Theoretische Informatik 1

Effizienz von Algorithmen

Vorlesung an der Technischen Universität Chemnitz

Wintersemester 1998/99

Prof. Dr. Werner Dilger

Inhalt Seite

Literatur

T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms. MIT Press, Cambridge, MA, 1994.

J.H. Kingston: Algorithms and Data Structures. Design, Correctness, Analysis. [2] Addison-Wesley, Harlow, 1998

1. Übersicht über die Theoretische Informatik

Wenn man den Gegenstand der Informatik allgemein als die Entwicklung und Verarbeitung von Algorithmen auffasst, dann beschäftigt sich die Theoretische Informatik mit folgenden Fragestellungen:

1. Was kann man überhaupt alles mit Algorithmen tun, d.h. was alles ist auf einem Rechner berechenbar und was nicht?

Berechenbarkeit: Baue ein Modell eines Computers, genannt *Turingmaschine*, und beschreibe, was mit ihm alles berechnet werden kann. Das, und nur das, wird als *berechenbar* bezeichnet. Es stellt sich heraus, dass es dasselbe leistet wie alle anderen Modelle von Computern, die man konstruieren kann. Deshalb nimmt man an, dass alles, was man intuitiv als berechenbar bezeichnen würde, mit der Turingmaschine berechnet werden kann.

2. Wie schwierig sind Probleme, d.h. welchen Rechenaufwand benötigt ein Problem zu seiner Lösung?

Komplexitätstheorie: Stelle fest, wie komplex ein gegebenes Problem ist, d.h. welchen Zeitaufwand seine Berechnung größenordnungsmäßig erfordert. Definiere hierfür, wie sich die Rechenzeit für ein Problem größenordnungsmäßig erfassen lässt und lege nach diesem Maß geeignete Komplexitätsklassen fest. Versuche für gegebene Probleme zu bestimmen, in welche Komplexitätsklasse sie gehören.

3. Nach welchen Kriterien muss eine Programmiersprache definiert sein, damit sie einerseits ausdrucksstark ist und andererseits gut zu verarbeiten?

Formale Sprachen und Automatentheorie: Definiere formale Sprachen durch Angabe verschiedener Typen generativer Grammatiken und beschreibe vergleichend die Eigenschaften dieser Sprachen. Definiere zu jeder Sprachklasse einen Automatentyp, der Eingaben aus der jeweils zugehörigen Sprachklasse verarbeiten kann.

4. Wie lässt sich die Effizienz von Algorithmen, d.h. die Rechenzeit, die sie benötigen, bestimmen und welche effizienten Algorithmen lassen sich für Standardprobleme der Informatik angeben?

Effizienz von Algorithmen: Für Standardprobleme der Informatik werden Algorithmen betrachtet und deren Speicherplatzbedarf sowie ihre Laufzeit in abstrakter Weise bestimmt. Die angegebenen Algorithmen sind besonders effizient.

Grobgliederung der Theoretischen Informatik 1 und 2:

TI 1: 1. Effizienz von Algorithmen

TI 2: 2. Berechenbarkeit
3. Komplexitätstheorie
4. Formale Sprachen und Automatentheorie

Im ersten Teil der Vorlesung werden algorithmische Lösungen für verschiedene Standardprobleme behandelt. Relativ breiten Raum nehmen die Graphalgorithmen ein. Bei ihnen geht es um das systematische Aufsuchen aller Knoten eines beliebigen, gerichteten oder ungerichteten Graphen. Als die beiden hauptsächlichen Lösungswege werden die Breitensuche und die Tiefensuche behandelt. Vor allem mit der Tiefensuche ist es möglich, verschiedene Strukturen in Graphen zu bestimmen. Als

nächstes Problem wird die Bestimmung kürzester Wege in Graphen betrachtet und es werden verschiedene Verfahren dafür eingeführt. Weiterhin werden für eine besondere Struktur in Graphen, die minimalen Spannbäume, zwei Algorithmen behandelt.

Nach der Betrachtung der Graphalgorithmen werden effiziente Algorithmen zur Lösung verschiedener anderer Probleme eingeführt. Der erste untersuchte Typ von Problemen bei geordneten Mengen von Daten ist das Sortieren. Dafür werden zwei Algorithmen betrachtet. Ebenfalls für solche Mengen wird das Auswahlproblem behandelt. Für die Aufgabe der Matrizenmultiplikation wird eine Divide-and-conquer-Strategie untersucht. Bei vielen Problemen entsteht während des Lösungsweges die Situation, dass auf dem eingeschlagenen Weg nicht fortgefahren werden kann. Dann muss man an eine frühere Stelle des Lösungsweges zurückspringen um einen anderen Weg zu verfolgen. Diese Vorgehensweise heißt Backtracking. Um die möglichen Lösungswege bei einer Aufgabe einzuschränken lassen sich Branch-and-Bound-Verfahren einsetzen. Schließlich werden noch die lokale Suche in Graphen und das Dynamische Programmieren behandelt.

2. Effizienz von Algorithmen

2.1. Motivation

Die Effizienz eines Algorithmus oder eines Programms wird durch zwei Größen bestimmt:

1. Laufzeit
2. Benötigter Speicherplatz

Beispiel

Bestimme die Laufzeit des folgenden Programms:

```
Var  A: array [1..amax] of integer
     amax, zaehler: integer
1.   read(amax)
2.   for i = 1 to amax
3.      do read A[i]
4.   zaehler := 0
5.   for i = 1 to amax
6.      do if A[i] = 7 then zaehler := zaehler +1
7.   write(zaehler)
```

Laufzeit für einzelne Befehle:

Für jeden einfachen Befehl (Zuweisungsbefehl, read-Befehl, write-Befehl) wird die Laufzeit 1 angesetzt, ebenso für jeden Test, für das Erhöhen des Schleifenzählers und für die Schleifenverwaltung, d.h. für den Test, ob das Ende der Schleife erreicht ist.

if-Befehle enthalten implizit einen Sprungbefehl. Für diesen wird die Laufzeit 1 angesetzt.

Die gesamte Laufzeit des Programms hängt vom Inhalt von A und von amax ab. Die Laufzeit kann als Funktion

$$T: Eingabe \rightarrow IN$$

Hier ist speziell T(A[1..amax], amax) = $t \in \mathbb{N}$ mit $3 + 7 \cdot amax \leq t \leq 3 + 8 \cdot amax$.

Das Problem bei der Bestimmung von T ist allgemein, d.h. für beliebige Programme, dass T eine etwas undurchsichtige und schwer zu ermittelnde Funktion ist. Um trotzdem einen Wert für T zu finden, betrachtet man alle Eingaben einer festen "Größe" und den schlechtesten Fall aller Eingaben dieser Größe.

Im Folgenden wird die schlechteste Zeit T_{wc} genannt (wc für *worst case*). Es ist

$$T_{wc}: IN \to IN$$

Die *Größe* entspricht der Anzahl der Bits für die Eingabe (alternativ Anzahl der Speicherplätze), es ist nicht die Größe der Eingabewerte als Zahlen gemeint, denn das Programm arbeitet mit Bitstrings und kennt nicht den Wert einer Zahl. Im Beispiel ist deshalb die Größe amax. Wir setzen abkürzend M(n) := {T(A[1], A[2], ..., A[amax], amax) | amax = n}.

T_{wc} wird definiert als das Maximum von M(n), also $T_{wc}(n) = \max M(n)$.

Verfeinerung der Laufzeitabschätzung

Sei z eine elementare Programmzeile (kein Prozeduraufruf). Zu z gibt es eine Konstante c_z, so dass die Ausführung der Zeile z auf einem Rechner eine Anzahl $m \le c_z$ von Maschinenbefehlen benötigt.

Sei $C = \max\{c_z \mid z \text{ ist Programmzeile}\}$. Dann ist die Gesamtlaufzeit eines Programms gemessen nach der Zahl der Maschinenbefehle $t \le C \cdot T_{wc}(n)$.

Die (worst case-) Laufzeit eines Programms wird nur bis auf einen konstanten Faktor spezifiziert. Das heißt, bis auf einen konstanten Faktor ist $T_{wc}(n) = n$. Genauer bedeutet dies, dass es Konstanten D und C gibt, die nur vom Programm abhängen, so dass gilt:

$$D \cdot n \le T_{wc}(n) \le C \cdot n$$

Weitere Gründe für die Vernachlässigung konstanter Faktoren bei der Bestimmung der Laufzeit:

- Bei neuen Rechnergenerationen verkürzt sich die Ausführungszeit von Maschinenbefehlen, entsprechend ändert sich der konstante Faktor.
- Die Laufzeitbestimmung wird theoretisch besser handhabbar.

2.2. **Definition** (Θ-Notation)

Ist $f: IN \to IN$, so wird definiert: Eine Funktion $g: IN \to IN$ *ist (von)* $\Theta(f(n))$ genau dann, wenn es zwei Konstanten C, D > 0 und ein $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \ge n_0$ gilt:

$$D \cdot f(n) \le g(n) \le C \cdot f(n)$$

2.3. **Definition** (O-Notation)

Ist $f: IN \to IN$, so ist eine Funktion $g: IN \to IN$ *von* $O(f)$ genau dann, wenn es C > 0, $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \ge n_0$ gilt:

$$g(n) \le C \cdot f(n)$$

Beachte: Ist g von Θ(f), dann ist g auch von O(f). Die Umkehrung gilt nicht!

2.4. **Definition** (Ω-Notation)

Ist $f: IN \to IN$, so ist eine Funktion $g: IN \to IN$ *von* $\Omega(f)$ genau dann, wenn es $D > 0$, $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt:

$$g(n) \geq D \cdot f(n)$$

Beachte: Ist g von Θ(f), dann ist auch g von Ω(f).

2.5. Bestimmung der worst case-Laufzeit eines Programms

Einmalige Ausführung einer Programmzeile z: Θ(1).

Da ein Programm nur aus endlich vielen Programmzeilen besteht, gibt es Konstanten D' und C', die nur vom Programm abhängig sind, so dass für jede Zeile gilt:

$$D' \leq \text{Laufzeit der Zeile} \leq C'.$$

Es reicht also zu zählen, wie oft jede Programmzeile im worst case ausgeführt wird (in Abhängigkeit von der Größe der Eingabe, z.B. n). Diese Anzahl sei eine Funktion von n, etwa f(n). Dann gilt für die worst case-Laufzeit T_{wc} des Programms

$$D \cdot f(n) \leq T_{wc}(n) \leq C \cdot f(n)$$

also ist $T_{wc}(n)$ von Θ(f(n)).

2.6. Beispiele

(a) Betrachte die Schleife

$$S \equiv \text{for } i = 1 \text{ to } 2 \cdot n \text{ do } A$$

Sei n die Größe der Eingabe und sei die worst case-Laufzeit von *A* $O(n^2)$. Daher gibt es eine Funktion g(n) von $O(n^2)$ und eine Konstante C von Θ(1), so dass gilt:

$$\text{worst case-Laufzeit von } S \leq \underbrace{C + g(n)}_{1.\text{Durchlauf}} + \underbrace{C + g(n)}_{2.\text{Durchlauf}} + \ldots + \underbrace{C + g(n)}_{2n.\text{Durchlauf}} = 2n \cdot C + 2n \cdot g(n)$$

Die worst case-Laufzeit ist also von O(n) + O(n·g(n)). Da g(n) von $O(n^2)$ ist, überwiegt O(n·g(n)), deshalb ist die worst case-Laufzeit von $O(n^3)$.

(b) Wie groß ist die Laufzeit eines Programms der folgenden Art:

1. $x_1 := e_1$
2. $x_2 := e_2$
3. $x_3 := e_3$

.
.
.

100. $x_{100} := e_{100}$

3. Graphentheoretische Grundlagen für Graphalgorithmen

3.1. **Definition** (Gerichteter Graph)

Ein *gerichteter Graph* ist ein Paar G = (V, E) mit den folgenden Komponenten:

- V ist eine Menge von Knoten
- $E \subseteq \{(u, v) \mid u, v \in V, u \neq v\}$ ist eine Menge von Kanten

3.2. **Definition** (Ungerichteter Graph)

Ein *ungerichteter Graph* ist ein Paar G = (V, E) mit den folgenden Komponenten:

- V ist eine Menge von Knoten
- $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ ist eine Menge von Kanten

3.3. **Definition** (Teilgraph)

Sei G = (V, E) ein gerichteter oder ungerichteter Graph. G' = (V', E') ist Teilgraph von G genau dann, wenn $V' \subseteq V$ und $E' \subseteq E$.

3.4. **Definition** (Wege in Graphen)

Sei G = (V, E) ein ungerichteter Graph. Ein *Weg* (oder *Pfad*) der Länge k ($k \geq 0$) von u nach u' ist eine Folge von k+1 Knoten

$$(v_0, v_1, ..., v_k) = (v_i)_{0 \leq i \leq k}$$

mit (i) $v_0 = u$
(ii) $v_k = u'$
(iii) $(v_{i-1}, v_i) \in E$ für alle $i \in \{1, ..., k\}$

Ein Weg heißt *einfach* genau dann, wenn $v_i \neq v_j$ für alle $i \neq j$. u' heißt von u aus *erreichbar* genau dann, wenn es einen Weg von u nach u' gibt. Die *Distanz* zwischen u und u' ist definiert durch

$$Dist(u,u') = \begin{cases} 0 & \text{falls } u = u' \\ \min\{k \mid \text{es gibt einen Weg der Länge } k \text{ von } u \text{ nach } u'\} & \text{falls } u \text{ von } u' \text{ aus erreichbar} \\ \infty & \text{falls es keinen Weg von } u \text{ nach } u' \text{ gibt} \end{cases}$$

Ein Weg in einem gerichteten Graph ist entsprechend definiert.

3.5. **Definition** (Kreise in gerichteten Graphen)

Sei G = (V, E) ein gerichteter Graph und sei $k \geq 2$. Ein Weg $(v_0, v_1, ..., v_k)$ in G ist ein *Kreis* genau dann, wenn $v_0 = v_k$. Ein Kreis $(v_0, v_1, ..., v_k)$ heißt *einfach* genau dann, wenn der Weg $(v_0, v_1, ..., v_k)$ einfach ist.

3.6. **Definition** (Kreise in ungerichteten Graphen)

Sei G = (V, E) ein ungerichteter Graph und sei $k \geq 3$. Ein Weg $(v_0, v_1, ..., v_k)$ in G ist ein *Kreis* genau dann, wenn der Weg $(v_0, v_1, ..., v_{k-1})$ einfach ist $v_0 = v_k$.

3.7. **Definition** (Grad von Knoten und Adjazenz)

Sei G = (V, E) ein ungerichteter Graph und v ∈ V. Der *Grad* von v ist definiert durch

$$Grad(v) = \{(v, u) \mid (v, u) \in E\} = \{(u, v) \mid (u, v) \in E\}$$

Die *Nachbarn* von v oder die zu v *adjazenten* Knoten sind definiert durch

$$A(v) = \{u \mid (v, u) \in E\}$$

Sei G = (V, E) ein gerichteter Graph und v ∈ V. Dann ist

$$Ausgangsgrad(v) = \{(v, u) \mid (v, u) \in E\}$$
$$Eingangsgrad(v) = \{(u, v) \mid (u, v) \in E\}$$

Die *Nachbarn* von v oder die zu v *adjazenten* Knoten sind ebenso definiert wie beim ungerichteten Graph.

3.8. **Definition** (Darstellung durch Adjazenzlisten und Adjazenzmatrizen)

Sei G = (V, E) ein (gerichteter oder ungerichteter) Graph. Die Darstellung von G durch *Adjazenzlisten* ist ein Array Adj[1..|V|] bestehend aus |V| Adjazenzlisten. Der Array enthält für jeden Knoten u eine Adjazenzliste, nämlich Adj[u], die genau die zu dem Knoten adjazenten Knoten enthält.

Statt durch Adjazenzlisten lassen sich Graphen auch durch *Adjazenzmatrizen* darstellen. Die Adjazenzmatrix eines Graphen G = (V, E) ist eine |V|×|V|-Matrix. Die Position (v_i, v_j) der Matrix enthält eine 1, falls $(v_i, v_j) \in E$, sonst eine 0.

4. Breitensuche in Graphen

4.1. **Definition** (Datenstruktur Schlange oder Queue)

Array: Q
Variablen: head, tail
Prozedur: Enqueue(Q, x)
Funktion: Dequeue(Q)

Enqueue(Q, x)

```
1.  Q[tail] := x
2.  if tail = length[Q]
3.      then tail := 1
4.      else tail := tail + 1
```

Die Laufzeit von Enqueue(Q, x) ist O(1).

Dequeue(Q)

```
1.  x := Q[head]
2.  if head = length[Q]
3.      then head := 1
```

4. else head := head + 1
5. return x

Die Laufzeit von Dequeue(Q) ist O(1).

Eine *Schlange* ist eine Datenstruktur zur Verwaltung von Mengen. Das Einfügen und Löschen erfolgt nach dem FIFO-Prinzip in Zeit O(1). Die Suche, ob ein Element in der Schlange enthalten ist, erfordert die Zeit O(n), wobei n die Länge der Schlange ist.

4.2. **Algorithmus** (Breitensuche, breadth first search)

Zur Motivation des Problems der Suche in Graphen wird das folgende Beispiel betrachtet. Gegeben sei der ungerichtete Graph G = (V, E) mit V = {r, s, t, u, v, w, x, y} und E = {(r, s), (r, v), (s, w), (w, t), (w, x), (t, u), (t, x), (u, y), (x, y)}. Er ist in Abbildung 4.1 dargestellt.

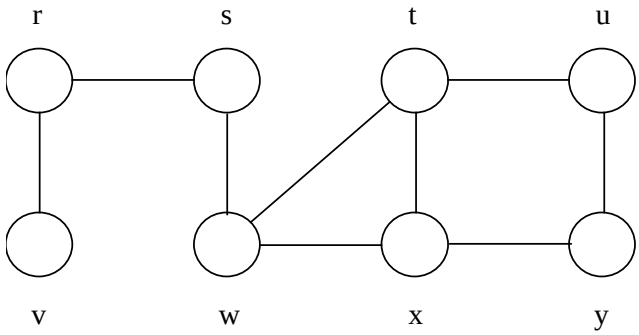

Abbildung 4.1

Darstellung des Prozesses der Durchsuchung des Graphen von Abbildung 4.1 in Abbildung 4.2.

Sei G = (V, E) ein (gerichteter oder ungerichteter) Graph und $s \in V$ (Startknoten). Der Array d[1..|V|] wird definiert durch d[i] := Dist(s, i) für alle $i \in V$. Der Array π[1..|V|] wird definiert durch π[i] ist der Vorgängerknoten bei der Durchsuchung des Graphen G, d.h. derjenige Knoten, von dem aus i gefunden wurde. Das heißt, $(\pi[i], i) \in E$.

Prozedur BFS(G, s)

```
1.  for each u ∈ V\{s}              ; Inititalisierungsphase, alle Elemente von V werden in
2.     do  color[u] := weiß                unspezifizierter Reihenfolge durchgegangen
3.         d[u] := ∞                ; es sind noch keine Distanzen bekannt
4.         π[u] := nil              ; es sind noch keine Knoten aufgesucht worden

5.  color[s] := grau
6.  d[s] := 0
7.  π[s] := nil                     ; s ist aufgesucht, aber noch keine Nachbarn von s
8.  Q := {s}                        ; Q ist die Menge der grau gefärbten Knoten (als Schlange ver-
                                          waltet)

9.  while Q ≠ ∅                             ; durch tail ≠ head testbar
10.    do u := head
11.       for each v ∈ Adj[u]               ; die Adjazenzliste von u wird abgearbeitet
12.          do if color[v] = weiß          ; hier ist noch d[v] = ∞, v ist offen
```

```
13.         then color[v] := grau
14.              d[v] := d[u] + 1          ; d[v] wird nur einmal gesetzt
15.              π[v] := u
16.              Enqueue(Q, v)             ; v wird in Q eingetragen
17.         Dequeue(Q)                     ; u wird aus Q gelöscht
18.         color[u] := schwarz            ; u ist abgeschlossen
```

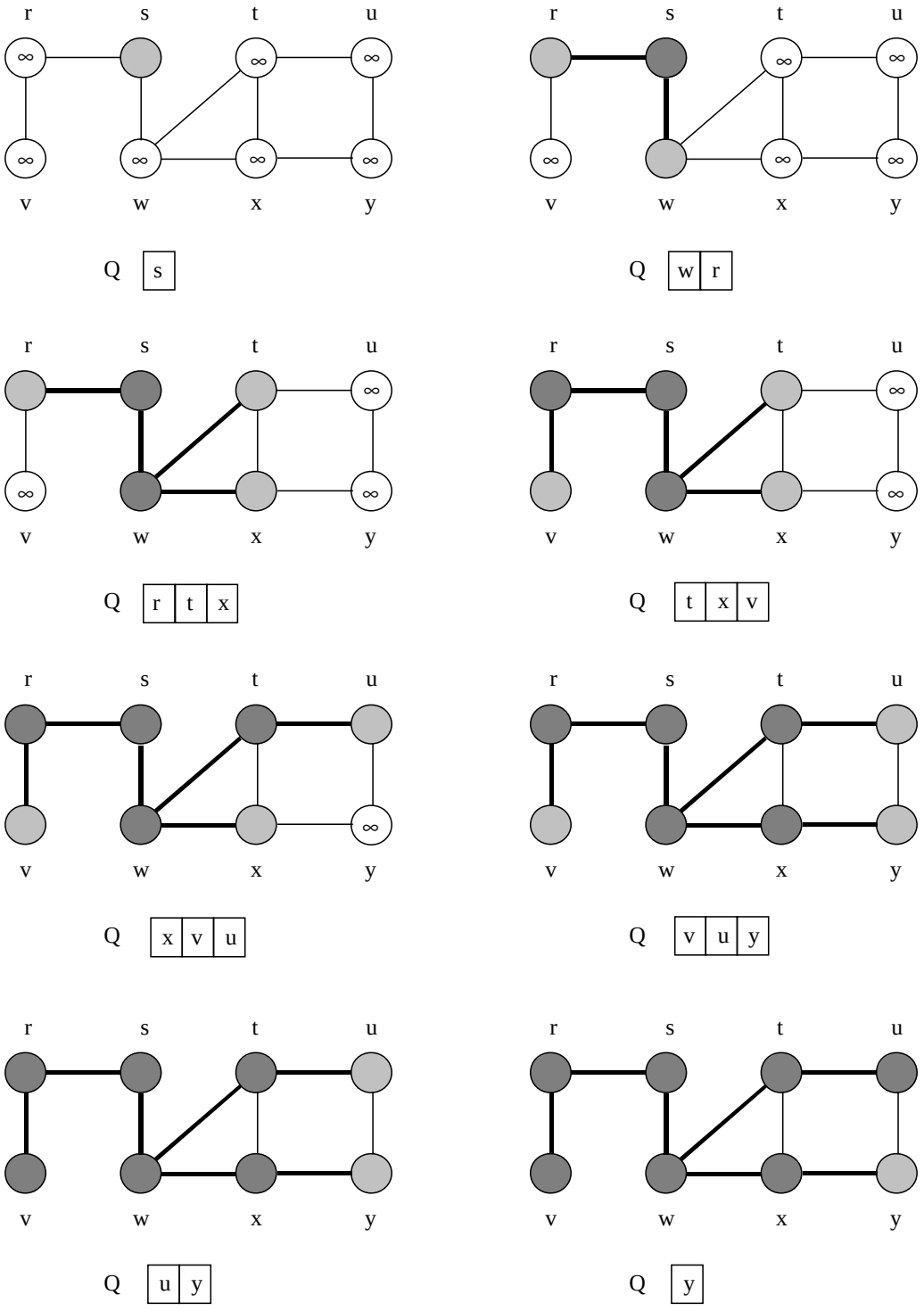

Abbildung 4.2

4.3. Satz (Laufzeit von BFS)

Sei $T_{wc}(n) = \max\{\text{Laufzeit von BFS}(G, s) \mid G = (V, E), s \in V \text{ und } |V| + |E| = n\}$. Dann gilt:

$$T_{wc}(n) = O(n)$$

4.4. Definition (Belegung)

Der Ablauf des Algorithmus BFS ist eine Folge von *Belegungen* B der Variablen (color, d, π, Q, head, tail).

Ein Knoten v wird in einer Belegung B aufgesucht genau dann, wenn B die erste Belegung in der Folge von Belegungen ist mit color[v] = grau.

4.5. Satz (Korrektheit von BFS)

Sei G = (V, E) ein (gerichteter oder ungerichteter) Graph und sei $s \in V$. Am Ende der Berechnung durch BFS(G, s) gelten die folgenden beiden Aussagen.

(i) Für alle $v \in V$, die nicht von s aus erreichbar sind, ist Dist(s, v) = d[v] = ∞ und π[v] = nil.

(ii) Für alle $v \in V$, die von s aus erreichbar sind, ist Dist(s, v) = d[v] ($< \infty$) und falls $s \neq v$ enthält der Array π einen kürzesten Weg von s nach v. Dieser Weg ist (s, ..., $\pi[\pi[v]]$, π[v], v).

4.6. Definition (Vorgängersubgraph)

Sei G ein gerichteter oder ungerichteter Graph. Sei $s \in V$ und π ergebe sich am Ende von BFS(G, s). Der *Vorgängersubgraph* von BFS(G, s) ist der Graph $G_\pi = (V_\pi, E_\pi)$ mit

$$V_\pi = \{v \in V \mid \pi[v] \neq nil \text{ oder } v = s\}$$
$$E_\pi = \{(\pi[v], v) \mid v \in V_\pi \setminus \{s\}\} \subseteq E$$

4.7. Definition (zusammenhängend)

Sei G = (V, E) ein ungerichteter Graph. G heißt *zusammenhängend* genau dann, wenn es für alle u, v $\in V$ einen Weg von u nach v gibt.

4.8. Folgerung

(a) G_π als ungerichteter Graph betrachtet ist ein kreisfreier, zusammenhängender Graph, d.h. ein Baum. s ist die Wurzel des Baums.

(b) In G_π gibt es für jedes $v \in V$ genau einen einfachen Weg von s nach v. Dieser ist zugleich ein kürzester Weg von s nach v in G. (Beachte, dass es in G mehrere kürzeste Wege geben kann, da G nicht kreisfrei sein muss.)

4.9. **Definition** (Zusammenhangskomponente)

Sei G = (V, E) ein ungerichteter Graph. Ein Teilgraph von G, H = (W, F) (d.h. $W \subseteq V$, $F \subseteq E$) ist eine *(Zusammenhangs-) Komponente* genau dann, wenn H ein maximaler zusammenhängender Teilgraph von G ist.

4.10. **Algorithmus** (Finden von Zusammenhangskomponenten)

Eingabe: Ein ungerichteter Graph G = (V, E).
Ausgabe: Array A[1..|V|] mit A[u] = A[v] $\Leftrightarrow$ u, v liegen in derselben Komponente.

1. Initialisierung von A mit *nil*
2. Initialisierung von BSF
3. for each $w \in V$
4. if color[w] = weiß
5. then BFS(G, w), aber so modifiziert, dass keine Initialisierung erfolgt, also A[u] = w, falls u im Verlauf aufgesucht wird

5. Tiefensuche in Graphen

5.1. **Algorithmus** (Tiefensuche, depth first search)

Gegeben sei der gerichtete Graph G = (V, E) mit V = {u, v, w, x, y, z} und E = {(u, v), (u, x), (v, y), (w, y), (w, z), (x, v), (y, x)}. Er ist in Abbildung 5.1 dargestellt.

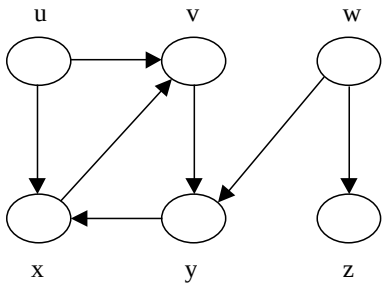

Abbildung 5.1

Tiefensuche im Beispielgraph von Abbildung 5.1, vgl. Abbildung 5.2:

Sei G = (V, E) ein (gerichteter oder ungerichteter) Graph. Es werden folgende Arrays definiert:

d[1..|V|] - Entdeckzeit (Discovery time), d.h. Zeitpunkt des ersten Aufsuchens eines Knotens. Der Knoten wird hellgrau eingefärbt.
f[1..|V|] - Beendezeit (Finishing time), d.h. Zeitpunkt des letzten Aufsuchens eines Knotens. Der Knoten wird dunkelgrau eingefärbt.
π[1..|V|] - Liste der Vorgängerknoten (wie bei der Breitensuche).

Prozedur DFS(G)

1. for each $u \in V$; Initialisierung
2. do color[u] := weiß
3. π[u] := nil
4. time := 0

5. for each u ∈ V
6. do if color[u] = weiß ; man fängt also mit irgendeinem Knoten an
7. then DFS-visit[u]

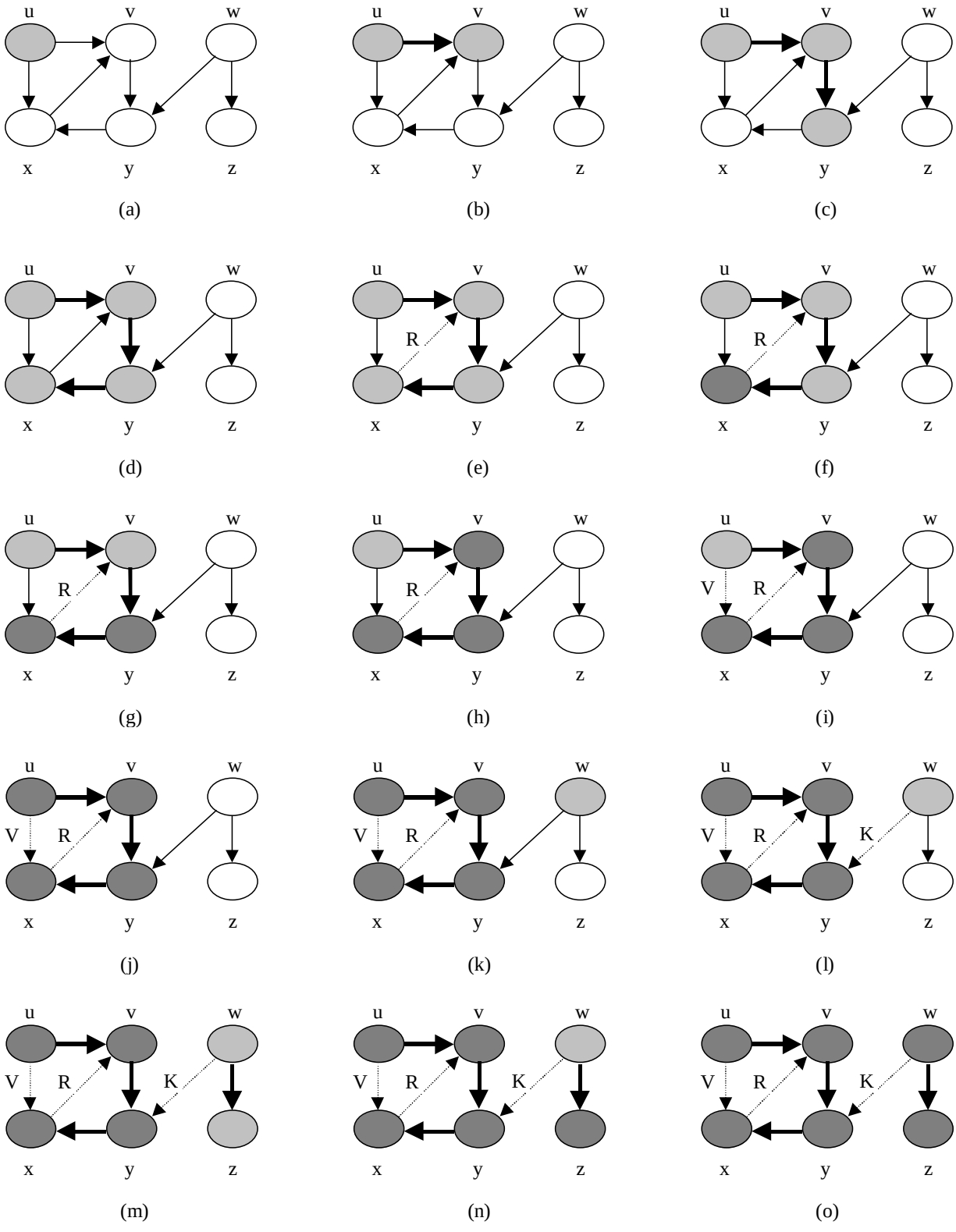

Abbildung 5.2

Prozedur DFS-visit(u)

1. color[u] := grau ; G und time werden als globale Parameter betrachtet

```
2.  d[u] := time
3.  time := time + 1
4.  for each v ∈ Adj[u]                ; die Kante (u, v) wird untersucht
5.     do  if color[v] = weiß
6.         then π[v] := u
7.              DFS-visit(v)           ; rekursiver Aufruf von DFS-visit
8.  color[u] := schwarz
9.  f[u] := time
10. time := time + 1
```

Es gilt: $\{f[u] \mid u \in V\} \cup \{d[u] \mid u \in V\} = \{1, 2, ..., 2 \cdot |V|\}$. Zwischen Zeiten und Färbung besteht folgender Zusammenhang:

Farbe des Knotens u	Zeitverhältnis
weiß	time < d[u]
grau	d[u] ≤ time < f[u]
schwarz	f[u] ≤ time

Beobachtungen

1. Wird DFS-visit(v) direkt oder vermittelt über weitere Aufrufe von DFS-visit(u) aus aufgerufen, d.h. wird v über u aufgesucht, dann gibt es zum Zeitpunkt d[u] einen Weg über lauter weiße Knoten von u nach v. Der Grund dafür ist, dass DFS-visit(v) nur aufgerufen wird, wenn color[v] = weiß.

2. Im Allgemeinen werden nicht alle von einem Knoten u aus in G erreichbaren Knoten über u aufgesucht.

3. Im Allgemeinen werden nicht alle von einem Knoten u aus in G erreichbaren Knoten, die zum Zeitpunkt d[u] weiß sind, über u aufgesucht.

5.2. Laufzeitbestimmung von DFS

Organisation des Hauptspeichers bei rekursiven Prozeduraufrufen, vgl. Abbildung 5.3.

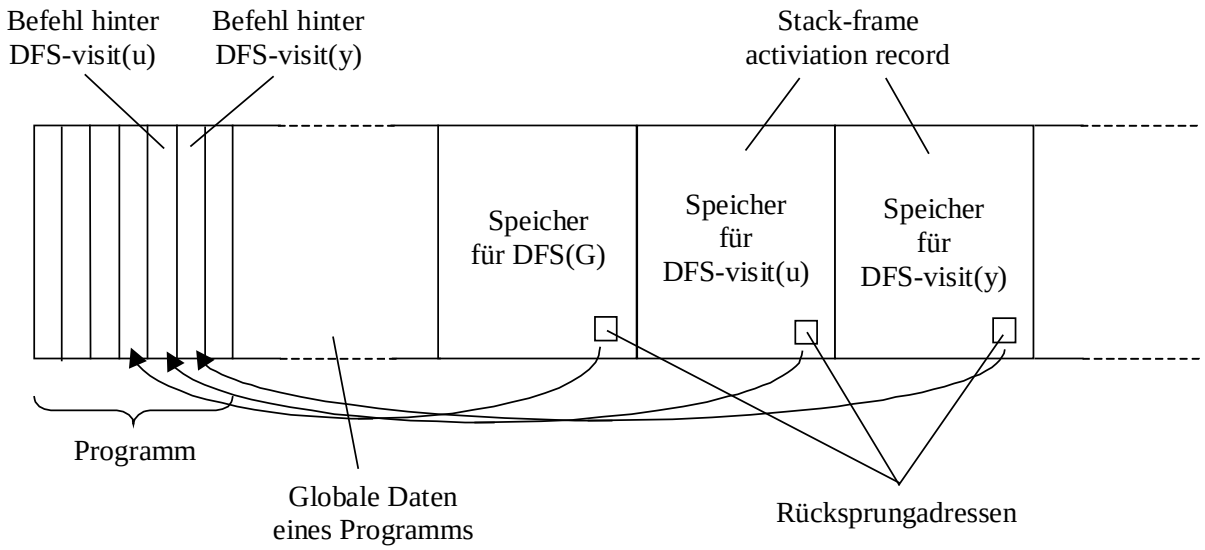

Abbildung 5.3

Für jeden Aufruf der Prozedur wird ein eigenes Stück des Hauptspeichers, genannt *Frame*, reserviert. Dort werden alle Variablen für den jeweiligen Aufruf samt ihren Werten abgelegt. Am Ende jedes dieser Stücke steht jeweils eine Rücksprungadresse ins Programm. Der Hauptspeicher wird Stack-artig verwaltet, d.h. man arbeitet jeweils am Ende des Speicherabschnitts. Der Verwaltungsaufwand für das Einrichten und Löschen eines Frames ist $\Theta(1)$.

Der Zeitaufwand für DFS und DFS-visit wird folgendermaßen berechnet: Der Durchlauf durch eine Programmzeile, inklusive Prozeduraufruf, erfordert die Zeit $\Theta(1)$. Die gesamte Zeit für das einmalige Durchlaufen von DFS-visit ist dann

Zeit bei Aufruf + Zeit im Prozedurrumpf

5.3. Satz (Laufzeitbestimmung von DFS)

Sei G = (V, E). Die Laufzeit von DFS(G) ist in $\Theta(|V| + |E|)$, also linear in der Größe der Adjazenzlistendarstellung.

5.4. Definition (Vorgängersubgraph)

Sei G = (V, E) ein Graph. Nach Abarbeiten von G durch den Algorithmus DFS lässt sich der Vorgängersubgraph von G definieren als der Graph $G_\pi = (V, E_\pi)$ mit

$$E_\pi = \{(\pi[v], v) \mid v \in V, \pi[v] \neq nil\} \subseteq E$$

G_π ist Teilgraph von G.

5.5. Folgerung

Die ungerichtete Version von G_π ist ein Wald, d.h. eine Menge von Bäumen. Die Wurzeln der Bäume sind die Knoten v mit $\pi[v] = nil$ als Wurzel. G_π stellt die Aufrufstruktur der Prozedur DFS-visit dar.

5.6. Definition (Kantenarten)

Sei G = (V, E) ein gerichteter Graph und sei $G_\pi = (V, E_\pi)$ der depth first-Wald, der sich bei der Ausführung von DFS(G) ergibt. Mittels G_π werden die Kanten von G in verschiedener Weise klassifiziert.

(a) (u, v) heißt *Baumkante* genau dann, wenn $(u, v) \in E_\pi$, d.h. v wird von u aus aufgesucht.

(b) (u, v) heißt *Rückwärtskante* genau dann, wenn $(u, v) \notin E_\pi$ und v ist (direkter oder indirekter) Vorgänger von u im depth first-Wald.

(c) (u, v) heißt *Vorwärtskante* genau dann, wenn $(u, v) \notin E_\pi$ und v ist (direkter oder indirekter) Nachfolger von u im depth first-Wald.

(d) (u, v) heißt *Kreuzkante* genau dann, wenn $(u, v) \notin E_\pi$ und u ist weder Nachfolger noch Vorgänger von v im depth first-Wald. u und v können im selben oder in verschiedenen Bäumen des Waldes liegen.

Diese Kantenarten sind im gerichteten Graphen wohldefiniert, da jede Kante nur einmal überquert wird. Im ungerichteten Graphen ist die Richtung des ersten Überquerens einer Kante maßgebend. Deshalb gibt es im ungerichteten Graphen keine Kreuz- und keine Vorwärtskanten.

5.7. **Satz** (Klammerstruktur)

Sei G = (V, E) ein gerichteter oder ungerichteter Graph. Sind u, v mit u ≠ v zwei Knoten, dann gelten nach Abarbeitung von DFS(G) die Aussagen (a) und (c) oder (b) und (c):

(a) $\{d[u], d[u] + 1, ..., f[u]\} \cap \{d[v], d[v] + 1, ..., f[v]\} = \varnothing$

(b) $\{d[v], d[v] + 1, ..., f[v]\} \subseteq \{d[u], d[u] + 1, ..., f[u]\}$

(c) v ist (direkter oder indirekter) Nachfolger von u in G_π genau dann, wenn $d[u] < d[v] < f[v] < f[u]$

5.8. **Satz** (Weißer-Weg-Satz)

In einem depth first-Wald ist v (direkter oder indirekter) Nachfolger von u genau dann, wenn es zur Zeit d[u] einen weißen Weg von u nach v gibt.

5.9. **Lemma**

Ist G = (V, E) ein ungerichteter Graph, dann ist jede Kante entweder Baum- oder Rückwärtskante.

6. Anwendung der Tiefensuche (I): Topologisches Sortieren

6.1. **Definition** (Topologische Sortierung)

Sei G = (V, E) ein gerichteter azyklischer Graph, kurz *dag* (directed acyclic graph). Eine topologische Sortierung von G ist eine totale Anordnung der Knoten von G

$$v_1 < v_2 < v_3 < ... < v_n$$

wobei $V = \{v_1, v_2, v_3, ..., v_n\}$, $v_i \neq v_j$ für $i \neq j$, so dass für alle Kanten $(v_i, v_j) \in E$ gilt: $v_i < v_j$.

Beispiel

Gegeben sei der dag von Abbildung 6.1. Er kann z.B. als Darstellung eines Plans für die sequentielle Ausführung von Jobs mit Abhängigkeiten in der Fertigung interpretiert werden.

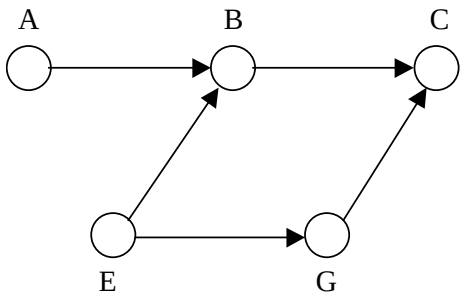

Abbildung 6.1

6.2. **Algorithmus** (Topologische Sortierung)

Eingabe ist ein dag G = (V, E)

Prozedur Top-Sort(G)

1. Rufe DFS(G) so modifiziert auf, dass jeder Knoten v zur Zeit f[v] an den Anfang einer Liste eingefügt wird.
2. Die Liste stellt eine topologische Sortierung dar mit dem ersten Element als kleinstem Element.

Der Algorithmus benötigt (wie DFS) die Zeit $\Theta(|V| + |E|)$.

6.3. **Lemma**

Ein gerichteter Graph G = (V, E) ist azyklisch genau dann, wenn DFS(G) keine Rückwärtskante liefert.

6.4. **Satz** (Korrektheit von Top-Sort(G))

Top-Sort(G) erzeugt eine topologische Sortierung eines dag G.

7. Anwendung der Tiefensuche (II): Starke Zusammenhangskomponenten

7.1. **Definition** (Starke Zusammenhangskomponente)

(a) Sei G = (V, E) ein gerichteter Graph. G heißt *stark zusammenhängend* genau dann, wenn es für alle u, $v \in V$ einen Weg von u nach v und einen Weg von v nach u gibt.

(b) Sei G = (V, E) ein gerichteter Graph. Ein Teilgraph H = (W, F) von G ist eine starke Zusammenhangskomponente genau dann, wenn H ein maximaler Teilgraph und stark zusamenhängend ist.

7.2. **Definition** (Partition)

Sei M eine Menge und seien $S_1, S_2, ..., S_k \subseteq M$ so dass für alle i, j mit $i \neq j$ gilt: $S_i \cap S_j = \emptyset$ und $S_1 \cup S_2 \cup ... \cup S_k = M$. Dann heißen die Mengen $S_1, S_2, ..., S_k$ eine *Partition* von M.

7.3. **Lemma** (Partition durch starke Zusammenhangskomponenten)

Sind $H_1 = (W_1, F_1)$, $H_2 = (W_2, F_2)$, ..., $H_k = (W_k, F_k)$ die starken Zusammenhangskomponenten des Graphen G = (V, E), dann ist $W_1, W_2, ..., W_k$ eine Partition von V. Es gilt auch $F_i \cap F_j = \emptyset$ für $i \neq j$, aber $F_1 \cup F_2 \cup ... \cup F_k \subset E$.

7.4. **Definition** (Umkehrung eines gerichteten Graphen)

Sei G = (V, E) ein gerichteter Graph. Die *Umkehrung* von G, $G^U = (V, E^U)$ ist definiert durch

$$E^U = \{(u, v) \mid (v, u) \in E\}$$

Die Adjazenzlistendarstellung von G^U kann in Zeit $\Theta(|V| + |E|)$ aus der von G ermittelt werden. Dies geschieht dadurch, dass man für alle Elemente k_1, k_2, ..., k_n von Adj[i] in G in die Adjazenzlisten $Adj^U[k_1]$, $Adj^U[k_2]$, ..., $Adj^U[k_n]$ den Wert i einträgt.

Beispiel

Abbildung 7.1 zeigt einen gerichteten Graph und seine Umkehrung. Die Knoten bleiben also gleich, die Kanten ändern ihre Richtung.

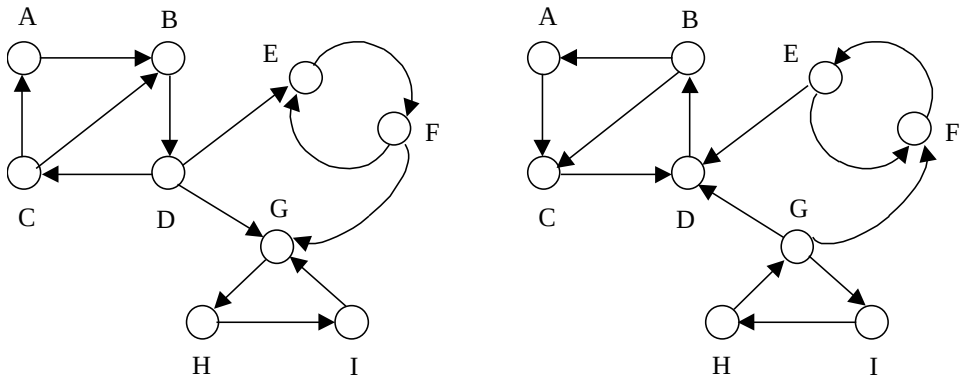

Abbildung 7.1

7.5. **Algorithmus** (Starke Zusammenhangskomponenten)

Eingabe ist ein gerichteter Graph G = (V, E).

Prozedur Komp(G)

1. Berechne DFS(G) und speichere alle Knoten in einer Liste (v_1, v_2, ..., v_n), so dass $f[v_i] > f[v_{i+1}]$ für alle i = 1, ..., n-1.

   v_n ist also der Knoten, der zuerst schwarz gefärbt wird, v_1 zuletzt.
2. Ermittle G^U.
3. Berechne DFS(G^U) und besuche dabei die Knoten in der Hauptschleife von DFS nach absteigenden Werten f[u] (gemäß der Liste aus 1.). Notiere die Knoten jedes Baums in einer Liste.
4. Gib die Listen aus 3. als Knoten der starken Zusammenhangskomponenten aus.

Die Laufzeit von Komp(G) ergibt sich im Wesentlichen aus dem zweimaligen Aufruf von DFS, einmal für G und einmal für G^U, und ist damit $\Theta(|V| + |E|)$.

7.6. Lemma

(a) Liegen die Knoten u und v in derselben starken Zusammenhangskomponente, dann gibt es keinen Weg von u nach v, der diese Komponente verlässt, d.h. es gibt keinen Weg, in auf dem ein Knoten w liegt, der nicht zu der Komponente gehört.

(b) Alle Knoten einer starken Zusammenhangskomponente liegen bei jeder Tiefensuche in einem einzigen Baum.

7.7. **Definition** (Stammvater)

Sei eine Tiefensuche in einem gerichteten Graphen G gegeben. Der Stammvater (forefather) eines Knoten u, $\Phi(u)$, ist der eindeutig bestimmte Knoten mit den Eigenschaften

(i) $\Phi(u)$ ist in G von u aus erreichbar, d.h. es gibt in G einen Weg von u nach $\Phi(u)$.
(ii) $\Phi(u)$ hat die maximale Beendezeit unter allen Knoten, die von u aus erreichbar sind.

7.8. Lemma

Für alle Knoten u eines gerichteten Graphen gilt:

(a) $f[u] \leq f[\Phi(u)]$

(b) $\Phi(\Phi(u)) = \Phi(u)$

7.9. Satz

Sei G = (V, E) ein gerichteter Graph und sei eine feste Tiefensuche auf G gegeben. Dann gilt für jeden Knoten u: $\Phi(u)$ ist Vorgänger von u bezüglich der gegebenen Tiefensuche.

7.10. Korollar

Sei G = (V, E) ein gerichteter Graph. Es gelten die folgenden Aussagen.

(a) Bei jeder Tiefensuche in G liegen für einen beliebigen Knoten u sowohl u als auch $\Phi(u)$ in derselben starken Zusammenhangskomponente.

(b) Sei eine Tiefensuche in G gegeben. Für alle $u, v \in V$ gilt: u und v sind in derselben starken Zusammenhangskomponente genau dann, wenn $\Phi(u) = \Phi(v)$.

(c) Sei eine Tiefensuche in G gegeben und $M \subseteq V$. M ist die Menge der Knoten einer starken Zusammenhangskomponente genau dann, wenn es ein $u \in M$ gibt, so dass $M = \{v \mid \Phi(v) = u\}$.

7.11. Satz

Die Tiefensuche durch $DFS(G^U)$ in Algorithmus Komp ergebe den Wald T_1, ..., T_k in dieser Reihenfolge. Dann gilt: Für jedes j = 1, ..., k ist

Knoten(T_j) = Menge der Knoten einer starken Zusammenhangskomponente

8. Anwendung der Tiefensuche (III): Zweifache Zusammenhangskomponenten

Im Folgenden ist stets G = (E, V) ein ungerichteter, zusammenhängender Graph, d.h. $|E| \geq |V| - 1$.

8.1. **Definition** (Zweifach zusammenhängend)

G heißt *zweifach zusammenhängend* genau dann, wenn für alle $v \in V$ der Graph $G' = (V - \{v\}, E - Adj(v))$ zusammenhängend ist.

8.2. **Definition** (Trennende Knotenmenge)

Seien A, B $\subseteq$ V. Eine Menge X $\subseteq$ V *trennt* A und B genau dann, wenn jeder Weg von einem Knoten in A zu einem Knoten in B mindestens einen Knoten aus X enthält. Wege dieser Art werden A-B-Wege genannt.

8.3. **Satz** (Zweifach zusammenhängend)

G ist zweifach zusammenhängend genau dann, wenn es für je zwei Knoten u, v $\in$ V mindestens zwei Wege der Form (u, u_1, ..., u_m, v) und (u, u_1', ..., u_l', v) mit m, $l \geq 0$ gibt, so dass die Mengen der Zwischenknoten disjunkt sind, d.h. $\{u_1, ..., u_m\} \cap \{u_1', ..., u_l'\} = \emptyset$.

8.4. Beispiel

Der Graph von Abbildung 8.1 stellt zwei zweifach zusammenhängende Graphen dar.

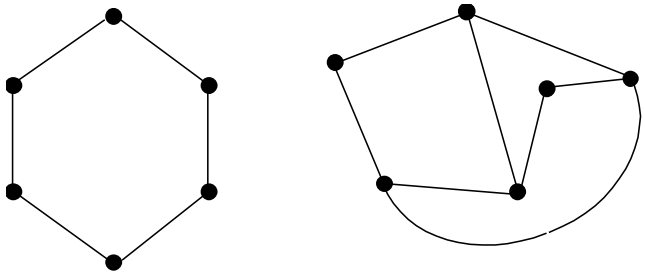

Abbildung 8.1

8.5. Definition

(a) Ein Teilgraph H = (W, F) von G ist eine *zweifache Zusammenhangskomponente* von G genau dann, wenn H ein bezüglich Knoten und Kanten maximaler zweifach zusammenhängender Teilgraph ist.

(b) Eine zweifache Zusammenhangskomponente, die nur aus zwei Knoten und einer sie verbindenden Kante besteht, heißt *Brückenkante*.

(c) Ein Knoten v $\in$ V ist ein *Artikulationspunkt* von G genau dann, wenn G\{v} nicht zusammenhängend ist.

8.6. Lemma

Sei eine Tiefensuche in G gegeben. Der zugehörige depth first-Baum sei $G_\pi = (V, E_\pi)$. Dann gilt:

(a) Die Wurzel r von G_π ist Artikulationspunkt von G genau dann, wenn r mindestens zwei Söhne in G_π hat.

(b) Sei a nicht Wurzel von G_π. a ist Artikulationspunkt von G genau dann, wenn es in G_π einen Sohn s von a gibt, so dass es keine Rückwärtskante von einem Nachfolger von s (einschließlich s) zu einem echten Vorgänger von a (also ausschließlich a) gibt.

8.7. **Definition** (low[v])

Sei eine Tiefensuche in G = (V, E) gegeben und sei v $\in$ V. In Abhängigkeit von dieser Tiefensuche wird definiert: