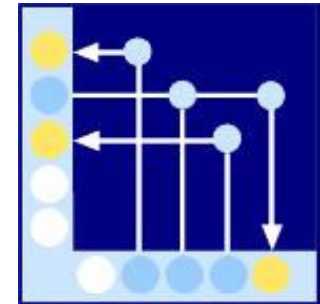




Hochschule Aalen

Fakultät Elektronik und Informatik

Studiengang Informatik



Algorithmen und Datenstrukturen 2

Vorlesung im Wintersemester 2018/2019

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Einleitung und Überblick

1.1 Vorlesungsüberblick

- ❑ Streuwerttabellen (hash tables)
 - Implementierung mit Verkettung (chaining)
 - Implementierung mit offener Adressierung (open addressing)
- ❑ Vorrangwarteschlangen (priority queues)
 - Implementierung mit binären Halden (binary heaps)
 - Implementierung mit Binomial-Halden (binomial heaps)
- ❑ Programmieretechniken
 - Nächstbest-Algorithmen (greedy algorithms)
 - Tabellengestützte Programmierung (dynamic programming)

- ❑ Elementare Graphalgorithmen
 - Breitensuche (breadth-first search)
 - Tiefensuche (depth-first search)
 - Starke Zusammenhangskomponenten (strongly connected components)

- ❑ Minimale Spannbäume (minimum spanning trees)
 - Algorithmus von Kruskal
 - Algorithmus von Prim

- ❑ Kürzeste Wege in Graphen (shortest path algorithms)
 - Algorithmus von Bellman und Ford
 - Algorithmus von Dijkstra
 - Algorithmus von Floyd und Warshall

- ❑ Anwendung mathematischer Methoden
 - Korrektheitsbeweise
 - Laufzeitabschätzungen

1.2 Organisatorisches

- Vorlesung
 - Folien sind online verfügbar
 - in der Regel kapitelweise, möglichst vor Beginn des jeweiligen Kapitels
 - bei Bedarf nachträgliche Ergänzungen oder Korrekturen
 - Beweise werden zum Teil an der Tafel entwickelt
 - Programme werden zum Teil interaktiv am Rechner entwickelt
- „Papier und Bleistift“-Übungsaufgaben zu jedem Kapitel
- 3 größere Programmieraufgaben während des Semesters als Teil der Prüfungsleistung (Anteil 1/3)
- 90-minütige Klausur im Prüfungszeitraum (Anteil 2/3)
- Voraussetzungen
 - Keine formalen Voraussetzungen
 - Aber die Inhalte folgender Vorlesungen werden inhaltlich vorausgesetzt:
 - Mathematik-Vorlesungen des 1. und 2. Semesters
 - Strukturierte und Objektorientierte Programmierung
 - Algorithmen und Datenstrukturen 1

1.3 Literaturhinweise

- ❑ T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung* (3. Auflage). R. Oldenbourg Verlag, München, 2010.

Stellenweise werden Sachverhalte in dieser Vorlesung jedoch bewusst anders dargestellt als in diesem Standardwerk.

- ❑ R. Sedgewick, K. Wayne: *Algorithmen* (4., aktualisierte Auflage). Pearson, Hallbergmoos, 2010.
- ❑ U. Schöning: *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, 2001.
- ❑ D. E. Knuth: *The Art of Computer Programming* (Band 1 bis 3). Addison-Wesley, Reading, MA, 2001.
- ❑ Wikipedia, die freie Enzyklopädie: <https://de.wikipedia.org>
Wikipedia, the free encyclopedia: <https://en.wikipedia.org>
Wikipedia ist häufig eine nützliche Informationsquelle.
Stellenweise sind die Artikel jedoch ungenau oder widersprüchlich, z. B. beim Thema Graphen.

2 Streuwerttabellen (hash tables)

2.1 Einleitung und Motivation

2.1.1 Aufgabe

- Schreiben Sie ein Programm (z. B. in Java),
 - das als Eingabe eine Folge von Wörtern erhält (z. B. ein Wort pro Zeile)
 - und als Ausgabe die Häufigkeit jedes Worts liefert.
- Beispieleringabe (links) und zugehörige Ausgabe (rechts, die Reihenfolge der Ausgabezeilen ist beliebig):

```
eins          3 eins
zwei         2 zwei
drei         1 drei
zwei
eins
eins
```

2.1.2 Lösung mit verketteter Liste

```
import java.io.*;

// Listenelement.
class Elem {
    String word;           // Wort.
    int count;            // Häufigkeit.
    Elem next;           // Verkettung.

    Elem (String w, Elem n) {
        word = w; count = 1; next = n;
    }
}

// Hauptprogramm.
class WordCount {
    public static void main (String [] args) throws IOException {
        BufferedReader r = // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
        Elem h = null;     // Listenanfang (head).
        String w;         // Aktuelles Wort.
```

```
// Eingabe zeilenweise lesen und verarbeiten.
input: // readLine kann IOException werfen.
while ((w = r.readLine()) != null) {
    // Element e mit Wort w suchen.
    for (Elem e = h; e != null; e = e.next) {
        // Wenn vorhanden, Zähler erhöhen.
        if (e.word.equals(w)) {
            e.count++;
            continue input;
        }
    }
    // Andernfalls neues Element mit Zähler 1 erzeugen.
    h = new Elem(w, h);
}

// Ausgabe produzieren.
for (Elem e = h; e != null; e = e.next) {
    System.out.println(e.count + " " + e.word);
}
}
```

❑ Problem: Linearer Aufwand für jede Suche

2.1.3 Lösung mit Suchbaum

- ❑ Zum Beispiel Rot-schwarz-Baum
- ❑ Siehe „Algorithmen und Datenstrukturen 1“
- ❑ Probleme:
 - Immer noch logarithmischer Aufwand für jede Suche
 - Jeder Baumknoten braucht zusätzlich Platz für Zeiger und Farbinformation

2.1.4 Lösung mit Streuwerttabelle

```
import java.io.*;

// Tabellenelement.
class Elem {
    String word;           // Wort.
    int count;            // Häufigkeit.

    Elem (String w) {
        word = w; count = 1;
    }
}

// Hauptprogramm.
class WordCount {
    public static void main (String [] args) throws IOException {
        BufferedReader r = // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
        Elem [] tab = new Elem [1000]; // Tabelle.
        String w;           // Aktuelles Wort.
```

```
// Eingabe zeilenweise lesen und verarbeiten.  
// readLine kann IOException werfen.  
while ((w = r.readLine()) != null) {  
    // Streuwert des Worts als Index in die Tabelle verwenden.  
    int i = w.hashCode() % tab.length;  
    if (i < 0) i = -i;  
    Elem e = tab[i];  
    if (e != null) {  
        // Wenn dort bereits ein Element ist, Zähler erhöhen.  
        e.count++;  
    }  
    else {  
        // Andernfalls neues Element mit Zähler 1 erzeugen.  
        tab[i] = new Elem(w);  
    }  
}  
  
// Ausgabe produzieren.  
for (Elem e : tab) if (e != null) {  
    System.out.println(e.count + " " + e.word);  
}  
}
```

Diskussion

☐ Vorteile

- Konstanter Aufwand für jede „Suche“ → maximal effizient
- Kompakte Speicherung der Tabelle als Feld (array)

☐ Nachteil

- Tabelle kann nicht wachsen (müsste bei Bedarf umkopiert werden)

☐ Problem

- Verschiedene Wörter können „zufällig“ den gleichen Streuwert besitzen (konkret z. B. Aa und BB).

☐ Fragen

- Wie löst man dieses Problem?
- Wie berechnet man `hashCode` sinnvoll?

2.1.5 Lösung mit `java.util.HashMap`

```
import java.io.*;
import java.util.*;

// Hauptprogramm.
class WordCount {
    public static void main (String [] args) throws IOException {
        BufferedReader r = // Standardeingabe als BufferedReader.
            new BufferedReader(new InputStreamReader(System.in));
        Map<String, Integer> tab = // Tabelle.
            new HashMap<String, Integer>(1000);
        String w; // Aktuelles Wort.

        // Eingabe zeilenweise lesen und verarbeiten.
        // readLine kann IOException werfen.
        while ((w = r.readLine()) != null) {
            // Entweder einen neuen Eintrag mit Zähler 1 einfügen
            // oder den Zähler des vorhandenen Eintrags um 1 erhöhen.
            Integer c = tab.get(w);
            if (c == null) c = 0;
            tab.put(w, c + 1);
        }
    }
}
```

```
// Ausgabe produzieren.  
for (Map.Entry<String, Integer> e : tab.entrySet()) {  
    System.out.println(e.getValue() + " " + e.getKey());  
}  
}  
}
```

2.1.6 Lösung in C++11 mit `std::unordered_map`

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// Hauptprogramm.
int main () {
    unordered_map<string, int> tab (1000); // Tabelle.
    string w;                          // Aktuelles Wort.

    // Eingabe zeilenweise lesen und verarbeiten.
    while (getline(cin, w)) {
        // tab[w] liefert den Zähler zu Wort w mit Anfangswert 0.
        tab[w]++;
    }

    // Ausgabe produzieren.
    for (pair<string, int> e : tab) {
        cout << e.second << " " << e.first << endl;
    }
}
```

2.1.7 Lösung in der Skriptsprache AWK

```
# Eingabe zeilenweise lesen und verarbeiten.  
{ tab[$0]++ }  
  
# Ausgabe produzieren.  
END { for (w in tab) print tab[w], w }
```

Erläuterungen

- Der Block `{ }` wird automatisch für jede Eingabezeile ausgeführt. Der Inhalt der Zeile ist jeweils als `$0` verfügbar.
- Das Feld `tab` ist in Wirklichkeit ein assoziativer Container analog zu `java.util.Map` und `std::unordered_map`, der intern (vermutlich) als Streuwerttabelle implementiert ist.
- Der Block `END { }` wird automatisch nach Verarbeitung der Eingabe ausgeführt.
- Die Schleife `for (w in tab)` durchläuft alle Schlüsselwerte der Tabelle `tab`.

2.1.8 Allgemeines Prinzip

- ❑ Die in einer Streuwerttabelle gespeicherten Objekte bestehen normalerweise aus einem *Schlüssel* (im Beispiel ein Wort) und einem zugehörigen *Wert* (im Beispiel der zugehörige Zähler).
- ❑ Schlüssel und Wert können selbst wiederum aus mehreren Teilen bestehen. In einer Tabelle mit Personendaten könnte der Schlüssel eines Objekts z. B. aus Vor- und Nachname einer Person bestehen, während der Wert aus beliebig vielen weiteren Daten bestehen kann (z. B. Geburtsdatum, Adresse etc.).
- ❑ Die Werte können aber auch fehlen, z. B. wenn das Beispielprogramm nur die Menge aller verschiedenen Wörter ohne ihre Häufigkeit ausgeben soll. (Dann bezeichnet man die Streuwerttabelle eher als *Streuwertmenge*.)
- ❑ Für die *Gleichheit* zweier Objekte sind in jedem Fall nur ihre Schlüssel relevant, ebenso für die Berechnung ihres Streuwerts.
- ❑ Gleiche Objekte müssen immer den gleichen Streuwert besitzen.
- ❑ Umgekehrt kann es aber durchaus auch verschiedene Objekte mit dem gleichen Streuwert geben.

2.1.9 Grundoperationen auf Streuwerttabellen

- ❑ Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v
 - 1 Wenn die Tabelle bereits ein Objekt (k, v') mit irgendeinem Wert v' enthält, wird es durch (k, v) ersetzt.
 - 2 Andernfalls wird das Objekt (k, v) zur Tabelle hinzugefügt, sofern diese noch nicht voll ist.

- ❑ Suchen eines Schlüssels k
 - 1 Wenn die Tabelle ein Objekt (k, v) mit irgendeinem Wert v enthält, wird der Wert v zurückgeliefert.
 - 2 Andernfalls wird das Nichtvorhandensein eines solchen Objekts durch einen geeigneten Resultatwert \perp (z. B. `null` in Java) angezeigt.

- ❑ Löschen eines Schlüssels k
 - 1 Wenn die Tabelle ein Objekt (k, v) mit irgendeinem Wert v enthält, wird es aus der Tabelle entfernt.
 - 2 Andernfalls ist die Operation wirkungslos.

2.1.10 Weitere Anwendungsbeispiele für Streuwerttabellen

- ❑ Online-Katalog → Bestellnummer als Schlüssel
- ❑ Personaldatenbank → Personalnummer als Schlüssel
- ❑ Kfz-Datenbank → Kennzeichen als Schlüssel
- ❑ Cache eines Webbrowsers → URL als Schlüssel
- ❑ Variablentabelle eines Compilers → Variablenname als Schlüssel
- ❑ Große, dünn besetzte Matrix → Paar von Zeilen- und Spaltenindex als Schlüssel

2.2 Streuwertfunktionen

2.2.1 Grundregeln

- ❑ Eine *Streuwertfunktion* ordnet jedem Objekt (aus einer bestimmten Grundmenge) bzw. jedem Schlüssel eine ganze Zahl zu.
- ❑ Inhaltlich gleiche Objekte *müssen* den gleichen Streuwert besitzen (vgl. § 2.1.8). (Das heißt, wenn man in einer Java-Klasse `equals` überschreibt, muss man i. d. R. auch `hashCode` überschreiben.)
- ❑ Inhaltlich verschiedene Objekte *sollten* möglichst verschiedene Streuwerte besitzen.
- ❑ Letzteres ist aber häufig nicht möglich, weil es z. B. viel mehr (prinzipiell unendlich viele) inhaltlich verschiedene `String`-Objekte als `int`-Werte gibt.

2.2.2 Beispiel 1: Punkte

```
// Punkt im zweidimensionalen Raum.
class Point {
    // Koordinaten des Punkts.
    public final double x, y;

    // Punkt mit Koordinaten x und y konstruieren.
    public Point (double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Zeichenketten-Darstellung des aktuellen Objekts liefern.
    public String toString () {
        return "(" + x + ", " + y + ")";
    }
}
```

```
// Vergleich des aktuellen Objekts this (mit Typ Point)
// mit irgendeinem anderen Objekt other (mit beliebigem
// dynamischem Typ).
public boolean equals (Object other) {
    // 1. Wenn other kein Point ist, kann es nicht gleich this sein.
    if (!(other instanceof Point)) return false;

    // 2. Andernfalls kann other in Point that umgewandelt werden.
    Point that = (Point)other;

    // 3. Dann können this und that inhaltlich verglichen werden.
    return this.x == that.x && this.y == that.y;
}

// Streuwert für das aktuelle Objekt liefern.
public int hashCode () {
    // Für Punkte, die gemäß equals gleich sind,
    // erhält man den gleichen Streuwert.
    // return (int)(x + y);
    // Oder besser:
    return (int)x << 16 | (int)y;
}
}
```

2.2.3 Beispiel 2: Zeichenketten

```
// (Gedachter) Ausschnitt aus der Bibliotheksklasse java.lang.String
public class String {
    private int n;           // Länge der Zeichenkette.
    private char [] s;      // Feld mit den einzelnen Zeichen.

    // Streuwert der Zeichenkette berechnen.
    public int hashCode () {
        // Die einzelnen Zeichen werden quasi als Ziffern einer
        // Zahl h mit Basis 31 interpretiert (obwohl es natürlich
        // mehr als 31 verschiedene Zeichen gibt).
        // Durch arithmetischen Überlauf, der in Java wohldefiniert
        // ist, können auch negative Werte entstehen.
        int h = 0;
        for (int i = 0; i < n; i++) {
            h = h * 31 + s[i];
        }
        return h;
    }

    .....
}
```

2.3 Einschränkung des Wertebereichs einer Streuwertfunktion

2.3.1 Grundsätzlich

- Damit ein Streuwert h als Index i in eine Tabelle (array) der Größe N verwendet werden kann, muss er noch auf den Wertebereich von 0 einschließlich bis N ausschließlich eingeschränkt werden.

2.3.2 Divisionsrestmethode

❑ Mathematisch:

$$i = h \bmod N$$

❑ Problem in Java, C und vielen anderen Programmiersprachen:

- $h \% N$ stimmt nur für $h \geq 0$ (und $N > 0$) garantiert mit der mathematischen Definition von $h \bmod N$ überein.
- Beispielsweise ist $-14 \% 3$ in Java gleich -2 (statt 1) und liegt damit nicht im verlangten Wertebereich.

❑ Mögliche Lösungen:

$$i = (h \geq 0 ? h : -h) \% N$$

$$i = (h \% N + N) \% N$$

❑ Erfahrungsgemäß hängt die Güte der Divisionsrestmethode stark vom Wert N ab:

- Wenn N eine Zweierpotenz 2^p ist, besteht $h \bmod N$ aus den niedrigsten p Bits des Werts h , d. h. die übrigen Bits werden ignoriert, was zu schlechter Streuung führen kann.
- Gut geeignet sind meist Primzahlen N , die nicht zu nahe an einer Zweierpotenz liegen.

2.3.3 Multiplikationsmethode

□ Mathematisch:

- Gegeben sei eine beliebige Konstante $A \in (0, 1)$,
z. B. $A = \frac{\sqrt{5} - 1}{2} = 0.61803399\dots$ (vgl. Goldener Schnitt).
- Berechne $u = h \cdot A$, $v = u \bmod 1 = u - \lfloor u \rfloor \in [0, 1)$
und $i = \lfloor v \cdot N \rfloor \in \{0, \dots, N - 1\}$,
d. h. multipliziere die Nachkommastellen von $h \cdot A$ mit N .
- Erfahrungsgemäß funktioniert diese Methode mit den meisten Werten von N und A gut.
- Laut Knuth liefert die obige Wahl von A aber besonders gute Ergebnisse.
- Für $A = \frac{1}{N}$ erhält man gerade die Divisionsrestmethode.

□ Praktische Berechnung mit Ganzzahlarithmetik:

- Die Tabellengröße N muss hierfür eine Zweierpotenz 2^p sein.
- Die Wortlänge des Computers sei w , d. h. ganze Zahlen bestehen aus w Bits.
- Gegeben sei eine beliebige ganzzahlige Konstante $A' \in (0, 2^w)$ anstelle von $A \in (0, 1)$, aus der $A = \frac{A'}{2^w}$ berechnet werden könnte.
- Berechne ganzzahlig $u' = h \cdot A'$, $v' = u' \bmod 2^w$, $i' = v' \cdot N = v' \cdot 2^p$ und $i = \frac{i'}{2^w} = \frac{v' \cdot 2^p}{2^w} = \frac{v'}{2^{w-p}} \in \{0, \dots, N - 1\}$.
- Damit sind A' , u' , v' und i' jeweils um den Faktor 2^w größer als A bzw. u bzw. v bzw. i .
- $h \cdot A'$ ist prinzipiell eine Zahl mit $2w$ Bits, von der die oberen w Bits bei der praktischen Berechnung mit Wortlänge w jedoch verlorengelassen werden, sodass das Ergebnis dieser Berechnung tatsächlich bereits v' entspricht.
- Die abschließende Division durch die Zweierpotenz 2^{w-p} kann effizient durch eine Bitverschiebung um $w - p$ Positionen nach rechts ausgeführt werden.
- Damit verwendet man faktisch die obersten p Bits des Produkts $h \cdot A'$ als Index.
- Konkret z. B. in Java mit Wortlänge $w = 32$ und $s = A'$:

$$i = h * s >>> 32 - p = (h * s) >>> (32 - p)$$

□ Beispiel

○ Sei $N = 2^{10} = 1024$ und $A = \frac{\sqrt{5} - 1}{2} = 0.61803399\dots$

○ Dann ergibt sich für den Streuwert $h = 15341$ mit Gleitkomma-Arithmetik:

$$u = h \cdot A = 9481.25942141\dots$$

$$v = u \bmod 1 = 0.25942141\dots$$

$$i = \lfloor v \cdot N \rfloor = 265$$

○ Mit 16-Bit-Ganzzahlarithmetik und $A' = A \cdot 2^{16} \approx 40503$ ergibt sich:

$$u' = h \cdot A' = 621356523 = 0010010100001001 \ 0010010111101011_2$$

$$v' = u' \bmod 2^{16} = 9707 = 0010010111101011_2$$

$$i = \frac{v'}{2^{16-10}} = 151 = 0010010111_2 \text{ (die obersten 10 Bit von } v')$$

○ Aufgrund von Rundungsfehlern ergeben sich also unterschiedliche Ergebnisse, was für die Praxis jedoch kein Problem darstellt.

2.4 Behandlung von Kollisionen

- ❑ Wenn zwei verschiedene Objekte bzw. Schlüssel (nach der Einschränkung des Wertebereichs) den gleichen Streuwert bzw. Index besitzen, spricht man von einer *Kollision*.

- ❑ Zur Auflösung solcher Kollisionen gibt es prinzipiell zwei Möglichkeiten:
 - *Verkettung*:
Alle Objekte mit dem gleichen Streuwert bzw. Index werden in einer verketteten Liste am gleichen Platz der Tabelle gespeichert.

 - *Offene Adressierung*:
Wenn der Platz, in dem ein Objekt eigentlich gespeichert werden sollte, bereits belegt ist, wird nach irgendeiner geeigneten Methode ein noch freier „Ersatzplatz“ gesucht.

- ❑ Beide Verfahren mit ihren unterschiedlichen Vor- und Nachteilen werden im folgenden genauer betrachtet.

2.5 Verkettung (chaining)

2.5.1 Grundoperationen

- Gegeben sei ein Feld tab der Größe N zur Speicherung verketteter Listen sowie eine Funktion h , die jedem Schlüssel k einen Index $i \in \{0, \dots, N - 1\}$ zuordnet.
(Das heißt, h ist eine Streuwertfunktion mit Einschränkung des Wertebereichs.)
- Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v
 - 1 Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k, v') mit irgendeinem Wert v' .
 - 2 Wenn es ein solches Objekt gibt, ersetze es durch das Objekt (k, v) .
 - 3 Andernfalls füge das Objekt (k, v) am Anfang der Liste ein.
(Prinzipiell könnte das Objekt irgendwo in die Liste eingefügt werden, aber am Anfang geht es am einfachsten und schnellsten.)
- Suchen eines Schlüssels k
 - 1 Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k, v) mit irgendeinem Wert v .
 - 2 Wenn es ein solches Objekt gibt, liefere den Wert v zurück.
 - 3 Andernfalls liefere \perp .

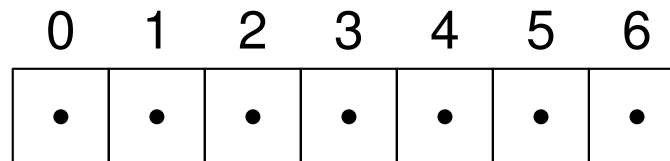
□ Löschen eines Schlüssels k

- 1 Berechne den Index $i = h(k)$ und durchsuche die Liste $tab[i]$ nach einem Objekt (k, v) mit irgendeinem Wert v .
- 2 Wenn es ein solches Objekt gibt, entferne es aus der Liste.

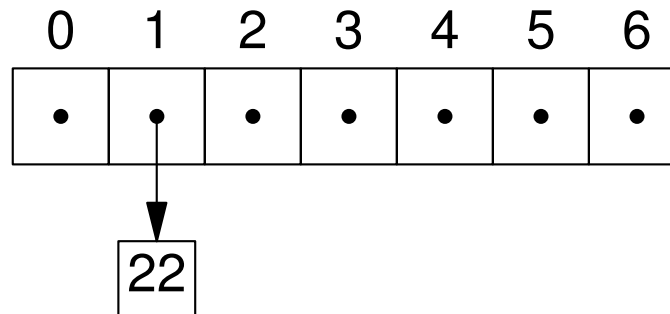
2.5.2 Beispiel

□ Sei $N = 7$ und $h(k) = k \bmod 7$ (d. h. die Schlüssel sind bereits ganze Zahlen).

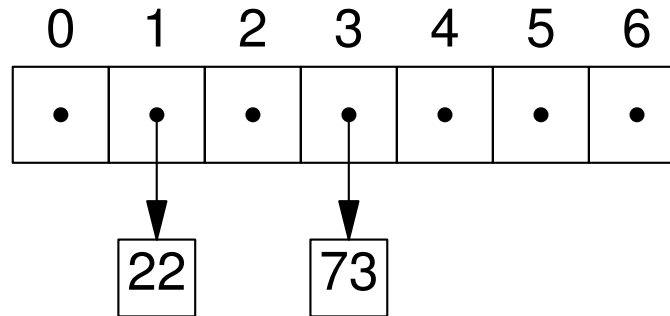
□ Leere Tabelle:



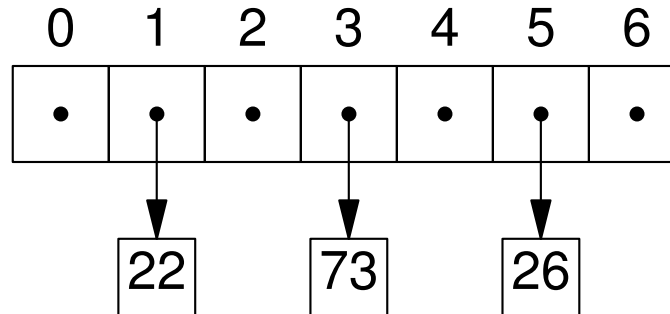
□ Einfügen von $k = 22$ mit $h(k) = 22 \bmod 7 = 1$:



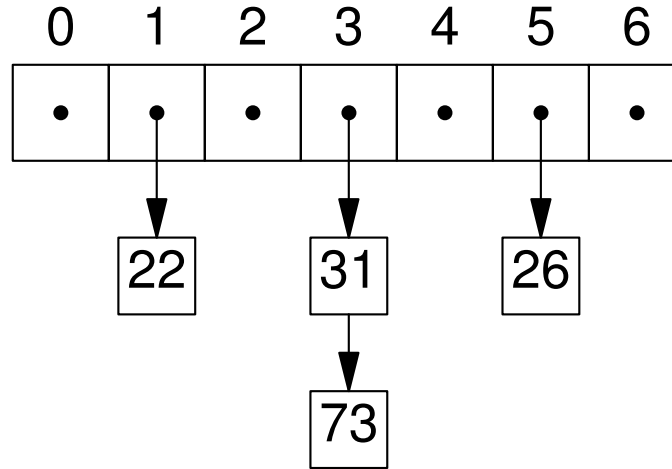
□ Einfügen von $k = 73$ mit $h(k) = 73 \bmod 7 = 3$:



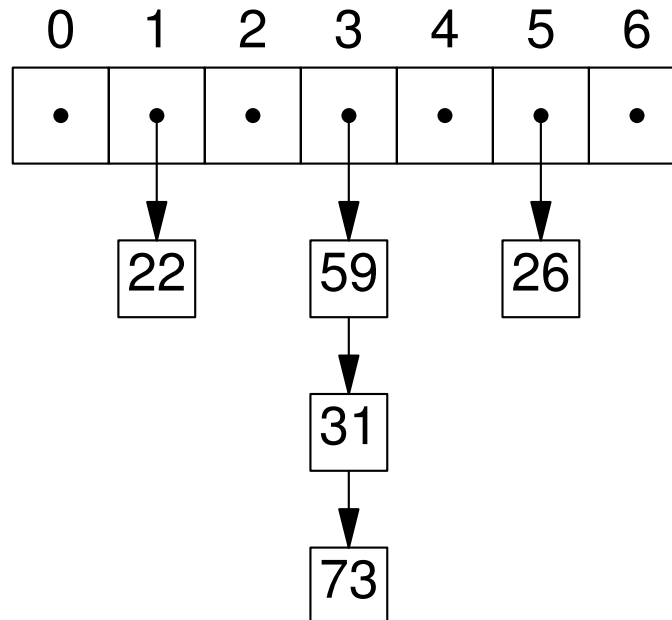
□ Einfügen von $k = 26$ mit $h(k) = 26 \bmod 7 = 5$:



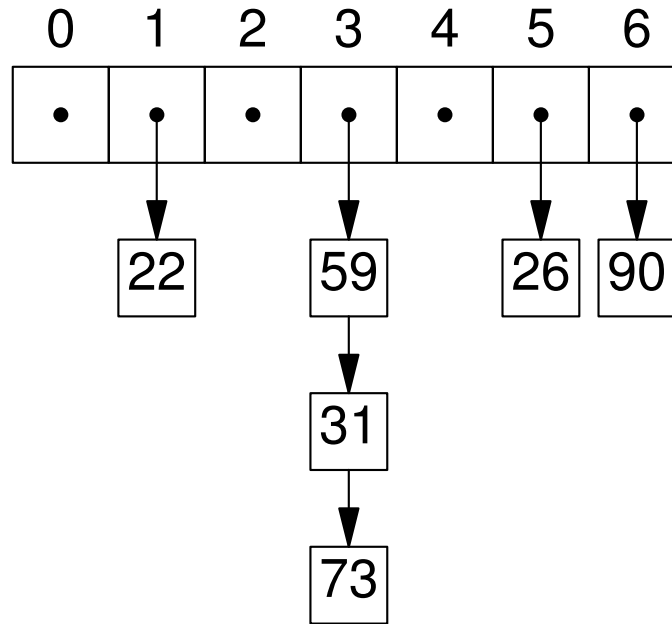
□ Einfügen von $k = 31$ mit $h(k) = 31 \bmod 7 = 3$:



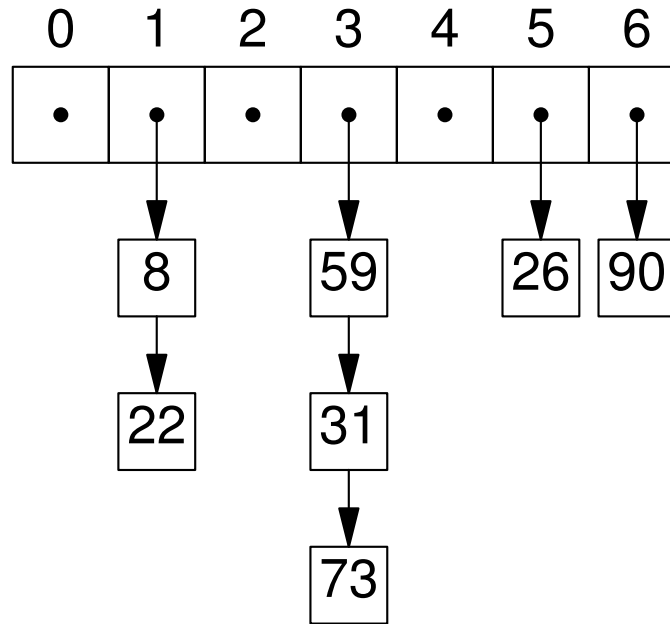
□ Einfügen von $k = 59$ mit $h(k) = 59 \bmod 7 = 3$:



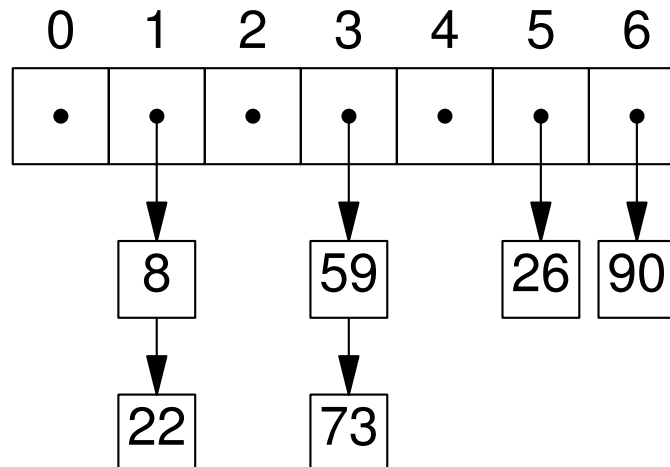
□ Einfügen von $k = 90$ mit $h(k) = 90 \bmod 7 = 6$:



□ Einfügen von $k = 8$ mit $h(k) = 8 \bmod 7 = 1$:



□ Löschen von $k = 31$ mit $h(k) = 31 \bmod 7 = 3$:



2.5.3 Laufzeitanalyse

- ❑ Betrachte eine Tabelle der Größe N , die momentan m Objekte enthält, d. h. ihr *Belegungsfaktor* oder *Füllgrad* ist $\alpha = \frac{m}{N}$.
- ❑ α ist gleichzeitig die durchschnittliche Anzahl von Objekten pro Platz, d. h. die durchschnittliche Länge der gespeicherten Listen.
- ❑ Als Maß für die *Laufzeit* einer Operation (Einfügen/Ersetzen eines Objekts, Suchen und Löschen eines Schlüssels) wird die Anzahl der erforderlichen Schlüsselvergleiche verwendet.

Schlimmster Fall

- ❑ Alle Schlüssel besitzen den gleichen Streuwert/Index i .
 - $tab[i]$ enthält eine Liste mit m Objekten, die übrigen Plätze sind leer.
 - Bei jeder Operation muss die Liste $tab[i]$ u. U. vollständig durchlaufen werden
 - Laufzeit jeweils $O(m)$

Idealfall

- ❑ Eine Streuwertfunktion *streut ideal*, wenn jeder Index $0, \dots, N - 1$ mit der gleichen Wahrscheinlichkeit $\frac{1}{N}$ als Funktionswert auftritt.

(Dies wird auch als Simple-uniform-hashing-Annahme bezeichnet.)

- ❑ Anmerkungen:

- Trotzdem ist die Wahrscheinlichkeit, dass die Listen aller Plätze exakt gleich lang sind, äußerst gering.

(Vergleich: Wenn man sehr oft mit einem idealen Würfel würfelt, ist die Wahrscheinlichkeit, dass jede Augenzahl exakt gleich oft auftritt, auch sehr gering.)

- Und die Wahrscheinlichkeit, dass in einer Menge von Objekten mindestens zwei den gleichen Streuwert/Index besitzen, ist relativ hoch, auch wenn m deutlich kleiner als N ist.

Vgl. Geburtstagsparadoxon: Befinden sich in einem Raum mindestens 23 Personen, dann ist die Chance, dass zwei oder mehr dieser Personen am gleichen Tag (ohne Beachtung des Jahrganges) Geburtstag haben, größer als 50%. (Wikipedia)

- ❑ Bei den folgenden Analysen wird eine ideal streuende Funktion vorausgesetzt.

Erfolglose Suche

Behauptung:

- ❑ Bei einer Suche nach einem Schlüssel k , der nicht in der Tabelle enthalten ist, werden im Durchschnitt α Schlüsselvergleiche ausgeführt.

Anmerkung:

- ❑ „Im Durchschnitt“ bedeutet:
Wenn diese Operation für sehr viele Schlüssel k und sehr viele Tabellen mit unterschiedlicher Verteilung der Objekte ausgeführt wird, dann werden im Mittel α Schlüsselvergleiche pro Operation ausgeführt.

Beweis:

- ❑ Sei $i = h(k)$, wobei jeder Index i gleich wahrscheinlich ist.
- ❑ Die Länge der Liste $tab[i]$ ist im Durchschnitt α .
- ❑ Da die Liste kein Objekt mit Schlüssel k enthält, muss sie komplett durchsucht werden.
- ❑ Dabei werden im Durchschnitt α Schlüsselvergleiche ausgeführt.

Erfolgreiche Suche

Behauptung:

- Bei einer Suche nach einem Schlüssel k , der in der Tabelle enthalten ist, werden im Durchschnitt $1 + \frac{\alpha}{2} - \frac{1}{2N} \approx 1 + \frac{\alpha}{2}$ Schlüsselvergleiche ausgeführt.

Überprüfung der Behauptung z. B. für $m = 1$ und $m = 2$

Beweis:

- Sei $i = h(k)$.
- Um das Objekt x mit Schlüssel k zu finden, müssen alle Objekte überprüft werden, die sich in der Liste $tab[i]$ vor diesem Objekt befinden, sowie dieses Objekt selbst.
- Also hängt die Laufzeit von der Anzahl der Objekte ab, die später als x in die Tabelle bzw. in diese Liste eingefügt wurden (weil neue Objekte gemäß § 2.5.1 immer am Anfang einer Liste eingefügt werden).
- Für $j = 1, \dots, m$ sei x_j das j -te Objekt, das in die Tabelle eingefügt wurde.

- Die Anzahl der Objekte, die später als x_j in die Tabelle eingefügt wurden, beträgt $m - j$.
- Die Wahrscheinlichkeit, dass eines dieser Objekte den gleichen Streuwert besitzt wie x_j , beträgt gemäß Simple-uniform-hashing-Annahme $\frac{1}{N}$.
- Daher ist die durchschnittliche Anzahl von Objekten, die später als x_j in die gleiche Liste eingefügt wurden, gleich $\frac{m - j}{N}$.
- Somit ist die durchschnittliche Laufzeit einer Suche nach dem Schlüssel von x_j gleich $1 + \frac{m - j}{N}$ ($j = 1, \dots, m$).
- Die durchschnittliche Laufzeit einer Suche nach dem Schlüssel irgendeines dieser Objekte x erhält man als Durchschnitt dieser Werte über alle j :

$$\frac{1}{m} \sum_{j=1}^m \left(1 + \frac{m - j}{N} \right) = \frac{1}{m} \sum_{j=1}^m 1 + \frac{1}{m} \sum_{j=1}^m \frac{m - j}{N} = 1 + \frac{m}{N} - \frac{1}{mN} \sum_{j=1}^m j = 1 + \frac{m}{N} - \frac{1}{mN} \frac{m(m+1)}{2} =$$

$$1 + \frac{2m}{2N} - \frac{m+1}{2N} = 1 + \frac{m-1}{2N} = 1 + \frac{m}{2N} - \frac{1}{2N} = 1 + \frac{\alpha}{2} - \frac{1}{2N}$$

Einfügen und Ersetzen

- ❑ Beim Einfügen eines neuen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.
- ❑ Beim Ersetzen eines bereits vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.

Erfolgreiches und erfolgloses Löschen

- ❑ Beim Löschen eines vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.
- ❑ Beim Löschen eines nicht vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.

2.6 Offene Adressierung (open addressing)

2.6.1 Grundoperationen

- Gegeben sei ein Feld tab der Größe N zur Speicherung von Objekten sowie eine *Sondierungsfunktion* s , die jedem Schlüssel k eine Permutation $(s_0(k), \dots, s_{N-1}(k))$ der Indexwerte $0, \dots, N - 1$ zuordnet.
- Hilfsoperation: Suchen des Platzes eines Schlüssels k
 - 1 Für $j = 0, \dots, N - 1$:
 - 1 Berechne den Index $i = s_j(k)$.
 - 2 Wenn $tab[i]$ leer ist, liefere „nicht vorhanden“ und entweder den gemerkten Index (falls es bereits einen gibt) oder (andernfalls) den Index i zurück.
 - 3 Wenn $tab[i]$ eine Löschmarkierung (siehe unten) enthält und bis jetzt noch kein Index gemerkt wurde, merke den Index i .
 - 4 Wenn $tab[i]$ ein Objekt (k, v) mit irgendeinem Wert v enthält, liefere „vorhanden“ und den Index i zurück.
 - 2 Wenn während der Schleife ein Index gemerkt wurde, liefere „nicht vorhanden“ und diesen Index zurück.
 - 3 Andernfalls liefere „Tabelle voll“ zurück.

- ❑ Einfügen/Ersetzen eines Objekts (k, v) mit Schlüssel k und Wert v
 - 1 Führe die obige Hilfsoperation aus.
 - 2 Wenn sie einen Index i zurückliefert, speichere das Objekt (k, v) in $tab[i]$.
 - 3 Andernfalls signalisiere einen Fehler (Tabelle voll).

- ❑ Suchen eines Schlüssels k
 - 1 Führe die obige Hilfsoperation aus.
 - 2 Wenn sie „vorhanden“ und einen Index i zurückliefert, liefere den Wert v des in $tab[i]$ gespeicherten Objekts (k, v) zurück.
 - 3 Andernfalls liefere \perp .

- ❑ Löschen eines Schlüssels k
 - 1 Führe die obige Hilfsoperation aus.
 - 2 Wenn sie „vorhanden“ und einen Index i zurückliefert, schreibe eine Löschmarkierung in $tab[i]$.

2.6.2 Anmerkungen

- ❑ Anders als bei Verkettung, wo eine Tabelle prinzipiell beliebig viele Objekte aufnehmen kann, ist die Kapazität bei offener Adressierung durch die Tabellengröße N beschränkt.
- ❑ Daraus folgt, dass der Belegungsfaktor bei offener Adressierung nicht größer als 1 sein kann.
- ❑ Je voller die Tabelle ist, desto mehr Sondierungsschritte sind im Durchschnitt nötig, um noch einen freien Platz zur Speicherung eines Objekts zu finden.
- ❑ Löschmarkierungen werden gebraucht, damit ein Platz beim Einfügen wie ein leerer Platz, beim Suchen aber wie ein belegter Platz behandelt wird.

2.6.3 Lineare Sondierung (linear probing)

- ❑ Gegeben sei eine Tabelle der Größe N sowie eine Streuwertfunktion h .
- ❑ Definiere $s_j(k) = (h(k) + j) \bmod N$ für $j = 0, \dots, N - 1$,
d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt mit seinem Streuwert $h(k)$ (eingeschränkt auf den Wertebereich $\{0, \dots, N - 1\}$)
und durchläuft dann der Reihe nach alle weiteren Plätze der Tabelle.
- ❑ Problem der direkten Verstopfung (primary clustering problem):
 - Wenn zwei oder mehr Objekte den gleichen Streuwert i besitzen (was auch bei idealer Verteilung der Streuwerte relativ häufig auftritt, vgl. § 2.5.3), entsteht an dieser Stelle der Tabelle ein „Klumpen“ (cluster) von Objekten.
 - Jedes weitere Objekt, dessen Streuwert in diesem Klumpen liegt, vergrößert diesen, selbst wenn sein Streuwert verschieden von i ist.
 - Je größer ein derartiger Klumpen ist, desto höher ist die Laufzeit aller Operationen für die betroffenen Objekte.
- ❑ Beispiel: $N = 16$, $h(k) = k$
Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6

2.6.4 Quadratische Sondierung (quadratic probing)

□ Gegeben sei eine Tabelle der Größe N sowie eine Streuwertfunktion h .

□ Definiere $s_j(k) = \left(h(k) + \frac{j + j^2}{2} \right) \bmod N$ für $j = 0, \dots, N - 1$,

d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt wieder mit seinem Streuwert $h(k)$ (eingeschränkt auf den Wertebereich $\{0, \dots, N - 1\}$) und durchläuft dann der Reihe nach die Plätze mit Abstand 1, 3, 6, 10, ... von diesem Anfangsplatz.

□ Praktische Berechnung:

$$s_j(k) = \left(h(k) + \frac{j + j^2}{2} \right) \bmod N = \left(h(k) + \frac{j(j+1)}{2} \right) \bmod N = \left(h(k) + \sum_{i=1}^j i \right) \bmod N =$$

$$= \begin{cases} h(k) \bmod N & \text{für } j = 0 \\ \left(h(k) + \sum_{i=1}^{j-1} i + j \right) \bmod N = (s_{j-1}(k) + j) \bmod N & \text{für } j = 1, \dots, N - 1 \end{cases}$$

□ Zu zeigen:

Die Sondierungssequenz $(s_0(k), \dots, s_{N-1}(k))$ ist für geeignete Werte von N tatsächlich eine Permutation der Indexwerte $0, \dots, N - 1$, d. h. jeder Indexwert tritt in der Sequenz genau einmal auf.

□ Beweis für Zweierpotenzen $N = 2^p$ mit $p \in \mathbb{N}_0$:

- Zu zeigen: Für alle $i, j \in \{0, \dots, N - 1\}$ mit $i \neq j$ gilt: $s_i(k) \neq s_j(k)$.
- Annahme: Es gibt $i, j \in \{0, \dots, N - 1\}$ mit $i \neq j$ (o. B. d. A. $i < j$), für die gilt: $s_i(k) = s_j(k)$.
- Das heißt:

$$\begin{aligned}
 h(k) + \frac{i + i^2}{2} &\equiv h(k) + \frac{j + j^2}{2} \pmod{N} \\
 \Leftrightarrow \frac{i + i^2}{2} &\equiv \frac{j + j^2}{2} \pmod{N} \\
 \Leftrightarrow \frac{j + j^2}{2} - \frac{i + i^2}{2} &= cN = c2^p \text{ für ein } c \in \mathbb{N} \\
 \Leftrightarrow c2^{p+1} &= j + j^2 - i - i^2 = (j - i)(i + j + 1) \\
 \Leftrightarrow (j - i)(i + j + 1) &\text{ ist durch } 2^{p+1} \text{ teilbar}
 \end{aligned}$$

□ Fallunterscheidung:

- Wenn i und j entweder beide gerade oder beide ungerade sind, ist der Faktor $i + j + 1$ ungerade und somit nicht durch 2 teilbar.

Also muss der andere Faktor $j - i$ durch 2^{p+1} teilbar sein.

Wegen $i < j$ sowie $j \leq N - 1$ und $i \geq 0$ gilt jedoch:

$$0 < j - i \leq (N - 1) - 0 = N - 1 = 2^p - 1 < 2^{p+1},$$

also kann $j - i$ nicht durch 2^{p+1} teilbar sein.

- Wenn i gerade und j ungerade ist oder umgekehrt, ist der Faktor $j - i$ ungerade und somit nicht durch 2 teilbar.

Also muss der andere Faktor $i + j + 1$ durch 2^{p+1} teilbar sein.

Wegen $0 \leq i \leq N - 1$ und $0 \leq j \leq N - 1$ gilt jedoch:

$$0 < i + j + 1 \leq (N - 1) + (N - 1) + 1 = 2N - 1 = 2^{p+1} - 1 < 2^{p+1},$$

also kann $i + j + 1$ nicht durch 2^{p+1} teilbar sein.

- Da alle möglichen Fälle zum Widerspruch führen, muss die obige Annahme falsch und damit die Behauptung richtig sein.

- Quadratische Sondierung vermeidet das Problem der direkten Verstopfung:
 - Wenn zwei oder mehr Objekte den gleichen Streuwert i besitzen, werden sie in den Plätzen $i, i + 1, i + 3, i + 6, \dots \pmod{N}$ gespeichert, d. h. die Plätze $i + 2, i + 4, i + 5, \dots \pmod{N}$ bleiben frei, womit die „Verstopfung“ der gesamten Nachbarschaft des Platzes i vermieden wird.
 - Somit können Objekte mit diesen Streuwerten normalerweise direkt in diesen Plätzen gespeichert werden (sofern sie nicht bereits durch andere Objekte mit gleichem Streuwert belegt sind).
 - Objekte mit Streuwerten $i + 1, i + 3, i + 6, \dots \pmod{N}$ können normalerweise in den direkten Nachbarplätzen $i + 2, i + 4, i + 7, \dots \pmod{N}$ gespeichert werden.

- Problem der indirekten Verstopfung (secondary clustering problem):
 - Trotzdem führen viele Objekte mit gleichem Streuwert i nach wie vor zur Bildung einer „Kette“ $i, i + 1, i + 3, i + 6, \dots \pmod{N}$, die bei Operationen für diese Objekte jedesmal ganz oder teilweise durchlaufen werden muss.

- Beispiel: $N = 16, h(k) = k$
Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6
(vgl. § 2.6.3)

2.6.5 Doppelte Streuung (double hashing)

- Gegeben sei eine Tabelle der Größe N sowie zwei Streuwertfunktionen h_1 und h_2 .
- Definiere $s_j(k) = (h_1(k) + j h_2(k)) \bmod N$ für $j = 0, \dots, N - 1$,
 d. h. die Sondierungssequenz $s(k)$ eines Schlüssels k beginnt mit dem Streuwert $h_1(k)$ (eingeschränkt auf den Wertebereich $\{0, \dots, N - 1\}$) und durchläuft dann der Reihe nach die Plätze mit Abstand $h_2(k), 2 h_2(k), 3 h_2(k), \dots$ von diesem Anfangsplatz.
- Damit ist doppelte Streuung ähnlich zu linearer Sondierung, aber mit einer „Schrittweite“, die vom Schlüssel k abhängt.
- Damit werden normalerweise beide Verstopfungs-Probleme vermieden:
 - Wenn zwei oder mehr Objekte den gleichen h_1 -Streuwert i besitzen, besitzen sie in der Regel unterschiedliche h_2 -Streuwerte und somit auch unterschiedliche Sondierungssequenzen.
 - Somit entsteht an der Stelle i normalerweise weder ein Klumpen noch eine Kette.
- Beispiel: $N = 16$, $h_1(k) = k$, $h_2(k) = k \bmod 15 + 1 - k \bmod 15 \bmod 2$
 Einfügen von 0, 16, 32, 48, 64, 80, 1, 2, 3, 4, 5, 6
 (vgl. § 2.6.3 und § 2.6.4)

□ Zu zeigen:

Die Sondierungssequenz $(s_0(k), \dots, s_{N-1}(k))$ ist unter geeigneten Bedingungen tatsächlich eine Permutation der Indexwerte $0, \dots, N - 1$, d. h. jeder Indexwert tritt in der Sequenz genau einmal auf.

□ Beweis für den Fall, dass $h_2(k)$ und N für alle Schlüssel k teilerfremd sind:

○ Zu zeigen: Für alle $i, j \in \{0, \dots, N - 1\}$ mit $i \neq j$ gilt: $s_i(k) \neq s_j(k)$.

○ Annahme: Es gibt $i, j \in \{0, \dots, N - 1\}$ mit $i \neq j$, für die gilt: $s_i(k) = s_j(k)$.

○ Das heißt:

$$h_1(k) + i h_2(k) \equiv h_1(k) + j h_2(k) \pmod{N}$$

$$\Leftrightarrow i h_2(k) \equiv j h_2(k) \pmod{N}$$

$$\Leftrightarrow i \equiv j \pmod{N} \quad (\text{da } h_2(k) \text{ und } N \text{ teilerfremd sind und } h_2(k) \text{ deshalb ein multiplikatives Inverses modulo } N \text{ besitzt})$$

$$\Leftrightarrow i = j \quad (\text{da } i, j \in \{0, \dots, N - 1\})$$

○ Widerspruch!

□ Anmerkung:

Damit $h_2(k)$ und N teilerfremd sind, darf $h_2(k)$ insbesondere nicht 0 sein.

□ Praktische Realisierungsmöglichkeit 1:

- Die Tabellengröße N ist eine Zweierpotenz 2^p mit $p \in \mathbb{N}_0$.
- Die Streuwertfunktion h_2 liefert nur ungerade Zahlen.
- Dann sind $h_2(k)$ und N für alle Schlüssel k teilerfremd.

□ Beispiel:

- $N = 2^{10} = 1024$
- $h_1(k) = k$
- $h_2(k) = 2k + 1$

□ Für $k = 123456$ ergibt sich zum Beispiel:

- $h_1(k) = 123456 \equiv 576 \pmod{1024}$
- $h_2(k) = 2 \cdot 123456 + 1 = 246913 \equiv 129 \pmod{1024}$
- $s_j(k) = (123456 + j \cdot 246913) \pmod{1024} = (576 + j \cdot 129) \pmod{1024}$
für $j = 0, \dots, 1023$

□ Praktische Realisierungsmöglichkeit 2:

- Die Tabellengröße N ist eine Primzahl.
- Die Zahl N' ist „etwas kleiner“ als N , z. B. $N' = N - 1$.
- $h_1(k) = k$
- $h_2(k) = k \bmod N' + 1$
- Wegen $k \bmod N' \in \{0, \dots, N' - 1\}$ und $N' \leq N - 1$ ist $h_2(k) \in \{1, \dots, N'\} \subseteq \{1, \dots, N - 1\}$.
- Da N eine Primzahl ist, sind somit alle Werte von $h_2(k)$ teilerfremd zu N .

□ Beispiel:

- $N = 701, N' = 700$

□ Für $k = 123456$ ergibt sich zum Beispiel:

- $h_1(k) = 123456 \equiv 80 \pmod{701}$
- $h_2(k) = 123456 \bmod 700 + 1 = 257$
- $s_j(k) = (123456 + j \cdot 257) \bmod 701 = (80 + j \cdot 257) \bmod 701$ für $j = 0, \dots, 700$

2.6.6 Laufzeitanalyse

Idealfall

- ❑ Eine Sondierungsfunktion *streut ideal*, wenn jede Permutation der Indexwerte $\{0, \dots, N - 1\}$ mit der gleichen Wahrscheinlichkeit $\frac{1}{N!}$ als Funktionswert auftritt. (Dies wird auch als Uniform-hashing-Annahme bezeichnet.)
- ❑ Bei linearer und quadratischer Sondierung liefert die Sondierungsfunktion maximal N verschiedene Permutationen, die jeweils mit Wahrscheinlichkeit $\frac{1}{N}$ auftreten, sofern die zugrundeliegende Streuwertfunktion ideal streut.
- ❑ Bei doppelter Streuung liefert die Sondierungsfunktion maximal $N(N - 1)$ verschiedene Permutationen, d. h. auch hier ist man theoretisch weit von einer idealen Streuung entfernt. Trotzdem funktioniert doppelte Streuung erfahrungsgemäß sehr gut.
- ❑ Bei den folgenden Analysen wird eine ideal streuende Sondierungsfunktion vorausgesetzt, obwohl diese Annahme in der Praxis nicht wirklich zutreffend ist.
- ❑ Außerdem wird wieder eine Tabelle der Größe N betrachtet, die momentan m Objekte enthält, d. h. $\alpha = \frac{m}{N}$.

Erfolglose Suche

Behauptung

- ❑ Bei einer Suche nach einem Schlüssel k , der nicht in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{\alpha}{1 - \alpha}$ Schlüsselvergleiche ausgeführt.

Anmerkungen

- ❑ Die Behauptung ist nur für $\alpha < 1$ sinnvoll.
- ❑ Die Formulierung „höchstens“ bedeutet nicht, dass sich die Aussage auf den schlimmsten Fall bezieht, sondern dass die *durchschnittliche* Anzahl der Schlüsselvergleiche auf jeden Fall nicht größer als $\frac{\alpha}{1 - \alpha}$ ist.

Beweis

- ❑ Bei der Suche nach einem Objekt mit Schlüssel k werden so lange Schlüssel aus der Tabelle mit k verglichen, bis man auf einen leeren Platz trifft.
(Wegen $\alpha < 1$ gibt es mindestens einen leeren Platz.)
- ❑ Beim ersten Sondierungsversuch sind von insgesamt N Plätzen m Plätze belegt.

- ❑ Beim zweiten Sondierungsversuch (der nur ausgeführt wird, wenn der Platz beim ersten Versuch belegt ist) sind von insgesamt $N - 1$ verbleibenden Plätzen $m - 1$ belegt.
- ❑ Beim dritten Sondierungsversuch (der nur ausgeführt wird, wenn die Plätze beim ersten und zweiten Versuch belegt sind) sind von insgesamt $N - 2$ verbleibenden Plätzen $m - 2$ belegt.
- ❑ Usw.
- ❑ Beim j -ten Sondierungsversuch (der nur ausgeführt wird, wenn die Plätze bei allen vorhergehenden Versuchen belegt sind) sind von insgesamt $N - j + 1$ verbleibenden Plätzen $m - j + 1$ belegt.
- ❑ Daraus folgt: Die Wahrscheinlichkeit $P(V \geq j)$, dass die Anzahl V der Schlüsselvergleiche mindestens j ist, d. h. dass die Plätze der ersten j Sondierungsversuche belegt sind, beträgt für $j = 1, \dots, m$:

$$P(V \geq j) = \frac{m}{N} \cdot \frac{m-1}{N-1} \cdot \frac{m-2}{N-2} \cdots \frac{m-j+1}{N-j+1} \leq \left(\frac{m}{N}\right)^j = \alpha^j$$

(Beachte: Für $0 \leq i \leq m < N$ gilt $\frac{m-i}{N-i} \leq \frac{m}{N}$.

Aufgrund der Uniform-hashing-Annahme hängt der Ausgang eines Sondierungsversuchs nicht von den vorhergehenden Versuchen ab.)

- ❑ Für $j > m$ gilt offensichtlich („unmögliches Ereignis“): $P(V \geq j) = 0 \leq \alpha^j$
- ❑ Somit gilt für alle $j = 1, 2, \dots$: $P(V \geq j) \leq \alpha^j$
- ❑ Für die Wahrscheinlichkeit $P(V = j)$, dass die Anzahl V der Schlüsselvergleiche gleich j ist, gilt offensichtlich für $j = 0, 1, 2, \dots$: $P(V = j) = P(V \geq j) - P(V \geq j + 1)$
- ❑ Für die durchschnittliche Anzahl $E(V)$ der Schlüsselvergleiche („Erwartungswert von V “) gilt somit:

$$E(V) = \sum_{j=0}^{\infty} j \cdot P(V = j) = \sum_{j=0}^{\infty} j \cdot (P(V \geq j) - P(V \geq j + 1)) =$$

$$\sum_{j=0}^{\infty} j \cdot P(V \geq j) - \sum_{j=0}^{\infty} j \cdot P(V \geq j + 1) = \sum_{j=0}^{\infty} j \cdot P(V \geq j) - \sum_{j=1}^{\infty} (j - 1) \cdot P(V \geq j) =$$

$$0 \cdot P(V \geq 0) + \sum_{j=1}^{\infty} (j - (j - 1)) \cdot P(V \geq j) = \sum_{j=1}^{\infty} P(V \geq j) \leq \sum_{j=1}^{\infty} \alpha^j = \frac{\alpha}{1 - \alpha}$$

Einfügen

- ❑ Beim Einfügen eines neuen Objekts (k, v) wird die gleiche Folge von Sondierungsversuchen durchlaufen und damit auch die gleiche Anzahl von Schlüsselvergleichen ausgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k .
- ❑ Deshalb beträgt die durchschnittliche Anzahl von Schlüsselvergleichen bei einer solchen Operation ebenfalls höchstens $\frac{\alpha}{1 - \alpha}$.

Erfolgreiche Suche

Behauptung

- ❑ Bei einer Suche nach einem Schlüssel k , der in der Tabelle enthalten ist, werden im Durchschnitt höchstens $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$ Schlüsselvergleiche ausgeführt.

Beweis

- ❑ Bei der Suche nach dem Objekt mit Schlüssel k wird wiederum die gleiche Folge von Sondierungsversuchen durchlaufen wie beim Einfügen dieses Objekts. Allerdings findet man beim letzten Sondierungsversuch keinen leeren Platz, sondern das gesuchte Objekt, sodass ein Schlüsselvergleich mehr ausgeführt wird.

□ Wenn dieses Objekt das $(j + 1)$ -te eingefügte Objekt ist ($j = 0, \dots, m - 1$), war der Belegungsfaktor α zum Zeitpunkt seiner Einfügung gleich $\frac{j}{N}$.

□ Somit wurden bei dieser Einfügung höchstens $\frac{\frac{j}{N}}{1 - \frac{j}{N}} = \frac{j}{N - j}$ Schlüsselvergleiche ausgeführt.

□ Also werden bei einer Suche nach diesem Objekt höchstens $\frac{j}{N - j} + 1 = \frac{N}{N - j}$ Schlüsselvergleiche ausgeführt.

□ Der Durchschnittswert über alle bis jetzt eingefügten Objekte beträgt:

$$\frac{1}{m} \sum_{j=0}^{m-1} \frac{N}{N-j} = \frac{N}{m} \sum_{j=0}^{m-1} \frac{1}{N-j} = \frac{1}{\alpha} \sum_{i=N-m+1}^N \frac{1}{i} \leq \frac{1}{\alpha} \int_{N-m}^N \frac{1}{x} dx = \frac{1}{\alpha} \left[\ln x \right]_{N-m}^N =$$

$$\frac{1}{\alpha} (\ln N - \ln (N - m)) = \frac{1}{\alpha} \ln \frac{N}{N - m} = \frac{1}{\alpha} \ln \frac{1}{1 - \frac{m}{N}} = \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

(Für die Abschätzung wird das aus der Analysis bekannte Integralkriterium verwendet.)

Ersetzen

- ❑ Beim Ersetzen eines bereits vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts, d. h. höchstens $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$.

Erfolgreiches und erfolgloses Löschen

- ❑ Beim Löschen eines vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolgreichen Suche nach dem Schlüssel k dieses Objekts.
- ❑ Beim Löschen eines nicht vorhandenen Objekts (k, v) werden genauso viele Schlüsselvergleiche durchgeführt wie bei einer erfolglosen Suche nach dem Schlüssel k dieses Objekts.

2.7 Laufzeitvergleich

- Die folgende Tabelle zeigt die durchschnittliche Anzahl von Schlüsselvergleichen bei einer erfolglosen bzw. erfolgreichen Suche in Abhängigkeit vom Belegungsfaktor α bei Verwendung von Verkettung bzw. offener Adressierung, jeweils unter der entsprechenden Uniform-hashing-Annahme.
- Bei den Formeln für offene Adressierung handelt es sich jedoch um Abschätzungen nach oben, d. h. die tatsächlichen Zahlenwerte sind u. U. deutlich kleiner.

α	Verkettung		Offene Adressierung	
	erfolglos	erfolgreich	erfolglos	erfolgreich
	α	$1 + \frac{\alpha}{2}$	$\frac{\alpha}{1 - \alpha}$	$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$
0.1	0.1	1.05	0.111111	1.05361
0.2	0.2	1.1	0.25	1.11572
0.5	0.5	1.25	1	1.38629
0.75	0.75	1.375	3	1.84839
0.9	0.9	1.45	9	2.55843
0.95	0.95	1.475	19	3.15340

2.8 Vergrößern und Verkleinern von Streuwerttabellen

2.8.1 Arithmetische Vergrößerung von Feldern

- ❑ Gegeben sei ein Feld mit Anfangsgröße $N_1 = N$, das sukzessive gefüllt wird.
- ❑ Wenn es voll ist, wird es um N Elemente auf $N_2 = 2N$ „vergrößert“, indem ein neues Feld der Größe N_2 beschafft und die bereits vorhandenen N_1 Elemente umkopiert werden.
- ❑ Wenn dieses Feld wieder voll ist, wird es erneut um N Elemente auf $N_3 = 3N$ vergrößert, wobei die jetzt vorhandenen N_2 Elemente umkopiert werden müssen.
- ❑ Usw.
- ❑ Unmittelbar vor der k -ten Vergrößerung besitzt das Feld die Größe $N_k = kN$, und es mussten insgesamt $N_1 + \dots + N_{k-1} = \sum_{j=1}^{k-1} N_j = \sum_{j=1}^{k-1} jN = \frac{k(k-1)}{2}N$ Elemente kopiert werden.

- Verteilt man diesen Gesamtkopieraufwand gleichmäßig auf die $k N$ Elemente, ergibt sich als *amortisierter Aufwand* für das Hinzufügen eines einzelnen Elements

$$\frac{\frac{k(k-1)}{2} N}{k N} = \frac{k-1}{2},$$

d. h. dieser Aufwand wird umso größer, je öfter das Feld vergrößert wird.

2.8.2 Geometrische Vergrößerung von Feldern

- Gegeben sei ein Feld mit Anfangsgröße $N_0 = N$, das sukzessive gefüllt wird.
- Wenn es voll ist, wird es um den Faktor q auf $N_1 = q N$ vergrößert, wobei die bereits vorhandenen N_0 Elemente umkopiert werden müssen.
- Wenn es wieder voll ist, wird es erneut um den Faktor q auf $N_2 = q^2 N$ vergrößert, wobei die jetzt vorhandenen N_1 Elemente umkopiert werden müssen.
- Usw.
- Unmittelbar vor der $(k + 1)$ -ten Vergrößerung besitzt das Feld die Größe $N_k = q^k N$, und es mussten insgesamt

$$N_0 + \dots + N_{k-1} = \sum_{j=0}^{k-1} N_j = \sum_{j=0}^{k-1} q^j N = N \sum_{j=0}^{k-1} q^j = N \frac{q^k - 1}{q - 1} < N \frac{q^k}{q - 1}$$
 Elemente kopiert werden.

- Verteilt man diesen Gesamtkopieraufwand wieder gleichmäßig auf die $q^k N$ Elemente, ergibt sich als amortisierter Aufwand für das Hinzufügen eines

einzelnen Elements weniger als $\frac{N \frac{q^k}{q-1}}{q^k N} = \frac{1}{q-1}$,

d. h. dieser Aufwand bleibt unabhängig von der Anzahl der Vergrößerungen konstant. (Der Kopieraufwand für eine einzelne Vergrößerung wird natürlich trotzdem jedesmal größer.)

2.8.3 Einfaches Vergrößern und Verkleinern von Streuwerttabellen

- ❑ Wenn der Belegungsfaktor einer Streuwerttabelle einen bestimmten kritischen Wert (typischerweise etwa $3/4$, vgl. die Zahlenwerte in § 2.7) überschreitet, ist es ratsam, die Tabelle zu vergrößern.
- ❑ Aufgrund der vorangegangenen Überlegungen in § 2.8.1 und § 2.8.2 ist eine (in etwa) geometrische Vergrößerung sinnvoll (ggf. unter Beachtung der Kriterien für „gute“ Tabellengrößen in § 2.3).
- ❑ Dabei muss auch die Streuwertfunktion an die neue Tabellengröße angepasst werden.
- ❑ Beim Umkopieren der vorhandenen Objekte muss ihr Platz in der neuen Tabelle jeweils mit der neuen Streuwertfunktion berechnet werden, was den Aufwand für das Vergrößern nochmals erhöht.
- ❑ Wenn der Belegungsfaktor kleiner als ein bestimmter Wert wird, kann entsprechend auch eine Verkleinerung der Tabelle sinnvoll sein, um Speicherplatz zu sparen.
- ❑ Dabei ist jedoch darauf zu achten, dass die verkleinerte Tabelle wieder ausreichend Platz für neue Objekte besitzt, bevor sie erneut vergrößert werden muss, um ein ständiges Pendeln zwischen Vergrößern und Verkleinern auch unter ungünstigen Bedingungen zu vermeiden.

2.8.4 Inkrementelles Vergrößern und Verkleinern von Streuwerttabellen

- ❑ Obwohl der amortisierte Aufwand jeder Einfügeoperation bei geometrischer Vergrößerung konstant ist, ist der tatsächliche Aufwand derjenigen Einfügeoperation, die zu einer Vergrößerung der Tabelle führt, proportional zur momentanen Tabellengröße und damit potentiell sehr hoch.
- ❑ Dies kann insbesondere für interaktive und Echtzeitanwendungen problematisch sein.
- ❑ Beim inkrementellen Vergrößern wird die alte (kleine) Tabelle zunächst aufbewahrt und das aufwendige Umkopieren der Objekte schrittweise ausgeführt:
 - Neue Objekte werden mit der neuen Streuwertfunktion in die neue (vergrößerte) Tabelle eingefügt.
 - Bei jeder solchen Einfügung werden außerdem ein paar weitere Objekte von der alten in die neue Tabelle umkopiert.
 - Wenn alle Objekte umkopiert wurden, kann die alte Tabelle freigegeben werden.
 - Um ein Objekt zu suchen oder zu löschen, muss es ggf. in der alten und in der neuen Tabelle (jeweils mit der zugehörigen Streuwertfunktion) gesucht werden.

- ❑ Um sicherzustellen, dass alle Objekte umkopiert wurden, bevor die (neue) Tabelle erneut vergrößert werden muss, muss die Anzahl der umkopierten Objekte pro Einfügeoperation geeignet gewählt werden.
- ❑ Zum Beispiel: Wenn die Tabellengröße bei jeder Vergrößerung verdoppelt wird, muss bei jeder Einfügeoperation mindestens ein weiteres Objekt umkopiert werden.

2.9 Ausblick

- (Minimale) perfekte Streuwertfunktionen
- Kryptographische Streuwertfunktionen

3 Vorrangwarteschlangen (priority queues)

3.1 Einleitung

- ❑ Eine *Maximum-Vorrangwarteschlange* ist eine Datenstruktur, die folgende Operationen möglichst effizient unterstützt:
 - Einfügen eines Objekts mit einer bestimmten Priorität
 - Auslesen und ggf. Entnehmen eines Objekts mit maximaler Priorität
 - Nachträgliches Ändern der Priorität eines Objekts
 - Entfernen eines Objekts
 - Eventuell zusätzlich: Vereinigen zweier Warteschlangen
- ❑ Eine *Minimum-Vorrangwarteschlange* unterstützt entsprechend das Auslesen und ggf. Entnehmen eines Objekts mit minimaler statt maximaler Priorität.
- ❑ Anwendungsbeispiele:
 - Prozess-Disponent (scheduler) eines Betriebssystems
 - Huffman-Kodierung (vgl. § 4.3)
 - Bestimmte Graphalgorithmen (vgl. § 5.5.7 und § 5.6.5)

3.2 Binäre Halden (binary heaps)

3.2.1 Interpretation eines Felds als Binärbaum

- ❑ Element 1 = 1_2 : Wurzelknoten
- ❑ Element 2 = $10_2 = 2 \cdot 1 + 0$: Linker Nachfolger des Wurzelknotens
- ❑ Element 3 = $11_2 = 2 \cdot 1 + 1$: Rechter Nachfolger des Wurzelknotens
- ❑ Element 4 = $100_2 = 2 \cdot 2 + 0$:
Linker Nachfolger des linken Nachfolgers des Wurzelknotens
- ❑ Element 5 = $101_2 = 2 \cdot 2 + 1$:
Rechter Nachfolger des linken Nachfolgers des Wurzelknotens
- ❑ Element 6 = $110_2 = 2 \cdot 3 + 0$:
Linker Nachfolger des rechten Nachfolgers des Wurzelknotens
- ❑ Element 7 = $111_2 = 2 \cdot 3 + 1$:
Rechter Nachfolger des rechten Nachfolgers des Wurzelknotens
- ❑ Usw.

Allgemein

- ❑ Zu einem Element i ist
 - Element $2i + 0$ sein linker Nachfolger (falls $2i + 0 \leq m$)
 - Element $2i + 1$ sein rechter Nachfolger (falls $2i + 1 \leq m$)
 - Element $\left\lfloor \frac{i}{2} \right\rfloor$ sein Vorgänger (falls $i > 1$)
- ❑ Dabei bezeichnet m die Gesamtzahl der Elemente des Baums, die auch kleiner als die Größe N des Felds sein kann.
- ❑ Ebene l ($l = 0, 1, \dots$) enthält die Elemente 2^l einschließlich bis 2^{l+1} ausschließlich (jedoch bis maximal m einschließlich).
- ❑ Damit befindet sich Element i auf Ebene $\lfloor \log_2 i \rfloor$.

Anmerkungen

- ❑ Die Multiplikation $2i$ ist effizient als Verschiebung des Bitmusters von i um eine Position nach links ($i \ll 1$ in C, Java, ...) berechenbar.
- ❑ Die ganzzahlige Division $\left\lfloor \frac{i}{2} \right\rfloor$ ist effizient als Verschiebung des Bitmusters von i um eine Position nach rechts ($i \gg 1$ in C, Java, ...) berechenbar.

3.2.2 Minimum- und Maximum-Halden

❑ Definition:

Ein Baum erfüllt die *Maximum-Bedingung*, wenn jeder seiner Knoten außer der Wurzel höchstens so groß wie sein Vorgänger ist.

❑ Folgerung:

Alle Knoten eines Teilbaums sind höchstens so groß wie die Wurzel dieses Teilbaums.

❑ Insbesondere:

Die Wurzel des gesamten Baums ist ein maximales Element des Baums.

❑ Definition:

Die ersten m Elemente eines Felds stellen eine *binäre Maximum-Halde* dar, wenn der durch diese Elemente definierte Binärbaum die Maximum-Bedingung erfüllt.

❑ *Minimum-Bedingung* und *Minimum-Halde* sind analog definiert.

3.2.3 Operationen auf Maximum-Halden (auf Minimum-Halden analog)

Absenken eines Elements

- ❑ Vorbedingung:
Linker und rechter Teilbaum (falls vorhanden) des Elements i erfüllen die Maximum-Bedingung.
- ❑ Nachbedingung/Ziel:
Der Teilbaum mit Wurzelknoten i erfüllt die Maximum-Bedingung.
- ❑ Ablauf:
 - 1 Wenn Knoten i keine Nachfolger besitzt, ist nichts zu tun.
 - 2 Wenn er genau einen (d. h. einen linken) Nachfolger besitzt, ist dies natürlich der größte Nachfolger.
 - 3 Wenn er zwei Nachfolger besitzt, bestimme den größeren von ihnen.
(Falls beide gleich sind, wähle willkürlich einen von beiden.)
 - 4 Wenn Element i kleiner als dieser größte Nachfolger ist, vertausche es mit ihm und wiederhole dann die gesamte Operation für diesen Nachfolger.

- Beweis der Korrektheit durch Induktion nach der Tiefe t des betrachteten Teilbaums:
 - Induktionsanfang $t = 0$:
In diesem Fall tut die Operation nichts, was korrekt ist, weil der Teilbaum bereits die Maximum-Bedingung erfüllt.
 - Induktionsschritt $t \rightarrow t + 1$:
 - Wenn Element i kleiner als sein größter Nachfolger ist, wird es mit ihm vertauscht, wodurch die Maximum-Bedingung für den entsprechenden Teilbaum verletzt werden kann.
 - Für die Teilbäume dieses Teilbaums bleibt die Bedingung jedoch erhalten. Daher stellt der rekursive Aufruf der Operation für diesen Teilbaum die Maximum-Bedingung gemäß Induktionsvoraussetzung wieder her. (Dieser Teilbaum hat Tiefe $\leq t$.)
 - Aufgrund der zuvor durchgeführten Vertauschung ist die Bedingung dann auch für den Teilbaum mit Wurzelknoten i erfüllt.
 - Andernfalls (Element i nicht kleiner als sein größter Nachfolger) tut die Operation nichts, was korrekt ist:
Da Element i mindestens so groß wie seine Nachfolger ist und seine Teilbäume die Maximum-Bedingung gemäß Vorbedingung erfüllen, erfüllt auch der Teilbaum mit Wurzelknoten i die Bedingung.
- Die Laufzeit der Operation ist offenbar proportional zur Tiefe t des betrachteten Teilbaums und damit höchstens $O(\log_2 m)$.

Exkurs

- ❑ Eine *Schleifeninvariante* ist eine Aussage, die an folgenden Stellen gilt:
 - Vor Beginn einer Schleife
 - Am Anfang und am Ende jedes Schleifendurchlaufs
 - Nach Beendigung der Schleife

- ❑ Um zu beweisen, dass eine bestimmte Aussage tatsächlich eine Schleifeninvariante darstellt, genügt es (ähnlich wie bei vollständiger Induktion), zwei Dinge zu zeigen (unter der Annahme, dass die Auswertung der Schleifenbedingung keine Nebeneffekte verursacht):
 - *Initialisierung:*
Die Invariante gilt vor Beginn der Schleife.
(Dann gilt sie auch am Anfang des ersten Durchlaufs, sofern es überhaupt einen Durchlauf gibt.)

 - *Aufrechterhaltung:*
Wenn die Invariante am Anfang eines Durchlaufs gilt, dann gilt sie auch am Ende dieses Durchlaufs (und damit auch am Anfang des nächsten Durchlaufs, sofern es einen solchen gibt).
(Zusammen mit der Initialisierung folgt dann durch vollständige Induktion, dass die Invariante nach *jedem* Durchlauf gilt.)

□ Daraus folgt dann automatisch:

○ *Terminierung:*

Wenn die Schleife terminiert (was ggf. anderweitig gezeigt werden muss, sofern es nicht offensichtlich ist), dann gilt die Invariante auch nach ihrer Beendigung.

□ Beispiel:

```
int pow (int x, int n) {
    int p = 1, k = 0;
    while (k < n) {
        p = p * x;
        k = k + 1;
    }
    return p;
}
```

Um zu zeigen, dass diese Funktion für $n \geq 0$ tatsächlich x^n berechnet, kann die Schleifeninvariante $p = x^k$ verwendet werden:

○ Initialisierung:

Vor Beginn der Schleife gilt: $p = 1 = x^0 = x^k$

○ Aufrechterhaltung:

Wenn die Aussage $p = x^k$ am Anfang eines Durchlaufs gilt, dann gilt am Ende dieses Durchlaufs: $p' = p \cdot x = x^k \cdot x = x^{k+1} = x^{k'}$.

Dabei bezeichnen p und k die Werte der entsprechenden Variablen am Anfang des Durchlaufs, p' und k' ihre Werte am Ende des Durchlaufs.

○ Terminierung:

Nach Beendigung der Schleife ist $k = n$ (sofern $n \geq 0$ ist) und somit gilt:

$$p = x^k = x^n.$$

□ Anmerkung:

for-Schleifen sind für derartige Beweise aus mehreren Gründen schlecht geeignet:

○ Die Veränderung der Laufvariable erfolgt (je nach konkreter syntaktischer Form) mehr oder weniger „versteckt“.

○ Vor Beginn und nach Beendigung der Schleife existiert die Laufvariable meist gar nicht.

Deshalb sollten for-Schleifen in äquivalente while-Schleifen umgeschrieben werden, wenn die Laufvariable in der Schleifeninvariante gebraucht wird.

Herstellen einer Maximum-Halde

□ Gegeben: Ein Feld der Größe N mit beliebigen Elementen.

□ Nachbedingung/Ziel:

Die ersten m Elemente des Felds stellen eine Maximum-Halde dar.

□ Ablauf:

Für $i = \left\lfloor \frac{m}{2} \right\rfloor, \dots, 1$:

Führe die zuvor beschriebene Operation „Absenken“ für das Element i aus.

□ Äquivalente Formulierung mit while-Schleife:

1 Setze $i = \left\lfloor \frac{m}{2} \right\rfloor$.

2 Solange $i \geq 1$ ist:

1 Führe die zuvor beschriebene Operation „Absenken“ für das Element i aus.

2 Erniedrige i um 1.

□ Beweis der Korrektheit mit Hilfe folgender Schleifeninvariante:

Alle Teilbäume mit Wurzelknoten $i + 1, \dots, m$ erfüllen die Maximum-Bedingung.

○ Initialisierung:

– Vor Beginn der Schleife ist $i = \lfloor \frac{m}{2} \rfloor$.

– Wegen $\lfloor x \rfloor > x - 1$ gilt für $j = i + 1, \dots, m$:

$$2j \geq 2(i + 1) = 2\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right) > 2\left(\left(\frac{m}{2} - 1\right) + 1\right) = 2 \frac{m}{2} = m.$$

– Daher besitzen diese Knoten j gemäß § 3.2.1 keine Nachfolger.

– Also erfüllen die entsprechenden Teilbäume die Maximum-Bedingung.

○ Aufrechterhaltung:

– Zu Beginn eines Schleifendurchlaufs erfüllen alle Teilbäume mit Wurzelknoten $i + 1, \dots, m$ aufgrund der Invariante die Maximum-Bedingung.

– Wegen $2i + 0 \geq i + 1$ und $2i + 1 \geq i + 1$ erfüllen insbesondere die Teilbäume des Knotens i die Maximum-Bedingung, sofern sie existieren.

– Somit ist vor dem Aufruf der Operation „Absenken“ deren Vorbedingung und damit nach ihrem Aufruf auch ihre Nachbedingung erfüllt, d. h. auch der Teilbaum mit Wurzelknoten i erfüllt dann die Maximum-Bedingung.

– Da am Ende des Schleifendurchlaufs $i' = i - 1$ ist, erfüllen damit alle Teilbäume mit Wurzelknoten $i' + 1, \dots, m$ die Maximum-Bedingung.

○ Terminierung:

- Nach Beendigung der Schleife ist $i = 0$.
- Aufgrund der Invariante erfüllen also alle Teilbäume mit Wurzelknoten $1, \dots, m$ die Maximum-Bedingung.
- Insbesondere erfüllt der gesamte Baum mit Wurzelknoten 1 die Maximum-Bedingung, d. h. die ersten m Elemente des Felds stellen eine Maximum-Halde dar.

□ Laufzeit

- Die Tiefe des gesamten Baums ist $t = \lfloor \log_2 m \rfloor$.
- Ein Teilbaum, dessen Wurzelknoten sich auf Ebene $l = 0, 1, \dots$ befindet, hat höchstens Tiefe $t - l$. Dementsprechend ist die Laufzeit der Operation „Absenken“ für einen solchen Teilbaum $O(t - l)$.
- Auf Ebene $l = 0, \dots, t - 1$ gibt es höchstens 2^l abzusenkende Knoten. (Auf der untersten Ebene t werden keine Knoten abgesenkt.)
- Also ist die Gesamtlaufzeit höchstens:

$$\sum_{l=0}^{t-1} 2^l \cdot (t - l) = \sum_{k=t-l}^t 2^{t-k} \cdot k = 2^t \cdot \sum_{k=1}^t k \cdot \left(\frac{1}{2}\right)^k \leq m \cdot \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\right)^k = m \cdot \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} = 2m$$

(Beachte: Für $|q| < 1$ gilt: $\sum_{k=1}^{\infty} k \cdot q^k = \frac{q}{(1 - q)^2}$.)

Sortieren eines Felds

- ❑ Gegeben: Ein Feld der Größe N mit beliebigen Elementen.
- ❑ Nachbedingung/Ziel: Die Elemente des Felds sind aufsteigend sortiert.
- ❑ Ablauf:
 - 1 Setze $m = N$
 - 2 Führe die Operation „Herstellen einer Maximum-Halde“ aus.
 - 3 Solange $m > 1$ ist:
 - 1 Vertausche Element 1 und m des Felds.
 - 2 Verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
 - 3 Führe die Operation „Absenken“ für Element 1 aus.

□ Beweis der Korrektheit mit Hilfe folgender Schleifeninvariante:

1. Die ersten m Elemente des Felds stellen eine Maximum-Halbe dar.
2. Die übrigen Elemente des Felds sind aufsteigend sortiert und mindestens so groß wie die ersten m Elemente.

○ Initialisierung:

- Vor Beginn der Schleife stellen die ersten m Elemente eine Maximum-Halbe dar.
- Weitere Elemente gibt es zu diesem Zeitpunkt nicht.

○ Aufrechterhaltung:

- Aufgrund der Invariante gilt zu Beginn eines Durchlaufs: Element 1 ist mindestens so groß wie die Elemente $2, \dots, m$ und höchstens so groß wie die übrigen Elemente des Felds.
- Also gilt nach dem Vertauschen von Element 1 und m und dem Erniedrigen von m Teil 2 der Invariante.
- Nach dem Ausführen der Operation „Absenken“ gilt auch Teil 1 der Invariante wieder, der durch das Vertauschen eventuell ungültig geworden war.

○ Terminierung:

- Nach Beendigung der Schleife ist $m = 1$.
- Also sind nach Teil 2 der Invariante die Elemente $2, \dots, N$ aufsteigend sortiert und mindestens so groß wie Element 1.
- Damit ist das gesamte Feld aufsteigend sortiert.

□ Laufzeit

- Es findet ein Aufruf der Operation „Herstellen einer Maximum-Halde“ mit Laufzeit $O(N)$ sowie $N - 1 = O(N)$ Aufrufe der Operation „Absenken“ statt, deren Laufzeit jeweils höchstens $O(\log N)$ beträgt.
- Damit ergibt sich eine Gesamtlaufzeit von $O(N) + O(N) \cdot O(\log N) = O(N \log N)$.

3.2.4 Implementierung von Vorrangwarteschlangen

- ❑ Eine Maximum-Vorrangwarteschlange kann wie folgt durch eine binäre Maximum-Halde implementiert werden (und analog kann eine Minimum-Vorrangwarteschlange durch eine Minimum-Halde implementiert werden).
- ❑ N bezeichne die maximale Kapazität der Warteschlange, d. h. die Größe des zugrundeliegenden Felds.
- ❑ m bezeichne die momentane Länge der Warteschlange, d. h. die Anzahl der momentan gespeicherten Objekte.
Dies ist die gleichzeitig die momentane Größe der Halde.

Hilfsoperationen

- ❑ Anheben eines Elements

Solange das Element einen Vorgänger besitzt und seine Priorität größer als die des Vorgängers ist, vertausche das Element mit seinem Vorgänger.

☐ Anheben oder Absenken eines Elements

- 1 Wenn die Priorität des Elements größer als die seines Vorgängers ist, führe die Operation „Anheben“ für das Element aus.
- 2 Andernfalls führe die Operation „Absenken“ für das Element aus (wobei sich die dabei durchgeführten Vergleiche auf die Prioritäten der Elemente beziehen).

Operationen

☐ Einfügen eines Objekts mit einer bestimmten Priorität

- 1 Vergrößere die Halde um ein Element, d. h. erhöhe m um 1, und speichere das neue Objekt an Position m .
- 2 Führe die Hilfsoperation „Anheben“ für das Element an Position m aus.

☐ Auslesen eines Objekts mit maximaler Priorität

Liefere das Objekt an Position 1 der Halde.

❑ Entnehmen eines Objekts mit maximaler Priorität

- 1 Merke das Objekt an Position 1 der Halde, um es später als Resultat liefern zu können.
- 2 Versetze das Objekt an Position m der Halde an Position 1 und verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
- 3 Führe die Operation „Absenken“ für das Element an Position 1 aus.
- 4 Liefere das anfangs gemerkte Objekt zurück.

❑ Nachträgliches Ändern der Priorität eines Objekts

- 1 Ändere die Priorität des Objekts.
- 2 Führe die Operation „Anheben oder Absenken“ für das Objekt aus.

❑ Entfernen eines Objekts

- 1 Ersetze das zu entfernende Objekt durch das Objekt an Position m der Halde und verkleinere die Halde um ein Element, d. h. erniedrige m um 1.
- 2 Führe die Operation „Anheben oder Absenken“ für das ersetzte Element aus.

Laufzeit der Operationen

- ❑ Das Auslesen eines Objekts mit maximaler Priorität benötigt offensichtlich Laufzeit $O(1)$.

- ❑ Alle anderen Operationen verschieben ein Objekt innerhalb der Halde ggf. um eine oder mehrere Ebenen nach oben oder unten.
Dementsprechend ist ihre Laufzeit höchstens so groß wie die Tiefe des Baums, d. h. $O(\log m)$.

3.2.5 Praktisches Problem

- ❑ Wie finden die Operationen „Ändern der Priorität“ und „Entfernen“ innerhalb der Halde effizient (d. h. ohne sequentielle Suche) das jeweilige Objekt?
- ❑ Erste Idee:
 - Die Operation „Einfügen“ liefert die Position im Feld zurück, an der sie das Objekt gespeichert hat.
 - Die Operationen „Ändern der Priorität“ und „Entfernen“ erhalten diese Position anstelle des Objekts selbst.
- ❑ Neues Problem: Die Position eines Objekts ist nicht stabil, weil viele Operationen Objekte in der Halde vertauschen.
- ❑ Lösung (wie für viele andere Informatik-Probleme) durch eine zusätzliche Indirektion:
 - Objekte werden nicht direkt in der Halde, sondern in einem zweiten Feld gespeichert.
 - Die Operation „Einfügen“ liefert die Position des Objekts in diesem Feld zurück.
 - In der Halde werden nur diese stabilen Positionen der Objekte gespeichert.
 - Umgekehrt wird im zweiten Feld für jedes Objekt auch seine momentane Position in der Halde gespeichert und bei Bedarf (d. h. wenn sie sich aufgrund einer Vertauschung ändert) aktualisiert.

3.3 Binomial-Halden (binomial heaps)

3.3.1 Binomial-Bäume

Definition

- Ein *Binomial-Baum* mit Grad 0 besteht aus einem einzelnen Knoten.
- Für $k \geq 1$ entsteht ein *Binomial-Baum* mit Grad k aus zwei Binomial-Bäumen mit Grad $k - 1$, indem einer von ihnen zu einem Nachfolger des anderen gemacht wird.

Eigenschaften von Binomial-Bäumen

Für jeden Binomial-Baum mit Grad $k \in \mathbb{N}_0$ gilt:

1. Die Tiefe des Baums ist k .
2. Der Grad seines Wurzelknotens ist k .
3. Der Grad aller anderen Knoten ist kleiner als k .
4. Die Nachfolger des Wurzelknotens sind Binomial-Bäume mit Grad $k - 1, \dots, 0$.
5. Der Baum besitzt 2^k Knoten.
6. Auf Ebene l ($l = 0, \dots, k$) gibt es genau $\binom{k}{l}$ Knoten.

3.3.2 Minimum- und Maximum-Halden

□ Definition:

- Ein *Maximum-Binomial-Baum* ist ein Binomial-Baum, der die Maximum-Bedingung (vgl. § 3.2.2) erfüllt.
- Eine *Maximum-Binomial-Halde* ist eine (endliche) Folge von Maximum-Binomial-Bäumen, deren Grad streng monoton wächst.
- *Minimum-Binomial-Baum* und *Minimum-Binomial-Halde* sind analog definiert.

□ Folgerung:

- Alle Binomial-Bäume innerhalb einer Binomial-Halde besitzen unterschiedlichen Grad.
- Eine Binomial-Halde mit N Elementen besteht aus Binomial-Bäumen mit Grad $k_1 < \dots < k_p$, sodass gilt: $N = \sum_{i=1}^p 2^{k_i}$, d. h. k_1, \dots, k_p sind genau die Ziffern mit Wert 1 in der Dualdarstellung von N .
- Sowohl die Anzahl der Bäume in einer solchen Halde als auch deren Grad und Tiefe ist jeweils höchstens $O(\log_2 N)$.

3.3.3 Repräsentation von Binomial-Halden

- Jeder Knoten eines Binomial-Baums bzw. einer Binomial-Halde kann durch eine Datenstruktur mit folgenden Attributen repräsentiert werden:
 - `degree` speichert den Grad des Knotens.
 - `parent` verweist auf den Vorgänger des Knotens.
Bei einem Wurzelknoten ist `parent` gleich \perp .
 - `child` verweist auf den Nachfolger des Knotens mit dem größten Grad.
Bei einem Blattknoten ist `child` gleich \perp .
 - `sibling` verkettet alle Nachfolger eines Knotens in einer nach aufsteigendem Grad sortierten zirkulären Liste, das heißt:
 - Der Nachfolger mit dem größten Grad verweist auf den Nachfolger mit dem kleinsten Grad.
 - Jeder andere Nachfolger verweist auf den Nachfolger mit dem nächstgrößeren Grad.
 - Wenn es nur einen Nachfolger gibt, verweist er auf sich selbst.
 Außerdem verkettet `sibling` die Wurzelknoten aller Bäume der Halde in einer nach aufsteigendem Grad sortierten einfachen Liste, d. h. jeder Wurzelknoten verweist ggf. auf den Wurzelknoten mit dem nächstgrößeren Grad.
 - `entry` verweist auf das Objekt, das in diesem Knoten gespeichert werden soll und das seinerseits auf diesen Knoten zurückverweist (vgl. auch § 3.2.5).

- ❑ Die gesamte Halde wird durch einen Verweis auf den Wurzelknoten des Baums mit dem kleinsten Grad repräsentiert. (Bei einer leeren Halde ist dieser Verweis gleich \perp .)

Erläuterungen

- ❑ Mit den Attributen `child` und `sibling` können ohne Verwendung dynamischer Felder o. ä. für jeden Knoten beliebig viele Nachfolger gespeichert werden.
- ❑ Die Gründe für die weiteren Details der Organisation (Reihenfolge der `sibling`-Verkettungen, Verwendung einfacher oder zyklischer Verkettung) werden in § 3.3.5 erläutert, nachdem in § 3.3.4 die Operationen auf dieser Datenstruktur beschrieben wurden.

3.3.4 Implementierung von Vorrangwarteschlangen

- Eine Minimum-Vorrangwarteschlange kann wie folgt durch eine Minimum-Binomial-Halbe implementiert werden (und analog kann eine Maximum-Vorrangwarteschlange durch eine Maximum-Binomial-Halbe implementiert werden).

Hilfsoperation

- Zusammenfassen zweier Bäume B_1 und B_2 mit Grad k zu einem Baum mit Grad $k + 1$

- 1 Wenn die Priorität des Wurzelknotens von B_1 größer als die Priorität des Wurzelknotens von B_2 ist, mache B_1 zum Nachfolger mit dem größten Grad von B_2 :

```

B2.sibling = nil
B2.degree = B2.degree + 1
B1.parent = B2
if B2.child == nil then
    B2.child = B1.sibling = B1
else
    B1.sibling = B2.child.sibling
    B2.child = B2.child.sibling = B1
end
    
```

- 2 Andernfalls mache B_2 zum Nachfolger mit dem größten Grad von B_1 .

Operationen

- Vereinigen zweier Halden H_1 und H_2 zu einer neuen Halde H
 - 1 Erstelle einen leeren Zwischenspeicher für bis zu drei Bäume.
 - 2 Setze $k = 0$.
 - 3 Solange H_1 oder H_2 oder der Zwischenspeicher nicht leer sind:
 - 1 Wenn der erste (verbleibende) Baum von H_1 Grad k besitzt, entnimm ihn aus H_1 und füge ihn zum Zwischenspeicher hinzu.
 - 2 Entsprechend für H_2 .
 - 3 Wenn der Zwischenspeicher jetzt einen oder drei Bäume enthält, entnimm einen von ihnen und füge ihn am Ende von H an.
 - 4 Wenn der Zwischenspeicher jetzt noch zwei Bäume enthält, fasse sie zu einem Baum mit Grad $k + 1$ zusammen, der als „Übertrag“ für den nächsten Schritt im Zwischenspeicher verbleibt.
 - 5 Erhöhe k um 1.

Beispiel: H_1 mit $2^2 + 2^1 + 2^0 = 7$ Elementen und H_2 mit $2^3 + 2^2 + 2^1 = 14$ Elementen ergibt H mit $2^4 + 2^2 + 2^0 = 21$ Elementen

❑ Einfügen eines Objekts mit einer bestimmten Priorität

Erzeuge eine temporäre Halde mit einem einzigen Baum mit Grad 0, die das Objekt enthält, und vereinige sie mit der aktuellen Halde.

❑ Auslesen eines Objekts mit minimaler Priorität

Suche in der Liste der Wurzelknoten ein Objekt mit minimaler Priorität.

❑ Entnehmen eines Objekts mit minimaler Priorität

1 Suche in der Liste der Wurzelknoten ein Objekt mit minimaler Priorität und entferne diesen Knoten aus der Liste.

2 Wenn dieser Knoten Nachfolger besitzt:

Vereinige die Liste seiner Nachfolger (beginnend mit dem Nachfolger mit dem kleinsten Grad, der über `child` → `sibling` direkt zugreifbar ist) mit der verbleibenden Halde.

❑ Nachträgliches Ändern der Priorität eines Objekts

- 1 Wenn die neue Priorität des Objekts kleiner oder gleich der alten ist:
 - 1 Ändere die Priorität des Objekts.
 - 2 Solange die Priorität des Objekts kleiner als die seines Vorgängers ist:
Vertausche die `entry`-Verweise der beiden Knoten
und aktualisiere die zugehörigen Rückverweise der Objekte auf diese Knoten.
- 2 Andernfalls, sofern sich das Objekt nicht in einem Blattknoten befindet:
Entferne das Objekt und füge es mit der neuen Priorität wieder ein.

❑ Entfernen eines Objekts

- 1 Ändere die Priorität des Objekts quasi auf $-\infty$.
- 2 Führe dann die Operation „Entnehmen“ aus.

Laufzeit der Operationen

- ❑ Wenn die Halde N Objekte enthält, besitzen alle `sibling`-Listen höchstens Länge $O(\log_2 N)$ und alle Bäume höchstens Tiefe $O(\log_2 N)$ (vgl. § 3.3.2).
- ❑ Daher besitzen alle Operationen höchstens Laufzeit $O(\log_2 N)$.

3.3.5 Entwurfsüberlegungen für die Datenstruktur

- ❑ Beim Vereinigen zweier Halden müssen ihre Bäume nach aufsteigendem Grad durchlaufen werden.
Deshalb sind die Wurzelknoten in dieser Reihenfolge verkettet.
- ❑ Beim Entnehmen eines Wurzelknotens muss die Liste seiner Nachfolger mit der verbleibenden Wurzelliste vereinigt werden.
Deshalb ist die Liste der Nachfolger eines Knotens ebenfalls nach aufsteigendem Grad sortiert verkettet.
Dies weicht von der Darstellung in Cormen et al. (vgl. § 1.3) ab, wo die Liste nach absteigendem Grad verkettet ist und deshalb vor einer Vereinigung erst einmal invertiert werden muss.
- ❑ Beim Zusammenfassen zweier Bäume mit dem gleichen Grad wird einer der Bäume zum Nachfolger mit dem größten Grad des anderen Baums, d. h. er muss am Ende der Nachfolgerliste angehängt werden.
Damit dies möglichst einfach und effizient geht, verweist `child` auf den Nachfolger mit dem größten Grad.
- ❑ Damit aber auch der Nachfolger mit dem kleinsten Grad direkt zugreifbar ist, ist die Liste der Nachfolger zirkulär verkettet.
- ❑ Für die Wurzelliste genügt jedoch eine einfache Verkettung.

3.4 Vergleich von binären und Binomial-Halden

- ❑ Die Laufzeit aller Operationen ist in beiden Implementierungen höchstens $O(\log_2 N)$.
- ❑ Bei der Implementierung mit binären Halden ist die Laufzeit der Operation „Auslesen“ sogar nur $O(1)$.
- ❑ Die Knoten von Binomial-Halden brauchen relativ viel Platz für Verwaltungsdaten (`degree`, `parent`, `child` und `sibling`), während binäre Halden nur die reinen Nutzdaten speichern.
- ❑ Binomial-Halden besitzen aber zwei Vorteile gegenüber binären Halden:
 - Ihre Kapazität ist prinzipiell nicht begrenzt.
 - Zwei Halden können auch effizient vereinigt werden.

4 Nächstbest-Algorithmen (greedy algorithms)

4.1 Optimierungsprobleme

- ❑ Ein *Optimierungsproblem* ist charakterisiert durch eine (häufig sehr große) Menge möglicher *Lösungen* und eine *Bewertungsfunktion*, die jeder Lösung einen Wert (z. B. eine natürliche oder reelle Zahl) zuordnet.
- ❑ Bei einem *Maximierungsproblem* wird eine Lösung mit maximalem Wert gesucht, bei einem *Minimierungsproblem* eine Lösung mit minimalem Wert.

Beispiele

- ❑ Finde in einem Straßennetz den kürzesten Weg von A nach B.
- ❑ Problem des Handlungsreisenden (traveling salesman problem):
Finde eine kürzeste Rundreise durch N Orte A_1, \dots, A_N .
- ❑ Blocksatz: Verteile eine Folge von Wörtern so auf Zeilen einer bestimmten Länge, dass der zum Auffüllen der Zeilen zusätzlich benötigte Zwischenraum möglichst gleichmäßig verteilt ist.
Problem: Was bedeutet „möglichst gleichmäßig“, d. h. wie lautet eine sinnvolle Bewertungsfunktion?

4.2 Nächstbest-Algorithmen

- ❑ Ein *Nächstbest-Algorithmus* konstruiert schrittweise eine Lösung eines Optimierungsproblems, indem er in jedem Schritt die nächstbeste Erweiterung der bis jetzt konstruierten Teillösung wählt.
- ❑ Wenn das so jeweils gewählte *lokale Optimum* auch ein *globales Optimum* darstellt, findet der Algorithmus tatsächlich eine optimale Lösung.
- ❑ Andernfalls stellt die gefundene Lösung nur eine mehr oder weniger gute *Näherungslösung* oder *Approximation* der optimalen Lösung dar, die aber oft mit wesentlich weniger Aufwand als die optimale Lösung ermittelt werden kann.

Beispiele

- ❑ Der Handlungsreisende besucht als nächstes immer diejenige bis jetzt noch nicht besuchte Stadt, die der momentan besuchten Stadt am nächsten liegt (lokales Optimum, nearest neighbour algorithm).

Auf diese Weise findet er für N Orte mit Aufwand $O(N^2)$ eine Rundreise, für die gezeigt werden kann, dass ihre Länge höchstens $\frac{\lceil \log_2 N \rceil + 1}{2}$ -mal so groß ist wie die einer kürzesten Rundreise (globales Optimum).

- Die meisten Textverarbeitungsprogramme füllen jede Ausgabezeile mit möglichst vielen Wörtern (lokales Optimum), ohne darauf zu achten, ob diese Strategie für nachfolgende Zeilen günstig ist oder nicht (globales Optimum), zum Beispiel:

Wenn man beim Blocksatz jede Zeile für sich betrachtet und mit möglichst vielen Wörtern füllt, wird dieser Text bei einer Zeilenbreite von 60 extrem schlecht umgebrochen. Schuld ist die URL https://en.wikipedia.org/wiki/Line_wrap_and_word_wrap, weil sie sehr lang ist und eigentlich nicht sinnvoll getrennt werden kann. Daher ist es besser, einen Absatz als Ganzes zu betrachten und den Zwischenraum möglichst gleichmäßig über alle Zeilen zu verteilen, wie es z. B. von TeX gemacht wird.

Besser wäre:

Wenn man beim Blocksatz jede Zeile für sich betrachtet und mit möglichst vielen Wörtern füllt, wird dieser Text bei einer Zeilenbreite von 60 extrem schlecht umgebrochen. Schuld ist die URL https://en.wikipedia.org/wiki/Line_wrap_and_word_wrap, weil sie sehr lang ist und eigentlich nicht sinnvoll getrennt werden kann. Daher ist es besser, einen Absatz als Ganzes zu betrachten und den Zwischenraum möglichst gleichmäßig über alle Zeilen zu verteilen, wie es z. B. von TeX gemacht wird.

4.3 Huffman-Kodierung

4.3.1 Idee

- Buchstaben in natürlichsprachlichen Texten sind nicht gleichverteilt.
 Für deutsche Texte gilt z. B. laut Wikipedia:

<i>Buchstabe</i>	<i>Häufigkeit</i>	<i>Buchstabe</i>	<i>Häufigkeit</i>	<i>Buchstabe</i>	<i>Häufigkeit</i>
E	17,40 %	U	4,35 %	K	1,21 %
N	9,78 %	L	3,44 %	Z	1,13 %
I	7,55 %	C	3,06 %	P	0,79 %
S	7,27 %	G	3,01 %	V	0,67 %
R	7,00 %	M	2,53 %	ß	0,31 %
A	6,51 %	O	2,51 %	J	0,27 %
T	6,15 %	B	1,89 %	Y	0,04 %
D	5,08 %	W	1,89 %	X	0,03 %
H	4,76 %	F	1,66 %	Q	0,02 %

- Wenn man häufige Buchstaben durch kürzere *Kodewörter* darstellt als seltene (*Kode variabler Länge*), brauchen Texte weniger Platz, als wenn alle Kodewörter gleich lang sind (*Kode fester Länge*), d. h. man erreicht eine *verlustfreie Kompression* (auch als *Entropiekodierung* bezeichnet).

- ❑ Um den kodierten Text wieder eindeutig dekodieren zu können, muss der Code jedoch *präfixfrei* sein, d. h. kein Kodewort darf ein Präfix eines anderen Kodeworts sein.

- ❑ Beispiel:
 - Um die o. g. 27 Buchstaben mit jeweils gleich vielen Bits zu kodieren, benötigt man 5 Bit pro Buchstabe.
 - Damit benötigt ein Text mit 10 000 Zeichen 50 000 Bit.
 - Mit einem für die obige Verteilung optimalen *Präfixkode* (vgl. § 4.3.2) bräuchte man im Durchschnitt nur etwa 41 000 Bit.
 - Mit einem Präfixkode, der optimal für die tatsächliche Buchstabenverteilung im betrachteten Text ist, kann man u. U. noch eine deutlich höhere Kompression erzielen.

- ❑ Nach dem gleichen Prinzip lassen sich auch binäre Daten komprimieren, indem man häufig auftretende Bitfolgen (z. B. Bytes oder Maschinenwörter) durch weniger Bits kodiert als seltene.

- ❑ Dies wird z. B. als letzter Schritt bei der (ansonsten normalerweise verlustbehafteten) JPEG-Komprimierung eingesetzt.

4.3.2 Binäre Präfixkodes

- ❑ Ein *binärer Kode* ordnet jedem Zeichen des Eingabealphabets ein aus Nullen und Einsen bestehendes *Kodewort* zu.
- ❑ Bei einem *Präfixkode* darf kein Kodewort ein Präfix eines anderen Kodeworts sein.
- ❑ Ein binärer Präfixkode kann durch einen binären Baum dargestellt werden, in dem jedem Knoten wie folgt ein Kodewort zugeordnet ist:
 - Das Kodewort des Wurzelknotens ist die leere Zeichenkette.
 - Das Kodewort des linken bzw. rechten Nachfolgers eines Knotens entsteht durch Anhängen von 0 bzw. 1 an das Kodewort dieses Knotens.
 - Jedes Zeichen des Eingabealphabets entspricht einem Blattknoten mit dem entsprechenden Kodewort.
- ❑ Kodierung einer Zeichenfolge
Für jedes Zeichen der Zeichenfolge:
Gib das zugehörige Kodewort aus.

❑ Dekodierung einer Bitfolge

- 1 Setze einen Zeiger auf den Wurzelknoten des Codebaums.
- 2 Für jedes Bit der Bitfolge:
 - 1 Wenn das Bit gleich 0 bzw. 1 ist, setze den Zeiger auf den linken bzw. rechten Nachfolger des aktuellen Knotens.
 - 2 Wenn dieser Knoten ein Blattknoten ist:
Gib das zugehörige Zeichen des Eingabealphabets aus
und setze den Zeiger wieder auf den Wurzelknoten.

❑ Beispiel

- Präfixkode als Baum
- Kodierung einer Zeichenfolge
- Dekodierung einer Bitfolge

4.3.3 Optimale binäre Präfixkodes

- Gegeben sei ein Eingabealphabet Σ und eine zu kodierende Zeichenfolge $s \in \Sigma^*$.
- Für jedes Zeichen $c \in \Sigma$ sei $f_s(c)$ die Häufigkeit von c in s .
- Für einen Code bzw. Codebaum T sei $d_T(c)$ die Länge des Kodeworts von c in diesem Code, d. h. die Tiefe des zugehörigen Blattknotens im Baum T .
- Wenn die Zeichenfolge s mit dem Code T kodiert wird, ergibt sich eine Bitfolge der Länge $L_T(s) = \sum_{c \in \Sigma} f_s(c) d_T(c)$.
- Ein Code T ist *optimal* für die Zeichenfolge s , wenn $L_T(s)$ minimal ist, d. h. wenn $L_T(s) \leq L_{T'}(s)$ für alle Codes T' .
- Frage: Wie findet man einen solchen optimalen Code?

Lemma 1

- Wenn x und y die Zeichen in Σ mit der kleinsten Häufigkeit sind, dann gibt es einen optimalen Präfixkode, in dem sich die Kodewörter von x und y nur in ihrem letzten Bit unterscheiden, d. h. im Kodebaum sind die zu x und y gehörenden Blattknoten Geschwister.

Beweisidee

- Überführe irgendeinen optimalen Kodebaum T in einen ebenfalls optimalen Kodebaum T'' , der die behauptete Eigenschaft besitzt.

Beweis

- Sei T irgendein optimaler Kodebaum.
- Betrachte in diesem Baum zwei Geschwisterblätter a und b mit maximaler Tiefe, d. h. $d_T(a) = d_T(b) \geq d_T(z)$ für alle $z \in \Sigma$.
- O. B. d. A. gilt: $f_s(a) \leq f_s(b)$ und $f_s(x) \leq f_s(y)$.
(Andernfalls vertausche a und b bzw. x und y .)

□ Da x und y die Zeichen in Σ mit der kleinsten Häufigkeit sind, folgt:
 $f_s(x) \leq f_s(a)$ und $f_s(y) \leq f_s(b)$.

□ Sei T' der Kodebaum, der aus T durch Vertauschen der Blattknoten a und x entsteht.

□ Somit gilt:

$$L_T(s) - L_{T'}(s) = \sum_{c \in \Sigma} f_s(c) d_T(c) - \sum_{c \in \Sigma} f_s(c) d_{T'}(c) =$$

$$f_s(x) d_T(x) + f_s(a) d_T(a) - f_s(x) d_{T'}(x) - f_s(a) d_{T'}(a) =$$

$$f_s(x) d_T(x) + f_s(a) d_T(a) - f_s(x) d_T(a) - f_s(a) d_T(x) =$$

$$(f_s(a) - f_s(x)) (d_T(a) - d_T(x)) \geq 0$$

wegen $f_s(a) \geq f_s(x)$ und $d_T(a) \geq d_T(x)$.

□ Daraus folgt: $L_T(s) \geq L_{T'}(s)$.

□ Da T optimal ist, gilt jedoch auch: $L_T(s) \leq L_{T'}(s)$.

□ Also gilt: $L_T(s) = L_{T'}(s)$, d. h. auch T' ist ein optimaler Kode.

- ❑ Sei nun T'' der Kodebaum, der aus T' durch Vertauschen der Blattknoten b und y entsteht.
- ❑ Für diesen gilt analog: $L_{T''}(s) = L_{T'}(s) = L_T(s)$, d. h. T'' ist ein optimaler Kodebaum.
- ❑ Da a und b im ursprünglichen Kodebaum T Geschwisterblätter sind, sind x und y im resultierenden Kodebaum T'' ebenfalls Geschwisterblätter, sodass T'' tatsächlich die behauptete Eigenschaft besitzt.

Lemma 2

- x und y seien wieder die Zeichen in Σ mit der kleinsten Häufigkeit.
- Das Alphabet Σ' sei definiert als $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$,
wobei $z \notin \Sigma$ ein künstliches neues Zeichen mit Häufigkeit $f_{s'}(z) = f_s(x) + f_s(y)$ sei,
d. h. gedanklich ersetzt man jedes Auftreten von x und y in s durch z .
- Sei T' ein optimaler Kodebaum für die so modifizierte Zeichenfolge s'
und T der Baum, der aus T' entsteht, wenn man den Blattknoten z durch einen
inneren Knoten ersetzt, der die Blattknoten x und y als Nachfolger besitzt.
- Dann ist T ein optimaler Kodebaum für die Zeichenfolge s .

Beweis:

- Aufgrund der Konstruktion von T aus T' gilt:
 - $d_T(x) = d_T(y) = d_{T'}(z) + 1$
 - $d_T(c) = d_{T'}(c)$ für alle $c \in \Sigma \setminus \{x, y\} = \Sigma' \setminus \{z\}$

□ Daraus folgt:

$$L_T(s) - L_{T'}(s') = \sum_{c \in \Sigma} f_s(c) d_T(c) - \sum_{c \in \Sigma'} f_{s'}(c) d_{T'}(c) =$$

$$f_s(x) d_T(x) + f_s(y) d_T(y) - f_{s'}(z) d_{T'}(z) =$$

$$(f_s(x) + f_s(y)) (d_{T'}(z) + 1) - f_{s'}(z) d_{T'}(z) =$$

$$f_{s'}(z) (d_{T'}(z) + 1) - f_{s'}(z) d_{T'}(z) = f_{s'}(z)$$

□ Also gilt: $L_T(s) = L_{T'}(s') + f_{s'}(z)$

□ Sei nun \tilde{T} ein optimaler Kodebaum für s ,
 der o. B. d. A. die Eigenschaft von Lemma 1 erfüllt,
 und \tilde{T}' der Baum, der aus \tilde{T} entsteht, wenn man die Blattknoten x und y und ihren
 gemeinsamen Vorgänger durch einen neuen Knoten z mit Häufigkeit
 $f_{s'}(z) = f_s(x) + f_s(y)$ ersetzt.

□ Dann gilt entsprechend aufgrund der Konstruktion von \tilde{T}' aus \tilde{T} :

$$L_{\tilde{T}}(s) = L_{\tilde{T}'}(s') + f_{s'}(z).$$

□ Da \tilde{T} und \tilde{T}' jeweils optimal sind, folgt daraus:

$$L_{\tilde{T}}(s) \leq L_T(s) = L_{T'}(s') + f_{s'}(z) \leq L_{\tilde{T}'}(s') + f_{s'}(z) = L_{\tilde{T}}(s)$$

□ Also gilt: $L_T(s) = L_{\tilde{T}}(s)$, d. h. T ist ein optimaler Kodebaum.

Rekursiver Algorithmus zur Bestimmung eines optimalen Codes

- 1 Wenn das Alphabet Σ genau zwei Zeichen x und y enthält:
Ordne x das Kodewort 0 und y das Kodewort 1 zu.
- 2 Andernfalls (d. h. wenn Σ mehr als zwei Zeichen enthält):
 - 1 Ermittle die Zeichen x und y mit der kleinsten Häufigkeit.
 - 2 Erzeuge ein künstliches neues Zeichen z mit Häufigkeit $f(z) = f(x) + f(y)$.
 - 3 Führe den Algorithmus rekursiv für das Alphabet $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$ aus.
 - 4 Ordne x bzw. y das Kodewort von z gefolgt von 0 bzw. 1 zu und übernehme die Kodewörter für alle anderen Zeichen aus $\Sigma \setminus \{x, y\} = \Sigma' \setminus \{z\}$.

Anmerkungen

- Die Korrektheit des Algorithmus folgt direkt aus Lemma 2.
- Problem: Wie findet man effizient die Zeichen mit der kleinsten Häufigkeit?
- Die Implementierung ist aufwendig, weil für jeden rekursiven Aufruf ein neues Alphabet Σ' und eine modifizierte Häufigkeitsverteilung konstruiert werden muss.

Iterativer Nächstbest-Algorithmus mit Vorrangwarteschlange

- 1 Für jedes Zeichen $c \in \Sigma$:
Erzeuge einen zugehörigen Blattknoten
und füge ihn mit Priorität $f(c)$ in eine Minimum-Vorrangwarteschlange ein.
- 2 Solange die Warteschlange mindestens zwei Elemente enthält:
 - 1 Entnimm die Knoten x und y mit minimaler Priorität/Häufigkeit.
 - 2 Erzeuge einen inneren Knoten mit x und y als Nachfolgern
und füge ihn mit Priorität $f(x) + f(y)$ in die Warteschlange ein.
- 3 Entnimm den verbleibenden Knoten aus der Warteschlange,
der den Wurzelknoten des gesamten Codebaums darstellt.
- 4 Weise jedem Zeichen $c \in \Sigma$ sein Kodewort anhand dieses Baums zu (vgl. § 4.3.2).

Laufzeit des iterativen Algorithmus

- Bei geeigneter Implementierung der Vorrangwarteschlange (vgl. Kapitel 3) hat der Algorithmus die Laufzeit $O(N \log_2 N)$, wenn N die Größe des Alphabets Σ bezeichnet.

Beispiel: $f(A) = 1$, $f(B) = 2$, $f(C) = 3$, $f(D) = 4$, $f(E) = 5$, $f(F) = 6$

4.4 Das Problem der stabilen Ehen (stable marriage problem)

4.4.1 Problemstellung

- Gegeben seien N Männer und N Frauen.
- Die *Präferenzliste* einer Person ist eine Permutation der Personen des anderen Geschlechts.
- Eine Person P *bevorzugt* eine Person Q des anderen Geschlechts gegenüber einer weiteren Person R des anderen Geschlechts (in Zeichen: $Q \underset{P}{<} R$), wenn Q in der Präferenzliste von P vor R steht.
- Eine *Zuordnung* ist eine bijektive Funktion, die jeder Person P eineindeutig einen Partner P° des anderen Geschlechts zuordnet.
- Eine Zuordnung ist *labil*, wenn es einen Mann M und eine Frau F gibt, die zusammen *die Stabilität der Zuordnung gefährden*, das heißt:
 - $F \underset{M}{<} M^\circ$, d. h. M bevorzugt F gegenüber der ihm zugeordneten Frau M°
 - $M \underset{F}{<} F^\circ$, d. h. F bevorzugt M gegenüber dem ihr zugeordneten Mann F°
 Andernfalls ist die Zuordnung *stabil*.
- Gesucht ist eine stabile Zuordnung.

4.4.2 Lösung mit Nächstbest-Algorithmus

Herrenwahl

- 1 Am Anfang sind alle Personen solo
- 2 Solange es einen Mann M gibt, der noch oder wieder solo ist:
 - 1 M fragt die erste Frau F auf seiner Präferenzliste, die er noch nicht gefragt hat
 - 2 Wenn F noch solo ist:
 M und F bilden ein (vorläufiges) Paar, d. h. beide sind dann nicht mehr solo
 - 3 Andernfalls hat F bereits einen (vorläufigen) Mann \tilde{M}
Wenn $M \underset{F}{<} \tilde{M}$:
 - 1 F trennt sich von \tilde{M} , d. h. anschließend ist \tilde{M} wieder solo
 - 2 M und F bilden ein neues (vorläufiges) Paar

Damenwahl

Analog mit vertauschten Rollen

4.4.3 Beispiel

Anton	Christian	Emil	Gustav
1. Hanna 2. Berta 3. Doris 4. Frieda	1. Hanna 2. Berta 3. Frieda 4. Doris	1. Doris 2. Frieda 3. Berta 4. Hanna	1. Berta 2. Hanna 3. Frieda 4. Doris

Berta	Doris	Frieda	Hanna
1. Christian 2. Anton 3. Gustav 4. Emil	1. Anton 2. Christian 3. Gustav 4. Emil	1. Gustav 2. Anton 3. Christian 4. Emil	1. Emil 2. Christian 3. Anton 4. Gustav

4.4.4 Korrektheit und weitere Eigenschaften

- Die folgenden Aussagen gelten für Herrenwahl.
- Wenn man die Rollen von Männern und Frauen vertauscht, erhält man entsprechende Aussagen für Damenwahl.

Behauptung 1

- Es gibt keinen Mann, der von jeder Frau abgewiesen wird.

Beweis

- Wenn ein Mann von einer Frau abgewiesen wird, dann hat diese in diesem Moment einen anderen Mann.
- Sobald eine Frau einmal einen Mann hat, hat sie bis zum Ende immer einen Mann.
- Wenn ein Mann von allen Frauen abgewiesen werden würde, dann wäre er am Ende solo, während jede Frau einen Mann haben müsste (weil sie den fraglichen Mann abgewiesen hat).
- Widerspruch, da es gleich viele Männer und Frauen gibt.

Folgerungen

- ❑ Der Algorithmus terminiert:
Spätestens nach N „Versuchen“ hat jeder Mann eine Frau, die er nicht wieder verliert, d. h. er ist dann dauerhaft nicht mehr solo.
- ❑ Der Algorithmus ist wohldefiniert:
Wenn ein Mann solo ist, dann gibt es noch mindestens eine Frau auf seiner Präferenzliste, die er noch nicht gefragt hat.
- ❑ Am Ende hat jeder Mann eine Frau und damit auch jede Frau einen Mann, d. h. der Algorithmus ermittelt wirklich eine Zuordnung.

Behauptung 2

□ Der Algorithmus ermittelt eine stabile Zuordnung.

Beweis

□ Nach Ausführung des Algorithmus gilt für jeden Mann M und jede Frau F :

○ Wenn $F \underset{M}{\geq} M^\circ$ gilt, dann stellen M und F keine Gefahr für die Stabilität der Zuordnung dar.

○ Wenn $F \underset{M}{<} M^\circ$ gilt, dann hat M F vor M° gefragt und wurde offenbar (sofort oder später) wegen eines anderen Manns \tilde{M} mit $\tilde{M} \underset{F}{<} M$ abgewiesen.

○ Da sich eine Frau während der Durchführung des Algorithmus nicht mehr „verschlechtern“ kann, gilt außerdem $F^\circ \underset{F}{\leq} \tilde{M}$ und somit $F^\circ \underset{F}{<} M$.

○ Also stellen M und F auch in diesem Fall keine Gefahr für die Stabilität der Zuordnung dar.

□ Also ist die Zuordnung stabil.

Folgerung

□ Es gibt immer eine stabile Zuordnung.

Definition

- Für eine Person P sei P^+ der beste und P^- der schlechteste Partner, den P bei einer stabilen Zuordnung bekommen kann, das heißt:
 - Es gibt eine stabile Zuordnung, bei der $P^\circ = P^+$ ist.
 - Es gibt eine (möglicherweise andere) stabile Zuordnung, bei der $P^\circ = P^-$ ist.
 - Für jede stabile Zuordnung gilt: $P^+ \underset{P}{\leq} P^\circ \underset{P}{\leq} P^-$.

Behauptung 3

- Der Algorithmus ordnet jedem Mann M seine optimale Frau M^+ zu.

Beweis durch Widerspruch

- Annahme: Es gibt einen Mann M , dem vom Algorithmus eine Frau M° mit $M^\circ \underset{M}{>} M^+$ zugeordnet wird.
- Das heißt: M wird während der Durchführung des Algorithmus von seiner optimalen Frau M^+ wegen eines anderen Manns \tilde{M} mit $\tilde{M} \underset{M^+}{<} M$ abgewiesen, d. h. \tilde{M} und M^+ bilden in diesem Moment ein Paar.

- Wenn es mehrere Männer gibt, denen das passiert, sei M derjenige, dem dies während der Durchführung des Algorithmus als erstem passiert.
- \tilde{M} wurde bis dahin genau von den Frauen F abgewiesen, für die $F <_{\tilde{M}} M^+$ gilt.
- Da M der erste Mann ist, der von seiner optimalen Frau abgewiesen wird, wurde \tilde{M} von seiner optimalen Frau \tilde{M}^+ noch nicht abgewiesen, d. h. für \tilde{M}^+ gilt nicht $\tilde{M}^+ <_{\tilde{M}} M^+$, sondern vielmehr $\tilde{M}^+ \geq_{\tilde{M}} M^+$.
- Sei Z irgendeine stabile Zuordnung, bei der M seine optimale Frau M^+ bekommt, und sei \tilde{F} die Frau von \tilde{M} bei dieser Zuordnung.
- Für diese Frau \tilde{F} gilt natürlich: $\tilde{F} \geq_{\tilde{M}} \tilde{M}^+$.
- Aus $\tilde{F} \geq_{\tilde{M}} \tilde{M}^+$ und $\tilde{M}^+ \geq_{\tilde{M}} M^+$ folgt insgesamt: $\tilde{F} \geq_{\tilde{M}} M^+$.
- Da \tilde{F} in der Zuordnung Z zu \tilde{M} und M^+ zu M gehört, muss $\tilde{F} \neq M^+$ sein und somit $\tilde{F} >_{\tilde{M}} M^+$ gelten.

- Da außerdem $\tilde{M} <_{M^+} M$ gilt (siehe oben), stellen \tilde{M} und M^+ eine Gefahr für die Stabilität der Zuordnung Z dar.
- Dies steht im Widerspruch dazu, dass Z eine stabile Zuordnung ist.

Folgerung

- Da der Algorithmus jedem Mann – unabhängig von der Reihenfolge, in der die Männer durchlaufen werden – immer seine optimale Frau zuordnet, spielt die Reihenfolge offenbar keine Rolle.

Behauptung 4

- Der Algorithmus ordnet jeder Frau F den schlechtestmöglichen Mann F^- zu.

Beweis durch Widerspruch

- Annahme: Es gibt eine Frau F , der vom Algorithmus ein Mann $\tilde{M} = F^\circ$ mit $\tilde{M} <_F F^-$ zugeordnet wird.
- Sei Z eine stabile Zuordnung, bei der F und F^- ein Paar bilden, und sei \tilde{F} die Frau von \tilde{M} bei dieser Zuordnung.
- Da der Algorithmus jedem Mann seine optimale Frau zuordnet, gilt: $F = \tilde{M}^\circ <_{\tilde{M}} \tilde{F}$.
- Damit stellen F und \tilde{M} eine Gefahr für die Stabilität der Zuordnung Z dar.
- Dies steht im Widerspruch dazu, dass Z eine stabile Zuordnung ist.

4.4.5 Laufzeit

- ❑ Jeder der N Männer fragt im schlimmsten Fall jede der N Frauen.
- ❑ Daher beträgt die Laufzeit des Algorithmus $O(N^2)$.

4.4.6 Anwendungsbeispiel

- ❑ Gegeben seien Projekte P_1, \dots, P_K mit Mitarbeiterbedarf N_1, \dots, N_K sowie $N = N_1 + \dots + N_K$ Mitarbeiter.
- ❑ Jeder Mitarbeiter erstellt eine Präferenzliste der Projekte (d. h. eine Permutation von P_1, \dots, P_K).
- ❑ Jedes Projekt P_i wird durch N_i „Stroh Männer“ P_{ij} ($j = 1, \dots, N_i$) repräsentiert, die alle die gleiche Präferenzliste der Mitarbeiter (d. h. eine Permutation von M_1, \dots, M_N) haben, die z. B. vom Projektleiter erstellt wird.
- ❑ In den Präferenzlisten der Mitarbeiter wird jedes Projekt P_i durch die Folge der zugehörigen Stroh Männer ersetzt.
- ❑ Dann kann der Algorithmus mit den N Mitarbeitern und den N Stroh Männern ausgeführt werden, um eine möglichst gute Zuordnung der Mitarbeiter zu den Projekten zu ermitteln.
- ❑ Je nachdem, ob die Mitarbeiter oder die Stroh Männer die Rolle der Männer spielen, erhält man entweder eine für die Mitarbeiter oder für die Projekte (bzw. Projektleiter) optimale Zuordnung.

5 Graphalgorithmen

5.1 Definitionen

5.1.1 Gerichtete und ungerichtete Graphen

- ❑ Ein *Graph* ist ein Paar $G = (V, E)$ mit einer endlichen Menge V von *Knoten* oder *Ecken* (engl. vertices) und einer zugehörigen Menge E von *Kanten* (engl. edges).
- ❑ Wenn E eine Teilmenge von $V \times V = \{(u, v) \mid u, v \in V\}$ ist, heißt der Graph *gerichtet*. (Eine Kante $(u, v) \in E$ geht von u nach v , aber nicht umgekehrt.)
- ❑ Wenn E eine Teilmenge von $\{\{u, v\} \mid u, v \in V\}$ ist, heißt der Graph *ungerichtet*. (Eine Kante $\{u, v\} = \{v, u\} \in E$ geht sowohl von u nach v als auch umgekehrt.)
- ❑ Ein ungerichteter Graph kann auch als gerichteter Graph aufgefasst werden, in dem es zu jeder Kante (u, v) auch die entgegengesetzte Kante (v, u) gibt.
- ❑ Eine Kante (v, v) oder $\{v, v\} = \{v\}$ von einem Knoten v zu sich selbst heißt *Schlinge*.

5.1.2 Adjazenzlisten und -matrizen

- ❑ Wenn es in einem Graphen eine Kante von einem Knoten u zu einem Knoten v gibt, heißt u *Vorgänger* von v und v *Nachfolger* von u .
- ❑ Die *Adjazenzliste* eines Knotens enthält alle Nachfolger dieses Knotens in irgendeiner Reihenfolge.
- ❑ Die *Adjazenzlistendarstellung* eines Graphen besteht aus den Adjazenzlisten aller Knoten des Graphen.
- ❑ Die *Adjazenzmatrix* eines Graphen mit $N = |V|$ Knoten ist eine Matrix A mit $N \times N$ Elementen a_{uv} . Jedes a_{uv} ist 1, wenn es eine Kante von u nach v gibt, andernfalls 0.

$$\text{Formal: } A: V \times V \rightarrow \{0, 1\} \text{ mit } A(u, v) = \begin{cases} 1, & \text{wenn } (u, v) \in E \text{ bzw. } \{u, v\} \in E \\ 0 & \text{sonst} \end{cases}$$

$$\text{Folgerung: } |E| \leq |V|^2$$

- ❑ Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch.
- ❑ Die Adjazenzlistendarstellung eines Graphen $G = (V, E)$ hat die Größe $O(|V| + |E|)$, die Adjazenzmatrix $O(|V|^2)$.

5.1.3 Weitere Begriffe

- ❑ Ein *Weg* oder *Pfad* von einem Knoten u zu einem Knoten v ist eine Folge paarweise verschiedener Knoten w_0, \dots, w_n mit $u = w_0$, Kanten von w_{i-1} nach w_i für $i = 1, \dots, n$ und $w_n = v$.
- ❑ Die Anzahl n der Kanten heißt *Länge* des Wegs.
(Der Fall $n = 0$ als *leerer Weg* von einem Knoten zu sich selbst ist zulässig.)
- ❑ Wenn es einen Weg von u nach v gibt, heißt v von u aus *erreichbar*.
(Insbesondere ist jeder Knoten von sich selbst aus erreichbar.)
- ❑ Die *Distanz* $\delta(u, v)$ zwischen u und v ist entweder die Länge eines kürzesten Wegs von u nach v , falls v von u aus erreichbar ist, oder andernfalls ∞ .
(Insbesondere ist $\delta(v, v) = 0$ für jeden Knoten $v \in V$.)
- ❑ Wenn w_0, \dots, w_n ein Weg ist und es eine Kante von w_n nach w_0 gibt, heißt die Knotenfolge w_0, \dots, w_n, w_0 *Zyklus* oder *Kreis*.
(Beachte: Die Knotenfolge w_0, \dots, w_n, w_0 ist kein Weg, weil ihre Knoten nicht alle verschieden sind.)
- ❑ Ein Graph ohne Zyklen heißt *azyklisch* oder *zyklenfrei*.

- ❑ Eine *Verbindung* zweier Knoten u und v ist eine Folge paarweise verschiedener Knoten w_0, \dots, w_n mit $u = w_0$, Kanten von w_{i-1} nach w_i oder umgekehrt für $i = 1, \dots, n$ und $w_n = v$. (In einem ungerichteten Graphen sind Verbindung und Weg gleichbedeutend.)
- ❑ Ein Graph heißt *zusammenhängend*, wenn es von jedem Knoten eine Verbindung zu jedem anderen Knoten gibt.
- ❑ Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es von jedem Knoten einen Weg zu jedem anderen Knoten gibt, d. h. wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- ❑ Für einen zusammenhängenden Graphen gilt: $|E| \geq |V| - 1$.
- ❑ Ein zusammenhängender Graph $G = (V, E)$ heißt *Baum*, wenn $|E| = |V| - 1$ gilt. (Zu jedem Knoten außer der Wurzel gibt es genau eine Kante, die ihn mit seinem übergeordneten Knoten verbindet. Je nach Art des Baums, können diese Kanten ungerichtet, „von oben nach unten“ gerichtet oder „von unten nach oben“ gerichtet sein. Bei einem ungerichteten Baum kann jeder Knoten die Rolle des Wurzelknotens spielen.)
- ❑ Eine Menge von Bäumen heißt *Wald*.

Alternative Definition von Bäumen und Wäldern

- ❑ Ein Graph ohne Schlingen heißt Baum, wenn es zwischen je zwei Knoten genau eine Verbindung gibt.
- ❑ Ein Graph ohne Schlingen heißt Wald, wenn es zwischen je zwei Knoten höchstens eine Verbindung gibt.

Anmerkungen zur Definition von Zyklen

- ❑ Aus der obigen Definition eines Zyklus folgt, dass ein ungerichteter Graph mit mindestens einer Kante $\{u, v\} = \{v, u\}$ immer Zyklen u, v, u und v, u, v enthält.
- ❑ Damit ist die Charakterisierung eines Baums als zusammenhängender, azyklischer Graph (vgl. Wikipedia) für ungerichtete Graphen falsch.
- ❑ Um solche „unerwünschten“ Zyklen zu vermeiden, könnte man zusätzlich verlangen, dass die Kanten auf einem Zyklus paarweise verschieden sind.
- ❑ Dann bestünde jedoch ein feiner Unterschied zwischen einem ungerichteten Graphen und dem entsprechenden gerichteten Graphen, in dem es zu jeder Kante auch die entgegengesetzte Kante gibt: Der ungerichtete Graph könnte dann azyklisch sein, der gerichtete Graph wäre jedoch immer zyklisch (sofern es mindestens eine Kante gibt).

5.2 Breitensuche (breadth-first search)

5.2.1 Problemstellung

Gegeben

- Graph $G = (V, E)$
- Startknoten $s \in V$

Gesucht

- Alle Knoten $v \in V$, die vom Startknoten s aus erreichbar sind

Genauer

- $\delta(s, v)$ für alle Knoten $v \in V$
- Ein kürzester Weg von s nach v für alle von s aus erreichbaren Knoten $v \in V$

5.2.2 Algorithmus

- 1 Für jeden Knoten $v \in V \setminus \{s\}$:
Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
- 2 Setze $\delta(s) = 0$ und $\pi(s) = \perp$.
- 3 Füge s in eine FIFO-Warteschlange ein.
- 4 Solange die Warteschlange nicht leer ist:
 - 1 Entnimm den ersten Knoten u aus der Warteschlange.
 - 2 Für jeden Nachfolger v von u :
Wenn $\delta(v) = \infty$ ist:
 - 1 Setze $\delta(v) = \delta(u) + 1$ und $\pi(v) = u$.
 - 2 Füge v am Ende der Warteschlange an.

5.2.3 Laufzeit

- ❑ Initialisierung (Schritt 1 und 2): $O(|V|)$
- ❑ Eigentliche Suche (Schritt 3 und 4): $O(|V| + |E|)$
 - Die äußere Schleife wird für jeden Knoten höchstens einmal durchlaufen, da jeder Knoten höchstens einmal in die Warteschlange eingefügt wird.
 - Damit wird die innere Schleife insgesamt höchstens $\sum_{u \in V} \chi(u) = |E|$ mal durchlaufen, wobei $\chi(u) = \text{Grad des Knotens } u = \text{Anzahl der Nachfolger von } u = \text{Anzahl der von } u \text{ ausgehenden Kanten}$.
- ❑ Insgesamt also: $O(|V| + |E|)$

5.2.4 Ergebnis

Nach Ausführung des Algorithmus gilt:

- ❑ $\delta(v) = \delta(s, v)$ für alle $v \in V$
- ❑ Wenn $v \neq s$ von s aus erreichbar ist, dann ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v , das heißt: Der kürzeste Weg von s nach v lautet w_n, \dots, w_0 mit $w_0 = v$ und $w_i = \pi(w_{i-1})$ für $i = 1, \dots, n$ mit $n = \delta(v)$.

5.2.5 Vorgängergraph einer Breitensuche (Breitensuchebaum)

□ Der Graph $G_\pi = (V_\pi, E_\pi)$ mit

○ $V_\pi = \{v \in V \mid \delta(v) < \infty\}$ = Menge aller vom Startknoten s aus erreichbaren Knoten

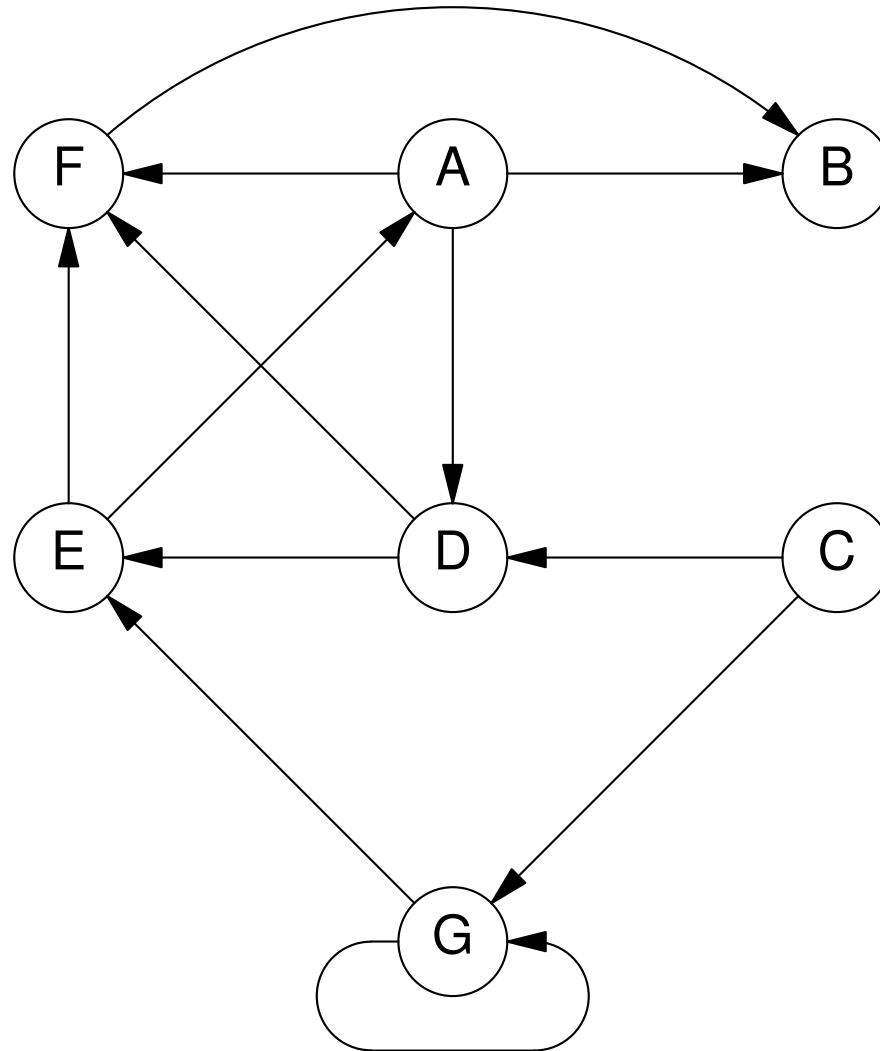
○ $E_\pi = \{(\pi(v), v) \mid v \in V_\pi \setminus \{s\}\} = \{(\pi(v), v) \mid \pi(v) \neq \perp\} \subseteq E$

ist ein Baum mit Wurzel s ,

der alle von s aus erreichbaren Knoten des Graphen G enthält.

□ Auf Ebene d dieses Baums befinden sich alle Knoten v mit $\delta(s, v) = d$.

5.2.6 Beispiel



5.2.7 Anwendungsbeispiel

Automatische Speicherbereinigung (garbage collection), z. B. in Java

- ❑ $V = \{v \mid \text{Objekt } v \text{ wurde mit } \texttt{new} \text{ erzeugt}\}$
- ❑ $E = \{(u, v) \in V \times V \mid \text{eine Objektvariable von } u \text{ verweist auf } v\}$
- ❑ $S = \{s \in V \mid \text{eine Klassenvariable oder eine lokale Variable verweist auf Objekt } s\}$
- ❑ Die Breitensuche wird (nach einmaliger Initialisierung aller Knoten/Objekte) für alle Objekte $s \in S$ nacheinander ausgeführt.
- ❑ $\pi(v)$ wird nicht benötigt.
- ❑ Statt $\delta(v)$ genügt die binäre Information, ob v von irgendeinem Objekt $s \in S$ aus erreichbar ist oder nicht.
- ❑ Objekte, die auf diese Weise nicht erreichbar sind, können gelöscht werden.

5.3 Tiefensuche (depth-first search) und topologische Sortierung

5.3.1 Problemstellung

- ❑ Ordne die Knoten eines Graphen $G = (V, E)$ von links nach rechts in „Hierarchien“ an, sodass keine Kanten von links nach rechts zeigen.
- ❑ Nebenbei kann überprüft werden, ob der Graph einen Zyklus enthält oder nicht.
- ❑ Wenn er azyklisch ist, können die Knoten auch sequentiell von links nach rechts angeordnet werden, ohne dass Kanten von links nach rechts zeigen (*topologische Sortierung*).
- ❑ Während der Suche erhält jeder Knoten $v \in V$ einen *Vorgänger* $\pi(v)$ im *Tiefensuchewald* sowie eine *Entdeckungszeit* $\delta(v) \in \mathbb{N}$ und eine *Abschlusszeit* $\phi(v) \in \mathbb{N}$ mit $1 \leq \delta(v) < \phi(v) \leq 2 |V|$.
- ❑ Damit besitzt jeder Knoten außerdem zu jedem Zeitpunkt eine *Farbe*:
 - Ein Knoten ist *weiß*, wenn er noch nicht entdeckt wurde, d. h. wenn er noch keine Entdeckungs- und Abschlusszeit besitzt.
 - Ein Knoten ist *grau*, wenn er gerade bearbeitet wird, d. h. wenn er eine Entdeckungs-, aber noch keine Abschlusszeit besitzt.
 - Ein Knoten ist *schwarz*, wenn seine Bearbeitung abgeschlossen ist, d. h. wenn er sowohl eine Entdeckungs- als auch eine Abschlusszeit besitzt.

5.3.2 Algorithmus

Für jeden Knoten $u \in V$:

Wenn u weiß ist:

- 1 Setze $\pi(u) = \perp$.
- 2 Durchsuche den zu u gehörenden Teilgraphen, das heißt:
 - 1 Setze $\delta(u)$ auf den nächsten Zeitwert aus der Menge $\{1, \dots, 2 \cdot |V|\}$.
 - 2 Für jeden Nachfolger v von u :

Wenn v weiß ist:

- 1 Setze $\pi(v) = u$.
- 2 Durchsuche rekursiv den zu v gehörenden Teilgraphen.
- 3 Setze $\varphi(u)$ auf den nächsten Zeitwert aus der Menge $\{1, \dots, 2 \cdot |V|\}$.

5.3.3 Laufzeit

- ❑ Die äußere Schleife wird für jeden Knoten genau einmal durchlaufen.
- ❑ Die rekursive Operation „Durchsuchen“ wird für jeden Knoten genau einmal ausgeführt.
- ❑ Damit wird die innere Schleife in dieser Operation insgesamt $\sum_{u \in V} \chi(u) = |E|$ mal durchlaufen.
- ❑ Damit ergibt sich als Laufzeit: $O(|V| + |E|)$

5.3.4 Vorgängergraph einer Tiefensuche (Tiefensuchewald)

- ❑ Der Graph $G_\pi = (V, E_\pi)$ mit $E_\pi = \{(\pi(v), v) \mid v \in V, \pi(v) \neq \perp\} \subseteq E$ ist eine Menge von Bäumen, d. h. ein Wald.
- ❑ Jeder Knoten v mit $\pi(v) = \perp$ ist der Wurzelknoten eines dieser Bäume.

5.3.5 Klassifikation der Kanten

Nach der Ausführung einer Tiefensuche gilt für die Bearbeitungszeiträume $\Sigma(u) = [\delta(u), \varphi(u)]$ und $\Sigma(v) = [\delta(v), \varphi(v)]$ zweier beliebiger Knoten $u, v \in V$ mit $u \neq v$ immer genau eine der folgenden Beziehungen:

- $\Sigma(u) \supset \Sigma(v)$, d. h. $\delta(u) < \delta(v) < \varphi(v) < \varphi(u)$
 - v wurde während der Bearbeitung von u entdeckt.
 - Damit ist v ein direkter oder indirekter Nachfolger von u in einem Baum des Tiefensuchewalds.
 - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von oben nach unten und wird deshalb entweder als *Baumkante* (tree edge) – wenn sie zur Menge E_π gehört – oder andernfalls als *Abwärts-* oder *Vorwärtskante* (forward edge) bezeichnet.

- $\Sigma(u) \subset \Sigma(v)$, d. h. $\delta(v) < \delta(u) < \varphi(u) < \varphi(v)$
 - u wurde während der Bearbeitung von v entdeckt.
 - Damit ist v ein direkter oder indirekter Vorgänger von u in einem Baum des Tiefensuchewalds.
 - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von unten nach oben und wird deshalb als *Aufwärts-* oder *Rückwärtskante* (back edge) bezeichnet.
 - Eine Schlinge wird ebenfalls so bezeichnet.

- $\Sigma(u) > \Sigma(v)$, d. h. $\delta(v) < \varphi(v) < \delta(u) < \varphi(u)$
 - u wurde nach der Bearbeitung von v entdeckt.
 - Damit ist v weder ein Nachfolger noch ein Vorgänger von u in einem Baum des Tiefensuchewalds.
 - Wenn es eine Kante von u nach v gibt, geht sie im Tiefensuchewald von rechts nach links und wird deshalb als *Querkante* (cross edge) bezeichnet.
- $\Sigma(u) < \Sigma(v)$, d. h. $\delta(u) < \varphi(u) < \delta(v) < \varphi(v)$
 - v wurde nach der Bearbeitung von u entdeckt.
 - Damit ist v weder ein Nachfolger noch ein Vorgänger von u in einem Baum des Tiefensuchewalds.
 - Wenn es eine Kante von u nach v gäbe, müsste sie im Tiefensuchewald von links nach rechts gehen.
 - Aber wenn es eine solche Kante gäbe, dann hätte der Algorithmus den Knoten v spätestens während der Bearbeitung von u über diese Kante entdeckt, d. h. dann würde entweder $\Sigma(v) < \Sigma(u)$ (Entdeckung und Bearbeitung von v bereits vor der Bearbeitung von u) oder $\Sigma(v) \subset \Sigma(u)$ (Entdeckung und Bearbeitung von v während der Bearbeitung von u) gelten.
 - Daher gibt es im Tiefensuchewald niemals Kanten von links nach rechts.

Aufgrund der Rekursionsstruktur des Algorithmus ist eine Überlappung der Bearbeitungszeiträume, d. h. $\delta(u) < \delta(v) < \varphi(u) < \varphi(v)$ oder $\delta(v) < \delta(u) < \varphi(v) < \varphi(u)$, nicht möglich.

5.3.6 Topologische Sortierung

Problemstellung

- ❑ Die Kanten eines azyklischen gerichteten Graphen definieren eine *partielle Ordnung* der Knoten: $u \leq v$ gdw. u ist von v aus erreichbar.
- ❑ Gesucht ist eine sequentielle Anordnung der Knoten von links nach rechts, die mit dieser partiellen Ordnung *kompatibel* ist, d. h. in der keine Kanten von links nach rechts verlaufen.

Lösung: Erweiterte Tiefensuche

- ❑ Für jeden Nachfolger v von u wird in Schritt 2.2 der Tiefensuche zusätzlich überprüft, ob er grau ist. Wenn ja, handelt es sich bei der Kante (u, v) um eine Rückwärtskante. In diesem Fall enthält der Graph einen Zyklus und kann daher nicht topologisch sortiert werden.
- ❑ Wenn jeder Knoten u nach Abschluss seiner Verarbeitung am Ende einer linearen Liste angefügt wird, enthält diese Liste nach Beendigung der Tiefensuche die Knoten in der gewünschten Reihenfolge.

Anwendungsbeispiele

- ❑ Abhängigkeiten zwischen Software-Komponenten, Begriffsdefinitionen, ...

5.4 Zusammenhangskomponenten

5.4.1 Definitionen

- ❑ Eine *Zusammenhangskomponente* eines Graphen ist eine Äquivalenzklasse der Relation „es gibt eine Verbindung von u nach v “, d. h. zwei Knoten gehören genau dann zur gleichen Zusammenhangskomponente, wenn es eine Verbindung zwischen ihnen gibt.
- ❑ Eine *starke Zusammenhangskomponente* eines gerichteten Graphen ist eine Äquivalenzklasse der Relation „ u ist von v aus erreichbar und umgekehrt“, d. h. zwei Knoten u und v gehören genau dann zur gleichen starken Zusammenhangskomponente, wenn u von v aus erreichbar ist und umgekehrt.
- ❑ Für einen gerichteten Graphen $G = (V, E)$ heißt der Graph $G^T = (V, E^T)$ mit $E^T = \{(v, u) \mid (u, v) \in E\}$ *transponierter Graph* von G .
(Die Adjazenzmatrix von G^T ist die transponierte Matrix der Adjazenzmatrix von G .)
- ❑ Offensichtlich besitzen G und G^T die gleichen starken Zusammenhangskomponenten.

5.4.2 Bestimmung von Zusammenhangskomponenten

- ❑ Die Zusammenhangskomponenten eines ungerichteten Graphen können direkt durch eine Tiefensuche bestimmt werden:
Jeder Baum des resultierenden Tiefensuchewalds entspricht direkt einer Zusammenhangskomponente.
- ❑ Die Zusammenhangskomponenten eines gerichteten Graphen $G = (V, E)$ können analog durch eine Tiefensuche im Graphen $G' = (V, E \cup E^T)$ bestimmt werden.
- ❑ Die starken Zusammenhangskomponenten eines gerichteten Graphen $G = (V, E)$ können wie folgt durch zwei aufeinanderfolgende Tiefensuchen bestimmt werden:
 - Führe eine erste Tiefensuche auf G aus, um die Abschlusszeiten $\varphi(u)$ aller Knoten $u \in V$ zu bestimmen.
 - Führe eine zweite Tiefensuche auf G^T aus, in der die Knoten u in der äußeren Schleife in absteigender Reihenfolge dieser Abschlusszeiten $\varphi(u)$ durchlaufen werden.
 - Jeder Baum des resultierenden Tiefensuchewalds der zweiten Tiefensuche entspricht einer starken Zusammenhangskomponente von G (und von G^T).
- ❑ Ein Graph ist genau dann (stark) zusammenhängend, wenn er genau eine (starke) Zusammenhangskomponente besitzt.

5.4.3 Korrektheit der Bestimmung starker Zusammenhangskomponenten

Behauptung

Jeder Baum des zweiten Tiefensuchewalds ist eine starke Zusammenhangskomponente des Graphen G .

Beweis durch vollständige Induktion

nach der Anzahl n der bis jetzt von der zweiten Tiefensuche ermittelten Bäume

Induktionsanfang $n = 0$: Hier ist nichts zu zeigen.

Induktionsschritt $n \rightarrow n + 1$:

- ❑ Nach Induktionsvoraussetzung sind die bis jetzt ermittelten n Bäume starke Zusammenhangskomponenten des Graphen.
- ❑ Zu zeigen:
Der nächste ermittelte Baum ist ebenfalls eine starke Zusammenhangskomponente.
- ❑ Sei r der Knoten, für den in der äußeren Schleife der zweiten Suche als nächstes die Operation „Durchsuchen“ ausgeführt wird und der somit die Wurzel dieses nächsten Tiefensuchebaums wird.

- Da die Knoten in dieser Schleife nach absteigenden Abschlusszeiten der ersten Suche durchlaufen werden, besitzt r von allen Knoten, die in der zweiten Suche noch nicht entdeckt wurden, die größte Abschlusszeit.
- Somit gilt für alle anderen Knoten v , die von der zweiten Suche noch nicht entdeckt wurden, $\varphi(v) < \varphi(r)$ und somit (vgl. § 5.3.5):

 - Entweder: $\delta(r) < \delta(v) < \varphi(v) < \varphi(r)$,
 d. h. v ist ein direkter oder indirekter Nachfolger von r in einem Baum der ersten Suche und somit in G von r aus erreichbar.
 - Oder: $\delta(v) < \varphi(v) < \delta(r) < \varphi(r)$,
 d. h. v liegt im ersten Tiefensuchewald links von r .
 Da es in diesem Wald keine Kanten des Graphen G von links nach rechts gibt, gibt es im transponierten Graphen G^T keine Kanten von rechts nach links.
 Daraus folgt, dass v in G^T von r aus *nicht* erreichbar ist.
- Daraus folgt, dass die zweite Suche in G^T von r aus *nur* Knoten finden wird, die auch in G von r aus erreichbar sind und somit zur starken Zusammenhangskomponente von r gehören.
 (Diese Komponente enthält ja genau diejenigen Knoten, die sowohl in G als auch in G^T von r aus erreichbar sind.)

- ❑ Außerdem werden dabei *alle* Knoten gefunden, die in G^T von r aus erreichbar sind und von der zweiten Suche nicht schon früher gefunden wurden.
- ❑ Das bedeutet umgekehrt für alle Knoten v , die dabei nicht gefunden werden:
 - Entweder ist v in G^T nicht von r aus erreichbar und gehört deshalb nicht zur starken Zusammenhangskomponente von r .
 - Oder v wurde bereits früher gefunden und gehört deshalb nach Induktionsvoraussetzung zu einer anderen starken Zusammenhangskomponente.
- ❑ Also werden *genau* diejenigen Knoten gefunden, die zur starken Zusammenhangskomponente von r gehören.

5.5 Minimale Spann­bäume und Gerüste

5.5.1 Spann­bäume und Gerüste

- ❑ Ein *Gerüst* oder *Spannwald* (spanning forest) eines ungerichteten Graphen $G = (V, E)$ ist ein Wald (V, F) mit $F \subseteq E$, der die gleichen Zusammenhangskomponenten wie G besitzt.
- ❑ Das heißt:
 - Ein Gerüst eines Graphen enthält alle Knoten des Graphen und eine möglichst kleine Teilmenge seiner Kanten.
 - Jeder Zusammenhangskomponente des Graphen entspricht ein Baum im Gerüst.
- ❑ Abkürzend wird auch die Kantenmenge F eines Gerüsts (V, F) als Gerüst bezeichnet.
- ❑ Wenn der Graph G zusammenhängend ist, ist jedes Gerüst von G ein Baum, der dann als *Spannbaum* (spanning tree) von G bezeichnet wird.

Lemma

- Ein Wald (V, F) ist ein Gerüst eines Graphen $G = (V, E)$, wenn es für jede Kante $\{u, v\} \in E$ des Graphen einen Weg im Gerüst von u nach v gibt.

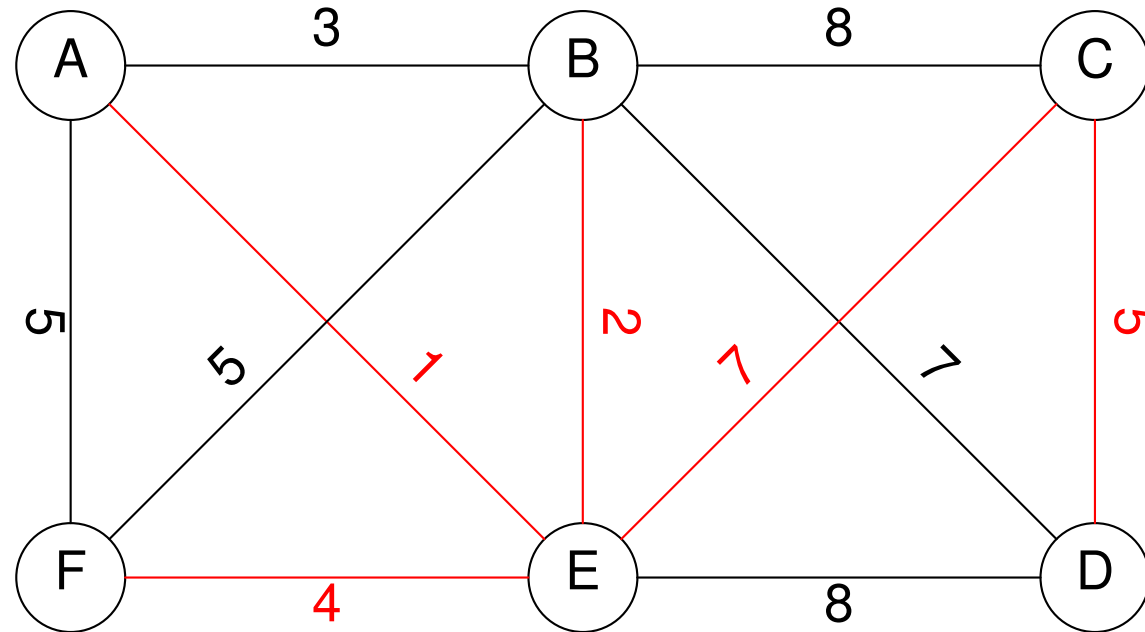
Beweis

- Damit F die gleichen Zusammenhangskomponenten wie G besitzt, muss gelten:
Wenn es in G einen Weg von einem Knoten u zu einem Knoten v gibt, dann gibt es auch in F einen Weg von u nach v , und umgekehrt.
- Wenn es in G einen Weg w_0, \dots, w_n von $u = w_0$ nach $v = w_n$ gibt, dann gibt es nach Voraussetzung für jede Kante $\{w_{i-1}, w_i\}$ dieses Wegs einen Weg in F von w_{i-1} nach w_i .
Wenn man diese Wege zu einer einzigen Knotenfolge zusammenfügt und ggf. darin enthaltene Zyklen w, \dots, w entfernt, erhält man einen Weg in F von u nach v .
- Die Umkehrung ist wegen $F \subseteq E$ trivialerweise erfüllt:
Wenn es in F einen Weg von u nach v gibt, dann gibt es denselben Weg auch in G .
- Also besitzt F tatsächlich die gleichen Zusammenhangskomponenten wie G und ist deshalb ein Gerüst von G .

5.5.2 Minimale Spannbäume und Gerüste

- Ein *gewichteter Graph* $G = (V, E, \rho)$ ist ein Graph (V, E) mit einer zugehörigen *Gewichtsfunktion* $\rho: E \rightarrow \mathbb{R}$, die jeder Kante $e \in E$ ein *Gewicht* $\rho(e) \in \mathbb{R}$ zuordnet.
- Abkürzend wird auch $\rho(u, v)$ statt $\rho((u, v))$ (Gewicht der gerichteten Kante (u, v)) bzw. $\rho(\{u, v\})$ (Gewicht der ungerichteten Kante $\{u, v\}$) geschrieben.
- Sofern nichts anderes gesagt wird, sind prinzipiell auch negative Kantengewichte zulässig.
- Das Gewicht einer Kantenmenge $F \subseteq E$ ist die Summe $\rho(F) = \sum_{e \in F} \rho(e)$ der Gewichte aller Kanten $e \in F$.
- Ein *Minimalgerüst* (oder *minimales Gerüst*, *minimaler Spannwald*; minimum spanning forest) eines ungerichteten, gewichteten Graphen $G = (V, E, \rho)$ ist ein Gerüst F des Graphen mit minimalem Gewicht, d. h. $\rho(F) \leq \rho(F')$ für jedes Gerüst F' von G .
- Wenn der Graph zusammenhängend ist, ist jedes Minimalgerüst ein Baum, der dann als *minimaler Spannbaum* (minimum spanning tree) des Graphen bezeichnet wird.

5.5.3 Beispiel



Die roten Kanten bilden einen von zwei möglichen minimalen Spannbäumen des Graphen.

5.5.4 Anwendungsbeispiele

- Verbindung elektrischer Bauteile mit möglichst wenig Draht
- Approximationsverfahren zur Lösung des Problems des Handlungsreisenden

5.5.5 Abstrakter Nächstbest-Algorithmus

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$

Algorithmus

- 1 Setze $F = \emptyset$.
- 2 Solange F kein Gerüst von G ist:
 - 1 „Zerschneide“ die Knotenmenge V in disjunkte Teilmengen $P \subseteq V$ und $Q = V \setminus P$, sodass gilt:
 - Die Menge $C = \{ \{p, q\} \in E \mid p \in P, q \in Q \}$ der Kanten, die diesen Schnitt „kreuzen“, ist nicht leer.
 - Die Mengen C und F sind disjunkt, d. h. für jede Kante $\{u, v\} \in F$ gilt entweder $u, v \in P$ oder $u, v \in Q$.
 - 2 Füge zur Menge F eine Kante $c \in C$ mit minimalem Gewicht hinzu (d. h. $\rho(c) \leq \rho(c')$ für alle $c' \in C$).

Korrektheit

Lemma 1

- a) Wenn man aus einem Baum eine Kante entfernt, zerfällt er in zwei Bäume.
- b) Wenn man zwei Bäume durch eine Kante verbindet, entsteht ein einziger Baum.
- c) Wenn man zu einem Baum eine Kante von einem Knoten u zu einem Knoten v hinzufügt und gleichzeitig eine Kante auf der Verbindung von u nach v entfernt, entsteht wieder ein Baum.

Beweis jeweils mit der Definition, dass es in einem Baum zwischen je zwei Knoten genau eine Verbindung gibt.

Lemma 2

- Aus einem Gerüst F eines Graphen G kann man weder eine Kante entfernen noch eine weitere Kante hinzufügen.

Beweis

- F besitzt als Gerüst die gleichen Zusammenhangskomponenten wie G .
- Wenn man aus F eine Kante entfernt, zerfällt gemäß Lemma 1.a einer seiner Bäume in zwei Bäume.

Deshalb besitzt die resultierende Menge \tilde{F} eine Zusammenhangskomponente mehr als G und kann deshalb kein Gerüst von G sein.

- Wenn man zu F eine weitere Kante hinzufügt:
 - Entweder verbindet diese Kante zwei Bäume zu einem einzigen Baum. Dann besitzt die resultierende Menge \tilde{F} eine Zusammenhangskomponente weniger als G und kann deshalb wiederum kein Gerüst von G sein.
 - Oder die Kante verbindet zwei Knoten u und v innerhalb desselben Baums. Dann ist die resultierende Menge \tilde{F} kein Wald – und damit auch kein Gerüst – mehr, weil es dann zwei Verbindungen von u nach v gibt (die in F bereits vorhandene Verbindung und die neue Kante von u nach v).

Satz

- Die Kantenmenge F ist zu jedem Zeitpunkt Teilmenge eines Minimalgerüsts von G , d. h. es gibt immer ein Minimalgerüst $M \subseteq E$ mit $F \subseteq M$.

Beweis durch „Schleifeninduktion“

- Vor Beginn der Schleife in Schritt 2 ist $F = \emptyset$, und somit gilt trivialerweise $F \subseteq M$ für jedes Minimalgerüst M .
- Wenn in Schritt 2.2 eine Kante $c = \{p, q\}$ mit $p \in P$ und $q \in Q$ zur Menge F hinzugefügt wird, ist die ursprüngliche Menge F' nach Induktionsvoraussetzung Teilmenge eines Minimalgerüsts M .
- Falls $c \in M$ gilt, gilt für die neue Menge F trivialerweise $F \subseteq M$.
- Andernfalls gibt es in M einen Weg von p nach q , da M ein Gerüst von G ist.
- Sei p' der letzte Knoten auf diesem Weg mit $p' \in P$ (eventuell ist $p' = p$) und q' der nächste Knoten auf diesem Weg, für den somit $q' \in Q$ gilt (eventuell ist $q' = q$), d. h. $c' = \{p', q'\} \in C$ und somit (wegen $C \cap F' = \emptyset$) $c' \notin F'$.

- Deshalb gilt für die Menge F nach Hinzufügen der Kante c : $F \subseteq \tilde{M} := M \setminus \{c'\} \cup \{c\}$, wobei \tilde{M} aufgrund von Lemma 1.c ebenfalls ein Gerüst von G ist.
- Wegen $\rho(c) \leq \rho(c')$ gilt $\rho(\tilde{M}) = \rho(M) + \rho(c) - \rho(c') \leq \rho(M)$.
- Da M bereits ein Minimalgerüst von G ist, muss somit auch \tilde{M} ein Minimalgerüst sein.
- Wegen $F \subseteq \tilde{M}$ ist F also wiederum Teilmenge eines Minimalgerüsts von G .

Korollar

- Nach Ausführung des Algorithmus ist F ein Minimalgerüst von G .

Beweis

- Die Schleife in Schritt 2 terminiert erst, wenn F ein Gerüst von G ist.
- Nach dem vorigen Satz gilt außerdem $F \subseteq M$ für ein Minimalgerüst M von G .
- Da man zu F gemäß Lemma 2 aber keine weitere Kante mehr hinzufügen kann, muss tatsächlich $F = M$ gelten.

5.5.6 Algorithmus von Kruskal

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$

Hilfsdatenstruktur

- Der Algorithmus arbeitet mit einer *Menge disjunkter Teilmengen* von V , deren Vereinigung immer gleich V ist, d. h. jeder Knoten $v \in V$ gehört zu jedem Zeitpunkt zu genau einer Teilmenge.
- Die Bedingung $\text{disjoint}(u, v)$ ist genau dann erfüllt, wenn die Knoten u und v momentan zu unterschiedlichen Teilmengen gehören.
- Wenn dies der Fall ist, vereinigt die Operation $\text{join}(u, v)$ die beiden Teilmengen, zu denen u bzw. v gehören, zu einer einzigen Teilmenge.
(Das heißt, anschließend ist $\text{disjoint}(u, v)$ sowie $\text{disjoint}(u', v')$ für alle Knoten u' und v' , die vorher zur gleichen Menge wie u bzw. v gehörten, nicht mehr erfüllt.)
- Zu Beginn des Algorithmus gehört jeder Knoten v zu einer eigenen Teilmenge, d. h. $\text{disjoint}(u, v)$ ist für alle $u \neq v$ erfüllt.

Algorithmus

- 1 Setze $F = \emptyset$.
- 2 Sortiere die Kantenmenge E nach aufsteigenden Gewichten.
- 3 Für jede Kante $\{u, v\} \in E$ in dieser sortierten Reihenfolge:
Wenn die Bedingung $\text{disjoint}(u, v)$ erfüllt ist:
 - 1 Führe $\text{join}(u, v)$ aus.
 - 2 Füge die Kante $\{u, v\}$ zur Menge F hinzu.

Ergebnis

- Nach Ausführung des Algorithmus ist die Kantenmenge F ein Minimalgerüst von G .
- Wenn G zusammenhängend ist, ist F folglich ein minimaler Spannbaum von G .
- Andernfalls kann der Algorithmus dies bei Bedarf feststellen:
Wenn am Ende $|F| < |V| - 1$ ist, ist der Graph nicht zusammenhängend.

Laufzeit

- Sortieren der Kantenmenge E (mit einem geeigneten Verfahren):
 $O(|E| \log |E|) = O(|E| \log |V|^2) = O(|E| 2 \log |V|) = O(|E| \log |V|)$
(Für jeden Graphen gilt: $|E| \leq |V|^2$.)

- Operationen auf der Hilfsdatenstruktur:

- $|E|$ -mal disjoint(u, v)
- Höchstens $|E|$ -mal join(u, v)

Bei geeigneter Implementierung ist die Laufzeit dieser Operationen jeweils $O(\log |V|)$.

Somit: $O(|E| \log |V|)$

- Gesamtlaufzeit somit: $O(|E| \log |V|)$

Beispiel

Korrektheit

Lemma 1

- Nachdem eine Kante $\{u, v\} \in E$ in Schritt 3 des Algorithmus verarbeitet wurde, ist die Bedingung $\text{disjoint}(u, v)$ dauerhaft nicht mehr erfüllt.

Beweis

- Wenn die Bedingung $\text{disjoint}(u, v)$ zum Zeitpunkt der Betrachtung der Kante noch erfüllt ist, wird in Schritt 3.1 die Operation $\text{join}(u, v)$ ausgeführt, sodass sie anschließend nicht mehr erfüllt ist.
- Sobald die Bedingung zu einem Zeitpunkt nicht mehr erfüllt ist, bleibt sie dauerhaft nicht erfüllt.

Korollar

- Die Menge F enthält zu jedem Zeitpunkt nur Kanten $\{u, v\}$, für die die Bedingung $\text{disjoint}(u, v)$ nicht erfüllt ist.

Satz 1

- Die Kantenmenge F ist zu jedem Zeitpunkt Teilmenge eines Minimalgerüsts von G .

Beweis

- Der Algorithmus von Kruskal konkretisiert den abstrakten Algorithmus aus § 5.5.5 wie folgt, wodurch die Behauptung aus dem dortigen Satz folgt:
- Jeder Schleifendurchlauf, in dem die Bedingung $\text{disjoint}(u, v)$ erfüllt ist, entspricht einem Schleifendurchlauf des abstrakten Algorithmus. (Die anderen Schleifendurchläufe sind irrelevant, weil sie die Menge F nicht verändern.)
- P entspricht dabei derjenigen Teilmenge der Hilfsdatenstruktur, die den Knoten u enthält.
- Q entspricht damit der Vereinigung aller anderen Teilmengen der Hilfsdatenstruktur.
- Damit gilt: $C = \{ \{p, q\} \in E \mid p \in P, q \in Q \} \subseteq \{ \{p, q\} \in E \mid \text{disjoint}(p, q) \}$.
- Da F gemäß Korollar nur Kanten enthält, für die die entsprechende disjoint-Bedingung nicht erfüllt ist, gilt $C \cap F = \emptyset$.
- Da die Bedingung $\text{disjoint}(u, v)$ erfüllt ist und $u \in P$ ist, ist $v \notin P$, also $v \in Q$ und somit $c = \{u, v\} \in C$ (woraus auch $C \neq \emptyset$ folgt).

- Da alle Kanten $c' = \{u', v'\}$ mit $\rho(c') < \rho(c)$ vom Algorithmus schon früher verarbeitet wurden, ist für sie die Bedingung $\text{disjoint}(u', v')$ gemäß Lemma 1 nicht erfüllt. Deshalb gilt für diese Kanten: $c' \notin C$.
- Somit ist c eine Kante aus C mit minimalem Gewicht.

Lemma 2

- Jede Teilmenge U in der Hilfsdatenstruktur ist zu jedem Zeitpunkt ein Baum.
- Genauer:
Für jede solche Teilmenge U ist (U, F_U) mit $F_U = \{\{u, v\} \in F \mid u, v \in U\}$ ein Baum.

Beweis durch „Schleifeninduktion“

- Vor Beginn der Schleife in Schritt 3 ist $F = \emptyset$, und jede Teilmenge in der Hilfsdatenstruktur enthält genau einen Knoten und ist deshalb trivialerweise ein Baum.
- Wenn in Schritt 3.1 die Teilmengen, zu denen u und v gehören – und die nach Induktionsvoraussetzung Bäume sind –, zu einer einzigen Teilmenge zusammengefasst werden, wird gleichzeitig in Schritt 3.2 die Kante $\{u, v\}$, die diese beiden Bäume zu einem einzigen Baum verbindet, zur Kantenmenge F hinzugefügt.

Satz 2

□ Nach Ausführung des Algorithmus ist die Kantenmenge F ein Gerüst des Graphen G .

Beweis

□ Gemäß Lemma in § 5.5.1 genügt es zu zeigen:

Für jede Kante $\{u, v\} \in E$ des Graphen gibt es einen Weg in F von u nach v .

□ Fall 1: $\{u, v\} \in F$

Dann gibt es trivialerweise einen Weg in F von u nach v .

□ Fall 2: $\{u, v\} \notin F$

○ Da der Algorithmus die Kante $\{u, v\}$ in Schritt 3 offenbar nicht „genommen“ hat, war die Bedingung $\text{disjoint}(u, v)$ zu diesem Zeitpunkt nicht (mehr) erfüllt.

○ Also gehörten u und v zu diesem Zeitpunkt bereits zur gleichen Teilmenge U der Hilfsdatenstruktur, die nach Lemma 2 ein Baum (U, F_U) ist.

○ Also gab es zu diesem Zeitpunkt bereits einen Weg in $F_U \subseteq F$ von u nach v .

○ Da aus der Menge F keine Kanten mehr entfernt werden, gibt es diesen Weg in F auch nach Ausführung des Algorithmus.

Aus Satz 1 und 2 folgt zusammen mit dem Korollar aus § 5.5.5 die Korrektheit des Algorithmus.

Implementierung der Hilfsdatenstruktur

- ❑ Jede Teilmenge der Hilfsdatenstruktur ist ein Baum
(der aber nichts mit den Bäumen des zu konstruierenden Gerüsts zu tun hat.)
- ❑ Für jeden Knoten $v \in V$ werden $\pi(v)$, $\tau(v)$ und $\delta(v)$ wie folgt definiert.
- ❑ Für einen Wurzelknoten v eines Baums ist $\pi(v) = \perp$,
für einen anderen Knoten ist $\pi(v)$ sein Vorgänger im Baum.
- ❑ Durch Verfolgen der Vorgängerkette findet man zu einem Knoten v den zugehörigen
Wurzelknoten $\tau(v) = \begin{cases} v, & \text{falls } \pi(v) = \perp \\ \tau(\pi(v)) & \text{sonst} \end{cases}$
- ❑ Für einen Knoten v ist $\delta(v)$ die Tiefe des Teilbaums mit Wurzel v .
- ❑ Initialisierung: Setze für jeden Knoten $v \in V$: $\pi(v) = \perp$ und $\delta(v) = 0$.
- ❑ Zwei Knoten u und v gehören genau dann zur gleichen Teilmenge, wenn sie den gleichen Wurzelknoten besitzen, d. h. die Bedingung $\text{disjoint}(u, v)$ ist genau dann erfüllt, wenn $\tau(u) \neq \tau(v)$ ist.

- Die Teilmengen, zu denen die Knoten u und v gehören, werden vereinigt, indem einer der beiden Bäume (im Zweifelsfall der mit der geringeren Tiefe) in den anderen eingehängt wird (auf diese Weise wird die Tiefe der Bäume möglichst klein gehalten):
 - Sei $u' = \pi(u)$ und $v' = \pi(v)$.
 - Wenn $\delta(u') < \delta(v')$:
 Setze $\pi(u') = v'$ (d. h. hänge den Baum mit Wurzel u' in den Baum mit Wurzel v' ein, dessen Tiefe dabei unverändert bleibt).
 - Wenn $\delta(u') > \delta(v')$:
 Setze $\pi(v') = u'$ (also gerade umgekehrt).
 - Wenn $\delta(u') = \delta(v')$:
 Setze $\pi(u') = v'$ und $\delta(v') = \delta(v') + 1$ (d. h. hänge den Baum mit Wurzel u' in den Baum mit Wurzel v' ein, dessen Tiefe dabei um eins größer wird).

Optimierungsmöglichkeit

- Jedesmal, wenn mittels $\pi(v)$ der zu einem Knoten v gehörende Wurzelknoten r bestimmt wurde, wird für alle dabei durchlaufenen Knoten u ihr Vorgänger $\pi(u)$ auf r gesetzt, d. h. diese Knoten werden direkt in den Wurzelknoten r eingehängt.
- Damit werden anschließende Berechnungen von $\pi(u)$ für diese Knoten u. U. erheblich beschleunigt.

Laufzeit der Operationen auf der Hilfsdatenstruktur

Behauptung:

- Ein Baum der Hilfsdatenstruktur mit Tiefe k enthält mindestens 2^k Knoten.

Beweis durch vollständige Induktion nach k :

- Induktionsanfang $k = 0$: Ein Baum mit Tiefe $k = 0$ enthält genau $1 = 2^0 = 2^k$ Knoten.
- Induktionsschritt $k \rightarrow k + 1$: Ein Baum mit Tiefe $k + 1$ ist aus zwei Bäumen mit Tiefe k und eventuell weiteren Bäumen mit Tiefe $\leq k$ entstanden und enthält deshalb nach Induktionsvoraussetzung mindestens $2 \cdot 2^k = 2^{k+1}$ Knoten.

Daraus folgt:

- Ein Baum mit N Knoten hat höchstens Tiefe $\log_2 N$.
- Die Länge der Vorgängerkette eines Knotens ist $O(\log |V|)$.
- Die Laufzeit der Operationen $\text{disjoint}(u, v)$ und $\text{join}(u, v)$ ist $O(\log |V|)$.

5.5.7 Algorithmus von Prim

Gegeben

- Ungerichteter, gewichteter Graph $G = (V, E, \rho)$

Algorithmus

- 1 Für jeden Knoten $v \in V$:
 - 1 Füge v mit Priorität $\delta(v) = \infty$ in eine Minimum-Vorrangwarteschlange Q ein.
 - 2 Setze $\pi(v) = \perp$.
- 2 Solange Q nicht leer ist:
 - 1 Entnimm einen Knoten u mit minimaler Priorität.
 - 2 Für jeden Nachfolger v von u :

Wenn $v \in Q$ und $\rho(u, v) < \delta(v)$:

 - 1 Erniedrige die Priorität $\delta(v)$ auf $\rho(u, v)$.
 - 2 Setze $\pi(v) = u$.

Laufzeit

□ Operationen auf der Vorrangwarteschlange:

- $|V|$ -mal Einfügen eines Knotens
- $|V|$ -mal Test, ob die Warteschlange leer ist
- $|V|$ -mal Entnehmen eines Knotens mit minimaler Priorität
- $|E|$ -mal Test, ob ein Knoten enthalten ist
(Der Rumpf der inneren Schleife wird insgesamt $|E|$ -mal ausgeführt.)
- Maximal $|E|$ -mal Erniedrigen der Priorität eines Knotens

Insgesamt: $O(|V| + |E|)$

□ Laufzeit jeder solchen Operation: $O(\log |V|)$, da die Warteschlange maximal $|V|$ Einträge enthält.

□ Gesamtlaufzeit somit: $O((|V| + |E|) \log |V|)$

□ Für zusammenhängende Graphen: $O(|E| \log |V|)$, da für sie $|E| \geq |V| - 1$ gilt.

Ergebnis

- ❑ Nach Ausführung des Algorithmus ist die Kantenmenge $F = \{ \{ \pi(v), v \} \mid \pi(v) \neq \perp \}$ ein Minimalgerüst von G .
- ❑ Wenn G zusammenhängend ist, ist F folglich ein minimaler Spannbaum von G .
- ❑ Andernfalls kann der Algorithmus dies bei Bedarf feststellen:
Wenn mehr als einmal ein Knoten u mit $\delta(u) = \infty$ entnommen wird, ist der Graph nicht zusammenhängend.

Beispiel

Korrektheit

Definitionen

$$\square P := V \setminus Q$$

$$\square \rho(u, v) := \infty, \text{ falls } \{u, v\} \notin E$$

$$\square \delta(U, v) := \min_{u \in U} \rho(u, v) \text{ für } U \subseteq V \text{ und } v \in V$$

$$\square \delta(\emptyset, v) = \min_{u \in \emptyset} \rho(u, v) := \infty$$

Lemma 1

$$\square \text{ Zu jedem Zeitpunkt gilt für jeden Knoten } v \in Q: \delta(v) = \delta(P, v)$$

Beweis durch „Schleifeninduktion“

□ Am Anfang gilt für jeden Knoten $v \in Q = V$:

○ $\delta(v) = \infty$

○ $\delta(P, v) = \delta(\emptyset, v) = \infty$, da $P = V \setminus Q = \emptyset$

Also: $\delta(v) = \delta(P, V)$

□ Nach der Entnahme eines Knotens u in Schritt 2.1 und der Verarbeitung seiner Nachfolger in Schritt 2.2 gilt für jeden Knoten $v \in Q$:

○
$$\delta(v) = \begin{cases} \rho(u, v), & \text{wenn } v \text{ ein Nachfolger von } u \text{ ist und } \rho(u, v) < \delta(v)' \text{ ist} \\ \delta(v)' & \text{sonst} \end{cases}$$

$$= \min(\delta(v)', \rho(u, v)),$$

wobei $\delta(v)'$ den Wert von $\delta(v)$ zu Beginn des Schleifendurchlaufs bezeichnet

○
$$\delta(P, v) = \min_{p \in P} \rho(p, v) = \min_{p \in P' \cup \{u\}} \rho(p, v) = \min\left(\min_{p \in P'} \rho(p, v), \rho(u, v)\right)$$

$$= \min(\delta(v)', \rho(u, v)),$$

wobei P' die Menge P zu Beginn des Schleifendurchlaufs bezeichnet

Also: $\delta(v) = \delta(P, V)$

Lemma 2

- Zu jedem Zeitpunkt gilt für jeden Knoten $v \in V$ genau eine der folgenden Aussagen:
 - a) $\pi(v) = \perp$ und $\delta(v) = \infty$
 - b) $\pi(v) \in P$ und $\delta(v) = \rho(\pi(v), v) < \infty$

Beweis durch „Schleifeninduktion“

- Am Anfang gilt für jeden Knoten $v \in V$: $\delta(v) = \infty$ und $\pi(v) = \perp$, d. h. für jeden Knoten v trifft Teil a der Behauptung zu.
- Wenn in Schritt 2.2.1 $\delta(v) = \rho(u, v) < \infty$ gesetzt wird, wird gleichzeitig in Schritt 2.2.2 $\pi(v) = u \in P$ gesetzt, d. h. dann trifft für den Knoten v Teil b der Behauptung zu. Da aus P niemals Elemente entfernt werden, bleibt die Aussage $\pi(v) \in P$ dann auch dauerhaft gültig.

Satz 1

- $F = \{ \{ \pi(v), v \} \mid \pi(v) \neq \perp, v \in P \}$ ist zu jedem Zeitpunkt Teilmenge eines Minimalgerüsts von G .

Beweis:

- Der Algorithmus von Prim konkretisiert den abstrakten Algorithmus aus § 5.5.5 wie folgt, wodurch die Behauptung aus dem dortigen Satz folgt:
- Jeder Schleifendurchlauf, in dem $\pi(u) \neq \perp$ ist, entspricht einem Schleifendurchlauf des abstrakten Algorithmus. (Die anderen Schleifendurchläufe sind irrelevant, weil sie die Menge F nicht verändern.)
- P bzw. Q entspricht dabei der Menge aller Knoten, die bereits aus der Warteschlange entnommen wurden bzw. die sich noch in der Warteschlange befinden. Der in diesem Schleifendurchlauf entnommene Knoten u wird noch zur Menge Q gezählt.
- Damit gilt für die Kante $c = \{ \pi(u), u \}$:
 - $\pi(u) \neq \perp$, und damit nach Lemma 2: $\pi(u) \in P$
 - $u \in Q$

Daraus folgt: $c \in C = \{ \{ p, q \} \in E \mid p \in P, q \in Q \}$.

□ Für jede Kante $d \in F = \{ \{ \pi(v), v \} \mid \pi(v) \neq \perp, v \in P \}$ gilt:

○ $\pi(v) \neq \perp$, und damit nach Lemma 2: $\pi(v) \in P$

○ $v \in P$

Daraus folgt: $d \notin C = \{ \{ p, q \} \mid p \in P, q \in Q \}$, und somit $F \cap C = \emptyset$.

□ Da in Schritt 2.1 immer ein Knoten u mit minimaler Priorität entnommen wird, gilt:

$$\rho(c) = \rho(\pi(u), u) \stackrel{\substack{\text{Lemma} \\ 2.b}}{=} \delta(u) = \min_{q \in Q} \delta(q) \stackrel{\substack{\text{Lemma} \\ 1}}{=} \min_{q \in Q} \delta(P, q) = \min_{q \in Q} \min_{p \in P} \rho(p, q) = \min_{c' \in C} \rho(c'),$$

d. h. c ist eine Kante aus C mit minimalem Gewicht.

Lemma 3

- Wenn für zwei verschiedene Knoten $u, v \in V$ nach Ausführung des Algorithmus $\pi(u) = \pi(v) = \perp$ gilt, dann gibt es in G keinen Weg von u nach v .

Beweis

- O. B. d. A. wurde u vor v aus Q entnommen.
- Unmittelbar vor der Entnahme von v gilt:
- $\pi(v) = \perp$, und somit nach Lemma 2: $\delta(v) = \infty$.
- Da in Schritt 2.1 immer ein Knoten mit minimaler Priorität entnommen wird, muss zu diesem Zeitpunkt gelten: $\delta(q) = \infty$ für alle Knoten $q \in Q$.
- Annahme: Es gibt einen Weg von u nach v .
- Sei p der letzte Knoten auf diesem Weg, für den $p \in P$ gilt (eventuell ist $p = u$), und q der nächste Knoten auf diesem Weg, für den dementsprechend $q \in Q$ gilt (eventuell ist $q = v$). Also ist q ein Nachfolger von p .
- Deshalb hat der Algorithmus, als er den Knoten p aus der Vorrangwarteschlange Q entnommen hat, die Priorität von q auf $\delta(q) = \rho(p, q) < \infty$ erniedrigt, sofern sie nicht schon vorher $< \infty$ war. Widerspruch!

Satz 2

- Nach Ausführung des Algorithmus ist die Kantenmenge F ein Gerüst des Graphen.

Beweis

- Gemäß Lemma in § 5.5.1 genügt es zu zeigen:
 Für jede Kante $\{u, v\} \in E$ des Graphen gibt es einen Weg in F von u nach v .
- Sei $u' = \pi(u)$ und $v' = \pi(v)$ mit $\pi(w) := \begin{cases} w, & \text{falls } \pi(w) = \perp \\ \pi(\pi(w)) & \text{sonst} \end{cases}$
 (Eventuell ist $u' = u$ und/oder $v' = v$.)
- Damit sind $u, \pi(u), \pi(\pi(u)), \dots, u'$ und $v, \pi(v), \pi(\pi(v)), \dots, v'$ bidirektionale Wege in F .
- Fall 1: $u' = v'$
 - Dann ist $u, \dots, u' = v', \dots, v$ ein Weg in F von u nach v .

□ Fall 2: $u' \neq v'$

- Wegen $\pi(u') = \pi(v') = \perp$ gibt es nach Lemma 3 keinen Weg in G von u' nach v' .
- Tatsächlich gibt es jedoch den Weg $u', \dots, u, v, \dots, v'$ von u' nach v' .
- Aufgrund dieses Widerspruchs kann Fall 2 also nie eintreten.

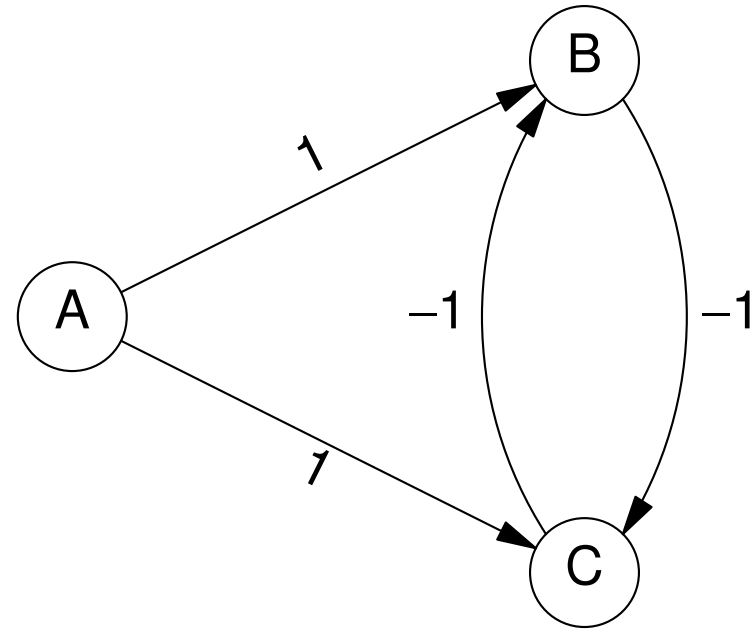
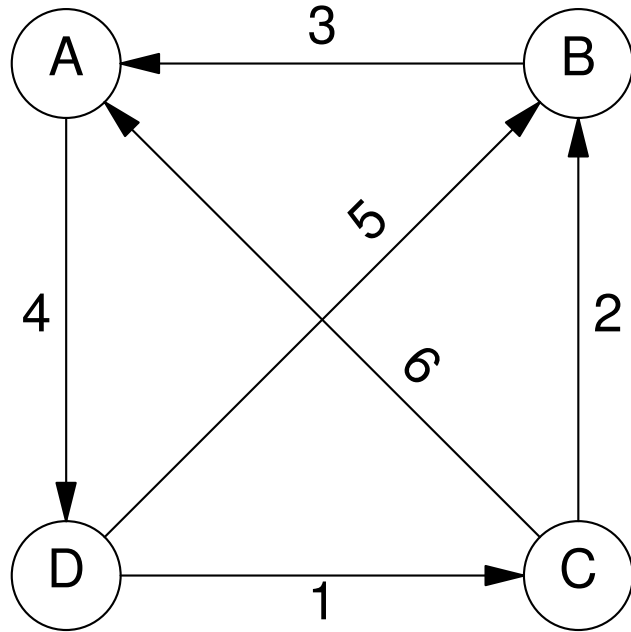
Aus Satz 1 und 2 folgt zusammen mit dem Korollar aus § 5.5.5 die Korrektheit des Algorithmus.

5.6 Kürzeste Wege (shortest paths)

5.6.1 Definitionen

- Gegeben sei ein (gerichteter oder ungerichteter) gewichteter Graph $G = (V, E, \rho)$.
- Das *Gewicht* eines Wegs $w = w_0, \dots, w_n$ ist die Summe $\rho(w) = \sum_{i=1}^n \rho(w_{i-1}, w_i)$ der Gewichte aller Kanten auf dem Weg.
- Wenn es einen Weg von einem Knoten u zu einem Knoten v gibt, ist ein *kürzester Weg* von u nach v ein Weg von u nach v mit minimalem Gewicht.
- In diesem Fall ist der *Abstand* $\delta(u, v)$ von u nach v das Gewicht eines kürzesten Wegs von u nach v . Andernfalls ist $\delta(u, v) = \infty$.
- Anmerkung: Da Kanten und Wege ein „Gewicht“ besitzen, müsste man eigentlich von „leichtesten“ statt von „kürzesten“ Wegen sprechen. Auch der Begriff des „Abstands“ passt eigentlich nicht zu dem des „Gewichts“. Trotzdem werden die Bezeichnungen üblicherweise so verwendet.

Beispiele



5.6.2 Problemstellungen

- ❑ Ein Knotenpaar:
Finde einen kürzesten Weg von einem bestimmten Startknoten s zu einem bestimmten Zielknoten t .

- ❑ Fester Startknoten:
Finde jeweils einen kürzesten Weg von einem bestimmten Startknoten s zu allen Knoten des Graphen.

- ❑ Fester Zielknoten:
Finde jeweils einen kürzesten Weg von allen Knoten des Graphen zu einem bestimmten Zielknoten t .

- ❑ Alle Knotenpaare:
Finde jeweils einen kürzesten Weg von jedem Knoten des Graphen zu jedem anderen.

Anmerkungen

- ❑ Das Problem mit festem Zielknoten lässt sich auf das Problem mit festem Startknoten zurückführen, indem man den transponierten Graphen betrachtet.
- ❑ Es sind keine Algorithmen bekannt, die das Problem für ein einzelnes Knotenpaar effizienter lösen als das Problem mit festem Startknoten.
- ❑ Das Problem für alle Knotenpaare lässt sich prinzipiell auf das Problem mit festem Startknoten zurückführen, indem man jeden Knoten des Graphen einmal als Startknoten wählt.
Hier gibt es jedoch spezielle Algorithmen, die das Problem für alle Knotenpaare effizienter lösen.
- ❑ Aufgrund dieser Beobachtungen werden im folgenden nur Algorithmen für das Problem mit festem Startknoten sowie solche für alle Knotenpaare betrachtet.

5.6.3 Hilfsmittel für das Problem mit festem Startknoten

- ❑ Für jeden Knoten $v \in V$ wird das Gewicht $\delta(v)$ des kürzesten bis jetzt gefundenen Wegs vom Startknoten s zu v sowie der Vorgänger $\pi(v)$ von v auf diesem Weg gespeichert.
- ❑ Solange noch kein Weg von s nach v gefunden wurde, ist $\delta(v) = \infty$ und $\pi(v) = \perp$.
- ❑ Initialisierung:
 - Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
 - Setze dann $\delta(s) = 0$.
- ❑ Verwerten einer Kante von u nach v :

Wenn $\delta(u) + \rho(u, v) < \delta(v)$ ist,
d. h. wenn der Weg von s nach v über u kürzer als der kürzeste bis jetzt gefundene Weg von s nach v ist:

Setze $\delta(v) = \delta(u) + \rho(u, v)$ und $\pi(v) = u$,
d. h. speichere den Weg über u als neuen kürzesten Weg nach v .

5.6.4 Algorithmus von Bellman und Ford

Gegeben

- ❑ Gewichteter Graph $G = (V, E, \rho)$, Startknoten $s \in V$
(Das heißt, der Algorithmus löst das Problem mit festem Startknoten.)
- ❑ Einschränkung:
Der Graph darf keine *negativen Zyklen* enthalten (d. h. Zyklen w_0, \dots, w_n mit $w_0 = w_n$, deren Gewicht $\sum_{i=1}^n \rho(w_{i-1}, w_i)$ negativ ist), die vom Startknoten s aus erreichbar sind.
- ❑ Die Einhaltung dieser Einschränkung wird vom Algorithmus selbst überprüft.
Wenn sie verletzt ist, bricht der Algorithmus ab.
- ❑ Abgesehen davon, sind Kanten mit negativem Gewicht erlaubt.

Algorithmus

- 1 Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
Setze dann $\delta(s) = 0$.
- 2 Wiederhole $(|V| - 1)$ -mal:
Für jede Kante $(u, v) \in E$:
Verwerte die Kante (vgl. § 5.6.3).
- 3 Für jede Kante $(u, v) \in E$:
Wenn $\delta(u) + \rho(u, v) < \delta(v)$:
Abbruch, weil der Graph einen von s aus erreichbaren negativen Zyklus enthält.

Ergebnis

- Wenn der Graph einen von s aus erreichbaren negativen Zyklus enthält, bricht der Algorithmus in Schritt 3 ab.
- Andernfalls gilt nach Ausführung des Algorithmus für jeden Knoten $v \in V$:
 - $\delta(v) = \delta(s, v)$
 - Wenn dieser Wert kleiner als ∞ ist, ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v .

Laufzeit

- ❑ Initialisierung (Schritt 1): $O(|V|)$
- ❑ Eigentlicher Algorithmus (Schritt 2): $O(|V| \cdot |E|)$
- ❑ Überprüfung auf negative Zyklen (Schritt 3): $O(|E|)$
- ❑ Gesamtlaufzeit also: $O(|V| \cdot |E|)$

Beispiel

- ❑ Siehe § 5.6.1.

Korrektheit

Definition

- ❑ Eine *Route* (oder ein *Multiweg*) der Länge n von einem Knoten u zu einem Knoten v ist eine Folge von Knoten w_0, \dots, w_n mit $u = w_0$, Kanten von w_{i-1} nach w_i für $i = 1, \dots, n$ und $w_n = v$.
(Im Gegensatz zu einem Weg, darf ein Knoten in einer Route auch mehrmals vorkommen, d. h. eine Route darf auch Zyklen enthalten.)

Lemma 1

- Für jeden Knoten $v \in V$ gilt zu jedem Zeitpunkt entweder $\delta(v) = \infty$ oder

$\delta(v) = \rho(w) = \sum_{i=1}^n \rho(w_{i-1}, w_i)$ für eine Route $w = w_0, \dots, w_n$ vom Startknoten s zum Knoten v .

Beweis durch vollständige Induktion nach der Anzahl der bis jetzt verwerteten Kanten

- Am Anfang gilt für $v = s$: $\delta(v) = 0 = \rho(w)$ für die leere Route w von s nach v , und $\delta(v) = \infty$ für alle anderen Knoten $v \neq s$.
- Nach der Verwertung einer Kante (u, v) in Schritt 2 ist $\delta(v)$ entweder unverändert – dann folgt die Behauptung für diesen Knoten v direkt aus der Induktionsvoraussetzung –, oder es gilt: $\delta(v) = \delta(u) + \rho(u, v) = \rho(w) + \rho(u, v) = \rho(w')$ für eine Route $w = s, \dots, u$ – die es nach Induktionsvoraussetzung gibt – und die Route $w' = s, \dots, u, v$.
Für alle anderen Knoten v bleibt $\delta(v)$ unverändert.

Lemma 2

- Nach dem n -ten Durchlauf der Schleife in Schritt 2 gilt für jede Route $w = s, \dots, v$ der Länge n vom Startknoten s zu irgendeinem Knoten v : $\delta(v) \leq \rho(w)$.

Beweis durch vollständige Induktion nach n

- Induktionsanfang $n = 0$

- Die einzige Route w der Länge 0 vom Startknoten s zu irgendeinem Knoten v ist die leere Route von s nach s , für die gilt: $\rho(w) = 0$.
- Also gilt für $v = s$: $\delta(v) = 0 \leq \rho(w)$.

- Induktionsschritt $n \rightarrow n + 1$

- Sei $w = s, \dots, u, v$ eine Route der Länge $n + 1$ und $w' = s, \dots, u$.
- Nach dem n -ten Durchlauf der Schleife gilt nach Induktionsvoraussetzung:
 $\delta(u) \leq \rho(w')$.
- Im $(n + 1)$ -ten Durchlauf der Schleife wird u. a. die Kante (u, v) verwertet, woraus
 $\delta(v) \leq \delta(u) + \rho(u, v) \leq \rho(w') + \rho(u, v) = \rho(w)$ folgt.
- Durch die Verwertung weiterer Kanten im selben oder späteren Durchläufen wird der Wert von $\delta(v)$ eventuell noch kleiner, aber niemals größer.

Lemma 3

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, gilt für jeden Knoten $v \in V$ zu jedem Zeitpunkt: $\delta(v) \geq \delta(s, v)$.

Beweis

- Für $\delta(v) = \infty$ gilt trivialerweise $\delta(v) \geq \delta(s, v)$, unabhängig vom Wert von $\delta(s, v)$.
- Für $\delta(v) < \infty$ gilt nach Lemma 1: $\delta(v) = \rho(w)$ für eine Route w von s nach v .
- Da es keinen von s aus erreichbaren negativen Zyklus gibt, kann diese Route keinen negativen Zyklus enthalten.
- Für den Weg w' von s nach v , der entsteht, wenn man aus der Route w eventuell vorhandene Zyklen entfernt, gilt nach Definition von $\delta(s, v)$: $\rho(w') \geq \delta(s, v)$.
- Da das Gewicht der eventuell entfernten Zyklen nicht negativ ist, gilt:
$$\delta(v) = \rho(w) \geq \rho(w') \geq \delta(s, v).$$

Satz 1

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, gilt nach Ausführung von Schritt 2 für jeden Knoten $v \in V$: $\delta(v) = \delta(s, v)$.

Beweis:

- Nach Lemma 3 gilt für jeden Knoten $v \in V$: $\delta(v) \geq \delta(s, v)$.
Deshalb genügt es zu zeigen, dass für jeden Knoten $v \in V$ gilt: $\delta(v) \leq \delta(s, v)$.
- Wenn v von s aus nicht erreichbar ist, ist $\delta(s, v) = \infty$ und somit trivialerweise $\delta(v) \leq \delta(s, v)$.
- Wenn v von s aus erreichbar ist, gibt es einen Weg und damit auch einen kürzesten Weg $w = s, \dots, v$ von s nach v , d. h. $\delta(s, v) = \rho(w)$.
- Da ein Weg keinen Knoten mehrmals enthalten kann, ist die Länge dieses Wegs höchstens $|V| - 1$.
- Deshalb gilt nach dem $(|V| - 1)$ -ten Durchlauf der Schleife gemäß Lemma 2: $\delta(v) \leq \rho(w) = \delta(s, v)$.

Korollar

- Wenn G keinen vom Startknoten s aus erreichbaren negativen Zyklus enthält, bricht der Algorithmus in Schritt 3 nicht ab.

Beweis

- Annahme: Es gibt eine Kante $(u, v) \in E$, für die in Schritt 3 die Abbruchbedingung $\delta(u) + \rho(u, v) < \delta(v)$ erfüllt ist.
- Nach Satz 1 gilt zu diesem Zeitpunkt: $\delta(u) = \delta(s, u)$ und $\delta(v) = \delta(s, v)$.
Also muss für die Kante (u, v) die Bedingung $\delta(s, u) + \rho(u, v) < \delta(s, v)$ erfüllt sein, woraus insbesondere $\delta(s, u) < \infty$ folgt.
- Deshalb gibt es einen kürzesten Weg $w = s, \dots, u$ von s nach u mit $\rho(w) = \delta(s, u)$.
- Dann ist $w' = s, \dots, u, v$ ein Weg von s nach v mit Gewicht $\delta(s, u) + \rho(u, v) < \delta(s, v)$.
- Widerspruch zur Definition von $\delta(s, v)$!
- Also gibt es keine Kante, für die in Schritt 3 die Abbruchbedingung erfüllt ist.

Satz 2

- Wenn G einen von s aus erreichbaren negativen Zyklus $w = w_0, \dots, w_n$ enthält, bricht der Algorithmus in Schritt 3 ab.

Beweis

- Da w_0, \dots, w_n ein Zyklus ist, gilt $w_0 = w_n$ und deshalb $S = \sum_{i=1}^n \delta(w_i) = \sum_{i=1}^n \delta(w_{i-1})$.
- Da der Zyklus von s aus erreichbar ist, gibt es für jeden seiner Knoten w_i einen Weg w von s nach w_i mit Länge $\leq |V| - 1$.
 Deshalb gilt nach Lemma 2: $\delta(w_i) \leq \rho(w) < \infty$ für jeden Knoten w_i des Zyklus und somit auch $S = \sum_{i=1}^n \delta(w_i) < \infty$.
- Annahme: In Schritt 3 gilt für keine Kante $(u, v) \in E$ die Abbruchbedingung $\delta(u) + \rho(u, v) < \delta(v)$, d. h. für jede Kante $(u, v) \in E$ gilt: $\delta(v) \leq \delta(u) + \rho(u, v)$.
- Insbesondere gilt $\delta(w_i) \leq \delta(w_{i-1}) + \rho(w_{i-1}, w_i)$ für jede Kante (w_{i-1}, w_i) des Zyklus und somit: $S = \sum_{i=1}^n \delta(w_i) \leq \sum_{i=1}^n (\delta(w_{i-1}) + \rho(w_{i-1}, w_i)) = \sum_{i=1}^n \delta(w_{i-1}) + \sum_{i=1}^n \rho(w_{i-1}, w_i) = S + \rho(w)$.
- Daraus folgt wegen $S < \infty$: $0 \leq \rho(w)$. Widerspruch, da w ein negativer Zyklus ist.

5.6.5 Algorithmus von Dijkstra

Gegeben

- ❑ Gewichteter Graph $G = (V, E, \rho)$, Startknoten $s \in V$
(Das heißt, der Algorithmus löst das Problem mit festem Startknoten.)
- ❑ Einschränkung:
Der Graph darf keine Kanten mit negativem Gewicht enthalten.
- ❑ Die Einhaltung dieser Einschränkung wird vom Algorithmus *nicht* überprüft.
Wenn sie verletzt ist, ist das Ergebnis des Algorithmus undefiniert.

Algorithmus

- 1 Für alle Knoten $v \in V$: Setze $\delta(v) = \infty$ und $\pi(v) = \perp$.
Setze dann $\delta(s) = 0$.
- 2 Für alle Knoten $v \in V$:
Füge v mit Priorität $\delta(v)$ in eine Minimum-Vorrangwarteschlange ein.
- 3 Solange die Warteschlange nicht leer ist:
 - 1 Entnimm einen Knoten u mit minimaler Priorität.
 - 2 Für jeden Nachfolger v von u , der sich noch in der Warteschlange befindet:
 - 1 Verwerte die Kante (u, v) (vgl. § 5.6.3).
 - 2 Wenn $\delta(v)$ dadurch erniedrigt wurde:
Erniedrige die Priorität von v in der Warteschlange entsprechend.

Ergebnis

- Wenn der Graph keine Kanten mit negativem Gewicht enthält, gilt nach Ausführung des Algorithmus für alle Knoten $v \in V$:
 - $\delta(v) = \delta(s, v)$
 - Wenn dieser Wert kleiner als ∞ ist, ist $\pi(v)$ der Vorgänger von v auf einem kürzesten Weg von s nach v .

- Andernfalls ist das Ergebnis des Algorithmus undefiniert.

Beispiel

- Siehe § 5.6.1.

Laufzeit

- ❑ Initialisierung (Schritt 1): $O(|V|)$
- ❑ Operationen auf der Vorrangwarteschlange:
 - $|V|$ -mal Einfügen eines Knotens
 - $|V|$ -mal Test, ob die Warteschlange leer ist
 - $|V|$ -mal Entnehmen eines Knotens mit minimaler Priorität
 - $|E|$ -mal Test, ob ein Knoten enthalten ist
 - Maximal $|E|$ -mal Erniedrigen der Priorität eines Knotens
(Der Rumpf der inneren Schleife wird höchstens $|E|$ -mal ausgeführt.)

Insgesamt $O(|V| + |E|)$

- ❑ Laufzeit jeder solchen Operation: $O(\log |V|)$,
da die Warteschlange maximal $|V|$ Einträge enthält.
- ❑ Gesamtlaufzeit somit: $O((|V| + |E|) \log |V|)$

Korrektheit

1. Sei Q die Menge der Knoten, die sich zu einem bestimmten Zeitpunkt noch in der Warteschlange befinden, und $P = V \setminus Q$.
2. Am Ende jedes Durchlaufs durch die Schleife in Schritt 3 gilt: $\delta(v) \leq \delta(u) + \rho(u, v)$ für jeden Nachfolger v von u , der sich noch in der Warteschlange befindet.
(Dies folgt unmittelbar aus der Definition der Operation „Verwerten“.)
3. Zum Zeitpunkt der Entnahme eines Knotens u aus der Warteschlange gilt:
 $\delta(u) = \delta(s, u)$.
4. Da $\delta(u)$ für Knoten $u \in P$ nicht mehr verändert wird und $\delta(v)$ für Knoten $v \in Q$ später höchstens noch verkleinert wird, gelten diese Aussagen auch zu jedem späteren Zeitpunkt.
5. Damit gilt insbesondere nach Ausführung des Algorithmus: $\delta(u) = \delta(s, u)$ für jeden Knoten $u \in V$.

Beweis von Aussage 3 durch „Schleifeninduktion“:

- Zum Zeitpunkt der Entnahme von $u = s$ gilt: $\delta(s) = 0 = \delta(s, s)$
- Zum Zeitpunkt der Entnahme eines anderen Knotens u gilt:

- Da es keine negativen Kantengewichte und somit auch keine negativen Zyklen gibt, gilt nach Lemma 3 in § 5.6.4: $\delta(u) \geq \delta(s, u)$.
- Wenn u von s aus nicht erreichbar ist, gilt somit: $\delta(u) \geq \delta(s, u) = \infty \Rightarrow \delta(u) = \delta(s, u)$.
- Wenn u von s aus erreichbar ist, gibt es einen kürzesten Weg $w = s, \dots, u$ von s nach u mit Gewicht $\rho(w) = \delta(s, u)$.
- Sei p der letzte Knoten auf diesem Weg, für den $p \in P$ gilt (eventuell ist $p = s$), und q der nächste Knoten auf diesem Weg (eventuell ist $q = u$), d. h. q ist ein Nachfolger von p , und es gilt $q \in Q$.
- Somit gilt: $\delta(s, u) = \rho(w) = \rho(s, \dots, p, q, \dots, u) = \rho(s, \dots, p) + \rho(p, q) + \rho(q, \dots, u) \stackrel{(a)}{\geq}$

$$\rho(s, \dots, p) + \rho(p, q) \stackrel{(b)}{\geq} \delta(s, p) + \rho(p, q) \stackrel{(c)}{=} \delta(p) + \rho(p, q) \stackrel{(d)}{\geq} \delta(q) \stackrel{(e)}{\geq} \delta(u),$$
 denn:
 - a) $\rho(q, \dots, u) \geq 0$, da es keine negativen Kantengewichte gibt.
 - b) $\rho(s, \dots, p) \geq \delta(s, p)$ für jeden Weg von s nach p
 - c) Induktionsvoraussetzung
 - d) Aussage 2
 - e) u ist der Knoten mit minimaler Priorität, d. h. $\delta(u) \leq \delta(q)$ für alle $q \in Q$.
- Damit gilt einerseits $\delta(u) \geq \delta(s, u)$ und andererseits $\delta(u) \leq \delta(s, u)$ und somit $\delta(u) = \delta(s, u)$.

5.6.6 Algorithmus von Floyd und Warshall

Gegeben

- ❑ Gewichteter Graph $G = (V, E, \rho)$
(Das heißt, der Algorithmus löst das Problem für alle Knotenpaare.)
- ❑ Einschränkung: Der Graph darf keine negativen Zyklen enthalten.
- ❑ Die Einhaltung dieser Einschränkung wird vom Algorithmus *nicht* überprüft.
Wenn sie verletzt ist, ist das Ergebnis des Algorithmus undefiniert.
- ❑ Abgesehen davon, sind Kanten mit negativem Gewicht erlaubt.

Definitionen

- ❑ Zur Vereinfachung der Notation sei die Knotenmenge des Graphen $V = \{1, \dots, m\}$.
- ❑ Für $k = 0, \dots, m$ sei ein Weg von u nach v *via* k ein Weg w_0, \dots, w_n mit $w_0 = u$, $w_n = v$ und $w_1, \dots, w_{n-1} \leq k$, d. h. alle *Zwischenknoten* (sofern es welche gibt) gehören zur Menge $\{1, \dots, k\}$ (die für $k = 0$ leer ist).

- Für $k = 0, \dots, m$ und $u, v \in V$ sei
 - $\Delta_k(u, v)$ entweder das Gewicht eines kürzesten Wegs von u nach v via k oder ∞ (falls es keinen solchen Weg gibt)
 - $\Pi_k(u, v)$ entweder der Vorgänger von v auf diesem Weg oder \perp .

Rekursionsgleichungen

- Für $k = 0$ gilt:

$\Delta_0(u, v) =$	$\Pi_0(u, v) =$	wenn
0	\perp	$u = v$
$\rho(u, v)$	u	$u \neq v$ und $(u, v) \in E$ bzw. $\{u, v\} \in E$
∞	\perp	sonst

- Für $k = 1, \dots, m$ gilt:

$\Delta_k(u, v) =$	$\Pi_k(u, v) =$	wenn
$\Delta_{k-1}(u, v)$	$\Pi_{k-1}(u, v)$	$\Delta_{k-1}(u, v) \leq \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$
$\Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$	$\Pi_{k-1}(k, v)$	$\Delta_{k-1}(u, v) > \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$

Begründung

Mit den Bezeichnungen $d = \Delta_k(u, v)$, $d_1 = \Delta_{k-1}(u, v)$ und $d_2 = \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)$ gelten folgende Aussagen:

1. $d = d_1$ oder $d = d_2$

Beweis:

- Wenn es keinen Weg von u nach v via k gibt, ist $d = \infty$.
In diesem Fall gibt es auch keinen Weg von u nach v via $k - 1$, d. h. es ist auch $d_1 = \infty$ und somit $d = d_1$.
- Wenn es einen kürzesten Weg von u nach v via k (mit Gewicht d) gibt und dieser den Knoten k *nicht* enthält, stimmt er mit einem kürzesten Weg von u nach v via $k - 1$ (mit Gewicht d_1) überein, d. h. es gilt wiederum $d = d_1$.
- Andernfalls besteht dieser Weg (mit Gewicht d) aus einem (eventuell leeren) kürzesten Teilweg von u nach k via $k - 1$ (mit Gewicht $\Delta_{k-1}(u, k)$) und einem (eventuell leeren) kürzesten Teilweg von k nach v via $k - 1$ (mit Gewicht $\Delta_{k-1}(k, v)$), d. h. es gilt $d = d_2$.
(Beachte: Jeder Teilweg eines kürzesten Wegs ist ebenfalls ein kürzester Weg, sofern es keine negativen Zyklen gibt.)

2. $d = \min(d_1, d_2)$

Anmerkung:

- Diese Aussage folgt *nicht* unmittelbar aus der vorigen Aussage 1!
- Wenn ein Graph einen negativen Zyklus enthält, kann $d = d_1$ gelten, obwohl $d_1 > d_2$ ist.
- Deshalb verwendet der folgende Beweis an einer entscheidenden Stelle die Voraussetzung, dass der Graph *keinen* negativen Zyklus enthält.

Beweis:

- Wenn $d_1 = d_2$ ist (insbesondere auch, wenn $d_1 = d_2 = \infty$ ist), folgt die Behauptung unmittelbar aus der vorigen Aussage 1.
- Wenn $d_1 < d_2$ ist (woraus insbesondere $d_1 < \infty$ folgt), dann gibt es einen kürzesten Weg von u nach v via $k - 1$ mit Gewicht d_1 , der auch ein Weg von u nach v via k ist.

Daraus folgt $d \leq d_1$, und zusammen mit $d_1 < d_2$ und Aussage 1:
 $d = d_1 = \min(d_1, d_2)$.

- Wenn $d_2 < d_1$ ist (woraus insbesondere $d_2 < \infty$ folgt), dann gibt es einen kürzesten Weg von u nach k via $k - 1$ mit Gewicht $\Delta_{k-1}(u, k)$ sowie einen kürzesten Weg von k nach v via $k - 1$ mit Gewicht $\Delta_{k-1}(k, v)$.
 - Wenn man diese Wege zusammensetzt, entsteht eine *Route* von u nach v via k mit Gewicht d_2 .
 - Annahme: Diese Route u, \dots, k, \dots, v ist kein Weg, d. h. sie enthält (mindestens) einen Zyklus w, \dots, w .
 - Da die Teilrouten u, \dots, k und k, \dots, v Wege sind, d. h. keinen Knoten mehrfach enthalten, muss der Zyklus den Knoten k enthalten, d. h. die Route muss $u, \dots, w, \dots, k, \dots, w, \dots, v$ lauten.
 - Wenn man den Zyklus (bzw. die Zyklen) entfernt, entsteht somit ein Weg von u nach v via $k - 1$ mit Gewicht $\leq d_2$, da das Gewicht der Zyklen nicht negativ ist.
 - Zusammen mit $d_2 < d_1$ ist dies ein Widerspruch dazu, dass d_1 das Gewicht eines kürzesten Wegs von u nach v via $k - 1$ ist.
 - Also muss die o. g. Route immer ein *Weg* von u nach v via k mit Gewicht d_2 sein.
 - Daraus folgt $d \leq d_2$, und zusammen mit $d_2 < d_1$ und Aussage 1:
 $d = d_2 = \min(d_1, d_2)$.

Praktische Berechnung

□ Zur Berechnung des Werts $\Delta_k(u, v)$ in Zeile u und Spalte v der Matrix Δ_k werden neben dem korrespondierenden Wert $\Delta_{k-1}(u, v)$ der „vorigen“ Matrix Δ_{k-1} noch die Werte $\Delta_{k-1}(u, k)$ und $\Delta_{k-1}(k, v)$ benötigt, die sich in Spalte bzw. Zeile k dieser Matrix befinden.

□ Diese Werte in Spalte und Zeile k bleiben beim Übergang von Δ_{k-1} zu Δ_k unverändert, denn es gilt:

○ Für $v = k$ (d. h. Spalte k):

$$\begin{aligned} \Delta_k(u, v) &= \min(\Delta_{k-1}(u, v), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(u, k), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, k)) = \\ &= \min(\Delta_{k-1}(u, k), \Delta_{k-1}(u, k) + 0) = \Delta_{k-1}(u, k) = \Delta_{k-1}(u, v) \end{aligned}$$

○ Für $u = k$ (d. h. Zeile k):

$$\begin{aligned} \Delta_k(u, v) &= \min(\Delta_{k-1}(u, v), \Delta_{k-1}(u, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(k, v), \Delta_{k-1}(k, k) + \Delta_{k-1}(k, v)) = \\ &= \min(\Delta_{k-1}(k, v), 0 + \Delta_{k-1}(k, v)) = \Delta_{k-1}(k, v) = \Delta_{k-1}(u, v) \end{aligned}$$

(Beachte: $\Delta_{k-1}(k, k) = 0$, weil ein kürzester Weg von k nach k immer Gewicht 0 besitzt.)

□ Somit können die Werte der Matrix Δ_{k-1} (und analog Π_{k-1}) jeweils gefahrlos durch die korrespondierenden Werte der Matrix Δ_k (und analog Π_k) überschrieben werden.

Algorithmus

- 1 Für $u = 1, \dots, m$:
 - 1 Für $v = 1, \dots, m$: Setze $\Delta(u, v) = \infty$ und $\Pi(u, v) = \perp$.
 - 2 Für jeden Nachfolger v von u : Setze $\Delta(u, v) = \rho(u, v)$ und $\Pi(u, v) = u$.
 - 3 Setze $\Delta(u, u) = 0$ und $\Pi(u, u) = \perp$.
- 2 Für $k = 1, \dots, m$:
Für $u = 1, \dots, m$ und $v = 1, \dots, m$:
Wenn $\Delta(u, v) > \Delta(u, k) + \Delta(k, v)$:
Setze $\Delta(u, v) = \Delta(u, k) + \Delta(k, v)$ und $\Pi(u, v) = \Pi(k, v)$.

Ergebnis

- Nach Ausführung des Algorithmus gilt für alle Knoten $u, v \in V$:
 - $\Delta(u, v) = \Delta_m(u, v) =$ Gewicht eines kürzesten Wegs von u nach v via $m = \delta(u, v) =$ Gewicht eines beliebigen kürzesten Wegs von u nach v , falls ein solcher Weg existiert, andernfalls ∞
 - $\Pi(u, v) = \Pi_m(u, v) =$ Vorgänger von v auf diesem Weg bzw. \perp

Laufzeit

□ Offensichtlich $O(m^3) = O(|V|^3)$

Beispiel

□ Siehe § 5.6.1.

5.6.7 Laufzeitvergleich

Fester Startknoten

- Das Problem mit festem Startknoten kann mit folgenden Algorithmen gelöst werden, sofern die jeweilige Voraussetzung erfüllt ist:
 - Dijkstra (D), wenn es keine negativen Kantengewichte gibt
 - Bellman-Ford (BF),
wenn es keinen vom Startknoten aus erreichbaren negativen Zyklus gibt
 - Floyd-Warshall (FW), wenn es überhaupt keinen negativen Zyklus gibt

- Die nachfolgende Tabelle zeigt die jeweiligen Laufzeiten im Vergleich

<i>Algorithmus</i>	<i>Laufzeit</i>		
	allgemein	wenn $ E \approx V $	wenn $ E \approx V ^2$
D	$O((V + E) \log V)$	$O(V \log V)$	$O(V ^2 \log V)$
BF	$O(V \cdot E)$	$O(V ^2)$	$O(V ^3)$
FW	$O(V ^3)$	$O(V ^3)$	$O(V ^3)$

Alle Knotenpaare

- Das Problem für alle Knotenpaare kann mit folgenden Algorithmen gelöst werden, sofern die jeweilige Voraussetzung erfüllt ist:
 - $|V|$ -mal Dijkstra (D), wenn es keine negativen Kantengewichte gibt
 - $|V|$ -mal Bellman-Ford (BF), wenn es keinen negativen Zyklus gibt
 - 1-mal Floyd-Warshall (FW), wenn es keinen negativen Zyklus gibt

- Die nachfolgende Tabelle zeigt die jeweiligen Gesamtlaufzeiten im Vergleich

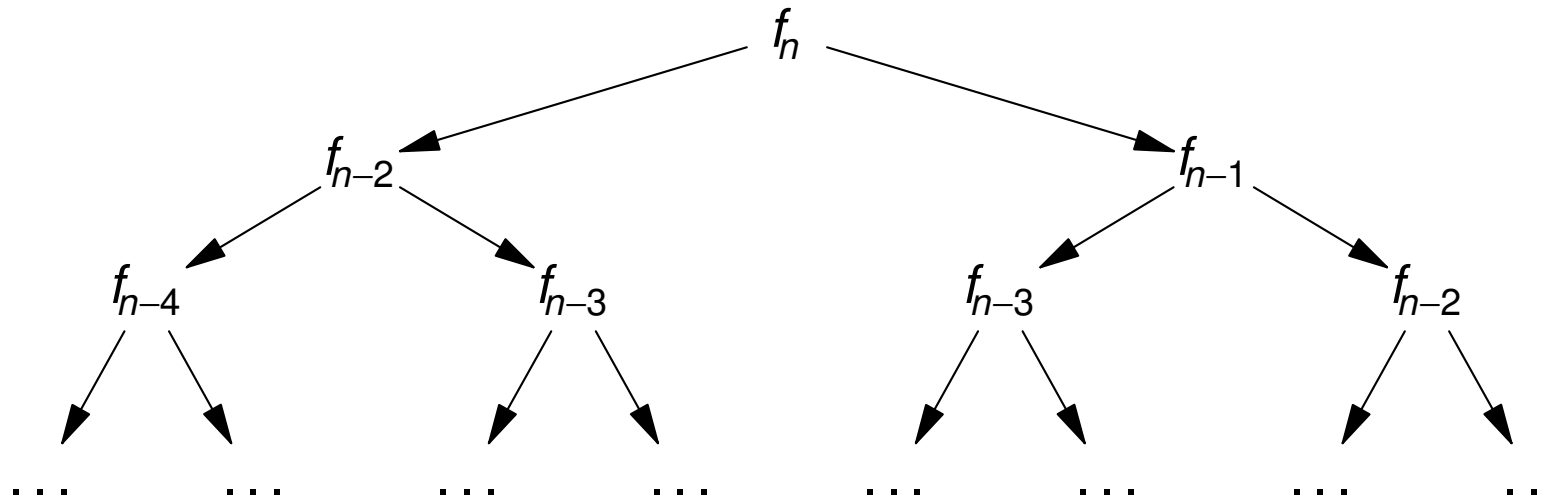
<i>Algorithmus</i>	<i>Laufzeit</i>		
	allgemein	wenn $ E \approx V $	wenn $ E \approx V ^2$
D	$O(V (V + E) \log V)$	$O(V ^2 \log V)$	$O(V ^3 \log V)$
BF	$O(V ^2 \cdot E)$	$O(V ^3)$	$O(V ^4)$
FW	$O(V ^3)$	$O(V ^3)$	$O(V ^3)$

6 Tabellengestützte Programmierung (dynamic programming)

6.1 Einführende Beispiele

6.1.1 Fibonacci-Zahlen

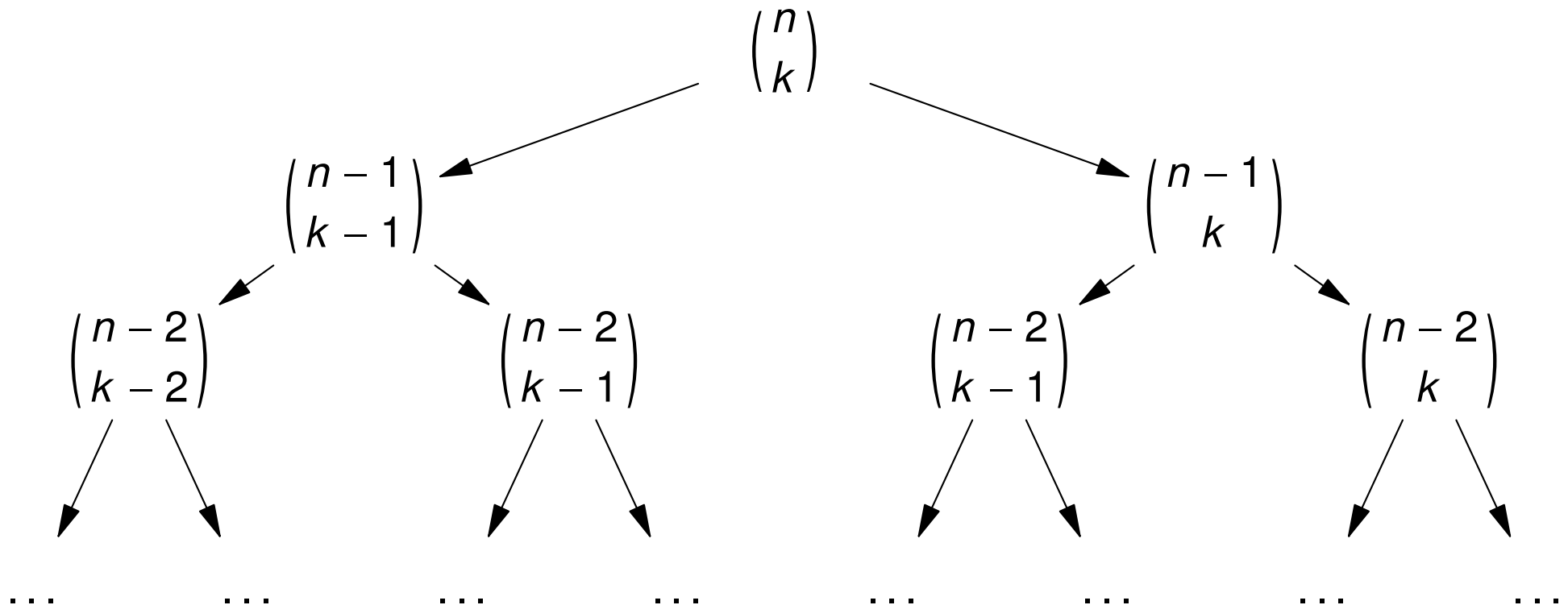
□ Rekursive Definition für $n \in \mathbb{N}_0$: $f_n = \begin{cases} n & \text{für } n \leq 1 \\ f_{n-2} + f_{n-1} & \text{für } n \geq 2 \end{cases}$



6.1.2 Binomialkoeffizienten

□ Rekursive Berechnung für $n \in \mathbb{N}_0$ und $k = 0, \dots, n$ (vgl. Pascalsches Dreieck):

$$\binom{n}{k} = \begin{cases} 1 & \text{für } k = 0 \text{ oder } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$



6.1.3 Allgemeines Prinzip

Problem

- Bei der rekursiven Berechnung werden Teilergebnisse immer wieder – mit sehr hohem Aufwand – neu berechnet.

Lösungsidee

- Einmal berechnete Teilergebnisse werden in einer Tabelle (beispielsweise in einer Streuwerttabelle) gespeichert.
- Vor der Berechnung eines Teilergebnisses wird anhand der Tabelle überprüft, ob es bereits vorhanden ist.

Effekt

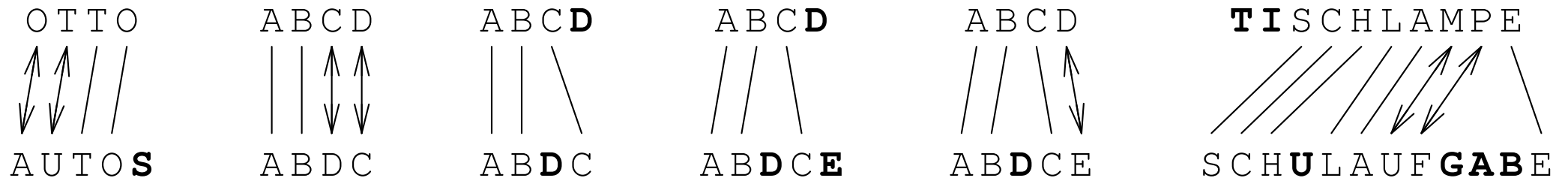
- Berechnungen, die ohne diese Optimierung extrem hohe (oft exponentielle) Laufzeit besitzen, können wesentlich effizienter (normalerweise mit polynomieller Laufzeit) durchgeführt werden.

6.2 Editierdistanz

Definition

- Die *Editierdistanz* (oder Levenshtein-Distanz) $D(s, t)$ zweier Zeichenketten s und t ist die minimale Anzahl elementarer *Editieroperationen*, mit denen s in t überführt werden kann.
- Elementare Editieroperationen sind:
 - Entferne das Zeichen ... an Position ...
 - Füge nach Position ... das Zeichen ... ein
 - Ersetze das Zeichen ... an Position ... durch das Zeichen ...

Beispiele



Beobachtungen

- ❑ Die Editierdistanz ist offenbar symmetrisch, d. h. es gilt $D(s, t) = D(t, s)$ für alle Zeichenketten s und t .

- ❑ Folgende Operationen sind offensichtlich unnötig:
 - Löschen eines zuvor eingefügten Zeichens
→ Zeichen gar nicht einfügen

 - Löschen eines zuvor ersetzten Zeichens
→ Gleich das ursprüngliche Zeichen löschen

 - Ersetzen eines zuvor eingefügten Zeichens
→ Gleich das endgültige Zeichen einfügen

 - Mehrmaliges Ersetzen eines Zeichens
→ Zeichen einmalig durch das endgültige Zeichen ersetzen

- ❑ Ohne offensichtlich unnötige Operationen gibt es für jedes Zeichen aus s genau drei Möglichkeiten:
 - Es bleibt unverändert.
 - Es wird durch ein anderes Zeichen ersetzt, das anschließend unverändert bleibt.
 - Es wird entfernt.

- ❑ Umgekehrt gibt es für jedes Zeichen aus t genau drei Möglichkeiten:
 - Es wurde unverändert aus s übernommen.
 - Es ist durch Ersetzung eines Zeichens aus s entstanden.
 - Es wurde neu eingefügt.

Satz 1

- Für eine beliebige Zeichenkette s und die leere Zeichenkette ε gilt:
 $D(s, \varepsilon) = D(\varepsilon, s) = |s| = \text{Länge von } s.$

Beweis:

- Um s in die leere Zeichenkette zu überführen, sind offenbar genau $|s|$ Löschoptionen nötig.
- Um die leere Zeichenkette in s zu überführen, sind offenbar genau $|s|$ Einfügeoptionen nötig.

Satz 2

- Für nichtleere Zeichenketten s und t gilt: $D(s, t) = \min \left\{ \begin{array}{l} D(s', t) + 1 \\ D(s, t') + 1 \\ D(s', t') + \delta \end{array} \right\}.$

- Dabei bezeichne s' bzw. t' die Zeichenkette s bzw. t ohne ihr letztes Zeichen.
- δ ist 0, wenn die letzten Zeichen von s und t gleich sind, sonst 1.

Beweis:

□ Zu zeigen:

Die minimale Anzahl elementarer Editieroperationen, um s in t zu überführen, ist gleich dem durch das Minimum definierten Wert M .

□ Das heißt:

- a) Es gibt eine Folge von M Editieroperationen, mit denen s in t überführt werden kann.
- b) Es gibt keine kürzere Folge, d. h. für jede Folge E , die s in t überführt, gilt $M \leq |E|$.

Teil a:

- ❑ Nach Definition von D gibt es eine Folge mit $D(s', t)$ Editieroperationen, die s' in t überführt.
Wenn man diese Operationen auf s statt auf s' anwendet und anschließend noch das letzte Zeichen der resultierenden Zeichenkette entfernt, hat man s mit $D(s', t) + 1$ Operationen in t überführt.
- ❑ Ebenso gibt es eine Folge mit $D(s, t')$ Editieroperationen, die s in t' überführt.
Wenn man diese Operationen auf s anwendet und anschließend noch das letzte Zeichen von t am Ende der resultierenden Zeichenkette anfügt, hat man s mit $D(s, t') + 1$ Operationen in t überführt.
- ❑ Ebenso gibt es eine Folge mit $D(s', t')$ Editieroperationen, die s' in t' überführt.
Wenn man diese Operationen auf s statt auf s' anwendet und anschließend, wenn nötig, noch das letzte Zeichen der resultierenden Zeichenkette durch das letzte Zeichen von t ersetzt, hat man s mit $D(s', t') + \delta$ Operationen in t überführt.
- ❑ Somit gibt es Folgen zur Überführung von s in t mit $D(s', t) + 1$, $D(s, t') + 1$ und $D(s', t') + \delta$ Operationen.
Da M einer dieser drei Werte ist, gibt es somit auch eine Folge mit M Operationen.

Teil b:

- Sei E eine beliebige Folge von Editieroperationen, die s in t überführt.
- O. B. d. A. enthält E keine offensichtlich unnötigen Operationen.
(Andernfalls werden diese entfernt. Wenn dann für die verkürzte Folge $M \leq |E|$ gilt, gilt dies auch für die ursprüngliche Folge.)
- Fallunterscheidung nach der Herkunft des letzten Zeichens von t :
- Fall 1: Das letzte Zeichen von t ist das letzte Zeichen von s (wie im 6. Beispiel):
 - Dann kann man mit derselben Folge von Editieroperationen s' in t' überführen.
 - Daher gilt: $M \underset{(i)}{\leq} D(s', t') + \underset{(ii)}{\delta} = \underset{(iii)}{D(s', t')} \leq |E|$.
 - (i) $D(s', t') + \delta$ ist einer der drei Werte, deren Minimum M bezeichnet.
 - (ii) $\delta = 0$, da das letzte Zeichen von t gleich dem letzten Zeichen von s ist.
 - (iii) $D(s', t')$ ist gemäß Definition von D die *minimale* Anzahl von Editieroperationen, mit denen s' in t' überführt werden kann.

□ Fall 2:

Das letzte Zeichen von t ist durch Ersetzung des letzten Zeichens von s entstanden (wie im 5. Beispiel):

- Wenn man in der Folge E die entsprechende Ersetzungsoperation weglässt, erhält man eine Folge zur Überführung von s' in t' mit $|E| - 1$ Operationen.
- Daher gilt (mit ähnlichen Begründungen wie in Fall 1):
$$M \leq D(s', t') + \delta = D(s', t') + 1 \leq (|E| - 1) + 1 = |E|.$$

□ Fall 3: Das letzte Zeichen von t wurde während der Überführung hinzugefügt (wie im 1. Beispiel):

- Wenn man in der Folge E die entsprechende Einfügeoperation weglässt, erhält man eine Folge zur Überführung von s in t' mit $|E| - 1$ Operationen.
- Daher gilt: $M \leq D(s, t') + 1 \leq (|E| - 1) + 1 = |E|.$

- Fall 4: Keiner der obigen Fälle ist zutreffend, das heißt:
Das letzte Zeichen von t ist entweder ein Zeichen aus s , oder es ist durch eine Ersetzung eines Zeichens aus s entstanden, aber dieses Zeichen ist nicht das letzte Zeichen von s (wie im 3. Beispiel):
 - Dann müssen während der Überführung alle Zeichen von s hinter diesem Zeichen entfernt worden sein.
Insbesondere muss das letzte Zeichen von s irgendwann entfernt worden sein.
 - Wenn man in der Folge E die entsprechende Löschoption weglässt, erhält man eine Folge zur Überführung von s' in t mit $|E| - 1$ Operationen.
 - Daher gilt: $M \leq D(s', t) + 1 \leq (|E| - 1) + 1 = |E|$.

- Somit gilt in allen Fällen: $M \leq |E|$.

Praktische Berechnung

- Für zwei Zeichenketten s und t sei $D_{i,j} = D_{i,j}(s, t)$ die Editierdistanz zwischen dem Präfix der Länge i von s und dem Präfix der Länge j von t .
- Aufgrund der obigen Sätze können die Werte $D_{i,j}$ wie folgt berechnet werden:
 - $D_{0,0} = 0$
 - $D_{i,0} = i$ für $i = 1, \dots, |s|$
 - $D_{0,j} = j$ für $j = 1, \dots, |t|$
 - $D_{i,j} = \min \left\{ \begin{array}{l} D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \\ D_{i-1,j-1} + \delta \end{array} \right\}$ für $i = 1, \dots, |s|$ und $j = 1, \dots, |t|$
- $D(s, t)$ ist gleich $D_{|s|, |t|}(s, t)$.

Ermittlung der tatsächlichen Editieroperationen

- Bei der Berechnung eines Werts $D_{i,j}$ kann man notieren, aus welchem seiner drei „Nachbarn“ $D_{i-1,j}$, $D_{i,j-1}$ oder $D_{i-1,j-1}$ er „entstanden“ ist bzw. entstanden sein kann.

- ❑ Diese Information kann aber auch nachträglich durch Vergleich eines Werts mit seinen drei Nachbarn ermittelt werden.
- ❑ Wenn $D_{i,j}$ aus $D_{i-1,j}$ entstanden ist (oben), dann muss an dieser Stelle die Operation „Entferne das Zeichen s_i an Position i “ ausgeführt werden.
- ❑ Wenn $D_{i,j}$ aus $D_{i,j-1}$ entstanden ist (links), dann muss an dieser Stelle die Operation „Füge nach Position i das Zeichen t_j ein“ ausgeführt werden.
- ❑ Wenn $D_{i,j}$ aus $D_{i-1,j-1}$ entstanden ist (diagonal), dann muss an dieser Stelle entweder nichts gemacht werden (wenn $s_i = t_j$) oder die Operation „Ersetze das Zeichen s_i an Position i durch das Zeichen t_j “ ausgeführt werden (wenn $s_i \neq t_j$).
- ❑ Um die Folge aller Editieroperationen auszugeben, beginnt man mit $D_{|s|, |t|}$, ermittelt den passenden Nachbarn, gibt die zugehörige Operation aus und wiederholt den Vorgang mit diesem Nachbarn, bis man bei $D_{0,0}$ angekommen ist.
- ❑ Auf diese Weise entsteht die Folge der Editieroperationen automatisch von hinten nach vorn.
Will man die Operationen in umgekehrter Reihenfolge ausführen, muss man beachten, dass Einfügungen und Entfernungen die Positionen der nachfolgenden Zeichen bzw. Operationen verändern.

Laufzeit

$$\square O(|s| \cdot |t|)$$

Beispiel

$$\square s = \text{ABCD}, t = \text{ABDCE}$$

	<i>j</i>	0	1	2	3	4	5
<i>i</i>		<i>t</i>	A	B	D	C	E
0	<i>s</i>	0	1	2	3	4	5
1	A	1	0	1	2	3	4
2	B	2	1	0	1	2	3
3	C	3	2	1	1	1	2
4	D	4	3	2	1	2	2

$$\square \text{Ergebnis: } D(s, t) = D_{4,5} = 2$$

\square Die fettgedruckten Zahlen zeigen den „Weg“ von $D_{4,5}$ zurück zu $D_{0,0}$.

\square Die Editieroperationen zur Überführung von s in t lauten demnach:

- Ersetze das Zeichen an Position 4 durch das Zeichen E.
- Füge nach Position 2 das Zeichen D ein.

6.3 Blocksatz

6.3.1 Problemstellung

- Gegeben sei eine Folge von Wörtern w_1, \dots, w_n mit Breiten $\lambda_1, \dots, \lambda_n$, die in dieser Reihenfolge auf Zeilen der Breite λ verteilt werden sollen, sodass der zum Auffüllen der Zeilen zusätzlich benötigte Zwischenraum möglichst gleichmäßig verteilt ist (vgl. § 4.1 und § 4.2).
- Frage: Was bedeutet „möglichst gleichmäßig verteilt“, d. h. wie lautet eine sinnvolle Bewertungsfunktion für das Optimierungsproblem?
- Mögliche Antwort: Um einen großen Zwischenraum härter zu „bestrafen“ als mehrere kleine, soll die Summe der *Quadrate* aller Zwischenraumbreiten möglichst klein sein.

6.3.2 Malus einer einzelnen Zeile

- Wenn eine Zeile der Breite λ die Wörter w_i, \dots, w_j enthält und zwischen je zwei Wörtern ein Mindestabstand σ_0 ist, ist die Gesamtbreite des zusätzlich vorhandenen Zwischenraums in dieser Zeile: $\sigma = \lambda - \sum_{k=i}^j \lambda_k - m \sigma_0$ mit $m = j - i =$ Anzahl der Wortzwischenräume in dieser Zeile (d. h. 1 weniger als die Anzahl der Wörter).

- Wenn dieser Zwischenraum gleichmäßig verteilt wird, ergibt sich als Summe der

$$\text{Quadrate der Einzelzwischenräume: } \mu(i, j) = \begin{cases} m \left(\frac{\sigma}{m} \right)^2 = \frac{\sigma^2}{m} & \text{für } m > 0 \quad (*) \\ \sigma^2 & \text{für } m = 0 \quad (**) \end{cases}$$

(*) Bei ganzzahliger Rechnung: $(m - r) \cdot q^2 + r \cdot (q + 1)^2$ mit $q = \left\lfloor \frac{\sigma}{m} \right\rfloor$ und $r = \sigma \bmod m$

(**) Wenn die Zeile nur ein Wort enthält ($i = j \Leftrightarrow m = 0$), befindet sich der gesamte Zwischenraum σ am Ende der Zeile.

- Sonderfall: Übervolle Zeile

- Wenn $\sigma < 0$ ist, ist die Zeile *übervoll*.
- Im Fall $i = j$ lässt sich dies nicht vermeiden, weil die Breite λ_i des einzigen Worts auf der Zeile dann größer als die Zeilenbreite λ ist. In diesem Fall sei $\mu(i, j) = 0$.
- Im Fall $i < j$ ist die Zeile *unnötig übervoll*, d. h. die Anzahl ihrer Wörter sollte reduziert werden. In diesem Fall sei $\mu(i, j) = \infty$.

- Sonderfall: Letzte Zeile

Da Zwischenraum in der letzten Zeile (normalerweise) nicht bestraft werden soll, sei $\mu(i, j) = 0$ für $j = n$, sofern $\sigma \geq 0$ ist.

- $\mu(i, j)$ heißt *Malus* der Zeile mit den Wörtern w_i, \dots, w_j .

6.3.3 Malus eines gesamten Absatzes

- Für $0 < i_1 < \dots < i_k = j \leq n$ ist $\mu(\{i_1, \dots, i_k\}) = \mu(1, i_1) + \mu(i_1 + 1, i_2) + \dots + \mu(i_{k-1} + 1, i_k)$ der Gesamtmalus aller Zeilen, der sich ergibt, wenn man die Wörter w_1, \dots, w_j so auf Zeilen verteilt, dass nach den Wörtern i_1, \dots, i_k jeweils ein Zeilenumbruch erfolgt.
- Für $j = 1, \dots, n$ sei $\mu(j) = \min \{ \mu(\{i_1, \dots, i_k\}) \mid 0 < i_1 < \dots < i_k = j \}$,
d. h. $\mu(j)$ ist der kleinste Malus, den man erreichen kann, wenn man die Wörter w_1, \dots, w_j irgendwie auf Zeilen verteilt.
- Zusätzlich sei $\mu(0) = 0$.
- Gesucht ist somit $\mu(n)$ sowie die Indizes $0 < i_1 < \dots < i_k = n$ der Wörter, nach denen jeweils ein Zeilenumbruch erfolgen soll, um diesen minimalen Malus zu erreichen.

6.3.4 Rekursionsgleichung

- Um die ersten j Wörter w_1, \dots, w_j beliebig auf Zeilen zu verteilen,
 - wählt man eine beliebige Stelle $i \in \{0, \dots, j - 1\}$ für den letzten Zeilenumbruch (d. h. dieser soll nach Wort i erfolgen),
 - verteilt die ersten i Wörter beliebig auf Zeilen
 - und setzt die restlichen Wörter $i + 1, \dots, j$ auf die letzte Zeile.

- Um die ersten j Wörter *optimal* (d. h. mit minimalem Malus) auf Zeilen zu verteilen,
 - wählt man für die Verteilung der ersten i Wörter jeweils eine Verteilung mit minimalem Malus $\mu(i)$,
 - addiert dazu den Malus $\mu(i + 1, j)$ der resultierenden letzten Zeile
 - und wählt von diesen Möglichkeiten mit Gesamtmalus $\mu(i) + \mu(i + 1, j)$ für $i = 0, \dots, j - 1$ das Minimum.

- Somit gilt für $j = 1, \dots, n$:
$$\mu(j) = \min \{ \mu(i) + \mu(i + 1, j) \mid i = 0, \dots, j - 1 \}.$$

6.3.5 Praktische Berechnung

- Für die praktische Berechnung des Minimums $\mu(j)$ lässt man i rückwärts von $j - 1$ bis 0 laufen und beendet die Schleife, sobald $\mu(i + 1, j) = \infty$ ist, d. h. wenn die letzte Zeile unnötig übervoll wäre, weil $\mu(i + 1, j)$ dann für alle weiteren Werte von i ebenfalls ∞ wäre und sich somit kein kleinerer Gesamtwert mehr ergeben würde.

- Damit kann $\mu(j)$ nacheinander für $j = 0, \dots, n$ wie folgt berechnet und in einer Tabelle gespeichert werden:
 - 1 Setze $\mu(0) = 0$.
 - 2 Für $j = 1, \dots, n$:
 - 1 Setze $\mu(j) = \infty$.
 - 2 Für $i = j - 1, \dots, 0$:
 - 1 Berechne $\mu = \mu(i + 1, j)$.
 - 2 Wenn $\mu = \infty$ ist, beende die Schleife.
 - 3 Setze $\mu = \mu(i) + \mu$.
 - 4 Wenn $\mu < \mu(j)$ ist: Setze $\mu(j) = \mu$ und $\pi(j) = i$.

- Die optimalen Umbruchstellen i_k, \dots, i_1 erhält man „rückwärts“ als $n, \pi(n), \pi(\pi(n)), \dots$

6.3.6 Beispiel

- Wenn man den Text `a b c d e f g h i j` in „Schreibmaschinenschrift“ (alle Zeichen einschließlich Leerzeichen besitzen Breite 1) bei Zeilenbreite 7 mit dem in § 4.2 beschriebenen Nächstbest-Algorithmus formatiert, ergibt sich:

```

a b c d
e     f
g h i j
    
```

- Die Anwendung des zuvor beschriebenen Algorithmus ergibt:

j	i $\pi(j)$	Letzte Zeile (Wort $i + 1$ bis j)	$\mu(i + 1, j)$	$\mu(i)$	$\mu(i) + \mu(i + 1, j)$ $\mu(j)$
0					0
1	0	a++++++	$6^2 = 36$	0	36
2	1	b++++++	$6^2 = 36$	36	72
	0	a-++++b	$4^2 = 16$	0	16
3	2	c++++++	$6^2 = 36$	16	52
	1	b-++++c	$4^2 = 16$	36	52
	0	a-+b-+c	$1^2 + 1^2 = 2$	0	2

j	i $\pi(j)$	Letzte Zeile (Wort $i + 1$ bis j)	$\mu(i + 1, j)$	$\mu(i)$	$\mu(i) + \mu(i + 1, j)$ $\mu(j)$
4	3	d+++++	$6^2 = 36$	2	38
	2	c-++++d	$4^2 = 16$	16	32
	1	b-+c-+d	$1^2 + 1^2 = 2$	36	38
	0	a-b-c-d	0	0	0
5	4	e+++++	$6^2 = 36$	0	36
	3	d-++++e	$4^2 = 16$	2	18
	2	c-+d-+e	$1^2 + 1^2 = 2$	16	18
	1	b-c-d-e	0	36	36
	0	a-b-c-d-e	∞		
6	5	f+++++	$6^2 = 36$	18	54
	4	e-++++f	$4^2 = 16$	0	16
	3	d-+e-+f	$1^2 + 1^2 = 2$	2	4
	2	c-d-e-f	0	16	16
	1	b-c-d-e-f	∞		
7	6	ghij	0	4	4
	5	f-ghij	0	18	18
	4	e-f-ghij	∞		

□ Erläuterung zu den Tabellen:

- Für jeden Wert von j läuft i rückwärts von $j - 1$ bis 0 , solange $\mu(i + 1, j) < \infty$ ist.
- Die fettgedruckten Werte in der zweiten bzw. letzten Spalte entsprechen $\pi(j)$ bzw. $\mu(j)$.
- In der dritten Spalte symbolisieren Minuszeichen den Mindestabstand zwischen zwei Wörtern und Pluszeichen den zusätzlich benötigten Zwischenraum, der für die Berechnung von $\mu(i + 1, j)$ verwendet wird.

□ Die optimalen Umbruchstellen sind: $n = 7$, $\pi(7) = 6$, $\pi(6) = 3$, $\pi(3) = 0$

□ Damit ergibt sich folgende optimale Formatierung:

```
a  b  c
d  e  f
ghij
```

6.3.7 Laufzeit

- ❑ Da zwischen je zwei Wörtern ein Mindestabstand der Breite σ_0 sein soll, haben auf einer Zeile der Breite λ maximal $m = \left\lfloor \frac{\lambda}{\sigma_0} \right\rfloor + 1$ Wörter Platz.
- ❑ Deshalb wird die innere Schleife zur Berechnung von $\mu(j)$ nach höchstens $m = O(1)$ Durchläufen beendet.
- ❑ Somit ist die Gesamtlaufzeit zur Berechnung von $\mu(0), \dots, \mu(n)$ gleich $O(n)$ (und nicht $O(n^2)$, wie in https://en.wikipedia.org/wiki/Line_wrap_and_word_wrap behauptet wird).
- ❑ Zum Vergleich: Die Gesamtzahl der Möglichkeiten, n Wörter auf Zeilen zu verteilen, ist 2^{n-1} : Nach jedem Wort außer dem letzten kann entweder ein Zeilenumbruch erfolgen oder nicht.
- ❑ Also führt die tabellengestützte Programmierung zu einer immensen Laufzeitverbesserung.

6.3.8 Erweiterungsmöglichkeiten

- ❑ Trennung langer Wörter
 - Silbentrennung (mit unterschiedlich „guten“ Trennstellen)
 - Trennung vor oder nach bestimmten Zeichen (z. B. nach Schrägstrichen in URLs)
 - Trennung mathematischer Formeln
- ❑ Zusätzliche Strafpunkte für einzelne Trennungen oder mehrere aufeinanderfolgende Zeilen mit Trennungen
- ❑ Zeilen mit einem einzigen Wort noch „härter bestrafen“
- ❑ Verbotene und bevorzugte Zeilenumbrüche
- ❑ Einen Absatz künstlich um eine oder mehrere Zeilen verlängern, um
 - „Schusterjungen“ und „Hurenkinder“ zu vermeiden
 - alle Spalten auf der letzten Seite eines mehrspaltigen Texts gleich lang zu machen
- ❑ Aber: Auch der beste Algorithmus kann nicht „zaubern“. Manchmal muss man einen Text einfach umformulieren, um einen störenden Umbruch zu vermeiden.

derit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Diese Zeile ist ein Schusterjunge, aber es kommt noch

schlimmer: Diese ist ein Hurenkind.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.