergrößernd. Aus einer Zuordnung, die einen vergrößernden Weg besitzt, kann man offensichtlich eine größere Zuordnung gewinnen, indem man entlang des vergrößernden Weges jede freie Kante zu einer gebundenen und jede gebundene zu einer freien Kante macht. Im Beispiel der Abbildung 8.44 kann man also die Kanten $(x_2, y_3), (x_4, y_4)$ und (x_6, y_5) zu gebundenen Kanten und die Kanten (x_2, y_4) und (x_4, y_5) zu freien Kanten machen und somit die Größe der gezeigten Zuordnung um 1 erhöhen. Das Konzept vergrößernder Wege kann man auch in allgemeinen, also nicht bipartiten, Graphen einsetzen.

8.8.2 Maximale Zuordnungen im allgemeinen Fall

Besitzt eine Zuordnung Z in einem Graphen $G = (V,E)$ einen vergrößernden Weg, so ist Z nicht von maximaler Größe. In dem in Abbildung 8.41 gezeigten Beispiel besitzt die Zuordnung {(Adam, Eva), (Dick, Doof)} gleich mehrere vergrößernde Wege, darunter den Weg Zeus, Eva, Adam, Doof, Dick, Hera. Macht man auf diesem Weg alle gebundenen Kanten zu freien Kanten und alle freien Kanten zu gebundenen Kanten, so erhält man die vergrößerte Zuordnung {(Zeus, Eva), (Adam, Doof), (Dick, Hera)}. Im gezeigten Beispiel ist dies sogar eine maximale Zuordnung.

Daß man mit vergrößernden Wegen schließlich auch wirklich eine maximale Zuordnung erreicht, zeigt folgende Überlegung. Sei Z eine beliebige Zuordnung und Z_{max} eine größte Zuordnung für einen gegebenen Graphen; sei $k = |Z_{max}| - |Z|$ der Unterschied in der Größe beider Zuordnungen. Für den in Abbildung 8.46 gezeigten Graphen ist beispielsweise $Z_{max} = \{(1,2),(3,4),(5,8),(6,7),(9,12),(10,11)\}$.

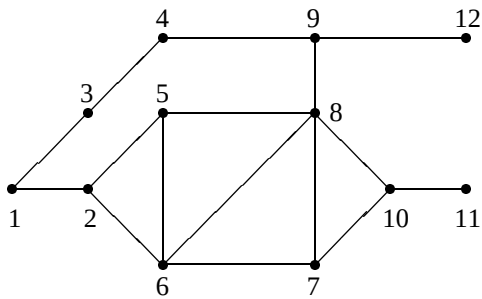


Abbildung 8.46

Für $Z = \{(4,9),(5,6),(7,8),(10,11)\}$ ergibt sich $k = |Z_{max}| - |Z| = 6 - 4 = 2$. Betrachten wir nun die symmetrische Differenz Z_{sym} von Z_{max} und Z, also $Z_{sym} = (Z_{max} - Z) \cup (Z - Z_{max})$. Im gezeigten Beispiel ergibt sich $Z_{sym} = \{(1,2), (3,4), (4,9), (5,6), (5,8), (6,7), (7,8), (9,12)\}$. Jeder Knoten des Graphen inzidiert mit höchstens zwei Kanten in Z_{sym}, nämlich höchstens einer von Z und einer von Z_{max}. In unserem Beispiel inzidieren gerade die Knoten 4, 5, 6, 7, 8 und 9 mit jeweils zwei Kanten. Der durch Z_{sym} induzierte Teilgraph von G kann keinen Zyklus ungerader Länge enthalten, weil jede Kante des Zyklus aus Z_{max} oder aus Z kommen muß und sich im Zyklus Kanten von Z_{max} mit Kanten von Z abwechseln müssen. Der durch Z_{sym} induzierte Teilgraph kann also nur Zyklen gerader Länge und natürlich Wege beliebiger Länge enthalten. Auf jedem Weg und in jedem Zyklus müssen die Kanten bezüglich Z alternieren, d.h. abwechselnd gebunden und frei sein; dasselbe gilt natürlich für Z_{max}.

Weil Z_{max} k Kanten mehr enthält als Z, und in Z_{sym} alle Kanten von $Z_{max} \cup Z$ außer den gemeinsamen Kanten enthalten sind, ist in Z_{sym} die Anzahl der aus Z_{max} stammenden Kanten um k höher als die Anzahl der aus Z stammenden Kanten. In unserem Beispiel stammen fünf der acht Kanten in Z_{sym} aus Z_{max}, das sind $k = 2$ Kanten mehr als aus Z.

Da jeder Zyklus in Z_{sym} genauso viele Kanten aus Z_{max} wie aus Z enthält, müssen auf Wegen (ohne Zyklen) in Z_{sym} k Kanten mehr aus Z_{max} stammen als aus Z. Daher muß es in Z_{sym} wenigstens k Wege geben, die mit einer Kante aus Z_{max} beginnen und mit einer solchen enden, und auf denen Kanten aus Z_{max} und aus Z alternieren. In unserem Beispiel gibt es zwei solche Wege, nämlich den Weg 1, 2 und den Weg 3, 4, 9, 12. Weil Z_{max} eine Zuordnung ist, können solche Wege keine gemeinsamen Knoten haben. All diese alternierenden Wege sind also knotendisjunkt und vergrößernd für Z, weil beide Endknoten bezüglich Z frei sind. Weil Z_{max} und Z Zuordnungen sind, ist die Summe der Längen aller solchen Wege durch die Anzahl der Knoten des Graphen beschränkt. Bei wenigstens k knotendisjunkten Wegen hat also wenigstens ein solcher Weg höchstens die Länge $|V|/k-1$.

Wir können also jetzt eine beliebige, aber noch nicht maximale Zuordnung vergrößern, indem wir vergrößernde Wege finden und die Zuordnung entsprechend anpassen. Bei bipartiten Graphen kann man für eine gegebene Zuordnung Z einen vergrößernden Weg finden, indem man mit der Suche bei einem freien Knoten beginnt und entlang eines bezüglich Z alternierenden Weges fortschreitet. Sobald man wieder bei einem freien Knoten angekommen ist, ist ein vergrößernder Weg gefunden. Zu einem freien Startknoten kann man einen entsprechenden alternierenden Baum mit Hilfe einer Breitensuche ermitteln. Abbildung 8.47 zeigt einen alternierenden Breitensuchbaum für die in Abbildung 8.44 gezeigte Zuordnung und den Startknoten y_3 der Breitensuche.

In allgemeinen Graphen kann man mit einer solch einfachen Breitensuche vergrößernde Wege nicht unbedingt finden. Betrachten wir als Beispiel den in Abbildung 8.46 gezeigten Graphen und die Zuordnung $Z = \{(6,7),(8,10)\}$ und versuchen wir nun, vom freien Knoten 2 aus mit Hilfe eines alternierenden Baums einen vergrößernden Weg zu finden. Wenn wir den alternierenden Baum auf einen Teilgraphen beschränken, so hat er beispielsweise die in Abbildung 8.48 gezeigte Gestalt. Die Breitensuche sorgt dafür, daß Knoten 10 besucht wird, bevor die Nachfolger von Knoten 8 im alternierenden Baum in Betracht gezogen werden. Wenn jeder Knoten, wie bei der Breitensuche üblich, nur einmal besucht werden darf, so verhindert das Finden des alternierenden Weges 2, 6, 7, 10, der nicht mit einem freien Knoten endet, daß der alternierende Weg 2, 6, 7, 8, 10, 11 gefunden wird, obwohl dieser mit einem freien Knoten enden würde. Die reine Breitensuche ist also hier nicht in der Lage, vergrößernde Wege auch wirklich zu finden.

Die Ursache des Problems liegt darin, daß ein und derselbe Knoten auf mehreren verschiedenen alternierenden Wegen in gerader und in ungerader Entfernung vom Startknoten auftreten kann. So tritt in unserem Beispiel Knoten 10 auf dem alternierenden Weg 2, 6, 7, 10 in ungerader Entfernung vom Startknoten 2 auf, während er auf dem alternierenden Weg 2, 6, 7, 8, 10 in gerader Entfernung vom Startknoten auftritt. Man kann aber nicht einfach in einer Abänderung der reinen Breitensuche das zweimalige Besuchen eines jeden Knotens erlauben, nämlich je einmal für die gerade und einmal für die ungerade Entfernung vom Startknoten, denn dann können auch Knotenfolgen gefunden werden, die keinen vergrößernden Weg beschreiben. Eine entsprechend modifizierte Breitensuche kann für den in Abbildung 8.46 gezeigten Graphen und die Zuordnung $Z = \{(6,7),(8,10)\}$ für Startknoten 2 die Knotenfolge 2, 6, 7, 8, 10, 7, 6, 5 liefern, obwohl diese Knotenfolge keinen vergrößernden Weg beschreibt.

Man kann sich überlegen, daß das Finden eines vergrößernden Weges von einem freien Knoten v aus nur dann schwierig ist, wenn es einen alternierenden Weg p von v

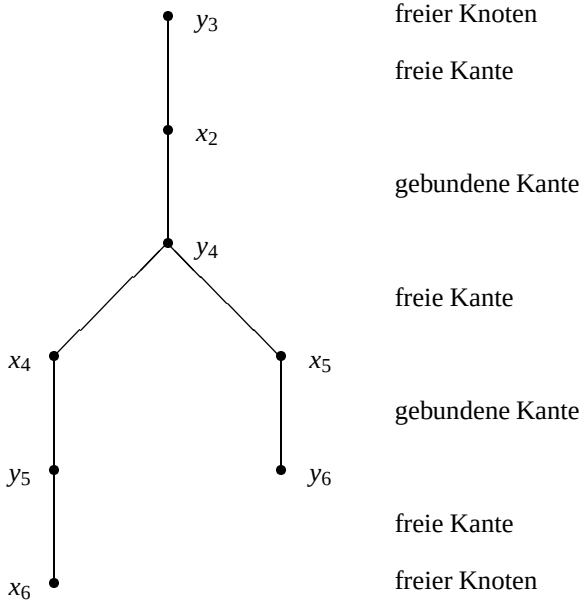


Abbildung 8.47

zu einem Knoten v' in gerader Entfernung von v gibt, und wenn eine Kante v' mit einem anderen Knoten v'' verbindet, der auf dem Weg p ebenfalls in gerader Entfernung von v liegt (vgl. Abbildung 8.49).

Der Teil des Weges p von v'' nach v' heißt zusammen mit der Kante (v', v'') *Blüte*; eine Blüte ist also ein Zyklus ungerader Länge. Knoten v'' heißt *Basis* der Blüte. Der Teil des Weges p von v nach v'' heißt *Stiel* der Blüte.

In dem in Abbildung 8.49 gezeigten Beispiel gibt es sowohl einen alternierenden Weg von v nach i als auch einen alternierenden Weg von v nach j. Den ersteren erhält man, wenn man im Zyklus ungerader Länge im Uhrzeigersinn fortschreitet, den letzteren erhält man durch Besuchen einiger Knoten des Zyklus entgegen dem Uhrzeigersinn. Diese beiden Wege kann man finden, wenn man die Blüte auf einen Knoten schrumpfen läßt, also den Zyklus ungerader Länge in einen Knoten kollabiert. Jede Kante, die vor dem Schrumpfen mit einem Knoten des Zyklus inzident war, ist nach dem Schrumpfen mit dem die Blüte repräsentierenden Knoten inzident. Abbildung 8.50 zeigt den Effekt des Schrumpfens der Blüte für die in Abbildung 8.49 gezeigte Situation.

Wenn ein Graph G' aus einem Graphen G durch Schrumpfen einer Blüte entsteht, so gibt es in G' genau dann einen vergrößernden Weg, wenn es einen solchen in G gibt [44]. Davon kann man sich wie folgt überzeugen. Schließen wir zunächst aus der Existenz eines vergrößernden Weges in G' auf die Existenz eines solchen Weges in G. Dies ist offensichtlich, wenn ein in G' betrachteter vergrößernder Weg die Blüte nicht

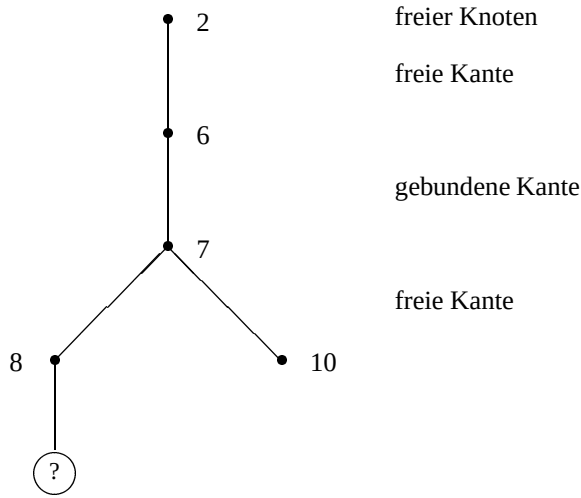


Abbildung 8.48

enthält. Enthält dagegen der betrachtete Weg in G' den Knoten b, der die geschrumpfte Blüte repräsentiert, so expandieren wir b zur vollen Blüte. Falls der betrachtete Weg nur einen Knoten der Blüte passiert, bleibt er erhalten, wenn b durch diesen Knoten ersetzt wird (vgl. Abbildung 8.51 (a)). Falls der betrachtete Weg jedoch mehr als einen Knoten der Blüte passiert, so eignet sich genau einer der beiden möglichen Wege durch einen Teil der Blüte als Verbindung zwischen den beiden Teilen des zerfallenen Weges (vgl. Abbildung 8.51 (b)).

Der Schluß auf die Existenz eines vergrößernden Weges in G' aus der Existenz eines solchen Weges in G ist schwieriger. Wir führen den Nachweis indirekt, indem wir einen

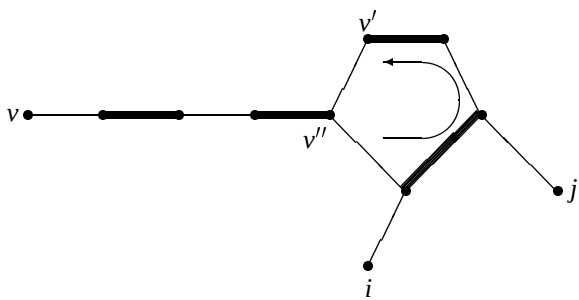


Abbildung 8.49

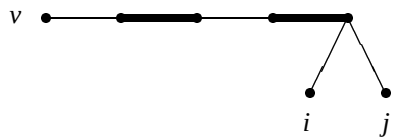
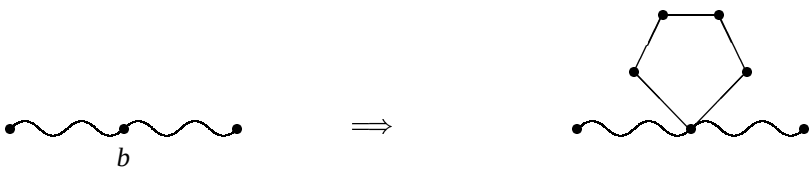


Abbildung 8.50

Algorithmus angeben, der einen vergrößernden Weg mit Hilfe des Schrumpfens von Blüten findet.



(a)

(b)

Abbildung 8.51

Der von Edmonds [44] vorgeschlagene Algorithmus beginnt das Durchlaufen eines Graphen bei einem freien Knoten und konstruiert dabei einen Wald von Bäumen mit alternierenden Wegen. Sowie eine Blüte entdeckt ist, wird sie zu einem Knoten geschrumpft. Zum Zwecke des Durchlaufens des Graphen ersetzen wir jede Kante (v,v') durch die beiden Pfeile (v,v') und (v',v). Jeder Knoten hat stets einen von drei Zuständen: Er ist entweder *unerreicht*, *gerade* oder *ungerade*. Zu jedem Knoten v merken wir uns dessen Vorgänger $N(v)$ beim Durchlaufen des Graphen. Für einen gebundenen Knoten v bezeichnet $Partner(v)$ denjenigen Knoten, der mit derselben gebundenen Kante inzidiert wie v. Dann findet der folgende Algorithmus einen vergrößernden Weg in G', wenn es einen solchen in G gibt:

Algorithmus *Vergrößernder Weg* [44]
{*liefert zu einem Digraphen* $G=(V,E)$ *und einer Zuordnung* $Z \subseteq E$
einen vergrößernden Weg in G bezüglich Z, falls es einen solchen

```
gibt}
begin
  1.  {Initialisiere:}
      for all v ∈ V, v frei bezüglich Z, do
        v.Zustand := gerade;
      for all v ∈ V, v gebunden bezüglich Z, do
        v.Zustand := unerreicht;
  2.  {Suche vergrößernden Weg:}
      repeat {prüfe einen Pfeil:}
        wähle einen noch nicht untersuchten Pfeil (v,v'),
          für den v.Zustand = gerade ist;
        case v'.Zustand of
        ungerade : {Fall 1} tue nichts;
        unerreicht : begin {Fall 2}
                       v'.Zustand := ungerade;
                       Partner(v').Zustand := gerade;
                       p(v') := v;
                       p(Partner(v')) := v';
                     end;
        gerade :     if v und v' sind im selben Baum then
                       begin {Fall 3}
                         v'' := nächster gemeinsamer Vorfahr
                           von v und v' im Baum;
                         schrumpfe die Blüte v,v',...,v'',...,v in
                           den Knoten v'' und passe dabei p an
                       end
                     else {Fall 4}
                       verbinde v und v'
                       {dies ergibt vergrößernden Weg zwischen den
                        Wurzeln der Bäume, die v und v' enthalten}
      until  v'.Zustand = gerade und v und v' sind nicht
               im selben Baum  {Fall 4 ist aufgetreten}
             or kein Pfeil (v,v') mit v.Zustand = gerade ist
               noch nicht untersucht
end {Vergrößernder Weg}
```

Die entscheidenden Aktionen im Algorithmus finden in den mit Fall 2, Fall 3 und Fall 4 markierten Situationen statt; Fall 1 ist unkritisch (er tritt beispielsweise bei Zyklen gerader Länge auf). Im Fall 2 wird ein bisher gefundener alternierender Weg um eine freie und eine gebundene Kante verlängert. Im Fall 3 wird eine Blüte geschrumpft. Im Fall 4 müssen zwei bereits gefundene alternierende Wege mit jeweils gerader Kantenzahl durch Hinzunahme einer freien Kante verbunden werden; damit erhält man einen vergrößernden Weg, und die Ausführung des Algorithmus ist beendet.

Betrachten wir als Beispiel den in Abbildung 8.46 dargestellten Graphen mit der Zuordnung $Z = \{(6,7),(8,10)\}$. Abbildung 8.52 zeigt einen Ausschnitt dieses Graphen, wobei der Zustand *gerade* für einen Knoten durch ein Pluszeichen angegeben ist; den Zustand *ungerade* werden wir durch ein Minuszeichen angeben. Wählen wir als ersten

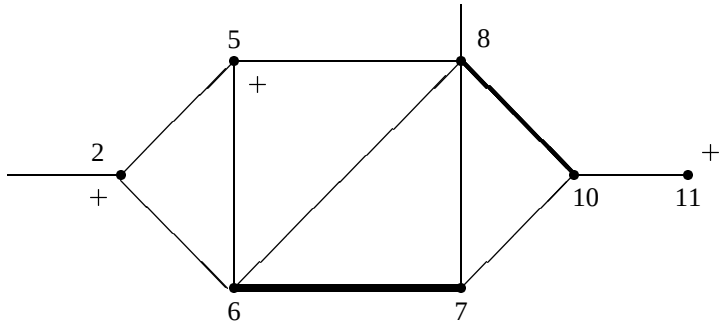


Abbildung 8.52

Pfeil den Pfeil (2,6), so liegt Fall 2 vor, weil Knoten 6 bislang unerreicht ist. Wir setzen also den Zustand von Knoten 6 auf *ungerade*, den Zustand von Knoten 7, das ist der Zuordnungspartner von Knoten 6, auf *gerade*, und merken uns Knoten 2 als Vorgänger von Knoten 6 und Knoten 6 als Vorgänger von Knoten 7. Die entstehende Situation ist in Abbildung 8.53 gezeigt.

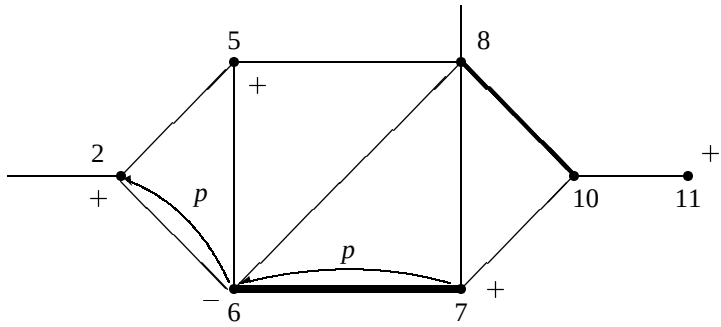


Abbildung 8.53

Im Effekt ist also aus einem Weg der Länge 0, nämlich Knoten 2 alleine, durch Hinzunahme der freien Kante (2,6) und der gebundenen Kante (6,7) ein alternierender Weg der Länge 2 konstruiert worden. Wählen wir beim nächsten Durchlauf der **repeat**-Schleife des Algorithmus *Vergrößernder Weg* den Pfeil (7,10), so liegt wieder Fall 2 vor, und der Weg 2, 6, 7 wird über Knoten 7 hinaus zu Knoten 10 und Knoten 8 weitergeführt. Abbildung 8.54 zeigt die entstehende Situation.

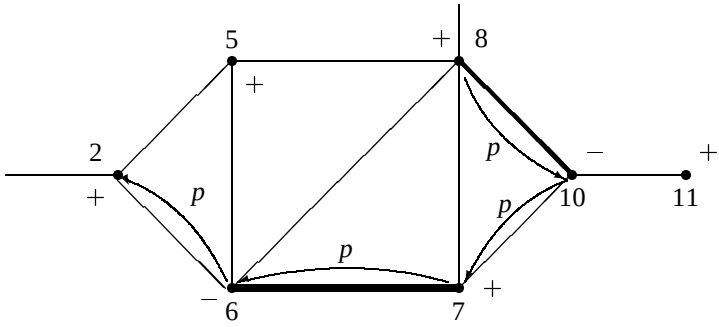


Abbildung 8.54

Wählen wir bei der nächsten Iteration den Pfeil (11,10), so liegt Fall 1 vor. Pfeil (11,10) ist damit untersucht, ohne daß sich an einem Weg etwas geändert hat. Wählen wir bei der nächsten Iteration Pfeil (8,7), so tritt erstmals Fall 3 ein. Knoten 8 und Knoten 7 befinden sich im Baum mit Wurzel 2. Der nächste gemeinsame Vorfahr von Knoten 8 und Knoten 7 ist Knoten 7. Wir schrumpfen also die Blüte 8, 7, 10 in den Knoten 7 und bezeichnen diesen Knoten jetzt als $7'$. Abbildung 8.55 zeigt die entstandene Situation.

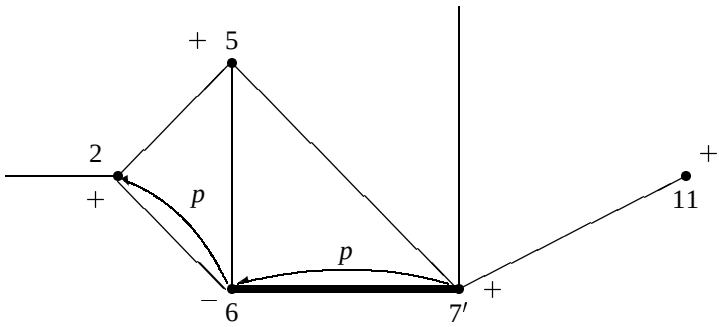


Abbildung 8.55

Knoten $7'$ ist im Zustand *gerade*, weil Knoten 7 vor dem Schrumpfen der Blüte im Zustand *gerade* war. Betrachten wir bei der nächsten Iteration den Pfeil $(11, 7')$, so liegt Fall 4 vor. Knoten $7'$ befindet sich im Baum mit Wurzel 2, und Knoten 11 bildet einen eigenen Baum. Jetzt werden die beiden Bäume mit der Kante $(11, 7')$ verbunden; es ent-

steht ein vergrößernder Weg zwischen den beiden Wurzeln der Bäume, also zwischen Knoten 2 und Knoten 11.

Um einen vergrößernden Weg im ursprünglich gegebenen Graphen zu finden, werden alle betroffenen Blüten wieder expandiert. Für den expandierten Weg von Knoten 2 nach Knoten 11 gibt es nun zwei Möglichkeiten, die Blüte 8, 7, 10 zu durchlaufen. Die eine Möglichkeit, nämlich die Kante (7,10) in der Blüte zu wählen, scheidet aus, weil der entstehende Weg 2, 6, 7, 10, 11 nicht alternierend ist. Ein alternierender Weg von Knoten 2 nach Knoten 11 ergibt sich, wenn man innerhalb der Blüte den Weg 7, 8, 10 einschlägt. Der entstehende, alternierende Weg 2, 6, 7, 8, 10, 11 ist ein vergrößernder Weg, weil beide Endknoten frei sind.

In der Literatur sind verschiedene effiziente Implementierungen dieses Algorithmus von Edmonds vorgeschlagen worden. In [62] ist eine spezielle Struktur zur Verwaltung disjunkter Mengen für eingeschränkte Fälle vorgestellt worden, die sich zum Verwalten von Blüten und Teilbäumen im Graphen eignet. Mit dieser Struktur gelingt es, eine maximale Zuordnung für einen Graphen $G = (V,E)$ in Zeit $O(|V| \cdot |E|)$ zu finden.

Später [125] wurde ein Algorithmus gefunden, der im wesentlichen den Algorithmus für eine bipartite Zuordnung um das Schrumpfen von Blüten ergänzt und mit einer Laufzeit von $O(\sqrt{|V|} \cdot |E|)$ auskommt. Somit ist das Berechnen einer maximalen Zuordnung für einen beliebigen Graphen gr[illegible]ordnungsmäßig nicht teurer als für einen bipartiten Graphen, ein beachtliches Ergebnis.

8.8.3 [illegible]ximale gewichtete Zuordnungen

Das Berechnen maximaler gewichteter Zuordnungen ähnelt dem Berechnen maximaler Zuordnungen ohne Kantengewichte sehr stark. Außer alternierenden Wegen betrachten wir hier aber auch alternierende Zyklen, weil auch diese das Gewicht einer Zuordnung vergrößern können. Das Gewicht $w(p)$ eines alternierenden Weges oder Zyklus p ist das Gesamtgewicht der freien Kanten in p abzüglich dem Gesamtgewicht der gebundenen Kanten in p, also $w(p) = \sum_{e \in p, e \notin Z} w(e) - \sum_{e \in p, e \in Z} w(e)$. Wir vergrößern eine Zuordnung Z, indem wir die Anzahl der Kanten in Z um 1 erhöhen. Daß dies stets durch Berücksichtigung eines Z vergrößernden Wegs mit maximalem Gewicht geschehen kann, zeigt folgende Überlegung. Sei Z eine Zuordnung maximalen Gewichts unter allen Zuordnungen der Größe $|Z|$, und sei p ein vergrößernder Weg für Z mit maximalem Gewicht. Dann ist das Resultat der Vergrößerung von Z durch p eine Zuordnung mit maximalem Gewicht unter allen Zuordnungen der Größe $|Z|+1$. Um dies einzusehen, betrachten wir eine Zuordnung Z_{max} mit maximalem Gewicht unter allen Zuordnungen der Größe $|Z|+1$. Betrachten wir jetzt die symmetrische Differenz Z_{sym} zwischen Z und Z_{max}, also $Z_{sym} = (Z - Z_{max}) \cup (Z_{max} - Z)$. Definieren wir nun das Gewicht eines Wegs oder Zyklus in Z_{sym} mit Bezug auf Z, also als Differenz der Gewichte freier Kanten bezüglich Z und gebundener Kanten bezüglich Z, so hat jeder Zyklus oder Weg gerader Länge das Gewicht 0. Dies muß gelten, weil sich Kanten aus Z_{max} mit Kanten aus Z abwechseln und sowohl Z_{max} als auch Z maximales Gewicht haben muß, denn hätte Z ein von Z_{max} verschiedenes Gewicht, so könnte man die Zuordnung mit dem geringeren Gewicht durch diejenige mit dem größeren Gewicht ersetzen. Weil durch einen vergrößernden Weg zu Z genau eine Kante hinzukommt, stammt in Z_{sym} genau eine Kante

mehr aus Z_{max} als aus Z. Die Wege in Z_{sym} können so zu Paaren zusammengefaßt werden, daß für jedes Paar gleich viele Kanten aus Z und aus Z_{max} kommen, und daß das Gewicht jedes Paares von Wegen 0 ist, mit Ausnahme eines einzigen Wegs ungerader Länge. Dieser Weg kann zur Vergrößerung für Z verwendet werden; er führt zu einer Zuordnung der Größe $|Z|+1$ mit demselben Gewicht wie Z_{max}.

Die dieser Überlegung entsprechende iterierte Vergrößerung des Gewichts einer Zuordnung verläuft mit abnehmender Zunahme des Gewichts in jedem Iterationsschritt. Zum Berechnen einer Zuordnung maximalen Gewichts genügt es also, die Iteration anzuhalten, wenn die Gewichtszunahme negativ würde. Die Überlegungen zu Blüten gelten wie für maximale Zuordnungen. Wegen der zusätzlichen Berücksichtigung von Kantengewichten sind die bekannten Algorithmen für maximale gewichtete Zuordnungen aber nicht ganz so effizient. Die schnellsten Implementierungen [61, 104] bzw. [63] erreichen eine Laufzeit von $O(|V|^3)$ bzw. $O(|V| \cdot |E| \log |V|)$.

8.9 Aufgaben

Aufgabe 8.1

a) Geben Sie an, wie der in Abbildung 8.8 dargestellte Graph in einer Adjazenzmatrix, in Adjazenzlisten und in einer doppelt verketteten Pfeilliste gespeichert wird.

b) Ignorieren Sie die Pfeilrichtungen dieses Graphen und deuten Sie ihn als ungerichteten Graphen, so daß also jeder Pfeil als Kante interpretiert wird. Geben Sie an, wie der so definierte ungerichtete Graph in einer Adjazenzmatrix, in Adjazenzlisten und in einer doppelt verketteten Pfeilliste gespeichert wird.

c) Welche Besonderheiten ergeben sich im allgemeinen beim Speichern ungerichteter Graphen gegenüber gerichteten Graphen für die drei Speicherungsformen?

Aufgabe 8.2

a) Schreiben Sie ein Pascal-Programm, das es gestattet, eine der drei Speicherungsformen zu wählen und einen Graphen einzugeben. Die Eingabe soll interaktiv durch Angabe von Knoten und Pfeilen bzw. Kanten in beliebiger Reihenfolge erfolgen können. Außerdem soll es möglich sein, einen auf einer externen Datei gespeicherten Graphen einzulesen und einen Graphen auf einer externen Datei zu speichern.

b) Ergänzen Sie das Programm aus Teilaufgabe a) um einige Prozeduren zum Editieren eines Graphen. Es soll mindestens möglich sein, einzelne Knoten und Kanten bzw. Pfeile hinzuzufügen und zu löschen. Löscht man einen Knoten, so sollen auch alle inzidenten Kanten bzw. Pfeile gelöscht werden.

c) Ergänzen Sie das Programm aus Teilaufgabe b) um eine graphische Ausgabemöglichkeit von Graphen. Weil ein automatisches, schönes Zeichnen von Graphen sehr schwierig ist, sollen Positionen von Knoten (z.B. Koordinaten) mit dem Graphen abgespeichert und ebenfalls editiert werden können. Kanten bzw. Pfeile solle als geradlinige Verbindungen der entsprechenden Knoten gezeichnet werden.

d) Ergänzen Sie das Programm aus Teilaufgabe c) um eine interaktive, graphische Benutzerschnittstelle. Es soll also nicht nur die Ausgabe graphisch möglich sein, sondern auch die Eingabe durch den Benutzer. Man sollte wenigstens Knoten und Kanten bzw. Pfeile graphisch selektieren können (Anklicken), beispielweise um sie zu löschen oder um Knoten zu verschieben. Folgeeffekte, wie das Löschen der mit einem gelöschten Knoten inzidenten Kanten oder das Verziehen von Kanten sollen automatisch graphisch berücksichtigt werden.

Aufgabe 8.3

Geben Sie für jede der drei Speicherungsformen (möglichst sinnvolle) Operationen an, die bei dieser Speicherungsform zumindest in gewissen Fällen

a) effizienter als bei den beiden anderen

b) weniger effizient als bei den beiden anderen

ausgeführt werden können.

Aufgabe 8.4

a) Berechnen Sie nach dem in Abschnitt 8.1 vorgestellten Algorithmus eine topologische Sortierung des in Abbildung 8.3 dargestellten Digraphen. Wieviele verschiedene topologische Sortierungen gibt es in diesem Beispiel?

b) Modifizieren Sie den Algorithmus zur topologischen Sortierung so, daß er diese für einen in einer Adjazenzmatrix mit Zusatzinformation über bedeutsame Einträge gespeicherten Digraphen berechnet. Welche Laufzeit hat der modifizierte Algorithmus?

Aufgabe 8.5

a) In Mehrbenutzer-Betriebssystemen konkurrieren verzahnt ablaufende Prozesse um Betriebsmittel. Hat beispielsweise ein Prozeß p den Farbdrucker f gerade belegt, und benötigt ein anderer Prozeß p' ebenfalls f, so muß p' warten, bis p wieder f freigibt. Dies definiert eine binäre Relation: p' wartet auf p. Wenn in dieser Relation ein Zyklus auftritt (p' wartet wegen des Farbdruckers auf p; p wartet wegen des Lochstreifenlesers auf p'), so ist das Fortsetzen der Prozesse auf Dauer behindert, die Prozesse sind verklemmt. Es ist dann wünschenswert, gewisse Prozesse abzubrechen, die gebundenen Betriebsmittel freizugeben und diese Prozesse später erneut zu starten.

   Entwerfen Sie einen möglichst effizienten Algorithmus, der in einer Menge von Prozessen und Warte-Beziehungen der Prozesse untereinander feststellt, wie durch Abbrechen einer möglichst kleinen Anzahl von Prozessen alle bestehenden Verklemmungen aufgelöst werden könen. Welche Laufzeit hat Ihr Algorithmus?

b) Entwerfen Sie einen möglichst effizienten Algorithmus, der aus einem gegebenen Digraphen durch Entfernen einer möglichst kleinen Anzahl von Pfeilen einen zyklenfreien Digraphen herstellt. Welche Laufzeit hat Ihr Algorithmus?

Aufgabe 8.6

a) Entwerfen Sie einen möglichst effizienten Algorithmus, der zu einem gegebenen, zyklenfreien Digraphen die Anzahl der verschiedenen topologischen Sortierungen berechnet. Welche Laufzeit hat Ihr Algorithmus?

b) Entwerfen Sie einen möglichst effizienten Algorithmus, der zu einem gegebenen Digraphen die Anzahl der verschiedenen einfachen Zyklen berechnet (für den Digraphen in Abbildung 8.4 ist diese Anzahl 3). Welche Laufzeit hat Ihr Algorithmus?

Aufgabe 8.7

Entwerfen Sie einen möglichst effizienten Algorithmus zur Berechnung der reflexiven, transitiven Hülle zu einem beliebigen, in Adjazenzlistenrepräsentation gegebenen Digraphen. Welche Laufzeit hat dieser Algorithmus, insbesondere für Graphen mit wenigen Kanten?

Aufgabe 8.8

Bei einer Meinungsumfrage hat ein Befragter aus einer vorgelegten Liste von Tätigkeiten Paare von Tätigkeiten gebildet, wobei er die erste Tätigkeit der zweiten vorzieht; über manche Tätigkeitspaare hat er keine Aussage gemacht. So äußert er beispielweise, Bier trinken oder fernsehen sei schöner als Holz hacken, Holz hacken schöner als Schach spielen, und Bier trinken sei schöner als Schach spielen. Auf die zuletzt genannte Präferenz hätte der Interviewer allerdings auch (durch Transitivität) selbst schließen können. Nehmen Sie an, daß der Befragte konsistent geantwortet hat, also keine Tätigkeit schöner findet als diese selbst (über Transitivität).

a) Entwerfen Sie einen möglichst effizienten Algorithmus, der aus der Menge aller Tätigkeitspaare, die ein Befragter angegeben hat, diejenigen entfernt, auf die man durch die verbleibenden schließen kann. Geben Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von der Anzahl der angegebenen und der übrigbleibenden Tätigkeitspaare an.

b) Entwerfen Sie einen möglichst effizienten Algorithmus, der zur Menge aller durch einen Befragten angegebenen Tätigkeitspaare all diejenigen Tätigkeitspaare ermittelt, auf die man nicht über Transitivität schließen kann. Geben Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von der Anzahl der angegebenen und der Anzahl der zu ermittelnden Tätigkeitspaare an.

c) Entwerfen Sie einen möglichst effizienten Algorithmus, der die Menge der Tätigkeiten so in kleinstmögliche Teilmengen zerlegt, daß über Tätigkeiten aus verschiedenen Teilmengen nie eine Präferenzaussage vorliegt. Geben Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von der Anzahl der Tätigkeiten und der Anzahl der angegebenen Tätigkeitspaare an.

Aufgabe 8.9

Berechnen Sie den DFBIndex und den DFEIndex eines jeden Knotens sowie die Klassifikation aller Pfeile in Baum-, Vorwärts-, Rückwärts- und Seitwärtspfeile für den Graphen in Abbildung 8.11 und jeden der Startknoten 2, 3, 4 und 5 einer Tiefensuche. In welchen dieser Fälle kann man die Knoten nach dem allgemeinen Knotenbesuchsalgorithmus auch in einer anderen Reihenfolge besuchen?

Aufgabe 8.10

Ein Graph ist dreifach zusammenhängend, wenn er nach dem Entfernen zweier beliebiger Knoten samt aller inzidenten Kanten noch zusammenhängend ist.

a) Entwerfen Sie einen möglichst effizienten Algorithmus, der prüft, ob ein gegebener Graph dreifach zusammenhängend ist. Welche Laufzeit hat Ihr Algorithmus?

b) Entwerfen Sie einen möglichst effizienten Algorithmus, der alle dreifachen Zusammenhangskomponenten eines gegebenen Graphen berechnet.

c) Wenden Sie Ihren Algorithmus auf das in Abbildung 8.12 (a) dargestellte Beispiel an.

Aufgabe 8.11

Eine Kante in einem zusammenhängenden, ungerichteten Graphen heißt *Brücke*, wenn das Entfernen dieser Kante den Graphen in zwei Teile zerfallen läßt.

Entwerfen Sie einen möglichst effizienten Algorithmus, der zu einem gegebenen Graphen alle Brücken ermittelt. Wie schnell arbeitet Ihr Algorithmus?

Aufgabe 8.12

In einer Stadt verspricht man sich eine Beschleunigung des Verkehrsflusses, wenn man aus den bisher in beiden Fahrtrichtungen benutzbaren, oftmals engen Straßen Einbahnstraßen macht. Danach soll es natürlich noch möglich sein, von jedem Ort an jeden anderen zu gelangen.

Entwerfen Sie einen möglichst effizienten Algorithmus, der zu einem gegebenen Netz von Straßen, die in beiden Richtungen befahrbar sind, ein solches Einbahnstraßennetz findet, wann immer dies möglich ist. Zeigen Sie, daß dies genau dann möglich ist, wenn das gegebene Netz zusammenhängend ist und keine Brücken enthält. (Mehr über dieses und ähnliche Probleme findet man in [162].)

Aufgabe 8.13

Verfolgen Sie anhand des Beispiels von Abbildung 8.18 Dijkstras Algorithmus zum Finden aller kürzesten Wege von Knoten 4 aus, wenn die Randknoten in einem Fibonacci-Heap verwaltet werden.

Aufgabe 8.14

Um eine wichtige, geheime Botschaft von A nach B zu befördern, werden aus Sicherheitsgründen zwei Kuriere losgeschickt, die völlig verschiedene Wege von A nach B in einem Netz von Wegen wählen müssen. Diese Wege sollen so gewählt werden, daß der längere der beiden möglichst kurz ist. Entwerfen Sie einen Algorithmus, der zwei solche Wege wählt, wenn

a) Wege im Netz in beiden Richtungen benutzbar sind;

b) Wege nur in einer Richtung benutzbar sind.

Aufgabe 8.15

Das Finden eines kürzesten Weges in einem bewerteten, ungerichteten Graphen mit beliebiger Kantenbewertung scheitert im allgemeinen an der möglichen Existenz negativer Zyklen; in diesem Fall existiert kein kürzester Weg, weil der negative Zyklus mehrmals durchlaufen werden kann. Dieses Problem verschwindet, wenn wir nur einfache Wege suchen, also solche Wege, die jeden Knoten höchstens einmal betreten.

Entwerfen Sie für diesen Fall einen möglichst effizienten Algorithmus zum Finden eines kürzesten Weges zwischen zwei gegebenen Knoten. Welche Laufzeit hat Ihr Algorithmus?

Aufgabe 8.16

Versehen Sie die Pfeile in Abbildung 8.3 mit Werten für die Dauern der entsprechenden Vorgänge und berechnen Sie mit einem Auswahlverfahren nach Ford die Mindestdauer des Gesamtprojekts. Wählen Sie dabei Randknoten gemäß einer topologischen Sortierung.

Aufgabe 8.17

In einem Distanzgraphen kann man hoffen, einen kürzesten Weg zwischen zwei gegebenen Knoten schnell zu finden, wenn man eine Breitensuche wie bei Dijkstras Algorithmus nicht nur bei einem der beiden Knoten startet, sondern gleichzeitig bei beiden. Präzisieren Sie diese Idee und entwerfen Sie einen entsprechenden Algorithmus. Implementieren Sie Ihren Algorithmus und Dijkstras Algorithmus und experimentieren Sie.

Aufgabe 8.18

Entwerfen Sie einen Algorithmus zur Berechnung eines kürzesten Weges zwischen zwei gegebenen Knoten eines Distanzgraphen, der in Matrixform gespeichert ist. Der Algorithmus soll auf der Matrix operieren. Welche Laufzeit hat Ihr Algorithmus? Welchen Effekt hat die Matrixspeicherung auf die Berechnung aller kürzesten Wege im Graphen?

Aufgabe 8.19

Geben Sie an, wie man Dijkstras Algorithmus zur Berechnung kürzester Wege so modifizieren kann, daß er neben der Länge auch die Anzahl der kürzesten Wege von einem gegebenen Startknoten zu einem anderen Knoten berechnet.

Aufgabe 8.20

Entwerfen Sie einen möglichst effizienten Algorithmus, der in einem bewerteten, ungerichteten Graphen einen Weg zwischen zwei gegebenen Knoten findet, bei dem

a) die Länge der längsten Kante möglichst klein ist;

b) die Länge der kürzesten Kante möglichst groß ist.

Aufgabe 8.21
Verfolgen Sie die Berechnung eines minimalen, spannenden Baums für den in Abbildung 8.18 dargestellten Graphen nach jedem der in Abschnitt 8.6 vorgestellten Verfahren.

Aufgabe 8.22
Entwerfen Sie einen Algorithmus zur Berechnung eines spannenden Baums für einen gegebenen Distanzgraphen, bei dem

a) die Länge der längsten Kante möglichst klein ist;

b) die Länge des längsten Weges zwischen zwei Knoten — das ist der Durchmesser des Baumes — möglichst klein ist;

c) der größte Knotengrad möglichst klein ist.

Aufgabe 8.23
Entwerfen sie einen möglichst effizienten Algorithmus, der in einem gegebenen Distanzgraphen einen Knoten — das Zentrum — findet, dessen größte Entfernung zu irgendeinem anderen Knoten des Graphen minimal ist. Welche Laufzeit hat Ihr Algorithmus?

Aufgabe 8.24
Legen Sie für jede Kante des in Abbildung 8.46 gezeigten Graphen eine Kapazität und eine Orientierung fest, so daß sich ein Kapazitätsdigraph mit Quelle 11 und Senke 12 ergibt. Berechnen Sie einen maximalen Fluß von der Quelle zur Senke nach dem Algorithmus

a) Flußvergrößerung durch einzelne kürzeste zunehmende Wege;

b) Flußvergrößerung durch kürzesten Weg im Niveaugraphen.

Aufgabe 8.25
Ändern Sie die zur Berechnung eines maximalen Flusses vorgestellten Algorithmen so, daß auch Mindestkapazitäten von Pfeilen berücksichtigt werden. Dabei soll der Fluß entlang eines Pfeiles für jeden Pfeil

a) zwischen der Mindest- und der Maximalkapazität für diesen Pfeil liegen;

b) entweder 0 sein oder zwischen der Mindest- und der Maximalkapazität für diesen Pfeil liegen.

Aufgabe 8.26
Entwerfen Sie einen möglichst effizienten Algorithmus, der zu einem gegebenen Kapazitätsdigraphen einen Fluß

a) mit möglichst vielen gesättigten Pfeilen;

b) mit mindestens einer Flußeinheit für jeden Pfeil;

c) mit einem möglichst niedrigen Durchfluß durch den Knoten mit größtem Durchfluß bei einem maximalen Fluß

berechnet.

Aufgabe 8.27

Bestimmen Sie für den Graphen in Abbildung 8.18 eine maximale Zuordnung nach der Methode der vergrößernden Wege; ignorieren Sie die Bewertungen der Kanten.

Aufgabe 8.28

Bestimmen Sie für den Graphen in Abbildung 8.18 eine maximale, gewichtete Zuordnung.

Aufgabe 8.29

Entwerfen Sie einen möglichst effizienten Algorithmus, der für einen gegebenen, ungerichteten Graphen die Anzahl der maximalen Zuordnungen ermittelt.

Aufgabe 8.30

Bestimmen Sie eine möglichst scharfe obere Schranke für die Anzahl der freien Knoten bezüglich einer maximalen Zuordnung in einem beliebigen, ungerichteten Graphen.

Aufgabe 8.31

Entwerfen Sie einen möglichst effizienten Algorithmus, der eine gegebene Zuordnung in einem ungerichteten Graphen maximal erweitert. Gesucht ist dabei eine Zuordnung, in der alle als gebunden gegebenen Kanten gebunden sind, und die maximal ist unter allen solchen Zuordnungen.

Aufgabe 8.32

a) Wir verallgemeinern den Begriff der Zuordnung so, daß ein gebundener Knoten zu mehr als einer gebundenen Kante gehören darf. Entwerfen Sie einen möglichst effizienten Algorithmus, der für einen gegebenen, ungerichteten Graphen eine möglichst kleine Menge gebundener Kanten berechnet, so daß alle Knoten gebunden sind.

b) Entwerfen Sie einen möglichst effizienten Algorithmus, der eine kleinstmögliche Menge von Knoten eines gegebenen, ungerichteten Graphen wählt, so daß jede Kante mit wenigstens einem Knoten inzidiert.

c) Entwerfen Sie einen möglichst effizienten Algorithmus, der eine größtmögliche Menge von Knoten eines gegebenen Graphen wählt, so daß jede Kante mit höchstens einem Knoten inzidiert.

Wie vereinfachen sich diese Probleme, wenn wir nur bipartite Graphen als Eingabe zulassen?