# Algorithmen und Lernverfahren

**Vorlesung von Prof. Dr. Boris Hollas**

Kay Förster

17. Oktober 2015

# Inhaltsverzeichnis

# 1 O-Notation, Laufzeitanalyse

Man kann den Zeitaufwand von Algorithmen nicht eindeutig bestimmen. Viel zu viele Faktoren (Hardware, parallel laufende Programme, Eingabereihenfolge, ...) spielen eine Rolle, so dass man mit normalen Mitteln niemals eine genaue und allgemeine Aussage über die benötigte Zeit machen kann. Es werden nun nicht mehr die benötigten Zeiten, sondern die benötigten "greifbaren" Schritte bei einer bestimmten Eingabelänge n beschrieben. Somit können Programme in Klassen (konstant, logarithmisch, lineas, polynomial, exponentiell, u.a.) eingeteilt werden.

## 1.1 O-Notation

Wir verwenden die $\mathcal{O}$-Notation um die Laufzeit und den Platzbedarf von Algorithmen anzugeben. Dazu betrachten wir die maximale Anzahl Schritte, die ein Algorithmus ausführt.

#### Beispiel

Es soll die Laufzeit der lineare Suche berechnet werden.

```
1 for (k := 1 to n) {
2     if (a[k] == gesuchter Wert)
3         return true;
4 }
5 return false;
```

Listing 1.1: Pseudocode zur Berechnung der Laufzeit

Lösung:

$$\begin{aligned} LZ &\leq n \cdot c_1 + c_2 \\ &\leq n \cdot c + c \text{ wobei } c = \max\{c_1, c_2\} \\ &\leq n \cdot c + n \cdot c \\ &= 2c \cdot n \in \mathcal{O}(n) \end{aligned}$$

Die Laufzeit der linearen Suche ist $\leq c \cdot n$, wobei $c$ eine Konstante ist, die von der Implentierung und dem Computer abhängt.

**Def.:** Für eine Funktion $f \geq 0$ ist $\mathcal{O}(f)$ die Menge aller Funktionen $g$ mit

$$0 \leq g(n) \leq c \cdot f(n)$$

für eine Konstante $c > 0$ für alle hinreichend großen $n$.

$\mathcal{O}(f) = \{g \mid 0 \leq g(n) \leq c \cdot f(n)$ für ein $c > 0$ und allen großen $n \in \mathcal{N}\}$

Die $\mathcal{O}$-Notation stellt somit die maximale Laufzeit (Worst Case Laufzeit) eines Algorithmuses dar. Die Funktion $g(n)$ ist die konkrete Laufzeit einer gegebenen Implentierung.

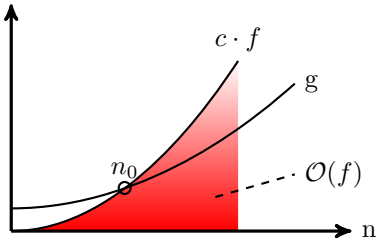



Abbildung 1.1: Grafische Darstellung der $\mathcal{O}$-Funktion

Es gelten folgende Rechenregeln:

- $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(\max\{f, g\})$
- $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$
- $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$
- $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$ (für alle $c \geq 0$)
- $f \leq g \Rightarrow \mathcal{O}(f) \subseteq \mathcal{O}(g)$
- $\mathcal{O}(c) = \mathcal{O}(1)$
- $c = \mathcal{O}(1)$

### Übung

$$\begin{aligned} 0 &\leq 17n^3 + 5n^2 + 2n + 8 \\ &\leq 17n^3 + 5n^3 + 2n^3 + 8n^3 \\ &\leq 32n^3 \in \mathcal{O}(n^3) \end{aligned}$$

### Übung

Berechnen Sie die Laufzeit des folgenden Codes:

```
1 for (k := 1 to n-1)
2     for (l := k+1 to n)
3         if (a[k] == a[l])
4             return Duplikat vorhanden;
5 return Kein Duplikat vorhanden;
```

Listing 1.2: Pseudocode zur Berechnung der Laufzeit

1. Möglichkeit zur Abschätzung der Laufzeit

$$\begin{aligned} LZ &\leq n \cdot n \cdot c + c' \\ &\leq n^2 \cdot c + n^2 \cdot c' \\ &\leq (c + c')n^2 \\ &\leq \mathcal{O}(n^2) \end{aligned}$$

2. Möglichkeit zur genaueren Abschätzung der Laufzeit

$$\begin{aligned} LZ &\leq \binom{n}{2} \cdot c + c' \\ &\leq \binom{n}{2} \cdot c + \binom{n}{2} \\ &= \binom{n}{2}(c+1) \\ &= \frac{n(n-1)}{2}(c+1) \\ \curvearrowright LZ &\leq n^2 \cdot \frac{c+1}{2} \in \mathcal{O}(n^2) \end{aligned}$$

### Übung

Berechnen Sie die Laufzeit des folgenden Algorithmuses:

$$\sum_{m=0}^{k} a_m n^m \text{ mit } a_k > 0, a_m \geq 0$$

Lösung:

$$\begin{aligned} LZ &\leq \sum_{m=0}^{k} a_m n^m \\ &\leq a_0 n^0 + \cdots + a_k n^k \\ &\leq a_k n^k \in \mathcal{O}(n^k) \end{aligned}$$

**Übung**

Zeigen Sie:

- $\log(n!) \in \mathcal{O}(n \log n)$

$$\begin{aligned} \mathcal{O}(\log(n!)) &\subseteq \mathcal{O}(n \log n) \\ \mathcal{O}(\log(n \cdot n-1 \cdot \ldots \cdot n-(n-1))) &\subseteq \\ \mathcal{O}(\log(n^n)) &\subseteq \end{aligned}$$

- $\binom{n}{k} \in \mathcal{O}(n^k)$
- $2^{n+\mathcal{O}(1)} \in \mathcal{O}(2^n)$

$$2^{n+\mathcal{O}(1)} \subseteq 2^{n+c} = 2^n \cdot 2^c \in \mathcal{O}(2^n)$$

## 1.2 Omega-Notation

Die Omega-Notation beschreibt die untere Schranke, d.h. wie lange ein Algorithmus bzw. ein Programm mindestens läuft (Best Case Laufzeit).

**Def.:** $\Omega(f) = \{g \mid$ Es gibt ein $c > 0$, sodass $g(n) \geq c \cdot f(n)$ für alle großen $n$ gilt.$\}$

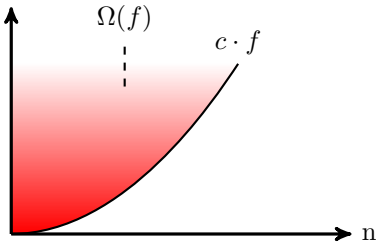



Abbildung 1.2: Grafische Darstellung der Omega-Funktion

## 1.3 Theta-Notation

Die Theta-Notation dient dazu, gleichzeitg eine obere und eine untere Schranke zu definieren.

**Def.:** $\Theta(f) = \{\mathcal{O} \cap \Omega(f)\}$

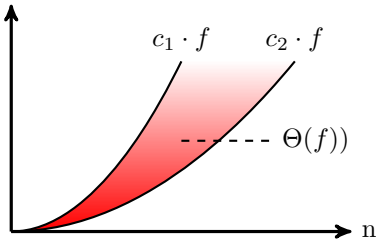



Abbildung 1.3: Grafische Darstellung der Theta-Funktion

### Übung

Zeigen Sie das $\frac{n(n+1)}{2} \in \Theta(n^2)$ liegt.

$$\mathcal{O}(\frac{n(n+1)}{2}) = \mathcal{O}(n(n+1)) = \mathcal{O}(n) \cdot \mathcal{O}(n+1) = \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$$

$$\frac{n(n+1)}{2} \geq \frac{n^2}{2} \in \Theta(n^2)$$
$$\curvearrowright \mathcal{O}(n^2) \cap \Theta(n^2) = \Omega(n^2)$$

# 2 Suchen und Sortieren

## 2.1 Lineare Suche

Die Lineare Suche ist der einfachste Suchalgorithmus überhaupt. Bei ihr wird solange ein Element nach dem anderen durchlaufen, bis ein Element mit dem gesuchten Schlüssel angetroffen wird. Die lineare Suche hat eine Laufzeit von $\mathcal{O}(n)$ (n ist die Anzahl der Elemente der Liste) und kann sowohl auf sortierte als auch unsortierte Listen angewendet werden.

```
1 public static int linearSuche(int gesucht, int[] daten)
2 {
3     for (int i = 0; i < daten.length; i++)
4         if (daten[i] == gesucht)
5             return i;
6     return -1;
7 }
```

Listing 2.1: Beispielimplementierung in Java

## 2.2 Binäre Suche

Die binäre Suche ist ein Algorithmus, der in einem Array sehr effizient ein gesuchtes Element findet bzw. eine zuverlässige Aussage über das Fehlen dieses Elementes liefert. Voraussetzung ist, dass die Elemente in dem Array entsprechend sortiert sind.

Dazu wird immer das mittlere Element eines Felds überprüft. Ist das Element gleich dem gesuchten Element ist die Suche beendet. Ansonsten wird geprüft ob das Element kleiner als das gesuchte Element ist, dann muss sich das Element in der vorderen Hälfte befinden, ansonsten in der hinteren. Dadurch wird der Suchbereich Schritt für Schritt halbiert bis das gesuchte Element gefunden ist oder nur noch eine Element vorhanden ist.

Für $n = 2^k$ lässt sich das Verhalten bei erfolgloser Suche als vollständiger Binärbaum darstellen. Jeder Knoten entspricht dabei ein Vergleich mit einem der n Elemente des Arrays. In einem vollständigen Binärbaum mit genau $n = 2^k$ Blättern besitzt jeder Pfad von der Wurzel zu einem Blatt die Länge $k = \log_2 n$. Daraus folgt: Die Laufzeit der binären Suche liegt bei $\mathcal{O}(\log_2 n)$.

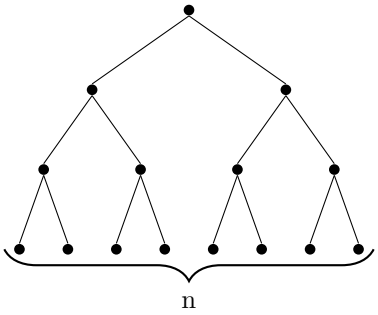



Abbildung 2.1: Einfacher Binärbaum

### Induktionsbeweis

Behauptung: Ein vollständiger Binärbaum der Tiefe $d$ besitzt genau $2^d$ Blätter.

$d = 0$ Ein vollständiger Binärbaum der Tiefe 0 besitzt $1 = 2^0$ Blätter.

$d \to d + 1$ Wenn wir in einem vollständigen Binärbaum der Tiefe $d + 1$ die Wurzel entfernen, erhalten wir zwei vollständige Binärbäume der Tiefe $d$. Diese besitzen nach Induktionsvorraussetzung jeweils genau $2^d$ Blätter. Daraus folgt, dass der Binärbaum der Tiefe $d + 1$ genau $2 \cdot 2^d = 2^{d+1}$ Blätter besitzt.

### Rekursionsgleichung

Eine alternative Möglichkeit die Laufzeit des Algorithmuses zu bestimmen ist es die Rekursionsgleichung zu lösen. Seien $n = 2^k$ und $V(n)$ die Anzahl der Vergleiche mit $\leq$. So gilt bei erfolgloser Suche:

$$\begin{aligned} V(1) &= 1 \\ V(n) &= 1 + V(\frac{n}{2}) \\ &= 1 + 1 + V(\frac{n}{4}) \\ &= 1 + 1 + 1 + V(\frac{n}{8}) = 3 + V(\frac{n}{2^3}) \\ &= k + V(\frac{n}{2^k}) = k + V(1) = k + 1 \in \mathcal{O}(\log n) \end{aligned}$$

#### Alternative Herrleitung

Eine weitere Alternative Herrleitung geht über die Herrleitung einer Schleife:

$$\text{Anzahl Schleifendurchläufe} \cdot \text{Aufwand pro Schleife}$$

$$\mathcal{O}(k) \cdot \mathcal{O}(1) = \mathcal{O}(k) = \mathcal{O}(\log n)$$

## 2.3 Binäre Suchbäume

Um auch in dynamischen Datenstrukturen zu suchen, lassen sich Suchbäume verwenden. Ein Suchbaum ist ein Binärbaum in dem gilt: Jeder in einem Knoten gespeicherte Wert ist größer als alle Knoten im linken Teilbaum und kleiner als alle Knoten im rechten Teilbaum. Folgende Funktionen sind nötig:

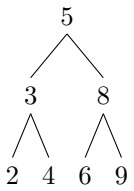



Abbildung 2.2: Beispiel für einen Suchbaum

**Suchen nach einem Wert** Das wird der Suchbaum, beginnend an der Wurzel, rekursiv durchgesucht. Die Laufzeit liegt bei $\mathcal{O}(n)$ für einen linear entarteten Baum und $\mathcal{O}(\log n)$ für einen vollständigen Baum.

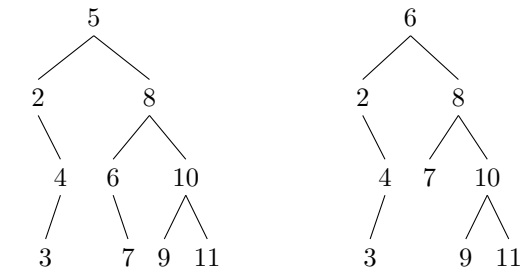
**Wert hinzufügen** Dazu wird der Baum wie oben durchsucht und ein Blatt mit dem neuen Wert hinzugefügt falls der Wert noch nicht vorhanden ist. Die Laufzeit ist gleich der des durchsuchens eines Baumes.

**Suchbaum aufbauen** Dazu kann mehrfach die obrige Funktion aufgerufen werden. Jedoch kann dabei ein unbalancierter Baum entstehen. Es gibt einen Algorithmus der einen optimalen Suchbaum aufbaut.

**Wert entfernen** Es werden im allgemeinen zwei Fälle unterschieden:

**einfacher Fall** Der zu entfernende Knoten hat keine oder genau einen Nachfolger

**schwieriger Fall** Knoten besitzt zwei Nachfolger. Eine einfache Lösung ist es, den zu entfernenden Knoten mit dem kleinsten Knoten im rechten Unterbaum zu ersetzen. Dazu wird der Knoten mit minimalen Wert im rechten Unterbaum gesucht und durch einen rekursiven Aufruf entfernt und als eine Wurzel angehängt.




(a) Ausgangssuchbaum (b) Resultat

Abbildung 2.3: Beispiel für das entfenen eines Wertes aus einem Suchbaum

## 2.4 Hashing

Gegeben sei eine Menge $U$ von Schlüsseln und eine Menge $S \subseteq U$ von zu verwaltenden Schlüssel. Hashing ist geeignet, wenn $|S|$ deutlich kleiner $|U|$ ist und sich $|S|$ nicht stark ändert. Beispiel:

- U ... alle möglichen ISBN-Nummern
- S ... tatsächlich in einer Buchhandlung vorkomenden ISBN-Nummern

Wir verwenden eine Hashfunktion $h : U \to T$, die eine Hashtabelle $T$ abbildet. Die Hastabelle $T$ ist ein Array von Zeigern welches auf die eigentlichen Datensätze zeigt.

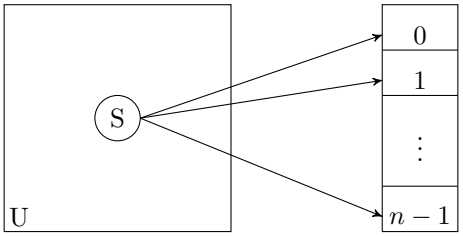



Abbildung 2.4: Hashfunktion die auf ein Array abbildet

Wenn wir annehmen, dass $h$ injektiv ist, lässt sich die Hashfunktionen sehr einfach implentieren:

**Suche** if $T(h(s)) \neq NIL \ldots$

**Einfügen** $T(h(s)) = \ldots$

**Löschen** $T(h(s)) = NIL$

**Wdh. injektiv:** Jedes Element der Zielmenge zeigt höchstens einmal oder gar nicht (zurück nach links) auf ein (Funktions-)Element der Ausgangs- oder Definitionsmenge. Es werden also keine zwei verschiedenen Elemente der Definitionsmenge (nach rechts) auf ein und dasselbe Element der Zielmenge, durch Anwendung der mathematischen Funktion, abgebildet. Die Zielmenge darf also nicht kleiner als die Definitionsmenge sein, d.h. nicht weniger Elemente beinhalten als die Definitionsmenge.

Es gilt also $|S| \leq m$ und zwei Schlüssel $s, s' \in S$ haben verschiedenene Hashwerte $h(s) \neq h(s')$.

Da $h$ jedoch nicht injektiv ist, tretten Kollisionen auf. Dies soll durch das folgende Beispiel verdeutlicht werden: In einer Hashtabelle mit $m = 100$ Einträgen werden $k$ zufällige Schlüssel eingetragen. Ab wann ist die Wahrscheinlickeit für eine Kollision $\geq 0,5$?

$$\begin{aligned} P(Kollesion) &= 1 - (\text{kleine Kollesion}) \\ &= 1 - 1 \cdot \frac{99}{100} \cdot \frac{98}{100} \cdot \ldots \cdot \frac{100-k+1}{100} > \frac{1}{2} \\ &\Leftrightarrow \quad k \geq 13 \end{aligned}$$

Allgemein ist bei etwa $\sqrt{2m}$ vielen Einträgen mit einer Kollesion zu rechnen. Die Hashfunktion sollte daher gut streuen. Eine gute Funktion ist $h(s) = s \mod m$ wobei $m$ eine Primzahl sein sollte. Die einfache Art der Kollisionsbehandlung ist Hashing mit Verkettung. Dabei ist jedes Element der Hashtabelle eine Liste.

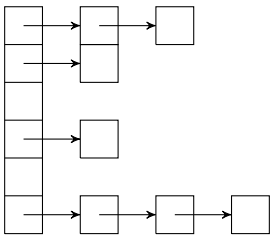


Abbildung 2.5: Hashing mit Verkettung; alle Daten, deren Schlüssel auf denselben Hashwert führen, werden in die entsprechende Liste eingetragen

Der Belegungsfaktor einer Hastabelle mit $n$ Elementen ist $\beta = \frac{n}{m}$. Man kann zeigen, dass die mittlere Länge der Überlaufliste $\beta$ ist. Daraus ergibt sich die mittlere Anzahl Suchschritte $1 + \beta$. Wenn $\beta$ beschränkt ist, bleibt die Suchzeit dabei in $\mathcal{O}(1)$.

Beim Überschreiten einen Schwellwertes für $\beta$ (typisch $\beta = \frac{3}{4}$) werden alle Einträge in eine größere Hashtabelle kopiert. Entsprechend beim Unterschreiten eine Schwellwertes in eine kleinere Hashtabelle. Die amortisierte Laufzeit (d.h. wir verteilen den Aufwand zum Kopieren auf alle vorherigen Hashoperationen) beträgt ebenfalls $\mathcal{O}(1)$.

## 2.5 Sortieren

**Satz.:** In einem Binärbaum mit mind. $2^k$ Blättern gibt es einen Pfad der Länge $k$.

Der Satz lässt sich mit Hilfe des Beweises im Kapitel 2.2 indirekt beweisen.

$$a \to b \equiv \neg b \to \neg a$$

Dies würde bedeuten wenn alle Pfade kürzer als $k$ sind, besitzt der Baum $< 2^k$ Blätter, was zu einem Widerspruch führt.

Ein Sortierverfahren, dass ausschließlich paarweise Vergleiche verwendet, lässt sich als Binärbaum darstellen. In jedem Knoten wird ein Vergleich $a \leq b$ ausgeführt. Jedes Blatt entspricht einer sortierten Folge, Jede sortierte Folge entspricht einer Permutation der Ausgangsfolge. Der Binärbaum besitzt daher $n!$ Blätter und deshalb einen Pfad der Länge $\log_2 n!$. Ein Sortierverfahren, dass paarweise Vergleiche ausführt besitzt daher eine Worst-Case Laufzeit in $\Omega(\log_2 n!) = \Omega(n \log_2 n) - \mathcal{O}(n))$.

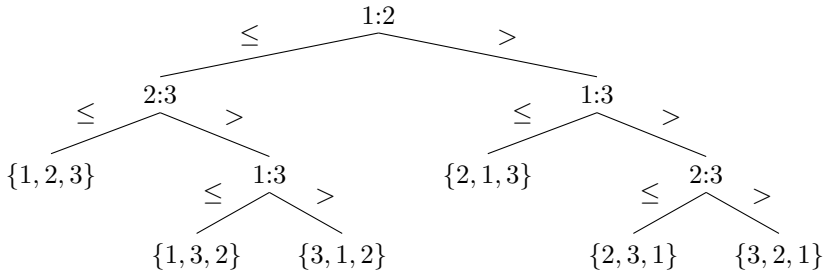



Abbildung 2.6: Entscheidungsbaum für 3 Elemente ($\{1, 2, 3\}$)

### 2.5.1 Mergesort

Mergesort betrachtet die zu sortierenden Daten als Liste und zerlegt sie in kleinere Listen, die jede für sich sortiert werden. Die sortierten kleinen Listen werden dann zu größeren Listen zusammengefügt, bis wieder eine sortierte Gesamtliste erreicht ist. Das Verfahren arbeitet bei Arrays in der Regel nicht in-place.

Das Verhalten lässt sich wie in Abbildung 2.7 als Baum darstellen. Vereinfacht nehmen wir $n = 2^k$ an.

Beim Zusammenfügen fällt auf jeder Ebene der Aufwand $\mathcal{O}(n)$ an. Der Baum besitzt $n$ Blätter, die Liste der der Länge 1 entsprechen (Mehr als n Listen der Länge 1 können nicht erzeugt werden $\curvearrowright$ n Blätter). Der Baum hat daher die Tiefe $\log_2 n$ (d.h. k). Die

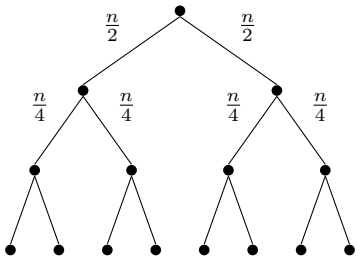



Abbildung 2.7: Darstellung als Binärbaum

Laufzeit liegt daher in in $\mathcal{O}(n \cdot \log n)$. Auch die mittlere Laufzeit von Quicksort liegt in $\mathcal{O}(n \log n)$.

### Übung

Bestimmen Sie die Laufzeit durch eine Rekursionsgleichung. $V(n)$ ist dabei die Anzahl der Vergleiche.

$$
\begin{aligned}
V(1) &= 0 \\
V(n) &= \underbrace{n}_{\text{Zusammenfügen}} + \underbrace{V(\frac{n}{2}) + V(\frac{n}{2})}_{\text{Sortieren}} = n + 2V(\frac{n}{2}) \\
V(n+1) &= 2n + 4V(\frac{2}{4}) \\
&\vdots \\
&= k \cdot n + 2^k \cdot V(\frac{2}{2^k}) \\
&= n \cdot \log n
\end{aligned}
$$

### Worst Case Laufzeit von Quicksort

Im Worst Case wird das Pivotelement stehts so gewählt, dass es das größte oder das kleinste Element der Liste ist. Dies ist etwa der Fall, wenn als Pivotelement stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt.

$$
\begin{aligned}
V(1) &= n - 1 + V(n-1) \\
&= n - 1 + n - 2 + V(n-2) \\
&= \sum_{k=1}^{n-1} k + V(0) = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)
\end{aligned}
$$

### 2.5.2 Heap Sort

Ein Heap ist ein Binärbaum, in dem jeder Knoten einen kleineren (Min-Heap) bzw. einen größeren (Max-Heap) Wert besitzt als seine Nachfolger (Abbildung 2.8). Ein

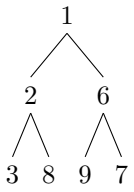



Abbildung 2.8: Beispiel für einen Min-Heap

Linksbaum ist ein Heap, der effiziente Heapoperationen ermöglicht. Wir stellen Binärbäume als erweiterte Binärbäume dar, so dass jeder innere Knoten genau 2 Nachfolger hat (Abbildung 2.9). Für einen Knoten $x$ ist $s(x)$ die Länge des Pfades zu einem Blatt.

(a) Binärbaum (b) erweiterte Binärbaum

Abbildung 2.9: Erweiterung eines Binärbäumes

Ein Binärbaum ist ein Linksbaum wenn für jeden inneren Knoten $x$ gilt:

$$s(\text{linker Nachfolger}(x)) \geq s(\text{rechter Nachfolger}(x))$$

#### Eigenschaften

1. $s(root)$ ist die Länge des Pfades ganz rechts.
2. Für die Anzahl $n$ der inneren Knoten gilt:

$$n \geq \sum_{k=0}^{s(root)-1} 2^k = 2^{s(root)} - 1$$

3. Aus 1, 2 folgt: Die Länge des Pfades ganz rechts liegt in $\mathcal{O}(\log n)$.

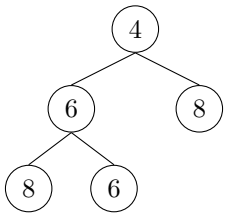
### Operationen

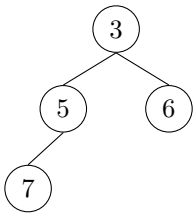
- put
- removeMin
- init
- merge

### Merge-Operation für einen Linksbaum

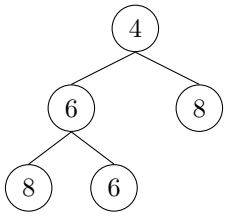
Wir suchen den Baum mit dem kleineren Wurzelwert und betrachten dessen rechten Teilbaum. Merge wird wird rekursiv aufgerufen für diesen rechten Teilbaum und den anderen Baum.

Gegebene Bäume:




Begin der Rekursion:



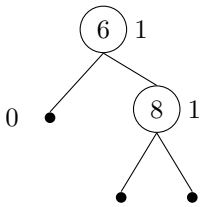


Nächster Schritt:

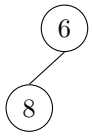


Hier endet die Rekursion, da der rechte Teilbaum des Baumes mit der kleineren Wurzel leer ist. Unter dem Baum mit der kleinen Wurzel wird rechts der andere Baum gehängt. Wenn dabei die Linksbaumeigenschaft verletzt wird, werden die Teilbäume vertauscht.

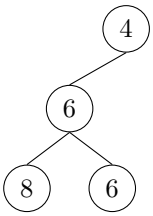
Da dies kein Linksbaum ist,

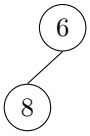



ergibt sich

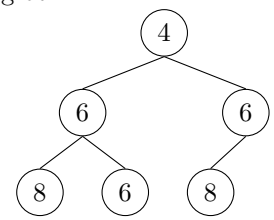



Nächster Schritt: Merge von

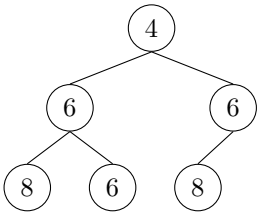


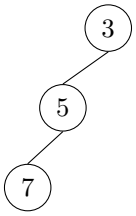




ergibt

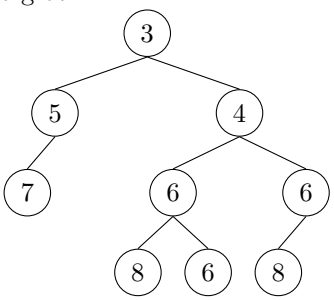



Nächster Schritt: Merge von

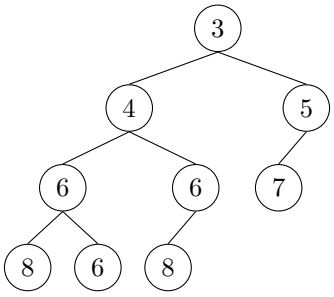



ergibt




Da dieser Baum die Linksbaumeigenschaft verletzt muss er umgeformt werden



### Laufzeitanalyse der Operation Merge

Für die Laufzeitanalyse sind bei der Implentierung folgende Schritte notwendig:

- Erzeugen der Teilbäume vor jeden Rekursionsschritt: $\mathcal{O}(1)$
- Zusammenbau der Teilbäume nach der Rekursion (nur wenn die s-Werte zwischengespeichert werden): $\mathcal{O}(1)$
- Anzahl der rekursiven Aufrufe: Die Rekursion endet, wenn der rechte Teilbaum nur aus einem Knoten besteht. Bei jedem rekursiven Aufruf verkleinert sich einer der rechten Teilbäume. Die Laufzeit beträgt daher $\mathcal{O}(\log n)$

Mit Hilfe der Merge Operation können alle anderen Heap-Operationen realisiert werden:

**put** merge(heap, <neuer Knoten>)

**removeMin** Wurzel enfernen und die zwei neuen Teilbäume mergen

**init** Mit Hilfe von put in der Zeit $\mathcal{O}(n \log n)$. Es gibt einen besseren Algorithmus für init welcher eine Laufzeit von $\mathcal{O}(n)$ besitzt.

Mergesort kann man durch folgenden Algorithmus darstellen:

- Max-Heap aufbauen: $\mathcal{O}(n)$
- n-mal die Funktion removeMax aufrufen und Elemente am Kopf einer Liste anfügen: $\mathcal{O}(n \log n)$

Damit ist die Laufzeit $\mathcal{O}(n \log n)$, welche gleich der Average Laufzeit von Quicksort ist. Es gibt ebenfalls eine In-Place-Implentierung die dem Heap als Array darstellt. Damit hat HeapSort alle Vorteile von Quicksort und Mergesort ohne deren Nachteile zu besitzten.

# 3 Dynamisches Programmieren

Dynamische Programmierung ist eine Methode zum algorithmischen Lösen von Optimierungsproblemen. Es kann dann erfolgreich eingesetzt werden, wenn das Optimierungsproblem aus vielen gleichartigen Teilproblemen besteht, und eine optimale Lösung des Problems sich aus optimalen Lösungen der Teilprobleme zusammensetzt. In der dynamischen Programmierung werden zuerst die optimalen Lösungen der kleinsten Teilprobleme direkt berechnet und dann geeignet zu einer Lösung eines nächstgrößeren Teilproblems zusammengesetzt. Dieses Verfahren setzt man fort, bis das ursprüngliche Problem gelöst wurde. Einmal berechnete Teilergebnisse werden in einer Tabelle gespeichert. Bei nachfolgenden Berechnungen gleichartiger Teilprobleme wird auf diese Zwischenlösungen zurückgegriffen, anstatt sie jedes Mal neu zu berechnen. Wird die dynamische Programmierung konsequent eingesetzt, vermeidet sie kostspielige Rekursionen, weil bekannte Teilergebnisse wiederverwendet werden.

### Beispiel

Ein Läufer nimmt in jeden Schritt 1 oder 2 Stufen auf einmal. Wieviele Möglichkeiten gibt es, $n$ Stufen herrauf zu steigen?

$$\begin{aligned} t_1 &= 1 \\ t_2 &= 2 \\ t_n &= t_{n-1} + t_{n-2} \end{aligned}$$

Man kann zeigen das $t_n \in \mathcal{O}(1,618^n)$ liegt. Wenn $t_n$ rekursiv berechnet wird, werden die meisten $t_n$ mehrfach berechnet, siehe Abbildung 3.1. Um $t_n$ durch dynamische Programmierung zu berechnen, speichert man die Werte von $t_n$ in einem Feld. Die Laufzeit beträgt somit: $\mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

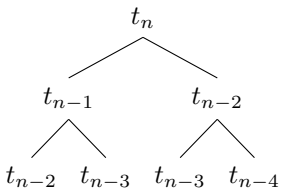



Abbildung 3.1: Verdeutlichung der mehrfachen Berechnung von Werten

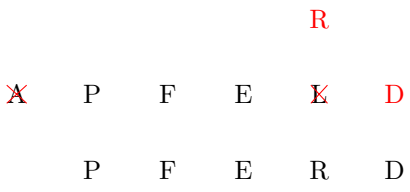
```
for(i=1; i<=n; i++)
{
    if(n==1)
        t[n] = 1;
    else if(n==2)
        t[n] = 2;
    else
        t[n] = t[n-1] + t[n-2];
}
```

Listing 3.1: Beispielimplementierung in Java

## 3.1 Editierdistanz

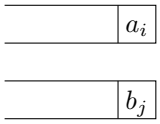
Die Editierdistanz zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln. In der Praxis wird die Editierdistanz zur Bestimmung der Ähnlichkeit von Zeichenketten beispielsweise zur Rechtschreibprüfung, DNA-Sequenzvergleich oder bei der Duplikaterkennung angewandt.

Gegeben seien zwei Strings a, b, wieviele Editieroperationen sind nötig, um a in b zu überführen? Lösung: Die Editierdistanz beträgt 3.



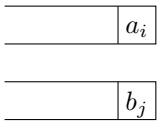

Sei $d(i,j)$ die Editierdistanz zwischen den Teilwörtern $a_1 \dots a_i, b_1 \dots b_j$. So gibt es folgende Möglichkeiten:

- Ein Matching kann verlängert werden (MATCH)



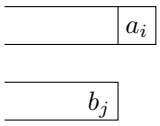

  $a_i = b_j \rightarrow$ Bewertung $d(i-1, j-1)$

- Missmatch

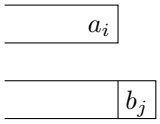



  $a_i \neq b_j \rightarrow$ Bewertung $d(i-1, j-1) + 1$

- Löschen / Hinzufügen




  Bewertung $d(i-1, j) + 1$




  Bewertung $d(i, j-1) + 1$

Man wählt in jeden Schritt die Möglichkeit mit der besten Bewertung

$$d(i,j) = \begin{cases} j & \text{für } i = 0 \\ i & \text{für } j = 0 \\ \min\{ \quad d(i-1, j-1) + 1_{a_i \neq b_j}, & \\ \qquad d(i-1, j) + 1, & \text{für } i, j > 0 \\ \qquad d(i, j-1) + 1\} & \end{cases}$$

wobei gilt:

$$1_{a_i \neq b_j} = \begin{cases} 0 & \text{für } x = y \\ 1 & \text{für } x \neq y \end{cases}$$

Der Aufwand für die Berechnung in einem 2-dimensionalten Feld beträgt:

$$\underbrace{\mathcal{O}(n \cdot m)}_{\text{Größe der Tabelle}} \cdot \underbrace{\mathcal{O}(1)}_{\text{Vergleichsoperation}}$$

**Beispiel**

Berechen Sie die Edierdistanz der Wörter: APFEL, PFERD.

| | $\varepsilon$ | A | P | F | E | L |
|---|---|---|---|---|---|---|
| $\varepsilon$ | 0 | 1 | 2 | 3 | 4 | 5 |
| P | 1 | 1 | 1 | 2 | 3 | 4 |
| F | 2 | 2 | 2 | 1 | 2 | 3 |
| E | 3 | 3 | 3 | 2 | 1 | 2 |
| R | 4 | 4 | 4 | 3 | 2 | 2 |
| D | 5 | 5 | 5 | 4 | 3 | **3** |

## 3.2 Längste gemeinsame Teilffolge

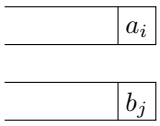
Eine längste gemeinsame Teilffolge kann durch Streichen von Zeichen erzeugt werden. Bespiel: Die längste gemeinsame Teilffolge ist in diesem Bespiel 4.

A N A N ~~A~~ ~~S~~

~~B~~ A N A N ~~E~~

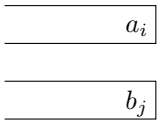
Sei $d(i, j)$ die Länge der längsten gemeinsamen Teilffolge von $a_1 \dots a_i, b_1 \dots b_j$. So gibt es folgende Möglichkeiten:

- Teilffolge verlängern (sodass die letzten beiden Zeichen übereinstimmen)

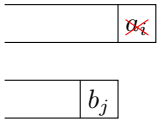
  $a_i = b_j \rightarrow$ Bewertung: $d(i-1, j-1) + 1$

- Zeichen streichen

  $a_i \neq b_j \rightarrow$ Bewertung: $d(i-1, j-1)$

- Eines der letzten beiden Zeichen streichen




  Bewertung: $d(i-1,j)$,
  entsprechend andere Fall: $d(i,j-1)$

Damit gilt:

$$d(i,j) = \begin{cases} 0 & \text{für } i=0 \text{ oder } j=0 \\ \max\{ \quad d(i-1,j-1) + 1_{a_i=b_j}, & \\ \qquad d(i-1,j), & \text{für } i,j>0 \\ \qquad d(i,j-1)\} & \end{cases}$$

Für die Laufzeit beträgt wie bei der Edierdistanz: $\mathcal{O}(n \cdot m) \cdot \mathcal{O}(1) = \mathcal{O}(mn)$

#### Beispiel

Berechen Sie die längste gemeinsame Teilfolge der Wörter: BANANE, ANANAS.

| | | B | A | N | A | N | E |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| N | 0 | 0 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 0 | 1 | 2 | 3 | 3 | 3 |
| N | 0 | 0 | 1 | 2 | 3 | 4 | 4 |
| A | 0 | 0 | 1 | 2 | 3 | 4 | 4 |
| S | 0 | 0 | 1 | 2 | 3 | 4 | **4** |

## 3.3 Komplexitätsklassen

- P enthält alle Problme, die sich in der Zeit $\mathcal{O}(n^k)$ für ein $k > 0$ lösen lassen, wobei $n$ die Länge der Eingabe ist. Beispiel: Sortieren ($\mathcal{O}(n \log n)$, Editierdistanz ($\mathcal{O}(m \cdot n) \subseteq \mathcal{O}((m+n)^2)$.
- Klasse der NP-vollständigen Probleme. Nicht in Polynoialzeit lösbar.

  Besteht aus allen Problemen, für die kein Algorithmus in $\mathcal{O}(n^k)$ bekannt ist. Es gilt: Wenn es ein NP-vollständiges Problem gibt, das in P liegt, dann liegen alle NP-vollständigen Probleme in P.

  Wichtige NP-vollständige Probleme:

  – Erfüllbarkeitsproblem der Aussagenlogik (Gegeben sei eine Formel der Aussagenlogik. Ist diese erfüllbar?)

– Hamilton-Kreis (besitzt ein Graph einen Kreis, der jeden Knoten genau einmal besucht?)

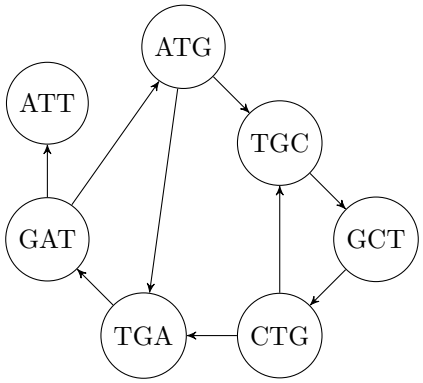
– Travelling Salesmann Problem

– Rucksack-Problem

**Anwendungen**

- Erfüllbarkeitsproblem der Aussagenlogik (Bsp. Äquivalenz von Schaltkreisem)
- Hamilton-Pfad (Bsp. DNA Sequenzierung)

  Bsp. ATGCTGATT → Bruchstücke ATG, GCT, TGC, CTG, TGA, GAT, ATT
  Suche nach einen Kreis in dem jedes Bruchstück genau einmal vorkommt.




Alle Probleme in NP (und damit auch alle NP-vollständigen Probleme) lassen sich in der Zeit $2^{\mathcal{O}(n^k)}$ entscheiden.

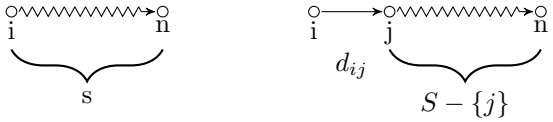
## 3.4 Travelling Salesmann Problem

Gesucht ist die kürzeste Rundreise durch $n$ Städte, wobei jede Stadt genau einmal besucht wird. Das TSP ist NP-vollständig.

Sei $(d_{ij})_{1 \leq i,j \leq n}$ die entsprechende Entfernungsmatrix. Wir betrachten den allgemeinen Fall in dem $d_{ij} \neq d_{ji}$ sowie $d_{ij} = \infty$ gelten kann. Der naive Algorithmus prüft alle $n!$ Kombinationen und hat eine Laufzeit in $\Omega(n!) \cdot \mathcal{O}(n) = \Omega(n!)$.

Optimalitätsprinzip: Wenn eine optimale Rundreise bei Stadt 1 beginnt, dann durch Stadt k führt, dann muss der Weg von k durch die Städte in $\{2, \ldots, n\} - \{k\}$ ebenfalls optimal sein.

Sein $l(S,i)$ die Länge des kürzesten Pfades, der bei $i$ beginnt, dann durch jedes $j \in S$ genau einmal führt und bei $n$ endet. Die Länge des kürzesten Rundweges ist dann

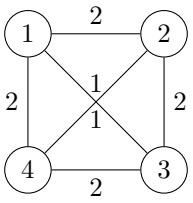



$l(\{1,\ldots,n-1\},n)$. Es gilt:

$$l(S,i) = \begin{cases} d_{in} & \text{für } S = \varnothing \text{(keine Zwischenstädte)} \\ \min\limits_{j \in S}\{d_{ij} + l(S-\{j\},j)\} & \text{für alle } S \neq \varnothing \end{cases}$$

Mit Hilfe dieses Algorithmuses kann eine Matix erstellt werden, in welcher die Länge der Rundwege gepeichert werden. In der x-Achse werden die Knoten von 1 bis $n-1$ eingetragen und in der y-Achse die Teilmengen in aufsteigender Mächtigkeit. Dieser besitzt eine Laufzweit von $\mathcal{O}(2^{n-1}\cdot(n-1))\cdot\mathcal{O}(n) = \mathcal{O}(n^2\cdot 2^n)$. Diese wächst langsamer als $\mathcal{O}(n!)$.

Gegeben sei der nachfolgende Graph mit 4 Städten. Gesucht ist der kürzeste Weg zur Stadt 4.




| | | | | |
|---|---|---|---|---|
| {1,2,3} | X | X | X | $2+4=6$<br>Lösung: $1+5=6$<br>$2+4=6$ |
| {2,3} | $1+3=4$<br>$2+4=6$ | X | X | |
| {1,3} | X | $2+3=5$<br>$2+3=5$ | X | |
| {1,2} | X | X | $2+4=6$<br>$1+3=4$ | |
| {3} | $2+2=3$ | $2+2=4$ | X | |
| {2} | $2+1=3$ | X | $2+1=3$ | |
| {1} | X | $2+2=4$ | $1+2=3$ | |
| {∅} | 2 | 1 | 2 | |
| | 1 | 2 | 3 | 4 |

**Def.:** Sei $\varepsilon > 1$. Ein Minimierungsproblem heißt $\varepsilon$-approximierbar, wenn es einen Algorithmus mit polynomieller Laufzeit gibt, der eine Lösung liefert, die höchstems um $\varepsilon$ größer ist als das Optimum.

Falls $P \neq NP$ gilt, ist für kein $\varepsilon > 1$ TSP $\varepsilon$-approximierbar. Falls die Entfernungsmatrix jedoch die Dreiecksungleichung

$$d_{uv} \leq d_{uw} + d_{wv}$$

gilt ($\Delta$-TSP), dann ist das Problem $\frac{3}{2}$-approximierbar. Wenn es für die Anwendung nicht notwendig ist, dass kein Punkt mehrfach besucht wird, kann die Dreiecksungleichung immer erfüllt werden, indem die Kante von $u$ nach $v$ durch die kürzeste Verbindung von $u$ nach $v$ ersetzt wird.

Ein Spezialfall des $\Delta$-TSP ist das Euklidischen TSP, bei dem die Entfernungen gleich dem geometrischen Abstand sind. Für jedes $\delta > 1$ ist das Euklidische TSP in der Ebene $1 + \frac{1}{\delta}$-approximierbar. Der zugehörige Approximationsalgorithmus besitzt eine Laufzeit in $\mathcal{O}(n \log(n))^{\mathcal{O}(\delta)}$.

Anwendungen des TSP:

- Roboter soll Löcher in eine Platine bohren
- Auf einer Fertigungsstraße sollen Produkte $P_1, \ldots, P_n$ herrgestellt werden. Dabei muss die Fertigungsstraße jeweils umgerüstet werden. Wenn $d_{uv}$ die Zeit ist, die für das umrüsten von $P_U$ nach $P_V$ benötigt wird, muss ein TSP für $(d_{uv})$ gelöst werden.. Für kleine $n$ kann das exakte TSP und sonst das $\Delta$-TSP (Bedingung: Dreiecksungleichung)zur Approximation des Optimums verwendet werden.
- DNA-Sequenzierung (Erzeugung von DNA-Bruchstücken, Suche nach Überlappungen, kürzester Pfad durch diese Knoten (Besser wäre allerdings der Aufbau einen Graphen und Suche eines euklidischen Kreises))

## 3.5 Rucksackproblem

Das Rucksackproblem ist ein Optimierungsproblem der Kombinatorik. Aus einer Menge von Objekten, die jeweils ein Gewicht und einen Nutzwert haben, soll eine Teilmenge ausgewählt werden, deren Gesamtgewicht eine vorgegebene Gewichtsschranke nicht überschreitet. Unter dieser Bedingung soll der Nutzwert der ausgewählten Objekte maximiert werden. Anwendungen:

- öffentliche Haushaltsführung
- Reduzierung des Verschnits (Bsp. Fließen, Folien)
- Logistik (Bsp. Transport mittels Frachschiff)

Gegebenen seien n Gegenstände mit den Werten $x_1, \ldots, x_n$ und gesucht sei eine Menge $S \subseteq \{1, \ldots, n\}$ mit $\max \sum_{s \in S} x_s$ und $\sum_{s \in S} x_s \leq y$.

Das Rucksackproblem ist NP-vollständig, es lässt sich mit einem dynamischen Programmier-Algorithmus wie folgt lösen: Sei $r(n, y)$ der Wert einer Lösung des Rucksackproblems für die Werte $x_1, \ldots, x_n$ und der Rucksackgröße $y$ dann gilt:

- $r(n, y) = 0$ für $n = 0$
- $r(n, y) = r(n-1, y)$ für $y > 0$ und $x_n > y$
- Für $n > 0$ und $x_n < y$ kann der Gegenstand n entweder eingepackt oder nicht eingepackt werden
  - Wenn Gegenstand nicht eingepackt wird, gilt:

    $$r(n, y) = r(n-1, y)$$

  - Wenn Gegenstand n eingepackt wird:

    $$r(n, y) = r(n-1, y - x_n) + x_n$$

$$r(n, y) = \begin{cases} 0 & \text{für } n = 0 \\ r(n-1, y) & \text{für } n > 0 \wedge x_n > y \\ \max\{r(n-1, y), & \\ \quad r(n-1, y - x_n) + x_n\} & \text{sonst} \end{cases}$$

Die laufzeit des Rucksack-Algorithmuses beträgt: $\mathcal{O}(ny)$. Es stellt sich daher die Frage ob dies ein polynomieller Algorithmus für das Rucksackproblem ist (woraus P=NP folgen würde)?

Die Laufzeit eines Algorithmuses wird gemessen in der Länge der Eingabe. Im Falle des Rucksackproblems ist die Eingabe das Tupel $(x_1, \ldots, x_n, y)$. Wenn diese Werte binär codiert werden, hat dieses Tupel eine Länge $\leq (n+1)|y| = (n+1)\log(y) + \mathcal{O}(1)$. Die Laufzeit liegt daher in $\Omega(ny) = \Omega(n \cdot 2^{\log(y)+\mathcal{O}(1)}) = \Omega(n \cdot 2^{|x|})$

**Def.:** Ein Algorithmus heißt pseudopolynomiell, wenn seine Laufzeit durch ein Polynom in der Eingabelänge und der größten, in der Eingabe vorkomenden Zahl beschränkt ist.

Mit Hilfe eines NP-vollständigen Problems kann man die NP-Vollständigkeit weitere Probleme nachweisen. Prinzip: Wenn sich mit ein angenommener pseudopolynomieller Algorithmus für ein Problem B auch für ein NP vollständiges Problem A in polynomieller Zeit lösen lässt.

```
boolean A(Input x)
    return B(f(x))
```

Listing 3.2: Pseudocode des Prinzips

Dabei "übersetzt" f das Problem A in das Problem B.

### Übung

Sei

$$KP = \bigcup_{n \geq 1} \{(x_1, \ldots, x_n, y, v_1, \ldots, v_n, t) \mid \text{Es gibt ein } S \subseteq \{1, \ldots, n\} \text{mit}$$

$$\sum_{s \in S} x_s \leq y \text{ und } t \leq \sum_{s \in S} v_s\}$$

Zeigen Sie, dass KP NP-vollständig ist.

```
set Rucksack(x1 ... xn, y)
    return KP(x1, ... xn, y, x1 ... xn)
```

Listing 3.3: Beispiellösung Übung

Wenn KP polynomial ist, dann ist auch Rucksack polynomial. Widerspruch da Rucksack NP.

# 4 Graphalgorithmen

Aus der Vorlesung Künstliche Intelligenz sind bereits die Algorithmen Breiten- und Tiefensuche bekannt.

```
boolean bfs(start, goal) {

    // Anfangs sind keine Knoten besucht
    for(v in V)
        discovered[v] = false;

    // Mit Start-Knoten beginnen
    queue.enqueue(start)
    discovered[start] = true;

    while(!queue.isEmpty()){
        // Erstes Element von der queue nehmen
        u = queue.dequeue;

        // Testen ob Zielknoten gefunden
        if(u == goal)
            return true;

        // alle Nachfolge-Knoten, ...
        for(v in adj[u]))
            // ... die noch nicht besucht wurden ...
            if(!discovered[v]){
                // ... zur queue hinzufuegen ...
                queue.enqueue(v);
                // ... und als bereits gesehen markieren
                discovered[u] = true;
            }
    }
    return false;
}
```

Listing 4.1: Beispiel Algorithmus für die Breitensuche

Für dieses Beispiel fallen folgende Laufzeiten an:

- Zeile 4-5: $\mathcal{O}(|V|)$
- Zeile 8-9: $\mathcal{O}(1)$
- Zeile 13-17: $\mathcal{O}(1)$
- Zeile 20-27: $\mathcal{O}(\deg(u))$
- Zeile 29: $\mathcal{O}(1)$

$\deg(u)$ steht dabei für den Grad des Knotens (=Anzahl der Nachbarn). Ein Knoten $v$ hat den Grad $k$ wenn $v$ mit genau $k$ anderen Knoten verbunden ist. Wir schreiben dafür $\deg(u) = k$.

**Satz:** Die Laufzeit der Breitensuche liegt in $\mathcal{O}(|V| + |E|)$.

Beweis:

Das initialisieren des Feldes discovered benötigt die Zeit $\mathcal{O}(|V|)$. Um die unbesuchten Nachbarn des Knotens $u$ zu bestimmen fällt der Aufwand $\mathcal{O}(\deg(u))$ an. Da jeder Knoten höchstens einmal aus der Warteschlange entnommen wird, wird auch die while Schleife für jeden Knoten höchstens einmal durchlaufen. Der gesamte Aufwand ist damit

$$\mathcal{O}(|V|) + \sum_{u \in V} \mathcal{O}(\deg(u) = \mathcal{O}(|V|) + \mathcal{O}(|E|) = \mathcal{O}(|V| + |E|)$$

Die Tiefensuche lässt sich implentieren wie die Breitensuche, wenn anstelle der Warteschlange ein Stack verwendet wird. Die Laufzeit ist gleich der Breitensuche: $\mathcal{O}(|V| + |E|)$.

## 4.1 Verallgemeinerung der A*-Suche

Hierbei wird eine heuristische Bewertungsfunktion $f$ verwendet um Knoten einzusortieren. Die heuristische Bewertungsfunktion hat die Gestalt

$$f(v) = g(v) + h(v)$$

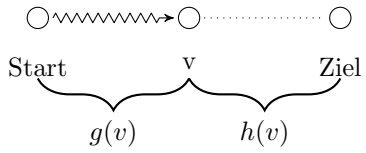
wobei

- $g(v)$ die Kosten bis zum Knoten v
- $h(v)$ eine zulässige Kostenschätzfunktion

sind. Eine Kostenschätzfunktion $h$ ist zulässig, wenn sie die Kosten zum Ziel nicht überschätzt.

Bei einer Navigation könnten die Funktionen wie folgt definiert sein:

- $g(v)$: Strecke von Start bis v




- $h(v)$: Luftlinieentfernung von v zum Ziel

#### Übung

Für ein Streckennetz sind Entfernungen und durchschnittliche Geschwindigkeiten bekannt. Wie kann die schnellste Route von A nach B gefunden werden?

- $g(v)$: Zeit für Strecke von Start bis v ($t = \frac{s}{v}$)
- $h(v)$: $t = \frac{\text{Luftlinie bis zum Ziel}}{\text{maximale Geschwindigkeit der verbleibenden Kanten}}$

## 4.2 Topologisches Sortieren

**Def.:** Ein DAG (Directed acyclic graph) ist ein gerichteter Graph, der keine gerichteten Kreise enthält. Eine topologische Sortierung eines DAG $G = (V, E)$ ist eine Abbildung

$$f : V \to \mathbb{N} \text{ mit } f(u) < f(v) \text{ für } (u, v) \in E$$

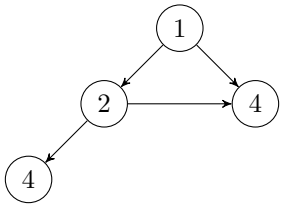



Abbildung 4.1: Beispiel für ein Directed acyclic graph (DAG)

Jeder vollständige Graph oder Kreis lässt sich nicht topologisch sortieren. Eine topologische Sortierung kann durch eine Tiefensuche bestimmt werden.

```
topsort:
    for (v in V)
        markiere v mit weiss
    for (v in V)
        tiefensuche(v)

tiefensuche(v):
    v grau:
        Fehler("Kreis vorhanden")
    v weiss:
        markiere v mit grau
        for (u in adj[v])
            tiefensuche(u)
        markiere v mit schwarz
        fuege v an den Kopf einer Liste
```

**Satz:** Für jeden DAG $G = (V, E)$ erzeugt Topsort eine topologische Sortierung von $G$.

### Beweis

Sei $(u, v) \in E$. In u und in v werden je eine Tiefensuche gestartet. Die in v gestartete Tiefensuche endet früher als die in u gestartete Tiefensuche. Daher wird u links von v in die Liste eingefügt und erhält daher eine kleinere Nummer als v.

# 5 Datenkompression

## 5.1 Huffman Codierung

**Idee:**

Häufige Zeichen erhalten kurze Codewörter, seltene Zeichen längere Codewörter. Gesucht ist ein Code, so dass die mittlere Codewortlänge minimal ist.

**Problem:**

Die Codierung muss eindeutig decodierbar sein.

**Def.:** Ein Präfixcode ist ein Code, so dass kein Codewort Präfix eines anderen Codewortes ist.

**Beispiel**

| Zeichen | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Wahrscheinlichkeit | 0,45 | 0,13 | 0,12 | 0,16 | 0,09 | 0,05 |
| Code | 0 | 101 | 100 | 111 | 1101 | 1100 |

Dieser Code ist optimal. In Abbildung 5.1 ist der Code als Codewortbaum dargestellt. Der Huffman-Algorithmus ist ein Greedy-Algorithmus (wählt den nächsten Schritt nach der besten Wahrscheinlichkeit) der einen optimalen Präfix-Code konstruiert.

### 5.1.1 Implentierung

```
1 Initialisierung: Jedes Zeichen ist ein Baum mit einem Knoten.
2
3 while (mehr als ein Baum vorhanden)
4     Verbinde zwei Baeume mit den beiden niedrigsten
5     Wahrscheinlichkeiten zu einem neuen Baum,
6     die Wahrscheinlichkeiteb addieren sich.
```

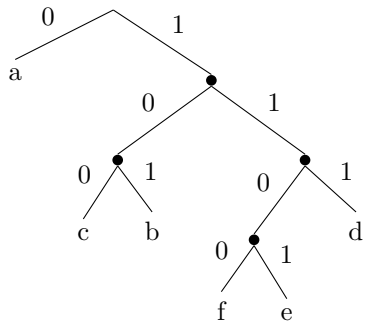
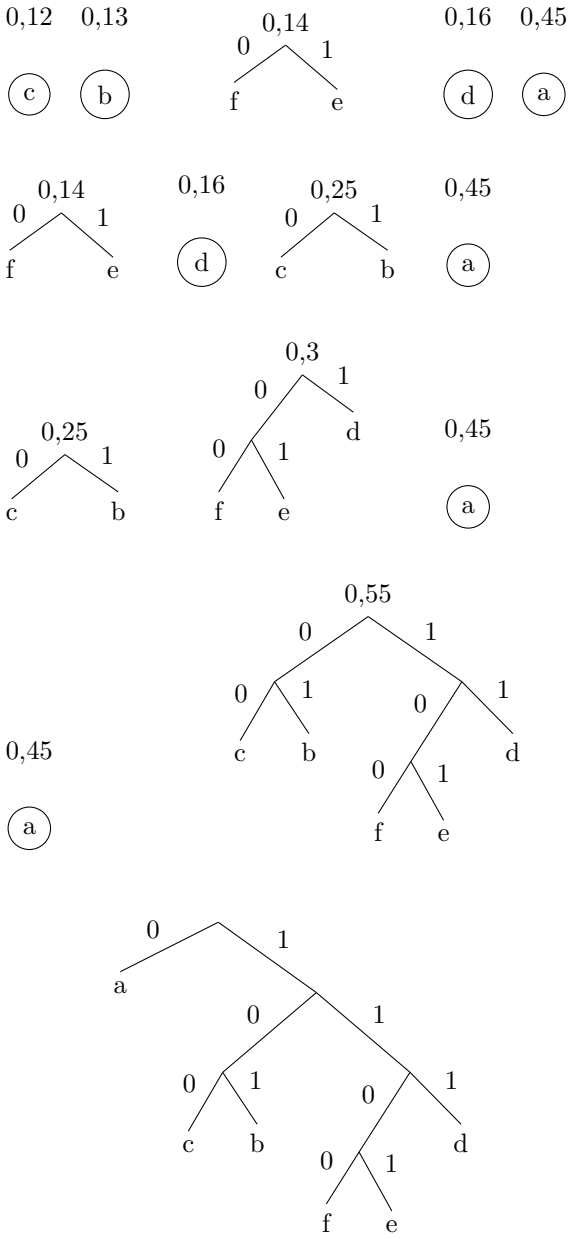



Abbildung 5.1: Die Folge 1001010 lässt sich eindeutig decodieren zu cba

**Beispiel**

| 0,05 | 0,09 | 0,12 | 0,13 | 0,16 | 0,45 |
|---|---|---|---|---|---|
| f | e | c | b | d | a |


0,12
0,13
0,14
0
1
f
e
0,16
0,45
c
b
d
a
0,14
0
1
f
e
0,16
d
0,25
0
1
c
b
0,45
a
0,25
0
1
c
b
0,3
0
1
d
0
1
f
e
0,45
a
0,45
a
0,55
0
1
0
1
0
1
c
b
d
0
1
f
e
0
1
a
0
1
0
1
0
1
c
b
d
0
1
f
e

# 6 Lernverfahren

## 6.1 Entscheidungsbäume

Entscheidungsbäume dienen der Klassifizierung von Daten. Die Inneren Knoten sind dabei die Attribute und die Blätter stellen die Zielvariablen da.

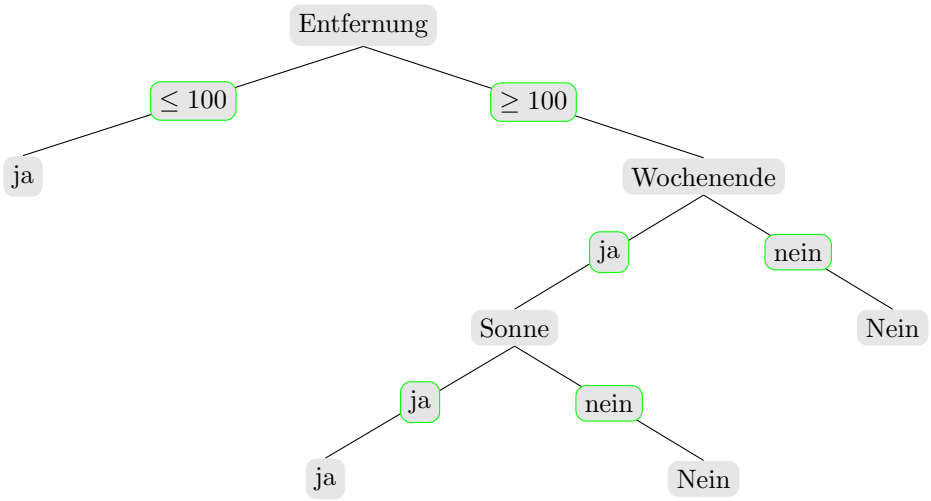



Abbildung 6.1: Entscheidungsbaum für die Klassifizierung “Fahren wir Ski?”

Da sich die Daten in Nr. 6, 7 wiedersprechen, können nicht alle Daten richtig klassifiziert werden.

Um den Entscheidungsbaum aus den Trainingsdaten aufzubauen, wird der Informationsgewinn (Informationstheoretischer Wert) der Attribute berechnet. Der Wurzelknoten unterscheidet nach dem Attribut mit dem höchsten Informationsgewinn. Auf die dadurch entstandenen Daten wird das Verfahren rekursiv angewendet, d.h. die anhand des ersten Knotens unterteilten Daten werden durch das Attribut mit dem höchsten Informationsgewinn weiter unterteilt. Das Verfahren endet, wenn es keine Attribute mehr gibt oder der verbleibende Informationsgewinn 0 ist.

Wir betrachten die Trainingsdaten als Realisierung von unabhängigen Zufallsvariablen $A_1, \ldots, A_k$ (Attribute) und einer davon abhängigen Zufallsvariable $y$ (Zielgröße).

| Nr. | Entfernung | Wochenende | Sonne | Ski |
|---|---|---|---|---|
| 1 | $\leq 100$ | j | j | j |
| 2 | $\leq 100$ | j | j | j |
| 3 | $\leq 100$ | j | n | j |
| 4 | $\leq 100$ | n | j | j |
| 5 | $> 100$ | j | j | j |
| 6 | $> 100$ | j | j | j |
| 7 | $> 100$ | j | j | n |
| 8 | $> 100$ | j | n | n |
| 9 | $> 100$ | n | j | n |
| 10 | $> 100$ | n | j | n |
| 11 | $> 100$ | n | n | n |

Tabelle 6.1: Trainingsdaten für Entscheidungsbaum in Abbildung 6.1

Eigenschaften der Entropie:

- Entropie ist maximal bei Gleichverteilung ($\log_2 n$)
- Entropie ist 0, wenn die Verteilung nur einen Wert annimmt.

#### Beispiel

Würfeln mit einem perfekten Würfel $Y \sim u\{1, \ldots, 6\}$

- $H(Y) = \log_2 6$
- $H(Y|Ygerade) = \log_2 3$
- $H(Y|Y = 6) = \log_2 1 = 0$

Der Informationsgewinn für y bei beobachteten A ist:

$$\begin{aligned} G(Y, A) &= H(Y) - EH(Y|A) \\ &= H(Y) - \sum_{a \in A(\Omega)} P(A = a) \cdot H(Y|A = a) \end{aligned}$$

Da $\underbrace{\sum P(A = a)}_{1} \cdot \underbrace{H(Y|A = a)}_{\leq H(Y)} \leq H(Y)$, folgt $G(Y, A) \geq 0$.

Da außerdem $EH(Y|A) \geq 0$, ist $G(Y|A) \leq H(Y)$. Folglich bewegt sich $G(Y|A)$ zwischen 0 und $H(Y)$

### Übung

Berechnen Sie $Y$ = Skifahren.

- $H(Y) = -(\frac{6}{11} \cdot \log_2 \frac{6}{11} + \frac{5}{11} \cdot \log_2 \frac{5}{11}) = 0,994$
- $G(Y|\text{Entfernung}) = H(Y) - EH(Y|E)$

$$\begin{aligned} H(Y|E \leq 100) &= 0 \\ H(Y|E > 100) &= -(\frac{2}{7} \cdot \log_2 \frac{2}{7} + \frac{5}{7} \cdot \log_2 \frac{5}{7}) = 0,863 \\ \curvearrowright G(Y|E) &= 0,994 - (\frac{4}{11} \cdot 0 + \frac{7}{11} \cdot 0,863) \\ &= 0,445 \end{aligned}$$

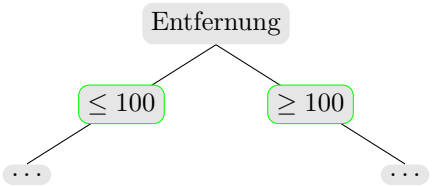
- $G(Y|\text{Wochenende}) = H(Y) - EH(Y|W)$

$$\begin{aligned} H(Y|W = \text{ja}) &= -(\frac{5}{7} \cdot \log_2 \frac{5}{7} + \frac{2}{7} \cdot \log_2 \frac{2}{7}) = 0,863 \\ H(Y|W = \text{nein}) &= -(\frac{1}{4} \cdot \log_2 \frac{1}{4} + \frac{3}{4} \cdot \log_2 \frac{3}{4}) = 0,811 \\ \curvearrowright G(Y|W) &= 0,994 - (\frac{7}{11} \cdot 0,863 + \frac{4}{11} \cdot 0,811) \\ &= 0,149 \approx 0,15 \end{aligned}$$

- $G(Y|\text{Sonne}) = H(Y) - EH(Y|S)$

$$\begin{aligned} H(Y|S = \text{ja}) &= -(\frac{5}{8} \cdot \log_2 \frac{5}{8} + \frac{3}{8} \cdot \log_2 \frac{3}{8}) = 0,954 \\ H(Y|S = \text{nein}) &= -(\frac{1}{3} \cdot \log_2 \frac{1}{3} + \frac{2}{3} \cdot \log_2 \frac{3}{3}) = 0,918 \\ \curvearrowright G(Y|S) &= 0,994 - (\frac{8}{11} \cdot 0,954 + \frac{3}{11} \cdot 0,918) \\ &= 0,049 \end{aligned}$$

Da das Attribut Entfernung den größten Informationsgewinn für die Zielgröße Y besitzt, wird die Entfernung zum Unterscheidungskriterium an der Wurzel des Entscheidungsbaums.



Da $H(Y|E \leq 100) = 0$, muss diese Datenmenge nicht weiter untereilt werden. Jedoch ist $H(Y|E \geq 100) > 0$. Für die Daten, für die $E$ 100 gilt, wird das Verfahren rekursiv fortgeführt, bis alle Attribute verwendet sind oder der verbleibende Informationsgewinn 0 ist. Es ergibt sich der Entscheidungsbaum aus Abbildung 6.1.

Vorteile von Entscheidungsbäumen:

- Kriterien des Entscheidungsbaums sind nachvollziehbar, keine Blackbox (Im Gegensatz zu neuronalen Netzen).
- Wichtigkeit der Kriterien anhand der Position im Entscheidungsbaum erkennbar. Interessant für Marktforschung und verkleinern des Entscheidungsbaums, fals er schlecht generalisiert.

Nachteile von Entscheidungsbäumen:

- Der Algorithmus kann nicht erkennen, ob ein Attribut mit hoher Entropie sinnvoll ist, z.B. Kreditkartennummer.
- Stetige Attribute müssen diskretisiert werden.

### Übung

Gegeben sei folgende Trainigsdaten von Pilzen:

| Farbe | Größe | Punkte | Essbar |
|---|---|---|---|
| rot | klein | ja | nein |
| braun | klein | nein | ja |
| braun | groß | ja | ja |
| grün | klein | nein | ja |
| rot | groß | nein | ja |

$$
\begin{aligned}
H(Y) &= -(\frac{1}{5} \cdot \log_2(\frac{1}{5}) + \frac{4}{5} \cdot \log_2(\frac{4}{5})) \\
&= 0,722
\end{aligned}
$$

$$
\begin{aligned}
G(Y,F) &= H(Y) - EH(Y|F) \\
&= 0,722 - (\frac{2}{5} \cdot 1 + \frac{2}{5} \cdot 0 + \frac{1}{5} \cdot 0) \\
&= 0,322 \\
H(Y|F = \text{rot}) &= \log_2 2 = 1 \\
H(Y|F = \text{braun}) &= \log_2 1 = 0 \\
H(Y|F = \text{grün}) &= \log_2 1 = 0
\end{aligned}
$$

$$
\begin{aligned}
G(Y,G) &= H(Y) - EH(Y|G) \\
&= 0,722 - (\frac{3}{5} \cdot 0,918 + \frac{2}{5} \cdot 0) \\
&= 0,171 \\
H(Y|G = \text{klein}) &= -(\frac{2}{3} \cdot \log_2(\frac{2}{3}) + \frac{1}{3} \cdot \log_2(\frac{1}{3})) = 0,918 \\
H(Y|G = \text{groß}) &= \log_2 1 = 0
\end{aligned}
$$

$$
\begin{aligned}
G(Y,P) &= H(Y) - EH(Y|P) \\
&= 0,722 - (\frac{2}{5} \cdot 1 + \frac{3}{5} \cdot 0) \\
&= 0,322 \\
H(Y|P = \text{ja}) &= \log_2 2 = 1 \\
H(Y|P = \text{nein}) &= \log_2 1 = 0
\end{aligned}
$$

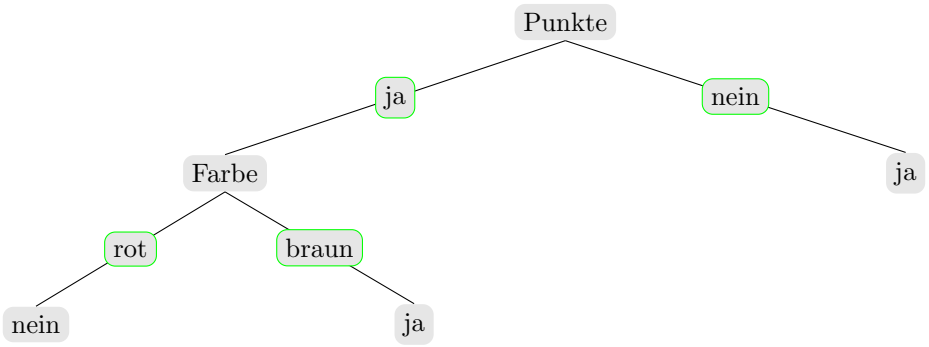
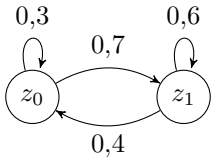



Abbildung 6.2: Entscheidungsbaum für die Klassifizierung “Pilze essbar?”

Die Größe kann vernachlässigt werden, da es für rot, Punkte nur einen Eintrag (nein) gibt und damit ist die Entropie 0.

# 7 Markov-Ketten, HMM

Eine Markov-Kette ist ein stochastischer Prozess, der durch Zustände und Übergangswahrscheinlichkeiten beschrieben wird.

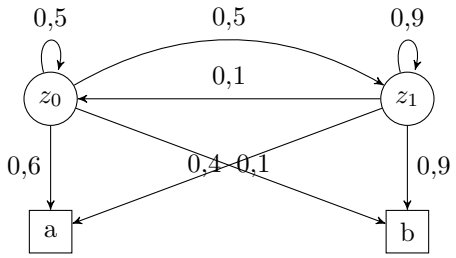



Damit lassen sich modellieren:

- Texte in natürlicher Sprache (Zustände: Buchstaben)
- DNA-Sequenzen (Zustände: A,C,G,T)
- Navigation im Internet (Zustände: Webseiten)

## 7.1 Hidden Markov Model (HMM)

**Def.:** Ein HHM ist eine Markov-Kette, die in jeden Zustand z ein Zeichen a ausgibt mit der Wahrscheinlichkeit $q_z(a)$.




Beobachtet werden nur die vom HMM ausgegebenen Zeichen. Unbekannt ist die Zustandsfolge. Mit HMM können modelliert werden z.B.:

- Codierende Regionen in DNA
  Zustände: A,C,T,G in codierenden und nicht codierenden Regionen
  Ausgegebene Zeichen: Jeweils A,C,T,G
- Nachrichtenübertragung
  Markov-Kette, die Texte modelliert
  In einem Zustand z wird das Zeichen z ausgegeben mit großer Wahrscheinlichkeit (z.B. 0,95), andere Zeichen mit geringer W'keit.
- Spracherkennung
  Vorgehen: Audio-Signal → FFT → Phoneme → Wort
  Zustände: Phoneme, die der Sprecher gesprochen hat.
  Ausgegebenen Zeichen: Phoneme, die das System erkannt hat oder das vom System erkannte Wort.

### Fragestellung

Gegeben ist eine Folge von Ausgangszeichen eines HMM, welches ist die wahrscheinlichste Folge von Zuständen, die das HMM durchlaufen hat?

### Mathematische Behandlung

Sei $Z_k$ eine Zufallsvariable, die den Zustand des HMM im Schritt $k$ angibt. Aus $Z_1$ ergibt sich die Startverteilung

$$\pi_k = P(Z_1 = k)$$

der Markov-Kette.

Da $Z_{k+1}$ nur von $Z_k$ abhängt, folgt:

$$\begin{aligned} & P(\underbrace{Z_{k+1}}_{\text{Zufallsvariable}} = \underbrace{k+1}_{\text{Zustand}} | Z_k = z_k, \ldots, Z_1 = z_1) \\ = \; & P(Z_{k+1} = z_{k+1} | Z_k = z_k) \\ =: \; & p(z_k, z_{k+1}) \end{aligned}$$

Sei $A_k$ die Zufallsvariable, die das in Schritt $k$ ausgegebene Zeichen angibt. $A_k$ hängt nur von $Z_k$ ab:

$$P(A_k = a | Z_k = z_k, \ldots, Z_1 = z_1) = P(A_k = a | Z_k = z_k) =: q_{z_k}(a)$$

Die charakteristischen Größen eines HMM sind damit $\pi_k, p(z, z'), q_z(a)$.

Für eine Folge $a_1, \ldots, a_n$ von beobachteten Zeichen suchen wir eine Folge $z_1, \ldots, z_n$ von Zuständen, so dass

$$P(Z_1 = z_1, \ldots, Z_n = z_n | A_1 = a_1, \ldots, A_n = a_n)$$

maximal ist. Da

$$\begin{aligned} & P(Z_1 = z_1, \ldots, Z_n = z_n | A_1 = a_1, \ldots A_n = a_n) \\ = & \frac{P(Z_1 = z_1, \ldots, Z_n = z_n) \cap P(A_1 = a_1, \ldots, A_n = a_n)}{P(A_1 = a_1, \ldots, A_n = a_n)} \\ = & \frac{P(Z_1 = z_1, \ldots, Z_n = z_n, A_1 = a_1, \ldots, A_n = a_n)}{P(A_1 = a_1, \ldots, A_n = a_n)} \end{aligned}$$

und der Nenner unabhängig von der Lösung ist (da er der Beobachtung entspricht), maximieren wir den Zähler.

Sei

$$\begin{aligned} t(z_n, n) &= P(Z_1 = z_1, \ldots, Z_n = z_n, A_1 = a_1, \ldots, A_n = a_n) \\ &= P(A_n = a_n | Z_1 = z_1, \ldots, Z_n = z_n, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \cdot \\ &\quad P(Z_1 = z_1, \ldots, Z_n = z_n, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \\ &= P(A_n = a_n | Z_n = z_n) \cdot \\ &\quad P(Z_1 = z_1, \ldots, Z_n = z_n, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \\ &= q_{z_n}(a_n) \cdot \\ &\quad P(Z_n = z_n | Z_1 = z_1, \ldots, Z_{n-1} = z_{n-1}, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \cdot \\ &\quad P(Z_1 = z_1, \ldots, Z_{n-1} = z_{n-1}, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \\ &= q_{z_n}(a_n) \cdot P(Z_n = z_n | Z_{n-1} = z_{n-1}) \cdot \\ &\quad P(Z_1 = z_1, \ldots, Z_{n-1} = z_{n-1}, A_1 = a_1, \ldots, A_{n-1} = a_{n-1}) \\ &= q_{z_n}(a_n) \cdot p(z_{n-1}, z_n) \cdot t(z_{n-1}, n-1) \end{aligned}$$

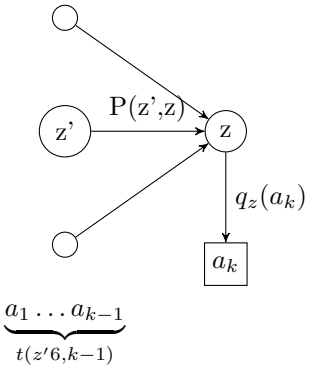
## 7.2 Viterbi-Algorithmus

Der Viterbi-Algorithmus ist ein Algorithmus der dynamischen Programmierung zur Bestimmung der wahrscheinlichsten Sequenz von verborgenen Zuständen bei einem gegebenen Hidden Markov Model (HMM) und einer beobachteten Sequenz von Symbolen. Diese Zustandssequenz wird auch als Viterbi-Pfad bezeichnet.

$$t(Z,k) = \begin{cases} q_Z(a_1) \cdot \pi_Z & \text{für } k = 1 \\ q_Z(a_k) \cdot \max\limits_{z'} \{p(z', z) \cdot t(z', k-1)\} & \text{für } k > 1 \end{cases}$$

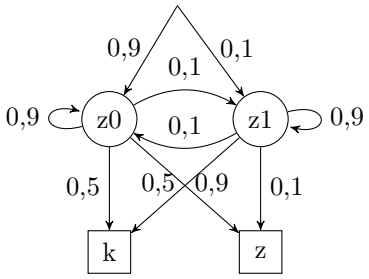
Die Laufzeut beträgt $\underbrace{\mathcal{O}(|Z| \cdot n)}_{\text{Größe der Tabelle}} \cdot \underbrace{\mathcal{O}(|Z|)}_{\text{Aufwand pro Zell}} = \mathcal{O}(|Z|^2 \cdot n)$

Um die numerische Stabilität des Verfahrens zu erhöhen, ist es Vorteilhaft, mit den Logarithmen zu rechnen.




**Beispiel**

Gegeben sei das folgende HMM-Model mit der Folge "kzzkkzkkkkkk". Berechnen Sie die wahrscheinlichste Zustandsfolge die diese Folge erzeugt hat



| | $z_0$ | $z_1$ | |
|---|---|---|---|
| $t(z', 1)$= k | $0,5 \cdot 0,9 = 0,4500;$ | $0,9 \cdot 0,1 = 0,0900$ | $z_0$ |
| $t(z', 2)$= z | $0,5 \cdot \max\{0,9 \cdot 0,45;$ $0,1 \cdot 0,09\}$ $=0,2025$ | $0,1 \cdot \max\{0,9 \cdot 0,09;$ $0,1 \cdot 0,45\}$ $=0,0081$ | $z_0$ |
| $t(z', 3)$= z | $0,5 \cdot \max\{0,9 \cdot 0,2;$ $0,1 \cdot 0,0081\})$ $=0,091125$ | $0,1 \cdot \max\{0,9 \cdot 0,008;$ $0,1 \cdot 0,2\}$ $=0,002025$ | $z_0$ |
| $t(z', 4)$= k | $0,5 \cdot \max\{0,9 \cdot 0,09;$ $0,1 \cdot 0,002\}$ $=0,04100625$ | $0,9 \cdot \max\{0,9 \cdot 0,002;$ $0,1 \cdot 0,09\}$ $=0,00820125$ | $z_0$ |
| $t(z', 5)$= k | $0,5 \cdot \max\{0,9 \cdot 0,041;$ $0,1 \cdot 0,0082\}$ $=0,0184528125$ | $0,9 \cdot \max\{0,9 \cdot 0,0082;$ $0,1 \cdot 0,041\}$ $=0,0066430125$ | $z_0$ |
| $t(z', 6)$= z | $0,5 \cdot \max\{0,9 \cdot 0,018;$ $0,1 \cdot 0,007\}$ $=0,0083037656$ | $0,1 \cdot \max\{0,9 \cdot 0,007;$ $0,1 \cdot 0,018\}$ $=0,00063$ | $z_0$ |
| $t(z', 7)$= k | $0,5 \cdot \max\{0,9 \cdot 0,0081;$ $0,1 \cdot 0,00063\}$ $=0,0037366945$ | $0,9 \cdot \max\{0,9 \cdot 0,00063;$ $0,1 \cdot 0,0083\}$ $=0,0007473389$ | $z_1$ |
| $t(z', 8)$= k | $0,5 \cdot \max\{0,9 \cdot 0,0037;$ $0,1 \cdot 0,00075\}$ $=0,0016815254$ | $0,9 \cdot \max\{0,9 \cdot 0,00075;$ $0,1 \cdot 0,0037\}$ $=0,0006053445$ | $z_1$ |
| $t(z', 9)$= k | $0,5 \cdot \max\{0,9 \cdot 0,00162;$ $0,1 \cdot 0,00059\}$ $=0,0007566806$ | $0,9 \cdot \max\{0,9 \cdot 0,0006;$ $0,1 \cdot 0,00162\}$ $=0,0004903291$ | $z_1$ |
| $t(z', 10)$= k | $0,5 \cdot \max\{0,9 \cdot 0,000729;$ $0,1 \cdot 0,0004779\}$ $=0,000328$ | $0,9 \cdot \max\{0,9 \cdot 0,0004779;$ $0,1 \cdot 0,000729\}$ $=0,0003971665$ | $z_1$ |
| $t(z', 11)$= k | $0,5 \cdot \max\{0,9 \cdot 0,000328;$ $0,1 \cdot 0,0006561\}$ $=0,0001476$ | $0,9 \cdot \max\{0,9 \cdot 0,0006561;$ $0,1 \cdot 0,000328\}$ $=0,0005314$ | $z_1$ |
| $t(z', 12)$= k | $0,5 \cdot \max\{0,9 \cdot 0,0001476;$ $0,1 \cdot 0,0005314\}$ $=0,00006642$ | $0,9 \cdot \max\{0,9 \cdot 0,0005314;$ $0,1 \cdot 0,001476\}$ $=0,0004304$ | $z_1$ |

Damit ergibt sich die Zustandsfolge $z_0 \to z_0 \to z_0 \to z_0 \to z_1 \to z_1 \to z_1 \to z_1 \to z_1 \to z_1$

## 7.3 Parameterschätzung

Die Parameter eines HMM sind Übergangs- und Emissionswahrscheinlichkeiten.

### 7.3.1 Überwachtes Lernen

Wenn für alle Trainingssequenzen die Zustandsfolge bekannt ist, lassen sich ML-Schätzer (Maximum-Likelihood) für die Übergangs- und Emissionswahrscheinlichkeiten angeben. Entsprechende Trainingssequenzen lassen sich häufig erzeugen, z.B.

- Nachrichtenübertragung: Nachricht mehrfach senden und empfangen.
- Sprachverarbeitung: Beispielsätze für die darin enthaltenen Phoneme bekannt sind, werden vorgelesen.

Seien $z, z$' Zustände des HMM und $h(z, z')$ die Häufigkeit des Übergangs von $z$ nach $z'$ in den Trainingssequenzen. Sei ferner $h_0(z)$ die Häufigkeit von $z$ als Startzustand. Da der Übergang von $z$ nach $z$' durch eine Bernoulli-verteilte Zufallsvariable beschrieben werden kann, sind

$$\hat{\pi}_z = \frac{h_0(z)}{\|Z\|} \qquad \hat{p}(z, z') = \frac{h(z, z')}{\sum_{z''} h(z, z'')}$$

Entsprechend ist

$$\hat{q}_z(a) = \frac{h_z(a)}{\sum_{a'} h_z(a')}$$

ein ML-Schätzer für $q_z(a)$, wobei $h_z(a)$ die Häufigkeit der Emission des Zeichens $a$ im Zustand $z$ in den Trainingssequenzen ist. Wenn im HMM gilt: $\sum_a q_z(a) = 1$, ist $\sum_{a'} h_z(a')$ die Summe der Längen aller Trainingssequenzen.

### 7.3.2 Unüberwachtes Lernen

#### Viterbi-Training

Wenn die Zustandsfolge nicht bekannt ist, können die unbekannten Parameter durch ein iteratives Verfahren geschätzt werden.

Idee:

- Aus den Trainingssequenzen Schätzwerte für $h_0(z), h(z, z'), h_z(a)$ berechnen.
- Mit obigen Formalen Schätzer für $\pi_z, p(z, z')$ und $q_z(a)$ berechnen

Diese Schritte werden wiederholt, bis ein Terminierungskriterium erreicht ist.

Algorithmus: Parameter $p(z, z'), \pi_z, q_z(a)$ durch überwachtes Lernen oder zufällig initialisieren.

1: **repeat**
2: Wende den Viterbi-Algorithmus auf die Trainingssequenzen an
3: Berechne $\hat{p}(z, z'), \hat{\pi}_z, \hat{q_z}(a)$
4: **until** keine Änderung an $\hat{p}(z, z'), \hat{\pi}_z, \hat{q_z}(a)$

Der Algorithmus terminiert, weil schließlich der Viterbi-Algorithmus stets die gleiche Folge liefert (ohne Beweis) und die geschätzten Parameter sich daher nicht mehr

ändern. Das Viterbi-Training liefert jedoch keinen ML-Schätzer für die unbekannten Parameter. Ein besserer Algorithmus ist der Baum-Welch-Algorithmus (auch Expectation-Maximization-Alg. genannt). Dieser findet ein lokales Maximum der Likelihood-Funktion:

$$P(A_1 = a_1, \ldots, A_n = a_n|\Theta)$$

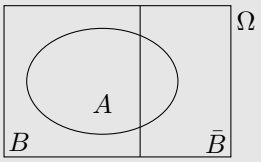
wobei $a_1, \ldots, a_n$ die beobachtete Ausgabe und $\Theta$ die Menge der zu schätzenden Parameter des HMM ist.

## 7.4 Forward-Algorithmus

**Wdh. Disjunkte Zerlegung:**

$$\begin{aligned} P(A) = P(A \cap \Omega) &= P(A \cap (B \cup \bar{B})) \\ &= P((A \cap B) \cup (A \cap \bar{B})) \\ &= P(A \cap B + A \cap \bar{B}) \\ &= P(A \cap B) + P(A \cap \bar{B}) \end{aligned}$$

$P(A) = \sum_i P(A \cap B_i)$, wenn $B_i\Omega$ partitioniert.




Für eine Beobachtung $a_1, \ldots, a_n$ eines HMM suchen wir $P(A_1 = a_1, \ldots, A_n = a_n)$. Die ist Schwierigkeit dabei das die Zustände nicht bekannt sind.

Der naiver Ansatz ist die disjunkte Zerlegung nach $Z_1, \ldots, Z_n$:

$$P(A_1 = a_1, \ldots, A_n = a_n) = \sum_{z_1,\ldots,z_n} P(A_1 = a_1, \ldots, A_n = a_n, Z_1 = z_1, \ldots, Z_n = z_n)$$

Da diese Summe aus $|Z|^n$ Summanden besteht, ist dies jedoch ineffizient.

Mit dynamischer Programmiereung berechnen wir

$$\begin{aligned}
\alpha_t(j) &= P(A_1 = a_1, \ldots, A_t = a_t, Z_t = j) \\
&= \sum_i P(A_1 = a_1, \ldots, A_t = a_t, Z_{t-1} = i, Z_t = j) \\
&= \sum_i P(A_t = a_t | A_1 = a_1, \ldots, _{t-1} = a_{t-1}, Z_{t-1} = i, Z_t = j) \\
&\quad \cdot P(A_1 = a_1, \ldots, A_{t-1} = a_{t-1}, Z_{t-1} = i, Z_t = j) \\
&= \sum_i P(A_t = a_t | Z_t = j) \cdot P(A_1 = a_1, \ldots, A_{t-1} = a_{t-1}, Z_{t-1} = i, Z_t = j) \\
&= \sum_i q_j(a_t) \cdot P(Z_t = j | Z_{t-1} = i) \cdot P(A_1 = a_1, \cdots, A_{t-1} = a_{t-1}, Z_{t-1} = i) \\
&= \sum_i q_j(a_t) \cdot p(i,j) \cdot \alpha_{t-1}(i) \qquad (7.1)
\end{aligned}$$

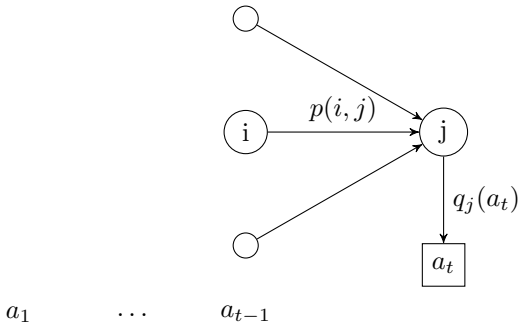



Abbildung 7.1: Skizze des Forward-Algorithmuses

Für t=1 gilt

$$\alpha_1(j) = P(A_1 = a_1, Z_1 = j) = q_j(a_1) \cdot \pi_j \qquad (7.2)$$

Ferner gilt

$$\begin{aligned}
P(A_1 = a_1, \ldots, A_n = a_n) &= \sum P(A_1 = a_1, \ldots, A_n = a_n, Z_n = j) \\
&= \sum_j \alpha_n(j) \qquad (7.3)
\end{aligned}$$

Mit (7.1), (7.2), (7.3) lässt sich die Wahrscheinlichkeit der Beobachtung $a_1, \ldots, a_n$ berechnen. Aufwand dazu $\underbrace{\mathcal{O}(n \cdot |Z|)}_{\text{Tabellengröße}} \cdot \underbrace{\mathcal{O}(|Z|)}_{\text{Aufwand pro Zelle}} + \underbrace{\mathcal{O}(|Z|)}_{\text{Aufsummierung}} = \mathcal{O}(n \cdot |Z|^2)$

## 7.5 Backward-Algorithmus

Gegeben eine Beobachtung $a_1, \ldots, a_n$, was ist der wahrscheinlichste Zustand in Schritt $t$? Gesucht ist also ein $j$ mit

$$P(Z_t = j | A_1 = a_1, \ldots, A_n = a_n)$$

maximal.

Ansatz:

$$\begin{aligned} & P(A_1 = a_1, \ldots, A_n = a_n, Z_t = j) \\ = \; & P(A_{t+1} = a_{t+1}, \ldots, A_n = a_n | A_1 = a_1, \ldots, A_t = a_t, Z_t = j) \\ & \cdot P(A_1 = a_1, \ldots, A_t = a_t, Z_t = j) \\ = \; & P(A_{t+1} = a_{t+1}, \ldots, A_n = a_n | Z_t = j) \cdot P(A_1 = a_1, \ldots, A_t = a_t, Z_t = j) \\ = \; & \beta_t(j) \cdot \alpha_t(j) \end{aligned}$$

Ferner sei $\beta_n(j) = 1$ für alle $j$. Ähnlich wie in Forward-Algorithmus folgt:

$$\beta_t(j) = \sum_i p(j,i) \cdot q_i(a_{t+1}) \cdot \beta_{t+1}(i)$$

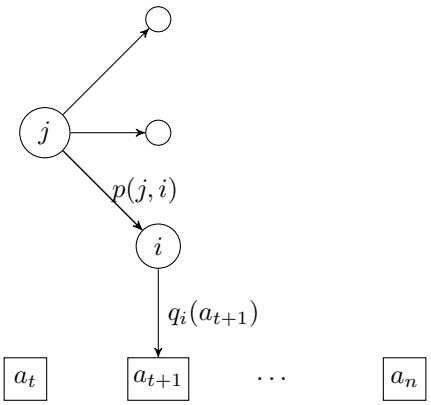



Abbildung 7.2: Skizze des Backward-Algorithmuses

Die Wahrscheinlichkeit für den Zustand $j$ im Schritt $t$ ergibt sich aus

$$P(Z_t = j | A_1 = a_1, \ldots, A_n = a_n) = \frac{\alpha_t(j) \cdot \beta_t(j)}{\sum_i \alpha_n(i)}$$

Aufwand dazu $\underbrace{\mathcal{O}(n \cdot |Z|^2)}_{\alpha\text{-Tabelle}} + \underbrace{\mathcal{O}(n \cdot |Z|^2)}_{\beta\text{-Tabelle}} + \underbrace{\mathcal{O}(|Z|)}_{\text{Summe}} + \underbrace{\mathcal{O}(1)}_{\text{Zugriff auf } \alpha_t(j) \text{und } \beta_t(j)} = \mathcal{O}(n \cdot |Z|^2)$

## 7.6 Posterior Decoding

Wenn der Viterbi-Algorithmus viele unterschiedliche Pfade mit annähernd gleicher Wahrscheinlichkeit liefert, dann lässt sich die Wahl des wahrscheinlichsten Pfades nicht gut rechtfertigen. Alternativ können wir mit dem Backward-Algorithmus die Folge der in jeden Zeitpunkt t wahrscheinlichsten Zustände

$$\hat{z}_k = \arg_j \max P(Z_t = j | A_1 = a_1, \ldots, A_n = a_n)$$

bestimmen. Diese muss jedoch keine zuverlässige Folge sein.

### Übung

Gegeben sei ein HMM. In diesem werden die Zustände verdoppelt. Wie muss sich die Länge der Trainigssequenz erhöhen, damit die gleiche Anzahl Zustandsübergänge vorhanden ist wie vorher. Vereinfacht sein angenohmen das alle Zustandsübergänge die gleiche Wahrscheinlichkeiten besitzen.

Lösung:

$$\hat{p}(z, z') = \frac{n}{|z^2|}$$

Die Trainigssequenz muss 4-fach solang sein.

## 7.7 Baum-Welch-Algorithmus

Der Baum-Welch-Algorithmus wird benutzt, um die unbekannten Parameter eines Hidden Markov Models (HMM) zu finden. Er nutzt dabei den Forward-Backward-Algorithmus zur Berechnung von Zwischenergebnissen, ist aber nicht mit diesem identisch. Der Baum-Welch-Algorithmus ist ein erwartungsmaximierender Algorithmus.

Idee:

1. HMM zufällig initialisieren
2. Mit dem Backward-Algorithmus die Wahrscheinlickeit

   $$P(Z_t = j \| A_1 = a_1, \ldots, A_n = a_n)$$

   berechnen. Auf ähnliche Weise lassen sich

   $$P(Z_t = i, Z_{t+1} = j \| A_1 = a_1, \ldots, A_n = a_n)$$

   berechnen. Damit lassen sich Erwartungswerte berechen für die Häufigkeit eines Zustandes, eines Zustandsübergangs oder einer Emission. Damit lassen sich Schätzer berechnen für die Parameter des HMM. Damit werden die Parameter des HMM geändert.

3. Mehrfach iterieren, bis sich an den Parametern nichts mehr ändert, beste Lösung ausgeben (größte Wahrscheinlichkeit für Ausgabesequenz)