

Algorithmentheorie  
Vorlesungsskript Sommersemester 2004

Zoltán Ádám Mann und András Orbán

29. April 2004

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>4</b>
<b>2</b>	<b>Suchen, Einfügen, Sortieren</b>	<b>6</b>
2.1	Ordnungrelation . . . . .	6
2.2	Suchen . . . . .	6
2.2.1	Lineare Suche . . . . .	7
2.2.2	Binäre (logarithmische) Suche . . . . .	7
2.2.3	Bemerkungen . . . . .	7
2.3	Einfügen . . . . .	7
2.4	Mischen (merge, összefésülés) . . . . .	8
2.4.1	Parallelisierung von Mischen: Batchersches Verfahren . . . . .	8
2.5	Sortieren (Ordnen) . . . . .	9
2.5.1	Eine untere Schranke . . . . .	9
2.5.2	Sortieren durch Tauschen (bubble sort) . . . . .	9
2.5.3	Sortieren durch Einfügen . . . . .	10
2.5.4	Sortieren durch Mischen (merge sort) . . . . .	10
2.5.5	Heap sort . . . . .	11
2.5.6	Quick sort . . . . .	13
2.6	Sortierung mit Schlüsselmanipulation . . . . .	14
2.6.1	Kistensortierung . . . . .	14
2.6.2	Radix-Sortierung . . . . .	14
2.7	Sortierung des Inhaltes von Hintergrundspeichern . . . . .	15
<b>3</b>	<b>Suchbaum (Searchtree, keresőfa)</b>	<b>16</b>
3.1	Binärer Baum . . . . .	16
3.2	Binärer Suchbaum . . . . .	16
3.3	2-3 Bäume . . . . .	17
3.4	B-Bäume . . . . .	18
3.5	AVL-Bäume . . . . .	18

3.6	Andere balancierte Bäume . . . . .	20
3.6.1	HB[k]-Bäume . . . . .	20
3.6.2	Gewichtet balancierte Bäume . . . . .	21
3.7	Eine weitere Anwendung . . . . .	21
3.8	S-Bäume . . . . .	22
3.9	Wortbäume (tries) . . . . .	23
<b>4</b>	<b>Gestreute Speicherung (Hashing)</b>	<b>24</b>
4.1	Speicherung in geketteten Listen . . . . .	24
4.2	Offene Adressierung . . . . .	25
4.2.1	Lineares Sondieren . . . . .	25
4.2.2	Quasi-zufälliges Sondieren . . . . .	26
4.2.3	Doppelhashing . . . . .	26
4.3	Hashfunktionen . . . . .	27
4.3.1	Modulo-Methode . . . . .	27
4.3.2	Multiplikationsmethode . . . . .	27
4.3.3	Die zweite Hashfunktion beim Doppelhashing . . . . .	27
4.4	Suchbäume oder Hashing? . . . . .	27
<b>5</b>	<b>Datenkomprimierung</b>	<b>28</b>
5.1	Huffman-Codes . . . . .	28
5.2	Lempel-Ziv-Welch-Methode . . . . .	29
<b>6</b>	<b>Graphentheoretische Algorithmen</b>	<b>31</b>
6.1	Einleitung . . . . .	31
6.1.1	Grundbegriffe und Notationen . . . . .	31
6.1.2	Speicherung von Graphen . . . . .	31
6.2	Kürzeste Wege . . . . .	32
6.2.1	Algorithmus von Dijkstra . . . . .	33
6.2.2	Die Methode von Bellman und Ford . . . . .	34
6.2.3	Algorithmus von Floyd . . . . .	35
6.3	Tiefensuche . . . . .	36
6.3.1	Gerichtete azyklische Graphen . . . . .	38
6.3.2	Stark zusammenhängende Komponenten . . . . .	39
6.4	Breitensuche . . . . .	41
6.5	Minimaler Spannender Baum . . . . .	41
6.5.1	Rot-blau Algorithmus . . . . .	41
6.5.2	Implementierung des Primschen Verfahrens . . . . .	42
6.5.3	Implementierung des Kruskalschen Verfahrens . . . . .	42
6.6	Maximale Paarungen in bipartiten Graphen . . . . .	44
6.7	Maximaler Fluss und minimale Schnittmenge . . . . .	44

<b>7</b>	<b>Turing-Maschinen</b>	<b>45</b>
7.1	Grundlegende Definitionen . . . . .	45
7.2	Simulationen . . . . .	47
7.3	Berechenbarkeit . . . . .	49
7.3.1	Definitionen . . . . .	49
7.3.2	Beziehung zwischen den Berechenbarkeitsbegriffen . . . . .	50
7.3.3	Weitere unentscheidbare Probleme . . . . .	52
7.4	RAM . . . . .	54
7.4.1	Einführung . . . . .	54
7.4.2	Kost einer RAM . . . . .	55
7.4.3	Simulation . . . . .	55
7.5	Kolmogorov-Komplexität . . . . .	55
7.5.1	Definition . . . . .	55
7.5.2	Einige Eigenschaften der Kolmogorov-Komplexität . . . . .	56
7.5.3	Anwendungen . . . . .	57
<b>8</b>	<b>Die <math>\mathcal{NP}</math> Sprachklasse</b>	<b>58</b>
8.1	Zeit- und speicher-begrenzte Maschinen, Sprachklassen . . . . .	58
8.2	Nicht-deterministische Turing-Maschinen (NTM) . . . . .	60
8.3	Einige Sprachen aus $\mathcal{NP}$ . . . . .	62
8.3.1	Färbung mit 3 Farben . . . . .	62
8.3.2	Hamilton-Kreis . . . . .	62
8.3.3	Planare Graphen . . . . .	62
8.3.4	Ein nicht-triviales Beispiel: Primzahlen . . . . .	63
8.3.5	Ein Beispiel für eine Funktion: Primfaktorzerlegung . . . . .	63
8.4	Karp-Reduktion . . . . .	64
8.5	Der Satz von Cook und Levin . . . . .	65
8.6	Weitere $\mathcal{NP}$ -vollständige Sprachen . . . . .	66
8.6.1	Formeln in konjunktiver Normalform . . . . .	66
8.6.2	Färbung mit 3 Farben . . . . .	67
8.6.3	Maximale unabhängige Punktmenge . . . . .	68
8.6.4	3-dimensionale Paarung . . . . .	69
8.6.5	Travelling Salesman Problem . . . . .	69
8.6.6	Das Rucksack Problem und verwandte Probleme . . . . .	69
8.7	$\mathcal{NPI}$ . . . . .	70
<b>9</b>	<b>Allgemeine Methoden für Planung von Algorithmen</b>	<b>71</b>
9.1	Branch and bound . . . . .	71
9.2	Dynamische Programmierung . . . . .	72
9.3	Näherungsmethoden . . . . .	73

# Kapitel 1

## Einleitung

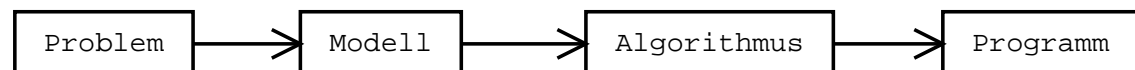
Dieses Vorlesungsskript basiert auf [1].

**Definition 1.1.** *Algorithmus (ungenau Definition; eine präzise Definition basierend auf Turing-Maschinen folgt später): Ein Algorithmus ist ein Verfahren, das formal genug spezifiziert ist, um maschinell ausgeführt werden zu können.*

Ursprung des Namen: das Buch von Al Khvarizmi (arabischer Mathematiker, IX. Jahrhundert) über einfache arithmetische Operationen.

Heute: Algorithmus  $\leftrightarrow$  Programm. Genauer: Algorithmen sind funktionale Bausteine von Programmen. (Aber hängen sehr stark mit Datenstrukturen zusammen  $\rightarrow$  OOP.) Bei grossen Programmen gibt es schon andere Probleme  $\rightarrow$  Softwaretechnik.

Schema der Modellbildung:



Im ersten Schritt wird das Problem formalisiert, präzisiert und vereinfacht. Mit dem zweiten Schritt werden wir uns befassen. Der dritte Schritt ist „nur noch“ Programmierung. Zum Algorithmus gehört immer auch eine Datenstruktur. Der fertige Algorithmus soll auch analysiert werden (Komplexität). Rücktritt und Iterationen sind auch möglich.

Die wichtigsten Fragen:

- Korrektheit
- Effizienz (Komplexität)
  - Zeitbedarf
  - Speicherbedarf
  - ...

Meistens interessiert uns der Zeitbedarf (Anzahl der Schritte). Sollte möglichst rechnerunabhängig sein. Die genauen Konstanten sind rechnerabhängig, deshalb sind sie nicht so interessant, nur die Grössenordnung. Der Zeitbedarf ist eine Funktion  $f(n)$  der Länge der Eingabe. Meistens interessiert uns der schlechteste Fall (worst case), manchmal können wir auch etwas über den Durchschnitt sagen. Manchmal kennen wir auch eine untere Schranke für die Komplexität.

**Definition 1.2.**  $f, g : \mathbb{N} \rightarrow \mathbb{R}$ .

$$f = \mathcal{O}(g) \iff \exists c, n_0 > 0 \quad \forall n \geq n_0 \quad |f(n)| \leq c|g(n)|$$

$$f = \Omega(g) \iff \exists c, n_0 > 0 \quad \forall n \geq n_0 \quad |f(n)| \geq c|g(n)|$$

$$f = \Theta(g) \iff f = \mathcal{O}(g) \text{ und } f = \Omega(g)$$

$$f = o(g) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

Themen:

1. Konkrete Algorithmen und Datenstrukturen (an sich wichtige Beispiele, die aber auch den Prozess der Algorithmenerstellung illustrieren)
  - Suchen, Einfügen, Sortieren
  - Datenstrukturen: Array, gekettete Liste, Queue, Stack, Suchbaum, Heap, Hash-Tabelle
  - Graphalgorithmen (Graphdurchlauf, Baumtraversierung, kürzeste Wege, Spannbäume, Paarungen, Netzwerkflüsse)
2. Theorie
  - Turing-Maschinen
  - Grundbegriffe der Berechenbarkeit (rekursive und rekursiv aufzählbare Sprachen und Mengen, rekursive und partiell rekursive Funktionen)
  - Negative Ergebnisse: Beispiele der Nicht-Berechenbarkeit
  - Simulationen, universale Turing-Maschine, Beschleunigungsmöglichkeiten
  - Kolmogorov-Komplexität
  - Andere Maschinenmodelle: RAM
  - Die wichtigsten Komplexitätsklassen ( $\mathcal{P}$ ,  $\mathcal{NP}$ ,  $\mathcal{NP}$ -Vollständigkeit, Karp-Reduktion)
  - Bekannte  $\mathcal{NP}$ -vollständige und  $\mathcal{NP}$ -schwere Probleme
3. Allgemeine Methoden
  - Gierige Algorithmen (greedy)
  - Branch-and-bound
  - Teile und herrsche (divide and conquer)
  - Dynamische Programmierung
  - Heuristische Algorithmen
  - Randomisierte Algorithmen
  - Parallele Algorithmen

# Kapitel 2

## Suchen, Einfügen, Sortieren

### 2.1 Ordnungrelation

Eine Ordnungsrelation ( $<$ ,  $\leq$ ) hat die folgende Eigenschaften:

- $x \not< x$  (irreflexiv);  $x \leq x$  (reflexiv)
- $x < y \wedge y < z \rightarrow x < z$ ;  $x \leq y \wedge y \leq z \rightarrow x \leq z$  (transitiv)
- $x < y \vee x = y \vee y < x$ , immer genau eine der drei Möglichkeiten;  $x \leq y \vee y \leq x$ , falls beide, dann  $x = y$  (lineare, oder vollständige Ordnung)

$(U, <)$ : geordnete Menge, bzw. geordneter Typ. Beispiele:

- Zahlen
- Zeichen (Reihenfolge  $\leftarrow$  ASCII-Tabelle)
- Zeichenketten: lexikographische Ordnung

Die Ordnung kann die Suche vereinfachen (z. B. Telefonbuch).

Im folgenden wird angenommen, dass zwei Elemente in einem elementaren Schritt verglichen werden können.

### 2.2 Suchen

Wichtige Aufgabe z. B. in Datenbanksystemen, Informationssystemen.

Input:  $M = \{s_1, \dots, s_n\}$  und  $s$  (Elemente aus der geordneten Menge  $U$ ) in einer linearen Datenstruktur (z. B. Array, gekettete Liste).

Frage: Ist  $s \in M$ ? Falls ja, an welcher Stelle?

Schritt=Vergleich.

### 2.2.1 Lineare Suche

Vergleich mit allen Elementen in der Reihenfolge  $s_1, \dots, s_n$ , bis  $s$  gefunden wird. Wenn die Daten geordnet sind und  $s_i > s$  wird gefunden, dann kann die Suche unterbrochen werden.

Im schlimmsten Fall:  $n$  Schritte (sowohl für ungeordnete als auch geordnete Daten). Im Durchschnitt für geordnete Daten (Annahme über die Verteilung: jedes Intervall mit gleicher Wahrscheinlichkeit):

$$\frac{1 + 2 + \dots + (n-1) + n + n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

### 2.2.2 Binäre (logarithmische) Suche

„Teile-und-herrsche“-Prinzip; funktioniert nur für geordnete Daten. Vergleich zuerst mit dem mittleren Element ( $s_k$  wenn  $n = 2k + 1$  und  $s_k$  oder  $s_{k+1}$  wenn  $n = 2k$ ). Falls  $s = s_{\text{mittel}}$ , dann fertig. Falls  $s < s_{\text{mittel}}$ , dann muss nur noch die Teilmenge  $M_1 = \{s_1, \dots, s_{\text{mittel}-1}\}$  durchsucht werden. Falls  $s > s_{\text{mittel}}$ , dann muss nur noch die Teilmenge  $M_1 = \{s_{\text{mittel}+1}, \dots, s_n\}$  durchsucht werden. Usw.

Somit wird die Menge der Kandidaten in jedem Schritt halbiert, also wird es nach  $\lfloor \log_2 n \rfloor$  Schritten nur noch einen einzigen Kandidaten geben, den man überprüfen soll (wenn das Element nicht schon früher gefunden wurde). Also braucht man insgesamt höchstens  $1 + \lfloor \log_2 n \rfloor$  Vergleiche.

Das ist auch optimal, was man mit der Gegner-Methode beweisen kann: man stelle sich vor, dass der Gegner immer so antwortet, dass immer nur die kleinere Hälfte ausgeschlossen werden kann.

### 2.2.3 Bemerkungen

Die binäre Suche ist viel schneller als die lineare Suche. Aber:

- Die lineare Suche funktioniert auch bei nicht geordneten Daten.
- Für die binäre Suche es ist notwendig, dass man sehr schnell das mittlere Element auswählen kann. Das ist nicht erfüllt bei z. B. geketteten Listen, oder externen Dateien, die nur sequentiell gelesen werden können (z. B. Magnetband).
- Die lineare Suche ist einfacher zu implementieren.

Also sollte binäre Suche typischerweise bei grossen, geordneten Arrays verwendet werden.

Variationen: Wiederholungen sind erlaubt oder nicht. Wenn alle gefunden werden müssen, dann hängt der Zeitbedarf auch von deren Anzahl ab. Schlüssel: Suche basiert auf ein einziges Element der Records.

Interpolationssuche: mit mehr Information über die Verteilung kann man noch schneller suchen, z. B. im Telefonbuch.

## 2.3 Einfügen

Input:  $M = \{s_1, \dots, s_n\}$  und  $s$  aus  $U$ , sowie die Position (Index oder Zeiger)  $k$ .

Aufgabe: füge  $s$  in  $M$  in Position  $k$  ein.

Hängt stark von der Datenstruktur ab. Bei einer geketteten Liste  $\mathcal{O}(1)$  Schritte, bei einem Array  $n + 1 - k$  Bewegungen nötig. Das sind im schlimmsten Fall  $n$  Schritte, im Durchschnitt etwa  $n/2$  Schritte.

Suchen und Einfügen zusammen können benutzt werden, um ein Element in eine geordnete Datenstruktur in seine korrekte Position einzufügen. Bemerke, dass für Suchen ein Array besser ist, für Einfügen aber eine gekettete Liste:

	Suchen	Einfügen
Array	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Gekettete Liste	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Deswegen werden wir später komplexere Datenstrukturen kennenlernen, in denen sowohl Suchen als auch Einfügen schnell ist.

## 2.4 Mischen (merge, összefésülés)

Aus zwei geordneten Mengen ( $A[1 : l]$  und  $B[1 : m]$ ) eine einzige. Falls  $A[1 : i]$  und  $B[1 : j]$  schon gemischt sind ( $C[1 : i + j]$ ), dann ist

$$C[i + j + 1] = \min(A[i + 1], B[j + 1])$$

Dazu braucht man  $l + m - 1$  Vergleiche (beim letzten Element ist es schon eindeutig) und  $l + m$  Bewegungen. Sehr effektiv. Braucht relativ viel Arbeitsspeicher; das kann verändert werden, aber dann wird der Algorithmus komplizierter.

### 2.4.1 Parallelisierung von Mischen: Batcher'sches Verfahren

Frage: wie kann man Mischen beschleunigen, wenn man beliebig viele Prozessoren benutzen kann?

Seien  $A = a_1 < \dots < a_l$  und  $B = b_1 < \dots < b_m$ .

$$u_1 < u_2 < \dots := \text{MERGE}(a_1 < a_3 < \dots, b_2 < b_4 < \dots)$$

$$v_1 < v_2 < \dots := \text{MERGE}(a_2 < a_4 < \dots, b_1 < b_3 < \dots)$$

**Satz 2.1.**

$$c_{2i-1} = \min\{u_i, v_i\} \quad c_{2i} = \max\{u_i, v_i\}$$

für jedes  $1 \leq i \leq \frac{l+m}{2}$

*Beweis.* Zuerst behaupten wir, dass

$$C_k := \{c_1, \dots, c_{2k}\} = \{u_1, \dots, u_k\} \cup \{v_1, \dots, v_k\}$$

für jedes  $1 \leq k \leq \frac{l+m}{2}$ . Sei

$$\{c_1, \dots, c_{2k}\} = \{a_1, \dots, a_s\} \cup \{b_1, \dots, b_{2k-s}\}$$

Dann gilt, dass  $\lceil \frac{s}{2} \rceil$   $u_i$  Elemente von  $C_k$  gehören zu  $A$  und  $\lfloor \frac{2k-s}{2} \rfloor$  zu  $B$ . Also in  $C_k$  sind insgesamt  $\lceil \frac{s}{2} \rceil + \lfloor \frac{2k-s}{2} \rfloor$   $u_i$  Elemente. Ähnlicherweise ist die Anzahl von  $v_j$  Elementen in  $C_k$ :  $\lfloor \frac{s}{2} \rfloor + \lceil \frac{2k-s}{2} \rceil$ . Die sind aber gleich.

Daraus folgt, dass  $\{c_1, \dots, c_{2k}\} = \{u_1, \dots, u_k\} \cup \{v_1, \dots, v_k\}$  und  $\{c_1, \dots, c_{2(k-1)}\} = \{u_1, \dots, u_{k-1}\} \cup \{v_1, \dots, v_{k-1}\}$ . Also  $\{c_{2k-1}, c_{2k}\} = \{u_k, v_k\}$ . Da  $c_{2k-1} \leq c_{2k}$ , ist der Beweis fertig.  $\square$

Also, um  $A$  und  $B$  zu mischen, brauchen wir zwei Mischoperationen an halb so grossen Eingaben durchzuführen (das geht parallel und rekursiv), und danach können wir das Endergebnis in einem Schritt bekommen (falls wir genug Prozessoren haben). Also, wenn  $T(n)$  die Anzahl der Schritte bedeutet, dann gilt die folgende Rekursion:  $T(n) = T(n/2) + 1$  und  $T(1) = 0$ . Die Lösung dieser Rekursion ist  $T(n) = \log_2 n$ .

## 2.5 Sortieren (Ordnen)

Einfache Sortierverfahren: keine Information über die Schlüssel, nur basierend auf Vergleiche.

Schritt: Vergleich, Bewegung.

Input:  $A[1 : n]$ , mit Elementen aus  $U$ .

Output:  $A[1 : n]$  sortiert gemäss „ $\leq$ “.

### 2.5.1 Eine untere Schranke

**Satz 2.2.** *Wenn ein einfacher Sortieralgorithmus  $n$  Elemente mit  $k$  Vergleichen sortieren kann, dann gilt  $k \geq \log_2(n!)$*

*Beweis.* Einfacher Sortieralgorithmus besteht aus solchen Schritten:

```
if Vergleich then Aktion1 else Aktion2
```

wobei die Aktionen beliebig komplizierte Bewegungen oder goto-Befehle sein können.

Zu jeder möglichen Reihenfolge der Zahlen  $s_1, \dots, s_n$  schreiben wir das Ergebnis der (höchstens  $k$ ) Vergleiche auf, so bekommt man ein Codewort aus *ja* und *nein* der Länge  $k$  für jede solche Reihenfolge. Da die Bewegungen ein-eindeutig und deshalb invertierbar sind, ist es nicht möglich, dass das Code-Wort für zwei verschiedene Reihenfolgen gleich wäre (dann wäre nämlich die Folge der Aktionen auch gleich).

Wenn ein Codewort kürzer ist als  $k$ , können wir es mit *ja*-s ergänzen; das gibt keine Probleme, weil kein Codewort ein Präfix eines anderen ist. (Denn, solange die Ergebnisse der Vergleiche gleich sind, sind die Aktionen auch gleich.)

Es folgt:  $2^k \geq n!$ , also  $k \geq \log_2(n!)$  □

Laut der Stirling-Formel:

$$n! \approx (n/e)^n \sqrt{2\pi n} \rightarrow \log_2(n!) \approx n(\log_2 n - \log_2 e) + \frac{1}{2} \log_2 n + \log_2 \sqrt{2\pi} \approx n(\log_2 n - 1,442)$$

### 2.5.2 Sortieren durch Tauschen (bubble sort)

Idee: wenn  $A[i] > A[i+1]$ , tauschen. Wenn kein solches Paar, dann fertig. Erste Iteration:  $i = 1, \dots, n-1$ . Danach ist das Maximum in  $A[n]$  (das Maximum ist aufgestiegen, wie eine Blase, deswegen der Name). Zweite Iteration:  $i = 1, \dots, n-2$ , usw.:

```
procedure BubbleSort
for (j=n-1, j>0, j=j-1) do
  for (i=1, i<j, i=i+1) do
    if (A[i]>A[i+1]) then swap(A[i],A[i+1])
```

Anzahl der Vergleiche:

$$(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2}$$

Im schlimmsten Fall ( $A[1] > A[2] > \dots > A[n]$ ) ist die Anzahl der Tauschoperationen das gleiche.

Also: ziemlich langsam. Aber einfach, und funktioniert auch für Listen. Konservativ: gleiche Elemente werden nicht getauscht.

### 2.5.3 Sortieren durch Einfügen

$A[1 : 1]$  ist geordnet. Mit Suchen und Einfügen kann garantiert werden, dass wenn  $A[1 : k]$  geordnet war, dann wird auch  $A[1 : k + 1]$  geordnet sein. Nach  $n$  Iterationen:  $A[1 : n]$  ist geordnet.

Wenn man lineare Suche benutzt: der schlimmste Fall ist, wenn die Daten schon am Anfang richtig geordnet waren! In diesem Fall ist die Anzahl der Vergleiche

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2}$$

Bei einem Array braucht man  $\frac{(n+2)(n-1)}{2}$  Bewegungen (schlimmster Fall: umgekehrt geordnete Daten.)

Im Durchschnitt braucht man etwa  $\frac{n(n-1)}{4}$  Vergleiche und auch etwa so viele Bewegungen.

Wenn man binäre Suche benutzt, dann ist die Anzahl der nötigen Vergleiche etwa

$$(1 + \log_2 2) + \dots + (1 + \log_2 n) = n - 1 + \log_2(n!) \approx n(\log_2 n - 0.442)$$

was fast optimal ist (siehe 2.5.1). Aber die Anzahl der Bewegungen ist im schlimmsten Fall genauso schlecht, wie beim anderen Verfahren.

### 2.5.4 Sortieren durch Mischen (merge sort)

„Teile und herrsche“-Prinzip. Sortiere die erste und die zweite Hälfte, und dann mische:

$\text{MSORT}(A[i : i]) = \text{NOP}$

$\text{MSORT}(A[i : j]) = \text{MERGE}(\text{MSORT}(A[i : (i+j)/2]), \text{MSORT}(A[(i+j)/2+1 : j]))$

Sei  $n = 2^k$ ,  $T(n)$  die Anzahl der Vergleiche im schlimmsten Fall. Dann gilt

$$T(n) \leq n - 1 + 2T(n/2) \leq n - 1 + 2(n/2 - 1 + 2T(n/4)) = n - 1 + 2(n/2 - 1) + 4T(n/4)$$

Usw. ( $T(1) = 0$ ):

$$T(n) \leq n - 1 + 2(n/2 - 1) + 4(n/4 - 1) + \dots + 2^{k-1}(n/2^{k-1} - 1) \leq n \log_2 n$$

Das kann auch für andere Zahlen verallgemeinert werden (nicht nur Potenzen von 2).

Statt Rekursion kann man auch Iteration benutzen.

In jeder Phase der Rekursion gibt es etwa  $2n$  Bewegungen, und es gibt  $\log_2 n$  Phasen, also ist die Anzahl der Bewegungen höchstens  $2n \log_2 n$ . Der Speicherbedarf ist  $2n$ , wenn man die Ergebnismenge in einen Hilfsarray schreibt. ( $n + \mathcal{O}(1)$  ist auch möglich.)

Insgesamt:  $\mathcal{O}(n \log n)$ , was – von der Größenordnung her – optimal ist.

Parallelisierung: die untere und obere Hälfte kann man parallel sortieren, danach kann man mit dem Batcher'schen Verfahren mischen. Also  $T_{par}(n) \leq T_{par}(n/2) + \log_2 n$ , woraus  $T_{par}(n) \leq \log_2^2 n$  folgt.

## 2.5.5 Heap sort

### Die abstrakte Datenstruktur Heap

Elemente  $s_1, \dots, s_n$  aus  $(U, <)$

Operationen:

void addElement(Element  $s$ )

Element removeMin()

Wichtige Anwendungen:

- Prioritätsschlange (priority queue) in Betriebssystemen
- Mischen von vielen geordneten Datenmengen
- Heap sort

### Eine effiziente Implementierung: Binäres Heap

**Definition 2.1.** *Binärer Baum:* Baum; die Knotenpunkte sind auf Etagen verteilt. In der ersten Etage ist ein einziger Knotenpunkt: die Wurzel. Jeder Knotenpunkt hat höchstens zwei Söhne: einen linken und einen rechten. Jeder Knotenpunkt, der nicht die Wurzel ist, hat genau einen Vater. Die untersten Knotenpunkte sind die Blätter, die anderen sind die inneren Knotenpunkte.

**Definition 2.2.** *Vollständiger binärer Baum:* ein binärer Baum; Blätter gibt es nur in den unteren höchstens zwei Etagen; jeder innere Knotenpunkt – mit höchstens einer Ausnahme – hat genau zwei Söhne.

Vollständige binäre Bäume werden so konstruiert, dass man nur dann eine neue Etage erstellt, falls die unterste Etage schon voll ist, und wenn es einen Knotenpunkt gibt, der nur einen Sohn hat, dann wird der neue Punkt sein zweiter Sohn sein. Meistens wird es auch angenommen, dass die Etagen von links nach rechts aufgefüllt werden.

Vollständige binäre Bäume können folgenderweise mit einem Array repräsentiert werden: der linke Sohn des Knotenpunktes  $A[i]$  ist  $A[2i]$ , sein rechter Sohn ist  $A[2i + 1]$ . Also, falls  $j > 1$ , dann ist der Vater des Knotenpunktes  $A[j]$  eben  $A[\lfloor j/2 \rfloor]$ .

Für jeden binären Baum mit  $n$  Elementen und  $l$  Etagen gilt:  $n \leq 2^l - 1$ . Für vollständige binäre Bäume gilt weiterhin:  $n \geq 2^{l-1}$  (sonst hätte der Baum weniger Etagen), also  $l \leq \log_2 n + 1$ .

**Definition 2.3.** *Binäres Heap:* vollständiger binärer Baum; in jedem Knotenpunkt ist ein  $s_i$  gespeichert;  $Sohn \geq Vater$ .

Implementierung von `addElement( $s$ )`: das neue Element wird in ein neues Blatt aufgenommen (mit Beibehaltung der Vollständigkeit), danach „sickert“ es nach oben (d.h. es wird immer wieder mit seinem jeweiligen Vater getauscht) solange  $s < Vater_s$  gilt. Kosten:  $\mathcal{O}(l) = \mathcal{O}(\log n)$ .

Implementierung von `removeMin`: das Minimum ist die Wurzel. Sie wird entfernt und ersetzt mit dem letzten Element des Arrays, das wiederum gelöscht wird. Danach muss das neue Wurzelement nach unten „sickern“, bis es seine richtige Position findet (der aktuelle Punkt ist  $b$ , seine Söhne sind  $b_1$  und  $b_2$ ):

```
proc NachUntenSickern( $b$ )
Wenn  $\min(b_1, b_2) < b$ , dann  $\text{swap}(b, \min(b_1, b_2))$ .
Wenn  $b$  mit  $b_i$  getauscht wurde, dann  $\text{NachUntenSickern}(b_i)$ 
```

Kosten:  $\mathcal{O}(l) = \mathcal{O}(\log n)$ .

Heapkonstruktion, 1. Möglichkeit: beginnend mit einem leeren Heap,  $n$ -mal `addElement`. Kosten:  $\mathcal{O}(n \log n)$ .

2. Möglichkeit: beginnend mit einem vollständigen binären Baum mit  $n$  Punkten, in dem die Elemente irgendwie platziert sind: für alle Elemente  $v$ , von unten nach oben, von rechts nach links (im Array: von rechts nach links): `NachUntenSickern(v)`. (Wichtig: als `NachUntenSickern(v)` aufgerufen wird, erfüllen seine beiden Teilbäume schon die Heap-Eigenschaft.)

Auf der  $i$ -ten Etage gibt es höchstens  $2^{i-1}$  Elemente, für Elemente dieser Etage sind die Kosten von `NachUntenSickern`: höchstens  $2(l-i)$  Vergleiche und  $l-i$  Bewegungen. Also insgesamt ist die Anzahl der Bewegungen ( $j = l-i, i = l-j$ ) höchstens:

$$\sum_{i=1}^l (l-i)2^{i-1} = \sum_{j=0}^{l-1} j2^{l-j-1} = 2^{l-1} \sum_{j=0}^{l-1} j/2^j \leq 2^l \leq 2n.$$

Zur vorletzten Ungleichung: jedes Glied der Summe entspricht einer Zeile der folgenden Tabelle (sie hat  $l-1$  Zeilen und  $l-1$  Spalten):

$$\begin{array}{cccc} 1/2 & & & \\ 1/4 & 1/4 & & \\ 1/8 & 1/8 & 1/8 & \\ \vdots & \vdots & \vdots & \ddots \\ 1/(2^{l-1}) & 1/(2^{l-1}) & 1/(2^{l-1}) & \dots & 1/(2^{l-1}) \end{array}$$

Die Summe der ersten Spalte ist weniger als 1, die der zweiten ist weniger als 1/2 usw. Die Summe dieser Teilsummen ist höchstens 2.

Also die Anzahl der nötigen Bewegungen ist höchstens  $2n$ , die der Vergleiche höchstens  $4n$ . Also kann das Heap in  $\mathcal{O}(n)$  Schritten aufgebaut werden.

## Heap Sort

Williams, Floyd (1964).

Heapkonstruktion, danach  $n$ -mal `removeMin`. Kosten:  $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$ . Das ist das theoretisch beste bekannte Sortierverfahren.

## d-Heap

Verallgemeinerung:  $d > 1$  ist eine ganze Zahl. Jeder Knotenpunkt hat höchstens  $d$  Söhne. Söhne des Knotenpunktes  $A[i]$ :  $A[d(i-1)+2], A[d(i-1)+3], \dots, A[di+1]$ , aber nur diejenigen, deren Index höchstens  $n$  ist. Falls  $j > 1$ , dann ist der Vater des Knotenpunktes  $A[j]$  eben  $A[\lfloor (j-1)/d \rfloor]$ . Alle obigen Gedanken können einfach verallgemeinert werden.

## Fibonacci-Heap

Eine kompliziertere Implementierung, sehr gute theoretische Laufzeiten in amortisierter Zeit (d.h. durchschnittlich in einer hinreichend langen Operationsfolge).

## Vergleich

Operation	binäres Heap	$d$ -Heap	Fibonacci-Heap (amortisierte Zeit)
Heapkonstruktion	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
addElement	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)$
removeMin	$\mathcal{O}(\log n)$	$\mathcal{O}(d \log_d n)$	$\mathcal{O}(\log n)$
decreaseKey	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)$

### 2.5.6 Quick sort

Hoare (1960). Teile und herrsche.

$s$  sei ein zufällig gewähltes Element aus dem Array  $A[1 : n]$ . Elemente, die kleiner sind als  $s$ , werden an den Anfang bewegt ( $A[1 : k]$ ), Elemente, die grösser oder gleich  $s$  sind, werden ans Ende bewegt ( $A[k+1 : n]$ );  $s$  wird  $A[k+1]$  sein. Nun kann Quick sort rekursiv für  $A[1 : k]$  und  $A[k+2 : n]$  aufgerufen werden.

Wie kann die Partition erstellt werden? Sei  $i = 1$ ,  $j = n$ .  $i$  wird erhöht, solange  $A[i] < s$  gilt. Dann wird  $j$  verringert, solange  $A[j] \geq s$  gilt. Falls  $i$  kleiner ist als  $j$ , dann werden  $A[i]$  und  $A[j]$  umgetauscht. Danach kann  $i$  wieder mit 1 erhöht werden, und  $j$  mit 1 verringert. Das macht man bis  $i = j$ . Am Ende bewegt man  $s$  auf Platz  $A[k+1]$ . Die Partition wurde in  $\mathcal{O}(n)$  Schritten (mit  $n - 1$  Vergleichen) erstellt.

Der Zeitbedarf von Quick sort hängt stark davon ab, wie gleichmässig die Partition ist. Sei  $C(n)$  die durchschnittliche Anzahl von Vergleichen.  $A[1 : n]$  enthalte jetzt einfachheitshalber die Zahlen  $1, \dots, n$ ; der Durchschnitt ist bezüglich die  $n!$  Permutationen zu verstehen. Sei  $C(n, j)$  die durchschnittliche Anzahl der Vergleiche in dem Fall, wenn das erste gewählte Element  $j$  ist. Es gilt

$$C(n, j) = n - 1 + C(j - 1) + C(n - j)$$

Ausserdem hat jedes  $j$  die gleiche Wahrscheinlichkeit, also:

$$C(n) = \frac{1}{n}(C(n, 1) + \dots + C(n, n))$$

Aus diesen beiden folgt:

$$C(n) = n - 1 + \frac{2}{n}(C(1) + \dots + C(n - 1))$$

oder

$$nC(n) = n(n - 1) + 2(C(1) + \dots + C(n - 1))$$

Wenn man das auch für  $n - 1$  aufschreibt, und die beiden Formeln auseinander subtrahiert, erhält man:

$$nC(n) - (n - 1)C(n - 1) = 2(n - 1) + 2C(n - 1)$$

Umgeformt und dividiert mit  $n(n + 1)$ :

$$\frac{C(n)}{n + 1} = \frac{2(n - 1)}{n(n + 1)} + \frac{C(n - 1)}{n} < \frac{2}{n} + \frac{C(n - 1)}{n}$$

Daraus folgt:

$$\frac{C(n)}{n + 1} < \frac{2}{n} + \frac{2}{n - 1} + \dots + \frac{2}{2} + \frac{2}{1} = 2H_n$$

wobei  $H_n$  die berühmte harmonische Zahlenfolge ist. Es ist bekannt, dass  $H_n = \ln n + \gamma + o(1)$ , wobei  $\gamma = 0,55721\dots$  die Eulersche Konstante ist. Daraus folgt:

$$C(n) < 1,39n \log_2 n + \mathcal{O}(n)$$

Quick sort kann zwar im schlimmsten Fall  $\mathcal{O}(n^2)$  Schritte brauchen, aber bezüglich durchschnittlichen Zeitbedarf ist es eines der besten bekannten Sortierverfahren.

## 2.6 Sortierung mit Schlüsselmanipulation

Bis jetzt wurde nichts über  $(U, <)$  angenommen. Falls wir mehr wissen, z. B.:

- Die Anzahl der Elemente in  $U$  ist bekannt  $\rightarrow$  Kistensortierung (binsort, ládarendezés)
- Die Struktur der Schlüssel in  $U$  ist bekannt  $\rightarrow$  Radix-Sortierung

Für solche Fälle gilt die obige untere Schranke von  $\Omega(n \log n)$  nicht.

### 2.6.1 Kistensortierung

Wir nehmen an, dass  $|U| = m$ . Die Aufgabe ist  $A[1 : n]$  zu ordnen. Vorteilhaft, falls  $m$  nicht viel grösser als  $n$  ist. Schritte:

1. Ein Feld  $B$  der Grösse  $m$  reservieren: die Elemente von  $B$  sind Listen, die am Anfang leer sind ( $\mathcal{O}(m)$ )
2.  $A$  durchlesen: falls der Schlüssel von  $A[i]$  eben  $s$  ist, dann soll  $A[i]$  zu der Liste in  $B[s]$  hingefügt werden ( $\mathcal{O}(n)$ )
3. Die Listen in  $B$  sollen die Reihe nach in  $A$  zurückgeschrieben werden ( $\mathcal{O}(n + m)$ )

(Wenn  $A$  nicht Rekords beinhaltet, sondern elementare Daten, dann noch einfacher.) Insgesamt  $\mathcal{O}(n + m)$  Schritte. Falls  $m \leq cn$ , dann  $\mathcal{O}(n)$ .

Beispiel:....

### 2.6.2 Radix-Sortierung

Annahme: Elemente von  $U$  haben die Form:

$$t_1 \dots t_k \quad \text{wo } t_i \in L_i, |L_i| = s_i$$

$L_i$  ist ein geordneter Typ.

Aufgabe: lexikographische Ordnung:

$$t_1 \dots t_k < r_1 \dots r_k \quad \text{falls } \exists i \forall j < i (t_j = r_j \wedge t_i < r_i)$$

Beispiel:

$$L_1 := \{1900, 1901, \dots, 1999\} \quad s_1 = 100$$

$$L_2 := \{\text{Januar}, \dots, \text{Dezember}\} \quad s_2 = 12$$

$$L_3 := \{1, 2, \dots, 31\} \quad s_3 = 31$$

Algorithmus: Sortiere zuerst die Elemente mit Kistensortierung laut ihrer  $k$ -ten (letzten) Komponente, dann laut der  $k-1$ -ten, usw. Man soll aufpassen, dass die Elemente bei der Kistensortierung immer an das Ende der Liste kommen.

Die Korrektheit der Methode für  $k=2$ : Sei  $X = x_1x_2$  und  $Y = y_1y_2$  und nehmen wir an, dass  $X < Y$ .

- Falls  $x_1 < y_1$ , dann wird  $X$  in der letzten Kistensortierung vor  $Y$  gesetzt.
- Falls  $x_1 = y_1$  und  $x_2 < y_2$ , dann überholt  $X$   $Y$  in der ersten Sortierung, und in der zweiten wird das nicht geändert (denn  $Y$  wird beim binsort an das Ende der Liste gesetzt.)

Beispiel: ....

Kost:  $k$  Mal Kost(binsort), also  $O(kn + \sum_{i=1}^k s_i)$

Falls  $k = \mathcal{O}(1)$  und  $s_i = \mathcal{O}(n)$ , dann  $\mathcal{O}(n)$ . Das kommt vor, falls Z.B. Zahlen aus  $[1, n^k - 1]$  sortiert werden sollen. Sie sind als  $k$ -stellige Zahlen interpretiert und mit Radix-Sortierung sortiert.

## 2.7 Sortierung des Inhaltes von Hintergrundspeichern

Nicht detailliert behandelt. Wichtigster Kostfaktor: Zugriff auf die Daten. Varianten von Merge Sort können eingesetzt werden.

## Kapitel 3

# Suchbaum (Searchtree, keresőfa)

Ziel: Daten von einem geordneten Typ  $(U, <)$  speichern, so dass *EINFÜGEN*, *SUCHEN*, *LÖSCHEN*, *MIN*, *MAX*, *VONBIS* effizient verwirklicht werden können. Die besten Implementationen benutzen Bäume.

### 3.1 Binärer Baum

Knotenpunkt: ein Rekord, zusammengesetzter Typ.

$Element(x)$ ,  $links(x)$ ,  $rechts(x)$ ,  $Vater(x)$

Beispiel:  $(8*5)+(9-6)$

preorder, inorder, postorder Durchfahren der Knotenpunkte:

```
pre(x)
begin
besuchen(x)
pre(links(x))
pre(rechts(x))
end
```

inorder, postorder: ähnlich, aber besuchen passiert zwischen bzw. nach den zwei rekursiven Aufrufen.

Ergebnis:

Pre: +\*85-96

In: 8\*5+9-6

Post: 85\*96-+

Kost: Sei  $T(n)$  die Kost.  $T(0) = 0$ , also:

$$T(n) \leq c + \max_{0 \leq i \leq n-1} \{T(i) + T(n-1-i)\}$$

wobei  $c$  die konstante Zeit für das Besuchen und für die Funktionsaufrufe ist. Mit Induktion folgt, dass  $T(n) \leq cn$ , also  $\mathcal{O}(n)$ .

### 3.2 Binärer Suchbaum

**Definition 3.1.** *Suchbaum-Eigenschaft:*

$\forall x \forall y \forall z (y \in links(x) \wedge z \in rechts(x) \longrightarrow Element(y) \leq Element(x) \wedge Element(x) \leq Element(z))$

**SUCHEN, MIN, MAX** trivial  $\mathcal{O}(l)$ .

**VONBIS(a,b)** ist  $\mathcal{O}(l+k)$ , wo  $k$  die Anzahl der Elemente zwischen  $a$  und  $b$  ist, falls die Inorder-Nachfolger der Knotenpunkte vorhanden sind, und  $\mathcal{O}(n)$ , falls nicht. (Der Inorder-Durchlauf dauert so lange).

**EINFÜGEN(s,S)**: zuerst **SUCHEN(s,S)**; falls  $s \in S$  dann soll man nichts machen, sonst soll  $s$  am Ende des Suchweges als ein Blatt eingefügt werden.

**LÖSCHEN(s, S)**: nichts, falls  $s \notin S$ . Falls  $s$  höchstens einen Sohn hat, dann wird  $s$  mit  $\text{Sohn}(s)$  überschrieben. Falls  $s$  zwei Söhne hat, dann  $y := \text{MAX}(\text{links}(s))$  und  $s$  wird mit  $y$  überschrieben.

**Kost der Suchbaumkonstruktion mit naivem Einfügen**:  $U$  bestehe aus  $b_1 < \dots < b_n$ . Die  $b_i$  Elemente kommen in einer zufälligen Reihenfolge  $a_1, \dots, a_n$ . Durchschnittskost:  $T(n)$ . Durchschnittskost bei Bäumen, wo  $a_1 = b_j: T(n, j)$ .

$$T(n) = \frac{1}{n} \sum_{j=1}^n T(n, j)$$

Wenn  $a_1 = b_j$ , dann werden im linken Teilbaum  $j - 1$  Elemente sein, im rechten  $n - j$ . Man braucht  $n - 1$  Vergleiche (alle anderen Elemente werden mit  $a_1$  verglichen), um diese Partition zu finden. Ausserdem braucht man  $T(j-1)$  bzw.  $T(n-j)$  Vergleiche, um die zwei Teilbäume zu konstruieren. Es folgt:  $T(n, j) = n - 1 + T(j - 1) + T(n - j)$ . Also

$$T(n) = n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} T(j)$$

Wie beim Quicksort, also

$$T(n) \approx 1,39n \log_2 n + \mathcal{O}(n)$$

Also die durchschnittliche Kost für Einfügen ist etwa  $1,39 \log n + \mathcal{O}(1)$ . Die durchschnittliche Kost für Suchen ist auch ähnlich. Auch die Durchschnittshöhe ist gut,  $\mathcal{O}(\log n)$ . (Es könnte auch nicht wesentlich besser sein.)

### 3.3 2-3 Bäume

Eigenschaften:

1. Die Rekords sind in den Blättern in der Grössenordnung von links nach rechts.
2. Alle innere Knotenpunkte haben entweder 2 oder 3 Teilbäume:

$$\boxed{p_1 \mid s_1 \mid p_2} \quad \text{oder} \quad \boxed{p_1 \mid s_1 \mid p_2 \mid s_2 \mid p_3}$$

( $p_i$  sind Zeiger,  $s_i$  sind Schlüssel.)  $s_1 < s_2$ , und jeder Schlüssel im Teilbaum  $p_1$  ist kleiner als  $s_1$ ; im Teilbaum  $p_2$  ist  $s_1$  der kleinste Schlüssel und alle sind kleiner als  $s_2$  usw.

3. Alle Wege von der Wurzel bis zu den Blättern sind gleich lang.

Es ist einfach zu sehen, dass  $l \leq \log_2 n + 1$ .

**SUCHEN** ist nicht nur durchschnittlich, sondern immer schnell.

**EINFÜGEN**: falls der letzte innere Knotenpunkt im Suchweg nur zwei Söhne hat, dann einfach. Sonst: *Knotenpunktschnitt*: Aus einem Knotenpunkt macht man zwei, und das verursacht ein Einfügen in Vater(x). Das kann höchstens bis zur Wurzel weitergeleitet werden, dann soll eine neue Wurzel gemacht werden.

**LÖSCHEN**: Sei  $x$  der letzte Knotenpunkt im Suchweg. Wenn  $x$  drei Söhne hat, dann einfach. Sonst, wenn ein Bruder von  $x$  drei Söhne hat, dann bekommt  $x$  einen neuen Sohn von ihm. Sonst *Knotenpunktleben*: Umgekehrt wie Knotenpunktschnitt.

Also geht Einfügen und Löschen in  $\mathcal{O}(\log n)$  Zeit.

### 3.4 B-Bäume

Beste Suchbaumimplementierung, falls die Daten auf Festplatte sind. Standard. 1972, Bayer, McCreight.  
 Kost=Seitenzugriffe. Knotenpunkte=Seiten.

Für  $m \geq 3$  ist der  $B_m$ -Baum wie folgt definiert:

1. Grad der Wurzel  $\geq 2$ .
2. Anzahl der Söhne der inneren Knotenpunkte  $\geq \lceil \frac{m}{2} \rceil$ .
3. Länge aller Wege von der Wurzel bis zu den Blättern ist gleich.
4. Jeder Knotenpunkt hat höchstens  $m$  Söhne.

Die Rekords sind in den Blättern.  $B_3$  ist ein 2-3-Baum. Im allgemeinen ( $\lceil \frac{m}{2} \rceil - 1 \leq i \leq m - 1$ ):

$$\boxed{p_0} \mid \boxed{s_1} \mid \boxed{p_1} \mid \cdots \mid \boxed{s_i} \mid \boxed{p_i}$$

Wenn  $n$  die Anzahl der Blätter ist, dann ist die Anzahl der Etagen höchstens  $2 + \frac{\log_2 n - 1}{\log_2 \lceil \frac{m}{2} \rceil}$ . (Also, wenn  $m$  höher ist, dann wird Suchen schneller.)

### 3.5 AVL-Bäume

Binärer Baum. Wichtig ist die Anzahl der Etagen in einem Baum. Falls  $l \leq c \log_2 n$ , wo  $c \gg 1$  dann heisst er ein balancierter Baum.

AVL: Adelson-Velskij, Landis, 1962.

$l(B)$ : Anzahl der Etagen in  $B$ .

**Definition 3.2.**

$$AVL\text{-Baum} \Leftrightarrow \forall x \ |l(links(x)) - l(rechts(x))| \leq 1$$

$G_k$ : minimale Anzahl der Knotenpunkte, falls die Anzahl der Etagen  $k$  ist.

$G_1 = 1, G_2 = 2, G_3 = 4, G_4 = 7$ . Allgemein:

$$G_k = 1 + G_{k-1} + G_{k-2}$$

So folgt mit Induktion, dass  $G_k = F_k - 1$ , wo  $F_k$  die  $k$ -te Fibonacci-Zahl ist. Da  $F_k \approx \left(\frac{1+\sqrt{5}}{2}\right)^k$ , so folgt, dass  $k = \mathcal{O}(\log n)$ . Also sind AVL-Bäume balanciert ( $c \approx 1.44$ ). Um die AVL-Eigenschaft zu bewahren:

*Drehung:*

$$\text{Sei } y := linkssohn(x), f := linksbaum(y), g := rechtsbaum(y), h := rechtsbaum(x)$$

Dann kann der Baum wie folgt (nach rechts) gedreht werden (siehe Abbildung 3.1):

$$f := linksbaum(y), x := rechtssohn(y), g := linksbaum(x), h := rechtsbaum(x)$$

Sein Inverz ist die links-Drehung.

*Doppeldrehung:* siehe Abbildung 3.2

**Satz 3.1.** Nach EINFÜGEN( $s,S$ ) braucht man höchstens eine Drehung oder Doppeldrehung um die AVL-Eigenschaft wiederherzustellen.

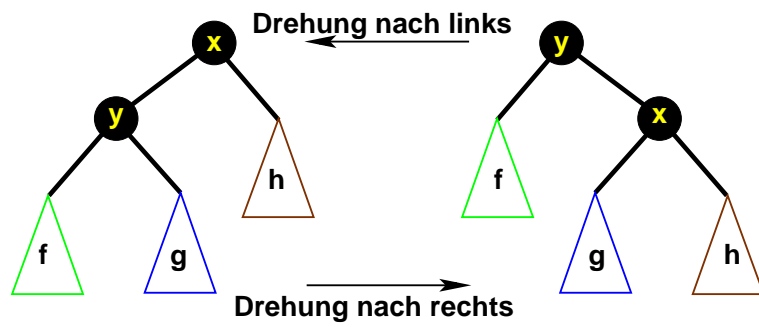


Abbildung 3.1: Drehungen in AVL-Bäumen

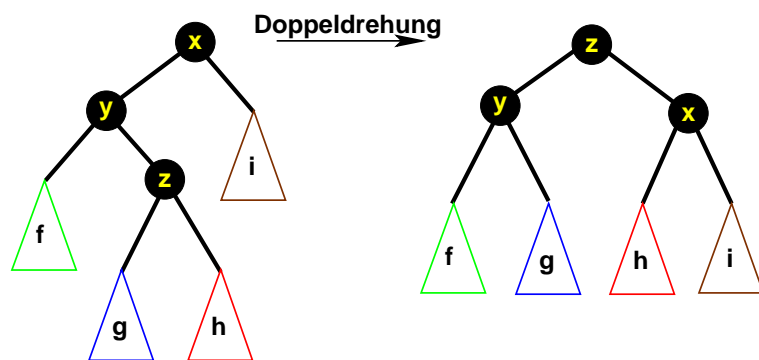


Abbildung 3.2: Doppeldrehung in AVL-Bäumen

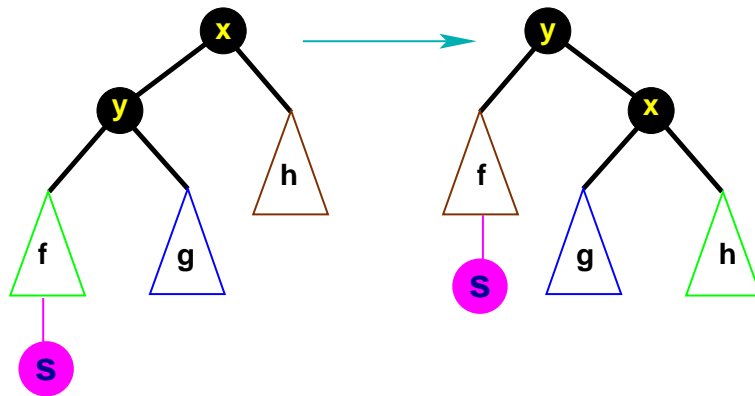


Abbildung 3.3: Fall 1

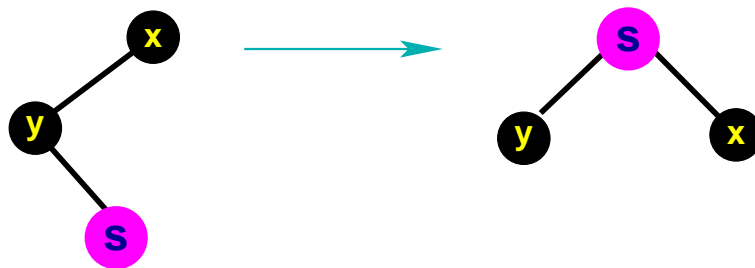


Abbildung 3.4: Fall 2a

*Beweis.* Sei  $x$  der unterste Knotenpunkt im Suchweg, wo nach dem Einfügen die AVL-Eigenschaft verletzt wird. Sei  $l(links(x)) = l$  (ohne  $s$ ) und  $l(rechts(x)) = l - 1$ . Seien die Buchstaben, wie in der Beschreibung der Drehung. Man muss folgende Fälle unterscheiden:

1.  $s$  kommt in  $f$  (siehe Abbildung 3.3)
2.  $s$  kommt in  $g$ 
  - (a)  $y$  ist ein Blatt (siehe Abbildung 3.4)
  - (b)  $y$  ist kein Blatt (siehe Abbildung 3.5)

In dem ersten Fall reicht eine rechts-Drehung, in dem zweiten eine Doppeldrehung. (Um die Gültigkeit der AVL-Eigenschaft zu kontrollieren, soll immer die Definition von  $x$  ausgenutzt werden.)  $\square$

Beim LÖSCHEN ebenso; wird nicht detailliert.

## 3.6 Andere balancierte Bäume

### 3.6.1 HB[k]-Bäume

Sei  $k \geq 1$ . HB[k]-Eigenschaft:

$$\forall x |l(links(x)) - l(rechts(x))| \leq k$$

Speziell für  $k = 1$ : AVL.

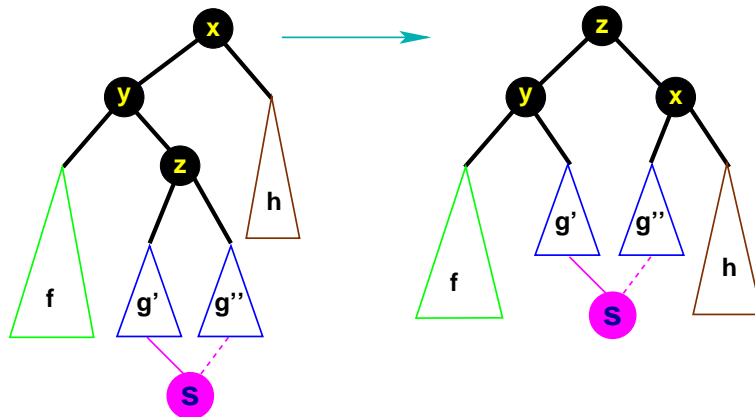


Abbildung 3.5: Fall 2b

### 3.6.2 Gewichtet balancierte Bäume

**Definition 3.3.** Sei das Gewicht eines Knotenpunkts  $w(x)$  die Anzahl von Knotenpunkten im Unterbaum von  $x$  (auch  $x$  zählt).

**Definition 3.4.** Ein Baum ist gewichtet balanciert, falls

$$\sqrt{2} - 1 < \frac{w(\text{links}(x))}{w(\text{rechts}(x))} < \sqrt{2} + 1$$

*Bemerkung:* Die Definition ist symmetrisch, da  $(\sqrt{2} - 1)^{-1} = \sqrt{2} + 1$ .

Wir zeigen, dass dieser Baum wirklich balanciert ist (mit  $c = 2$ ). Es gilt, dass:

$$w(x) > w(y) + w(z) > (\sqrt{2} - 1)w(z) + w(z) = \sqrt{2}w(z)$$

wo  $y$  und  $z$  die Söhne von  $x$  sind. Ebenso

$$w(x) > \sqrt{2}w(y)$$

Daraus folgt, dass

$$n = w(\text{Wurzel}) > (\sqrt{2})^{k-1}w(\text{Blatt}) = (\sqrt{2})^{k-1}$$

wo  $k$  die Anzahl der Etagen ist. So folgt, dass

$$k < 2 \log_2 n + 1$$

## 3.7 Eine weitere Anwendung

Listen konkatenieren mit Hilfe von AVL-Bäumen:

Seien  $S_1$  und  $S_2$  zwei AVL-Bäume, so dass jeder Schlüssel in  $S_2$  grösser ist als die Schlüssel in  $S_1$ . Eine geordnete Liste kann in einem AVL-Baum gespeichert werden, wobei das inorder Durchlaufen die Reihenfolge definiert. Konkatenieren geht wie folgt:

1. sei  $s := \text{MIN}(S_2)$ . Lösche  $s$  von  $S_2$ . So bekommt man  $S_3$ .
2. Sei  $x$  der erste Knotenpunkt in  $S_1$  angefangen von der Wurzel und immer nach rechts gehend, so dass  $l(x) - l(S_3) = 0$  oder 1.
3. Statt  $x$  kommt  $s$ , sein Rechtsbaum wird  $S_3$  sein, sein Linksbaum wird  $x$  und sein Unterbaum.

Also geht das in  $\mathcal{O}(\log n)$  Zeit, wenn die Gewichte bekannt sind.

### 3.8 S-Bäume

S-Baum=splay-tree

Binärer Suchbaum. Nicht unbedingt balanciert, es kann vorkommen, dass eine Suche ganz lange dauert. Aber er lernt während seines Lebens, und modifiziert die Form des Baumes dem Bedarf entsprechend. Eigenschaften:

- Falls eine Operationsfolge lang genug ist, dann ist die Durchschnittskost für eine Operation quasi-optimal.
- Falls die Ansprüche auf einzelne Elemente in Burst kommen, dann kann noch effektiver sein, als die balancierten Bäume. (Beispiel: Krankenhaus)

**Idee:** Falls ein Anspruch auf  $s \in U$  kommt, dann wird die Wichtigkeit von  $s$  erhöht.  $s$  wird mit Drehungen in die Wurzel gesetzt, und alle Elemente im Suchweg von  $s$  werden ein bisschen näher zur Wurzel gesetzt.

**Definition 3.5.**  $KLEBEN(f, f') :=$  macht aus den Bäumen  $f$  und  $f'$  einen Baum, falls

$$\forall x \forall y (x \in f \wedge y \in f' \rightarrow x < y)$$

gilt.

$SCHNEIDEN(x, f) :=$  aus  $f$  werden zwei Bäume  $f'$  und  $f''$  hergestellt, so dass

$$\forall y \forall z (y \in f' \wedge z \in f'' \rightarrow y \leq x \leq z)$$

$$UMKEHREN(x, f) := \begin{cases} x \text{ wird Wurzel von } f & \text{falls } x \in f \\ \min\{y : y > x\} \text{ oder } \max\{y : y < x\} \text{ wird Wurzel von } f & \text{falls } x \notin f \end{cases}$$

**Implementation von UMKEHREN(x, f):** Suchen(x, f). Als Ergebnis finden wir entweder  $x$  oder wir bleiben bei  $\min\{y : y > x\}$  oder  $\max\{y : y < x\}$  stehen. Nehmen wir an, dass wir  $x$  gefunden haben. Sei Drehung(x) die Drehung, die  $x$  mit seinem Vater verwechselt. Der folgende Schritt soll solange wiederholt werden, bis  $x$  in der Wurzel landet: (Sei  $y := \text{Vater}(x)$ )

1. Falls  $x$  die Wurzel ist  $\rightarrow$  STOP
2. Falls  $\nexists \text{Vater}(\text{Vater}(x)) \rightarrow$  Drehung(x).
3. Sonst: sind  $x$  und  $y$  beide Rechtssöhne (bzw. Linkssöhne)  $\rightarrow$  Drehung(y), Drehung(x)
4. Sonst: Drehung(x), Drehung(x)

Trivialerweise dauert UMKEHREN(x, f)  $\mathcal{O}(l)$  lang.

Bemerkung: alle Knotenpunkte auf dem Suchweg von  $x$  landen auch näher zur Wurzel durch UMKEHREN(x, f).

**Satz 3.2.** Alle Operationen können mit einer konstanten Anzahl von UMKEHREN(x, f) +  $\mathcal{O}(1)$  realisiert werden.

*Beweis.* Nur KLEBEN und LÖSCHEN.

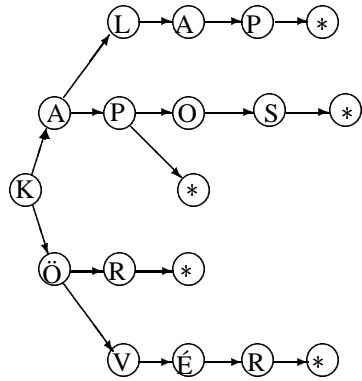
- $KLEBEN(f, f')$ :  $UMKEHREN(+\infty, f)$ . Danach wird das grösste Element von  $f$  in der Wurzel stehen, also hat die Wurzel keinen Rechtssohn.  $f'$  wird sein neuer Rechtssohn.
- $LÖSCHEN(x, f)$ :  $UMKEHREN(x, f)$ . Falls  $x$  nicht die Wurzel wird, dann  $x \notin f$ . Sonst ist  $x$  in der Wurzel. Seien die zwei Unterbäume von  $x$ :  $g$  und  $h$ . Dann  $KLEBEN(g, h)$ .

□

### 3.9 Wortbäume (tries)

Falls  $U$  spezielle Struktur hat, dann kann effektiver werden.  
Beispiel:  $\Sigma$ : Alphabet,  $\Sigma^*$ : Wörter. Ordnung: lexikographisch.

Implementierung mit Listen oder mit Arrays.



Es kann bewiesen werden, dass die Suche durchschnittlich  $c + \log_m n$  lang dauert, wo  $n$  die Anzahl der Wörter im Baum, und  $m = |\Sigma|$  ist.

## Kapitel 4

# Gestreute Speicherung (Hashing)

Ziel: wie bei den Suchbäumen, nur  $U$  muss nicht unbedingt geordnet sein. Die Methoden sind durchschnittlich schnell, im schlechtesten Fall können sie aber langsam sein.

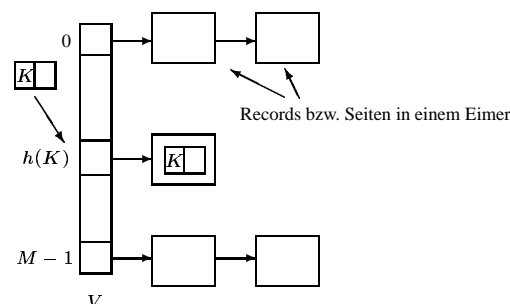
Beispiel: Personalnummer:  $2 \cdot 10^2 \cdot 12 \cdot 31 \cdot 10^3 = 74$  Million Möglichkeiten. Aber nur 10 Millionen leben in Ungarn. Also es reicht nur z. B. 12 Million Plätze zu reservieren, und eine passende Funktion  $h(K)$  zu finden, die zu einem Schlüssel  $K$  eine Adresse  $h(K)$  ordnet.

Probleme:

- Kollisionen müssen behoben werden
- Eine geeignete Hashfunktion muss gefunden werden

### 4.1 Speicherung in geketteten Listen

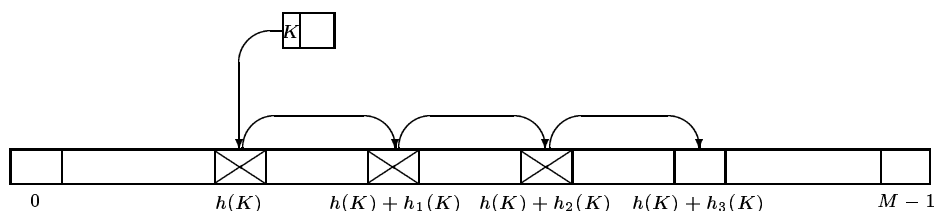
Hauptsächlich für Daten auf Hintergrundspeicher benutzt. Es gibt ein Eimerkatalog  $V[0 : M - 1]$ . Das Element mit Schlüssel  $K$  kommt ins Eimer  $V[h(K)]$ . Ein Eimer (bucket, Korb) ist eine gekettete Liste.



Die durchschnittliche Dauer der Suche ist  $\frac{N}{M} + 1$ , wo  $N$  die Anzahl der Rekords,  $M$  die Anzahl der Eimer ist. (Nur die Seitenzugriffe wurden gezählt.) Es ist wichtig, das Parameter  $M$  angemessen zu wählen. Typisch ist, dass  $\frac{N}{M} \approx 1$ , sogar etwas weniger ist.

## 4.2 Offene Adressierung

Daten sind im Arbeitsspeicher. Die Rekords werden im Feld  $A[0 : M - 1]$  gespeichert, wo  $[0 : M - 1]$  das Bild von  $h(\cdot)$  ist. Ein Rekord mit Schlüssel  $K$  kommt auf Platz  $h(K)$ , falls es frei ist. Sonst versucht man es in den Zellen  $h(K) + h_1(K), h(K) + h_2(K), \dots, h(K) + h_{M-1}(K) \pmod{M}$  zu unterbringen, wobei  $0, h_1(K), h_2(K), \dots, h_{M-1}(K)$  eine Permutation von  $[0 : M - 1]$  ist. Suchen geht genauso.



Achtung: Beim Löschen darf das Element nicht einfach entfernt werden, sonst können einige Elemente nicht mehr erreicht werden. Stattdessen wird ein spezielles Symbol (z.B. \*) angewendet. \* bedeutet für die Suche: weitersuchen, denn diese Zelle war besetzt. \* bedeutet für das Einfügen: die Zelle ist leer, also Einfügen ist erlaubt. (Implementierungsmöglichkeit: zu jeder Zelle gehört ein Zustandscode mit den folgenden möglichen Werten: FREI, BESETZT, GELÖSCHT.)

Bezeichnungen:

- $N$  Anzahl der Rekords in  $A$
- $M$  Anzahl der Zellen in  $A$
- $\alpha = \frac{N}{M}$  Belegungsfaktor
- $C_{N,M}$  Durchschnittliche Kosten einer erfolgreichen Suche
- $C'_{N,M}$  Durchschnittliche Kosten einer erfolglosen Suche

### 4.2.1 Lineares Sondieren

$h_i(K) := -i$  (unabhängig von  $K$ ). ( $h_i(K) = i$  wäre auch möglich; das ist nur etwas effizienter, weil man typischerweise schneller mit 0 vergleichen kann, als mit  $M$ .)

Kosten:

$$C_{N,M} = \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right) \quad \text{und} \quad C'_{N,M} = \frac{1}{2} \left( 1 + \left( \frac{1}{1-\alpha} \right)^2 \right)$$

$\alpha$	$C_{N,M}$	$C'_{N,M}$
2/3	2	5
0,8	3	13
0,9	5,5	50,5

Nachteil: Viele Elemente nebeneinander; die Wahrscheinlichkeit, dass ein neues Element auch in diese Strecke fällt, ist hoch, und wird immer höher (*primäre Knotenbildung*).

## 4.2.2 Quasi-zufälliges Sondieren

Solche Permutation  $h_i$  wird gesucht, die einfach zu kalkulieren ist, die aber die Elemente gleichmässig im Intervall  $[0 : M - 1]$  verstreut. So kommt keine primäre Knotenbildung vor, aber sekundäre Knotenbildung kann vorkommen: bei Rekords mit gleicher Hashfunktionswert. (Grund:  $h_i$  hängt nicht von  $K$  ab.)

Beispiel: quadratisches Sondieren: sei  $M$  eine Primzahl der Form  $4k + 3$ . Sei die Versuchsreihe:

$$0, 1^2, -(1^2), 2^2, -(2^2), \dots, \left(\frac{M-1}{2}\right)^2, -\left(\frac{M-1}{2}\right)^2.$$

**Satz 4.1.** Diese Zahlen geben alle Restklassen  $(\text{mod } M)$ .

*Beweis.* Das sind  $1 + 2 \cdot \frac{M-1}{2} = M$  Zahlen, man muss also nur beweisen, dass sie alle verschieden sind  $\text{mod } M$ .

Falls  $0 \leq i < j \leq \frac{M-1}{2}$ , dann  $i^2 \not\equiv j^2 \pmod{M}$ , denn im Produkt  $j^2 - i^2 = (j-i)(j+i)$  kann keins der Glieder mit  $M$  teilbar sein, also das Produkt auch nicht (weil  $M$  eine Primzahl ist).

Ebenso:  $-i^2 \not\equiv -j^2 \pmod{M}$ .

$i^2 \not\equiv -j^2 \pmod{M}$  gdw.  $(ij^{-1})^2 \not\equiv -1 \pmod{M}$ . Aber  $n^2 \equiv -1 \pmod{M}$  für ein beliebiges  $n$  würde bedeuten, dass

$$-1 \equiv (-1)^{\frac{M-1}{2}} \equiv n^{2 \cdot \frac{M-1}{2}} \equiv n^{M-1} \equiv 1 \pmod{M}.$$

(Die erste Äquivalenz folgt daraus, dass  $\frac{M-1}{2}$  ungerade ist; die letzte folgt aus dem (kleinen) Fermatschen Satz.) Das ist ein Widerspruch.

Ebenso:  $i^2 \not\equiv -i^2 \pmod{M}$ .

□

Kosten:

$$C_{N,M} \approx 1 + \log \frac{1}{1-\alpha} - \frac{\alpha}{2} \quad \text{und} \quad C'_{N,M} \approx \frac{1}{1-\alpha} - \alpha + \log \frac{1}{1-\alpha}$$

Dieser Zusammenhang ist gültig allgemein wenn  $h_i(K) = f_i(h(K))$ .

## 4.2.3 Doppelhashing

G. de Balbine, J. R. Bell, C. H. Kaman, 1970.

$h_i(K) := -ih'(K)$ , wobei  $h'(K)$  eine zweite Hashfunktion ist, wofür  $(h'(K), M) = 1$  gilt. Daraus folgt, dass

$$-ih'(K) \not\equiv -jh'(K) \pmod{M}$$

Wichtig: in diesem Fall hängt die Versuchsreihe von  $K$  ab, also werden beide Knotenbildungsarten vermieden.

Kosten:

$$C_{N,M} \approx \frac{1}{\alpha} \log \frac{1}{1-\alpha} \quad \text{und} \quad C'_{N,M} \approx \frac{1}{1-\alpha}.$$

Falls die Suche nicht erfolgreich ist, dann immer, sonst für  $\alpha \geq 0.8$  ist diese Methode schneller als lineares Sondieren.

## 4.3 Hashfunktionen

Anforderungen:

- schnell berechenbar
- verursacht wenig Kollisionen

### 4.3.1 Modulo-Methode

$h(K) := K \bmod M \in [0, M - 1]$ , wobei  $M$  die Grösse der Hashtabelle ist. Am besten ist  $M$  eine Primzahl.

### 4.3.2 Multiplikationsmethode

$h(K) := \lfloor M \cdot \{\beta K\} \rfloor$ , wobei  $\beta$  eine reelle Zahl ist. ( $\{\beta K\} \in [0, 1) \Rightarrow h(K) \in [0, M - 1]$ .) Wichtiges Beispiel:  $\beta = C/2^w$ , wobei  $w$  die Wortlänge des Rechners und  $C$  eine ungerade Zahl ist. In diesem Fall kann  $h(K)$  sehr effizient berechnet werden.

Die gleichmässigste Verteilung erhält man bei  $\beta = \frac{\sqrt{5}-1}{2}$ .

### 4.3.3 Die zweite Hashfunktion beim Doppelhashing

Anforderung:  $(M, h'(K)) = 1$ . Wenn  $M$  eine Primzahl ist, dann ist

$$h'(K) := (K \bmod (M - 1)) + 1$$

eine gute Wahl. ( $K \bmod (M - 1) \in [0, M - 2] \Rightarrow h'(K) \in [1, M - 1]$ .)

## 4.4 Suchbäume oder Hashing?

Bei Hashing sind die Grundoperationen im Durchschnitt schneller, aber bei Suchbäumen ist die obere Schranke für die Anzahl der Schritte kleiner.

Suchbäume sind besser geeignet für geordnete Datentypen. Andererseits, Hashing benötigt gar nicht, dass die Daten geordnet seien.

Suchbäume sind skalierbar. Bei Hashing ist die Vergrößerung der Tabelle ziemlich aufwendig.

# Kapitel 5

## Datenkomprimierung

### 5.1 Huffman-Codes

$\Sigma = \{b_1, \dots, b_n\}$ . Man möchte einen (langen) Text aus  $\Sigma^*$  als Bitfolge kodieren. Die Codes der Buchstaben sollen so gewählt werden, dass die Länge der Bitfolge minimal ist.

Wenn alle Codes die gleiche Länge haben, dann ist diese Länge mindestens  $\lceil \log_2 n \rceil$ . Idee: Buchstaben, die öfter vorkommen, sollen kürzere Codes haben, als Buchstaben, die selten vorkommen. (Annahme: die Häufigkeit der Buchstaben ist bekannt; die Häufigkeit von  $b_i$  sei  $q_i$ .)

Problem: wie erkennt man die Grenze zwischen nacheinanderkommenden Buchstaben in der Bitfolge?

**Definition 5.1.** *Präfix-Code:*  $(\{b_i, b_j \in \Sigma, w \in \{0, 1\}^*\} : (code(b_i) = code(b_j)w)$

Ein Präfix-Code kann eindeutig dekodiert werden: man liest solange, bis ein gültiger Code eingelesen wurde  $\rightarrow$  das ist die erste Buchstabe usw.

Zu einem Präfix-Code kann ein binärer Baum zugeordnet werden. Ein Weg von der Wurzel bis zu einem Blatt entspricht dem Code einer Buchstabe. Jene Kanten, die nach links gehen, entsprechen einer 0, die anderen einer 1. Die Präfixeigenschaft bedeutet, dass die Buchstaben mit den Blättern identifiziert werden können.

Die Aufgabe: unter den binären Bäumen mit den Blättern  $b_1, \dots, b_n$  soll  $I = \sum q_i h(b_i)$  minimiert werden, wobei  $h(x)$  die Länge des *Wurzel*  $\rightarrow x$  Weges bedeutet.

Wie sieht ein optimaler Baum  $B$  aus? Die Buchstaben mit geringer Häufigkeit müssen unten sein. Genauer gesagt: sei  $x$  ein Blatt mit maximalem  $h$ -Wert; sei  $y$  sein Vater. Falls  $y$  keinen anderen Sohn hätte, dann könnte man  $I$  reduzieren, indem man  $y$  mit  $x$  ersetzt. Sei also z.B.  $z$  der andere Sohn von  $y$ . Man kann annehmen, dass  $x$  und  $z$  die kleinsten Häufigkeiten (z.B.  $q_1$  und  $q_2$ ) haben. Sonst könnte man die Codes vertauschen, was  $I$  nicht erhöhen würde. (Es kann natürlich auch mehrere optimale Bäume geben.)

Wenn man  $x$  und  $z$  löscht, und  $y$  mit der Häufigkeit  $q_1 + q_2$  versieht, dann erhält man den Baum  $B'$ . Es gilt:  $I(B) = I(B') + q_1 + q_2$ .  $B'$  ist auch optimal, denn sonst könnte man  $B$  verbessern.

Also die Konstruktion (Huffman-Baum): Zuerst hat man  $n$  isolierte Punkte mit den Häufigkeitswerten. Die zwei Knotenpunkte mit der kleinsten Häufigkeit (z.B.  $q_1$  und  $q_2$ ) werden mit einem gemeinsamen Vater versehen, der die Zahl  $q_1 + q_2$  erhält. Im  $k$ -ten Schritt hat man  $n - k$  Bäume, deren Wurzeln mit Zahlen  $r_1, \dots, r_{n-k}$  versehen sind. Man vereinigt ähnlicherweise die beiden Bäume mit minimaler  $r_i$ . Nach  $n - 1$  Schritten ist der Huffman-Baum fertig.

Mit Induktion kann man leicht sehen, dass der Huffman-Baum optimal ist, weil er auch die obige Rekursion befriedigt.

Nachteil: Der Text muss zweimal gelesen werden.

Es gibt auch dynamische Varianten. Z.B. 4.2 BSD UNIX compact.

Anderes Problem: man muss auch die Codetabelle (oder Codebaum) übertragen. Alternative: fixe Codes (z.B. für englische Texte).

## 5.2 Lempel-Ziv-Welch-Methode

Nicht die Buchstaben, sondern die Wörter werden kodiert. → Wörterbuch mit (Wort,Code)-Paaren (Mögliche Implementierung: Trie). Am Anfang beinhaltet das Wörterbuch die Wörter mit Länge 1, später wird es erweitert. Folgendes wird immer erfüllt (Invariante): falls ein Wort im Wörterbuch enthalten ist, dann sind auch alle seine Präfixe enthalten. Zu jedem Wort im Wörterbuch gehört ein Code fixer Länge (meistens 12-15 bit). Jedes neue Wort bekommt den kleinsten, noch nicht benutzten Code.

Der Algorithmus ist sehr effizient: er liest den Text, bildet gleichzeitig das Wörterbuch, zerteilt den Text gleichzeitig in Segmente, die im Wörterbuch enthalten sind, und schreibt gleichzeitig die entsprechenden Codes aus. (Auch bei der Dekodierung muss der Text nur einmal gelesen werden, und es kann gleichzeitig dekodiert werden – siehe später.) Komprimierung entsteht dadurch, dass oft lange Segmente mit kurzen Codes ersetzt werden können. Asymptotisch optimal (→ Informationstheorie).

Sei  $W$  eine Variable für die Speicherung des aktuellen Wortsegments,  $b$  eine Variable für die Speicherung der aktuellen Buchstabe. Am Anfang besteht  $W$  aus der ersten Buchstabe des Texts. Während des Verfahrens wird immer garantiert, dass  $W$  im Wörterbuch enthalten ist. (Am Anfang ist es offensichtlich so.) Der allgemeine Schritt ist:

1. Die nächste Buchstabe wird in  $b$  eingelesen.
2. Wenn der Text zu Ende ist, dann wird  $code(W)$  ausgeschrieben, und STOP.
3. Wenn auch das Wort  $Wb$  im Wörterbuch enthalten ist, dann  $W := Wb$ , und gehe zu 1.
4. Sonst wird  $code(W)$  ausgeschrieben, und das Wort  $Wb$  wird ins Wörterbuch aufgenommen.  $W := b$  und gehe zu 1.

Der Algorithmus zerteilt den Text in Segmente, die im Wörterbuch enthalten sind. Dabei wird immer das längste, im Wörterbuch enthaltene Segment gewählt (gierige Methode). Ins Wörterbuch wird alles aufgenommen. Wenn das Wörterbuch voll ist (z.B. 12-bit Codes → 4096 Möglichkeiten), dann nicht mehr.

Beispiel:  $\Sigma = \{a, b, c\}$ ,  $code(a) = 1$ ,  $code(b) = 2$ ,  $code(c) = 3$ . Text:  $ababcbababaaaaaa$ . Die Segmente:  $a|b|ab|c|ba|bab|a|aa|aaa|a$ . Die Ausgabe: 1 2 4 3 5 8 1 10 11 1. Das Wörterbuch sieht am Ende so aus:

$a$	1
$b$	2
$c$	3
$ab$	4
$ba$	5
$abc$	6
$cb$	7
$bab$	8
$baba$	9
$aa$	10
$aaa$	11
$aaaa$	12

Bei der Dekodierung wird das Wörterbuch ähnlicherweise aufgebaut. Z.B. fängt die obige Zahlenfolge mit 1 2 an, so dass man weiss, dass der originale Text mit *ab* anfängt. Dann wurde bei der Kodierung *ab* ins Wörterbuch mit Kode 4 aufgenommen usw.

Problemfall: 1 2 4 3 5 wurde schon behandelt, man weiss schon, dass der Text mit *ababcba* anfängt, und die ersten 7 Zeilen des Wörterbuchs sind auch schon bekannt. Danach kommt Kode 8, was noch nicht bekannt ist. Man weiss aber, dass das Wort mit Kode 8 die Form *bax* hat, wobei *x* eine Buchstabe ist. Also fängt der Text mit *ababcba* an, und so ist das gesuchte Wort eben *bab*. Also kann man die Dekodierung fortsetzen.

Beispiel: UNIX *compress*. Ähnliche Idee: bei *zip* und *gzip*.

# Kapitel 6

## Graphentheoretische Algorithmen

### 6.1 Einleitung

Graphen kommen in sehr vielen algorithmischen Anwendungen vor. Sind sehr nützlich zur Repräsentierung von binären Relationen.

Einige Beispiele wurden schon gezeigt. Weitere Beispiele: Datenbanken, Internet, OOP, Automatentheorie, GUI-Design, Elektrische Netzwerke usw.

#### 6.1.1 Grundbegriffe und Notationen

$G = (V, E)$ ,  $n = |V|$ ,  $e = |E|$

Gerichtete und ungerichtete Graphen

Nachbar, Gradzahl

Schlingen, Parallelkanten

Kosten bzw. Gewichte:  $c : E \rightarrow \mathbf{R}$

Weg (Pfad), Kreis, einfacher Weg bzw. Kreis, gerichteter Weg bzw. Kreis

Zusammenhängend, Komponenten

Kreisfrei, Baum, Wald

#### 6.1.2 Speicherung von Graphen

##### Adjazenzmatrix

$$A[i, j] = \begin{cases} 0 & \text{falls } (i, j) \notin E, \\ 1 & \text{falls } (i, j) \in E. \end{cases}$$

Symmetrisch bei ungerichteten Graphen. Wenn auch Kosten gegeben sind:

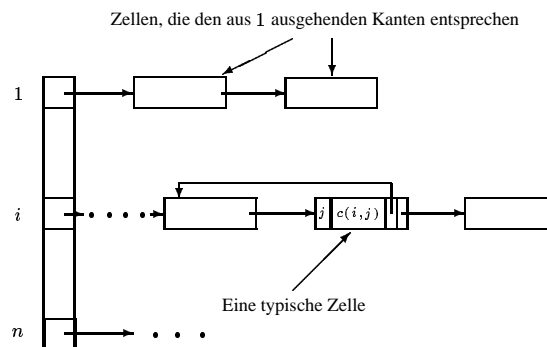
$$A[i, j] = \begin{cases} 0 & \text{falls } i = j, \\ c(i, j) & \text{falls } (i, j) \in E \text{ und } i \neq j, \\ * & \text{sonst.} \end{cases}$$

Manchmal wird statt \* auch  $\infty$  geschrieben.

Speicherbedarf:  $n^2$ . Vorteil: einfache Implementierung; direkter Zugriff auf Kante  $(i, j)$  möglich. Nachteil: Kein effizientes Verarbeiten der Nachbarn eines Knotenpunktes möglich. Sinnvoll bei dichten Graphen, bzw. bei Algorithmen, die direkten Zugriff auf die Kanten benötigen.

## Adjazenzlisten

Zu jedem Knotenpunkt gehört eine Liste mit den aus diesem Knotenpunkt ausgehenden Kanten. Eine mögliche Verwirklichung:



Speicherbedarf:  $n + e$  (bei ungerichteten Graphen:  $n + 2e$ ). Vorteil: kompakter; Nachbarn eines Knotenpunktes können sehr einfach verarbeitet werden. Nachteil: Kein direkter Zugriff auf die Kanteninformationen.

## 6.2 Kürzeste Wege

$G = (V, E)$  gerichteter Graph,  $c(e)$ : Gewichte der Kanten. Gewicht (Länge) eines Weges: Summe der Gewichte der Kanten. Kürzester  $x \rightsquigarrow y$  Weg: Weg mit minimalem Gewicht. Entfernung zwischen zwei Punkten:

$$d(x, y) = \begin{cases} 0 & \text{falls } x=y, \\ \infty & \text{falls es keinen } x \rightsquigarrow y \text{ Weg gibt,} \\ \text{Länge des kürzesten } x \rightsquigarrow y \text{ Weges} & \text{sonst.} \end{cases}$$

Man muss annehmen, dass der Graph keinen gerichteten Kreis mit einem negativen Gewicht enthält. (Sonst ist das Problem viel schwieriger, siehe später.)

Wichtige Fragen:

1. Kürzester Weg zwischen zwei gegebenen Punkten
2. Kürzester Weg zwischen einem gegebenen Punkt und allen anderen Punkten
3. Kürzester Weg zwischen allen Punktpaaren

Jeder bekannte Algorithmus, der die erste Frage beantwortet, beantwortet auch die zweite Frage.

Bemerkung: die *Breitensuche* ist eine Lösung für die zweite Frage im ungewichteten Fall.

## 6.2.1 Algorithmus von Dijkstra

### Verwirklichung mit Adjazenzmatrix

Funktioniert, wenn alle Gewichte nicht-negativ sind. Liefert die Entfernungen zwischen einem gegebenen Punkt  $s$  und allen anderen Punkten.

In einem Feld  $D$  wird die Annäherung der gesuchten Entfernungen gespeichert. Am Anfang ist  $D[x] := A[s, x]$ , wobei  $A$  die Adjazenzmatrix ist. Am Ende wird  $D[x] = d(s, x)$  gültig sein. Während des Algorithmus werden jene Knotenpunkte, für die diese Eigenschaft schon erfüllt ist, in der Menge  $F$  gesammelt. Am Anfang ist  $F := \{s\}$ .

Im allgemeinen Schritt macht man Folgendes:

1. Man wählt  $v \in V \setminus F$ , wofür  $D[v]$  minimal ist.
2.  $F := F \cup \{v\}$ .
3. Für jeden  $x \in V \setminus F$  ist die neue Näherung:  $D[x] := \min\{D[x], D[v] + A[v, x]\}$  (Relax-Schritt)

Ein *besonderer Weg* ist ein  $s \rightsquigarrow x$  Weg, dessen Knotenpunkte bis auf den Endpunkt in  $F$  sind.

**Behauptung 6.1.** *Nach jeder Iteration gelten die Folgenden:*

- (a)  $x \in F \Rightarrow D[x] = d(s, x)$
- (b) Wenn  $x \in F$ , dann gibt es einen kürzesten  $s \rightsquigarrow x$  Weg, dessen Punkte alle in  $F$  sind.
- (c) Wenn  $x \in V \setminus F$ , dann ist  $D[x]$  die Länge des kürzesten besonderen  $s \rightsquigarrow x$  Weges.

*Beweis.* Vollständige Induktion; am Anfang sind alle Eigenschaften erfüllt. Induktionsschritt:

(a) Man muss die Eigenschaft nur für  $v$  beweisen. Indirekt nehme man an, dass es einen kürzeren  $s \rightsquigarrow v$  Weg gibt. Dieser kann wegen der Induktionsbedingung für (c) kein besonderer Weg sein. Aber der Anfang dieses Weges ist ein besonderer Weg. Sei  $w$  der erste Punkt auf dem Weg, der nicht in  $F$  ist. Sei die Länge dieses  $s \rightsquigarrow v$  Weges  $d_1$ , die Länge des  $x \rightsquigarrow w$  Weges  $d_2$ . Dann gilt:

$$D[w] \stackrel{1}{\leq} d_2 \stackrel{2}{\leq} d_1 \stackrel{3}{<} D[v],$$

was ein Widerspruch ist, weil dann der Algorithmus nicht  $v$  sondern  $w$  gewählt hätte. (1: Induktionsbedingung für (c); 2: die Gewichte sind nicht negativ; 3: indirekte Annahme.)

(b) Man muss die Eigenschaft wieder nur für  $v$  beweisen.  $D[v] = d(s, v)$  ist schon bewiesen. Wegen der Induktionsbedingung für (c) war das eben die Länge eines kürzesten besonderen Weges. Dieser Weg läuft jetzt schon vollkommen in  $F$ , und er ist ein kürzester  $s \rightsquigarrow v$  Weg.

(c) Sei  $x \in V \setminus F$ , dann besteht ein kürzester besondere Weg  $W$  zu  $x$  aus einem Weg  $W_1$ , der ganz in  $F$ , zu einem Punkt  $y \in F$  läuft, und aus der Kante  $(y, x)$ . Dabei ist  $W_1$  eben ein kürzester  $s \rightsquigarrow y$  Weg, der ganz in  $F$  läuft. Wegen der Induktionsbedingung für (b) ist dieser eben ein kürzester  $s \rightsquigarrow y$  Weg, also  $c(W_1) = d(s, y)$ . Also ist die Länge des kürzesten besonderen  $s \rightsquigarrow x$  Weges

$$D' = \min_{y \in F} (d(s, y) + A[y, x]).$$

Also ist die Behauptung äquivalent mit  $D[x] = D'$ . Wegen der Induktionsbedingung ist es für den alten Wert von  $D[x]$  schon bekannt:

$$D^{alt}[x] = \min_{y \in F \setminus \{v\}} (d(s, y) + A[y, x]).$$

Wegen  $D[v] = d(s, v)$  folgt daraus wirklich die Behauptung für den neuen Wert von  $D[x]$ :

$$D' = \min\left(\min_{y \in F \setminus \{v\}} (d(s, y) + A[y, x]), d(s, v) + A[v, x]\right) = \min(D^{alt}[x], D[v] + A[v, x]) = D[x]$$

□

Am Ende ist  $V = F$ , also gilt für jeden Knotenpunkt  $x$ :  $D[x] = d(s, x)$ , so dass der Algorithmus korrekt ist.

Zeitbedarf: Initialisierung:  $\mathcal{O}(n)$ . Danach gibt es  $n - 1$  Iterationen; in jeder Iteration muss man das Minimum von  $D$  suchen und danach die  $D$ -Werte in  $V \setminus F$  korrigieren:  $\mathcal{O}(n)$ . Also insgesamt:  $\mathcal{O}(n^2)$ .

### Verwirklichung mit Adjazenzlisten

Wenn es wenig Kanten gibt:

Die Elemente von  $V \setminus F$  werden in einem Heap gespeichert, basierend auf ihre  $D$ -Werte. Das Heap kann in  $\mathcal{O}(n)$  Schritten erstellt werden. Das Auffinden und Löschen des Punktes mit minimalem  $D$  geschieht in  $\mathcal{O}(\log n)$  Schritten. Man muss  $D$  nur für die Nachbarn von  $v$  neu berechnen (deswegen ist die Adjazenzliste vorteilhaft). Sei  $n(v)$  die Anzahl der Nachbarn von  $v$ . Dann braucht man  $n(v) \log n$  Schritte, um die Heapeigenschaft wiederherzustellen. Insgesamt ist das

$$\sum_{v \in V} n(v) \log n = \log n \sum_{v \in V} n(v) = \mathcal{O}(e \log n)$$

Also ist der gesamte Zeitbedarf des Algorithmus  $\mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(e \log n) = \mathcal{O}((n + e) \log n)$ . Wenn  $e$  viel kleiner ist, als  $n^2$ , dann kann das viel effektiver sein. (Z.B. bei planaren Graphen ist  $e = \mathcal{O}(n)$ , also ist der Zeitbedarf  $\mathcal{O}(n \log n)$ .)

Wenn es sehr viele Kanten gibt:

$d$ -Heap kann verwendet werden, dann ist der Zeitbedarf  $\mathcal{O}(n + nd \log_d n + e \log_d n)$  (Heaperstellung, removeMin-Aufrufe, Wiederherstellung der Heap-Eigenschaft). Wenn Z.B.  $n^{1.5} \leq e \leq n^2$ , dann sei  $d := e/n$ . Daraus folgt  $\sqrt{n} \leq d$ , also  $\log_d n \leq 2$ . Daraus folgt:

$$\mathcal{O}(n + nd \log_d n + e \log_d n) = \mathcal{O}(n + nd + e) = \mathcal{O}(n + n \cdot e/n + e) = \mathcal{O}(e)$$

### Bemerkungen

Der Algorithmus funktioniert auch für ungerichtete Graphen, wenn man den ungerichteten Graphen als einen gerichteten Graphen auffasst, in dem alle Kanten in beide Richtungen enthalten sind. In diesem Fall ist die Bedingung, dass es keinen negativen Kreis gibt, äquivalent damit, dass alle Gewichte nicht-negativ sind.

Wenn man nicht nur die Entfernungen, sondern auch einen kürzesten Weg von  $s$  aus erhalten möchte, dann muss man den Algorithmus folgenderweise erweitern. Jeder Knotenpunkt  $x$  erhält einen Zeiger  $P[x]$  auf den vorletzten Knotenpunkt im kürzesten  $s \rightsquigarrow x$  Weg. Am Anfang ist  $P[x] = s$  für jedes  $x \in V$ . Im allgemeinen Schritt des Algorithmus, wenn  $D[x]$  geändert wird, dann muss auch  $P[x]$  entsprechend verändert werden. Am Ende kann man mit Hilfe dieser Zeiger sehr einfach den kürzesten Weg (rückwärts) traversieren.

## 6.2.2 Die Methode von Bellman und Ford

Problem wird verallgemeinert: die Gewichte können jetzt auch negativ sein, aber es darf keinen negativen Kreis geben. Leider wird der Zeitbedarf  $cn^3$  sein.

Es wird angenommen, dass  $G$  mit seiner Adjazenzmatrix gegeben ist, dass weiterhin  $V = \{1, \dots, n\}$  und  $s = 1$ .

Es kann einfach bewiesen werden, dass, falls es keine negative Kreise gibt, dann gibt es zwischen beliebigen zwei Punkten einen kürzesten Weg, der einfach ist. Daraus folgt, dass es zwischen beliebigen zwei Punkten einen kürzesten Weg gibt, der aus höchstens  $n - 1$  Kanten besteht.

Eine Tabelle  $T[1 : n - 1, 1 : n]$  wird ausgefüllt, so dass am Ende  $T[i, j]$  ( $1 \leq i \leq n - 1, 1 \leq j \leq n$ ) die Länge des kürzesten  $1 \rightsquigarrow j$  Weges, der aus höchstens  $i$  Kanten besteht, gibt. Dann wird  $T[n - 1, j] = d(1, j)$ .

In jeder Iteration wird eine neue Zeile ausgefüllt. In der ersten Iteration wird Zeile 1 ausgefüllt: natürlich  $T[1, j] = A[1, j]$ . Nach  $i$  Iterationen sind schon die ersten  $i$  Zeilen richtig ausgefüllt. Dann gilt:

$$T[i + 1, j] = \min(T[i, j], \min_{k \neq j}(T[i, k] + A[k, j])) \quad (6.1)$$

Sei nämlich  $W$  ein  $1 \rightsquigarrow j$  Weg minimaler Länge, der aus höchstens  $i + 1$  Kanten besteht. Falls  $W$  aus weniger als  $i + 1$  Kanten besteht, dann wurde er schon bei  $T[i, j]$  gezählt. Sonst besteht er aus genau  $i + 1$  Kanten. Sei  $v$  der vorletzte Punkt in  $W$ , die Strecke  $1 \rightsquigarrow v$  von  $W$  sei  $W_0$ . Dann ist  $W_0$  minimal unter den  $1 \rightsquigarrow v$  Wegen mit höchstens  $i$  Kanten (sonst könnte man  $W$  verringern – Optimalitätsprinzip), also ist seine Länge eben  $T[i, v]$ . Deswegen ist die Länge von  $W$  eben  $T[i, v] + A[v, j]$ . Auf jeden Fall kommt also die Länge von  $W$  auf der rechten Seite von (6.1) vor.

Um eine Zelle von  $T$  auszufüllen, braucht man  $\mathcal{O}(n)$  Additionen und Vergleiche. Also braucht man für die ganze Methode wirklich  $\mathcal{O}(n^3)$  Schritte.

Wenn man auch die kürzesten Wege erhalten möchte, und nicht nur die Entfernungen, dann kann man die gleiche Idee benutzen, wie bei dem Algorithmus von Dijkstra.

Dieser Algorithmus ist ein Beispiel für die dynamische Programmierung.

Es existiert auch eine Variante mit  $\mathcal{O}(ne)$  Zeitbedarf.

### 6.2.3 Algorithmus von Floyd

Ziel: einen kürzesten Weg zwischen allen Punktpaaren zu finden, falls auch negative Gewichte vorkommen können.

Falls die Gewichte positiv sind, eine mögliche Lösung: den Algorithmus von Dijkstra  $n$ -mal durchführen. Kost:  $\mathcal{O}(n^3)$ .

Sonst: sei  $G$  mit seiner Adjazenzmatrix  $A$  gegeben. Sei  $F \in \mathbf{R}^{n \times n}$ . Am Anfang  $F := A$ .  $F$  wird  $n$ -mal ausgefüllt so, dass in der  $k$ -ten Iteration  $F_k[i, j]$  = Länge des minimalen Weges zwischen  $i$  und  $j$  so, dass nur die Knotenpunkte  $1, \dots, k$  als inneren Knotenpunkte des Weges benutzt werden dürfen.  $F_k[i, j]$  kann wie folgt berechnet werden:

$$F_k[i, j] := \min\{F_{k-1}[i, j], F_{k-1}[i, k] + F_{k-1}[k, j]\} \quad (6.2)$$

Erklärung: falls der kürzeste  $i \rightsquigarrow j$  Weg den Punkt  $k$  nicht enthält, dann  $F_k[i, j] = F_{k-1}[i, j]$ . Falls dieser Weg  $k$  enthält, dann besteht er aus einem kürzesten  $i \rightsquigarrow k$  Weg und aus einem kürzesten  $k \rightsquigarrow j$  Weg, also in diesem Fall ist  $F_k[i, j] = F_{k-1}[i, k] + F_{k-1}[k, j]$ . Aus  $F_{k-1}[i, k] = F_k[i, k]$  und  $F_{k-1}[k, j] = F_k[k, j]$  folgt, dass der Algorithmus in einer einzigen Matrix  $F$  realisiert werden.

```

(1) for  $i := 1$  to  $n$  do
    for  $j := 1$  to  $n$  do
         $F[i, j] := C[i, j]$ 
(2) for  $k := 1$  to  $n$  do
    for  $i := 1$  to  $n$  do
        for  $j := 1$  to  $n$  do
             $F[i, j] := \min\{F[i, j], F[i, k] + F[k, j]\}$ 

```

Nach der  $n$ -ten Iteration enthält  $F$  die gesuchten Werte. Kost:  $\mathcal{O}(n^3)$

## Verfolgung der Wege

Falls man sich nicht nur für die Länge des kürzesten Weges interessiert, sondern auch den Weg selbst kennen möchte:

Sei  $P \in \mathbb{R}^{n \times n}$ . Am Anfang ist  $P = 0$ . Falls das Minimum in 6.2 in dem zweiten Glied aufgenommen wird, dann sei  $P(i, j) = k$ . So enthält  $P(i, j)$  am Ende den Index eines inneren Knotenpunktes im kürzesten Weg von  $i \rightsquigarrow j$ . Mit der folgenden Subroutine kann also der Weg ausgeschrieben werden:

```
procedure minWeg ( $i, j$ :Knotenpunkt);  
var  $k$ :Knotenpunkt;  
begin  
     $k := P[i, j]$ ;  
    if  $k = 0$  then return;  
    minWeg ( $i, k$ );  
    print( $k$ );  
    minWeg ( $k, j$ )  
end;
```

## Anwendungen

### 1. Transitiv Hülle:

Frage:  $\exists i \rightsquigarrow j$  Weg?

Lösung1: mit Floyd: falls  $F(i, j) = \infty$ , dann nicht, sonst ja.

Lösung2: Warshall. Statt Kanten-Gewicht nur 0 oder 1. Dann kann die Änderungsformel für  $F$  so formuliert werden:

$$T[i, j] := T[i, j] \vee (T[i, k] \wedge T[k, j]).$$

Kost:  $\mathcal{O}(n^3)$

### 2. Zentrum suchen:

**Definition 6.1.** Sei

$$e(v) = \max\{d(w, v) : w \in V\}$$

Ein Knotenpunkt  $v \in V$  ist ein Zentrum von  $G$ , falls  $e(v)$  minimal ist.

1. Führe den Algorithmus von Floyd aus. ( $\mathcal{O}(n^3)$ )
2. Bilde das Maximum der Spalten in  $F$ . So bekommt man  $e(v)$ . ( $\mathcal{O}(n^2)$ )
3. Bestimme das Minimum von  $e(v)$ . ( $\mathcal{O}(n)$ )

Insgesamt:  $\mathcal{O}(n^3)$ .

## 6.3 Tiefensuche

DFS: Depth First Search, mélyiségi bejárás.

Ziel: jeden Punkt genau einmal zu besuchen.

Methode: aus einem beliebigen Punkt angefangen so lange „tief“ in den Graphen gehen, bis man unbesuchte Punkte findet. Dann eine Kante zurücktreten, und weitersuchen, usw., bis es sogar in der Wurzel

keine Fortsetzungsmöglichkeit gibt. Falls nicht alle Punkte besucht sind, dann wähle eine neue Wurzel aus den unbesuchten Punkten. Zuerst betrachten wir gerichtete Graphen.

Präziser: sei  $L[v]$  die Liste der ausgehenden Kanten von  $v$ . Sei  $visited$  ein Boole-Vektor der Länge  $n$ :

```

for  $v := 1$  to  $n$  do
     $visited[v] := false$ ;
for  $v := 1$  to  $n$  do
    if  $visited[v] = false$  then
         $tiefensuche(v)$ 

```

**procedure**  $tiefensuche$  ( $v$ : Knotenpunkt)

```

    var
         $w$ : Knotenpunkt;
    begin
(1)     $visited[v] := true$ ;
        for alle Punkte  $w$  in  $L[v]$  do
(2)    if  $visited[w] = false$  then
             $tiefensuche(w)$ 
    end

```

Kost:  $\mathcal{O}(n + e)$ , denn die Funktion  $tiefensuche$  wird für jeden Punkt genau einmal aufgerufen, und in einem Punkt werden alle ausgehende Kanten untersucht.

Die Kanten, die besucht wurden, bilden einen (aufspannenden) Wald. Diese Kanten nennt man Baumkanten. Falls der Graph gerichtet ist, dann können die anderen Kanten in drei Gruppen geteilt werden: Vorwärtskanten, Rückwärtskanten und Kreuzkanten. Eine Kante  $(x, y) \in E$  ist eine Vorwärtskante, falls  $y$  aus  $x$  durch Baumkanten erreicht werden kann. Es ist eine Rückwärtskante, falls  $x$  aus  $y$  durch Baumkanten erreicht werden kann. Sonst heisst es eine Kreuzkante.

Um die Art einer Kante bestimmen zu können, soll man Informationen über den Graphen während der Tiefensuche sammeln.

**Definition 6.2.** Die Tiefennummerierung der Punkte eines Graphen  $G$  drückt die Reihenfolge aus, wie die Werte des  $visited$  Vektors in der Zeile (1) auf  $true$  gesetzt wurden. Die Tiefennummer eines Knotenpunkts  $v$  wird mit  $tn[v]$  bezeichnet.

**Definition 6.3.** Die Abschlussnummerierung der Punkte eines Graphen  $G$  drückt die Reihenfolge aus, wie der Funktionsaufruf  $tiefensuche$  beendet. Die Abschlussnummer eines Knotenpunkts  $v$  wird mit  $an[v]$  bezeichnet.

Klar, dass diese Nummerierungen während der Tiefensuche erstellt werden können, ohne die Grössenordnung der Laufzeit zu verändern. Beispiel...

Die Nummerierung ermöglicht auch, die Kanten während des Algorithmus zu klassifizieren. Falls eine Kante  $(x, y)$  Baumkante ist, dann ist  $y$  beim Auftauchen dieser Kante noch nicht besucht, das heisst:  $tn[y] = 0$ . Falls sie eine Vorwärtskante ist, dann  $tn[x] < tn[y]$  gilt, denn  $y$  kann aus  $x$  durch Baumkanten erreicht werden, und für jede Baumkante gilt, dass ihrer Endpunkt eine grössere Tiefennummer hat, als ihrer Ausgangspunkt. Falls  $(x, y)$  eine Rückwärtskante oder Kreuzkante ist, dann  $tn[x] \geq tn[y]$ . (Schlingen zählen zu Rückwärtskanten.) Bei der Rückwärtskante ist es triviell, und im Falle einer Kreuzkante muss man nur beobachten, dass sie immer von einem später besuchten Ast zu einem früher besuchten Ast führt, und nie umgekehrt. Bei einer Rückwärtskante ist  $an[x] \leq an[y]$ , also bei der Untersuchung der Kante ist  $y$  noch nicht abgeschlossen, also  $an[y] = 0$ . Bei einer Kreuzkante dagegen ist  $y$  schon abgeschlossen, weil  $y$  in einem früheren Ast des Baumes ist. Zusammenfassend:

$x \rightarrow y$	während der Untersuchung der Kante
Baumkante	$tn[y] = 0$
Rückwärtskante	$tn[y] \leq tn[x]$ und $an[y] = 0$
Vorwärtskante	$tn[y] > tn[x]$
Kreuzkante	$tn[y] < tn[x]$ und $an[y] > 0$

Die Klassifikation der Kanten kann auch in  $\mathcal{O}(n + e)$  Zeit gemacht werden.

Sei  $T_x$  der Unterbaum mit Wurzel  $x$  im aufspannenden Wald. Sei

$$S_x = \{y \in V \mid \exists W x \rightsquigarrow y \text{ Weg: } \forall v \in W (tn[v] \geq tn[x])\}$$

**Satz 6.2.**  $\forall x \in V : S_x = T_x$

*Beweis.*  $T_x \subseteq S_x$  ist trivial.

Indirekt sei  $y \in S_x \setminus T_x$ . Das heisst, dass  $\exists W x \rightsquigarrow y$  Weg. Sei  $v$  der letzte Punkt in  $W$  aus  $T_x$ . Es kann angenommen werden, dass  $y$  der erste Punkt von  $W$  aus  $S_x \setminus T_x$  ist, also  $(v, y) \in E$ . Aus  $tn[y] \geq tn[x]$  folgt, dass  $y$  später besucht wurde, als  $x$ , sogar später als alle Punkte in  $T_x$ , da nach  $x$  zuerst die Punkte von  $T_x$  besucht werden. Speziell  $tn[v] \leq tn[y]$ . Daraus folgt, dass  $(v, y)$  eine Baum- oder Vorwärtskante ist. Aber in beiden Fällen würde daraus  $y \in T_x$  folgen, was ein Widerspruch ist.  $\square$

**Folgerung 6.3.** Falls aus  $x$  jeder Punkt erreichbar ist, und  $x$  wird als der Anfangspunkt der Tiefensuche gewählt, dann besteht der aufspannende Wald der Tiefensuche aus einem einzigen Baum.

### 6.3.1 Gerichtete azyklische Graphen

**Definition 6.4.** Ein gerichteter Graph heisst DAG (Directed Acyclic Graph), falls er keinen gerichteten Kreis enthält.

Anwendungen: Projektmanagement (PERT), Datenbanktechnik (Wait-for-graph)

**Satz 6.4.**  $G$  DAG  $\iff$  In einer beliebigen Tiefensuche treten keine Rückwärtskante auf.

*Beweis.*  $\implies$ : triv.

$\impliedby$ : Nehmen wir an, dass  $G$  kein DAG ist. Dann beinhaltet er einen Kreis  $K$ . Sei  $v \in K$  so, dass  $tn[v]$  ist minimal in  $K$ . Sei  $(u, v) \in K \cap E$ . Dann  $tn[v] < tn[u]$ , also  $(u, v)$  ist entweder Rückwärtskante oder Kreuzkante. Aber  $\exists v \rightsquigarrow u$ , nämlich in  $K$ , also  $u \in S_x$ . Wegen Satz 6.2 ist  $u \in T_v$ , also  $(u, v)$  ist eine Rückwärtskante.  $\square$

**Definition 6.5.**  $v_1, v_2, \dots, v_n$  ist eine topologische Ordnung der Knotenpunkte, falls

$$(v_i, v_j) \in E \rightarrow i < j$$

**Satz 6.5.** Ein gerichteter Graph hat eine topologische Ordnung gdw. er ein DAG ist.

*Beweis.*  $\implies$ : triv.

$\impliedby$ : Sei  $v$  ein Quelle (es gibt eine Quelle in DAG – triv.). Mit Induktion existiert eine topologische Ordnung in  $G \setminus v$ . Sei das  $v_1, \dots, v_{n-1}$ . Dann  $v, v_1, \dots, v_{n-1}$  ist eine topologische Ordnung in  $G$ .  $\square$

Realisierung: mit Tiefensuche.

**Satz 6.6.** Die fallende Reihenfolge laut des Wertes  $an[v_i]$  (also  $an[v_i] > an[v_j]$  falls  $i < j$ ) ist eine topologische Ordnung der Punkten.

*Beweis.* Sei  $(v_i, v_j) \in E$ . Da  $G$  ein DAG ist, beinhaltet er keine Rückwärtskante. Also  $(v_i, v_j)$  ist entweder eine Baumkante, oder eine Vorwärtskante oder eine Kreuzkante. In allen Fällen gilt, dass  $an[v_i] > an[v_j]$ . Wegen der Ordnung ist also  $i < j$ . Dann ist es eine topologische Ordnung.  $\square$

### Kürzester und längster Weg in einem DAG

Kürzester Weg: statt  $\mathcal{O}(n^2)$  mit Dijkstra: hier in linearer Zeit:  $\mathcal{O}(n + e)$ .

Sei  $v_1, \dots, v_n$  eine topologische Ordnung der Knotenpunkte. Es kann angenommen werden, dass  $s = v_1$ , weil nur Kanten nach vorne gehen. Dann gilt:

$$d(s, v_i) = \min_{(v_j, v_i) \in E} \{d(s, v_j) + c(v_j, v_i)\},$$

wo  $c(v_j, v_i)$  ist das Gewicht der Kante  $i \rightarrow j$  ist. In  $v_i$  muss man  $d_{v_i}$  Kanten untersuchen, und ein Minimum bilden. (Aufgabe: wie kann man schnell die Kanten, die in  $v_i$  gehen in  $\mathcal{O}(n + e)$  Zeit aus der Adjazenzliste bekommen?) Also die Methode ist  $\mathcal{O}(n + e)$ .

Längster Weg: im allgemeinen sehr schwierig ( $\mathcal{NP}$ -vollständig), aber für DAG in lin. Zeit machbar:

$$l(s, v_i) = \max_{(v_j, v_i) \in E} \{l(s, v_j) + c(v_j, v_i)\} \quad (6.3)$$

### PERT

Program Evaluation and Review Technique.

Man kann die Teilaufgaben einer komplexen Aufgabe mit einem DAG representieren. Die Knotenpunkte sind die einzelnen Teilaufgaben, und zwischen zwei Teilaufgaben  $x$  und  $y$  führt eine Kante, falls  $x$  früher beendet werden muss als  $y$ . Jede Kanten besitzt auch ein Gewicht ( $c(x, y)$ ), das die minimale Zeit zwischen des Beginns der Aufgaben bedeutet. Zwei Teilaufgaben sind speziell:  $A$  (Anfang) und  $E$  (Ende).  $A$  ist eine Quelle,  $E$  ist eine Senke. Der Graph soll natürlich ein DAG sein.

Die Teilaufgabe  $x$  kann nicht früher starten, als  $l(A, x)$ . Das Dauer der ganzen Aufgabe ist dann  $l(A, E)$ . Diese Zeiten können also in  $\mathcal{O}(n + e)$  Zeit bestimmt werden.

**Definition 6.6.** Eine  $(x, y)$  Kante heisst kritisch, falls die ganze Aufgabe verspätet, falls  $y$  später startet, als  $start(x) + c(x, y)$ . Ein  $A \rightsquigarrow E$  Weg ist kritisch, falls er nur aus kritischen Kanten besteht. Ein Knotenpunkt ist kritisch, falls er ein Endpunkt einer kritischen Kante ist.

Feststellung der kritischen Punkten:

Sei  $E$  kritisch. Allgemein, falls  $v_i$  kritisch ist, dann wird  $v_j$  kritisch, falls  $(v_j, v_i) \in E$  und in Equation 6.3 für  $v_j$  Gleichheit ist. (Kost:  $\mathcal{O}(n + e)$ )

### 6.3.2 Stark zusammenhängende Komponenten

**Definition 6.7.** Ein gerichteter Graph heisst stark zusammenhängend, falls

$$\forall v, w \in V : \exists v \rightsquigarrow w$$

Beispiel: zus. aber nicht stark zus.

**Definition 6.8.** Die Relation  $\approx$  wird wie folgt zwischen zwei Punkten  $u, v \in V$  definiert:

$$u \approx v \iff \exists (u \rightsquigarrow v \wedge v \rightsquigarrow u)$$

Klar, dass diese Relation eine Äquivalenzrelation ist.

**Definition 6.9.** Die Äquivalenzklassen von  $\approx$  sind die stark zusammenhängenden (oder kurz: starken) Komponenten von  $G$ .

Seien  $K_1$  und  $K_2$  zwei starke Komponenten von  $G$ . Dann führen alle Kanten entweder von  $K_1$  nach  $K_2$  oder umgekehrt. (Sonst wäre  $K_1 \cup K_2$  eine starke Komponente.)

**Definition 6.10.** Der reduzierte Graph von  $G$  ist ein Graph  $G'$ :  $V(G')$  ist die Menge der starken Komponenten von  $G$ . Zwischen  $K_1, K_2 \in V(G')$  führt eine Kante, falls

$$\exists(u, v) \in E(G) : u \in K_1 \wedge v \in K_2$$

Der reduzierte Graph ist immer ein DAG. Wäre  $K_1, K_2, \dots, K_r$  ein Kreis in  $G'$ , dann wäre  $K_1 \cup K_2 \cup \dots \cup K_r$  eine starke Komponente von  $G$ .

### Bestimmung der starken Komponenten

1. Führe eine Tiefensuche in  $G$  aus.
2. Sei  $G^* = (V^*, E^*)$ , wo  $V^* = V$  und  $(u, v) \in E^* \Leftrightarrow (v, u) \in E$ .
3. Sei  $x$  der Knotenpunkt mit maximalem  $an$  Wert. Führe eine Tiefensuche in  $G^*$  von  $x$  aus. Falls eine neue Wurzel aus den nicht besuchten Punkten gewählt werden soll, wähle immer den Punkt mit grösstem  $an$  Wert.

**Satz 6.7.**  $x \approx y$  gdw. im Punkt (3)  $x$  und  $y$  im selben Baum ist.

Kost:  $\mathcal{O}(n + e)$ .

*Beweis.*  $\Rightarrow$ : Falls  $x \approx y$  in  $G^*$ , dann  $x$  und  $y$  werden in dem selben Baum der Tiefensuche in Punkt (3) sein (eigentlich in allen möglichen Tiefensuchen). Klar, dass die starken Komponenten von  $G^*$  und  $G$  gleich sind, also aus  $x \approx y$  folgt, dass  $x$  und  $y$  im selben Baum sind werden.

$\Leftarrow$ : Nehmen wir an, dass  $x$  und  $y$  in einem Baum nach Punkt (3) sind. Es soll bewiesen werden, dass  $x \approx y$ . Sei  $v$  die Wurzel des Bäumchen von  $x$  und  $y$ . Da  $x$  ein Nachfolger von  $v$  in  $G^*$  ist, es gibt einen  $x \rightsquigarrow v$  Weg  $L$  in  $G$ . Sei  $x' \in L$  der Punkt von  $L$  mit minimalem  $tn$  Wert in der ersten Tiefensuche. Wegen des Satzes 6.2 werden die Punkte des  $x' \rightsquigarrow v$  Teils von  $L$  in der ersten Tiefensuche die Nachfolger von  $x'$  sein. Aber in  $T'_x$  hat  $x'$  die grösste  $an$  Nummer, also ist wegen der Auswahlbedingung von  $v$   $x' = v$ . So  $tn[v]$  ist minimal,  $an[v]$  ist maximal in der ersten Tiefensuche. Das heisst, dass die Punkte von  $L$  Nachfolger von  $v$  sind. Speziell, es gibt einen  $v \rightsquigarrow x$  Weg in  $G$ .

Wir haben bewiesen, dass  $x \approx v$ . Ebenso gilt, dass  $y \approx v$ . Wegen der Transitivität von  $\approx$  gilt, dass  $x \approx y$ . □

### Tiefensuche in ungerichteten Graphen

...

Anwendung: Wie kann man einen Schnittpunkt finden?

**Definition 6.11.** Sei  $G$  ein zusammenhängender ungerichteter Graph. Ein Punkt  $v$  ist Schnittpunkt, falls  $G \setminus v$  besteht aus mehr als eine Komponente.

Algorithmus:

1. Führe eine Tiefensuche aus.

2. Berechne den Wert  $hoch[v]$  für jeden Punkt.  $hoch[v]$  gibt den minimalen  $tn$  Wert solcher Punkte an, die aus einem Nachfolger von  $v$  durch eine Rückwärtskante erreichbar sind. Um es zu verwirklichen, berechne den Wert von  $hoch[v]$  in der Reihenfolge des  $an$  Wertes, angefangen vom minimalen  $an$  Wert, wie folgt:

$$hoch[v] := \min \left\{ \begin{array}{l} tn[v], \\ \min\{tn[z], \text{ wo } v \rightarrow z \text{ Rückwärtskante}\}, \\ \min\{hoch[y], \text{ wo } y \text{ Sohn von } v \text{ ist}\} \end{array} \right\}$$

3. Besuche jeden Punkt im spannenden Baum und entscheide ob er Schnittpunkt ist oder nicht:
- (a) Die Wurzel ist Schnittpunkt, falls sie mindestens zwei Söhne hat.
  - (b) Ein anderer Punkt ist Schnittpunkt, falls er einen solchen Sohn  $y$  hat, wofür  $hoch[y] \geq tn[v]$ .

Kost:  $\mathcal{O}(n + e)$ .

## 6.4 Breitensuche

## 6.5 Minimaler Spannender Baum

**Definition 6.12.** Sei  $G$  ein zusammenhängender ungerichteter Graph. Sei eine Funktion  $c : E \rightarrow \mathbb{R}$  gegeben. Ein Teilgraph  $F$  von  $G$  ist ein spannender Baum, falls  $V(F) = V(G)$  und  $F$  zusammenhängend und kreisfrei ist.  $F$  ist minimal, falls die Summe der Gewichte auf den Kanten von  $F$  minimal ist (unter den spannenden Bäumen).

### 6.5.1 Rot-blau Algorithmus

Die Kanten werden mit zwei Farben, rot und blau gefärbt, und die blauen Kanten bilden den MSB. Während des Algorithmus soll die Färbung erlaubt sein:

**Definition 6.13.** Eine Färbung, die roten, blauen und farblosen Kanten enthält, ist erlaubt, falls  $G$  einen solchen MSB hat, der die blauen aber keine rote Kante enthält.

Die folgenden Regeln darf man benutzen:

**Blaue Regel** Wähle eine Teilmenge  $\emptyset \neq X \subset V$  aus, die keine ausgehende blaue Kante hat. Färbe die kleinste von  $X$  ausgehende farblose Kante auf blau.

**Rote Regel** Sei  $K$  ein Kreis ohne rote Kante. Färbe die grösste farblose Kante auf rot.

Im rot-blau Algorithmus darf man die beiden Regeln in beliebiger Reihenfolge anwenden, bis man kann.

**Satz 6.8.** Im rot-blau Algorithmus bekommt man in jedem Schritt eine erlaubte Färbung. Eine der Regeln kann immer angewendet werden, wenn farblose Kanten noch existieren.

*Beweis.* Nehmen wir an, dass in einem Zwischenzustand die Färbung erlaubt ist. Sei  $F$  ein MSB, der alle blaue Kanten enthält, aber keine roten. Sei  $f$  die Kante, die zunächst gefärbt wird.

(a) *Die blaue Regel wird angewendet:*  $f$  wird also blau. Falls  $f \in E(F)$ , dann  $F$  zeigt, dass die neue Färbung erlaubt ist. Falls  $f \notin E(F)$ : es gibt einen Weg in  $F$ , der zwischen den Endpunkten von  $f$  läuft, da  $F$  ein MSB ist. In diesem Weg gibt es eine Kante  $f'$ , die  $X$  verlässt, da auch  $f$   $X$  verlässt.  $f'$  darf nicht rot sein, weil  $f' \in F$ . Wegen der blauen Regel darf  $f'$  auch nicht blau sein und  $c(f') \geq c(f)$ . Sei

$F' := F \cup f \setminus f'$ . Klar, dass  $F'$  zeigt, dass die neue Färbung erlaubt ist.

(b) *Die rote Regel wird angewendet:*  $f$  wird also rot. Falls  $f \notin E(F)$ , dann  $F$  zeigt, dass die neue Färbung erlaubt ist. Falls  $f \in E(F)$ :  $F \setminus f$  besteht aus zwei Komponenten. Es gibt eine Kante  $f'$  im Kreis  $K$ , wofür die rote Regel angewendet wurde, die zwischen zwei Komponenten von  $F$  läuft.  $f'$  darf nicht rot sein, weil  $f' \in K$ , aber  $f'$  kann auch nicht blau sein, weil  $f' \notin F$ . Wegen der roten Regel ist  $c(f') \leq c(f)$ . Sei  $F' := F \cup f' \setminus f$ . Klar, dass  $F'$  zeigt, dass die neue Färbung erlaubt ist.

Es muss noch bewiesen werden, dass eine der Regeln immer angewendet werden kann. Sei  $f = (u, v)$  eine farblose Kante. Falls  $u$  und  $v$  im gleichen blauen Baum sind, dann kann die rote Regel auf den Kreis  $f \cup$  der blaue Weg zwischen  $u$  und  $v$  angewendet werden. Falls  $f$  zwischen zwei blauen Bäumen führt, dann sei  $X$  einer dieser Bäume.  $\square$

**Folgerung 6.9.** *Falls die Kanten gefärbt sind, dann bilden die blauen Kanten einen MSB. Sogar reicht es  $n - 1$  blauen Kanten zu färben.*

Varianten:

**Primisches Verfahren** Ein einziger blauer Baum ( $F$ ) wächst, am Anfang besteht er aus einem einzigen Punkt, in jedem Schritt wird er mit der blauer Regel ( $X = F$ ) um eins vergrößert.

**Kruskalsches Verfahren** Die nächste zu färbende Kante sei die minimale Kante. Die soll blau sein, falls sie keinen Kreis mit den bisherigen blauen Kanten verursacht, sonst rot. (Diese Methode benutzt auch die blaue-rote Regel, wie es in dem zweiten Teil des Beweises von Satz 6.8 geschrieben wurde.)

**Methode von Borůvka** Zu jedem blauen Baum wähle die kleinste ausgehende Kante. Diese Kanten werden blau. Seien diese Kanten  $f_1, \dots, f_k$ . Hier wird die blaue Regel  $k$ -mal, auf die Mengen  $F_1, \dots, F_k$  angewendet. ( $F_i$  ist die Punktmenge des  $i$ -ten blauen Baumes.) Die Reihenfolge soll so gewählt werden, dass  $c(f_1) < \dots < c(f_k)$ . In diesem Fall  $f_1, \dots, f_{i-1}$  hat keinen Endpunkt in  $F_i$ , sonst hätte man nicht  $f_i$  zu  $F_i$  gewählt.

Dieses Verfahren kann gut parallelisiert werden.

## 6.5.2 Implementierung des Primischen Verfahrens

Sehr ähnlich zum Algorithmus von Dijkstra:

Am Anfang  $F = \{s\}$ ,  $D[v] = A[s, v]$  für jedes  $v \in V$  ( $D[v]$  speichert die Länge der kürzesten Kante zwischen  $F$  und  $v$ ). Im allgemeinen Schritt:

1. Man wählt  $v \in V \setminus F$ , wofür  $D[v]$  minimal ist
2.  $F = F \cup \{v\}$
3. Für jedes  $w \in V \setminus F$ :  $D[w] = \min(D[w], A[v, w])$

Man hat die gleichen Implementierungsoptionen wie beim Algorithmus von Dijkstra (binäres Heap,  $d$ -Heap, usw.).

## 6.5.3 Implementierung des Kruskalschen Verfahrens

Mit Heap kann man die jeweilige minimale Kante in  $\mathcal{O}(\log e)$  Zeit auswählen, insgesamt braucht man dafür also  $\mathcal{O}(e \log e)$  Zeit. (Alternativ kann man die Kanten auch gleich am Anfang ordnen, das geht mit dem gleichen Zeitbedarf.) Die Frage ist, wie man entscheidet, ob diese Kante einen Kreis verursacht oder nicht.

Speichern wir die Punktmengen der blauen Bäume. Wenn die Endpunkte der nächsten Kante in derselben Menge sind, dann verursacht sie einen Kreis, falls sie zwischen zwei Bäumen führt, dann darf sie blau sein, und die zwei Mengen sollen vereinigt werden. Also sind zwei Operationen nötig: WOIST und VEREINIGUNG.

Präziser: sei  $S$  eine Menge. Sei  $U_1, \dots, U_m$  eine Partition von  $S$ .

$$\begin{aligned} \text{VEREINIGUNG}(U_i, U_j) &:= \{U_1, \dots, U_m\} \cup \{U_i \cup U_j\} \setminus \{U_i, U_j\} \\ \text{WOIST}(v) &:= \text{Name jener Menge } U_i, \text{ die } v \text{ beinhaltet.} \end{aligned}$$

Am Anfang besteht jedes  $U_j$  aus einem Element. In dem Kruskalschen Verfahren braucht man zwei WOIST Aufrufe, um zu entscheiden, ob die Endpunkte der nächsten Kante im gleichen Baum sind oder nicht. Falls nein, dann soll man eine VEREINIGUNG machen.

Sei  $U_j$  ein nach oben gerichteter Baum, mit einer Wurzel. Die Elemente von  $U_j$  sind in den Knotenpunkten, und jeder Knotenpunkt hat einen Zeiger auf seinen Vater. Der Name des Baumes ist die Wurzel. In der Wurzel ist ausserdem die Grösse des Baumes gespeichert. Ausserdem gibt es ein Feld der Grösse  $|S|$ , dessen Zellen auf das Vorkommen der Elemente im Baum zeigen.

VEREINIGUNG:  $U_i \cup U_j$  bekommt man wie folgt: falls  $|U_i| \leq |U_j|$ , dann wird  $U_i$  als ein Kind zu der Wurzel von  $U_j$  zugeknüpft.

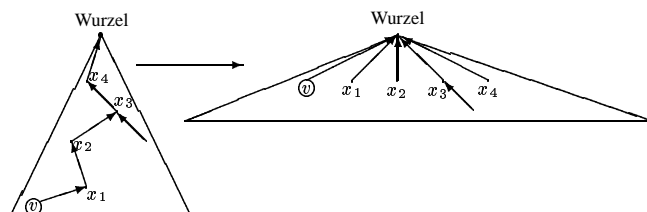
WOIST( $v$ ): den Vater-Zeigern folgend klettert man im Baum von  $v$  zur Wurzel.

Bei einem VEREINIGUNG Aufruf werden die Punkte im kleineren Baum mit eins ferner von der Wurzel liegen, die Punkte im grösseren Baum bleiben unverändert. Aber der kleinere Baum wird mindestens verdoppelt, also kann ein Knotenpunkt höchstens  $\log_2 n$ -mal im kleineren Baum sein. Daraus folgt, dass die Tiefe der Bäume höchstens  $\log n$  beträgt, also dauert WOIST  $\mathcal{O}(\log_2 n)$  lang.

Im Kruskalschen Verfahren macht man  $n - 1$  VEREINIGUNG und  $2e$  WOIST Aufrufe. Insgesamt also  $\mathcal{O}(e \log n + n) = \mathcal{O}(e \log n) = \mathcal{O}(e \log e)$ .

**Folgerung 6.10.** Die totale Kost der Kruskalschen Methode ist  $\mathcal{O}(e \log e)$ .

Eine mögliche Verbesserung: den Suchweg von WOIST zusammendrücken, damit spätere WOIST-Aufrufe beschleunigt werden. Nicht nur die Punkte auf dem Suchweg werden damit näher zur Wurzel sein, sondern auch andere.



**Satz 6.11.** Falls die Sortierung der Kanten in  $\mathcal{O}(e)$  Zeit lösbar ist (z.B. weil die Gewichte kleine ganze Zahlen sind, oder die Kanten sind schon am Anfang geordnet), dann kann die Kruskalsche Methode in  $\mathcal{O}(e\alpha(e))$  Zeit durchgeführt werden.  $\square$

Hier ist  $\alpha(x)$  die sogenannte invers Ackermann Funktion, die zwar gegen unendlich strebt, aber so langsam, dass sie in praktischen Fällen kleiner als 4 ist. Also ist die obige Grenze praktisch  $\mathcal{O}(e)$ .

## 6.6 Maximale Paarungen in bipartiten Graphen

Satz von König, Hall und Frobenius über maximale Paarungen in bipartiten Graphen. Satz von König über die Grösse der maximalen Paarung in bipartiten Graphen. Algorithmus von König für maximale Paarung in bipartiten Graphen,  $\mathcal{O}(ne)$  ( $\mathcal{O}(\sqrt{ne})$  auch möglich). Algorithmus von Egerváry für vollständige Paarung mit minimalem Gewicht in bipartiten Graphen (ungarische Methode, in Polynomzeit).

(Allgemeine Graphen: Satz von Tutte, Algorithmus von Edmonds,  $\mathcal{O}(n^2e)$ .)

## 6.7 Maximaler Fluss und minimale Schnittmenge

Max-flow-min-cut Satz, Algorithmus von Ford und Fulkerson (nicht unbedingt polinomial, sogar nicht unbedingt endlich). Edmonds-Karp Methode  $\mathcal{O}(ne^2)$ , Edmonds-Karp-Dinic  $\mathcal{O}(n^2e)$ . Preflow-push Algorithmus (Goldberg-Tarjan,  $\mathcal{O}(ne \log \frac{n^2}{e})$ ). Varianten: gewichteter Fluss, untere Kapazitäten usw. (auch in Polynomzeit).

Minimale Schnittmenge in ungerichteten Graphen: Algorithmus von Nagamochi und Ibaraki. Algorithmus von Karger (randomisiert).

# Kapitel 7

## Turing-Maschinen

In der Theorie der formalen Sprachen: zur Akzeptierung von Sprachen 0-ter Klasse.

Hier: zur Formalisierung der Begriffe *Algorithmus*, *Zeitbedarf*, *Speicherbedarf*, *Komplexität*, *Berechenbarkeit etc.*

### 7.1 Grundlegende Definitionen

Formale Definition:

$$TM = (Q, T, \sqcup, I, q_0, F, \delta)$$

wobei

$Q$ : endliche Menge der Zustände

$T$ : endliche Menge der Bandzeichen

$\sqcup$ : ein besonderes Zeichen mit der Bedeutung: Zelle ist leer ( $\sqcup \in T$ )

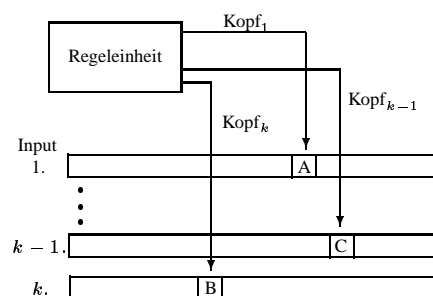
$I$ :  $I \subseteq T \setminus \{\sqcup\}$  ist das Eingabealphabet

$q_0$ :  $q_0 \in Q$  ist der Anfangszustand

$F$ :  $F \subseteq Q$  ist die Menge der Akzeptierungszustände (bei Entscheidungsproblemen)

$\delta$ :  $\delta : Q \times T^k \rightarrow Q \times (T \times \{rechts, links, bleibt\})^k$  ist eine partielle Funktion, die Übergangsfunktion.  
Wenn  $\delta$  in der aktuellen Situation nicht definiert ist, bleibt TM stehen. In den Zuständen  $q \in F$  ist  $\delta$  nicht definiert.

Turing-Maschine kann folgenderweise veranschaulicht werden:



- Endliche zentrale Regeleinheit (endliche Zustandmenge)
- $k$  Bänder; jedes Band ist in einer Richtung unbegrenzt
- Jedes Band besteht aus Zellen, jede Zelle beinhaltet ein Bandzeichen
- Zu jedem Band gehört ein Lese/Schreibekopf
- Jeder Kopf kann sich in beide Richtungen bewegen

Funktionsweise:

- Input ist am Anfang auf Band 1, beginnend mit der ersten Position; sonst ist in jeder Zelle  $\sqcup$
- In jedem Schritt werden die Zeichen unter den Köpfen gelesen, und die TM handelt basierend auf ihrem Zustand und den gelesenen Zeichen: sie übergeht in einen neuen Zustand, schreibt auf jedes Band ein Symbol und bewegt die Köpfe
- Die TM hält an, wenn  $\delta$  nicht definiert ist; in diesem Fall ist das Output der Zustand der TM und/oder das Inhalt von Band  $k$
- Endlosschleife auch möglich

TM kann benutzt werden, um

1. eine Sprache zu akzeptieren (Entscheidungsproblem)
2. eine Funktion zu berechnen

**Definition 7.1.** Die von der TM  $M$  akzeptierte Sprache  $L_M = \{s \in I^* \mid M \text{ gestartet mit Input } s \text{ hält nach endlich vielen Schritten in einem akzeptierenden Zustand}\}$

D.h.: wir haben einen „halben Algorithmus“. Für  $s \notin L_M$  bekommen wir nicht unbedingt nach endlich vielen Schritten eine Antwort.

**Definition 7.2.** Die von TM  $M$  berechnete Funktion ist eine partielle Funktion  $f_M : I^* \rightarrow I^*$ .  $f_M(s) = w$  bedeutet: wenn man  $M$  mit  $s$  als Eingabe startet, hält  $M$  nach endlich vielen Schritten und auf dem Outputband ist eben  $w$ .

Die Funktion ist partiell, weil sie nur für solche Eingaben definiert ist, für die  $M$  nach endlich vielen Schritten anhält.

TM  $\sim$  Computer mit einem Programm  $\sim$  Algorithmus

Zeitbedarf: die Anzahl von Schritten.

Speicherbedarf: die Anzahl der benutzten Zellen, aber es gibt mehrere Varianten. Es wird oft angenommen, dass das Inputband nur gelesen bzw. das Outputband nur geschrieben werden darf. Dann werden nur jene Zellen gezählt, die sowohl schreibbar als auch lesbar sind (also nur die Arbeitsbänder).

$T_M(n)$ : der maximale Zeitbedarf an Eingaben der Länge  $n$

$S_M(n)$ : der maximale Speicherbedarf an Eingaben der Länge  $n$

Wenn  $T_M(n) < T_N(n)$  für genügend grosse Werte von  $n$ , dann ist  $M$  effizienter als  $N$ .

Es gibt nicht immer einen besten Algorithmus:

**Satz 7.1 (Beschleunigungssatz).** Es gibt eine Sprache  $L$  mit den folgenden Eigenschaften:

1.  $\exists M$  TM mit  $L_M = L$  und  $\forall n \ T_M(n) < \infty$
2. Sei  $N$  eine beliebige TM mit  $L_N = L$ . Dann gibt es eine TM  $N'$  mit  $L_{N'} = L$  und  $T_{N'}(n) = \mathcal{O}(\log T_N(n))$

□

## 7.2 Simulationen

Mehrere verschiedene Maschinenmodelle, die äquivalent sind: sie können einander simulieren. Kompliziertere Maschine  $\rightarrow$  schnellere Algorithmen.

**Satz 7.2.** Sei  $M$  eine TM mit  $k$  Bändern. Dann existiert eine TM  $M'$  mit einem Band, wofür

1.  $L_M = L_{M'}$  bzw.  $f_M = f_{M'}$
2.  $T_{M'}(n) \leq 2T_M^2(n)$
3.  $S_{M'}(n) \leq S_M(n) + n$

*Beweisskizze.* Das Bandalphabet von  $M'$  wird so gross sein, dass ein Bandzeichen von  $M'$  Information über  $k$  Bandzeichen von  $M$  speichern kann und zusätzlich noch die Positionen der Köpfe. Veranschaulichung: das Band von  $M'$  besteht aus  $2k$  Spuren; Spur  $2i - 1$  entspricht Band  $i$  von  $M$ , und Spur  $2i$  beinhaltet nur ein einziges Zeichen auf der Position, wo sich der  $i$ -te Kopf von  $M$  befindet.

Band 1	$D$	$\dots$	$A$	$\dots$	$B$	$\dots$
Kopf 1	$\sqcup$	$\dots$	$\uparrow$	$\dots$	$\sqcup$	$\dots$
.						
.						
.						
Band $k$	$D$	$\dots$	$D$	$\dots$	$B$	$\dots$
Kopf $k$	$\sqcup$	$\dots$	$\sqcup$	$\dots$	$\uparrow$	$\dots$

$M'$  simuliert einen Schritt von  $M$  folgenderweise. Zuerst geht sie von der ersten Position des Bandes zur letzten benutzten Position. Dabei merkt sie sich (im Zustand) die Zeichen, die unter den Köpfen stehen. Aufgrund dieser Information und dem Zustand von  $M$  (was auch im Zustand von  $M'$  kodiert ist) stellt sich heraus, was  $M$  machen würde.  $M'$  übergeht in den entsprechenden Zustand, und geht zurück zur ersten Position. Dabei schreibt sie die Symbole auf die Spuren und bewegt die Köpfe.

Die letzte Position ist höchstens  $T_M(n)$ , also braucht  $M'$  für jeden Schritt von  $M$  höchstens  $2T_M(n)$  Schritte. Insgesamt sind das höchstens  $2T_M^2(n)$  Schritte.

Der Speicherbedarf von  $M'$  kann nur dadurch grösser sein, als der von  $M$ , dass die Eingabe bei  $M$  nicht mitgezählt wurde. □

Man kann auch beweisen, dass die quadratische Wachstum des Zeitbedarfs allgemein nicht verbessert werden kann. Denn: die Sprache der Palindrome kann mit einer TM mit einem Band nur in  $\Theta(n^2)$  Schritten akzeptiert werden (siehe Satz 7.30), aber mit einer TM mit 2 Bändern geht es schon in  $\Theta(n)$  Schritten.

Wenn man 2 Bänder erlaubt, ist der Verlust an Effizienz kleiner:

**Satz 7.3.** Sei  $M$  eine TM mit  $k$  Bändern. Dann existiert eine TM  $M'$  mit zwei Bändern, wofür

1.  $L_M = L_{M'}$  bzw.  $f_M = f_{M'}$
2.  $T_{M'}(n) = \mathcal{O}(T_M(n) \log T_M(n))$
3.  $S_{M'}(n) \leq S_M(n) + n$

□

Konstanten sind Hardware-abhängig:

**Satz 7.4.** Sei  $T_M(n) \leq cn$  ( $c > 0$ ). Sei  $\varepsilon > 0$ . Dann existiert eine TM  $N$  mit  $L_N = L_M$  und  $T_N(n) \leq n(1 + \varepsilon)$  ( $n \geq n_0$ ).

*Beweisskizze.* Das Bandalphabet von  $N$  wird so gross sein, dass  $m$  Bandzeichen von  $M$  in einem Bandzeichen von  $N$  gespeichert werden können.

Falls  $M$   $k$  Bänder hat, dann wird  $N$   $k + 1$  Bänder haben. Auf dem ersten Band ist die Eingabe von  $M$ .  $N$  fängt damit an, dass sie die Eingabe „komprimiert“ auf ein anderes Band kopiert. Danach wird Band 1 nicht mehr benutzt.

Betrachten wir  $m$  nacheinanderfolgenden Schritte von  $M$ . In diesen Schritten spielen nur höchstens drei komprimierte Zellen (auf jedem Band) eine Rolle: die aktuelle Zelle und ihre beiden Nachbarn, denn die Köpfe von  $M$  haben sich höchstens um  $m$  Zellen bewegt. Also funktioniert  $N$  wie folgt: anhand dieser 3 Zellen pro Band können die nächsten  $m$  Schritte von  $M$  simuliert werden. Die  $m$  Schritte können in höchstens 7 Schritten simuliert werden: links, rechts, rechts, links, links, rechts, rechts. Kosten:

**Komprimierung:**  $n$

**Zurückspulen:**  $\lceil \frac{n}{m} \rceil$

**Operation:**  $7 \lceil \frac{T_M(n)}{m} \rceil$

Insgesamt:

$$\begin{aligned} T_N(n) &\leq n + \lceil \frac{n}{m} \rceil + 7 \lceil \frac{T_M(n)}{m} \rceil \leq n + \frac{n}{m} + \frac{7T_M(n)}{m} + 8 \leq n + \frac{n}{m} + \frac{7cn}{m} + \frac{8n}{n} \leq \\ &\leq n \left( 1 + \frac{1}{m} + \frac{7c}{m} + \frac{8}{n} \right) \leq n(1 + \varepsilon) \end{aligned}$$

wenn  $m$  und  $n$  gross genug sind, damit  $\frac{1}{m} + \frac{7c}{m} + \frac{8}{n} \leq \varepsilon$  □

Umgekehrt: die Anzahl der Bandzeichen kann auf 2 reduziert werden; dabei wird der Zeitbedarf um eine multiplikative Konstante vergrössert.

## Universelle Turing-Maschine

Interpreter

Input besteht aus zwei Teilen:

1. Beschreibung einer TM  $M$  (Programm). Annahmen (zur Bequemlichkeit):  $M$  hat ein Band, einen Akzeptierungszustand ( $|F| = 1$ ) und  $I = \{0, 1\}$ .
2. Eine Eingabe  $s$  für  $M$ .

Eine universelle TM  $U$  macht genau das, was  $M$  machen würde, wenn man es mit der Eingabe  $s$  startet.

### Mögliche Beschreibung einer TM $M$

$$M = (Q, T, I, \sqcup, \delta, q_0, F)$$

Annahmen:

- $I = \{0, 1\}, T = \{0, 1, \dots, t\}, \sqcup = t$
- $Q = \{0, 1, \dots, q\}, q_0 = 0, F = \{q\}$
- $links = 0, rechts = 1, bleibt = 2$

Dann ist  $Code(M)$ :

$$q\#t\#q_1\#x_1\#q'_1\#x'_1\#b'_1\#\dots\#q_r\#x_r\#q'_r\#x'_r\#b'_r\#\#$$

wobei  $q_i\#x_i\#q'_i\#x'_i\#b'_i$  die Regel  $\delta(q_i, x_i) = (q'_i, x'_i, b'_i)$  kodiert.

Das kann man auch binär kodieren. Hauptsache:

**Satz 7.5.** *Jede TM  $M$  kann eindeutig mit einem Code  $w \in I^*$  kodiert werden. Zu einem beliebigen  $w \in I^*$  gehört höchstens eine TM  $M_w$ . Sowohl die Kodierung als auch die Dekodierung ist durchführbar mit einem Algorithmus.  $\square$*

**Satz 7.6.** *Es gibt eine universelle TM  $U$ , d.h.:*

*Wenn das Input von  $U$   $w\#s$  ist, und  $M_w$  existiert, dann akzeptiert  $U$  das Input (lehnt ab, bzw. fällt in Endlosschleife) genau dann wenn  $M_w$  die Eingabe  $s$  akzeptiert (ablehnt bzw. in Endlosschleife fällt).*

*Beweisskizze.*  $U$  hat 3 Bänder. Auf dem ersten Band ist das Input  $w\#s$ . Dieser Band dient zur Speicherung bzw. Interpretation der Übergangsfunktion von  $M_w$ . Band 2 entspricht dem einzigen Band von  $M_w$ . Band 3 enthält immer die Beschreibung des aktuellen Zustandes von  $M_w$ .

Zuerst prüft  $U$ , ob  $M_w$  existiert. Falls nein, dann lehnt  $U$  das Input ab. Sonst kopiert  $U$  die Eingabe  $s$  auf Band 2, und schreibt den Code des Anfangszustandes auf Band 3. Danach fängt die eigentliche Simulation an. In jedem Schritt liest  $U$  den Zustand und das Zeichen unter dem Kopf der simulierten TM, und bestimmt aufgrund der Übergangsfunktion den nächsten Schritt von  $M_w$ . Band 2 und 3 werden entsprechend modifiziert.

$U$  hält genau dann, falls  $M_w$  hält, in diesem Fall akzeptiert  $U$  das Input genau dann, wenn die Beschreibung des Akzeptierungszustands auf Band 3 ist.  $\square$

## 7.3 Berechenbarkeit

### 7.3.1 Definitionen

**Definition 7.3.** *Die Sprache  $L \subseteq I^*$  ist rekursiv aufzählbar, falls eine TM  $M$  existiert mit  $L = L_M$ . Die Menge der rekursiv aufzählbaren Sprachen wird mit  $\mathcal{RE}$  bezeichnet.*

**Definition 7.4.** *Die Sprache  $L \subseteq I^*$  ist rekursiv, falls eine TM  $M$  existiert mit  $L = L_M$  und  $M$  hält bei jeder Eingabe nach endlich vielen Schritten. Die Menge der rekursiven Sprachen wird mit  $\mathcal{R}$  bezeichnet.*

**Definition 7.5.** *Die partielle Funktion  $f : I^* \rightarrow I^*$  ist partiell rekursiv, falls eine TM  $M$  existiert mit  $f = f_M$ .*

**Definition 7.6.** *Die Funktion  $f : I^* \rightarrow I^*$  ist rekursiv, falls sie partiell rekursiv und überall definiert ist.*

Grund der Benennung „rekursiv“: die rekursiven Funktionen sind diejenigen, die aus gewissen einfachen Funktionen mit Hilfe von Rekursion gebildet werden können. (Wird hier nicht behandelt.)

Grund der Benennung „rekursiv aufzählbar“: Elemente einer solchen Sprache können mit Hilfe eines Algorithmus aufgezählt werden. (Siehe später.)

**Church-Turing These.** *Die Turing-Berechenbarkeit ist äquivalent mit der „wirklichen“ Berechenbarkeit:*

- Die partielle Funktion  $f : I^* \rightarrow I^*$  ist algorithmisch berechenbar  $\iff$   $f$  ist partiell rekursiv.
- Die Funktion  $f : I^* \rightarrow I^*$  ist algorithmisch berechenbar  $\iff$   $f$  ist rekursiv.
- Das Entscheidungsproblem  $w \stackrel{?}{\in} L$  ist algorithmisch entscheidbar  $\iff$   $L$  ist rekursiv.

**Definition 7.7.** *Algorithmisch berechenbar/entscheidbar: rekursiv. Algorithmisch nicht berechenbar/unentscheidbar: nicht rekursiv.*

## 7.3.2 Beziehung zwischen den Berechenbarkeitsbegriffen

### Rekursive und rekursiv aufzählbare Sprachen

**Behauptung 7.7.**  $\mathcal{R} \subseteq \mathcal{RE}$ . □

**Definition 7.8.** Sei  $X \subseteq 2^{I^*}$  eine Sprachenklasse. Dann ist  $coX = \{L \subseteq I^* : I^* \setminus L \in X\}$ .

**Behauptung 7.8.**

1.  $co(coX) = X$ .

2.  $X \subseteq Y \Rightarrow coX \subseteq coY$ . □

**Satz 7.9.**

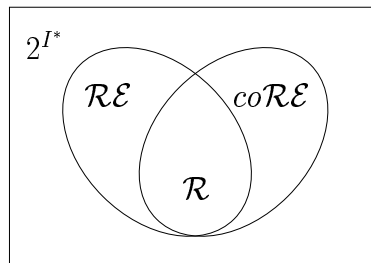
1.  $\mathcal{R} = co\mathcal{R}$ .

2.  $\mathcal{R} = \mathcal{RE} \cap co\mathcal{RE}$ .

*Beweis.*

1. Sei  $L \in \mathcal{R}$ . Dann gibt es eine TM  $M$ , die immer nach endlich vielen Schritten hält, mit  $L = L_M$ . Wenn man die Akzeptierungszustände und Nicht-Akzeptierungszustände von  $M$  wechselt, erhält man eine TM  $M'$  für  $I^* \setminus L$ , die auch immer nach endlich vielen Schritten hält. Daraus folgt:  $co\mathcal{R} \subseteq \mathcal{R}$ . Daraus und aus Behauptung 7.8 folgt  $\mathcal{R} = co(co\mathcal{R}) \subseteq co\mathcal{R}$ .

2. Aus Behauptung 7.7 und 7.8 folgt  $\mathcal{R} \subseteq \mathcal{RE} \cap co\mathcal{RE}$ . Sei nun  $L \in \mathcal{RE} \cap co\mathcal{RE}$ . Dann existieren TM  $M^1$  und  $M^2$  mit  $L_{M^1} = L$  und  $L_{M^2} = I^* \setminus L$ . Aus diesen beiden kann man eine TM  $M^3$  für  $L$  konstruieren, die immer nach endlich vielen Schritten hält.  $M^3$  simuliert parallel (oder abwechselnd) die Operation von  $M^1$  und  $M^2$  auf Eingabe  $s$ . Wenn  $s \in L$ , dann hält  $M^1$  nach endlich vielen Schritten. Wenn  $s \notin L$ , dann hält  $M^2$  nach endlich vielen Schritten. Auf jeden Fall kann  $M^3$  in endlich vielen Schritten das Enthaltungsproblem lösen. □



**Satz 7.10.** Es gibt Sprachen, die nicht rekursiv aufzählbar sind.

*Beweis.* Die Anzahl aller Sprachen ist continuum. Die Anzahl der Turing-Maschinen ist abzählbar unendlich. □

Ähnlicherweise: es gibt Sprachen in  $I^* \setminus co\mathcal{RE}$ , sogar in  $I^* \setminus (\mathcal{RE} \cup co\mathcal{RE})$ .

Die diagonale Sprache ist  $L_d = \{w \in I^* : M_w \text{ existiert, und } w \notin L_{M_w}\}$ .

**Satz 7.11.**  $L_d$  ist nicht rekursiv aufzählbar.

*Beweis.* Indirekt: man nehme an, dass  $L_d = L_M$ . Sei  $w = Code(M)$ , also  $M = M_w$ .

Falls  $w \in L_d$ , dann – laut der Definition von  $L_d$  –  $w \notin L_{M_w}$ . Aber  $L_{M_w} = L_d$ , also in diesem Fall  $w \notin L_d$ , was ein Widerspruch ist.

Falls  $w \notin L_d$ , dann – laut der Definition von  $L_d$  –  $w \in L_{M_w}$  (denn  $M_w$  existiert). Da  $L_{M_w} = L_d$ , folgt daraus  $w \in L_d$ , was wieder ein Widerspruch ist. □

**Behauptung 7.12.**  $L_d \in co\mathcal{RE}$ .

*Beweis.* Man simuliere  $M_w$  (falls es existiert) mit der Eingabe  $w$ . □

Die *universale Sprache* ist:  $L_u = \{w\#s : M_w \text{ existiert, und } s \in L_{M_w}\}$ .

**Satz 7.13.**  $L_u$  ist rekursiv aufzählbar, aber nicht rekursiv.

*Beweis.*  $L_u$  ist rekursiv aufzählbar, da  $L_u = L_U$ , wobei  $U$  eine universelle TM ist.

Man nehme indirekt an, dass  $L_u$  rekursiv ist;  $M$  sei eine TM mit  $L_M = L_u$ , die immer nach endlich vielen Schritten hält. Damit werden wir eine TM  $M'$  konstruieren mit  $L_{M'} = L_d$ .

$M'$ , gestartet mit der Eingabe  $w$ , überprüft zuerst, ob  $w$  ein gültiger Code ist, also ob  $M_w$  existiert. Falls nicht, dann hält  $M'$  in einem Nicht-Akzeptierungszustand. Sonst startet er  $M$  mit der Eingabe  $w\#w$ .  $M$  wird nach endlich vielen Schritten halten. Wenn  $M$  in einem Akzeptierungszustand gehalten ist, dann hält  $M'$  in einem Nicht-Akzeptierungszustand, sonst in einem Akzeptierungszustand. □

Die benutzte Idee kann so formalisiert werden:

**Definition 7.9.** Die Sprache  $L^1$  ist *reduzierbar auf die Sprache  $L^2$* , kurz  $L^1 \leq L^2$ , wenn es eine rekursive Funktion  $f : I^* \rightarrow I^*$  gibt, wofür Folgendes gilt:  $x \in L^1 \iff f(x) \in L^2$ .

**Behauptung 7.14.** Sei  $L^1 \leq L^2$ . Dann gilt:

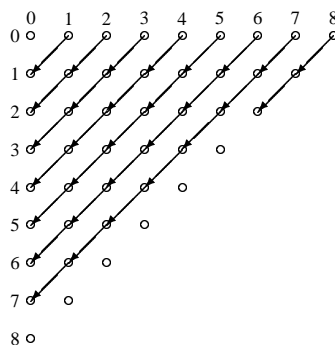
1.  $L^2 \in \mathcal{RE} \Rightarrow L^1 \in \mathcal{RE}$
2.  $L^2 \in \mathcal{R} \Rightarrow L^1 \in \mathcal{R}$
3.  $L^1 \text{ ist unentscheidbar} \Rightarrow L^2 \text{ ist unentscheidbar}$  □

## Sprachen und Funktionen

**Satz 7.15.**  $L \in \mathcal{RE}$  gdw. es gibt eine partiell rekursive Funktion  $f$  mit  $Im(f) = L$ .

*Beweis.* Sei  $L \in \mathcal{RE}$ , dann gibt es eine TM  $M$  mit  $L = L_M$ . Wir konstruieren eine neue TM  $M'$ , die Folgendes macht: zuerst kopiert sie ihr Eingabewort auf das Ausgabeband, und dann simuliert sie  $M$  (mit der gleichen Eingabe), mit dem einzigen Unterschied, dass  $M'$  in eine Endlosschleife gerät, falls  $M$  die Eingabe ablehnt. Daraus folgt, dass  $M'$  nur bei solchen  $s$  Eingaben hält, wofür  $s \in L$  gilt, und bei so einer Eingabe ist die Ausgabe eben  $s$ . Also gilt  $Im(f_{M'}) = L$ .

Umgekehrt, sei  $f$  eine partiell rekursive Funktion, also es gibt eine TM  $M$  mit  $f = f_M$ . Wir konstruieren eine neue TM  $M'$ , die eben die Wörter in  $Im(f)$  akzeptieren wird. Man betrachte eine Reihenfolge der Elemente von  $\mathbb{N} \times I^*$ , in der man das nächste Element algorithmisch generieren kann. Z.B.: man ordne  $I^*$  der Länge nach, und gleich lange Elemente lexikographisch. Dann kann man die  $(i, w_j)$  Paare folgenderweise ordnen:  $(0, w_0), (0, w_1), (1, w_0), (0, w_2), (1, w_1), (2, w_0)$  usw.



$M'$  iteriert in dieser Ordnung. Bei dem Paar  $(i, w_j)$  simuliert sie die ersten (höchstens)  $i$  Schritte von  $M$  bei der Eingabe  $w_j$ . Falls  $M$  in diesen  $i$  Schritten terminiert, dann überprüft  $M'$ , ob die Ausgabe von  $M$  eben gleich der Eingabe von  $M'$  ist. Falls ja, dann akzeptiert  $M'$  ihre Eingabe. In allen anderen Fällen iteriert  $M'$  weiter. Es ist einfach zu sehen, dass  $L_{M'} = \text{Im}(f)$ .  $\square$

Mit der gleichen Idee kann man zu einer beliebigen rekursiv aufzählbaren Sprache einen Algorithmus konstruieren, der eben die Elemente der Sprache aufzählt, daher die Benennung.

**Satz 7.16.** *Nehmen wir an, dass  $\{0, 1\} \subseteq I$ .  $L \subseteq I^*$  ist rekursiv gdw.  $\chi_L$  ist eine rekursive Funktion.*

*Beweis.* Sei  $L = L_M$ , dann soll  $M'$  zuerst  $M$  simulieren, und falls  $M$  die Eingabe akzeptiert, dann eine 1 ausgeben, sonst eine 0. Da  $M$  immer terminiert, terminiert auch  $M'$  immer, und es gilt  $\chi_L = f_{M'}$ . Umgekehrt geht es genauso.  $\square$

### 7.3.3 Weitere unentscheidbare Probleme

Der folgende Satz sagt aus, dass es für jede nicht-triviale Spracheigenschaft algorithmisch unmöglich ist zu entscheiden, ob die Sprache einer mit seiner Beschreibung gegebenen TM die Eigenschaft erfüllt oder nicht.

**Satz 7.17 (Rice).** *Sei  $\emptyset \neq A \subsetneq \mathcal{RE}$ . Sei  $L^A = \{w \in I^* : M_w \text{ existiert und } L_{M_w} \in A\}$ . Dann ist  $L^A$  nicht rekursiv.*

*Beweis.* Indirekt. Sei  $L^0$  die leere Sprache. Man kann annehmen, dass  $L^0 \notin A$ . (Sonst betrachte man  $\mathcal{RE} \setminus A$  statt  $A$ .) Sei  $L^1 \in A$  beliebig,  $L^1 = L_{M^1}$ .

Zu einer beliebigen TM-Beschreibung  $j$  ordne man eine andere TM-Beschreibung  $f(j)$  zu, so dass  $M_{f(j)}$  bei der Eingabe  $i$  Folgendes macht: zuerst simuliert es  $M_j$  mit Eingabe  $j$ , danach  $M^1$  mit Eingabe  $i$ .  $M_{f(j)}$  akzeptiert genau dann, wenn  $M^1$  akzeptiert.

Falls  $M_j$  bei der Eingabe  $j$  in eine Endlosschleife gerät, dann gerät  $M_{f(j)}$  bei jeder Eingabe in eine Endlosschleife, also gilt in diesem Fall  $L_{M_{f(j)}} = L^0 \notin A$ . Sonst ist  $L_{M_{f(j)}} = L_{M^1} \in A$ .

Also fragt man ab, ob  $f(j) \in L^A$  (das geht wegen der indirekten Annahme), und so weiss man, ob  $M_j$  bei der Eingabe  $j$  in eine Endlosschleife fallen würde, oder nicht. Aber mit dieser Information könnte man entscheiden, ob  $j \in L_{M_j}$  oder nicht, also wäre die diagonale Sprache rekursiv, was ein Widerspruch ist.  $\square$

**Haltungsproblem** Es ist unentscheidbar, ob eine TM für eine Eingabe hält oder nicht:

$$L_h := \{w \# s \in I^* \mid \exists M_w \text{ und die Maschine hält für die Eingabe } s\}$$

**Satz 7.18.**  $L_h \in \mathcal{RE} \setminus \mathcal{R}$

*Beweis.* Folgt aus Satz 7.19.  $\square$

Sogar die folgende einfache Version ist unentscheidbar:

$$L_\varepsilon := \{w \in I^* \mid \exists M_w \text{ und die Maschine hält für die leere Eingabe}\}$$

**Satz 7.19.**  $L_\varepsilon \in \mathcal{RE} \setminus \mathcal{R}$

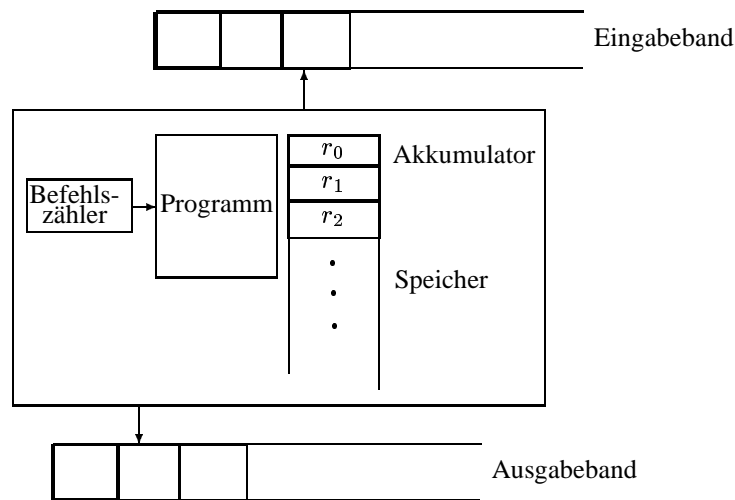
*Beweis.* Es folgt aus dem Satz von Rice (7.17): es ist unentscheidbar, ob eine Sprache  $\varepsilon$  beinhaltet oder nicht. Wäre  $L_\varepsilon \in \mathcal{R}$ , dann könnte man einfach entscheiden, ob  $\varepsilon \in L$ .  $\square$



## 7.4 RAM

### 7.4.1 Einführung

Random Access Machine: ganz ähnlich zu den wirklichen Rechnern. Die Maschine besteht aus einem Eingabeband, aus einem Ausgabeband, aus einem inneren Speicher und aus einem Programmspeicher. Die Bänder und der innere Speicher sind in einer Richtung unendlich. Die Zellen des inneren Speichers können in einem Schritt erreicht werden. Die erste Zelle des inneren Speichers heisst der Akkumulator, der den ersten Operand einer Operation speichert. Das Programm besteht aus Befehlen. Der Befehlszähler zeigt immer auf den aktuellen Befehl.



Der Befehlssatz einer RAM ist der folgende:

Arithmetik		Datenbewegung		Kontroll	
ADD	<i>Op</i>	LOAD	<i>Op</i>	JUMP	<i>Address</i>
SUB	<i>Op</i>	STORE	<i>Op</i>	JGTZ	<i>Address</i>
MULT	<i>Op</i>	READ	<i>Op</i>	JZERO	<i>Address</i>
DIV	<i>Op</i>	WRITE	<i>Op</i>	HALT	

*Op* kann in 3 Formen angegeben werden:

1. = *i* bedeutet die Zahl *i*
2. *i* bedeutet die Zahl in Register *i*
3. \**i* bedeutet die Zahl in Register  $r[i]$ , wo  $r[i]$  die Zahl in Register *i* ist

Bei den arithmetischen Befehlen ist der erste Operand immer das Inhalt des Akkumulators, und das Ergebnis wird in den Akkumulator geschrieben.

READ *Op* liest die aktuelle Zelle von dem Eingabeband in den angegebenen Speicherregister. WRITE *Op* funktioniert umgekehrt. In beiden Fällen bewegt sich der Kopf einen Schritt nach rechts. STORE *Op* speichert den Inhalt des Akkumulators in *Op*. LOAD *Op* wirkt umgekehrt.

Die Springoperationen testen eine Bedingung auf den Inhalt des Akkumulators, und springen auf den Befehl *Address*.

Das Modell ist idealisiert, da der Speicher und die Bänder unendlich lang sind und der Inhalt eines Registers eine beliebig grosse Zahl sein kann.

## 7.4.2 Kost einer RAM

**Uniform-Kost** Die Anzahl der ausgeführten Befehle

**Logarithmische Kost** Die Kost eines Befehls ist die Summe der Länge der Daten in dem Befehl. Die Kost eines Programms ist die Summe der Kost der Befehle.

Das Problem mit Uniform-Kost ist, dass die Länge der Operanden nicht in Betracht genommen wurde. So die Uniform-Kost von  $f(n) = 3^{2^n}$  ist  $\mathcal{O}(n)$ , aber in diesem Fall soll man mit Zahlen der Länge  $2^n$  Operationen durchführen, was nicht realistisch ist, falls  $n$  gross ist.

Beispiel: `ADD *1` hat Uniform-Kost 1, aber logarithmische Kost

$$\text{Länge}(r[0]) + \text{Länge}(r[1]) + \text{Länge}(r[r[1]]) + \text{Länge}(1)$$

Falls ein Programm keine DIV und MUL Operation enthält, dann ist die logarithmische Kost eine gute Messzahl für die Kost. Leider für DIV und MUL wurde bis jetzt keine Methode gefunden, die für Eingaben der Länge  $n$  in  $\mathcal{O}(n)$  Zeit funktioniert. Die besten Methoden brauchen  $\mathcal{O}(n \log n \log \log n)$  Schritte.

Uniform-Kost ist einfacher zu berechnen, aber logarithmische Kost ist genauer. Falls die Wortlänge der Operanden gegeben ist ( $l$ ), dann hat ein Algorithmus mit  $m$  Uniform-Kost  $\mathcal{O}(lm)$  logarithmische Kost. In praktischen Fällen ist es so, also ist Uniform-Kost hinreichend.

## 7.4.3 Simulation

**Satz 7.22.**

(1) Eine TM  $M$  kann mit einer RAM in  $\mathcal{O}(T_M(n) \log T_M(n))$  logarithmischer Kost simuliert werden. Die Uniform-Kost der Simulation ist  $\mathcal{O}(T_M(n))$ .

(2) Ein RAM-Programm mit  $t(n)$  logarithmischer Kost ohne MUL und DIV Operationen kann mit einer TM  $N$  simuliert werden so, dass  $T_N(n) = \mathcal{O}(t^2(n))$ .  $\square$

## 7.5 Kolmogorov-Komplexität

### 7.5.1 Definition

Ziel: die Komplexität (Informationsgehalt, Entropie, Zufälligkeit) von Zeichenketten zu definieren. Idee: Wie lange dauert es, eine Zeichenkette zu definieren (Komprimierbarkeit)? Z.B. ist zwar die Länge der Zahl  $n = 2^k$  eben  $k = \log n$ , aber man kann  $n$  auch mit der Formel  $n = 2^k$  und mit der Zahl  $k$  definieren, und diese Definition hat die Länge  $c + \log k = c + \log \log n$ .

Problem: Sei  $S$  die Menge aller natürlichen Zahlen, die man mit höchstens 1000 Buchstaben in deutscher Sprache definieren kann. Es gibt nur endlich viele Definitionen mit höchstens 1000 Buchstaben, also ist  $S$  auch endlich. Also gibt es eine kleinste natürliche Zahl  $n \notin S$ . Dann ist aber „Die kleinste natürliche Zahl, die man in deutscher Sprache mit höchstens 1000 Buchstaben nicht definieren kann“ eine Definition mit weniger als 1000 Buchstaben für  $n$ .

Das Problem ist, dass die „Definierbarkeit in deutscher Sprache“ kein genauer Begriff ist. Versuchen wir, die Definierbarkeit mit Hilfe von Turing-Maschinen zu definieren:

$$K_M(x) = \begin{cases} \min\{|y| : y \in I^*, f_M(y) = x\} & \text{wenn so ein } y \text{ existiert} \\ \infty & \text{sonst} \end{cases}$$

Leider hängt  $K_M(x)$  sehr stark von  $M$  ab, z.B. kann man zu jedem  $x$  eine TM  $M_1$  mit  $K_{M_1}(x) = 0$  und eine TM  $M_2$  mit  $K_{M_2}(x) = \infty$  konstruieren. Aber bei universellen Turing-Maschinen ist diese Abhängigkeit nicht so stark:

**Satz 7.23 (Invarianz-Satz).** Sei  $U$  eine UTM. Dann gibt es zu jeder TM  $M$  eine Konstante  $c_M$ , so dass für jedes  $x \in I^*$ :

$$K_U(x) \leq K_M(x) + c_M$$

*Beweis.* Sei  $w = \text{code}(M)$  und sei  $y$  eines der kürzesten Wörter mit  $f_M(y) = x$  (also  $K_M(x) = |y|$ ). Dann ist  $f_U(w\#y) = x$ , also gilt

$$K_U(x) \leq |w\#y| = |w\#| + |y| = |w\#| + K_M(x) = c_M + K_M(x)$$

□

**Folgerung 7.24.**  $U_1$  und  $U_2$  seien zwei UTM. Dann gibt es eine Konstante  $c$ , so dass für jedes  $x \in I^*$ :

$$|K_{U_1}(x) - K_{U_2}(x)| \leq c$$

□

Im folgenden betrachten wir eine konkrete UTM  $U$ .

**Definition 7.11.**  $K(x) = K_U(x)$  ist die Kolmogorov-Komplexität des Wortes  $x$ .

## 7.5.2 Einige Eigenschaften der Kolmogorov-Komplexität

Aus dem Invarianzsatz folgt:

**Behauptung 7.25.**  $K(x) \leq |x| + c$

□

Z.B. für die Wörter  $n = 2^k$  folgt sogar:  $K(n) \leq \log \log n + c$ , wie erwartet. ( $K(n)$  kann für manche Werte von  $k$  noch kleiner sein, falls auch  $k$  mit weniger als  $\log k$  Zeichen definiert werden kann.)

Aber die meisten Zeichenketten lassen sich nicht so gut komprimieren:

**Definition 7.12.**  $x$  ist unkomprimierbar, falls  $K(x) \geq |x|$ .

**Satz 7.26.** (i) Sei  $k$  eine positive ganze Zahl. Dann gibt es höchstens  $2^{k+1} - 1$  solche Wörter, deren Kolmogorov-Komplexität  $k$  nicht übersteigt.

(ii)  $\forall n \geq 1 \exists x \in I^* : |x| = n$  und  $x$  ist unkomprimierbar.

(iii) Falls  $n > 8$ , dann ist die Kolmogorov-Komplexität von mehr als 99% der Wörter der Länge  $n$  mehr als  $n - 8$ .

*Beweis.* (i) Sei  $S_k = \{x \in I^* : K(x) \leq k\}$ ,  $T_k = \{y \in I^* : |y| \leq k\}$ . Falls  $x \in S_k$ , dann gibt es mindestens ein Wort  $y$  mit  $|y| \leq k$  (also  $y \in T_k$ ) und  $f_U(y) = x$ . Wenn man zu jedem  $x \in S_k$  ein solches  $y \in T_k$  zuordnet, so bekommt man eine Funktion  $g : S_k \rightarrow T_k$ .  $g$  ist injektiv: wenn  $x_1 \neq x_2$ , dann kann  $g(x_1) = g(x_2)$  auch nicht sein, sonst wäre  $x_1 = f_U(g(x_1)) = f_U(g(x_2)) = x_2$ . Daraus folgt:  $|S_k| \leq |T_k| = 1 + 2 + \dots + 2^k = 2^{k+1} - 1$ .

(ii) Folgt aus (i) mit  $k = n - 1$ : es gibt  $2^n$  Wörter der Länge  $n$ , aber  $|S_{n-1}| \leq 2^n - 1$ . Also es gibt mindestens ein Wort der Länge  $n$ , das nicht in  $S_{n-1}$  enthalten ist; dieses Wort ist unkomprimierbar.

(iii) Folgt aus (i) mit  $k = n - 8$ :  $|S_{n-8}| \leq 2^{n-7} - 1 < 2^{n-7}$ , und es gibt  $2^n$  Wörter der Länge  $n$ , also ist der Anteil der Wörter mit  $K(x) \leq n - 8$  höchstens  $2^{n-7}/2^n = 1/128$ , was weniger als 1% ist. □

Also haben die meisten Wörter eine hohe Kolmogorov-Komplexität. Jedoch:

**Behauptung 7.27.** Es gibt keine TM, die für die Eingabe  $n$  ein Wort  $x_n$  der Länge  $n$  mit  $K(x_n) \geq n/2$  ausgibt.

*Beweis.* Indirekt: sei  $M$  eine solche TM. Dann gilt aber  $K_M(x_n) \leq \lceil \log n \rceil$ , und somit  $K(x_n) \leq \log n + c$ . Aber  $K(x_n) \geq n/2$ , was für grosse Werte von  $n$  ein Widerspruch ist.  $\square$

**Satz 7.28.**  $K$  ist nicht rekursiv.

*Beweis.* Indirekt nehme man an, dass  $K$  rekursiv ist. Dann könnte man eine TM  $M$  wie folgt konstruieren: bei der Eingabe  $n$  prüft  $M$  alle Wörter der Länge  $n$ , fragt ihre Kolmogorov-Komplexität ab, und schreibt nach endlich vielen Schritten ein Wort  $x$  der Länge  $n$  mit  $K(x) \geq n/2$  aus. Das ist möglich, denn es gibt endlich viele Wörter der Länge  $n$ , man kann diese Wörter in der Tat mit einer TM aufzählen, die  $K$ -Werte können in endlich vielen Schritten abgefragt werden, und es gibt wirklich Wörter der Länge  $n$  mit  $K(x) \geq n/2$ . Aber die Existenz von  $M$  widerspricht Behauptung 7.27.  $\square$

Bemerkung: aus diesem Grund tritt das Paradox, das am Anfang des Abschnittes erwähnt wurde, hier nicht auf.

### 7.5.3 Anwendungen

**Satz 7.29.**  $L_h$  ist nicht rekursiv.

*Beweis.* Wäre  $L_h$  rekursiv, dann könnte man  $K(x)$  folgenderweise mit einer TM  $M$  berechnen.  $M$  nimmt nacheinander alle Wörter der Länge 0, der Länge 1, der Länge 2 usw. Bei jedem Wort  $y$  fragt  $M$  ab, ob  $U$  mit der Eingabe  $y$  halten würde oder nicht. Falls nein, dann geht  $M$  zum nächsten Wort hinüber. Sonst startet er  $U$  mit der Eingabe  $y$ , und überprüft, ob die Ausgabe eben  $x$  ist. Falls nein, dann geht  $M$  zum nächsten Wort hinüber. Nach endlich vielen Schritten wird aber ein  $y$  gefunden mit  $f_U(y) = x$ , und  $|y|$  ist minimal, also gilt  $K(x) = |y|$ .  $\square$

Sei  $L_p$  die Sprache der Palindrome. Trivialerweise kann  $L_p$  mit einer TM mit 2 Bändern in linearer Zeit akzeptiert werden. Aber:

**Satz 7.30.** Sei  $M$  eine TM mit einem Band, wofür  $L_M = L_p$ . Dann gilt  $T_M(n) = \Omega(n^2)$ .  $\square$

Das bedeutet auch, dass der quadratische Geschwindigkeitsverlust von Satz 7.2 wirklich notwendig ist.

Man kann auch die „Zufälligkeit“ von Zahlenfolgen definieren. Sei  $x \in I^\infty$ , dann sei  $x_n$  die Folge der ersten  $n$  bits von  $x$ .

**Definition 7.13.**  $x \in I^\infty$  ist (Kolmogorov-)zufällig, falls  $\lim_{n \rightarrow \infty} \frac{K(x_n)}{n} = 1$

**Definition 7.14.**  $x \in I^\infty$  ist algorithmisch erstellbar, falls es eine TM  $M$  gibt, so dass  $f_M(n) = x_n$ .

**Behauptung 7.31.** Falls  $x$  algorithmisch erstellbar ist, dann ist  $x$  nicht zufällig.

*Beweis.* Sei  $M$  die TM, die  $x$  erstellt. Dann gilt  $K_M(x_n) \leq \lceil \log n \rceil$ , und damit  $K(x_n) \leq \log n + c$ . Daraus folgt  $\lim_{n \rightarrow \infty} \frac{K(x_n)}{n} = 0$ .  $\square$

Also ist z.B.  $x = 1111 \dots$  keine zufällige Zahlenfolge. Dieser Begriff entspricht auch in anderen Hinsichten unserem Zufallsbegriff, z.B. ist die Anzahl von 0-en und Anzahl von 1-en in Zufallsfolgen etwa gleich. Das ist aber natürlich anders als in der Wahrscheinlichkeitsrechnung, wo jede Zeichenkette genauso wahrscheinlich ist.

**Satz 7.32.** Man wähle die Bits von  $x$  unabhängig voneinander mit Wahrscheinlichkeit 1/2. Dann ist  $x$  mit Wahrscheinlichkeit 1 zufällig.  $\square$

# Kapitel 8

## Die $\mathcal{NP}$ Sprachenklasse

### 8.1 Zeit- und speicher-begrenzte Maschinen, Sprachenklassen

**Definition 8.1.** Sei  $t : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$  eine Funktion, wofür  $t(n) \geq n$  gilt. Die TM  $M$  ist  $t(n)$  zeit-begrenzt, falls  $T_M(n) \leq t(n)$ .

Die Bedingung  $t(n) \geq n$  bedeutet, dass die Maschine mindestens die Eingabe durchliest.

**Definition 8.2.**

$$TIME(t(n)) := \left\{ L \subseteq I^* \mid \begin{array}{l} L \text{ kann mit einer } \mathcal{O}(t(n)) \text{ zeit-begrenzten} \\ \text{Turing-Maschine } M \text{ erkannt werden.} \end{array} \right\}.$$

In der  $TIME(n)$  Sprachenklasse gehören die Sprachen, die mit einer  $c \cdot t(n)$  zeit-begrenzten TM erkannt werden können. Wichtig, dass die TM immer hält, also  $TIME(t(n))$  besteht nur aus rekursiven Sprachen.

**Definition 8.3.** Zur Sprachenklasse

$$P := \bigcup_{k \geq 1} TIME(n^k)$$

gehören solche Sprachen, die in Polynomzeit erkannt werden können.

**Definition 8.4.** Sei  $s : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$  eine Funktion, wofür  $s(n) \geq \log_2 n$  gilt. Die TM  $M$  ist  $t(n)$  speicher-begrenzt, falls  $S_M(n) \leq s(n)$ .

Die  $s(n) \geq \log_2 n$  Bedingung bedeutet, dass die Maschine mindestens die Eingabe adressieren muss.

**Definition 8.5.**

$$SPACE(s(n)) := \left\{ L \subseteq I^* \mid \begin{array}{l} L \text{ kann mit einer } \mathcal{O}(s(n)) \text{ speicher-begrenzten} \\ \text{Turing-Maschine } M \text{ erkannt werden.} \end{array} \right\}.$$

Ähnlich zu den Sprachen kann man die obigen Sprachenklassen für Funktionen definieren:

**Definition 8.6.**

$$FTIME(t(n)) := \left\{ f : I^* \rightarrow I^* \mid \begin{array}{l} f \text{ kann mit einer } \mathcal{O}(t(n)) \text{ zeit-begrenzten} \\ \text{Turing-Maschine } M \text{ berechnet werden.} \end{array} \right\}.$$

**Definition 8.7.**

$$FSPACE(s(n)) := \left\{ f : I^* \rightarrow I^* \mid \begin{array}{l} f \text{ kann mit einer } \mathcal{O}(s(n)) \text{ speicher-begrenzten} \\ \text{Turing-Maschine } M \text{ berechnet werden.} \end{array} \right\}.$$

Analog zu  $P$  kann man die Klasse der in Polynomzeit berechenbaren Funktionen definieren:

**Definition 8.8.**

$$FP := \bigcup_{k \geq 1} FTIME(n^k)$$

**Satz 8.1.**  $L \in SPACE(s(n)) \implies \exists c_L : L \in TIME(c_L^{s(n)})$

*Beweis.* Sei  $M$  eine  $S(n) := c_1 s(n)$  speicher-begrenzte TM mit  $k$  Bänder, die  $L$  erkennt. Wir möchten eine TM  $N$  konstruieren, die  $\mathcal{O}\left(c_L^{S(n)}\right)$  zeit-begrenzt und  $L$  akzeptiert.

Den aktuellen Zustand einer TM kann man mit dem aktuellen inneren Zustand, mit der Position der Köpfe und mit dem Inhalt der Bänder beschrieben werden. Falls zweimal der gleiche Zustand vorkommt, dann ist die TM in einer endlosen Schleife, also die Eingabe kann abgelehnt werden. Die Anzahl der Zustände  $Z$  kann geschätzt werden:

$$Z \leq |Q||T|^{S(n)} S(n)^k \leq c_2^{S(n)} =: t$$

wo  $|Q|$  die Anzahl der inneren Zustände,  $|T|$  die Anzahl der Symbole,  $S(n)^k$  die Anzahl der Positionen ist. Falls die Maschine nach  $t$  Schritten nicht hält, dann ist sie in einer endlosen Schleife. Das Problem ist, dass  $t$  nicht unbedingt rekursiv berechenbar ist. Folgende Idee löst das Problem.

$N$  beinhaltet zwei Exemplare von  $M$ :  $M_1, M_2$ .  $M_1$  wird mit der Eingabe  $x$  gestartet und nach jedem Schritt wird sie gehalten. Ist  $M_1$  eben im Schritt  $l$ , dann wird  $M_2$  neu gestartet und sie macht  $l - 1$  Schritte. Falls  $M_2$  in den gleichen Zustand kommt, wie  $M_1$ , dann wird  $x$  abgelehnt. Falls nicht, dann  $M_1$  darf einen neuen Schritt machen. Falls  $M_1$  akzeptiert  $x$ , dann wird  $x$  auch durch  $N$  akzeptiert, falls  $M_1$  lehnt  $x$  ab, dann wird  $x$  auch durch  $N$  abgelehnt.

Da  $l < t$  gilt, der maximale Zeitbedarf ist:

$$\mathcal{O}(t^2) = \mathcal{O}\left(\left(c_2^{S(n)}\right)^2\right) = \mathcal{O}\left(c_L^{s(n)}\right)$$

Es ist einfach zu sehen, dass der Speicherplatzbedarf von  $N$   $\mathcal{O}(s(n))$  ist. □

Ebenso kann bewiesen werden:

**Satz 8.2.**  $f \in FSPACE(s(n)) \implies \exists c_f : f \in FTIME(c_f^{s(n)})$  □

**Satz 8.3.**  $TIME(t(n)) = coTIME(t(n))$

*Beweis.* Man tausche die Akzeptierungszustände mit den Ablehnungszuständen. □

**Satz 8.4.**  $SPACE(s(n)) = coSPACE(s(n))$

*Beweis.* Die konstruierte TM im Beweis von Satz 8.1 ist ebenso  $\mathcal{O}(s(n))$  speicher-begrenzt, aber sie hält für jede Eingabe an. Für diese Maschine kann man schon die Akzeptierungszustände mit den Ablehnungszuständen tauschen. □

**Definition 8.9.**  $EXPTIME := \bigcup_{k \geq 1} TIME(2^{n^k})$

**Definition 8.10.**  $PSPACE := \bigcup_{k \geq 1} SPACE(n^k)$

$EXPTIME$  repräsentiert das Universum der in Wirklichkeit vorkommenden Probleme.

**Satz 8.5.**  $P \subseteq PSPACE \subseteq EXPTIME$

*Beweis.* Falls  $M$   $t(n)$  zeit-begrenzt ist, dann ist sie auch  $ct(n)$  speicher-begrenzt, da in einem Schritt kann sie höchstens  $k$  Felder erreichen. ( $k$  ist die Anzahl der Bänder von  $M$ .) Daraus folgt, dass  $TIME(n^k) \subseteq SPACE(n^k) \implies P \subseteq PSPACE$ .

Falls  $L \in PSPACE \implies L \in SPACE(n^k)$  für ein bestimmtes  $k$ , also wegen des Satzes 8.1  $\exists c : L \in TIME(c^{n^k}) \subseteq TIME(2^{n^{k+1}}) \subseteq EXPTIME$ .  $\square$

**Satz 8.6.**  $TIME(t(n)) \subset \mathcal{R}$ ,  $SPACE(s(n)) \subset \mathcal{R}$ ,  $EXPTIME \subset \mathcal{R}$

*Beweis.* Wir beweisen nur  $EXPTIME \subset \mathcal{R}$ . Sei

$$L := \{w \in I^* : \exists M_w, M_w \text{ lehnt } w \text{ in maximum } 2^{2^{|w|}} \text{ Schritten ab}\}$$

Wir zeigen, dass  $L \notin TIME(2^{2^{n-1}})$ . Da  $EXPTIME \subseteq TIME(2^{2^{n-1}})$ , folgt, dass  $L \notin EXPTIME$ . Indirekt, nehmen wir an, dass  $L$  mit einem  $c2^{2^{n-1}}$  zeit-begrenzten TM  $M$  akzeptiert werden kann.  $\exists n_0 : c2^{2^{n-1}} < 2^{2^n}$  falls  $n > n_0$ . Sei  $|w| > n_0$  und sei  $M_w$  eine TM, die  $L$  auch in  $c2^{2^{n-1}}$  Schritten akzeptiert. (Es gibt eine solche, da falls eine Maschine gibt, die  $L$  akzeptiert, dann gibt es unendlich viele.)

Falls  $w \in L$ , dann  $M_w$  akzeptiert  $w$  in  $c2^{2^{|w|}} < 2^{2^{|w|}}$  Schritten, also  $w \notin L$ .

Falls  $w \notin L$ , dann wegen der Definition von  $L$  lehnt  $M_w$   $w$  nicht ab in  $2^{2^{|w|}}$  Schritten, aber  $M_w$  terminiert schon vorher, also  $M_w$  muss  $w$  akzeptieren. Dann ist aber  $w \in L$ .

Das ist ein Widerspruch.  $\square$

Bemerkung: die Sprachenklassen  $P, FP, PSPACE, EXPTIME$  sind robust, das heisst, dass sie unabhängig vom Maschinenmodell sind.

## 8.2 Nicht-deterministische Turing-Maschinen (NTM)

**Definition 8.11.** Der Unterschied zwischen der TM und NTM ist nur in der Übergangfunktion  $\delta$ . Falls die NTM im Zustand  $q$  ist, und  $a$  liest, dann kann sie aus mehreren möglichen Fortsetzungen wählen:

$$\delta(q, a) \subseteq Q \times T \times \{\text{rechts, links, bleibt}\}$$

Die Maschine wählt immer eine mögliche Fortsetzung. Falls keine gibt, dann hält sie an.

**Definition 8.12.** Die NTM akzeptiert die Eingabe  $x$ , falls es mindestens einen Weg gibt, in dem die Maschine  $x$  akzeptiert.

Die mögliche Rechenwege können mit einem gerichteten Baum representiert werden.

**Definition 8.13.** Eine NTM ist  $t(n)$  zeit-begrenzt falls für eine Eingabe der Länge  $n$  alle(!) Wege kürzer als  $t(n)$  sind.

Leider kann heute niemand eine  $t(n)$  zeit-begrenzte NTM in  $\mathcal{O}(t(n))$  Zeit simulieren. Ebenso kann man die entsprechenden Sprachenklassen für die NTM definieren.

**Definition 8.14.**  $NTIME(t(n)) := \{\text{Die Sprachen akzeptierbar durch eine } \mathcal{O}(t(n)) \text{ zeit-begrenzten NTM}\}$

**Definition 8.15.**

$$\mathcal{NP} := \bigcup_{k \geq 1} NTIME(n^k)$$

**Satz 8.7.**  $P \subseteq \mathcal{NP}$   $\square$

Die wichtigste Frage in Rechentheorie:

$$P \stackrel{?}{=} \mathcal{NP}$$

**Satz 8.8.**  $P \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$

*Beweis.*  $P \subseteq \mathcal{NP} \implies \text{co}P \subseteq \text{co}\mathcal{NP}$ . Die Behauptung folgt dann aus  $P = \text{co}P$ . □

Die Frage taucht auf: wie kann man entscheiden, ob eine Sprache zu  $\mathcal{NP}$  gehört? Sei  $M$  eine deterministische TM mit 2 Bänder. Die Eingabe von  $M$  bestehe aus zwei Teilen  $x$  und  $y$ .  $x$  ist am Anfang auf dem ersten Band,  $y$  auf dem zweiten, der nur lesbar ist. Sei  $L_1$  die Sprache der  $(x, y)$  Paaren, die  $M$  akzeptiert.

**Definition 8.16.** Die durch  $M$  nicht-deterministisch akzeptierte Sprache  $L$  ist wie folgt definiert:

$$x \in L \iff \exists y : (x, y) \in L_1$$

$y$  ist ein entsprechender Hinweis (Zeuge), die garantiert, dass  $x \in L$  ist. Wichtig, dass nur die Existenz von  $y$  verlangt wurde, und nichts über die Methode, wie  $y$  gefunden werden soll.

**Satz 8.9 (Zeugensatz).** Für eine Sprache  $L \subseteq I^*$  sind folgende äquivalent:

1.  $L \in \mathcal{NP}$
2.  $\exists c > 0 \exists L_1 \in P : ((x, y) \in L_1 \rightarrow |y| \leq |x|^c) \wedge (x \in L \leftrightarrow \exists y : (x, y) \in L_1)$

*Beweisskizze. 1→2:* Da  $L \in \mathcal{NP}$ , existiert eine  $n^{c_1}$  zeit-begrenzte NTM  $N$ , die  $L$  akzeptiert. Sei  $d$  die maximale Anzahl von Bewegungsregeln für ein (Zustand, eingelesenes Symbol) Paar. Sei  $x \in L$ ,  $|x| = n$ .  $y$  wird so gewählt, dass  $y$  in jedem Schritt beschreibt, welche Regel aus den höchstmöglichen  $d$  gewählt werden soll, um  $x$  zu akzeptieren. Eine deterministische TM  $M$  bekommt sowohl  $x$  als auch  $y$ , und sie folgt laut  $y$  nur dem „guten“ Weg. Es gilt  $|y| \leq n^{c_1} \lceil \log_2(d+1) \rceil \leq n^c$  für eine passende Konstante  $c$ . Klar, dass falls  $x \in L$ , dann existiert ein solches  $y$ , sonst nicht.  $M$  erkennt  $L_1$  in Polynomzeit, also gilt auch, dass  $L_1 \in P$ .

*2→1:* Falls  $x \in L$  und  $|x| = n$ , dann existiert ein Hinweis  $y$  so, dass  $|y| \leq n^c$  und  $(x, y) \in L_1$ . Die maximale Anzahl solcher  $y$  Hinweise ist  $2^{n^c}$ . Es kann leicht gezeigt werden, dass so viele Möglichkeiten mit einer NTM  $N$  in  $n^c$  Schritten überprüft werden können. Also überprüft man in  $n^c$  Schritten nicht-deterministisch alle mögliche Zeugen  $y$ , und falls es ein solches  $y$  gibt, dass  $(x, y) \in L_1$  (was in Polynomzeit abgefragt werden kann), dann  $x \in L$ , also wird  $x$  akzeptiert, sonst nicht. □

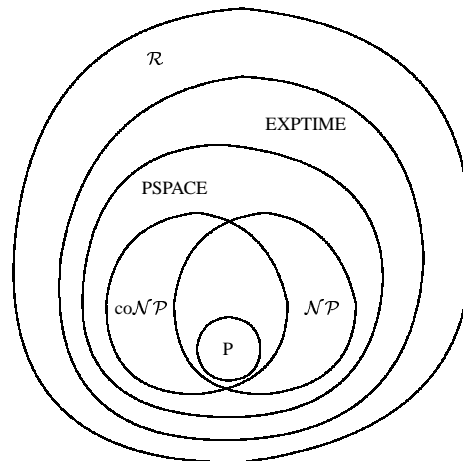
Anschaulich:  $\mathcal{NP}$  besteht aus solchen Entscheidungsproblemen, bei denen die „Ja“-Antwort mit Hilfe eines „Zeugen“ (also mit Hilfe zusätzlicher Information, die allerdings nicht sehr groß sein darf) effizient überprüft werden kann. Ähnliches gilt für  $\text{co}\mathcal{NP}$  im Zusammenhang mit der „Nein“-Antwort.

Der Zeuge soll nicht berechnet werden, sondern wird von einem „Orakel“ verraten.

**Folgerung 8.10.**  $\mathcal{NP} \subseteq PSPACE$

*Beweis.* Sei  $L \in \mathcal{NP}$ ,  $x$  die Eingabe,  $|x| = n$ . Man soll für alle mögliche Hinweise  $y$ , wo  $|y| \leq n^c$ , überprüfen, ob  $(x, y) \in L_1$ . Ein  $y$  Hinweis kann als eine 0-1 Folge der Länge  $n^c$  repräsentiert werden. Um das aktuelle  $y$  zu überprüfen, braucht man die vorherigen Ergebnisse nicht, also braucht man nur Platz für das aktuelle  $y$ . □

Die nächste Abbildung zeigt die Relation zwischen den gelernten Sprachenklassen.



## 8.3 Einige Sprachen aus $\mathcal{NP}$

### 8.3.1 Färbung mit 3 Farben

Ein Graph kann als eine 0-1 Folge kodiert werden. Sei 3-FARBE die Sprache, die die Codes solcher Graphen enthält, die mit 3 Farben färbbar sind.

**Behauptung 8.11.**  $3\text{-FARBE} \in \mathcal{NP}$

*Beweis.* Eine korrekte Färbung kann in  $2n$  Bits beschrieben werden. (Z. B. 00=gelb, 01=blau, 10=rot.) Ein guter Hinweis kann eine korrekte Färbung sein. Es kann in Polynomzeit überprüft werden, dass eine Färbung korrekt ist.  $\square$

### 8.3.2 Hamilton-Kreis

Sei H die Sprache solcher ungerichteten Graphen, die einen Hamiltonschen Kreis enthalten.

**Behauptung 8.12.**  $H \in \mathcal{NP}$

*Beweis.* Ein guter Hinweis kann die Beschreibung eines Hamilton-schen Kreises sein. Es kann in  $\mathcal{O}(n \log_2 n)$  Bits beschrieben werden, z. B. mit der Reihenfolge der Knotenpunkte. Es kann in Polynomzeit überprüft werden, ob das wirklich ein Hamiltonscher Kreis ist.  $\square$

Ebenso kann bewiesen werden, dass  $GH \in \mathcal{NP}$ , wo GH die Sprache solcher gerichteten Graphen bedeutet, die einen gerichteten Hamiltonschen Kreis enthalten.

### 8.3.3 Planare Graphen

Sei PLANAR die Sprache der planaren Graphen.

**Definition 8.17.** Eine Sprache  $L \in I^*$  heisst gut charakterisiert, falls  $L \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ .

**Behauptung 8.13.** PLANAR ist gut charakterisiert.

*Beweis.* Ein guter Hinweis kann die Koordinaten der Punkte in einer planaren Darstellung sein. (Laut dem Satz von Wagner und Fáry kann man annehmen, dass die Kanten in der planaren Darstellung gerade Linien sind.) Es kann in Polynomzeit überprüft werden, ob die Kanten einander schneiden oder nicht. Also  $\text{PLANAR} \in \mathcal{NP}$ .

Der Satz von Kuratowski zeigt, dass  $\text{PLANAR} \in \text{coNP}$ . □

### 8.3.4 Ein nicht-triviales Beispiel: Primzahlen

Sei  $\text{PRIM}$  die Sprache der Primzahlen (in binärer Darstellung). Klar, dass  $\text{PRIM} \in \text{coNP}$ , da ein Teiler beweisen kann, dass eine Zahl keine Primzahl ist.

**Satz 8.14 (V.R.Pratt, 1975).**  $\text{PRIM} \in \mathcal{NP}$

*Beweis.* Ohne Beweis benutzen wir die folgende Behauptung:  
Eine Zahl  $p$  ist Primzahl gdw.  $\exists 1 \leq g < p$ :

1.  $g^{p-1} \equiv 1 \pmod{p}$
2.  $g^{\frac{p-1}{r}} \not\equiv 1 \pmod{p}$  für jede Primzahl  $r$  wofür  $r|p-1$ .

Ausserdem benutzen wir, dass  $a^m \pmod{p}$  in Polynomzeit berechnet werden kann.

Ein Hinweis können die Zahlen  $g, r_1, \dots, r_k$  sein. Man muss überprüfen, dass  $r_i|p-1$  und die obige Kongruenzen gültig sind. Diese Tätigkeiten sind schnell. Weiterhin muss noch überprüft werden, dass  $r_i$  Primzahlen sind. Dafür benutzen wir die gleiche Methode, also der Hinweis in diesem Fall wird rekursiv definiert. Es kann bewiesen werden, dass dieser zusammengesetzte Hinweis  $\mathcal{O}(n^2)$  lang wird. □

**Bemerkung** Es war für eine lange Zeit eine wichtige offene Frage, ob  $\text{PRIM} \in \mathcal{P}$  gilt. Es wurden nur effiziente *randomisierte* Primtests entwickelt. 2002 wurde der erste polynomiale Algorithmus für dieses Problem von Agarwal, Kayal und Saxena veröffentlicht. In Praxis ist dieser Algorithmus jedoch sehr langsam, also werden weiterhin randomisierte Primtests verwendet.

### 8.3.5 Ein Beispiel für eine Funktion: Primfaktorzerlegung

Manchmal soll man nicht eine Sprache erkennen, sondern eine Funktion berechnen. Oft kann man das aber auf ein Erkennungsproblem zurückgeführt werden.

$$F := \{(a, c) \mid 1 < c \leq a \text{ sind ganze Zahlen, und } \exists b (1 < b \leq c) : b|a\}$$

**Satz 8.15.**  $F \in \mathcal{NP} \cap \text{coNP}$

*Beweis.* Ein guter Hinweis für  $(a, c) \in F$  ist ein  $b$ , wofür die Bedingungen gelten. Ein guter Hinweis für  $(a, c) \notin F$  kann die kanonische Form von  $a$  sein, plus die Beweise, dass die in der kanonischen Form vorkommenden Zahlen wirklich Primzahlen sind (siehe Satz von Pratt). Man soll noch überprüfen, ob  $c$  kleiner als jede Primzahl ist und  $a$  wirklich das Produkt dieser Zahlen ist. □

Das verbundene Problem ist die Primfaktorzerlegung. Wir zeigen, dass dieses Problem einfach auf das obige zurückgeführt werden kann:

**Satz 8.16.** Falls  $F \in \mathcal{P}$  dann  $\text{PRIMFAKTORZERLEGUNG} \in \mathcal{FP}$ .

*Beweis.* Falls  $(a, a - 1) \notin F$ , dann ist  $a$  eine Primzahl. Falls nicht, dann finden wir mit binärer Suche in  $\mathcal{O}(\log a)$  Zeit das minimale  $c$  so, dass  $(a, c) \in F$ . Dann ist  $c$  der minimale Primteiler von  $a$ . Die gleiche Methode wird dann für  $\frac{a}{c}$  verwendet. Da die Anzahl der Primteiler  $\mathcal{O}(\log a)$  ist, ist die Zurückführung polynomial.  $\square$

Bemerkung: Das ist eine offene Frage, und zwar eine sehr wichtige aus der Sicht der Kryptographie. Die beste bekannte Methode für die Primfaktorzerlegung benutzt  $c2^{\frac{n}{4}}$  Schritte. (D. Shanks)

## 8.4 Karp-Reduktion

Wann ist ein Problem  $L_1$  nicht bedeutend schwieriger als  $L_2$ ? Falls  $L_1$  mit Hilfe von  $L_2$  effizient lösbar ist.

**Definition 8.18.** Die Funktion  $f : I^* \rightarrow I^*$  ist eine Karp-Reduktion von  $L_1 \subseteq I^*$  auf  $L_2 \subseteq I^*$ , falls:

1.  $\forall x \in I^* : x \in L_1 \Leftrightarrow f(x) \in L_2$
2.  $f \in FP$

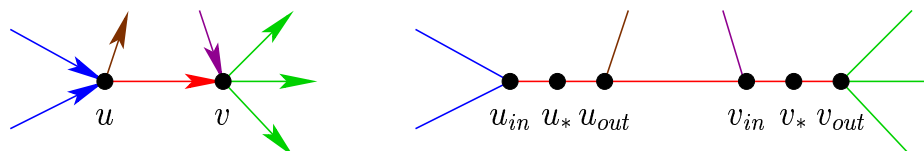
Bezeichnung:  $L_1 \prec L_2$ .

Falls man also einen Algorithmus für  $L_2$  hat, dann kann man entscheiden, ob  $x \in L_1$ . Man berechnet  $f(x)$  (schnell), und entscheidet, ob  $f(x) \in L_2$ .

Beispiel:

**Satz 8.17.**  $GH \prec H$

*Beweis.* Sei ein gerichteter Graph  $G(V, E)$  gegeben. Laut der Abbildung kann man in Polynomzeit einen ungerichteten Graphen  $G'(V', E')$  bilden.



Falls es einen gerichteten Hamiltonschen Kreis in  $G$  gibt, dann ist der entsprechende ungerichtete Kreis in  $G'$  ein Hamiltonscher Kreis in  $G'$ . Das heisst, dass aus  $G \in GH$   $G' \in H$  folgt.

Umgekehrt: falls es einen Hamiltonschen Kreis in  $G'$  gibt, dann besteht er aus solchen Teilen:  $u_{in} - u_* - u_{out} - v_{in} - v_* - v_{out}$ , also bilden die gerichteten Kanten  $u \rightarrow v$  einen gerichteten Hamiltonschen Kreis in  $G$ .  $\square$

**Behauptung 8.18.** Folgende Behauptungen gelten:

1.  $L_1 \prec L_2 \wedge L_2 \in \mathcal{P} \Rightarrow L_1 \in \mathcal{P}$
2.  $L_1 \prec L_2 \wedge L_2 \in \mathcal{NP} \Rightarrow L_1 \in \mathcal{NP}$
3.  $L_1 \prec L_2 \Rightarrow \overline{L_1} \prec \overline{L_2}$
4.  $L_1 \prec L_2 \wedge L_2 \in \text{coNP} \Rightarrow L_1 \in \text{coNP}$
5.  $L_1 \prec L_2 \wedge L_2 \in \mathcal{NP} \cap \text{coNP} \Rightarrow L_1 \in \mathcal{NP} \cap \text{coNP}$
6.  $L_1 \prec L_2 \wedge L_2 \prec L_3 \Rightarrow L_1 \prec L_3$   $\square$

Eine Karp-Reduktion zeigt eine Schwierigkeitsrelation zwischen Sprachen. In der  $\mathcal{NP}$  Sprachenklasse existieren „schwierigste“ Sprachen, wie bald gezeigt wird.

**Definition 8.19.** Die Sprache  $L \subseteq I^*$  heisst  $\mathcal{NP}$ -vollständig, falls

1.  $L \in \mathcal{NP}$
2.  $\forall L' \in \mathcal{NP} : L' \preceq L$

Die  $\mathcal{NP}$ -vollständigen Sprachen sind also mindestens so schwer, wie alle andere Sprachen in  $\mathcal{NP}$ . Falls man für eine solche Sprache beweisen könnte, dass sie in  $\mathcal{P}$  ist, dann wäre  $\mathcal{P} = \mathcal{NP}$ .

Die Frage ist, ob es überhaupt  $\mathcal{NP}$ -vollständige Sprachen gibt? Die Antwort ist ja:

## 8.5 Der Satz von Cook und Levin

Sei SAT die Sprache der erfüllbaren aussagenlogischen Formeln. (Booleschen Formeln).

**Satz 8.19 (Cook-Levin, 1971).** SAT ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Trivial, dass  $SAT \in \mathcal{NP}$ . Sei  $L$  eine beliebige Sprache aus  $\mathcal{NP}$ . Man braucht eine Karp-Reduktion von  $L$  auf SAT, also sollen wir angeben, wie man aus einer Eingabe  $x$  eine Formel  $\varphi$  konstruiert, die genau dann erfüllbar ist, wenn  $x \in L$ .

Da  $L \in \mathcal{NP}$ , existiert eine Sprache  $L_1 \in \mathcal{P}$  so, dass  $x \in L$  gdw.  $\exists y : |y| \leq |x|^d \wedge (x, y) \in L_1$ . Sei  $M$  eine TM mit einem Band, die  $L_1$  in  $n^c$  Schritten akzeptiert, wobei  $n = |x|$ . (Die Eingabe ist aber nicht nur  $x$ , sondern  $x\#y$ .) Nehmen wir an, dass das Alphapet aus 1, 0 und  $l$  besteht, wo  $l$  das leere Zeichen ist. Nehmen wir weiterhin an, dass  $M$  eine 1 auf den ersten Platz des Bandes schreibt, falls sie  $x$  akzeptiert, und sonst 0.

Führen wir folgende Boolesche Variablen ein:

$$\begin{aligned} 0x[i, j] &:= \begin{cases} 1 & \text{falls nach dem } i. \text{ Schritt die Zelle } j \text{ gleich 0 ist} \\ 0 & \text{sonst} \end{cases} \\ 1x[i, j] &:= \begin{cases} 1 & \text{falls nach dem } i. \text{ Schritt die Zelle } j \text{ gleich 1 ist} \\ 0 & \text{sonst} \end{cases} \\ leerx[i, j] &:= \begin{cases} 1 & \text{falls nach dem } i. \text{ Schritt die Zelle } j \text{ leer ist} \\ 0 & \text{sonst} \end{cases} \\ kopf[i, j] &:= \begin{cases} 1 & \text{falls nach dem } i. \text{ Schritt der Kopf auf Zelle } j \text{ steht} \\ 0 & \text{sonst} \end{cases} \\ q[i, s] &:= \begin{cases} 1 & \text{falls nach dem } i. \text{ Schritt der innere Zustand von } M \text{ gleich } s \text{ ist} \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Wir möchten  $M$  mit Hilfe dieser Variablen beschreiben. Folgende Behauptungen sollen mit aussagenlogischen Formeln ausgedrückt werden.

1. Auf einem Feld ist genau ein Symbol; der Kopf steht auf genau einem Feld; die Maschine besitzt immer genau einen inneren Zustand. Für jedes  $i, j$  ( $0 \leq i \leq n^c, 1 \leq j \leq n^c$ ) Paar

$$\begin{aligned} &(0x[i, j] \vee 1x[i, j] \vee leerx[i, j]) \wedge \\ &\wedge (\overline{0x[i, j]} \vee \overline{1x[i, j]}) \wedge (\overline{0x[i, j]} \vee \overline{leerx[i, j]}) \wedge (\overline{1x[i, j]} \vee \overline{leerx[i, j]}). \end{aligned}$$

Insgesamt  $(n^c + 1)n^c$  Formeln. Ähnlich können die anderen Eigenschaften formuliert werden.

2. Die  $\delta$  Funktion. Als Beispiel stehe hier die Regel  $\delta(q_s, 1) = (q_k, 0, \text{links})$ :

$$((q[i, s] \wedge 1x[i, j] \wedge \text{kopf}[i, j]) \longrightarrow (q[i + 1, k] \wedge 0x[i + 1, j] \wedge \text{kopf}[i + 1, j - 1]))$$

für jedes  $0 \leq i < n^c, 1 \leq j \leq n^c$ . (Bemerkung:  $x \rightarrow y \Leftrightarrow \bar{x} \vee y$ .)

Es soll noch ausgedrückt werden, dass die anderen Felder sich nicht ändern:

$$\overline{\text{kopf}[i, j]} \longrightarrow (0x[i, j] \leftrightarrow 0x[i + 1, j])$$

Ähnliche Formeln braucht man auch für die Symbole 1 und  $l$ .

3. Der Anfangszustand und die Eingabe:

$$q[0, 0] \wedge \text{kopf}[0, 1]$$

und z.B.

$$0x[0, 1] \wedge 1x[0, 2] \wedge 0x[0, 3] \wedge \dots$$

falls  $x = 010\dots$

Weiterhin braucht man die Bedingung der Akzeptanz auszudrücken:

$$1x[n^c, 1]$$

Die Konjunktion dieser Formeln ist ein gutes  $\varphi$ . Die Länge von  $\varphi$  ist  $\mathcal{O}(n^{2c})$ . Bemerkung: diejenigen Bits der Eingabe, die  $y$  entsprechen, sind einfach Variablen von  $\varphi$ .  $\square$

**Behauptung 8.20.** Falls  $L_1$   $\mathcal{NP}$ -vollständig ist,  $L_2 \in \mathcal{NP}$  und  $L_1 \prec L_2$ , dann ist  $L_2$  auch  $\mathcal{NP}$ -vollständig.

*Beweis.* Es folgt aus der Transitivität der  $\prec$  Relation.  $\square$

**Definition 8.20.** Eine Sprache  $L_2$  heisst  $\mathcal{NP}$ -schwer, falls  $L_1 \prec L_2$  gilt für eine  $\mathcal{NP}$ -vollständige Sprache  $L_1$ .

## 8.6 Weitere $\mathcal{NP}$ -vollständige Sprachen

### 8.6.1 Formeln in konjunktiver Normalform

**Definition 8.21.** Ein Literal ist entweder  $x_i$  oder  $\bar{x}_i$ .

**Definition 8.22.** Eine Boolesche Formel ist in konjunktiver Normalform (KNF), falls sie die Form

$$\varphi = \varphi_1 \wedge \dots \wedge \varphi_n$$

hat, wo  $\varphi_i = l_{i,1} \vee \dots \vee l_{i,t_i}$ , und  $l_{i,j}$  sind Literale.

Bemerkung: jede Formel hat eine KNF.

**Definition 8.23.** Eine Formel in KNF heisst  $k$ -KNF, falls  $\forall i : t_i \leq k$ .

Beispiel für eine Formel in 4-KNF:

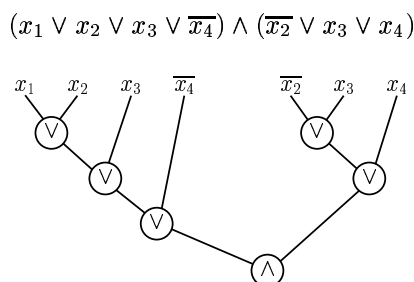
$$(\bar{x}_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_4 \vee \bar{x}_7 \vee \bar{x}_{18}) \wedge (x_9 \vee \bar{x}_{10} \vee \bar{x}_{13}).$$

Sei  $k$ -SAT die Sprache der erfüllbaren Formeln in  $k$ -KNF.

**Satz 8.21.**  $2\text{-SAT} \in P$ . □

**Satz 8.22.**  $3\text{-SAT}$  ist  $\mathcal{NP}$ -vollständig.

*Beweis.*  $\in \mathcal{NP}$  ist trivial. Wir zeigen eine  $\text{SAT} \leq 3\text{-SAT}$  Karp-Reduktion. Dazu braucht man zu einer beliebigen Formel  $\varphi$  eine äquivalente Formel  $\psi$  in 3-KNF konstruieren zu können. Zu der Formel  $\varphi$  generieren wir zuerst einen binären Baum wie folgt: z. B.



Zu jedem Punkt im Baum ordnen wir eine Boolesche Variable  $z_i$  zu. Die zu einem Blatt gehörende Variable entspricht der Variable im Blatt. Jene Variable  $z_i$ , die zu einem inneren Knotenpunkt mit Söhnen  $z_j$  und  $z_k$  zugeordnet ist, wird wie folgt entsprechend dem Symbol im Knotenpunkt definiert:

$$\begin{aligned} z_i \leftrightarrow (z_j \wedge z_k) &= (z_i \vee \overline{(z_j \wedge z_k)}) \wedge (\overline{z_i} \vee (z_j \wedge z_k)) = \\ &= (z_i \vee \overline{z_j} \vee \overline{z_k}) \wedge (\overline{z_i} \vee z_j) \wedge (\overline{z_i} \vee z_k) \\ \text{oder} \\ z_i \leftrightarrow (z_j \vee z_k) &= (z_i \vee \overline{(z_j \vee z_k)}) \wedge (\overline{z_i} \vee (z_j \vee z_k)) = \\ &= (z_i \vee \overline{z_j}) \wedge (z_i \vee \overline{z_k}) \wedge (\overline{z_i} \vee z_j \vee z_k) \end{aligned}$$

Diese Formel bezeichnen wir mit  $\psi_i$ . Die zusammengesetzte Formel

$$\psi := \left( \bigwedge \psi_i \right) \wedge z_m$$

wo  $z_m$  die Variable für die Wurzel ist, entspricht den Bedingungen. □

## 8.6.2 Färbung mit 3 Farben

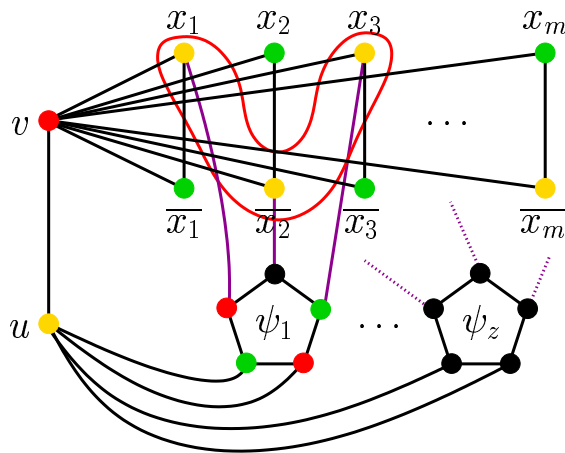
**Satz 8.23.**  $3\text{-FARBE}$  ist  $\mathcal{NP}$ -vollständig.

*Beweis.*  $3\text{-FARBE} \in \mathcal{NP}$  wurde schon gezeigt. Wir zeigen eine  $3\text{-SAT} \leq 3\text{-FARBE}$  Karp-Reduktion. Sei  $\psi$  eine Eingabe für 3-SAT, also eine Formel der Form:

$$\psi = \psi_1 \wedge \dots \wedge \psi_z$$

wo  $\psi_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ . ( $l_{i,j}$  sind Literale aus den Variablen  $x_1, \dots, x_m$ .)

Wir konstruieren dazu einen Graphen wie folgt (in diesem Beispiel ist  $\psi_1 = x_1 \vee \overline{x_2} \vee x_3$ ):



Sei die Farbe von  $v$  rot, die Farbe von  $u$  gelb.  $x_i$  und  $\bar{x}_i$  haben verschiedene Farben, die eine gelb, die andere grün. Die grünen Punkte werden als *true* interpretiert, die gelben als *false*. Es soll bewiesen werden, dass die Formel genau dann erfüllbar ist, wenn der Graph mit 3 Farben färbbar ist.

Falls der Graph mit 3 Farben färbbar ist, dann ist in jedem  $\psi_i$  ein Literal mit Farbe grün (also ist die Formel erfüllbar), sonst wären alle Punkte des dazugehörigen Fünfecks mit einem gelben Punkt benachbart, also sollte das Fünfeck mit 2 Farben (rot und grün) gefärbt werden, was unmöglich ist.

Umgekehrt, falls die Formel erfüllbar ist, dann soll man eine gute Färbung zeigen. Eigentlich muss man nur die Färbung der Fünfecke angeben, denn die Farbe der anderen Punkte haben wir schon definiert. Die unteren zwei Punkte dürfen nicht gelb sein, die oberen drei Punkte dürfen entweder nicht gelb oder nicht grün sein, aber die verbotene Farbe ist nicht das gleiche für jeden Punkt. Es ist leicht zu sehen, dass ein Fünfeck so färbbar ist. □

**Satz 8.24.** 2-FARBE  $\in \mathcal{P}$  □

### 8.6.3 Maximale unabhängige Punktmenge

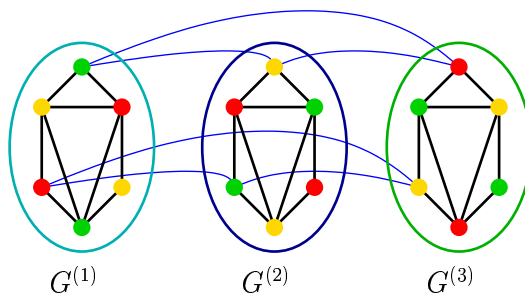
**Definition 8.24.** Eine Teilmenge der Punkte eines Graphen heisst unabhängig, falls keine Kante zwischen zwei Punkten dieser Menge führt.

Sei MAX-UNABHÄNGIG die folgende Sprache:

$$\text{MAX-UNABHÄNGIG} := \left\{ (G, k) \mid \begin{array}{l} G \text{ ist ein Graph, } k \in \mathbb{Z}^+ \text{ und} \\ G \text{ hat eine } k\text{-elementige unabhängige Teilmenge} \end{array} \right\}.$$

**Satz 8.25.** MAX-UNABHÄNGIG ist  $\mathcal{NP}$ -vollständig.

*Beweis.* MAX-UNABHÄNGIG  $\in \mathcal{NP}$  ist trivial. Wir zeigen eine 3-FARBE  $\leq$  MAX-UNABHÄNGIG Karp-Reduktion. Sei der zu färbende Graph  $G$ . Konstruieren wir daraus einen neuen Graphen  $G_1$ , wie in der Abbildung. Die Abbildung zeigt auch, wie eine 3-Färbung von  $G$  zu einer 3-Färbung von  $G_1$  ergänzt werden kann.



Sei  $k = |V(G)|$ . Wir behaupten, dass  $G$  genau dann 3-färbbar ist, falls  $(G_1, k) \in \text{MAX-UNABHÄNGIG}$ .

Falls  $G$  mit 3 Farben färbbar ist, dann zeigen wir eine unabhängige Menge  $H$  der Größe  $k$  in  $G_1$ . Bestehe  $H$  aus den roten Punkten. Klar, dass diese Menge unabhängig ist. Da  $v^{(1)}, v^{(2)}, v^{(3)}$  verschiedene Farben haben, ist genau eins von ihnen in  $H$ , also ist die Größe von  $H$  eben  $k$ .

Umgekehrt, falls es eine unabhängige Menge  $H$  der Größe  $k$  in  $G_1$  gibt, dann zeigen wir eine gute Färbung von  $G$ . Sei die Farbe von  $v \in V(G)$  rot, falls  $v^{(1)} \in H$ , grün falls  $v^{(2)} \in H$  und gelb sonst. So haben wir die Farbe aller Punkte in  $G$  eindeutig definiert. Da  $H$  unabhängig ist, führen zwischen Punkten in  $G^{(i)} \cap H$  ( $i = 1, 2, 3$ ) keine Kanten, also ist die Färbung korrekt.  $\square$

**Folgerung 8.26.** Die Sprache MAX-KLIKK ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Betrachte statt  $G$  sein Komplement  $\bar{G}$ .  $\square$

### 8.6.4 3-dimensionale Paarung

**Definition 8.25.** Seien  $U_1, U_2, U_3$  drei Mengen,  $|U_i| = t$ . Sei  $S \subseteq U_1 \times U_2 \times U_3$  gegeben. Die Sprache 3DM enthält solche  $U_1, U_2, U_3, S$  Mengen, aus denen eine  $S' \subseteq S, |S'| = t$  ausgewählt werden kann so, dass jeder Punkt durch  $S'$  bedeckt wird.

**Satz 8.27.** 3DM ist  $\mathcal{NP}$ -vollständig.  $\square$

**Definition 8.26.** Sei  $U$  eine endliche Menge,  $\mathcal{F} = \{X_1, \dots, X_k\}$ , wo  $X_i$  3-elementige Teilmengen von  $U$  sind. Sei X3C die Sprache solcher  $(U, \mathcal{F})$  Paare, wo aus  $\mathcal{F}$  disjunkte Mengen ausgewählt werden können, die zusammen  $U$  bedecken.

**Satz 8.28.** X3C ist  $\mathcal{NP}$ -vollständig.  $\square$

### 8.6.5 Travelling Salesman Problem

**Satz 8.29.** H und GH sind  $\mathcal{NP}$ -vollständig.  $\square$

Eine Verallgemeinerung dieser Probleme ist das folgende:

**Definition 8.27.**

$$\text{TSP} := \left\{ (G, k) \mid \begin{array}{l} G \text{ ist ein gewichteter Graph, } k \in \mathbb{Z}^+ \text{ und} \\ G \text{ hat einen Hamiltonschen Kreis mit maximal } k \text{ Gesamtkost} \end{array} \right\}.$$

**Satz 8.30.** TSP ist  $\mathcal{NP}$ -vollständig.  $\square$

### 8.6.6 Das Rucksack Problem und verwandte Probleme

**Definition 8.28.**

$$\text{RSP} := \left\{ (s_1, \dots, s_m; v_1, \dots, v_m; b; k) \mid \begin{array}{l} s_i, b, k \in \mathbb{Z}^+ \text{ und es gibt } I \subseteq \{1, \dots, m\} \\ \text{wofür } \sum_{i \in I} s_i \leq b \text{ und } \sum_{i \in I} v_i \geq k \end{array} \right\}.$$

**Satz 8.31.** RSP ist  $\mathcal{NP}$ -vollständig.  $\square$

**Definition 8.29.** Ein Spezialfall des obigen Problems ist das Teilmengen-Summe Problem.

$$\text{TMSP} := \left\{ (s_1, \dots, s_m; b) \mid \begin{array}{l} s_i, b \in \mathbb{Z}^+ \text{ und es gibt } I \subseteq \{1, \dots, m\} \\ \text{wofür } \sum_{i \in I} s_i = b \end{array} \right\}.$$

**Satz 8.32.** TMSP ist  $\mathcal{NP}$ -vollständig. □

**Definition 8.30.** Ein weiterer Spezialfall ist, falls  $b = \frac{\sum s_i}{2}$ , also soll man zwei solche Teilmengen finden, deren Summe gleich ist.

$$\text{PARTITION} := \left\{ (s_1, \dots, s_m) \left| \begin{array}{l} s_i \in \mathbb{Z}^+ \text{ und es gibt } I \subseteq \{1, \dots, m\} \\ \text{wofür } \sum_{i \in I} s_i = \frac{1}{2} \sum_{i=1}^m s_i \end{array} \right. \right\}.$$

**Satz 8.33.** PARTITION ist  $\mathcal{NP}$ -vollständig. □

**Definition 8.31.** Ein weiteres berühmtes Problem ist das BIN-PACKING:

$$\text{BIN-PACKING} := \left\{ (s_1, \dots, s_m; k) \left| \begin{array}{l} 0 < s_i \leq 1, s_i \in \mathbb{Q}, k \in \mathbb{Z}^+ \text{ und } m \text{ Gegenstände mit} \\ \text{Volumen } s_1, \dots, s_m \text{ passen in } k \text{ Kisten mit Volumen } 1 \text{ ein} \end{array} \right. \right\}.$$

**Satz 8.34.** BIN-PACKING ist  $\mathcal{NP}$ -vollständig. □

## 8.7 $\mathcal{NPI}$

Sei  $\mathcal{NPC}$  die Menge der  $\mathcal{NP}$ -vollständigen Sprachen,  $\mathcal{NPI} = \mathcal{NP} \setminus (\mathcal{P} \cup \mathcal{NPC})$ .

**Satz 8.35 (Ladner, 1975).** Falls  $\mathcal{P} \neq \mathcal{NP}$ , dann  $\mathcal{NPI} \neq \emptyset$ . □

## Kapitel 9

# Allgemeine Methoden für Planung von Algorithmen

Was kann man machen, falls das Problem  $\mathcal{NP}$ -vollständig ist?

Oft kann man eine Methode in  $\mathcal{O}(c^n)$  finden, aber der Wert von  $c$  ist sehr kritisch. In den folgenden Abschnitten zeigen wir solche Methoden, die besser funktionieren, als alle Möglichkeiten durchzusehen.

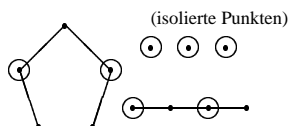
### 9.1 Branch and bound

Dieses Verfahren ist benutzt, falls das Problem auf das Durchsuchen eines riesigen Bäumchen reduzierbar ist. Dieser Baum ist meistens so gross, dass er gar nicht auf einmal gespeichert werden kann. Einige Regeln beschreiben aber, wie man die Kinder eines Knotenpunktes generieren kann. Ein Beispiel kann das Schachspiel sein. Die Knoten sind Stellungen, eine Kante bedeutet, dass eine Stellung aus einer anderen in einem Schritt erreichbar ist. Um den ganzen Baum zu besuchen hat man keine Zeit, deswegen lohnt es sich heuristische Beschränkungen zu wählen. In diesem Fall kann es z. B. folgende sein: falls ein Schritt das Verlieren der Königin verursacht, dann wird dieser Teil des Bäumchen nicht besucht.

#### Beispiel 1 MAX-UNABHÄNGIG

Die Aufgabe ist  $\mathcal{NP}$ -vollständig. Alle Möglichkeiten:  $2^n$ .

**Bemerkung 9.1.** Falls in  $G$  der Grad aller Punkten höchstens 2 ist, kann die Aufgabe in Polynomzeit gelöst werden. Der Graph ist nämlich die Vereinigung von disjunkten Kreisen und Wegen und isolierten Punkten.



Mit Hilfe dieses Algorithmus kann  $MU()$  konstruiert werden.  $MU()$  bekommt einen Graphen als Eingabe und gibt die maximale unabhängige Menge aus.

$MU(G)$

1. Falls in  $G$  alle Gradzahlen  $\leq 2$  sind, dann  $MU(G)$  gibt das obige Ergebnis aus und STOP.
2. Sei  $x \in G$ ,  $Grad(x) \geq 3$ .

- $$S_1 := \text{MU}(G \setminus \{x\})$$
- $$S_2 := \{x\} \cup \text{MU}(G \setminus \{x \text{ und seine Nachbarn}\}).$$
3. Sei  $S$  die nicht kleinere Menge aus  $S_1$  und  $S_2$ .
  4.  $\text{MU}(G) := S$ .

Die Methode zu verstehen soll man folgende zur Kenntnis nehmen: die maximale unabhängige Menge entweder enthält  $x$  und dann seine Nachbarn nicht, oder enthält  $x$  nicht.

Die Kost des Verfahrens  $T(n)$  kann triviell so geschätzt werden:

$$T(n) \leq T(n-1) + T(n-4) + 1$$

Daraus folgt, dass  $T(n) \leq c\gamma^n$ , wo  $\gamma \approx 1,381$ .

**Beispiel 2** Die einfachste Methode für 3-FARBE benutzt  $\mathcal{O}(3^n)$  Schritten.

**Bemerkung 9.2.** Falls die Punkten der roten Farbe gegeben sind, dann kann man in Polynomzeit kontrollieren ob es zu einer korrekten Färbung ergänzt werden kann. Es soll nämlich kontrolliert werden, ob die roten Punkten eine unabhängige Menge bilden und ob der Restgraph mit 2 Farben färbbar ist.

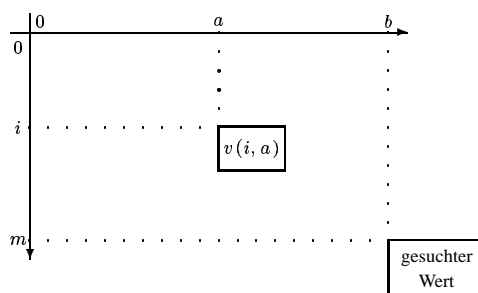
Mit dieser Beschränkung kann man die Aufgabe in  $\mathcal{O}(2^{cn^c})$  Zeit lösen. ( $c$  ist eine Konstante.)

## 9.2 Dynamische Programmierung

Man versucht das Optimum mit Hilfe des Optimums von kleineren Aufgaben zu bilden. Es ist ähnlich zum Ausfüllen einer Tabelle, wo die Werte früherer (optimalen) Elementen den Wert des aktuellen Elements bestimmt.

Schon gesehen im Pascal-schen Dreieck, Algorithmus von Ford, Algorithmus von Floyd.

**RUCKSACK Problem** Sei  $v(i, a)$  der maximal erreichbare Wert mit den  $s_1, \dots, s_i$  Gewichten und mit den zugehörigen  $v_1, \dots, v_i$  Werten, falls die Kapazität  $a$  ist. Klar, dass  $v(0, a) = v(i, 0) = 0$  für beliebige  $a$  und  $i$ . Die Aufgabe ist  $v(m, b)$  zu bestimmen. Genauer: zu entscheiden, ob das Element auf der Position  $(m, b)$  in der  $[v(i, a)]$  Tabelle grösser als  $k$  ist.



Folgende Rekursion hilft die Tabelle auszufüllen:

$$v(i, a) = \max\{v(i-1, a); v_i + v(i-1, a-s_i)\}$$

Der erste Wert in der Maximumbildung bedeutet den optimalen Wert, falls der  $i$ -te Gegenstand nicht im Optimum vorkommt, der zweite, falls er vorkommt.

Die Kost der Methode ist  $\mathcal{O}(bL)$ , wo  $L$  die Länge der Eingabe ist. Es ist nicht in Polynomzeit, da die Länge der Eingabe

$$L = \sum_{i=1}^m (\lceil \log_2(s_i + 1) \rceil + \lceil \log_2(v_i + 1) \rceil) + \lceil \log_2(k + 1) \rceil + \lceil \log_2(b + 1) \rceil$$

ist, so  $b$  kann exponentiell gross im Vergleich zu  $L$  sein. (Falls  $b$  nicht sehr gross ist, kann die Methode in Polynomzeit beendet werden.)

### 9.3 Näherungsmethoden

Es kann vorkommen, dass statt das absolute Optimum eine fast optimale Lösung auch reicht.

#### BIN-PACKING

Die **first-fit (FF)** Methode setzt den nächsten Gegenstand in die erste Kiste, die genug gross ist. Sei  $FF(I)$  die Anzahl der benutzten Kisten mit FF mit der Eingabe  $I$ . Sei  $OPT(I)$  der optimale Wert.

**Behauptung 9.3.**  $FF(I) \leq 2 \cdot OPT(I)$

*Beweis.* Klar, dass  $\lceil \sum_{i=1}^m s_i \rceil \leq OPT(I)$ . Weiterhin gilt, dass  $FF(I) \leq \lceil 2 \sum_{i=1}^m s_i \rceil$ , da bei FF höchstens eine solche Kiste vorkommt, die nicht bis ihrer Hälfte gefüllt ist.

$$FF(I) \leq \lceil 2 \sum_{i=1}^m s_i \rceil \leq 2 \lceil \sum_{i=1}^m s_i \rceil \leq 2 \cdot OPT(I)$$

□

**Satz 9.4 (D. S. Johnson et. al., 1976.).** Für eine beliebige Eingabe  $I$  gilt, dass  $FF(I) \leq \lfloor \frac{17}{10} OPT(I) \rfloor$ ; weiterhin es gibt unendlich viele solche Eingaben, wofür  $FF(I) \geq \frac{17}{10}(OPT(I) - 1)$ . □

Die **first-fit decreasing** Methode sortiert zuerst die Gegenstände in nicht steigende Reihenfolge nach ihrem Gewicht. Dann benutzt die FF Methode.

**Satz 9.5 (D. S. Johnson, 1973.).** Für eine beliebige Eingabe  $I$  gilt, dass  $FFD(I) \leq \frac{11}{9} OPT(I) + 4$ ; weiterhin es gibt unendlich viele solche Eingaben, wofür  $FFD(I) \geq \frac{11}{9} OPT(I)$ . □

**Satz 9.6 (F. de la Vega, G. S. Lueker.).** Für eine beliebige  $\varepsilon$  existiert ein linearer Algorithmus  $P$  so, dass  $P(I) \leq (1 + \varepsilon) OPT(I)$ . □

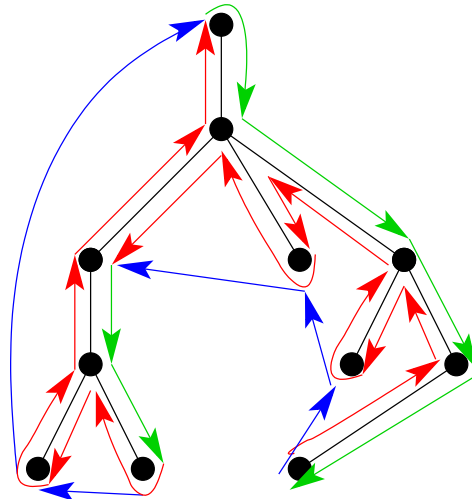
#### Euklidische TSP

Es ist ein Variant des TSP. Es gibt eine Funktion  $d$ , die zu jeder Kante von  $K_n$  eine nicht-negative Zahl ordnet. Weiterhin gilt, dass

$$d(u, w) \leq d(u, v) + d(v, w)$$

für beliebige Knoten  $u, v, w$ .

Sei  $T$  ein MSB im Graphen. Sei die Kost von  $T$  gleich  $s$ . Ein Euler-scher Kreis in  $T$  besucht alle Knoten, sogar mehrmals. Die Summe dieses Kreises ist  $2s$ . Man kann einen  $F$  Hamilton-schen Kreis konstruieren, falls man die überflüssigen Teile abschneidet, und damit wird der Kreis noch billiger.



Also die Kost von  $F$  ist höchstens  $2s$ . Falls man eine Kanten aus einem Hamilton-schen Kreis entfernt, der übrigbleibende Graph ist ein spannender Baum, also ist sein Gewicht minimum  $s$ . So folgt, dass ein Hamilton-scher Kreis minimum  $s$  wiegt. Deswegen kann  $F$  im schlechtesten Fall zweimal der Wert des Optimum sein.

# Literaturverzeichnis

- [1] L. Rónyai, G. Ivanyos, and R. Szabó. *Algoritmusok*. Typotex, Budapest, 1999.