Grundlagen der Informatik

Vorlesungsskript

Falk Jonatan Strube

Vorlesung von Dr. Boris Hollas

27. November 2015

Inhaltsverzeichnis

Allgemeine Informationen

Zugelassene Hilfsmittel Klausur: A-4 Blatt (doppelseitig, handbeschrieben)
Prüfungsvorleistung: alle paar Woche eine Lernabfrage in der Vorlesung (Bestanden wenn insgesamt im Schnitt 50%)
Grundlage der Vorlesung: Grundkurs theoretische Informatik [1]
Lernkontrolle ab 23.10.2015 alle zwei Wochen.

1 Aussagenlogik

Mit der Aussagenlogik lassen sich Aussagen formulieren, die entweder wahr oder falsch sind. Aussagen sind atomare Aussagen wie „die Straße ist nass“ oder mit Hilfe von logischen Operatoren zusammengesetzte Aussagen.

1.1 Syntax und Semantik

Definition: Die *Formeln der Aussagenlogik* sind induktiv definiert.

- Jede atomare Aussage ist eine Formel der Aussagenlogik. Diese heißen Atomformeln oder Variablen. Atomformeln bezeichnen wir mit Kleinbuchstaben oder durch Wörter in Kleinbuchstaben.
- Wenn F, G Formeln der Aussagenlogik sind, dann auch $(F \wedge G)$, $(F \vee G)$, $(\neg F)$.

Bsp.: Formeln der Aussagenlogik sind x, y, $x \wedge y$, $(x \wedge (y \wedge z)) \vee (\neg x \wedge (y \wedge \neg z))$, $regnet$, $regnet \wedge nass$, da sie jeweils aus atomaren Aussagen die nach der Definition zusammensetzen lassen bestehen.
Keine Formeln der Aussagenlogik sind $x\wedge$, $\vee x$, $x \wedge \vee y$.
Um Klammern zu sparen, legen wir Prioritäten fest:

Operator	Priorität
$\neg$	höchste
$\wedge$, $\vee$	
$\rightarrow$, $\leftrightarrow$	niedrigste

Definition: Eine *Belegung* einer Formel F der Aussagenlogik ist eine Zuordnung von Wahrheitswerten „wahr“ (1) oder „falsch“ (0) zu den Atomarformeln in F. Daraus ergibt sich der *Wahrheitswert* einer Formel:

- Eine Atomformel ist genau dann wahr, wenn sie mit „wahr“ belegt ist.
- Die Formel $F \wedge G$ ist genau dann wahr, wenn F „wahr“ ist und G „wahr“ ist.
  $F \vee G$ ist wahr, wenn F wahr ist oder G wahr.
  $\neg F$ ist wahr, wenn F falsch ist.

F	G	$F \wedge G$	$F \vee G$	$\neg F$	$F \rightarrow G$	$F \leftrightarrow G$
0	0	0	0	1	1	1
0	1	0	1	1	1	0
1	0	0	1	0	0	0
1	1	1	1	0	1	1

Bsp.: Wenn $regnet$ bedeutet: „Es regnet."
Wenn $nass$ bedeutet: „Die Straße ist nass."
Dann bedeutet $regnet \wedge nass$: „Es regnet und die Straße ist nass."
„Wenn es regnet, dann ist die Straße nass" ($regnet \rightarrow nass$). Es muss nur der Fall ausgeschlossen werden, der nicht eintreffen kann: $\neg(regnet \wedge \neg nass) \Rightarrow$ Folgendes darf nicht eintreffen: „Es regnet und die Straße ist nicht nass".

Alles andere („Es regnet nicht und die Straße ist nicht nass", „Es regnet nicht und die Straße ist nass" und „Es regnet und die Straße ist nass") darf eintreffen.

$regnet$	$nass$	$\neg(regnet \wedge \neg nass) = \neg regnet \vee nass$
0	0	1
0	1	1
1	0	0
1	1	1

Definition: Die Operatoren $\rightarrow$ (*Implikation*) und $\leftrightarrow$ (*Äquivalenz*) sind definiert durch:

- $F \rightarrow G = \neg F \vee G$
- $F \leftrightarrow G = (F \rightarrow G) \wedge (G \rightarrow F)$

(Siehe Tabelle oberhalb)

Bsp.: Berechnen des Betrags y einer Zahl x:

```
if (x>= 0)
  y=x;
else
  y=-x;
```

Dargestellt als Formel der Assagenlogik: $((x \geq 0) \rightarrow y = x) \wedge (\neg(x \geq 0) \rightarrow y = -x)$

Definition: Eine Formel F der Aussagenlogik heißt

- *erfüllbar*, wenn es eine Belegung gibt, sodass F wahr ist, sonst *unerfüllbar*. Mit $\perp$ bezeichnen wir eine unerfüllbare Formel (Widerspruch).
- *Tautologie* oder *gültig*, wenn F für jede Belegung wahr ist. Bezeichnung: $\top$

Bsp.:

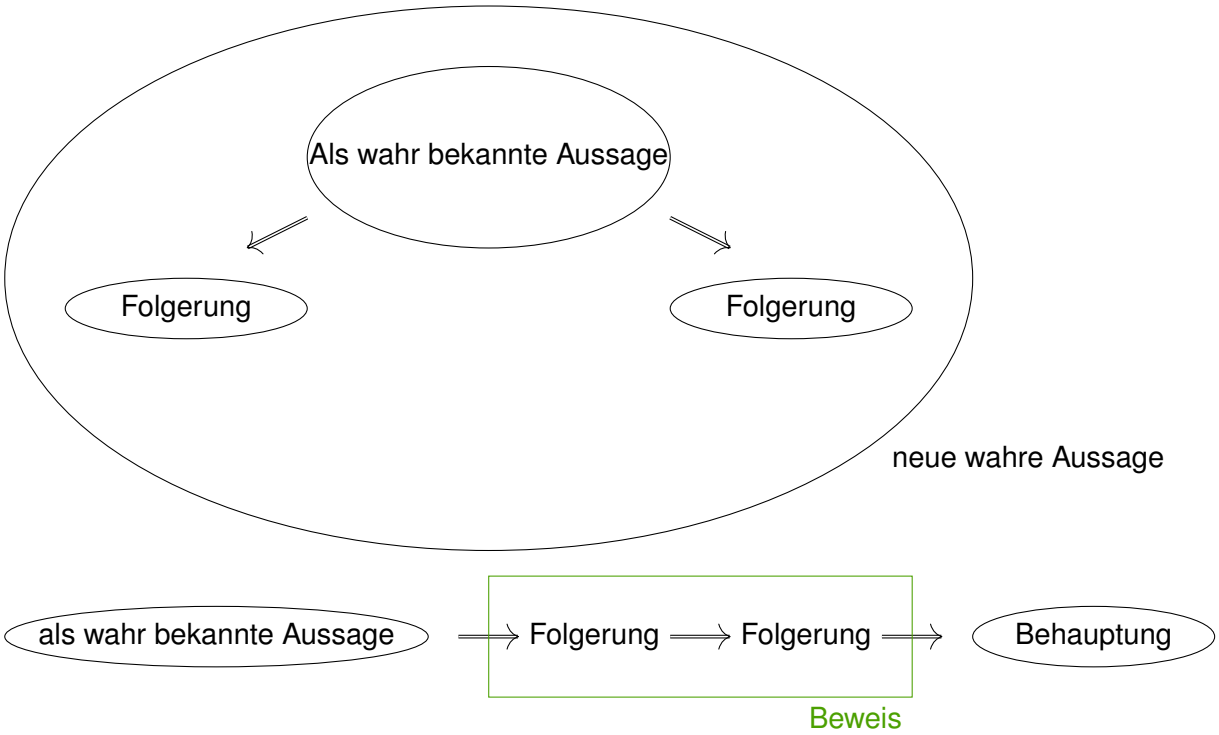
- $x \wedge y$ ist erfüllbar.
- $((\neg x \wedge y) \vee (x \wedge \neg y)) \wedge \neg(x \vee y)$ ist unerfüllbar (linke Seite: entweder x oder y falsch - rechte Seite: x oder y falsch)
- $x \vee \neg x$ ist eine Tautologie

Definition: Wir schreiben $F \equiv G$ („F ist äquivalent zu G"), wenn für jede Belegung gilt: $F \leftrightarrow G$ wahr (d.h., $F \leftrightarrow G$ ist gültig).

1.2 Rechenregeln

siehe Mathematik I

2 Beweistechniken



Direkter Beweis

Bsp.: Wenn $a \in \mathbb{Z}$ gerade ist, dann ist auch a^2 gerade.
($a \in \mathbb{Z}$ gerade $\Rightarrow a^2$ gerade)
Beweis:

- Wenn a gerade ist, gibt es ein n mit $a = 2 \cdot n$.
- Dann gilt $a^2 = 4 \cdot n^2 = 2 \cdot 2n^2$,
- woraus a^2 gerade folgt.

Indirekter Beweis Mit einem indirekten Beweis wird $A \Rightarrow B$ bewiesen, indem die äquivalente Aussage $\neg B \Rightarrow \neg A$ bewiesen wird.

Bsp.: Wenn a^2 gerade ist, dann auch a.
(a^2 gerade $\Rightarrow a$ gerade)
Beweis: Wir zeigen: Wenn a ungerade ist, dann auch a^2.

- Aus a ungerade folgt $a = 2n - 1$ für ein n.
- Dann ist $a^2 = 4n^2 - 4n + 1 = \underbrace{4(n^2 - n)}_{gerade} + \underbrace{1}_{ungerade}$,
- Aus gerade + ungerade folgt ungerade, woraus a^2 ungerade folgt.

Beweis durch Widerspruch Mit einem Beweis durch Widerspruch wird eine Aussage A bewiesen, indem gezeigt wird, dass die Annahme „A ist falsch“ zu einem Widerspruch führt.
(D.h., es wird $\neg A \to \bot$ gezeigt)

Bsp.: $\sqrt{2}$ ist irrational. Siehe Mathematik I.

Vollständige Induktion Mit einer vollständigen Induktion lassen sich Aussagen der Art „für alle $n \in \mathbb{N}$ gilt . . . “ beweisen.
Prinzip: Gegeben eine Aussage der Form „für alle $n \in \mathbb{N}$ gilt $A(n)$“

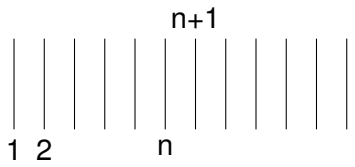
- *Induktionsanfang*: Man zeigt, die Wahrheit der Aussage für $n = 1$ (mit anderen Worten: Man zeigt, dass $A(1)$ wahr ist) [1: die kleinste mögliche Zahl $\Rightarrow$ kann auch 0 oder eine andere sein]
- *Induktionsvorussetzung*: Die Aussage ist für n wahr.
- *Induktionsschritt*: Wenn IV wahr ist, dann ist die Aussage auch für $n + 1$ wahr.

In Formeln: Man zeigt

- IA: $A(1)$
- IV: $A(n)$
- IS: für alle n: $A(n) \Rightarrow A(n + 1)$

Beispiel Dominosteine: Wenn der erste Stein fällt, fällt auch der Zweite. Und wenn der n-te Stein fällt, fällt auch der $n + 1$-te:

- IA: 1. Umstoßen
- IV: Wenn der vorherige Stein umfällt, fällt auch der nächste
- IS: Wenn n-ter umgestoßen wird, dann auch $n + 1$-ter



Bsp.: Für alle $n \geq 1$ gilt $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Beweis (Induktion):

IA $n = 1$: $1 = \frac{1 \cdot 2}{2}$ ist wahr.

IV Es gelte $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ ist wahr.

IS $n \to n+1$: Zu zeigen: $\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}$ Es gilt:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= (\sum_{k=1}^{n} k) + n + 1 \\ &\overset{IV}{=} \frac{n(n+1)}{2} + n + 1 \\ &= ... = \frac{(n+1)(n+2)}{2} \#\end{aligned}$$

3 Elementare Kombinatorik

Kreuzprodukt:
$A \times B = \{(a,b) | a \in A, b \in B\}$
$A^n = \underbrace{A \times ... \times A}_{n}$
Die *Potenzmenge* einer Menge M ist die Menge aller Teilmengen von M: $\mathcal{P}(M) = \{A | A \subseteq M\}$

Bsp.: $\mathcal{P}(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$

Definition: Die Mächtigkeit einer Menge A ist die Anzahl ihrer Elemente. Notation: $|A|$

Satz: Es gilt $|A^n| = |A|^n$

Beweis: Nach Def. ist $A^n = \{(a_1, ..., a_n) | a_1, ..., a_n \in A\}$. Um das n-Tupel $(a_1, ..., a_n)$ zu erzeugen, gibt es $|A|$ viele Möglichkeiten. Insgesamt gibt es daher $|A|^n$ Möglichkeiten das n-Tupel $(a_1, ..., a_n)$ auszuwählen.

Bsp.: Eine PIN bestehe aus 6 Ziffern. Mit $A = \{0, ..., 9\}$ ist A^6 die Menge aller PINs. Mit obigen Satz folgt: Die Anzahl aller PINs ist $|A^6| = |A|^6 = 10^6$

Bsp.: In dem Programm

```
for (i=1 to n)
  for (j=1 to n)
    a[i][j]=i+j;
```

werden alle Paare (i,j) erzeugt. Die Anzahl der Paare ist $|\{1, ..., n\}^2| = |\{1, ..., n\}|^2 = n^2$. Es gibt daher n^2 Schleifendurchläufe.

Satz: $|\mathcal{P}(M)| = 2^{|M|}$

Beweis: Für $M = \{m_1, ..., m_n\}$ identifizieren wir eine Teilmenge $A \subseteq M$ durch das n-Tupel $(a_1, ..., a_n)$ mit $a_k \begin{cases} 0 \text{ für } M_k \notin A \\ 1 \text{ für } m_k \in A \end{cases}$. Nach obigen Satz gibt es $|\{0,1\}^n| = 2^n = 2^{|M|}$ derartige Tupel.

Definition: Für eine n-elementige Menge ist $\binom{n}{k}$ die Anzahl ihrer k-elementigen Teilmengen $(n \geq k \geq 0)$.

Bsp.:
$\binom{n}{0} = 1$, da $\emptyset$ die einzige 0-elementige Teilmenge ist.
$\binom{n}{n} = 1$, da es nur eine n-elementige Teilmenge gibt (die Menge selber).
$\binom{n}{1} = n$, da es n 1-elementige Teilmengen gibt.
$\binom{n}{2} = \frac{n(n-1)}{2}$, denn für das 1. Element gibt es n Möglichkeiten, für das 2. Element $n-1$ Möglichkeiten. Da das Element $\{a, b\} = \{b, a\}$ hierbei doppelt gezählt wird, müssen wir durch 2 teilen.

Definition: Eine Permutation der Folge $1, ..., n$ ist eine neue Anordnung dieser Folge.

Bsp.: Alle Permutationen von $1, 2, 3$ sind $1, 2, 3$; $1, 3, 2$; $2, 1, 3$; $2, 3, 1$; $3, 1, 2$; $3, 2, 1$.

Definition: $n! = 1 \cdot ... \cdot n \qquad 0! = 1.$

Satz: Es gibt $n!$ Permutationen von n Zahlen.

Beweis: Für die 1. Stelle gibt es n Möglichkeiten, für die 2. Stelle $n-1$ usw. Für die letzte Stelle nur noch eine Möglichkeit. Insgesamt also $n \cdot ... \cdot 1 = n!$ Möglichkeiten.

Satz: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Beweis: Um aus einer n-elementigen Menge k Elemente auszuwählen, gibt es n Möglichkeiten, um das erste Element auszuwählen, für das zweite Element $n-1$ Möglichkeiten, ..., für das k. Element $n-k+1$ Möglichkeiten, insgesamt daher $n \cdot ... \cdot (n-k+1)$ Möglichkeiten. Da die Reihenfolge, in der diese k Elemente ausgewählt werden, keine Rolle spielt, muss dieses Produkt durch $k!$ geteilt werden.
Daher erhalten wir $\binom{n}{k} = \frac{n \cdot ... \cdot (n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$

4 O-Notation

Mit Hilfe der O-Notation lassen sich obere Schranken für die Laufzeit eines Algorithmus angeben (Abschätzung mit $\leq$, die die maximale Laufzeit eines Algorithmus angibt, bspw. $\leq c \cdot n^2$). Um die Laufzeit eines Algorithmus zu messen, bestimmen wir die Anzahl Schritte und geben mit Hilfe der O-Notation deren Größenordnung in Abhängigkeit der Länge der Eingabe an.

Beispiel: lineare Suche

```
int lsearch (int a[], int n, int k) {
  int i;
  for (i=0; i<n; i++)
    if ( a[i] == k) return 1; //gefunden
  return 0; // nicht gefunden
}
```

Laufzeit dieser Funktion:

$$\leq \underbrace{c_1 + n \cdot c_2 + c_3}_{g(n)} \overset{\text{Abschätzung}}{\leq} (c_1 + c_2 + c_3) \cdot n = c \cdot n$$

c_1 ... Deklarierung von i
c_2 ... Vergleich der Werte in der Schleife (in n Schleifedurchläufen)
c_3 ... Ausführung return
(Durch den Worst-Case von annähernd unendlich vielen Durchläufen spielen die Konstanten, egal wie groß, keine besonder Rolle mehr und können, wie in der Abschätzung zu sehen, zusammengefasst werden).
Die Laufzeit der linearen Suche liegt in $O(n)$

Definition: Für eine Funktion $f > 0$ ist $O(f)$ die Menge aller Funktionen g, für die gilt:
$g(n) \leq c \cdot f(n)$ für ein $c > 0$ für alle großen n.
ABB 31

Bsp.: $2n^3 - n + 5 \overset{1.)}{\leq} 2n^3 + 5 \leq 7n^3 \in O(n^3)$

1.) $-n$ ist kleiner Null, deswegen ist die rechte Seite ohne $-n$ nachgewiesener Maßen größer (Vorgehensweise Ungleichung aufstellen (siehe auch folgende): weg lassen, was kleiner Null ist; mit n^x o.ä. erweitern, um auszuklammern).

```
for (i=0; i<n-1; i++)
  for (j=i+1 ; j<n ; j++)
    if( a[i] == a[j] ) return 1;
return 0;
```

Die if-Anweisung wird höchstens $\binom{n}{2}$ mal durchlaufen. Die Laufzeit ist daher $\leq c_1 \cdot \binom{n}{2} + c_2 \leq (c_1+c_2) \cdot \binom{n}{2} = \frac{c_1+c_2}{2} \cdot n \cdot (n-1) \leq \frac{c_1+c_2}{2} \cdot n^2 \in O(n^2)$ (mit c_i ... Zeiteinheiten für Rechenaufwand).

Bsp.: $2 \cdot log(n^2 + 1)$
$\leq 2 \cdot log(n^2(1+1))$
$= 2log(2n^2) = 2(log(2) + log(n^2))$
$\leq 2(log(n) + 2log(n))$
$\leq 6log(n) \in O(log(n))$
Schneller mit:
$n^2 + 1 \leq n^3 \Leftrightarrow$
$\frac{1}{n} + \frac{1}{n^3} \leq 1 \Rightarrow$

$0 \leq 1$ für $n \to \infty$
$2 \cdot log(n^2 + 1) \leq 2log(n^3) = 6log(n) \in O(log(n))$

5 Graphen

ABB 41

Definition: Ein (ungerichteter) Graph ist ein Paar $G = (V, E)$, wobei

- V die Menge der Knoten und
- E die Menge der Kanten ist, die aus ungeordneten Paaren $\{u, v\}$ von Knoten besteht (also ungerichtet).

Bsp.:
ABB42
$(\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 4\}, \{3, 4\}\})$
(ABB43: Die Punkte und Kanten eines 3D-Objektes werden auf einen Graph abgebildet.)

Definition: Ein Graph heißt *vollständig*, wenn alle Knoten paarweise verbunden sind.

Ein vollständiger Graph mit n Knoten besitzt genau $\binom{n}{2}$ Kanten (jeder Knoten hat den Grad $n - 1$).

Definition: Ein Knoten v hat den Grad k, wenn v mit genau k anderen Knoten verbunden ist.
Notation: $deg(v) = k$

Satz: Für jeden Graphen gilt $\sum_{v \in V} deg(v) = 2|E|$ (Sprich: Die Summe der Grade aller Knoten ist die zweifache Kanten-Anzahl).

Beweis: Wenn wir jede Kante in der Mitte durchschneiden, ist jeder Knoten mit genau $deg(v)$ Hälften verbunden. Die Summe der Knotengrade ist dann die Anzahl der Kantenhälften, und diese ist $2|E|$.

Definition: Ein *Weg* ist eine Folge von Knoten $v_1, ..., v_k$ mit $\{v_l, v_{l+1}\} \in E$ für $l = 1, ..., k - 1$. Die Länge dieses Weges ist $k - 1$. Ein Weg heißt Kreis, wenn $v_1 = v_k$.
ABB 44
ABB 45 ist ein Graph: $(\{1, 2, 3, 4, 5\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{4, 5\}\}$
ABB 46 ist ein Graph: $(\{1, 2, 3, 4\}, \emptyset\}$
Ein Graph heißt *zusammenhängend*, wenn es für alle Paare von Knoten u, v einen Weg von u nach v gibt.
Ein *Pfad* ist ein Weg, der keinen Knoten mehrfach enthält.

5.1 Bäume

Definition: Ein Baum ist ein zusammenhängender Graph der keine Kreise enthält. Ein Blatt ist ein Knoten v mit $deg(v) \leq 1$ (dem Grad 1, also nur eine Kante hat).
Anmerkung: Auch ein Graph mit nur einem Knoten ist ein Baum - ein Baum der keine Blätter hat.

Satz: Sei $B = (V, E)$ ein Baum. Dann gilt $|E| = |V| - 1$.

Beweis: (Indkution)

IA: $|V| = 1$: Ein Baum mit nur einem Knoten enthält keine Kanten.

IV: $|E| = |V| - 1$.

IS: $|V| \to |V| + 1$: Sei B ein Baum mit $|V| + 1$ Knoten. B besitzt ein Blatt (siehe Übung). Indem wir dieses Blatt zusammen mit der zugehörogien Kante entfernen, erhalten wir ein Baum B' mit $|V|$ Kanten und nach Induktionsvoraussetzung $|V| - 1$ Kanten. Damit besitzt B $(|V| + 1) - 1$ Kanten.

Definiton: Ein *Wurzelbaum* ist ein Baum mit einem als Wurzel ausgezeichnetem Knoten.

Definition: Ein *binärer Wurzelbaum* ist ein Wurzelbaum, in dem jeder Knoten, der kein Blatt ist, genau zwei Nachfolger besitzt.
ABB 4

Definition: (induktiv) ABB 5

- Ein einzelner Knoten ist ein binärer Wurzelbaum
- Wenn W_1, W_2 binäre Wurzelbäume sind, dann erhalten wir einen neuen Wurzelbaum, indem die Wurzeln von W_1, W_2 mit einer neuen Wurzel verbunden werden.

Satz: Ein binärer Wurzelbaum mit Tiefe d (d.h. alle Pfade von Wurzel zu einem Blatt haben die Länge d) besitzt genau 2^d Blätter. ABB 6

Beweis: (Indkution)

IA: $d = 0$: Ein binärer Wurzelbaum, der nur aus der Wurzel besteht, enthält $2^0 = 1$ Blätter.

IV: $|V| = 2^d$

IS: $d \to d + 1$: Ein binärer Wurzelbaum der Tief $d + 1$ enthält zwei binäre Wurzelbäume (laut vorhergehender Definition) der Tiefe d.
ABB 7
Diese enthalten nach Induktionsvoraussetzung jeweils 2^d Blätter. Folglich besitzt der binäre Wurzelbaum der Tiefe $d + 1$ genau $2 \cdot 2^d = 2^{d+1}$ Blätter.

5.2 Datenstrukturen zur Repräsentation

Es gibt zwei Möglichkeiten, um Graphen darzustellen:

Adjazenzmatrix Für einen Graphen $G = (V, E)$ ist die Adjazenzmatrix eine $|V| \times |V|$-Matrix (a_{uv}) mit $a_{uv} = \begin{cases} 1 & \text{für } \{u, v\} \in E \\ 0 & \text{sonst} \end{cases}$

Bsp.: ABB 8

Adjazenzliste Ein Array hat den Nachteil, dass es nicht in der Länge geändert werden kann. Der Vorteil ist allerdings, dass auf Elemente des Arrays in kurzer Zeit zugegriffen werden kann.
Eine Liste kann wachsen und schrumpfen. Jedes Glied einer Liste verweist auf das nächste. Der Nachteil ist, dass der Zugriff auf Elemente aus der Liste nicht so schnell und einfach ist.
Die Adjazenzliste ist ein Array, das an jeder Position v eine Liste der mit v verbundenen Knoten enthält.

Bsp.:
ABB 9

Bäume, insbesondere Binärbäume, lassen sich noch einfacher darstellen: Jeder Knoten wird dargestellt durch eine Datenstruktur, die einen Verweis auf die Nachfolger enthält.

Bsp.: ABB 10

5.3 Grundlegende Graphalgorithmen

5.3.1 Breitensuche

ABB 11
Mit der Breitensuche kann ein Graph systematisch durchsucht werden. Von einem Startknoten ausgehend, besucht die Breitensuche zuerst die dem Startknoten benachbarten Knoten. Anschließend werden die noch nicht besuchten Nachbarn dieser Knoten besucht, usw., bis das Ziel gefunden wurde oder alle Knoten besucht wurden.

```
boolean bfs (node start, node goal){
  for (v ∈ V){
    discovered[v] = false
  }
  queue.enqueue(start)
  discovered[start] = true
  while (¬ queue.isEmpty){
    u = queue.dequeue()
    if (u = goal){
      return true
    }
    else{
      for (v ∈ adj[u]){
        if (¬discovered[v]){
          queue.enqueue(v)
          discovered[v] = true
        }
      }
    }
  }
  return false
}
```

Bsp.:
ABB71

5.3.2 Tiefensuche

Die Tiefensuche lässt sich implementieren...

1.) wie die Breitensuche, aber mit einem Stack anstelle einer Warteschlange

2.) rekursiv.

Eine Warteschlange ist eine FIFO (first in, first out) Datenstruktur, die sich implementieren lässt mit einer verketteten Liste, die einen Zeiger auf das letzte Element besitzt.
ABB 72
Ein Stack (auch „Keller“) ist eine LIFO (last in, first out) Datenstruktur, die sich durch eine verkettete Liste implementieren lässt.
ABB 73

```
boolean tfs (node start, node goal){
  for (v ∈ V){
    discovered[v] = false
  }
  stack.push(start)
  discovered[start] = true
  while (¬stack.isEmpty){
    u = stack.pop()
    if (u = goal){
      return true
    }
    else{
      for (v ∈ adj[u]){
        if (¬discovered[v]){
          stack.push(v)
          discovered[v] = true
        }
      }
    }
  }
  return false
}
```

Bsp.:
ABB 74

Problem bei Breiten und Tiefensuche: Man braucht das Feld „discovered“. Das kann bei großer Anzahl von Knoten ein Problem sein $\rightarrow$ Speicheraufwändig

5.3.3 Topologisches Sortieren

ABB 75

Def.: Ein gerichteter Graph ist ein Paar (V, E) mit $V \neq 0$ und $E \subseteq V \times V$. Die Begriffe Weg, Pfad, Kreis lassen sich entsprechend definieren. ABB 76

Def.: Sei $G = (V, E)$ ein gerichteter Graph. Eine *topologische Sortierung* von G ist eine Abbildung $t : V \to \mathbb{N}$ mit $(u, v) \in E \Rightarrow t(u) < t(v)$
ABB 77
Für einen Kreis oder einen Graph mit einer Schlinge existieren keine topologische Sortierungen.

Eine topologische Sortierung kann durch eine Tiefensuche bestimmt werden.
Alogrithmus TopSort:

```
for v ∈ V
  markiere v mit weiß
for v ∈ V
  tiefensuche(v)

tiefensuche(v){
  v grau: Fehler, Kreis vorhanden
  v weiß: markieren v mit grau
    for( u | (v,u) ∈ E )
      tiefensuche (v)
    markiere v mit schwarz und füge v an den Kopf einer Liste an
}
```

ABB 91
Laufzeit dieser topologischen Suche: $O(|V| + |E|)$
Sowohl die Tiefensuche als auch die Breitesuche besitzen eine Laufzeit in $O(|V| + |E|)$.

5.3.4 Suche

Lineare Suche Laufzeit: $O(n)$

Binäre Suche Voraussetzung: Sortiertes Array
ABB 92

Vorgehen: Es wird die Mitte des Arrays bestimmt (Länge/2 [abgerundet]) und der gesuchten Wert mit dem Wert an dieser Stelle verglichen. Dabei gibt es drei Möglichkeiten:

- Gleichheit: Wert gefunden.
- Gesuchter Wert kleiner als Wert an der Stelle im Array: Auf gleiche Weise weitersuchen in der linken Hälfte (ausschließlich des bereits betrachteten Elements).
- Gesuchter Wert größer: Auf gleiche Weise weitersuchen in der rechten Hälfte

Algorithmus wird beendet, wenn der Wert gefunden wurde oder die zu durchsuchende Arrayhälfte keine Elemente mehr enthält.

Laufzeit:
ABB 93
Zur Analyse der Laufzeit ändern wir den Algorithmus so, dass nur Vergleiche $\leq$ und $>$ vorgenommen werden. Ferner sei die Länge des Arrays eine Zweierpotenz und das gesuchte Element nicht vorhanden (worst-case).
In diesem Fall lässt sich das Verhalten des Algorithmus als vollständiger binärer Wurzelbaum darstellen.
ABB 94

Wenn $n = 2^k$ die Länge des Arrays ist, dann besitzt dieser Wurzelbaum genau 2^k Blätter (die Vergleichen in einen 1-elementigen Array entsprechen). Dieser Binärbaum besitzt daher die Tiefe $k = log_2(n)$. Die Laufzeit der binären Suche liegt daher in $O(log\ n)$ (gilt auch, wenn n keine Zweierpotenz ist).
Um auch dynamische Datenstrukturen effizient durchsuchen zu können, lassen sich binäre Suchbäume nutzen.

Suchbaum Ein *Suchbaum* ist ein binärer Wurzelbaum, in dem jeder linke Teilbaum eines Knotens kleinere Wert und jeder rechte Teilbaum größere Wert als der Vorgängerknoten besitzt.
Beispiel:
ABB 95
Ein Suchbaum lässt sich ähnlich der binären Suche rekursiv durchsuchen.
ABB 96
Die Laufzeit der Suche ist $O(log(n))$, wenn der Baum vollständig balanciert ist und $O(n)$, wenn er linear entartet ist.

Hashing Prinzip: Mit Hilfe einer Hashfunktion h werden Schlüssel auf eine Position in einem Array (Hashtabelle) abgebildet.
Beispiel für eine Hashfunktion:

```
h(s) = s mod m
```

wobei m die Größe der Hashtabelle ist.
Problem: Es können Kollisionen auftreten, d.h. Schlüssel s_1, s_2 mit $h(s_1) = h(s_2)$.
Lösung: Überlauflisten:
An Position $h(s)$ wird eine Liste aller Elemente gespeichert, die diesen Hashwert besitzen.
Unter geeigneten Voraussetzungen besitzt Hashing eine Laufzeit von $O(1)$.

Literatur

[1] Boris Hollas. *Grundkurs Theoretische Informatik mit Aufgaben und Prüfungsfragen*. Spektrum Akademischer Verlag, 2007.