Grundlagen der Informatik

Raphael Pour

22. Januar 2016

Inhaltsverzeichnis

1 Aussagelogik

- **atomar** (lassen sich innerhalb der Aussagelogik nicht weiter zerlegen)
- entweder wahr oder falsch
- formuliert eine Aussage (=Formel)

Bsp.: 'Die Straße ist nass'

1.1 Syntax & Semantik

Syntax $\widehat{=}$ Form/ Gestalt
Semantik $\widehat{=}$ Bedeutung einer syntaktisch korrekten Formel.

1.1.1 Definition - Formeln der Aussagelogik

Die **Formeln der Aussagelogik** sind induktiv ($\wedge=$ vollständige Induktion / schrittweise) definiert:

- Jede atomare Aussage ist eine Formel der Aussagelogik. Diese heißen **Atomformeln** oder Variablen. Atomformeln bezeichnen wir mit Kleinbuchstaben oder durch Wörter in Kleinbuchstaben (**lower-case**)
- Wenn F, G Formeln der Aussagelogik sind dann sind $\underbrace{(F \wedge G)}_{\text{UND}}, \underbrace{(F \vee G)}_{\text{ODER}}, \underbrace{(\neg F)}_{\text{NICHT}}$.

1.1.2 Beispiel - Formale Aussage

Formeln der Aussagelogik sind:

- x
- y
- $x \wedge y$
- $x \vee y$
- $(x \wedge (y \wedge z)) \vee ((\neg x) \wedge (y \wedge (\neg z)))$
- regnet
- nass
- regnet $\wedge$ nass

Keine Formeln der Aussagelogik sind:

- x$\wedge$
- $\vee$x
- x $\wedge\vee$ y

1.1.3 Prioritäten der Operatoren

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

1.1.4 Definition - Belegung/ Wahrheitswert

Eine Belegung einer Formel F der Aussagelogik ist eine Zuordnung von Wahrheitswerten 'wahr' (1) oder 'falsch' (0) zu den Atomformeln von F. Daraus ergibt sich der Wahrheitswert einer Formel.

- Eine Atomformel ist $\underbrace{\text{genau dann}}_{\text{dann und nur dann}}$ wahr, wenn sie mit wahr belegt ist.
- $F \wedge G$ ist genau dann wahr, wenn F wahr ist und G wahr ist.
- $F \vee G$ ist wahr, wenn F wahr ist oder G wahr ist (Inklusives ODER).
- $\neg F$ ist wahr, wenn F falsch ist.

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

Wenn $\underbrace{\text{regnet}}_{\text{Atomformel}}$ 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 ist die Straße nass: $\neg$ regnet $\vee$ nass.

1.1.5 Definition - Implikation/ Äquivalenz

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

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

Vgl. Beispiel Aufgabenblatt 1: regnet $\rightarrow$ nass

1.1.6 Beispiel - Implikation

Berechnen des Betrags y einer Zahl x.

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

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

1.1.7 Definition einer Formel F der Aussagelogik

Eine Formel F der Aussagelogik heißt

- erfüllbar, wenn es eine Belegung gibt, so dass F wahr ist sonst unerfüllbar. Mit $\perp$ bezeichnen wir eine unerfüllbare Formel bzw. Widerspruch.
- Tautologie oder gültig, wenn F für jede Belegung wahr ist. Mit $\top$ bezeichnen wir eine Tautologie.

1.1.8 Beispiel - Formel der Aussagelogik

- $x \wedge y$ ist erfüllbar (aber keine Tautologie)
- $((\neg \wedge y) \vee (x \wedge \neg y)) \wedge \neg(x \vee y)$ ist unerfüllbar
- $x \vee \neg x$ ist eine Tautologie

1.1.9 Definition - Äquivalenz

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)

$$F \rightarrow G \equiv \neg F \vee G \tag{1}$$

$$x \geq 0 \rightarrow x + 1 < 0 \tag{2}$$

$\Rightarrow entsprichtimmerWahr(Tautologie)$

1.2 Beweistechniken

Erwiesene Wahrheit. Beweis ist eine Folgerung aus bereits bekannten Aussagen.
Definitionen führen nur neue Begriffe ein (kein Wahrheitsgehalt). Keine Aussagen um Aussagen besser Formulieren zu können.
Vermutung (keine wahre Aussage) $\Rightarrow p$ ist unbekannt.
Beweis durch Beispiel ist nicht gestattet.
Zeigen $\Rightarrow$ Beweis führen.
Widerspruch $\Rightarrow$ Gegenbeweis ("Für alle....gilt" durch Gegenbeweis widerlegen)

1.2.1 Direkter Beweis

Beispiel Wenn $a \in Z$ gerade ist, dann ist a^2 auch gerade.

$$a \in Z\text{gerade} \Rightarrow a^2\text{gerade} \tag{3}$$

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

1.3 Indirekter Beweis

Beweis wird $A \Rightarrow B$ wird bewiesen, indem die äquivalente Aussage $\neg B \Rightarrow \neg A$ beweisen wird.

$$A \Rightarrow B \neg B \Rightarrow \neg A \tag{4}$$

$$\text{regnet} \Rightarrow \text{nass} \equiv \neg\text{nass} \Rightarrow \neg\text{regnet} \tag{5}$$

Beweis: Alle Wahrheitswerte p betrachten via Rechenregeln/Wahrheitstabelle.

Beispiel Wenn $a \in Z$ gerade ist, dann ist a^2 auch gerade.

$$a \in Z\text{gerade} \Rightarrow a^2\text{gerade} \tag{6}$$

Beweis Wir zeigen: Wenn a ungerade ist, dann auch a^2
Aus a ungerade folgt $a = 2n - 1$ für ein n. Dann ist $a^2 = 4n^2 - 4n + 1 = 4(n^2 - n) + 1$ ungerade. $\Rightarrow$ *Kontraposition*
Mit einem *Beweis durch Widerspruch* wird eine Aussage abgewiesen, indem gezeigt wird, dass die Annahme "A ist falsch" zu einem Widerspruch geführt wird. (d.h. es wird $\neg A \rightarrow \perp$ gezeigt. Dann ist $A \rightarrow \top$

1.3.1 Vollständige Infuktion

Bei einer vollständigen Induktion lassen sich Aussagen der Art "für $n \in N$ gilt..." beweisen.

Prinzip Gegeben sei eine Aussage der Form "fFür alle $m \in N^*$ gilt $A(n)$.

- Induktionsanfang: Man zeigt die Wahrheit für die Aussage $n = 1$
- Induktionsschritt: Man zeigt: Wenn die Aussage für n wahr ist (*Induktionsvoraussetzung*), dann ist sie auch für $n + 1$ wahr.

In Formeln: Man zeigt $A(1)$ und für alle $n : A(n) \Rightarrow A(n + 1)$[1]
Wichtig: $A(1)$ ist keine Induktionsvoraussetzung.

Beispiel Für alle $n \geq 1$ gilt $\sum_{k=1}^{n} \frac{n(n+1)}{2}$ (Gauss)
Beweis(Induktion):

- $n = 1 : 1 = 1 * \frac{2}{2}$ ist wahr
- $n \rightarrow n + 1$: Es gelte $\underbrace{\sum_{k=1}^{n} \frac{n(n+1)}{2}}_{Induktionsvoraussetzung}$

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

$$\sum_{k=1}^{n+1} k = (\sum_{k=1}^{n} k) + n + 1 = \frac{n(n+1)}{2} + n + 1 \tag{7}$$

Man muss innerhalb eines Induktionsschritt die Induktionsvoraussetzung anwenden.

2 Elementare Kombinatorik

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

[1] Vgl. Dominoeffekt: Kippt ein Stein dann fallen alle wegen der Kettenreaktion nacheinander um

2.0.2 Beispiel - Potenzmenge

$\mathcal{P}(\{1,2\} = \{\emptyset, \{1\}, \{2\}, \{1,2\}, \{1,1\}, \{2,2\}\}$

2.0.3 Definition Mächtigkeit

Die Mächtigkeit einer Menge A ist die Anzahl ihrer Elemente. Notation $|A|$ (nicht Betrag).

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.

2.0.4 Beispiel - PIN

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

2.0.5 Beispiel - Schleife

In dem Programm

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

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

2.0.6 Satz

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

Beweis: Für $M = \{m_1, ..., m_n\}$ identifizieren wir eine Teilmenge $A \subseteq M$ durch das n-Tupel $(a_1, ..., a_2)$ mit

$$a_k = \begin{cases} 0 & \text{für } m_k \notin A \\ 1 & \text{für } m_k \in A \end{cases}$$

Nach obigem Satz gibt es $|\{0,1\}^m| = 2^m = 2^{|M|}$ derartige Tupel.

2.0.7 Beispiel

$M = \{1,2,3\}\{2,3\} : \{0,0,1\}, \emptyset : \{0,0,0\}$

2.0.8 Definition Binominalkoeffizient

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

2.0.9 Beispiel - Binominalkoeffizient

- $\binom{n}{0} = 1$, da $\emptyset$ die einzige 0-elementige Teilmenge ist.

- $\binom{n}{n} = 1$, weil es nur eine n-elementige Menge in n gibt, n 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 Elemente $\{a, b\}\{b, a\}$ hierbei doppelt gezählt werden, müssen wir durch 2 teilen.

2.0.10 Definition Permutation

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

2.0.11 Beispiel - Permutation

Alle Permutationen von $1, 2, 3$ sind

- $1, 2, 3$
- $1, 3, 2$
- $2, 1, 3$
- $2, 3, 1$
- $3, 1, 2$
- $3, 2, 1$

2.0.12 Definition n!

$$n! = 1 * ... * n.\ 0! = 1.$$

2.0.13 Satz - n!

Es gibt n! Permutationen einer n-elementigen Menge.
Beweis: Für die 1. Stelle gibt es n Möglichkeiten, für die 2. Stelle n-1 usw. Für die letzte Stelle n-(n-1). Insgesamt als $n * ... * 1 = n!$ Möglichkeiten.

2.0.14 Satz - Binominalkoeffizient und Fakultät

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$
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 letzte Element n-k+1 Möglichkeiten, insgesamt daher $n * ... * (n - k + 1)$ Möglichkeiten. Da Reihenfolge, in der diese k-Elemente ausgewählt werden keine Rolle spielt, muss dieses Produkt durch k! geteilt werden. Daher erhalten wir:

$$\binom{n}{k} = \frac{n*(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

3 *O*-Notation

Mit Hilfe der *O*-Notation lassen sich obere Schranken für die Laufzeit eines Algorithmus[2] angeben. Um die Laufzeit eines Algorithmus zu messen bestimmen wir die Anzahl Schritte und geben mit Hilfe der *O*-Notation die Größenordnung in Abhängigkeit der Länge der Eingabedaten an.

[2] Algorithmus: Verfahren um ein Problem in endlich viele Schritten zu lösen bzw. Beschreibung eines Verfahrens

3.0.15 Beispiel - Lineare Suche

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
}
```

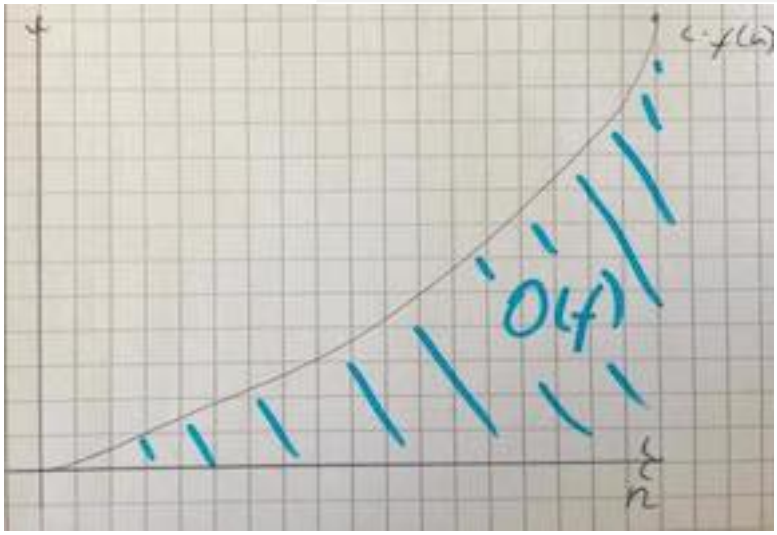
Laufzeit dieser Funktion:
$\leq \underbrace{c_1 + n * c_2 + c_3}_{g(n)} \leq (c_1 + c_2 + c_3) * n = \underbrace{c * n}_{f(n)}$

- n.....max Schleifendurchläufe
- c_1...Zeitaufwand i zu allozieren
- c_2...Adresse in Vektor ausrechnen + Vergleich

3.0.16 Definition $O(f)$

Für eine Funktion $f > 0$ ist $O(f)$ die Menge aller Funktionen g, für die gilt:

$g(n) \leq c * f(n)$ für $c > 0$ für alle großen n



Die Laufzeit (LZ) der linearen Suche liegt in $O(n)$.

3.0.17 Beispiel - O Notation

$2n^3 - n + 5 \leq 2n^3 + 5 \leq 7n^3 \in O(n^3)$

3.0.18 Beispiel - Array Elemente vergleichen

```
// jedes geordnete Paar wird genau einmal verglichen
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 LZ ist daher

$\leq \underbrace{c_1 * \binom{n}{2}}_{Schleifen} + \underbrace{c_2}_{return} \leq (c_1 + c_2)\binom{n}{2} = \frac{c_1+c_2}{2} * n(n-1) \leq \frac{c_1+c_2}{2} * n^2 \in O(n^2)$[3]

Übung: $2 * log(n^2 + 1)$

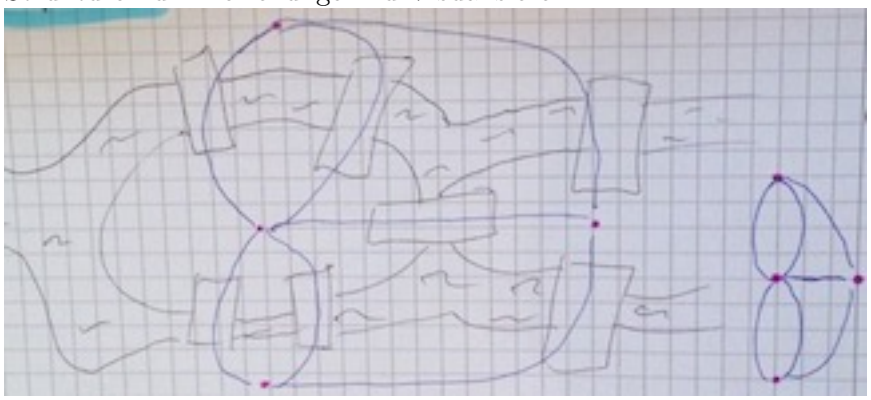
$n^2 + 1 \leq n^3 \Leftrightarrow \underbrace{\frac{1}{n}}_{0} + \underbrace{\frac{1}{n^3}}_{0} \leq 1$ für $n \to \infty$

$2log(n^2 + 1) \leq 2logn^3 = 6logn \in O(logn)$

4 Graphen

4.1 Ungerichtete Graphen

Strukturen um Beziehungen zu Visualisieren.

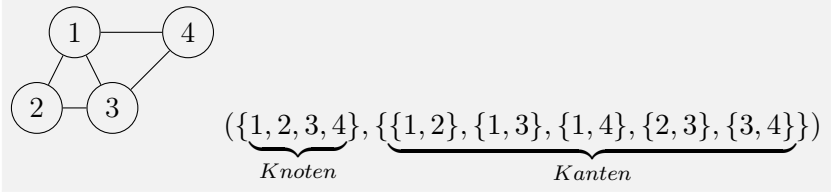


4.1.1 Definition - Ungerichteter Graph

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

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

4.1.2 Beispiel - Graph mit Notation



4.1.3 Definition - Vollständiger Graph

Ein Graph heißt vollständig, wenn alle Knoten paarweise verbunden sind.
Ein vollständiger Graph mit n Knoten besitzt genau $\binom{n}{2}$ Kanten.

[3] n^2 deswegen, weil beide Schleifen ungefähr n-mal durchlaufen

4.1.4 Definition - Grad K und $deg(v)$

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

4.1.5 Satz - Kantenanzahl

Für jeden Graphen gilt: $\sum_{v \in V} deg(v) = 2|E|$ 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 $2|E|$

4.1.6 Definition - Weg

Ein Weg ist eine Folge von Knoten $v_1, ..., v_k$ mit $\{v_l, v_{l+1}\} \in E$ für $l = 1, ..., k-1$

Ein Weg heißt Kreis, wenn $v_1 = v_k$

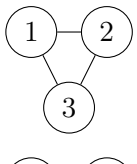
4.1.7 Definition - Zusammenhängender Graph

Ein Graph heißt **zusammenhängend**, wenn es für alle Paare von Knoten u, v einen Weg von u nach v gibt.

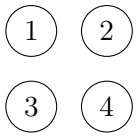
4.1.8 Definition - Pfad

Ein **Pfad** ist ein Weg, der keinen Knoten mehrfach enthält.

4.1.9 Beispiel - Graphen



$(\{1,2,3,4,5\}, \{\{1,2\},\{1,3\},\{2,3\},\{4,5\}\})$ Ist ein Graph



$(\{1,2,3,4\}, \{\})$ Ist ein Graph

4.1.10 Beispiel - Zusammenhängender Graph

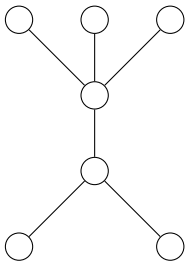
Ein Telefonnetz muss z.B. ein zusammenhängender Graph sein, damit jeder Teilnehmer jeden anderen Teilnehmer erreichen kann.

4.2 Bäume

4.2.1 Definition - Baum

Ein **Baum** ist ein zusammenhängender Graph, der keine Kreise enthält. Ein **Blatt** ist ein Knoten v mit $deg(v) \leq 1$

4.2.2 Beispiel - Baum



4.2.3 Satz - Kanten in einem Baum

Sei $B = (V, E)$ ein Baum. Dann gilt $|E| = |V| - 1$ (E=Kanten, V=Knoten).
Beweis (Induktion):
$|V| = 1$: Ein Baum mit nur einem Knoten enthält keine Kanten.
$|V| \rightarrow |V| + 1$: Sei B ein baum mit $|V| + 1$ Knoten. B besitzt ein Blatt (siehe Übung). Indem wir dieses Blatt zusammen mit der zugehörigen Kante entfernen erhalten wir einen Baum B' mit $|V|$ Knoten und nach Induktionsvoraussetzung $|V| - 1$ Knoten. Damit besitzt $B(|V| + 1) - 1$ Kanten. Danach ist jeder Baum minimal zusammenhängend (Entfernt man egal welche Kante ist der Baum nicht mehr zusammenhängend (und auch kein Baum mehr...)).

4.2.4 Definition - Wurzelbaum

Ein **Wurzelbaum** ist ein Baum mit einem als Wurzel ausgezeichneten Knoten.

4.2.5 Definition - Binärer Wurzelbaum

Ein **binärer Wurzelbaum** ist ein Wurzelbaum in dem jeder Knoten der kein Blatt ist genau zwei Nachfolger besitzt.
Gleichwertige Definition (induktiv):

- Ein einzelner Knoten ist ein binärer Wurzelbaum
- Wenn w_1, w_2 binäre Wurzelbäume sind, dann erhalten wir einen neuen Wurzelbaum in dem die Wurzeln von w_1, w_2 mit einer neuen Wurzel verbunden werden.

4.2.6 Satz - Tiefe

Ein binärer Wurzelbaum mit **Tiefe** d (d.h. alle Paare von Wurzel zu einem Blatt haben die Länge d) besitzt genau 2^d Blätter.
Beweis (Induktion):
$d = 0$: Ein binärer Wurzelbaum der nur aus der Wurzel besteht, enthält 2^d Blätter.
$d \rightarrow d+1$: Ein binärer Wurzelbaum der Tiefe $d+1$ enthält zwei Wurzelbäume der Tiefe d. Diese enthalten nach Induktionsvoraussetzung jeweils 2^d. Folglich besitzt der binäre Wurzelbaum der Tiefe $d + 1$ genau $2 * 2 = 2^{d+1}$ Blätter.

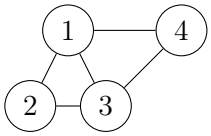
4.3 Datenstrukturen zur Repräsentation

Es gibt zwei Möglichkeiten um Graphen darzustellen:

4.3.1 Adjazenmatrix

Für einen Graphen $G = (V, E)$ ist die Adjazenmatrix eine $|V| \times |V|$ =Matrix(a_{uv})

mit $a_{uv} = \begin{cases} 1 & \text{für } \{U, V\} \in E \\ 0 & \text{sonst} \end{cases}$



$$\Bigg\} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

4.3.2 Adjazenliste

Die Adjazenliste ist ein Array, das an jeder Position v eine Liste der mit v verbundenen Knoten enthält.
Bäume, insbesondere Binärbäume lassen sich noch einfacher darstellen. Jeder Knoten wird dargestellt durch eine Datenstruktur die einen Verweis auf die Nachfolger enthält.

4.4 Grundlegende Graphenalgorithmen

4.4.1 Breitensuche

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 oder alle Knoten durchsucht wurden.

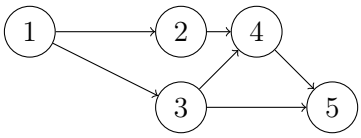
4.4.2 Tiefensuche

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 ein Zeiger auf das letzte Element besitzen.
Ein Stack ist ein LIFO (Last in/First out) Datenstruktur, die sich durch eine versteckte Liste implementieren lässt.

4.5 Topologisches Sortieren (Topsort

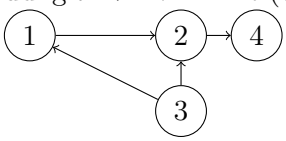


4.5.1 Definition - Gerichteter Graph

Ein gerichteter Graph ist ein Paar (V, E) mit $V + \emptyset$ und $E \subset V \times V$. Die Begriffe Weg, Pfad, Kreis lassen sich entsprechend definieren.

4.5.2 Definition - Topologische Sortierung

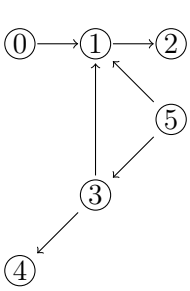
Sei $G = (V, E)$ ein gerichteter Graph. Eine **topologische Sortierung** von G ist eine Abbildung $t : V \to N$ mit $(u, v) \in E \Rightarrow t(u) < t(v)$



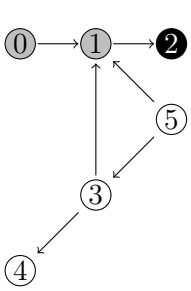
Eine topologische Sortierung kann durch eine Tiefensuche bestimmt werden.

4.5.3 Algorithmus - TopSort

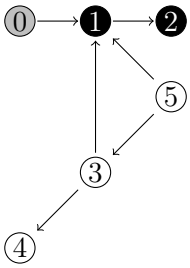
```
for v ∈ V
    markiere v mit weiß
for v ∈ V
    tiefensuche(v) tiefensuche(v):
    v == grau: Fehler: Kreis vorhanden
    v == weiß: markiere v mit grau
      for {u|(v, u) ∈ E}
        tiefensuche(u)
        markiere v mit schwarz und füge v an den Kopf der Liste an
    (v==schwarz: return)
```



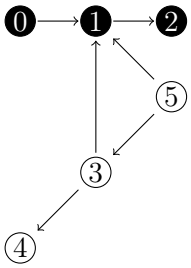
Stack



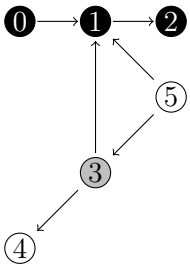
Stack
2



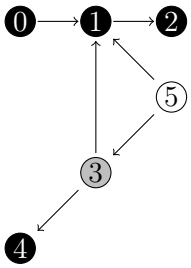
Stack
1
2



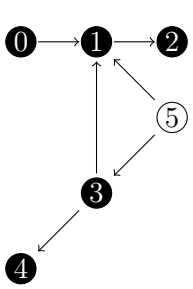
Stack
0
1
2



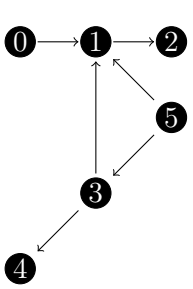
Stack
0
1
2



Stack
4
0
1
2



Stack
3
4
0
1
2



Stack	Index
5	0
3	1
4	2
0	3
1	4
2	5

Gleiche Laufzeit wie Tiefensuche/Breitensuche: $O(|V| + |E|)$
Zuerst werden alle Knoten mit weiß markiert $\wedge$=$O(|V|)$
Für jeden Knoten Tiefensuche gestartet. Algorithmus betrachtet Nachbarn verbraucht Zeit $\sum_{v \in V} deg(v) = 2|E|$.
Sowohl Tiefensuche als auch die Breitensuche besitzen die Laufzeit $O(|V| + |E|)$

4.6 Suche

4.6.1 Lineare Suche

Die Lineare Suche hat eine Laufzeit von $O(n)$ und wird gerne als naive Suche bezeichnet.

4.6.2 Binäre Suche:

Voraussetzung Sortiertes Array

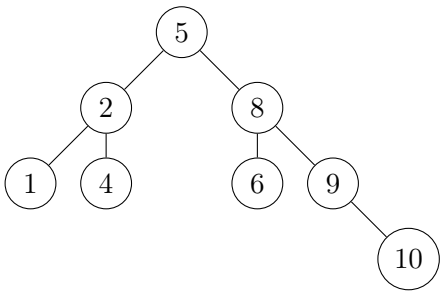
Vorgehen Im 1. Schritt wird die Mitte des Arrays bestimmt (Länge/2, abrunden) und der gesuchte Wert mit dem Wert an dieser Stelle verglichen. Dabei gibt es 3 Möglichkeiten:

1. Gleichheit: Wert gefunden
2. Gesuchter Wert ist kleiner: Auf gleiche Weise weitersuchen in der linken Hälfte.
3. Gesuchter Wert ist größer: Auf gleiche Weise weitersuchen auf der rechten Hälfte.

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

Analyse Zur Analyse der LZ (Laufzeit) ändern wir den Algorithmus so, dass nur vergleiche $\leq$ bzw. $>$ 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.
Wenn $n = 2^k$ die Länge des Arrays ist dann besitzt dieser Wurzelbaum genau 2^k Blätter (die Verleichen in einem 1-elementigen Array entsprechen). Dieser Binärbaum besitzt daher die Tiefe $k = log_2 n$. Die Lautzeit der binären Suche liegt daher in $O(logn)$ (gilt auch wenn n keine Zweierpotenz ist)
Um auch dynamische Datenstrukturen effizient durchsuchen zu können, lassen sich binäre Suchbäume nutzen.
Ein *Suchbaum*[4] ist eine binärer Wurzelbaum in dem jeder linke Teilbaum eines Knotens kleinere Werte und jeder rechte Teilbaum einen größeren Wert als der Vorgänger besitzt.

[4]Mit Hilfe von Structs mit 2 Pointern nur die jeweiligen Nachfolgern bzw. Nullpointern



Bsp.:
Ein Suchbaum lässt sich ähnlich der binären Suche rekursiv durchsuchen.
Die LZ der Suche ist $O(\log n)$, wenn der Baum vollständig balanciert ist und $O(n)$, wenn er linear entartet ist.

4.7 Hashing

Prinzip Mit Hilfe einer Hashfunktion werden Schlüsssel auf eine Position in einem Array (Hashtabelle) abgebildet.

4.7.1 Beispiel für eine Hashfunktion

$$h(s) = s \mathbf{mod} m \tag{8}$$

Wobei m die Größe der Hastabelle ist.

Problem Es können Kolissionen auftreten d.h. Schlüssel s_1, s_2 mit $h(s_1) = h(s_2)$

Lösung - Überlauflisten An die Position $h(s)$ wird eine Liste aller Elemente gespeichert, die diesen Hashwert besitzen.
Unter geeigneten Voraussetzungen besitzt Hashing eine LZ von $O(1)$

4.8 Sortierverfahren

4.8.1 Quicksort

Quicksort partitioniert das zu sortierende Feld anhand eines Privotelements und sortiert rekursiv die dadurch entstandenen Teilfelder.
...
...
...

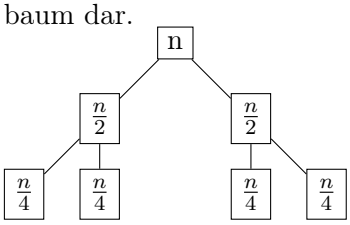
```
object qsort
        def Qs(l:list[int]):List[int]= dmatch {
                case Nil => Nil
                case h::t =>
                        val(li, re) = t.partition(x=>x<=h)
                        qs(li)::h::qs(re)
        }
def main(args Array[string]){
        val d = (for(i<-0 until 100000)yield l).toList
        val x = qs(l)
}
```

Die Laufzeit ist schwierig, weil die Länge der Teillisten vom Pivotelement abhängt. Wir betrachten stattdessen ein ähnliches Verfahren: Mergesort.

4.8.2 Mergesort

Dabei werden die Listen halbiert, rekursiv sortierung und dann zusammengehängt.

Laufzeitanalyse Wir stellen das Verhalden von Mergesort für $n = 2^k$ durch einen Binärbaum dar.



Zur Erzeugung der Hälften und dem Zusammenfügen fällt der Aufwand $O(|linkeListe| + |rechteListe|$. Dies ist $O(n)$ auf jeder Ebene des Baums. Der Baum hat $n = 2^k$ Blätter und hat daher die Tiefe k. Die LZ von Mergesort ist daher $O(n \log n)$

5 Codierungstheorie

5.0.3 Beispiel - Resteklassen (1)

Wichtigste Gruppe $(Z_2, +)$ (Resteklasse (0,1))
$a + b = a + bmod2, a + a^{-1} = 0, 1 + 1 \equiv 0(\text{mod}2)$

5.0.4 Beispiel - Resteklassen (2)

$(Z_p, +)$ (Besteht aus 0 bis $p - 1$)
$7 + 4 \equiv 0(\text{mod}11) \Rightarrow 4$ ist die Inverse von 7 in Z_{11}

Problem: Daten können bei der Übertragung verändert werden. Wie können diese Fehler erkannt und ggf. korrigiert werden?

Lösung: Häufige Lösung zur Fehlerbehebung: *Prüfsummen*

5.1 Paritätsprüfung

Für eine Folge von Bits $b_1, ..., b_{n-1} \in \{0, 1\}^5$ wird das *Paritätsbit*

$$b_n = \left(\sum_{i=1}^{n-1} b_i \right) \text{mod } 2 \tag{9}$$

berechnet. Das enthaltene Codewort ist $b_1...b_n$[6].

5.1.1 Beispiel - Parity-Check-Code berechnen

Für die Folge von Bits $0, 1, 1, 0, 1, 0$ ist 1 das *Paritätsbit*. Das Codewort ist 0110101[7].
Aus

$$\left(\sum_{i=1}^{n-1} b_i \right) \equiv b_n(\text{mod } 2) \tag{10}$$

[5] Bits $\wedge =$ Elemente aus Z_2
[6] Keine Folge sondern ein zusammengesetztes Wort!
[7] Die letzte 1 ist die Prüfsumme

folgt daher

$$\left(\sum_{i=1}^{n} b_i\right) \equiv 0(\text{mod } 2) \tag{11}$$

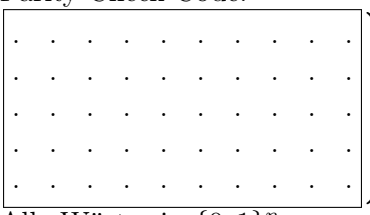
5.1.2 Definition Parity-Check-Code

Die Menge der Wörter des *Parity-Check-Codes* lässt sich darstellen durch

$$\{(b_1, ..., b_n) \in \{0,1\}^n | \left(\sum_{i=1}^{n} b_i\right) \equiv 0(\text{mod } 2)\} \tag{12}$$

5.1.3 Beispiel - Parity-Check-Code

Für $n = 8$ ist 00110011 ein Element des Parity-Check-Code, 01110011 kein Element des Parity-Check-Code.



5.1.4 Satz - Fehlererkennung

Der Parity-Check-Code ist 1-fehlererkennend.

5.1.5 Beweis - Fehlererkennung

Seien $b_1...b_n$ ein Codewort und $b_1^{'}...b_n^{'}$ ein Wort, das sich an Stelle k von $b_1...b_n$ unterscheidet. Angenommen $b_1^{'}...b_n^{'}$ ist ein Codewort. Dann gilt:

$$0 \equiv \left(\sum_{i=1}^{n} b_i^{'}\right) \equiv \left(\sum_{i=1}^{n} b_i + (b_k^{'} - b_k)\right)^{8} \tag{13}$$

$0 + -1 \equiv 1$, Widerspruch!

Bemerkung: Der Parity-Check-Code ist nicht 2-fehler-korrigierend.

Anmerkungen:

- Speichercontroller (RAM-Bausteine), Festplattencontroller
- Netzwerkprotokolle
- 7-bit ASCII-Code, Bit 8 als Paritätsbit
- ...

Ferner kann das Verfahren zur Rekonstruktion eines verloren gegangenen Bits verwendet werden, wenn die restlichen Bits fehlerfrei sind.

[8] Die beiden Summen unterscheiden sich an der k-ten Stelle

5.1.6 Beispiel - Verloren gegangenes Bit

$0100x110 \Rightarrow 1+1+1+1 = 0 \pmod 2 \Rightarrow x = 1$
Denn aus

$$0 \equiv \sum_{i=1}^{n} b_i \pmod 2 \tag{14}$$

folgt für $1 \leq k \leq n$

$$b_k \equiv \sum_{i=1} b_i \pmod 2 \tag{15}$$

Damit kann b_k aus den anderen Bits rekonstruiert werden.

Anmerkungen: RAID4, RAID5 Daten und Paritäten werden auf m Festplatten verteilt. Beim Ausfallen einer Platte können die Daten rechnerisch rekonstruiert werden.

5.2 ISBN-Code

Der ISBN-Code enthält eine Prüfsumme.

5.2.1 Beispiel - ISBN-Code

$\underbrace{382741826}_{\text{9-Stellige Buchnr.}} \overbrace{7}^{\text{Prüfziffer}(ISBN10)}$

Für die Prüfziffer gilt:

$$Z_{10} \left(\sum_{i=1}^{9} i * Z_i) \pmod{11} \right. \tag{16}$$

Dabei wird X für den Wert 10 verwendet. Wegen $10 + 1 \equiv 0 \pmod{11}$ ist 10 das inverse Element 1 bezüglich + (d.h. 10 entspricht -1).
Aus obiger Gleichung folgt damit:

$$0 \equiv \sum_{i=1}^{10} i * z_i \pmod{11} \tag{17}$$

Die Menge der Codewörter ist daher

$$\{(Z_1, \ldots, Z_{10}) | Z_1, \ldots, Z_9 \in \{0, \ldots, 9\}, Z_{10} \in \{0, \ldots, X\}, \sum_{i=1}^{10} i * z_i \equiv 0 \pmod{11}\} \tag{18}$$

$(Z, +, *)$ ist ein Körper.

5.2.2 Beispiel Zahlenkörper

$$1 + 10 \equiv 0$$
$$2 * 6 \equiv 1 (6 \wedge = 2^{-1})$$
$$6 * 2 \equiv 1 (2 \wedge = 6^{-1})$$
$$6 \equiv x \Leftrightarrow 1 \equiv 2x$$

5.2.3 Satz - ISBN Fehlererkennung

Der ISBN-Code ist 1-fehlererkennend.

5.2.4 Beweis - ISBN 1-fehlererkennend

Seien $z_1...z_{10}$ ein Codewort und $z_1^{'}...z^{'}10$ ein Wort, dass sich an der Stelle k von $z_1...z_{10}$ unterscheidet.
Angenommen, $z_1^{'}...z_{10}^{'}$ ist ein Codewort.
Dann gilt:

$$\sum_{i=1}^{10} i * z_i \equiv 0 \pmod{11} \tag{19}$$

Da sich $z_1^{'}...z_{10}^{'}$ an Stelle k von $z_1...z_{10}$ unterscheidet, folgt:

$$0 \equiv \sum_{i=1}^{10} i * z_i + k(z_k^{'} - z_k) \tag{20}$$

$$0 \equiv 0 + k(z_k^{'} - z_k) \tag{21}$$

Da $k \neq 0$, besitzt k ein bezüglich * (Multiplikation) inverses Element k^{-1}, womit folgt:

$$0 \equiv z_k^{'} - z_k \tag{22}$$

und damit

$$z_k \equiv z_k^{'} \tag{23}$$

Widerspruch! Wir haben definiert, dass $z_k \neq z_k^{'}$ ist.

5.2.5 Satz - ISBN Zahlendreher

Der ISBN-Code erkennt Vertauschungen von Ziffern (Zahlendreher).

5.2.6 Beweis - ISBN Zahlendreher

Seien $z_1, ..., z_{10}$ ein Codewort und $z_1^{'}, ..., z_{10}^{'}$ ein daraus erhaltenes Wort, in dem die Stellen $(k < l)$ vertauscht wurden.
Angenommen, $z_1^{'}, ..., z_{10}^{'}$ ist ein Codewort. Dann gilt:

$$0 \equiv \sum_{i=1}^{10} i * z_i \equiv 1 * z_i + ... + k * z_k^{'} + ... + l * z_l^{'} + 10 * z_{10}^{'} \tag{24}$$

$$0 \equiv 1 * z_1^{'} + ... + k * z_l + ... + l * z_l + ... + 10 * z_{10}^{'} \tag{25}$$

$$0 \equiv \sum_{10}^{i=1} i * z_i + k(z_l - z_k) + l(z_k - z_l) \tag{26}$$

$$0 \equiv 0 + k(z_l - z_k) + l(z_k - z_l) \equiv k(z_l - z_k) - l(z_l - z_k) \Leftrightarrow k(z_l - z_k) \equiv l(z_l - z_k) \tag{27}$$

Da für $z_l \neq z_k$ das Element $z_l - z_k = 0$ ist und daher $(z_l - z_k)^{-1}$ existiert folgt: $k \equiv l$. Widerspruch! Definiert ist $k < l$

5.3 Fehlerkorrigierende Codes

Für ein empfangenes Wort w suchen wir Codewort v, sodass der Abstand $d(v, w)$ minimal ist (nearest neighbour decoding)
....geile grafik...

5.3.1 Definition - Hamming-Abstand

Für Wörter $v, v^{'} \in \{0,1\}^n$ ist der *Hamming-Abstand* $d(v, v^{'})$ die Anzahl Stellen, in denen sich $v, v^{'}$ unterschieden. Der *Minimalabstand* eines Codes C mit $\{d(v, v^{'}|v, v^{'} \in C, v \neq v^{'}\}$

5.3.2 Beispiel - Hamming-Abstand

$d(0101, 1010) = 4$
$d(0101, 0110) = 2$
$d(0...011, 0...000) = 2$
Minimalabstand: 2

5.3.3 Satz - Fehlererkennung und Fehlerkorrektur

1. Ein Code ist k-fehlererkennend genau dann wenn sein Minimalabstand mindestens $k+1$ ist.

2. Ein Code ist k-fehlerkorrigierend gdw. sein Minimalabstand mindestens $2k+1$ ist.

5.3.4 Beispiel - Fehlererkennung und Fehlerkorrektur an Parity-Check-Code

Der Parity-Check-Code besitzt den Minimalabstand 2 und ist daher 1-fehlererkennend und 0-fehlerkorrigierend.

5.3.5 Beweis - Fehlererkennung/Fehlerkorrektur

1. Übung

2. (=>) Wenn der Code fehlerkorrigierend ist, darf es nur ein Codewort v mit $d(v, w) \leq k$ für ein w geben. Folglich muss $d(v^{'}, w) \geq k+1$ für alle Codewörter $v \neq v^{'}$ gelten, woraus $d(v, v^{'} \geq 2k+1$ folgt.
   (<=) Für ein Codewort sei $S(V) = \{w|d(v, w) \leq k\}$
   Zu zeigen: Aus $v, v^{'}$ Codewörter mit $v \neq v^{'}$ folgt $S(v) \cap S(v^{'}) = \emptyset$ (Disjunkt).
   Angenommen, es gibt ein $w \in (v) \cap S(v^{'})$. Dann gilt $d(v, w) \leq k$ und $d(w, v') \leq k$.
   Daraus folgt mit der Dreiecksungleichung $d(v, v^{'}) \leq d(v, w) + d(w, v^{'}) = 2k$.
   Dies ist ein Widerspruch, da der Minimalabstand des Codes mindestens $2k+1$ ist.
   Jedes $w \in S(v)$ lässt sich daher eindeutig zu v decodieren.

Naiver Ansatz zur Fehlerkorrektur: Die Nachricht mehrmals schicken.

5.3.6 Beispiel - Naive Fehlerkorrektur

$0 \to 000, 1 \to 111$ Der Code $\{000, 111\}$ hat den Minimalabstand 3 und ist daher 1-fehlererkennend.
Nachteil: Datenverschwendung. Effizienter sind Lineare Codes.

5.4 Lineare Codes

Die Decodierung durch eine *Nearest-Neighbour-Suche* im Coderaum ist ineffizient. Um ein effizienteres Verfahren zu erhalten beschreiben wir Codes durch Matrx-Vektor-Operationen.

5.4.1 Beispiel - Lineare Codes

Mit $A = \{\underbrace{1...1}_{n}\}$ lässt sich der Parity-Check-Code beschreiben durch $\{w \in \{0,1\}^n | Aw^T\}$

$$(1111)(0101)^T = (1111) * \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = 0$$

5.4.2 Definition - Linearer Code

Ein Codewort C heißt *linear*, wenn es eine Matrix A gibt, so dass

$$C = \{w | Aw^T = 0\} \tag{28}$$

Die Matrix A heißt *Parity-Check-Matrix*[9]

5.4.3 Satz - Lineare Codes und Vektorräume

Ein binärer linearer Code C ist ein Vektorraum über $Z_2(+, *)$

5.4.4 Beweis - Lineare Codes und Vektorräume

1. Abgeschlossenheit der Vektoraddition:
   zu zeigen: $w_1, w_2 \in C \Rightarrow w_1 + w_2 \in C$
   Aus $w_1, w_2 \in C$ folgt $Aw_1^T = \vec{0}, Aw_2^T = \vec{0}$[10]

2. Abgeschlossenheit der Multiplikation:
   zu zeigen: $\alpha \in Z_2, w \in C \Rightarrow \alpha * w \in C$
   Dies gilt, da
   $$\alpha * w \begin{cases} \vec{0} \text{für} \alpha = 0 \\ w \text{für} \alpha = 1 \end{cases} \tag{29}$$
   und $\vec{0}, w \in C$

Andere Vektoraxtiome folgen entsprechend.

Da ein linearer Code ein Vektorraum ist, besitzt er auch eine Basis.

5.4.5 Definition - Generatormatrix

Sei C ein linearer Code. Eine Matrix G, deren Zeilen eine Basis von C bilden, heißt *Generatormatrix.*

5.4.6 Beispiel - Generatormatrix

Sei $G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$, Da die Zeilen von G linear unabhängig sind, bilden sie eine Basis eines Vektorraums. Folglich ist G eine Generatormatrix.

[9] Diese Matrix hat nichts mit dem Parity-Check-Code zutun.

[10] Daraus folgt $w_1 + w_2 \in C \Rightarrow$ Durch Addition (bzw. Subtraktion) resultiert immer ein neues Codewort

Es gilt:

$$G = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}}_{\begin{pmatrix} b_1 & 0 & 0 & b_1 \\ 0 & b_2 & 0 & b_2 \\ 0 & 0 & b_3 & b_3 \end{pmatrix}} = (b_1, b_2, b_3, \sum_{i=1}^{3} b_i) \tag{30}$$

Der durch G erzeugte Code ist daher

$$C = \{(b_1, b_2, b_3, \sum_{i=1}^{3} b_i) | b_1, b_2, b_3 \in \{0, 1\}\} \tag{31}$$

Dies ist der Parity-Check-Code der Länge 4.
Wenn C ein linearer Code mit Parity-Check-Code-Matrix A und Generatormatrix G ist, dann können wir G zum codieren und A zum decodieren verwenden.

5.4.7 Beispiel - Betrachtung des Parity-Check-Code als linearer Code

Wir betrachten den Parity-Check-Code der Länge 4. Wir wollen die Nachricht (101) codieren.

$$(101) * \underbrace{\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}}_{G} = (1010) \tag{32}$$

zum Decodieren berechnen wir

$$\underbrace{(1111)}_{A} * \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = 0(\text{mod}2) \tag{33}$$

In diesem Fall wurde die Nachricht fehlerfrei übermittelt.

Vorteil dieses Ansatzes: Mit Hilfe einer Generatormatrix lassen sich beliebige weitere Codes definieren. Dies wird insbesondere zur Konstruktion von fehlerkorrigierenden Codes genutzt.

Eine auf dieser weise systematisch konstruierbare Klasse Codes sind die *Hamming-Codes*.

Die Generatormatrix eines *Hamming-Codes* besteht aus allen Vektoren in $\{0, 1\}^n$ außer der Nullvektor. Auf diese Weise lassen sich fehlerkorrigierende Codes erzeugen.

5.4.8 Beispiel - *Hamming-Code*

$$\left.\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array}\right| \tag{34}$$

[11] Die Codierung und Decodierung (einschließlich der Fehlerkorrektur) lässt sich für *Hamming-Codes* sehr effizient durch Matrix-Vektor-Operationen realisieren. Es gibt dazu eine umfangreiche Theorie. Die *Hamming-Codes* sind eine weit verbreitete und genutzte Klasse von fehlerkorrigierenden Codes.
EOS[12] ■

[11] Trotz Fehler sind *Hamming-Codes* sehr robust und können mit erhöhter Wahrscheinlichkeit eine fehlerhafte Nachricht wiederherstellen.
[12] End-Of-Semester