

Vorlesungsmitschrift

# Höhere Algorithmik

gelesen von Prof. Dr. Günter Rote

Tobias Höppner

Wintersemester 2014/2015

# Inhaltsverzeichnis

<b>1 Einführung</b> (Vorlesung 1 am 17.10.)	<b>1</b>
1.1 Organisatorisches . . . . .	1
1.2 Kuchen teilen . . . . .	1
1.2.1 1. Algorithmus (für 2 Personen) . . . . .	1
1.2.2 2. Algorithmus (für 3 Personen) . . . . .	1
1.2.3 3. Teilen und Trimmen . . . . .	2
1.2.4 4. Teilen mit bewegtem Messer . . . . .	2
1.2.5 5. Simuliertes bewegtes Messer . . . . .	2
1.2.6 6. Simuliertes Messer + Zufall . . . . .	2
1.2.7 7. Divide & Conquer . . . . .	3
1.2.8 8. Divide & Conquer + Zufall . . . . .	3
<b>2 Einführung Teil 2</b> (Vorlesung 2 am 20.10.)	<b>4</b>
2.1 Ziele der Vorlesung . . . . .	4
2.2 Rechnermodelle . . . . .	4
2.2.1 Turing-Maschine . . . . .	4
2.2.2 Registermaschine (RAM - random access machine) . . . . .	4
2.2.3 Berechnung der Laufzeit . . . . .	5
2.3 Laufzeit eines Algorithmus . . . . .	6
<b>3 Rechnermodelle (Fortsetzung)</b> (Vorlesung 3 am 24.10.)	<b>7</b>
3.1 Warum nicht die Turingmaschine? . . . . .	7
3.2 Elementare Operationen . . . . .	7
3.3 Teile und Herrsche . . . . .	8
3.3.1 Beispiel A: Quicksort . . . . .	8
3.3.2 Beispiel B: Mergesort (Sortieren durch Verschmelzen) . . . . .	8
3.3.3 Analysemöglichkeiten . . . . .	8
<b>4 Rekursion (Fortsetzung)</b> (Vorlesung 5 am 31.10.)	<b>19</b>
4.1 Motivation Master-Theorem . . . . .	19
4.2 Master-Theorem für divide and conquer-Rekursion . . . . .	19
4.2.1 Bemerkungen . . . . .	20
4.3 Beweis: Master-Theorem . . . . .	20
<b>5 Master Theorem (Fortsetzung)</b> (Vorlesung 6 am 3.11.)	<b>22</b>
5.1 Beweis Fortsetzung . . . . .	22
5.2 Zählen von Fehlständen (Inversion) . . . . .	23
5.2.1 Divide and Conquer - oder - Warum Mergesort so wichtig ist! . . . . .	23
5.2.2 Variante . . . . .	23
5.2.3 Laufzeit . . . . .	24

<b>6</b>	<b>Median</b> (Vorlesung 7 am 7.11.)	<b>25</b>
6.1	Bestimmung des k-kleinsten Elements (Medians) . . . . .	25
6.2	Quickselect . . . . .	25
6.2.1	Laufzeit . . . . .	25
6.3	randomisiertes Quickselect . . . . .	26
6.3.1	Laufzeit . . . . .	26
6.4	Quickselect nach Blum, Floyd, Pratt, Rivest, Tarjan (1973) . . . . .	26
6.4.1	Laufzeit . . . . .	27
<b>7</b>	<b>Das Rucksackproblem</b> (Vorlesung 8 am 10.11.)	<b>29</b>
7.1	Lösung: Dynamisches Programmierung / Optimierung . . . . .	29
7.1.1	Laufzeit und Speicherbedarf . . . . .	30
7.2	Dynamische Programmierung . . . . .	30
7.3	Die Tabelle als Netzwerk . . . . .	31
7.4	Speicheroptimierung . . . . .	31
<b>8</b>	<b>Dynamische Programmierung (Fortsetzung)</b> (Vorlesung 9 am 14.11.)	<b>32</b>
8.1	Gewichtete Intervallauswahl . . . . .	32
8.1.1	Vorverarbeitung (Normalisierung) . . . . .	32
8.1.2	Laufzeit . . . . .	32
8.1.3	Algorithmus . . . . .	32
8.2	Rundreiseproblem (Traveling Salesperson Problem[TSP]) . . . . .	32
8.2.1	Rekursion . . . . .	33
8.2.2	1. Möglichkeit . . . . .	33
<b>9</b>	<b>Dynamische Programmierung (Fortsetzung)</b> (Vorlesung 10 am 17.11.)	<b>34</b>
9.1	Optimale Triangulierung eines konvexen Polygons . . . . .	34
9.1.1	Triangulierung . . . . .	34
9.1.2	Laufzeit . . . . .	34
9.2	Isotone Regression . . . . .	34
9.2.1	Teilprobleme . . . . .	34
9.2.2	Optimallösung . . . . .	35
<b>10</b>	<b>Dynamische Programmierung (Fortsetzung)</b> (Vorlesung 11 am 21.11.)	<b>36</b>
10.1	Algorithmus . . . . .	36
<b>11</b>	<b>Der optimale Suchbaum</b>	<b>37</b>
11.1	Teilprobleme . . . . .	37
11.2	Rekursion . . . . .	37
11.3	Anfangswerte . . . . .	37
11.4	Gesamtlösung . . . . .	38
11.5	Laufzeit . . . . .	38

<b>12 Dynamische Programmierung (Einen hab ich noch!)</b>	<b>39</b>
12.1 Dynamische Programmierung - eine Zusammenfassung . . . . .	39
12.2 Editierabstand . . . . .	39
12.2.1 Problem . . . . .	39
12.2.2 Teilprobleme . . . . .	39
12.2.3 Rekursion . . . . .	39
12.2.4 Startwerte . . . . .	40
12.2.5 Graph . . . . .	40
12.2.6 Annahmen . . . . .	40
12.2.7 Speicherreduktion . . . . .	40
12.3 Laufzeit (inkl. Speicherreduktion) . . . . .	40
<b>13 Gierige Algorithmen</b>	<b>41</b>
13.1 Beispiel Rucksack . . . . .	41
13.2 Ungewichtete Intervallauswahl . . . . .	41
<b>14 Greedy Algorithmen</b> (Vorlesung 13 am 28.11.)	<b>42</b>
14.1 Intervallauswahl nach Endzeitpunkten . . . . .	42
14.1.1 Beweis . . . . .	42
14.2 Variante . . . . .	42
14.3 Greedy Algorithmus . . . . .	43
14.3.1 Beweis . . . . .	43
14.4 Interpretation als Graphenproblem . . . . .	43
14.5 Zeitplanung(Scheduling) . . . . .	43
14.5.1 Beweis . . . . .	44
14.6 EDD-rule (earliest due date rule . . . . .	44
<b>15 Der Klassiker für Greedy Algorithmen: minimal SPT</b>	<b>45</b>
15.1 Algorithmus von Kruskal . . . . .	45
<b>16 Kürzeste Wege</b> (Vorlesung 14 am 01.12.)	<b>46</b>
16.1 Wiederholung: Dijkstras Methode . . . . .	46
16.1.1 Algorithmus nach Dijkstra (1960-1961) . . . . .	46
16.2 Algorithmus von Bellman / Ford . . . . .	46
16.2.1 Rekursion . . . . .	46
16.2.2 Anfangswerte . . . . .	46
16.2.3 Beispiel . . . . .	46
16.2.4 Bemerkungen . . . . .	47
16.2.5 Implementierung der Rekursion . . . . .	47
16.2.6 Verbesserungsmöglichkeit . . . . .	47
16.2.7 Rekursion neu . . . . .	47
16.2.8 Verbesserter Algorithmus . . . . .	48
16.3 Algorithmus von Bellman, Ford, Moore . . . . .	48

<b>17 Das algebraische Wegproblem</b> (Vorlesung 15 am 05.12.)	<b>49</b>
17.1 Beispiel: Der sichereste Weg. . . . .	49
17.2 Algebraisches Wegeproblem . . . . .	49
<b>18 Halbring</b>	<b>49</b>
<b>19 Bellman-Ford-Algorithmus</b>	<b>51</b>
19.1 Die k-kürzesten Wege (k=2) . . . . .	52
<b>20 Fixed Parameter Tractability</b> (Vorlesung 16 am 08.12.)	<b>53</b>
20.1 Strategien . . . . .	53
20.2 Definition FPT . . . . .	53
20.3 Beispiel: Unabhängige Knotenmenge auf planaren Graphen . . . . .	53
20.3.1 Suchbaum . . . . .	53
20.3.2 Algorithmus . . . . .	53
20.3.3 Laufzeit . . . . .	54
20.3.4 Datenreduktion / Problemkern (Kernelization) . . . . .	54
20.3.5 Lemma . . . . .	55
<b>21 Fixed Parameter Tractability (Fortsetzung)</b> (Vorlesung 17 am 08.12.)	<b>56</b>
21.1 Algorithmus mit DP . . . . .	56
21.1.1 Teilprobleme . . . . .	57
21.1.2 Anker . . . . .	57
21.1.3 Rekursion . . . . .	57
21.1.4 Lösungen . . . . .	57
<b>22 Kürzeste Wege (Fortsetzung)</b> (Vorlesung 18 am 15.12.)	<b>58</b>
22.1 Berechnen aller Paare von kürzesten Wegen . . . . .	58
22.1.1 Verbesserung . . . . .	58
22.1.2 Algorithmus . . . . .	58
22.1.3 Laufzeit . . . . .	58
22.2 Floyd-Warshall Algorithmus zur Berechnung aller kürzesten Wege . . . . .	58
22.2.1 Teilprobleme . . . . .	58
22.2.2 Lösungen . . . . .	59
22.2.3 Rekursion . . . . .	59
22.2.4 Beispiele für $a^*$ . . . . .	59
22.2.5 Algorithmus . . . . .	59
22.3 Halbring der formalen Sprachen . . . . .	60
<b>23 Matrizenmultiplikation und kürzeste Wege</b> (Vorlesung 19 am 19.12.)	<b>61</b>
23.0.1 Algorithmus . . . . .	61

<b>24</b>	<b>Komplexitätsklassen P und NP</b> (Vorlesung 20 am 05.01.)	<b>63</b>
24.1	P . . . . .	63
24.1.1	Problem: Hamiltonscher Kreis . . . . .	63
24.1.2	Problem: Rundreise . . . . .	63
24.1.3	Reduzierbarkeit . . . . .	63
24.1.4	Beispiel . . . . .	64
24.2	NP . . . . .	64
24.2.1	Beispiel: Rundreiseproblem $\in NP$ . . . . .	64
<b>25</b>	<b>Alternative NP-Definition</b> (Vorlesung 21 am 09.01.)	<b>65</b>
25.1	Polynomiell verifizierbares Zertifikatskriterium . . . . .	65
25.1.1	Beispiel: Rundreiseproblem . . . . .	65
25.1.2	Beweis . . . . .	65
25.2	Komplexitätsklassen . . . . .	65
25.2.1	NP-vollständig . . . . .	66
25.3	SAT - satisfiability = Erfüllbarkeit . . . . .	66
25.3.1	Beispiel KNF . . . . .	66
25.3.2	SATZ: Cook-Levin, 1971 . . . . .	66
<b>26</b>	<b>Reduktionen</b> (Vorlesung 22 am 12.01.)	<b>68</b>
26.1	Unabhängige Menge (von Knoten) . . . . .	68
26.1.1	Beweis . . . . .	68
26.1.2	Anmerkungen: Wie kommt man drauf? . . . . .	68
26.2	Hamiltonkreis in gerichtetem Graph . . . . .	68
26.2.1	Variablen Gadgets (Vorrichtungen) . . . . .	69
26.2.2	Klausel Gadgets . . . . .	69
26.2.3	Beweis . . . . .	69
26.3	3-SAT . . . . .	69
<b>27</b>	<b>Weitere Reduktionen</b> (Vorlesung 23 am 16.01.)	<b>70</b>
27.1	Graphenfärbung . . . . .	70
27.2	3-Färbbarkeit . . . . .	70
27.2.1	Vorrichtungen (Gadgets) . . . . .	70
27.2.2	Reduktion . . . . .	70
27.2.3	Beweis . . . . .	70
27.3	Reduktion von gewöhnlichen 3-FARB auf 3-FARB in planaren Graphen . . . . .	71
27.4	Teilmengensumme (TMS) . . . . .	71
27.4.1	Beweis . . . . .	71
27.5	Rucksackproblem . . . . .	72
<b>28</b>	<b>Komplexitätstheorie</b> (Vorlesung 24 am 19.01.)	<b>73</b>
28.1	Stark-NP-vollständig . . . . .	73
28.2	Rundreiseproblem ist stark-NP-vollständig . . . . .	73
28.3	coNP . . . . .	73

28.4	PSPACE . . . . .	73
28.5	Turing-Reduzierbarkeit . . . . .	73
<b>29</b>	<b>Approximationsalgorithmen</b>	<b>73</b>
29.1	amortisierte Analyse . . . . .	73
29.1.1	Bsp: Fibonacci-Heap . . . . .	74
29.2	Binomialbäume . . . . .	74
29.2.1	merge . . . . .	75
29.2.2	insert . . . . .	75
29.2.3	deletemin . . . . .	75
<b>30</b>	<b>Fibonacci-Halden</b> (Vorlesung 25 am 23.01.)	<b>76</b>
30.1	decreasekey(x) . . . . .	76
30.2	merge . . . . .	76
30.3	deletemin . . . . .	76
30.3.1	mögliche Implementierung . . . . .	76
30.4	Invariante . . . . .	77
30.4.1	Beweis . . . . .	77
30.5	Kleinstmögliche F-Bäume mit Grad k . . . . .	77
30.6	Amortisierte Analyse . . . . .	77
30.6.1	Laufzeit . . . . .	77
30.6.2	Analyse für decreasekey . . . . .	77
30.6.3	Satz: Amortisierte Laufzeit . . . . .	78
30.6.4	Potentialfunktion . . . . .	78
<b>31</b>	<b>Kürzeste Spannbäume, UNION-FIND-Problem</b> (Vorlesung 26 am 26.01.)	<b>79</b>
31.1	Algorithmus von Kruskal für einen kürzesten Spannbaum . . . . .	79
31.2	UNION-FIND-Problem . . . . .	79
31.2.1	Operationen . . . . .	79
31.2.2	FIND(u) . . . . .	79
31.2.3	UNION(a,b) . . . . .	79
31.2.4	Implementierungsmöglichkeiten . . . . .	79
31.3	Analyse von $C'$ . . . . .	80
<b>32</b>	<b>UNION-FIND-Fortsetzung</b> (Vorlesung 27 am 30.01.)	<b>81</b>
32.1	Zerlegungslemma . . . . .	81
32.2	Die 4 Ereignisse von $\text{cost}(C)$ . . . . .	82

# 1 Einführung (Vorlesung 1 am 17.10.)

## 1.1 Organisatorisches

Mitschrift wird von Studenten erstellt.

Korrekturfarbe für Gummipunkte: Grün!

### Voraussetzungen

- O-Notation
- Turing-Maschine
- Sortieralgorithmen
- Schubfachprinzip
- Gauß-Nummer
- Harmonische Reihe

## 1.2 Kuchen teilen

**Problem:** Ein Kuchen soll unter zwei Personen aufgeteilt werden.

Zwei Lösungsideen:

- perfektes Teilen
- einer teilt den Kuchen und der andere sucht sich eine Hälfte aus.

Was passiert, wenn jemand die Teile des Kuchens unterschiedlich bewertet? (z.B. Kirsche auf einer Seite, viel Sahne auf der anderen Seite)

Perfektes teilen bedeutet, dass jemand *für sich* perfekt teilt. (nach seinem Maßstab)

**Ziel: Fairness** Jeder will  $\frac{1}{n}$  des Kuchens nach ihrem Maßstab. ( $n = \# \text{Personen}$ )

### 1.2.1 1. Algorithmus (für 2 Personen)

1. Erste teilt
2. Zweite sucht aus

Der Algorithmus ist toll, aber es gibt zu viele Schritte. Daher wollen wir den Algorithmus verbessern.

**Ziel:** möglichst wenige Schritte.

### 1.2.2 2. Algorithmus (für 3 Personen)

Anton, Berta und Clara:

1. Anton teilt  $\frac{1}{3} | \frac{2}{3}$
2. Berta teilt  $\frac{2}{3} | \frac{2}{3}$
3. Clara sucht aus.
4. Anton sucht aus.

Fall 1: Clara nimmt eines der rechten Stücke  $\Rightarrow$  Anton nimmt linkes Stück.



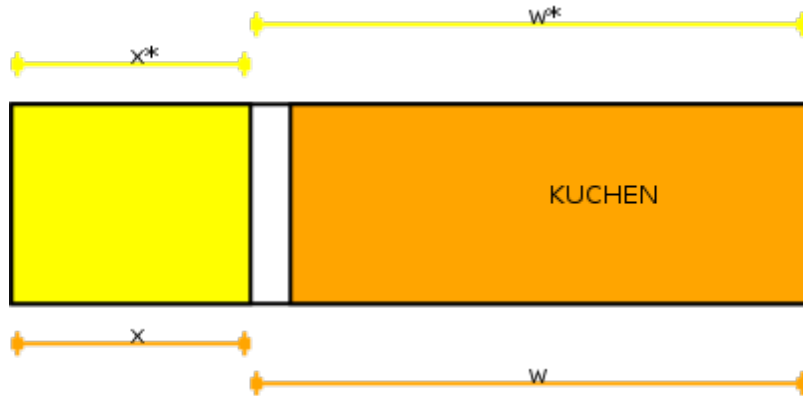
Fall 2: Clara nimmt linkes Stück.

Schubfachprinzip: eines der rechten Stücke ist mindestens  $\frac{1}{3}$

5. Berta ):

### 1.2.3 3. Teilen und Trimmen

1. Anton teilt:



2. Berta:

Fall 1: Berta denkt  $x \leq \frac{1}{3}$

Fall 2: Berta denkt  $x > \frac{1}{3} \Rightarrow$  Trimmen

3. Clara darf sich entscheiden:

Fall 1: will  $x^*$  dann Algorithmus 1. für den Rest

Fall 2: will  $x^*$  nicht.

$\Rightarrow w^* \geq \frac{2}{3}$  für Clara und Anton

### 1.2.4 4. Teilen mit bewegtem Messer

Man nimmt ein Messer und jede Person sagt einfach Stop, wenn die *perfekte Wahl* für die Person getroffen ist.

#Schritte =  $n - 1$

### 1.2.5 5. Simuliertes bewegtes Messer

- Jeder macht bei  $\frac{1}{n}$  eine Markierung
  - der/die Linkeste bekommt das Stück
- #Schritte =  $n + (n - 1) + \dots + 3 + 2 = \theta(n^2)$  (Gauß-Nummer)

### 1.2.6 6. Simuliertes Messer + Zufall

Wie 5., aber

1. Reihenfolge zufällig
2. nur neue Linkeste Markierung werden gemacht

3.  $T(n) = \# \text{erwartete Markierungen}$

$$= \underbrace{\frac{1}{n}}_{\text{Erwartete Anzahl der letzten Markierung}} + \underbrace{T(n-1)}_{\text{Erwartete Anzahl von Markierungen aller Anderen.}}$$

4.  $T(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \theta(\log n)$  (harmonische Reihe)

5. Gesamtlaufzeit  $\leq n * O(\log n) = O(n * \log n)$

### 1.2.7 7. Divide & Conquer

$n$  Personen

$n$  Markierungen bei  $\frac{k}{n}$

#Schritte im Worst Case  $T(n) = n + 2$

### 1.2.8 8. Divide & Conquer + Zufall

(erwartete) Laufzeit pro Teilen  $\theta(\log n)$  also insgesamt  $\theta(n)$

## 2 Einführung Teil 2 (Vorlesung 2 am 20.10.)

### 2.1 Ziele der Vorlesung

- Algorithmen nach den wichtigsten Entwurfsprinzipien entwerfen:
  - Devide and Conquer
  - dynamisches Programmieren
  - bound and bound
  - greedy-Algorithmen
- Algorithmen mit Analysetechniken analysieren im Bezug auf Laufzeit und Speicherbedarf (Stromverbrauch)
  - randomisierte Analyse
  - amortisierte randomisierte Analyse
  - Rekursionsgleichungen
- Vergleich und Beurteilung von Algorithmen nach Einsatzzweck
- Theorie der NP Vollständigkeit verstehen und einfache Vollständigkeitsbeweise führen

*(Stromverbrauch ist zunehmend wichtig, aber nicht Teil der Vorlesung. Allgemein sind Algorithmen mit weniger Laufzeit besser.)*

### 2.2 Rechnermodelle

#### 2.2.1 Turing-Maschine

Eine Turing-Maschine ist ein theoretisches Modell. Es handelt sich um ein unendliches Band mit Symbolen aus einem endlichen Alphabet mit endlichem Zustandsraum. In jedem Schritt wird ein Symbol gelesen, das Band entsprechend der Eingabe beschrieben und der Zustand verändert. Prinzipiell ist alles mit einer Turing-Maschine berechenbar, jedoch teilweise sehr umständlich, weil immer nur ein Symbol gelesen werden kann.

#### 2.2.2 Registermaschine (RAM - random access machine)

Eine RAM funktioniert nach einem ähnlichen Prinzip wie moderne Rechner arbeiten. Es gibt eine potentiell unendliche (unbeschränkte) Anzahl von Registern  $R_0, R_1, R_2, \dots$  wobei jedes Register eine ganze Zahl enthalten kann. Die Programmiersprache ist ähnlich wie Assembler.

*RAM ist auch als random access memory als Arbeitsspeicher bekannt*

### 1. Befehle

Zuweisung  $R_4 = R_{17}$

Rechenbefehl  $R_1 = R_2 + R_3$

$R_1 = R_2 - R_3$

$R_1 = R_2 * R_3$

$R_1 = R_2 / R_3$

### Operanden der Befehle

1. Register  $R_{17}$
2. direkte Operanden (Zahlen) 250
3. indirekte Adressen:  $(R_1)$

*den Inhalt des Registers, dessen Nummer in Register  $R_1$  steht.*

## 2. Sprünge

```
1 GOTO x
2 IF Ri = 0 THEN GOTO x
3
4 GZ R1, label ;if R1 is greater 0, goto label
```

Es sind nur die drei  
Vergleichsoperationen  
GLZ: < 0 , GGZ: >  
0 , GZ: = 0  
erlaubt!

x ist eine Sprungmarke im Programm.

```
1 loop:
2   \\ some commands
3   GOTO loop
```

## 3. HALT

Ein Programm endet immer mit **HALT**

### Ein- und Ausgabe

Eingabe: R0 = n= die Länge der Eingabe R1, R2, ... Rn. Alle andere Zellen sind auf 0 initialisiert.

Ausgabe steht am Ende im Speicher!

### 2.2.3 Berechnung der Laufzeit

#### a) Einheitskostenmaß (EKM)

Jede Operation dauert eine Zeiteinheit.

unfair, weil es Operationen gibt, die offensichtlich komplizierter sind.

#### b) logarithmisches Kostenmaß (LKM)

Laufzeit = Summe der Längen aller vorkommenden Adressen und Operanden.

$$l(x) = \lfloor \log_2 \max\{|x|, 1\} \rfloor + 1$$

$$\begin{aligned} R2 &= (R0) + 250 \\ \dots \text{Kosten} &= l(2) + l(0) + \underbrace{l(R0)}_{\text{Adresse}} + \underbrace{l((R0))}_{\text{Operand}} + \underbrace{l(250)}_{\text{Operanden}} \end{aligned}$$

Das LKM ist gerechter, als das EKM.

Im EKM kann man schwindeln:

Operationen auf langen Daten können in einem Schritt erledigt werden.

Andererseits ist das EKM näher an einem tatsächlichen Prozessor. Sofern die Operanden in ein Wort eines konventionellen Speichers (64 Bit) passen.

Abschätzung:  $LKM \leq O(EKM \cdot l(\text{längster vorkommender Operand oder Adresse}))$

Wenn die größten vorkommenden Zahlen nicht zu groß sind, dann ist das EKM realistisch.

LKM ist fairer, wenn es um sehr unterschiedliche Operanden geht (verschieden lang)

## 2.3 Laufzeit eines Algorithmus

Man muss den möglichen Eingaben eine Länge zuordnen.

$x$ .. Eingabe  $L(x)$

Bsp.  $n$  Zahlen  $x_1, x_2, \dots, x_n$  sortieren:  $L = n$

Bsp. Multiplikation von langen Zahlen  $x, y$ :  $L = \#$  Bits in der Eingabe.

Bsp. Lösen eines linearen Gleichungssystems:  $Ax = b$   $A \in \mathbb{Z}^{n \times n}$   $b \in \mathbb{Z}^n$   $x \in \mathbb{Q}^n$

Länge der Eingabe:  $n^2$

Gauß-Elimination  $O(n^3)$  Zeit, erfordert Rechnen mit rationalen Zahlen.

Man kann Zeigen, dass die Länge der Zähler und Nenner in den Zwischenergebnissen höchstens  $n$ -Mal ( $\leq n$ ) ist, wenn man Brüche immer kürzt.

Laufzeit im LKM:  $O(n^4, l(\text{größte Eingabezahl}))$

*Tendenziell  
kompliziertes Beispiel,  
um zu illustrieren, dass  
LKM nicht immer  
leicht zu berechnen ist.*

Was ist die Laufzeit eines Algorithmus?

$T(x)$  = Laufzeit des Algorithmus bei Eingabe  $x$

$$(Analyse\text{im}\text{schlimmsten}\text{Fall}).T(n) = \max\{T(x) | L(x) = n\}$$

### Andere Möglichkeiten

Analyse im Durchschnitt, Erwartungswert der Laufzeit

Benötigt eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben.

### 3 Rechnermodelle (Fortsetzung) (Vorlesung 3 am 24.10.)

#### 3.1 Warum nicht die Turingmaschine?

Die Registermaschine ist näher am heutigen Rechnermodell. Die Turingmaschine ist viel primitiver.

**Satz:**

- a) Ein Algorithmus, der auf einer Registermaschine Laufzeit  $T(n)$  im logarithmischen Kosteneinheitsmaß hat, kann auf einer Turingmaschine in Laufzeit  $O((T(n))^3)$  simuliert werden.
- b) Ein Algorithmus mit Laufzeit  $U(n)$  auf einer Turingmaschine kann mit Laufzeit  $O(U(n) \log U(n))$  auf einer Registermaschine im LKM simuliert werden.

zu b) In Zeit  $U(n)$  kann die Maschine höchstens die Felder  $-U(n) \dots + U(n)$  beschreiben. Adressen sind durch  $2U(n)$  beschränkt. Jeden Schritt der TM kann in konstant vielen Operationen der Registermaschine simuliert werden.

$\rightarrow O(\log U(n))$

zu a) Speicherinhalt auf dem Band notieren.

$i$  : (Inhalt von Register  $i$ ).  $(i+1)$  : Inhalt von Register  $(i+1)$ . ...

Register mit Inhalt 0 können weggelassen werden. Register werden in natürlicher Reihenfolge aufgeschrieben. Alle Zahlen binär oder dezimal (nach Belieben).

Die Länge des Bandes  $= L$  ist durch  $T(n)$  beschränkt.

Jede Adresse, jede Registereinheit wurde bei der letzten Benutzung in voller Länge bei  $T(n)$  berücksichtigt.

#### 3.2 Elementare Operationen

1. Adresse im Speicher suchen; (Adresse steht im linken Zwischenbereich)
2. entsprechenden Inhalt zwischen Speicher und Zwischenbereich übertragen
3. Rechenoperationen im Zwischenbereich

---

<sub>1</sub> R2 = (R17)

---

Jede Stelle, die verglichen wird, erfordert im schlimmsten Fall ein Wandern über das gesamte Band.

Operation 1 dauert  $O(L^2)$  Schritte, wobei  $L$  die Länge des Bandes ist.

Operation 2 ist ähnlich. Gegebenenfalls muss man den rechten Teil des Bandinhalts verschieben (Um eine Stelle verschieben dauert  $O(L)$  Zeit,  $\leq O(L^2)$  insgesamt).

Operation 3  $\leq O(L^2)$

$O(L^2)$  für 1 Schritt der Registermaschine  $= O(T(n))^2$

### 3.3 Teile und Herrsche

(eng. divide and conquer) (lat. divide et impera)

1. Zerlege das Problem  $P$  in Teilprobleme  $P_1, P_2, \dots, P_k$  (typischerweise  $k = 2$ )
2. Löse die Teilprobleme rekursiv.
3. Füge die Teillösung zur Lösung von  $P$  zusammen.

#### 3.3.1 Beispiel A: Quicksort

1. Wahl eines Pivotelementes  $a$ :  
Zerlegung in Elemente  $\underbrace{\leq a}_{\text{Teilproblem}}, = a, > a$
3. Teilfolgen aneinanderhängen.

*Bei Quicksort ist der erste Schritt aufwändiger, bei Mergesort der letzte Schritt.*

#### 3.3.2 Beispiel B: Mergesort (Sortieren durch Verschmelzen)

1. Zerlegung in 2 gleich große Teile
3. Verschmelzen der beiden sortierten Teillisten.

Laufzeit  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n)$

$n$  gerade  $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Lösung  $T(n) = O(n \log n)$

#### 3.3.3 Analysemöglichkeiten

- I. Lösung erraten und durch vollständige Induktion beweisen.
- II. Wiederholtes einsetzen auf der rechten Seite:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \quad (c > 0)$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{4}\right) + c * \frac{n}{2}$$

$$T(n) \leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn$$

$$= 4 \underbrace{T\left(\frac{n}{4}\right)}_{=2T\left(\frac{n}{8}\right)+c\frac{n}{4}} + cn + cn$$

$$\leq 8T\left(\frac{n}{8}\right) + cn + cn + cn$$

$$\leq 2^k T\left(\frac{n}{2^k}\right) + k.c.n$$

Annahme  $n = 2^l$  ist eine Zweierpotenz  $l = \log_2 n$

$$\begin{aligned} T(n) &= \underbrace{2^l}_n \underbrace{T(1)}_{\text{konst.}} + \underbrace{l}_{\log_2 n} \cdot c \cdot n = O(n \log n) \\ &= O(n) + O(n \log n) \end{aligned}$$

nur gültig für Zweierpotenzen.

Möglichkeit a)  $n$  auf die nächste  $n' = 2^l$  aufrunden.

$$n \leq n' < 2n$$

Sortieren von  $n$  Elementen kann nicht länger dauern als Sortieren von  $n'$  Elementen.  
(zu beweisen! z.B. mit vollst. Induktion anhand der Rekursion)

$$T(n) \leq T(n') = O(n' \log n') = O(2n \cdot \log(2n)) = O(n \log n) \checkmark$$

Möglichkeit b) Als Inspiration, um auf die Vermutung  $O(n \log n)$  zu bekommen. Beweis mit Methode I.

III. Rekursionsbaum  $\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor = \lfloor \frac{n}{4} \rfloor$  Laufzeit:  $2^l$  Probleme konstanter Größe.  $T(1), T(2) \leq c'$

$$\text{Ebene 0 : } \leq \Theta(n)$$

$$\text{Ebene 1 : } \leq 2\Theta(\lceil \frac{n}{2} \rceil)$$

$$\text{Ebene 2 : } \leq 4\Theta(\lceil \frac{n}{4} \rceil)$$

$$\Theta(n) \leq c \cdot n$$

$$\begin{aligned} \text{Summe} &\leq cn + 2c \lceil \frac{n}{2} \rceil + 4c \lceil \frac{n}{4} \rceil + \dots + 2^{l-1} c \lceil \frac{n}{2^{l-1}} \rceil + 2^l c' \\ &\leq cn + 2c(\frac{n}{2} + 1) + 4c(\frac{n}{4} + 1) + \dots \\ &= \underbrace{cn + cn + \dots + cn}_{l\text{-mal}} + \underbrace{2c + 4c + 8c + \dots + 2^{l-1}c}_{(2^l - 2)c} + 2^l c' \end{aligned}$$



## Höhere Algorithmik - 4. Vorlesung

### Bestimmung des Maximums und des Minimums von $n$ Zahlen

**Problemstellung:** Gegeben sind die Zahlen  $a_1, \dots, a_n$ . Gesucht werden das Maximum sowie das Minimum von diesen Zahlen.

Will man entweder nur das Maximum oder nur das Minimum dieser Zahlen bestimmen, vergleicht man die erste Zahl mit der zweiten. Je nach gesuchtem Ergebnis muss man entweder den größeren oder den kleineren der beiden Werte mit dem nächsten Wert vergleichen. Das Ganze wird fortgesetzt, bis alle Zahlen miteinander verglichen wurden.

Aus diesem Algorithmus folgt, dass zur Bestimmung des Maximums allein  $n - 1$  Vergleiche ausreichen. Genauso reichen zur Bestimmung des Minimums  $n - 1$  Vergleiche.

In der Summe sind dies  $2n - 2$  Vergleiche. Die asymptotische Laufzeit liegt in  $\mathcal{O}(n)$ .

Anschaulich ist auch sofort klar, dass es unmöglich ist, eine bessere als lineare Laufzeit zu erreichen, da sämtliche Zahlen betrachtet werden müssen. Daher ist dies einer der wenigen Fälle in dieser Vorlesung, in der die Konstante betrachtet und verbessert werden soll.

**Optimierung:** Für  $n \geq 2$  kann das Maximum nicht gleichzeitig das Minimum sein (und umgekehrt). Hieraus kann gefolgert werden, dass nur  $2n - 3$  Vergleiche benötigt werden.

### Teile und herrsche

Wir betrachten die Teilfolgen  $L$  (links) und  $R$  (rechts) mit  $\lfloor \frac{n}{2} \rfloor$  und  $\lceil \frac{n}{2} \rceil$  Elementen.

Das maximale Element der linken Teilfolge  $L$  sei  $l_{max}$ , das minimale  $l_{min}$ . Analog dazu seien  $r_{max}$  das maximale und  $r_{min}$  das minimale Element der rechten Teilfolge  $R$ .

Bestimme  $l_{min}, l_{max}, r_{min}, r_{max}$

$$T(n) = \text{Anzahl der Vergleiche}$$

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2$$

$$T(1) = 0$$

$$T(2) = 1$$

$$T(3) = 3 = 2 + 1 + 0$$

$$T(4) = 4 = 2 + 1 + 1 \leq 2n - 3$$

Zur Bestimmung des gesamten Maximums benötigen wir zwei weitere Vergleiche: Das maximale Element von der Gesamtfolge ist das Maximum von  $l_{max}$  und  $r_{max}$ , das minimale Element der Gesamtfolge ist das Minimum von  $l_{min}$  und  $r_{min}$ .

Analyse falls  $n$  eine Zweierpotenz ist ( $n = 2^k$ ):

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$

Ansatz:  $T(n) = An + B$  (lineare Funktion)

Durch Einsetzen unseres Ansatzes erhalten in die Rekursionsgleichung folgt:

$$\begin{aligned} An + B &= 2\left(A\frac{n}{2} + B\right) + 2 \\ &= An + 2B + 2 & |(-An - B) \\ 0 &= B + 2 & |(-2) \\ B &= -2 \end{aligned}$$

Bestimmung von  $A$  durch Einsetzen von  $B = -2$  in den Ansatz:

$$\begin{aligned} T(n) &= An + B & \text{für den Fall } T(2) = 1 \text{ folgt:} \\ T(2) &= A \cdot 2 + B = 1 \\ T(2) &= A \cdot 2 - 2 = 1 & (+2) \\ 2A &= 3 & (/2) \\ A &= \frac{3}{2} \end{aligned}$$

Für  $A = \frac{3}{2}$  und  $B = -2$  erfüllt  $A \cdot n + B$  also die Rekursion. Wir erhalten also als Lösung für den Fall  $n = 2^k$  ( $n$  ist Zweierpotenz):

$$T(n) = \frac{3}{2}n - 2$$

### Verschiebung im Wertebereich

Eine leicht zu lösende Rekursion hat zum Beispiel die Form:  $h(n) = 2h\left(\frac{n}{2}\right) = 4h\left(\frac{n}{4}\right) \Rightarrow h(n) =$   
 $an$

Schwieriger ist es, wenn die Gleichung einen Störfaktor enthält:  $f(n) = 2f\left(\frac{n}{2}\right) + \underbrace{2}_{\text{Störfaktor}}$

$$f(n) = 2f\left(\frac{n}{2}\right) + 2 \quad |(+2)$$

$$\Leftrightarrow f(n) + 2 = 2f\left(\frac{n}{2}\right) + 4 \quad (2 \text{ ausklammern})$$

$$\Leftrightarrow f(n) + 2 = 2\left(f\left(\frac{n}{2}\right) + 2\right)$$

definiere  $g(n) = f(n) + \underbrace{2}_{\text{Durch Ansatz}}$

$$g(n) = 2g\left(\frac{n}{2}\right)$$

additiver Störfaktor ist weg.

$$g(n) = f(n) + c$$

$$f(n) = 2f\left(\frac{n}{2}\right) + 2$$

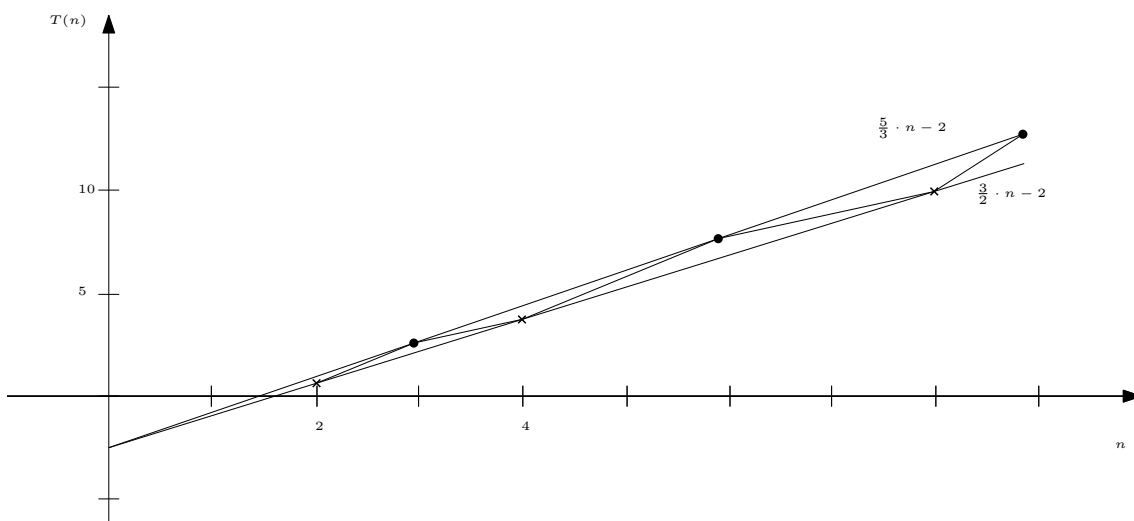
$$f(n) = g(n) - c$$

$$g(n) - c = 2\left(g\left(\frac{n}{2}\right) + 2\right) - c = 2g\left(\frac{n}{2}\right) - 2c + 2$$

$$-c = -2c + 2$$

$$c = 2$$

**Einschub:**



Beispiel: (Fibonacci-Folge mit Störfaktor)

$$a_n = a_{n-1} + a_{n-2} + 3 \quad |(+3)$$

$$(a_n + 3) = (a_{n-1} + 3) + (a_{n-2} + 3) \quad \text{Fibonacci um 3 verschoben}$$

### Verschiebung im Definitionsbereich

Ähnlich, wie die Verschiebung im Wertebereich funktioniert die Verschiebung im Definitionsbereich.

**Beispiel:**

$$\begin{aligned}
 g(n) &= 2 \left( g(\lfloor \frac{n+3}{2} \rfloor) \right) && \text{substituiere } n = m + 3 \\
 g(m+3) &= 2 \left( g(\lfloor \frac{m+6}{2} \rfloor) \right) \\
 g(m+3) &= 2 \left( g(\lfloor \frac{m}{2} + \frac{6}{2} \rfloor) \right) \\
 g(m+3) &= 2 \left( g(\lfloor \frac{m}{2} + 3 \rfloor) \right) \\
 g(m+3) &= 2 \left( g(\lfloor \frac{m}{2} \rfloor + 3) \right)
 \end{aligned}$$

Wir setzen  $h(n) = g(n+3)$  und erhalten:

$$h(n) = 2 \left( h(\lfloor \frac{n}{2} \rfloor) \right)$$

Auf diese Art haben wir den Störfaktor innerhalb des Funktionsaufrufs beseitigt und könnten jetzt regular mit der Bearbeitung dieser Aufgabe weitermachen.

### Erweiterung auf nicht-2er Potenzen

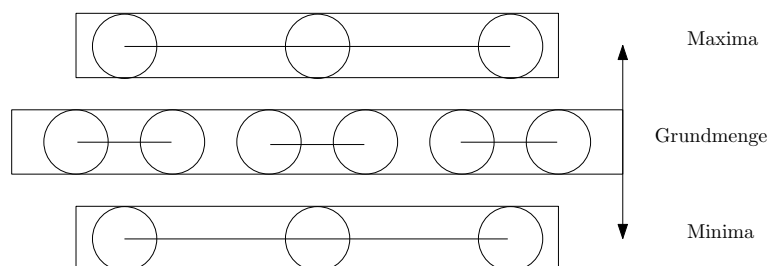


Abbildung 1: Vergleiche zum Finden von Maxima und Minima

Sei  $M$  eine  $n$  elementige Menge. Zur Vereinfachung gehen wir davon aus, dass  $n$  eine gerade Zahl ist. Wir bilden nun  $\frac{n}{2}$  Paare und vergleichen diese miteinander. Hierfür benötigen wir  $\frac{n}{2}$  Vergleiche.

Wir teilen die Elemente in zwei  $\frac{n}{2}$  elementige Untermengen (ähnlich, wie bei Mergesort), von denen eine die Maxima, die andere die Minima aus den vorangegangenen Vergleichen enthält. Beide Untermengen haben nun eine ungerade Anzahl an Elementen. Wir vergleichen wieder paarweise die Maxima und die Minima. Dafür benötigen wir jeweils  $\frac{n}{2} - 1$  Vergleiche:

$$\left. \begin{array}{l} \frac{n}{2} - 1 \\ \frac{n}{2} - 1 \end{array} \right\} 3 \frac{n}{2} - 2$$

Insgesamt benötigen wir also  $\frac{3}{2}n - 2$  Vergleiche - und damit genauso viele wie für den Fall dass  $n$  als Zweierpotenz darstellbar ist. Das Verfahren funktioniert nach dem Bottom-Up Prinzip.

### Multiplikation von zwei n-stelligen Zahlen

Gegeben ist eine Basis  $B$ , z.B.  $B = 2$  (binär) oder  $B = 10$  (dezimal) oder  $B = 2^{32}$

$$x = (x_{n-1}x_{n-2}\dots x_1x_0)_B = \sum_{i=0}^{n-1} x_i B^i$$

32 Bit	32 Bit	32 Bit	32 Bit	32 Bit
--------	--------	--------	--------	--------

Abbildung 2: 32 Bit Blöcke, auf diesen Schulmethode anwenden

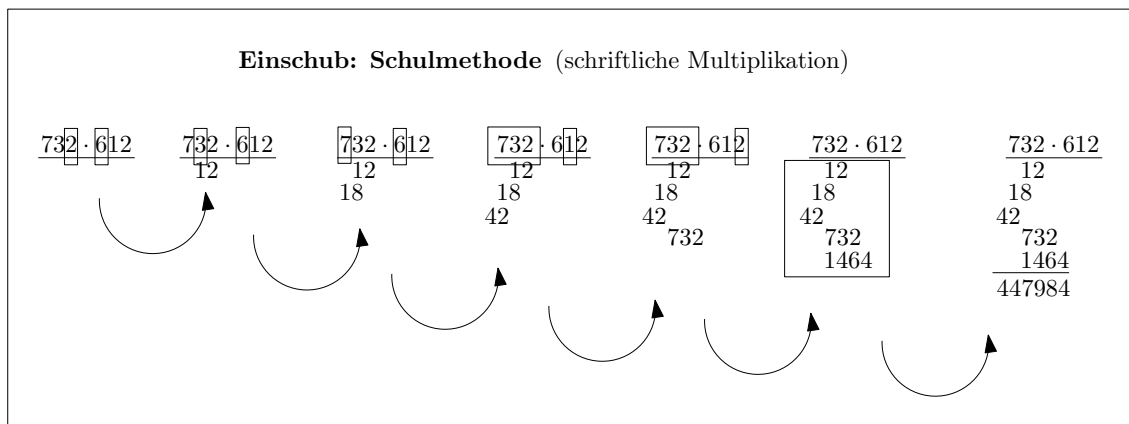


Abbildung 3: Schulmethode (ohne Übertrag)

$$y = (y_{n-1}y_{n-2}\dots y_1y_0)_B = \sum_{i=0}^{n-1} y_i B^i$$

**Schulmethode:** Multipliziere jedes  $x_i$  mit jedem  $y_j$  und addiere alle Produkte an die geeignete Stelle (wie in Abbildung 3).

$$x \cdot y = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j B^{i+j} \text{ Laufzeit in } \mathcal{O}(n^2)$$

**Teile und herrsche** (Basis  $B = 2$ )

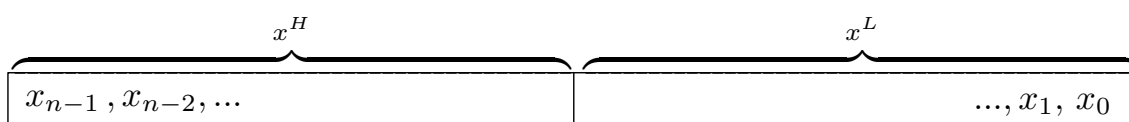


Abbildung 4: Teile und herrsche

Wie in Abbildung 4 zu sehen, sei  $x$  eine  $n$  Bit lange Zahl. Zur Vereinfachung nehmen wir an, dass  $n$  gerade ist.

Wir teilen  $x$  in zwei Teile, wobei ein Teil die höherwertigen Bits und der andere Teil die niedrigerwertigen Bits enthält. Den Teil mit den höherwertigen Bits nennen wir  $x^H$ , den mit den niedrigerwertigen Bits nennen wir  $x^L$ .

Dann gilt  $x = x^H \cdot 2^{\frac{n}{2}} + x^L$ .

Außerdem sei  $y$  eine entsprechend gewählte zweite Zahl für die gilt:

$$y = y^H \cdot 2^{\frac{n}{2}} + y^L$$

$$x^H := (x_{n-1}x_{n-2}\dots x_{\frac{n}{2}})_2 \quad \frac{n}{2} \text{ höherwertige Bits}$$

$$x^L := (x_{\frac{n}{2}-1}\dots x_0)_2 \quad \frac{n}{2} \text{ niederwertige Bits}$$

$$x = x^H \cdot \underbrace{2^{\frac{n}{2}}}_{\text{Linksshift um } 2^{\frac{n}{2}} \text{ Bits}} + x^L$$

$$\begin{aligned} xy &= (x^H \cdot 2^{\frac{n}{2}} + x^L)(y^H \cdot 2^{\frac{n}{2}} + y^L) \\ &= x^H y^H + (x^H y^L + x^L y^H)2^{\frac{n}{2}} + x^L y^L \end{aligned}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Diese Rekursion ist dargestellt in Abbildung 5.

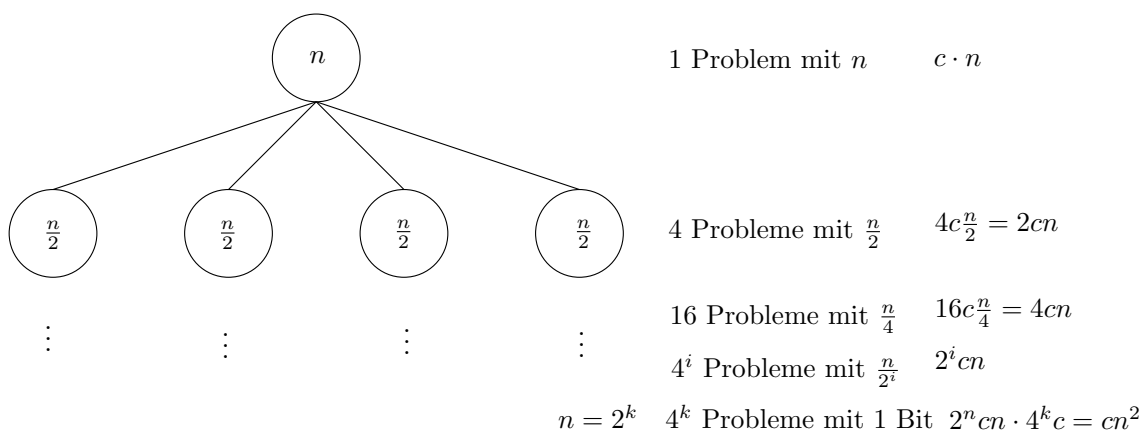


Abbildung 5: Rekursionsbaum - Schulmethode

$$\Rightarrow \text{Laufzeit: } T(n) = \mathcal{O}(n^2)$$

Somit ist bringt „Teile und Herrsche“ in diesem Fall keine Verbesserung. Grund hierfür ist, dass wir vier Teilbäume haben.

Um eine Verbesserung zu erzielen, müssen wir die Anzahl der Teilbäume auf drei reduzieren.

### Algorithmus von Karatsuba

**Satz:** Es existiert ein Algorithmus, mit dem die Multiplikation zweier  $n$ -stelliger Zahlen in weniger als  $\mathcal{O}(n^2)$  möglich ist.

Ein Algorithmus, der diesen Satz erfüllt ist der Algorithmus von Karatsuba.

Der Algorithmus von Karatsuba ist ein schneller Multiplikationsalgorithmus. Er reduziert für die Multiplikation zweier  $n$ -stelliger Zahlen die Anzahl der nötigen einstelligen Multiplikationen im Allgemeinen auf höchstens  $3n^{\log_2 3}$ . Für  $n$  die ein Vielfaches von zwei sind sogar exakt auf  $n^{\log_2 3}$ . Damit ist er schneller als die klassische Schulmethode und erfüllt die Forderung aus dem vorangegangenen Abschnitt.

Der zugehörige Rekursionsbaum ist in Abbildung 6 dargestellt.

1.  $z_1 = (x^L + x^H) \cdot (y^L + y^H) = x^L y^L + x^H y^L + x^L y^H + x^H y^H$  (1 Multiplikation)
2.  $z_2 = x^L y^L$  (1 Multiplikation)
3.  $z_3 = x^H y^H$  (1 Multiplikation)
4.  $z_4 = z_1 - z_2 - z_3$
5.  $xy = \underbrace{z_3 2^n}_{(*)} + z_4 2^{\frac{n}{2}} + z_2$

(\*) Diese Multiplikation kann durch shiften sehr effizient gemacht werden!

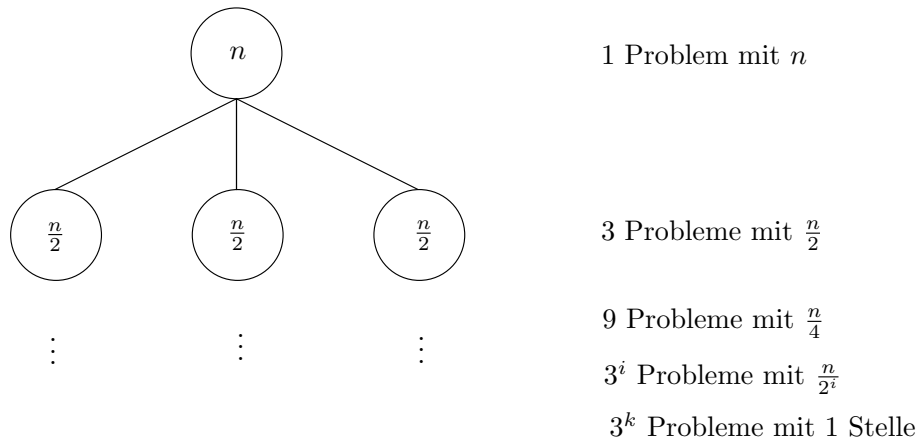


Abbildung 6: Rekursionsbaum - Algorithmus von Karatsuba

Summiert man den Aufwand pro Ebene auf, erhält man für Ebene  $i$  den Aufwand  $(\frac{3}{2})^i \cdot c \cdot n$ .  
Summiert man die Ebenen auf, erhält man als Gesamtaufwand:

$$\begin{aligned}
 &= \sum_{i=0}^k \left(\frac{3}{2}\right)^i \cdot c \cdot n \\
 &= c \cdot n \cdot \sum_{i=0}^k \left(\frac{3}{2}\right)^i
 \end{aligned}$$

Da diese Formel eine geometrische Reihe beschreibt, können wir eine endliche Partialsumme wie folgt berechnen:

$$\sum_{i=0}^k \left(\frac{3}{2}\right)^i = \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} = \mathcal{O}\left(\left(\frac{3}{2}\right)^k\right)$$

Für den Gesamtaufwand folgt hieraus:

$$\begin{aligned}
 \Rightarrow c \cdot n \cdot \sum_{i=0}^k \left(\frac{3}{2}\right)^i &= \mathcal{O}\left(c \cdot n \cdot \frac{3^k}{2^k}\right) && \text{(Gilt, da } n = 2^k \text{ )} \\
 &= \mathcal{O}(3^k)
 \end{aligned}$$

Da  $3^k = 3^{\log_2 n} = n^{\log_2 3}$  (Logarithmengesetze), folgt:

$$\mathcal{O}(n^\gamma), \text{ mit } \gamma = \log_2 3 \approx 1,7$$

### Teile-und-herrsche-Rekursionen

Die Strategie „Teile-und-herrsche“ zerlegt ein Problem  $T(n)$  in  $a$  Teilprobleme der Größe  $\frac{n}{b}$ . Hinzu kommt der Aufwand für das Zerlegen in die Teilprobleme und Zusammenfügen derselbigen. Allgemein kann man also folgende Formel dafür angeben:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$



## MASTER-Theorem

Das Master-Theorem, auch Hauptsatz der Laufzeitfunktionen, kann bei vielen rekursiven Funktionen, wie sie beispielsweise bei vielen Divide and Conquer Algorithmen auftreten, eine schnelle Einordnung in Laufzeitklassen ermöglichen.

Anzahl Probleme	Größe	Aufwand	
1	$n$	$f(n)$	$n^4$
$a$	$\frac{n}{b}$	$a f(\frac{n}{b})$	$n^4 \cdot \frac{a}{b^4}$
$a^2$	$\frac{n}{b^2}$	$a^2 f(\frac{n}{b^2})$	$n^4 \cdot (\frac{a}{b^4})^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a^k$	$\frac{n}{b^k}$	$a^k f(\frac{n}{b^k})$	$n^4 \cdot (\frac{a}{b^4})^k$

$$k = \log_b n$$

$$\frac{n}{b^k} \text{ konstant, z.B. wenn } f(n) = n^4$$

### 3 Fälle:

1. fallende geometrische Reihe:  $(\frac{a}{b^4} < 1): T(n) = \mathcal{O}(f(n))$
2. wachsende geometrische Reihe:  $(\frac{a}{b^4} > 1): T(n) = \mathcal{O}(a^k) = \mathcal{O}(n^{\log_b a})$
3. konstant:  $\frac{a}{b^4} = 1: T(n) = \mathcal{O}(n^{\log_b a} \log n)$

## 4 Rekursion (Fortsetzung) (Vorlesung 5 am 31.10.)

### 4.1 Motivation Master-Theorem

$$T(n) = \underbrace{T\left(\frac{n}{b}\right)}_{T(\lfloor \frac{n}{b} \rfloor) + \dots + T(\lceil \frac{n}{b} \rceil)} * a + f(n)$$

Für Probleme  $\leq n_0$  wird das Problem irgendwie direkt gelöst.

Startbedingung:  $1 \leq T(n) \leq M$  für  $n \leq n_0$

In der Praxis muss man natürlich irgendwann das  $n_0$  ausrechnen und kann nicht beliebig lange aufteilen.

Die Konstanten  $a \geq 1$  und  $b > 1$  müssen erfüllt sein und außerdem müssen wir fordern:

$$\begin{aligned} \lceil \frac{n}{b} \rceil &\leq n - 1 \text{ für } n > n_0 \\ \Leftrightarrow \frac{n}{b} &\leq n - 1 \\ n(1 - \frac{1}{b}) &\geq 1 \\ n &\geq \frac{b}{b-1} \\ \Rightarrow n_0 &\geq \frac{b}{b-1} \end{aligned}$$

sonst werden die Probleme nicht kleiner und die Rekursion kann nicht gelöst werden.

$$n \log_b n \text{ Elemente} \begin{cases} 1 \text{ Problem der Größe } n & \text{Aufwand } 1f(n)n^k \text{ Annahme } f(n) = n^k \\ 2 \text{ Probleme der Größe } \frac{n}{b} & \text{Aufwand } a * f(\frac{n}{b})a(\frac{n}{b})^k \\ 3 \text{ Probleme der Größe } \frac{n}{b^2} & \text{Aufwand } a^2 * f(\frac{n}{b^2})a^2(\frac{n}{b})^k \\ \vdots & \vdots \end{cases}$$

**Beispiel: Mergesort**

$$\begin{aligned} a &= b = 2 \\ \gamma &= \log_2 2 = 1 \end{aligned}$$

### 4.2 Master-Theorem für divide and conquer-Rekursion

$$\begin{aligned} a \geq 1, b > 1, M, n_0 \geq 1 & (\frac{n_0}{b} \leq n_0 - 1) \\ f(n), T(n) & \text{Funktionen auf den natürlichen Zahlen} \\ f(n) & \geq 0 \end{aligned}$$

Es gelten die Rekursionsbedingungen

$$\begin{aligned} T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) & (n > n_0) \\ T(n) &\geq aT(\lfloor \frac{n}{b} \rfloor) + f(n) & (n > n_0) \\ 1 &\leq T(n) \leq M \end{aligned}$$

Dann definieren wir den kritischen Exponenten

$$n = \log a > 0$$

(-) Wenn  $f(n) = \mathcal{O}(n^{\gamma-\epsilon})$  für ein  $\epsilon > 0$ , dann  
 $T(n) = \Theta(n^\gamma)$

(=) Wenn  $f(n) = \Theta(n^\gamma)$  ist, dann  
 $T(n) = \Theta(n^\gamma \log n)$

(+) Wenn  $f(n) = \Theta(n^{\gamma+\epsilon})$  für ein  $\epsilon > 0$  ist oder wenn die Regularitätsbedingung erfüllt ist  
 $\exists c < 1$ :

(\*)  $a \cdot f(\lceil \frac{n}{b} \rceil) < c \cdot f(n)$  für alle  $n > n_0$   
dann gilt:  $T(n) = \Theta(f(n))$

#### 4.2.1 Bemerkungen

1. Wenn  $f$  monoton ist, dann gelten die Schlussfolgerungen auch für beliebig gemischtes Auf- und Abrunden.
2. Mit (\*) kann man auch Funktionen wie  $f(n) = 2^n$  oder  $f(n) = 2^{\sqrt{n}}$  erfassen.
3.  $\Omega(n^{\gamma+\epsilon})$  im Fall (+) reicht leider nicht.
4.  $f(n) = n \log n, \gamma = 1$  wird nicht erfasst.

#### 4.3 Beweis: Master-Theorem

a.) Wir betrachten die oberen Schranken für die Fälle (-) und (=)

(a) Ersetze  $f(n)$  durch die oberen Schranke  $u \cdot n^k$   
 $f(n) \leq u \cdot n^k$  Finde eine Funktion  $P(n)$  mit  $(***) P(n) \geq aP(\lceil \frac{n}{b} \rceil) + u n^k$  für  
 $n \geq n_0$   
und  $P(n) \geq M$  für  $n \geq n_0$   
Dann ergibt sich durch vollständige Induktion:  $T(n) \leq P(n)$   
Basis: ( $n \leq n_0$ )

$$\begin{aligned} T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) \leq (\text{I.V.}) \\ &\leq aP(\lceil \frac{n}{b} \rceil) + f(n) \\ &\leq aP(\lceil \frac{n}{b} \rceil) + u n^k \leq P(n) \end{aligned}$$

(b) Verschiebung des Definitionsbereiches, um  $\lceil \rceil$  los zu werden.

$v = \frac{b}{b-1} \Rightarrow -\frac{v}{b} = 1 - v$   
 $P(n) = T'(n - v)$  bzw.  $T'(n) = P(n + v)$   
 $T'$  ist jetzt auf  $\mathbb{R}_{>0}$  definiert.  
Wir bestimmen dann  $T'$  so, dass  
 $(**) T'(n) = aT'(\frac{n}{b}) + u' n^k$  ( $u$  ist eine Konstante)

**Behauptung:** aus (\*\*) folgt (\*\*\*), falls  $T'$  monoton wächst

$$\begin{aligned}
 \underbrace{P(n)} &\geq aP(\lceil \frac{n}{b} \rceil) + un^k \\
 \text{L.S.} = P(n) &= T'(n-v) = aT'(\frac{n}{b} - \frac{v}{b}) + u'(n-v)^k \\
 \text{R.S.} &= aP(\lceil \frac{n}{b} \rceil) + un^k \\
 &= aT'(\lceil \frac{n}{b} \rceil - v) + un^k \\
 &< aT'(\frac{n}{b} + 1 - v) + un^k \\
 &= aT'(\frac{n-v}{b}) + un^k
 \end{aligned}$$

Jetzt müssen wir nur noch  $u'$  so wählen, dass  $u'(n-v)^k \geq un^k$  für  $n \geq n_0 u' \geq u \frac{n_0^k}{(n_0-v)^k}$

Lösen von (\*\*) durch Ansatz:

Fall (-)  $k = \gamma - \epsilon : T'(n) = Dn^\gamma + En^k$  Einsetzen in (\*\*)

$$\begin{aligned}
 Dn^\gamma + En^k &= aD(\frac{n}{b})^\gamma + aE(\frac{n}{b})^k + u'n^k \\
 &= Dn^\gamma \underbrace{\frac{a}{b^\gamma}}_1 + n^k(aE\frac{1}{b^k} + u')
 \end{aligned}$$

$$\begin{aligned}
 E(1 - \frac{a}{b^k}) &= u', E = \frac{u'}{1 - \frac{a}{b^2}} \\
 E(1 - \frac{b^\gamma}{b^{\gamma-\epsilon}}) &= u' \\
 E(1 - b^\epsilon) &= u' \\
 \underline{E} &= \frac{-u'}{b^\epsilon - 1} < 0
 \end{aligned}$$

$D$  ist noch frei: Wähle  $D$  groß genug, dass  $P(n) = T'(n-v) = D(n-v)^\gamma + E(n-v)^k \geq M$  für  $n \leq n_0$  ist.

Fall (=)

$$\begin{aligned}
 T'(n) &= Dn^\gamma + En^\gamma \log_b n \\
 \dots \Rightarrow E &= u', D \text{ bleibt frei. - } D \text{ groß genug.}
 \end{aligned}$$

Ergebnis im Fall (-)  $T(n) \leq D(n-v)^\gamma + E(n-v)^{\gamma-k} = \mathcal{O}(\backslash^\gamma)$

Ergebnis im Fall (=)  $= \mathcal{O}(\backslash^\gamma \log \frac{1}{\gamma})$

## 5 Master Theorem (Fortsetzung) (Vorlesung 6 am 3.11.)

### 5.1 Beweis Fortsetzung

Fall (+)

$$T(n) \leq T(\lceil \frac{n}{b} \rceil) + f(n)$$

$$T(n) \geq T(\lfloor \frac{n}{b} \rfloor) + f(n)$$

$$f(n) = \Theta(n^{\gamma+\epsilon})$$

$$\gamma = \log_b a$$

$$\text{oder: } \forall n > n_0 : a \cdot f(\lceil \frac{n}{b} \rceil) < c \cdot f(n)$$

$c < 1$  ist eine Konstante

$$\Rightarrow T(n) = \Theta(f(n))$$

### Beweis (Induktion)

untere Schranke  $T(n) \geq f(n)$  (aus der Rekursion)  $\Rightarrow T(n) = \Omega(f(n))$

obere Schranke: Ansatz:  $T(n) \leq D \cdot f(n)$

Versuch eines Beweises durch Induktion.

$n_0$  groß genug machen, dass  $\frac{n_0}{b} \leq n_0 - 1 \Rightarrow \frac{n}{b} \leq n - 1 \forall n \geq n_0$

$\Rightarrow \lceil \frac{n}{b} \rceil < n$  Induktion kann funktionieren.

Induktionsschritt:  $n \geq n_0$  für  $i < n$  sei  $T(i) \leq D \cdot f(i)$  schon bewiesen.

$$\begin{aligned} T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) \\ &\leq a \cdot D \cdot f(\lceil \frac{n}{b} \rceil) + f(n) \quad \text{nach I.V.} \\ &\leq D \cdot c f(n) + f(n) \quad \text{Regularitätsbedingung} \\ &\leq D \cdot f(n) \end{aligned}$$

$$\begin{aligned} &\underbrace{Dc + 1 \leq D}_{\text{notwendig}} \\ &\Leftrightarrow D(1 - c) \Leftrightarrow D \geq \frac{1}{1 - c} \end{aligned}$$

Induktionsbasis: Wähle  $D$  groß genug, dass  $T(i) \leq Df(i)$  für  $i = 1, 2, \dots, n_0 - 1$  gilt.

(Voraussetzung:  $f(i) > 0$ )

$$D = \max\left\{\frac{T(1)}{f(1)}, \frac{T(2)}{f(2)}, \dots, \frac{T(n_0)}{f(n_0)}, \frac{1}{1-c}\right\}$$

2. Fall:  $f(n) = \Theta(n^{\gamma+\epsilon}), \epsilon > 0$

Obere Schranke (a) Ersetze  $f(n)$  durch  $u \cdot n^{\gamma+\epsilon}$

Beweise, dass  $f(n) = u \cdot n^{\gamma+\epsilon}$  die Regularitätsbedingung erfüllt. (zunächst ohne Aufrunden, weil leichter).

$$a \cdot f(\frac{n}{b}) < c \cdot f(n) \text{ L.S. } = a \cdot u \cdot (\frac{n}{b})^{\gamma+\epsilon} = \frac{a \cdot u \cdot n^{\gamma+\epsilon}}{b^{\gamma+\epsilon}}$$

$$\text{R.S. } = c \cdot u \cdot n^{\gamma+\epsilon}$$

$n_0$  so groß wählen, dass  $\frac{(\frac{n}{b}+1)^{\gamma+\epsilon}}{(\frac{n}{b})^{\gamma+\epsilon}}$  nahe genug bei 1 ist, sodass die L.S. immer noch  $< cf(n)$  ist.

$\Leftrightarrow (1 + \frac{b}{n_0})^{\gamma+\epsilon} < b^\epsilon \leftarrow n_0$  groß genug wählen.

## 5.2 Zählen von Fehlständen (Inversion)

Ein Fehlstand ist ein Paar  $a_i > a_j$  mit  $i > j$ .

$(7, 3, 17, 12, 16, 20) = (a_1, \dots, a_n)$

$0 \leq \# \text{Fehlstände} \leq \binom{n}{2}$

### 5.2.1 Divide and Conquer - oder - Warum Mergesort so wichtig ist!

Fehlstände können zwischen linker und rechter Hälfte leicht bestimmt werden, wenn man die beiden sortierten Listen verschmelzt.

$(15610)(2479)$  Anzahl der Fehlstände = Anzahl der Fehlstände links + Anzahl der Fehlstände rechts

$F((a_1, \dots, a_n) \dots$  Ausgabe: Sortierte Liste  $(b_1, \dots, b_n), k$  wobei  $k = \# \text{Fehlstände}$

---

```

1 if n=0: return (a_1), 0
2 n' = \lfloor \frac{n}{2} \rfloor; n'' = n - n'
3 (b_1, \dots, b_n), F_L = F(a_1, \dots, a_{n'})
4 (c_1, \dots, c_{n''}), F_R = F(a_{n'+1}, \dots, a_n)
5 k = F_L + F_R
6 i = j = 1;
7 for l = 1, 2, \dots, n
8   if (b_i \leq c_j or j = n'' + 1) and i \leq n'
9     d_l = b_i; k = k + (j - 1)
10    i ++
11  else
12    d_l = c_j
13    j ++
14  return (d_1, \dots, d_n), k

```

---

### 5.2.2 Variante

Länge des Fehlstands ist  $j - i (a_i > a_j, j > i)$

$p$  = Gesamtlänge alle Fehlstände; wir brauchen zusätzlich zu jedem Element die Position in der ursprünglichen Liste.

Eingabe:  $a_1, \dots, a_n \dots$  Ausgabe ist  $(b_1, \dots, b_n), (q_1, \dots, q_n), k, p$

$q_i$  ist die Position von  $b_i$  in der Liste  $(a_1, \dots, a_n) \dots (q_i)$  ist eine Permutation von  $(1, \dots, n)$

Rekursive Aufrufe...

$(b_1, \dots, b_n), (q_1, \dots, q_n'), F_L, P_L = \text{rekursiv}(c_1, \dots, c_n), (r_1, \dots, r_n''), F_R, P_R =$

---

```

1 i = j = 1
2 for l = 1, \dots, n
3   if i <= n' and (j = n'' + 1 or b_i \leq c_j)
4     d_l = b_i
5     s_l = q_i
6     k = k + j - 1
7     // eckige klammer rechts neben die oberen 3 ausdrücken
8     p = p + (j - 1) (n' - q_i) + 1
9     // ende
10    i ++
11  else
12    d_l = c_j
13    s_l = r_j + n'

```

```

14    // eckige Klammer rechts neben der beiden oberen ausdrücke:
15    T = T + r_j
16    // ende
17    j ++
18    return (d_1, ..., d_n)(s_1, ..., s_n), k, p

```

---

### 5.2.3 Laufzeit

Nach Master-Theorem:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(\underbrace{n}_{n^\gamma(=)})$$

$$a = 2, b = 2, \gamma = \log_2 2 = 1$$

$$T(n) = \Theta(n \log n)$$

Oft teilt das Problem auf, dass man Größen in zwei (ungefähre) gleich große Teile zerlegen möchte, einen Teil mit den kleineren Werten, und einen Teil mit den größeren Werten.

Der **Median** ( $= \frac{n}{2}$ ) - größtes Element ist der ideale Trennungspunkt.

## 6 Median (Vorlesung 7 am 7.11.)

### 6.1 Bestimmung des k-kleinsten Elements (Medians)

Eingabe:  $(a_1, a_2, \dots, a_n)$  Liste mit Werten,  $k, 1 \leq k \leq n$

Bestimme das k-kleinste Element in sortierter Reihenfolge.

Sortierte Reihenfolge  $a^{(1)} \leq a^{(2)} \leq \dots \leq a^{(n)}$

Gesucht ist  $a^{(k)}$ ;  $k$  = Stelle in der sortierten Reihenfolge heißt der Rang des Elements

Beispiel:  $(4, 2, 1, 7, 9)$  Rang von  $a_4$  ist 3.

Das Element, das in der Mitte steht heißt der Median.

Oft hat man versucht das Element in der Mitte zu bestimmen, in dem man alle Werte aufsummiert und dann durch die Anzahl der Werte teilt. Das Ergebnis sollte dann der Mittelwert sein.

Das Problem sind allerdings Werte, die im Verhältnis zu allen anderen deutlich größer sind (bsp.  $(1, 2, 3, 4, 2, 3, 9000)$ ), weil sie den Mittelwert ungünstig verschieben, sodass er keinen Sinn ergibt.

Der **Median** kann wie folgt bestimmt werden:

für ungerade  $n$

$$a_{\frac{n+1}{2}}$$

für gerade  $n$

$$\frac{1}{2}(a_{\frac{n}{2}} + a_{\frac{n}{2}+1})$$

### 6.2 Quickselect

Algorithmus: Quickselect( $k, l$ ) mit  $l = (a_1, \dots, a_n)$

1. Wähle Pivotelement  $a$
2. Zähle, wie viele Elemente  $<, =, > a$  sind. Der Rang von  $a$  ist zwischen  $n_{<} + 1$  und  $n_{<} + n_{=}$
3. **if**  $k \leq n_{<}$  **then** Quickselect( $k, l_{\text{kleiner}}$ ), wobei  $|l_{<}| = n_{<}$  und  $l_{\text{kleiner}}$  enthält die Elemente  $< a$ .
4. **if**  $k > n_{<} + n_{=}$  **then** Quickselect( $k - n_{<} - n_{=}, l_{\text{groesser}}$ )
5. return  $a$

#### 6.2.1 Laufzeit

Laufzeit im schlimmsten Fall: Pivotelement immer das kleinste Element oder größte.

Liste wird nur um 1 kleiner in jeder Rekursion  $\rightarrow \Theta(n^2)$

Laufzeit im besten Fall: • Rang( $a$ ) =  $k$ , keine Rekursion notwendig  $\rightarrow \Theta(n)$  (GLÜCK!)

- Teilung in der Mitte:  $n_{<}, n_{>} \leq \frac{n}{2} : T(n) = T(\frac{n}{2}) + \Theta(n)$  Ein bisschen Mastertheorem:

$$T(n) = 1 * T(\frac{n}{2}) + \Theta(n)$$

$$a = 1$$

$$b = 2$$

$$f(n) = \Theta(n^1) 1 > 0$$

$$\gamma = \log_b a = 0 \rightarrow \text{Fall}(+) \Rightarrow T(n) = \Theta(f(n)) = \Theta(n)$$



Alternative (Einsetzen:)

$$T(n) = \Theta(n) + \Theta\left(\frac{n}{2}\right) + \Theta\left(\frac{n}{4}\right) \cdots = \Theta(n)$$

Der Algorithmus ist also stark davon abhängig welches Pivotelement wir wählen. Ideal wäre es den Median zu finden. Da wir aber hier versuchen den Median zu finden ist das ein Zirkelschluss. Dabei muss es nicht mal genau das Element genau in der Mitte sein, es reicht, wenn es nahe genug dran ist.

## 6.3 randomisiertes Quickselect

Wähle  $a$  zufällig aus der Liste.

Rang( $a$ ) ist gleich verteilt auf  $1, 2, \dots, n$ .

### 6.3.1 Laufzeit

Analyse der erwarteten Laufzeit:

Wir nennen den Aufruf von Quickselect erfolgreich, wenn:

$$\begin{aligned} n_{<} + n_{=} &= \frac{1}{4}n \\ n_{>} + n_{=} &= \frac{1}{4}n \end{aligned}$$

in der obersten Aufrufebeine ist.

$$\left[\frac{1}{4}n \leq \text{rang}(a) \leq \frac{3}{4}n\right]$$

wenn ( $a$ ) eindeutig ist.

Wahrscheinlichkeit(erfolgreich)  $\geq \frac{1}{2}$

Bei einem erfolgreichen Aufruf wird die Liste auf höchstens  $\frac{3}{4}n$  reduziert.

$T(n)$  = erwartete Laufzeit.

$$\begin{aligned} T(n) &\leq E(\#\text{Läufe bis zu einem erfolgreichen Lauf}) \cdot \mathcal{O}(n) + T\left(\frac{3}{4}n\right) \\ &= \frac{1}{p} \text{ wobei } p = \frac{1}{2} \text{ die Erfolgswahrscheinlichkeit ist.} \\ &= \leq 2 \end{aligned}$$

$$T(n) \leq T\left(\frac{3}{4}n\right) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n)$$

## 6.4 Quickselect nach Blum, Floyd, Pratt, Rivest, Tarjan (1973)

Deterministische Auswahl in  $\mathcal{O}(n)$  Zeit.

1. Falls  $n \leq n_0$ , sortiere
2. Andernfalls zerlege Folge in 5er-Gruppen und bestimme in jeder Gruppe den Median  $m_1, m_2, \dots, m_{\lfloor \frac{n}{5} \rfloor}$
3. Bestimme den Median  $m^*$  dieser Mediane rekursiv.
4. Wähle das Pivotelement  $a := m^*$  und verfähre weiter wie bei Quickselect.

#### 6.4.1 Laufzeit

Welche Aussagen treffen jetzt auf  $n_{<} + n_{=}$  und  $n_{>} + n_{=}$  zu?

$$n_{<} + n_{=} \geq 3 \frac{\lfloor \frac{n}{5} \rfloor}{2}$$

$$n_{>} + n_{=} \geq 3 \frac{\lfloor \frac{n}{5} \rfloor}{2}$$

$$\begin{aligned} n_{<} &= n - (n_{<} + n_{=}) \\ &= n - \frac{3}{2} * \lfloor \frac{n}{5} \rfloor \end{aligned}$$

Annahme  $n = 5l$

$$n_{<} \leq n - 0,3 = 0,7n$$

$$n = 5l + i$$

$$|i = 0, 1, 2, 3, 4 \quad l = \lfloor \frac{n}{5} \rfloor$$

$$n_{<} \leq n - \frac{3}{2}l = n - \frac{3}{2}(\frac{n-i}{5})$$

$$= n - \frac{3}{10}n + \frac{3}{10}i$$

$$\leq \frac{7}{10}n + \frac{12}{10} \leq \frac{7}{10}n + 3$$

$$n_{<} \leq \frac{7}{10}n + 3$$

Behauptung:  $T(n) = \mathcal{O}(n)$

Beweis: Annahme:

$$T(n) \leq Cn + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor 0,7n \rfloor + 3) \text{ für } n \geq 100$$

Behauptung  $T(n) \leq C'n$ , wenn  $C' \geq 20C$  ist und  $C'$  so groß ist, dass  $T(n) \leq C'n$  für  $n \geq 100$  ist.

Beweis mit vollständiger Induktion:  $n \geq 100$  geht!

für  $n > 100$ :

$$\begin{aligned}
 T(n) &\leq \mathcal{C}n + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor 0,7n \rfloor + 3) \\
 &\leq \mathcal{C}n + \mathcal{C}'\frac{n}{5} + \mathcal{C}' * 0,7n + \mathcal{C}'3 \\
 &\leq \mathcal{C}'\frac{n}{20} \\
 &\leq \mathcal{C}'n(0,05 + 0,2 + 0,7) + \mathcal{C}' \cdot 3 \\
 &= \mathcal{C}'(0,95n + 3) \leq \mathcal{C}'n \\
 0,95n + 3 &\leq n \\
 3 &\leq n \cdot 0,05 \quad (n \geq 100 \rightarrow n \cdot 0,05 \geq 5)
 \end{aligned}$$

## 7 Das Rucksackproblem (Vorlesung 8 am 10.11.)

Gegeben sind  $n$  Gegenstände.

Jeder Gegenstand hat einen Wert  $w$  und ein Gewicht  $g_i$ .

Es gibt eine Gewichtsschranke  $G$ .

### Problem

Finde eine Teilmenge mit möglichst großem Wert und Gesamtgewicht  $\leq G$

**Beispiel  $n = 5, G = 12$**

$i$	1	2	3	4	5
$g_i$	4	3	5	2	6
$w_i$	7	8	6	3	9

Formulierung mit Variablen:

$$\begin{aligned} &\text{maximiere} \quad \sum_{i=1}^n x_i w_i && |x_i \text{ gibt an, ob Gegenstand ausgewählt wird} \\ &\text{unter} \quad \sum_{i=1}^n x_i g_i \leq G \\ & && x_i \in \{0, 1\} \end{aligned}$$

→  $2^n$  Möglichkeiten.

- ganzzahliges RP:  $x_i \in \mathbb{N}$
- gebrochenes RP:  $0 \leq x_i \leq 1$

### 7.1 Lösung: Dynamisches Programmierung / Optimierung

Löse das Problem durch *systematisches Lösen von Teilproblemen*.

Große Teilprobleme werden auf kleinere zurückgeführt, die man schon vorher gelöst hat.

#### Teilprobleme?

Betrachte nur die ersten  $i$  Gegenstände.

zusätzlich: muss man das zulässige Gesamtgewicht variieren.

$f(i, b)$  = optimaler Wert mit den ersten  $i$  Gegenständen und das Gesamtgewicht  $\leq b$

$$= \max \left\{ \sum_{j=1}^i w_j x_j \mid \sum_{j=1}^i g_j x_j \leq b, x_j \in \{0, 1\} \right\}$$

$$\begin{aligned} f(i, b) &= \max \{ f(i-1, b), f(i-1, b-g_i) + w_i, \text{ falls } b \geq g_i \} \\ &= f(i-1, b), \text{ falls } g_i > b \end{aligned}$$

Lösung mit Tabelle:  $f(i, b)$  mit  $i = 0, \dots, n$  und  $b = 0, \dots, G$

$g_i$		4	3	5	2	6
$i$	0	1	2	3	4	5
$b = 0$	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	8	8	8	8
4	0	7 <sup>+</sup>	8	8	8	8
5	0	7	8	8	11	11
6	0	7	8	8 <sup>-</sup>	11	
7	0	7	15 <sup>+</sup>	15 <sup>-</sup>	15	
8	0	7	15	15	15	
9	0	7	15	15	18	
10	0	7	15	15	18	
11	0	7	15	15	18	
12	0	7	15	21 <sup>+</sup>	21 <sup>-</sup>	21 <sup>-</sup>

Mit <sup>+</sup> markierte Einträge in der Tabelle werden zur optimalen Gesamtlösung hinzugefügt.

$x_5 = 0 \rightarrow x_4 = 0 \rightarrow x_3 = 1 \rightarrow x_2 = 1 \rightarrow x_1 = 1$  Tabelle liefert  $f(5, 12) = 21 = f(n, G)$  den Wert der Optimallösung.

Um die Lösung selbst zu finden, müssen wir zurückverfolgen, wie dieser Wert zustande gekommen ist.

### Zurückverfolgen der Lösung

- man merkt sich bloß die Tabelle und rechnet beim Zurückgehen jeden Eintrag noch einmal nach. (Programmieraufwand)
- man speichert sich schon beim Berechnen Zusatzinformationen, wie der Wert zustande gekommen ist. (viel zusätzlicher Speicheraufwand)

#### 7.1.1 Laufzeit und Speicherbedarf

$\Theta(nG)$  = Größe der Tabelle = Speicherbedarf

Der Speicher lässt sich auf  $\Theta(G)$  reduzieren (allerdings verliert man die Möglichkeit der Rücknachvollziehbarkeit)

### 7.2 Dynamische Programmierung

- Definition der Teilprobleme  
nicht eindeutig vorgegeben.
- Rekursion (+ Anfangsbedingungen)  
Variante mit Gesamtgewicht =  $b$   
( $f(i, b) = -\infty$  falls es keine Lösung gibt.)  
(Rekursion bleibt unverändert, Anfangsbedingung ändert sich. Optimallösung in der ganzen Spalte suche)
- systematisches Ausfüllen (Zeilen- oder Spaltenweise) der Tabelle aller Teilprobleme
- Rückverfolgen der Lösung

### 7.3 Die Tabelle als Netzwerk

Betrachte die Tabelle als gerichteten Graphen. Jeder Eintrag = 1 Knoten.

Vorgänger = Einträge, von denen der Knoten abhängt.

Kantengewicht = Wert, der in Rekursion addiert und Knotenwert =  $f(i, b)$  = Längster Weg von der linken oberen Ecke  $(0, 0)$  zum Knoten  $(i, b)$ .

Sehr oft lässt sich eine DP-Rekursion als Wegeproblem in einem azyklischen Graphen modellieren. (kürzeste / längste Wege von einer Ecke zur anderen)

### 7.4 Speicheroptimierung

Der Speicher lässt sich optimieren(?) in dem man einen Faktor  $\log n$  zur Laufzeit hinzufügt. (unklar...)

## 8 Dynamische Programmierung (Fortsetzung) (Vorlesung 9 am 14.11.)

### 8.1 Gewichtete Intervallauswahl

Gegeben:  $n$  Intervalle  $[a_1, b_1), [a_2, b_2), \dots, [a_n, b_n)$  mit Gewichten  $w_1, \dots, w_n$

Gesucht: Disjunkte Intervalle mit dem größten Gesamtgewicht.

Teilprobleme: Betrachte nun Intervalle, die in  $(-\infty, x)$  enthalten sind,  $x \in \mathbb{R}$ . Für  $x$  reicht es, die Intervallendpunkte zu betrachten.

#### 8.1.1 Vorverarbeitung (Normalisierung)

Sortiere alle Intervallendpunkte, ändere die vorkommenden Werte in  $1, 2, 3, \dots, 2n, \Rightarrow x \in \{0, 1, 2, \dots, 2n\}$ . Das Problem wird dadurch nicht verändert.

$f(i)$  = größtes Gewicht einer Menge disjunkter Intervalle die in  $(-\infty, i)$  enthalten sind.

$f(i) = \max\{f(i-1), \max\{f(a_k) + w_k \mid \text{Intervalle } k \text{ mit } b_k = i\}\}$

$f(0) = 0$

Berechne  $f(1), f(2), \dots, f(n)$  mit der Rekursionsformel.  $f(m)$  ist die Optimale Lösung.

#### 8.1.2 Laufzeit

Jedes Intervall  $[a_k, b_k)$  kommt genau 1x in der Rekursion auf der rechten Seite vor.

#### 8.1.3 Algorithmus

1. ( $\mathcal{O}(n \log n)$ ) Sortieren und Umnummerieren der Endpunkte

2. ( $\mathcal{O}(n)$ ) Erstelle für  $i = 1, \dots, m$  eine Liste  $L_i$  der Intervalle  $k$  mit  $b_k = i$ , Initialisiere alle  $L_i = \emptyset$

---

```
1 for k = 1, .. n
2   L_b_k.append(k)
```

---

3.) Rekursion:

---

```
1 f(i) := max(f(i-1), max(f(a_k)+w_k) | k \in L_i)
```

---

$f$  wird in einem Feld gespeichert.

## 8.2 Rundreiseproblem (Traveling Salesperson Problem[TSP])

Gegeben ist ein gerichteter Graph mit Kantengewichten.

Gesucht ist ein Kreis, der jeden Knoten genau einmal besucht und geringste Gesamtlänge hat (Hamiltonkreis).

Mögliche Lösungen: Startknoten beliebig fixieren.

$(n-1)(n-2)(n-3) * \dots * 2 * 1 = (n-1)!$

falls der Graph vollständig ist.

Teilprobleme  $(T, i) \mid T \subseteq \{1, \dots, n\}, i \in T, 1 \in T$

$f(T, i)$  = der kürzeste Weg von 1 nach  $i$  der genau die Knoten in  $T$  besucht.

### 8.2.1 Rekursion

$$f(T, i) = \min_{j \in T - \{i\}, j, i \in E, j \neq 1} (f(T - \{i\}, j) c_{ji}) \quad , \text{ für } |T| \geq 3$$
$$f(\{1, i\}, i) = c_{1i} (\text{bzw. } \infty, \text{ falls } 1i \notin E)$$
$$\text{OPT} = \min_{j \neq 1, jn \in E} (f(\{1, \dots, n\}, j) + c_{jn})$$

Wieviele Teilprobleme gibt es?

$$\# \text{Teilprobleme} \leq 2^n \cdot n$$

$2^n - 1$  Teilmengen  $T$

Zu  $T$  gibt es  $|T| - 1$  Teilprobleme  $(T, i)$  ( $\sum_{k=1}^n \binom{n}{k} (k-1)$ )

Jedes Teilproblem benötigt  $\mathcal{O}(n)$  Zeit (eigentlich  $\mathcal{O}(|T|)$ )

Insgesamt  $\mathcal{O}(2^n n^2)$  Laufzeit

Speicher  $\mathcal{O}(2^n n)$

(exponentiell viel besser als  $\mathcal{O}((n-1)!)$ )

### 8.2.2 1. Möglichkeit

Tabelle mit  $2^{n-1} \times n$  Einträgen. Teilprobleme werden z.B. nicht wachsendem  $|T|$  gelöst. (Andere Möglichkeit:  $T$  als  $(n-1)$ -stellige Binärzahl darstellen, in numerischer Reihenfolge lösen.)

Wichtig:  $T \leq S$  und  $T$  vor  $S$  lösen.

### 2. Möglichkeit

Tabellieren (Memoization)

Top-down-Berechnung rekursiv nach Bedarf mit Speicher, der schon berechneten Ergebnisse.

Initialisieren der Tabelle  $M$  auf -1 (Annahme  $c_{ij} \geq 0$ )

---

```
1 def f (T,i):
2   if M[T,i] != -1: return M[T,i]
3   berechne E = f(T,i) nach der Rekursionsgleichung rekursiv.
4   M[T,i] = E
5   return E
```

---

Man kann sich überlegen, dass genau die Teilprobleme gelöst und gespeichert werden, für die es einen Weg von  $i$  nach 1 gibt, der gewanderte Knoten  $\{1, \dots, n\} - T$  als Zwischenknoten besucht. Wenn der Graph wenige Knoten enthält, dann können das viel weniger als  $2^n n$  Teilprobleme sein.

### Verwendung einer Hashtabelle für M

In der Praxis sind RRP mit bis zu 10.000 Ständen bis zur Optimalität lösbar und größere genügend gut approximierbar. Ein Ansatz ist branch-and-bound (Systematisches Durchsuchen von Lösungsbäumen)



## 9 Dynamische Programmierung (Fortsetzung) (Vorlesung 10 am 17.11.)

### 9.1 Optimale Triangulierung eines konvexen Polygons

$P = (p_1, p_2, \dots, p_n) | p_i \in \mathbb{R}^2$  ein Polygon in der Ebene. (konvex: Alle Innenwinkel  $< 180^\circ$ )

#### 9.1.1 Triangulierung

Zerlegung einer Polygonfläche in Dreiecke durch Diagonale Strecken  $p_i p_j$ , die im inneren verlaufen. Diagonalen dürfen sich nicht kreuzen. (erster Vorverarbeitungsschritt für viele geometrische Algorithmen)

kürzeste Triangulierung = kleinste Gesamtlänge aller Diagonalen

**Teilprobleme:**  $f_{ij}$  = kürzeste Triangulierung des Polygons  $p_i, p_{i+1}, \dots, p_j$  für  $1 \leq i < j \leq n$

Also...  $j = i + 2 \dots$  Dreieck.

$j = i + 1 \dots$  Zweieck?!

**Rekursion!**

$f_{ij} \rightarrow$  besteht aus einem Dreieck  $p_i p_k p_j$   $|i < k < j$

optimale Triangulierungen  $(p_i, \dots, p_k)$  und  $(p_k, \dots, p_j)$

$$f_{ij} = \min \{ f_{ik} + f_{kj} + ||p_i - p_k|| + ||p_k - p_j|| \mid i < k < j \}$$

$$1 \leq j, j \leq n, j \geq i + 2$$

Damit die Formel auch für  $k = i + 1$  oder  $k = j - 1$  stimmt, müssen wir  $f_{i,i+1} = -||p_i - p_{i+1}||$  setzen.

Beispiel:  $f_{3,5} = f_{34} + f_{45} + ||p_3 - p_4|| + ||p_4 - p_5|| = 0$  Endergebnis =  $f_{1n}$

#### 9.1.2 Laufzeit

$\binom{n}{2} = \mathcal{O}(n^2)$  Teilprobleme, jedes Teilproblem benötigt  $\mathcal{O}(n)$  Zeit.

$\rightarrow \mathcal{O}(n^3)$  Laufzeit,  $\mathcal{O}(n^2)$  Speicher.

### 9.2 Isotone Regression

Zum Vergleich lineare Regression. Messwerte  $(x_i, y_i)$  gesucht ist eine Gerade  $y = ax + b$ , sodass

$$\sum_{i=1}^n |y_i(ax_i + b)| \text{ oder } \sum (y_i - (ax_i + b))^2$$

Gegeben ist eine Folge von Messwerten:  $a_1, a_2, \dots, a_n \in \mathbb{R}$

Gesucht ist eine monoton wachsende Folge:  $x_1 \leq x_2 \leq \dots \leq x_n$ , die  $\sum_{i=1}^n w_i * |x_i - a_i|$  minimiert. ( $w_i > 0$  sind gewichtete Daten.)

#### 9.2.1 Teilprobleme

$$f_k(z) = \min \left\{ \sum_{i=1}^k w_i |x_i - a_i| : x_1 \leq x_2 \leq \dots \leq x_k = z \right\} \quad k = 0, \dots, n; z \in \mathbb{R}$$

$$f_k(z) = \min \{ f_{k-1}(x) : x \leq z \} + w_k |z - a_k| \quad k \geq 1, z \in \mathbb{R}$$

$$f_0(z) = 0 \quad \text{für alle } z$$

## Beispiel

$$\begin{aligned}a_1 &= 5, w_1 = 1, 3 \\f_1(z) &= \min\{f_0(x)|x \leq z\} + 1.3|z - 5| \\f_2(z) &= \underbrace{\min\{f_1(x)|x \leq z\}}_{g_1(z)} + 0.7 * |z - 1| \\g_2(z) &= \min\{f_1(x)|x \leq z\}\end{aligned}$$

## Lemma

- (a)  $f_k(z)$  ist eine stückweise lineare konvexe Funktion
- (b) Die Knicke liegen an einer Teilmenge der Eingabewerte  $a_1, \dots, a_k$
- (c) Die Steigung des ersten Stücks ist  $-w_1 - w_2 - \dots - w_k$
- (d) Die Steigung des letzten Stücks ist  $w_k$

Beweis durch Induktion.

Wie kommt man von  $f_{k-1}$  auf  $g_{k-1}$ ? Lösche alle aufsteigende Stücke und ersetze sie durch ein horizontales Stück.

Möglichkeiten der Speicherung einer stückweise linearen Funktion: Koordinaten der Knicke + Steigung des ersten und letzten Astes  $\rightarrow \mathcal{O}(n)$  Werte.

$\rightarrow$  Addition zwei solcher Funktionen:  $\mathcal{O}(n)$

### 9.2.2 Optimallösung

Wie bestimmt man die Optimallösung?

Das Minimum von  $f_{k-1}(z)$  sei an der Stelle  $p_{k-1}$

Optimalwert  $x_{k-1}^*$ , wenn  $x_k^*$  gegeben ist.

$$x_{k-1}^* = \min\{p_k, x_k^*\}$$

## 10 Dynamische Programmierung (Fortsetzung) (Vorlesung 11 am 21.11.)

### Nachtrag: Isotone Regression

$$f_k(z) = \min \left\{ \sum_{i=1}^k w_i |x_i - a_i| : x_1 \leq x_2 \leq \dots \leq x_{k-1} \leq x_k = z \right\}$$
$$f_k(z) = \min \left\{ \underbrace{f_{k-1}(\underbrace{x}_{x_{k-1}}) + w_k |a_k - z|}_{g_{k-1}(z)} \right\}$$

$g_{k-1}(z)$ : 2 Fälle:

$z \leq p_{k-1} \Rightarrow g_{k-1}(z) = f_{k-1}(z)$ ; der optimale Wert von  $x = x_{k-1}$  bei gegebenem Wert von  $z = x_k$  ist  $z$  selbst.

$z \geq p_{k-1} \Rightarrow$  der optimale Wert von  $x$  ist  $p_{k-1}$   $g_{k-1}(z) = f_{k-1}(p_{k-1})$

Wenn  $x_k^*$  der optimale Wert von  $x_k$  ist, dann ist der optimale Wert von  $x_{k-1}$ :

$$x_{k-1}^* = \min\{x_k^*, p_{k-1}\}$$

Darstellung einer stückweise linearen stetigen Funktion durch Differenzen von Steigungen.

Knick an der Stelle  $x_0$  mit Knickwert  $s' - s$

Bei Addition einer linearen Funktion bleibt der Knickwert unverändert! (und Knickstelle)

Die Funktion wird durch eine Folge von Knickwerten dargestellt. Jeder Knick ist ein Paar (Knickstelle, Knickwert)

Darstellung ...

### 10.1 Algorithmus

Übergang von  $f_{k-1}$  zu  $g_{k-1}$ :

$(x, \Delta)$  sei der rechteste Knick:

---

```
1 while s - \Delta \geq 0:
2   s := s - \Delta
3   lösche den rechtesten Knick
4   p_{k-1} := x (min von f_{k-1} gefunden)
5   neuen Knickwert des rechtesten Knicks = \Delta - s
6   s := 0
```

---

Übergang von  $g_{k-1}$  zu  $f_k$ :

---

```
1 Füge einen zusätzlichen Knick (a_k, 2w_k) ein.
```

---

Die Knicke können als Prioritätswarteschlange  $Q$  gespeichert werden. nach Schlüsselwert  $x$  geordnet.

---

```
1 Q = empty, s=0
2 for k = 1, ..., n:
3   Q.insert((a_k, 2w_k))
4   s = s + w_k
5   (x, \Delta) := Q.findmax()
6   while s - \Delta \geq 0:
7     s = s - \Delta
8     Q.deletemax()
```

```

9      (x,\Delta) := Q.findmax()
10     p_k = x
11     ersetze \Delta in Q.findmax() durch \Delta-s
12     s = 0
13 (Berechnung der Optimallösung)
14 x_n = p_n
15 for k = n-1, n-2, ... , 1
16     x_k = min(x_{k-1}, p_k)

```

---

## 11 Der optimale Suchbaum

Schlüssel:

```

1  AARON, p_1
2  KLAUS, p_3
3  DIETER, p_2

```

---

Mögliche Suchbäume:

Schlüssel  $x_1 < x_2 < \dots x_n$

Anfragehäufigkeiten  $p_1, p_2, \dots, p_n$  für die Schlüssel

und  $q_0, q_1, \dots, q_n$  für die Intervalle zwischen den Schlüsseln

Mittlere gewichtete Weglänge:

$$= \sum_{i=1}^n p_i \underbrace{(\text{Tiefe des Knotens mit Schlüssel } i)}_{\# \text{ Vergleiche} - 1} + \sum_{i=0}^n q_i \underbrace{(\text{Tiefe des Blattes, dass dem entsprechenden Intervall entspricht})}_{\# \text{ Vergleiche}}$$

soll minimiert werden.

Ansatz:  $q_i$  entspricht dem Intervall  $(x_i, x_{i+1})$   $x_0 = -\infty, x_{n+1} = +\infty$

Ein Teilbaum in einem Suchbaum entspricht einem Intervall  $(x_i, x_j)$  mit  $0 \leq i \leq j \leq n+1$

(Der Baum wird genau dann betreten, wenn der gesuchte Schlüssel in diesem Intervall liegt.)

### 11.1 Teilprobleme

$f(i, j)$   $0 \leq i \leq j \leq n+1$  optimaler Suchbaum für Schlüssel  $x_{i+1}, \dots, x_{j-1}$  und Häufigkeiten  $p_{i+1}, \dots, p_{j-1}$

### 11.2 Rekursion

$$f(i, j) = \min\{f(i, k) + f(k+1, j)\} + q_i + q_{i+1} + \dots + q_{j-1} + p_{i+1} + p_{i+2} + \dots + p_{j-1} - p_k \quad | i+1 \leq k \leq j-1, 0 \leq i, j \leq n+1, j \geq i+1$$

### 11.3 Anfangswerte

$$f(i, i+1) = 0, i = 0, \dots, n$$

$$f(2, 4) = f(2, 3) + f(3, 4) + q_2 + q_3 + p_3 - p_3$$

## 11.4 Gesamtlösung

$$f(0, n + 1)$$

## 11.5 Laufzeit

$\mathcal{O}(n^2)$  Teilprobleme.

$\mathcal{O}(n)$  Ausdrücke, über die minimiert wird.

Die Summe  $q_1 + \dots + p_{i-1}$  ist fest, für jedes Teilproblem nur einmal ausrechnen  $\rightarrow \mathcal{O}(n)$  Zeit.

$\Rightarrow \mathcal{O}(n^3)$  Zeit,  $\mathcal{O}(n^2)$ .

## 12 Dynamische Programmierung (Einen hab ich noch!)

12 am 24.11.)

(Vorlesung

### 12.1 Dynamische Programmierung - eine Zusammenfassung

Bisher gelöste Probleme mit dym. Prog.:

1. Rucksackproblem
2. optimale Triangulierung  $\mathcal{O}(n^3)$
3. Suchbaum  $\mathcal{O}(n^3)$
4. CYK (zweites Semester GTI) - Coche, Younger, Kasamy  
Eine Grammatik ist in Chomsky Normalform (CNF), wenn jede jede Formel einer Grammatik in ein Terminalsymbol "mündet".  
Ziel ist es für ein gegebenes Wort zu prüfen, ob es von der gegebenen CNF Grammatik erzeugt werden kann.  
$$f(i, j) = \{v | v \rightarrow^s x a_i, \dots, a_j\} | a_i, a_j \in \Sigma^*$$

### 12.2 Editierabstand

Wir nehmen an, dass wir folgendes Wort auf der Tastatur getippt haben:

A g o r h y t m u s

Aber eigentlich wollten wir Pfannku... ähm **Algorithmus** schreiben.

Wie kommen wir jetzt von Agorhytmus zu Algorithmus?

#### 12.2.1 Problem

**Gegeben:** zwei Wörter  $A = a_1, \dots, a_m$  und  $B = b_1, \dots, b_n$  aus  $\Sigma^*$

Wie können wir  $A$  in  $B$  durch folgende Operationen verwandeln?

1. löschen eines Buchstabens (Kosten  $k_L$ )
2. einfügen eines Buchstabens ( $k_E$ )
3. Buchstabe  $u$  durch  $x$  ersetzen ( $\delta_{ux}$ )

**Gesucht** ist die billigste Folge von Operationen wie  $A$  in  $B$  umgewandelt wird.

Die **Kosten**  $k_L, k_E, \delta_{ux}$  sind der **Editierabstand**.

Vor allem in der Bioinformatik ist das ein sehr wichtiges Problem auf sehr großen Datenmengen.

#### 12.2.2 Teilprobleme

$f(i, j)$  = Editierabstand zwischen  $a_1, \dots, a_i$  und  $b_1, \dots, b_j$

#### 12.2.3 Rekursion

$$f(i, j) = \min\{f(i-1, j) + k_L, f(i, j-1) + k_E, f(i-1, j-1) + \delta_{a_i, b_j}\} \quad | 1 \leq i \leq m, 1 \leq j \leq n$$

#### 12.2.4 Startwerte

Konvention:  $\delta_{xx} = 0 \forall x \in \Sigma$

$$f(i, 0) = f(i - 1, 0) + k_L = i * k_L$$

$$f(0, 0) = 0$$

$$f(0, j) = j k_E$$

#### 12.2.5 Graph

Graph mit  $(m+1)(n+1)$  Knoten. Jeder Knoten hat 3 Vorgänger (außer am Rand). Editierabstand ist der kürzeste Weg von  $(0, 0)$  zu  $(m, n)$

#### 12.2.6 Annahmen

Zwei aufeinanderfolgende Änderungen lohnen sich nicht:  $x \rightarrow y \rightarrow u$  (Dreiecksungleichung  $x$ :  $\delta_{xy} + \delta_{yu} \geq \delta_{xu}$ )

Löschen und Einfügen statt Ändern ist in diesem Algorithmus vorgesehen. Falls  $\delta_{xy} \leq k_L + k_E$

#### 12.2.7 Speicherreduktion

auf  $\mathcal{O}(m+n)$  durch divide & conquer.

1. Zur Berechnung der Kosten  $f(m, n)$  ist nun  $\mathcal{O}(m+n)$  Speicher notwendig: Es genügt, zwei aufeinanderfolgende Spalten im Speicher zu halten.
2. Abstände zum Zielknoten:

$$g(i, j) = \min\{g(i+1, j) + k_L, g(i, j+1) + k_E, g(i+1, j+1) + \delta_{a_{i+1}, b_{j+1}}\}$$

3. Optimallösung =

$$\min\{f(i, \lfloor \frac{n}{2} \rfloor), g(i, \lfloor \frac{n}{2} \rfloor) | i = 0..m\}$$

4. Es sei  $i_0$  das Optimum in 3.

Bestimme rekursiv den kürzesten Weg von  $(0, 0)$  zu  $(i_0, \lfloor \frac{n}{2} \rfloor)$  und von  $(i_0, \lfloor \frac{n}{2} \rfloor)$  zu  $(m, n)$

$\Rightarrow$  Speicherbedarf  $\mathcal{O}(m+n)$

#### 12.3 Laufzeit (inkl. Speicherreduktion)

Rekursion:  $T(m, n) = \mathcal{O}(m, n) + T(i_0, \lfloor \frac{n}{2} \rfloor) + T(m - i_0, n - \lfloor \frac{n}{2} \rfloor)$

$n$  sei eine zweier Potenz:

$$\begin{aligned}
T(n, m) &\leq cmn + T(i, \frac{n}{2}) + T(m - i, \frac{n}{2}) \\
&\leq cmn + \min\{T(i, \frac{n}{2}) + T(m - i, \frac{n}{2}) | 0 \leq i \leq m\}
\end{aligned}$$

Behauptung:  $T(m, n) \leq c'mn$ ; Beweis durch Induktion

Einsetzen: R.S.

$$\begin{aligned}
 & cmn + c'i * \frac{n}{2} + c'(m-i)\frac{n}{2} \\
 & = cmn + \frac{c'}{2}mn \leq c'mn
 \end{aligned}$$

wähle  $c' = 2c$ , dann gehts.

## 13 Gierige Algorithmen

Optimierungsalgorithmen, die eine Folge von Entscheidungen kurzfristig treffen und später nicht mehr rückgängig machen.

### 13.1 Beispiel Rucksack

Rucksackproblem  $g_1, \dots, g_n$  mit  $w_1, \dots, w_n$

1. Wählt die wertvollsten Gegenstände zuerst, bis Rucksack voll ist.

**NICHT OPTIMAL!**

$g_i$	10	1	1	...	10
$w_i$	100	99	99	...	

2. Sortiere Gegenstände  $\frac{w_1}{g_1} \geq \dots \geq \frac{w_n}{g_n}$  und wähle gierig in dieser Reihenfolge. **NICHT**

**OPTIMAL!**

$g_i$	8	5	5	...	$G = 10$
$w_i$	9	5	5	...	
$\frac{w_i}{g_i}$	$\frac{9}{8}$	1	1		

Bemerkung: Algorithmus 2. ist optimal für das gebrochene Rucksackproblem!

Wir sehen, dass Greedy-Algorithmen nicht immer die beste Lösung für gewisse Probleme sind.

### 13.2 Ungewichtete Intervallauswahl

$[a_1, b_1) \dots [a_n, b_n)$ , keine Gewichte  $w_i$

Wähle eine möglichst große **Anzahl** von undisjunkten Intervallen aus.

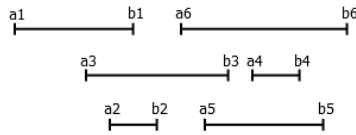
1. Wähle kürzeste Intervalle zuerst. (NICHT OPTIMAL)
2. Intervalle von links nach rechts. (NICHT OPTIMAL)
3. Intervall, das am wenigsten andere überlappt.
4. sortiert nach Endzeitpunkt  $b_i$



## 14 Greedy Algorithmen (Vorlesung 13 am 28.11.)

### 14.1 Intervallauswahl nach Endzeitpunkten

Greedy-Algorithmus Sortiere die Intervalle nach Endzeitpunkt  $b_1 \leq b_2 \leq \dots \leq b_n$

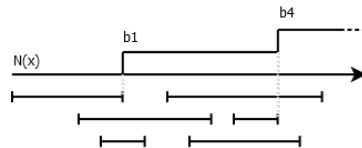


$r = -\infty$  für  $i = 1, \dots, n$

Wenn Intervalle  $[a_i, b_i]$  keine ausgewähltes Intervall überlappt (if  $a_i \geq r$ ), wähle es aus ( $r = b_i$ ).

Wir müssen uns den rechtesten Punkt merken, um zur optimalen Lösung zu kommen...  $r$  = der rechte Endpunkt der bisher ausgewählten Intervalle.

#### 14.1.1 Beweis



$\mathcal{N}(x) = \#$ der gewählten Intervalle innerhalb  $(-\infty, x)$  bei der Greedy-Lösung.

$\mathcal{N}^*(x) = \#$ der gewählten Intervalle bei einer beliebigen anderen Lösung.

Wir wollen zeigen, dass  $\mathcal{N} \geq \mathcal{N}^*$  für alle  $x$ .  $\mathcal{N}(\infty) \geq \mathcal{N}^*(\infty)$

$\mathcal{N}(x)$  kann sich nun an einem Punkt  $b_i$  ändern.

Beweis mit Induktion nach  $i$ .  $\mathcal{N}(b_i) \geq \mathcal{N}^*(b_i)$

Induktionsbehauptung:  $\mathcal{N}(-\infty) = \mathcal{N}^*(-\infty) = 0$

Annahme:  $\mathcal{N}^*(x)$  macht an der Stelle  $b_i$  einen Sprung:  $\mathcal{N}^*(b_i) = \mathcal{N}^*(b_{i-1}) + 1$ .

Die andere Lösung wählt das Intervall  $[a_i, b_i]$  aus.

$\mathcal{N}^*(b_i) = \mathcal{N}^*(a_i) + 1 \leq \mathcal{N}(a_i) + 1$

Im Intervall  $[a_i, b_i]$  hat der Greedy Algorithmus ebenfalls ein Intervall gewählt, das zu den bisherigen Intervallen disjunkt ist. (spätestens das Intervall  $[a_i, b_i]$  ist so ein Kandidat.  $\Rightarrow \mathcal{N}(b_i) \geq \mathcal{N}(a_i) + 1 \leq \mathcal{N}^*(b_i)$ )

Annahme (Fall 2):  $\mathcal{N}^*(x)$  macht keinen Sprung bei  $b_i$

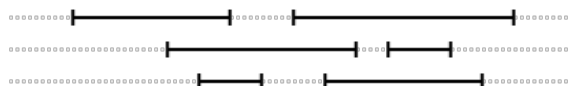
Im Vergleich zum Algorithmus der dynamischen Programmierung werden hier Vereinfachungen vorgenommen.

### 14.2 Variante

Wir müssen alle Intervalle akzeptieren und sie verschiedenen Maschinen zuordnen, wenn sie sich überlappen.

Gesucht ist die minimale Anzahl von Maschinen.

Beispiel: Es sind 3 Maschinen notwendig.

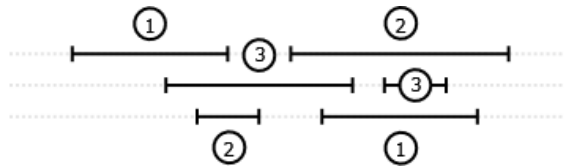


Untere Schranke, Anzahl Intervalle die einem gemeinsamen Punkt  $x$  enthalten.

### 14.3 Greedy Algorithmus

Betrachte die Intervalle aufsteigend vom Startpunkt  $a_1 \leq a_2 \leq \dots \leq a_n$

Ordne jedes Intervall  $[a_i, b_i)$  der Maschine mit der kleinsten Nummer  $j$  zu, die frei ist.



#### 14.3.1 Beweis

Behauptung: Wenn die Maschine  $j$  zugeordnet wird, ist der Punkt  $a_i$  in mindestens  $j$  Intervallen enthalten.

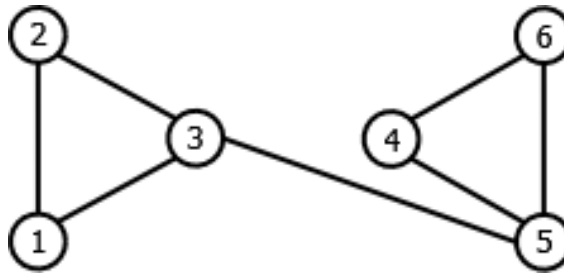
Die Maschinen  $1, 2, \dots, j-1$  sind belegt und dabei in anderen Intervallen enthalten. plus Intervall  $[a_i, b_i)$

### 14.4 Interpretation als Graphenproblem

Intervalle  $\rightarrow$  Knoten

überlappende Intervalle  $\rightarrow$  Kanten

Der entstehende Graph ist ein Intervallgraph. Ein Durchschnittsgraph von Intervallen. Teilmenge



von Intervallen, die sich nicht überlappen  $\rightarrow$  unabhängige (Knoten-)Menge.

Überlappungsfreie Zuordnung von Maschinen  $\rightarrow$  Graphenfärbung (chromatische Zahl  $\psi$ )

Punkt  $x$ , der in mehreren Intervallen enthalten ist  $\rightarrow$  Clique. (vollständiger Teilgraph)

$\leftarrow$  gilt in Intervallgraphen aber nicht in allgemeinen Durchschnittsgraphen.

Die Cliquenzahl (Größe der größten Clique) nennen wir  $\omega$

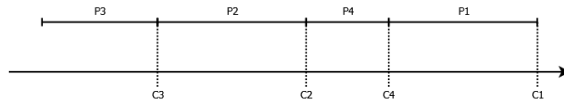
Ausserdem gilt für Intervallgraphen  $\omega = \psi$  - sonst gilt im allgemeinen Durchschnittsgraphen  $\omega \leq \psi$

### 14.5 Zeitplanung(Scheduling)

Man hat  $n$ -Aufträge und jeder Auftrag hat eine Bearbeitungszeit  $p_i$  und einen Termin  $d_i$  (deadline, due date) und die Aufträge müssen jetzt sequentiell abgearbeitet werden.

Gesucht ist eine Reihenfolge in der die Aufträge bearbeitet werden.

Aus der Reihenfolge ergibt sich anschließend eine Abschlusszeit  $C_i$ , wann der Auftrag fertig ist.

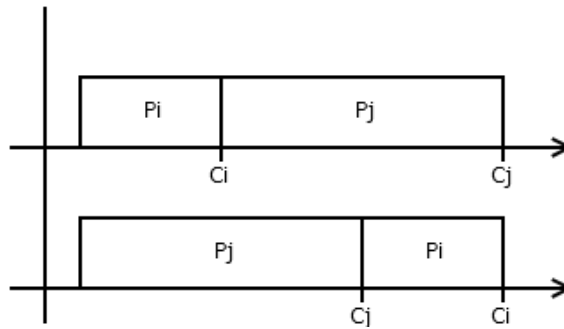


Die Verspätung  $L_i = \max\{C_i, d_i, 0\}$

Wir wollen die maximale Verspätung  $\max\{L_i | i = 1, \dots, n\}$  minimieren.

Was passiert, wenn man zwei benachbarte Aufträge vertauscht.

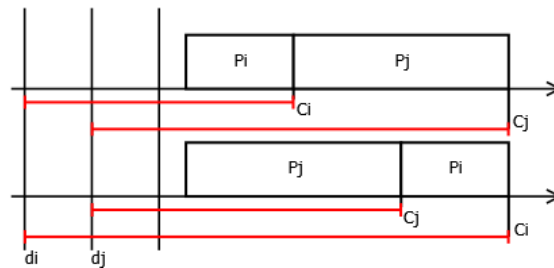
Wir vergleichen:  $\max\{C_j, C_i - d_i, 0\}$



$\max\{C'_j, C'_i - d_i, 0\}$

Es gilt:  $C_j = C'_i > C_i, C'_j$

Annahme, beide p's sind verspätet... Behauptung: wenn  $d_i \leq d_j$  ist, dann ist die Reihenfolge  $ij$



mindestens so gut wie die Reihenfolge  $ji$ .

$\max\{C_{max} - d_j, C_i - d_i\} \leq \max\{C_{max} - d_i, C'_j - d_j\}$

Wissen:  $-d_j \leq -d_i$

#### 14.5.1 Beweis

$$C_{max} - d_j \leq C_{max} - d_i$$

$$C_i - d_i \leq C_{max} - d_i$$

$$\max\{C_{max} - d_j, C_i - d_i\} \leq C_{max} - d_i \leq \max\{C_{max} - d_i, C'_j - d_j\}$$

#### 14.6 EDD-rule (earliest due date rule)

Bearbeite die Aufträge in der Reihenfolge der Termine  $d_i$ .

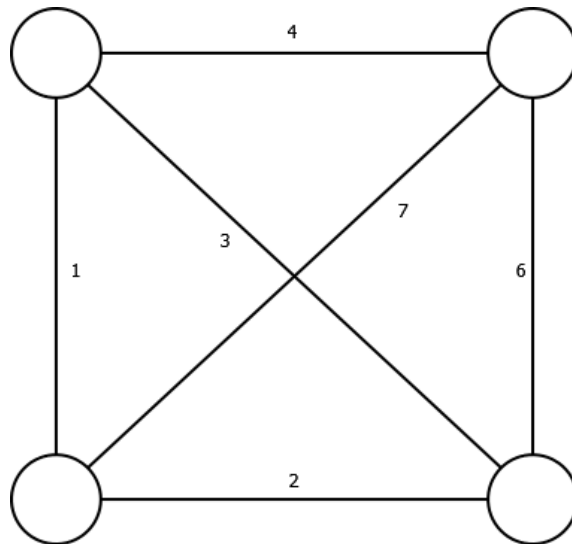
Beweis der Optimalität durch ein Austauschargument.

## 15 Der Klassiker für Greedy Algorithmen: minimal SPT

Gegeben ist ein zusammenhängender ungerichteter Graph mit Kantengewichten  $\geq 0$ .

Gesucht ist ein Spannbaum, der alle Knoten enthält mit kleinstem Gesamtgewicht.

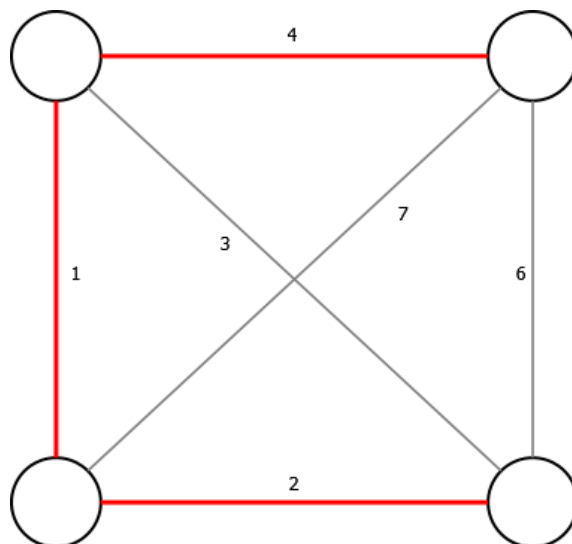
Beispiel: siehe Bild:



### 15.1 Algorithmus von Kruskal

Betrachte die Kanten in der Reihenfolge nach Gewicht. Wähle die Kante aus, wenn sie mit den bisher gewählten Kanten keinen Kreis bildet.

Dieser Algorithmus hat zu einem grundlegenden Umdenken und weitreichenden Verallgemei-



nerungen geführt. Makroide. Wenn man eine beliebige Matrix nimmt und man betrachtet die unabhängigen Spaltenmengen, dann bilden die ein Makroid. Diese bilden dann eine Basis der Matrix.

## 16 Kürzeste Wege (Vorlesung 14 am 01.12.)

### 16.1 Wiederholung: Dijkstras Methode

Eingabe: gerichteter Graph  $G = (V, E)$  mit  $m$  Kanten und  $n$  Knoten und Kantengewichten  $c_{uv}$  für  $u, v \in E$ .

#### 16.1.1 Algorithmus nach Dijkstra (1960-1961)

$$c_{uv} \geq 0$$

kürzester Weg von einem Startknoten zu allen anderen Knoten.

#### Laufzeiten

Laufzeiten optimieren sich hauptsächlich mit der verwendeten Speicherstruktur.

Methode	Laufzeit
primitiv	$\mathcal{O}(n^2)$
Halde (heap) als Prioritätswarteschlange	$\mathcal{O}((m+n) \log n)$
Fibonacci-Halde	$\mathcal{O}(m + n \log n)$

### 16.2 Algorithmus von Bellman / Ford

$c_{uv} \in \mathbb{R}$  beliebig.

ein Startknoten.

Laufzeit  $\mathcal{O}(nm)$

#### Algorithmus

$d_i^*$  = kürzeste Weglänge von Startknoten  $s$  zu Knoten  $i$  mit höchstens  $k$  Knoten.

$k = 0, 1, 2, \dots$

#### 16.2.1 Rekursion

$$d_j^k = \min\{d_i^{k-1} + c_{ij} \mid i \in V, ij \in E\} \cup \{d_j^{k-1}\}$$

#### 16.2.2 Anfangswerte

$$d_j^{(0)} = \begin{cases} 0, & j = s \\ \infty, & \text{sonst} \end{cases}$$

#### 16.2.3 Beispiel

j,k	0	1	2	3	4
1	0	0	0	0	0
2	$+\infty$	-1	-1	-1	-1
3	$+\infty$	6	1	1	1
4	$+\infty$	5	3	-2	-2

#### 16.2.4 Bemerkungen

Wenn der Graph keine negativen Kreise enthält, dann besucht ein kürzester Weg keinen Knoten mehrfach und hat somit  $n - 1$  Kanten.

$\Rightarrow d_j^{n-1}$  sind die kürzesten Weglängen.

Vektor  $d^{(k)} = \begin{pmatrix} d_1^{(k)} \\ \vdots \\ d_n^{(k)} \end{pmatrix}$ , Rekursion hat die Gestalt  $d^{(k)} = F(d^{(k-1)})$ ,  $d^{(k)} \leq d^{(k-1)}$

Sobald  $d^{(k-1)} = d^{(k)}$  ist, kann man abbrechen:

$F(d^{(k-1)}) = F(d^{(k)}) \Rightarrow d^{(k)} = d^{(k+1)} \Rightarrow d^{(k+2)} = d^{(k+3)} = \dots$

Wenn es einen negativen Kreis gibt, dann kann die Weglänge beliebig klein werden, indem man diesen Kreis beliebig oft durchläuft.

(Sofern der Kreis vom Startknoten erreichbar ist.)

Falls  $d^{(n-1)} \neq d^{(n)}$ , dann muss es einen negativen Kreis geben.  $\rightarrow$  ABBRUCH.

**Satz:** Der Algorithmus von Bellmann-Ford bestimmt in  $\mathcal{O}(nm)$  Zeit die kürzesten Wege von einem Startknoten zu allen anderen Knoten oder er stellt fest, dass der Graph einen negativen Kreis enthält.

#### 16.2.5 Implementierung der Rekursion

a) wie geschrieben for  $j = 1, \dots, n$  (for alle eingehenden Kanten )

b) "Vorwärtsrechnung" for  $j = 1, \dots, n$ :  $d_j^{(k)} := d_j^{(k-1)}$   
for  $i = 1, \dots, n$ : (for  
alle ausgehenden Kanten  $(i, j)$ :  
( if  $d_i^{(k-1)} + c_{ij} \leq d_j^{(k)}$  then  $d_j^{(k)} := d_i^{(k-1)} + c_{ij}$  )

#### 16.2.6 Verbesserungsmöglichkeit

$d_i^{(k-1)}$  und  $d_i^{(k)}$  nicht unterscheiden, sondern nur 1 Variable verwenden  $d_i$

#### 16.2.7 Rekursion neu

```
1 for i = 1, ..., n:  
2   # erforschen  
3   for alle Knoten (i, j):  
4     if  $d_i + c_{ij} < d_j$  then  
5        $d_j = d_i + c_{ij}$ 
```

$d_i^{(k-1)} \leq d_i^{(k)} \leq d_i^{(k+1)}$

Per Induktion ergibt sich:

Nach  $k$  Iterationen ist  $d_i^{(\text{NEU})} \leq d_i^{(k)}$

Wenn es keine neg. Kreise gibt, dann ist  $d_i^{(\text{NEU})}$  immer  $\leq d_i^{(n-1)}$  (tatsächlich kürzester Weg)!  
Jedes  $d_i$ , dass im Algorithmus ausgerechnet wird, ist die Länge eines Weges von  $s$  nach  $i$ .

Nach  $k$  Schritten gilt:

$d_i^{(k)} \leq d_i^{(\text{NEU})} \leq d_i^{(n-1)}$

$k = n - 1 \rightarrow d_i^{(\text{NEU})} = d_i^{(n-1)}$

Die Knoten  $i$ , deren Wert sich seit der letzten (Erforschung, Behandlung) geändert haben, speichert man in einer Warteschlange  $Q$ . (Falls sich der Wert nicht geändert hat, wäre es sinnlos ihn noch einmal zu erforschen).

---

```

1 erforsche(i):
2   for all Kante(i,j):
3     if  $d_i + c_{ij} < d_j$  then
4        $d_j = d_i + c_{ij}$ :
5       if  $j \notin Q$  then:
6          $Q.insert(j)$ 

```

---

### 16.2.8 Verbesserter Algorithmus

---

```

1  $d_s = 0, d_i = \infty$  für  $i \neq s$ 
2  $Q = \{s\}$ 
3 while  $Q \neq \emptyset$ :
4   Lösche das erste Element  $i$  aus  $Q$ 
5   erforsche(i)

```

---

## 16.3 Algorithmus von Bellman, Ford, Moore

Wir unterteilen den Algorithmus in Phasen. Phase 0 endet nach der Initialisierung  $Q := \{s\}$ . Phase  $k$  beginnt wenn Phase  $k - 1$  endet und dauert, bis die Knoten in  $Q$ , die zu Beginn der Phase in  $Q$  sind, erforscht sind.

Am Ende von Phase  $k$  gilt  $d_i \leq d_i^{(k)}$

### nach Induktion

Am Beginn der Phase gilt  $d_i \leq d_i^{(k-1)}$

Wir stellen uns vor, dass wir jetzt alle Knoten erforschen.

1 Rekursion "neu"  $\left\{ \begin{array}{l} \text{zuerst die Knoten, die} \\ \text{nicht in } Q \text{ sind} \\ \text{dann die Knoten in } Q \end{array} \right.$

Effekt:  $d_i \leq d_j^{(k)}$  danach.

Um im Falle eines neg. Kreises abbrechen zu können, muss man

- a) nach  $n^2$  Erforschungen abbrechen
- b) oder die Phasen mitzählen und die Phasengrenzen in  $Q$  markieren:

---

```

1  $Q = \{s, \_M\}$ 
2 Phase := 0
3 while  $Q \neq \emptyset$ :
4   if erstes Element =  $\_M$ :
5     entferne  $\_M$  und füge es am Ende ein.
6     Phase = Phase + 1
7   if Phase > n then Abbruch.
8   else:
9     lösche erstes Element  $i$  und erforsche(i)
10

```

---

## 17 Das algebraische Wegproblem (Vorlesung 15 am 05.12.)

Nicht immer interessieren uns die kürzesten Wege in einem Algorithmus. Dafür kann man den Bellman Ford Algorithmus derart verallgemeinern, dass er auch andere Wegeprobleme lösen kann.

### 17.1 Beispiel: Der sichereste Weg.

Gerichteter Graph mit Wahrscheinlichkeiten  $p_{ij}$  auf den Kanten. Die Kante fällt mit Wahrscheinlichkeit  $1 - p_{ij}$  aus.

Ausfallwahrscheinlichkeiten sind unabhängig. Die Wahrscheinlichkeit, dass ein ganzer Weg funktioniert = Produkt der Wahrscheinlichkeiten.

Gesucht ist der sicherste Weg. Dafür brauchen wir zwei Operationen, um das Wegeproblem zu definieren:

$\otimes$  berechnet den "Wert eines Weges aus den Kantengewichten  $c(i_1 i_2 \dots i_k) = c_{i_1 i_2} \otimes c_{i_2 i_3} \otimes \dots \otimes c_{i_{k-1} i_k}$

$\oplus$  Wir wollen die  $\oplus$ -Summe der Werte aller Wege von  $s$  nach  $t$  ausrechnen.

Wertebereich  $H$ :

sicherster Weg  $H = [0, 1]$

1.  $\otimes = *$

2.  $\oplus = \max$

Exkurs: kürzeste Wege

1.  $\otimes = +$

2.  $\oplus = \min$

### 17.2 Algebraisches Wegeproblem

Berechne  $\bigoplus c(P)$  über alle Wege  $P$  von  $s$  nach  $t$ .

$H[0, 1]$

1.  $\otimes = *$

2.  $\oplus = +$

Wenn ich im Knoten  $i$  bin, dann wähle ich einen Nachfolgeknoten  $j$  mit Wahrscheinlichkeit  $c_{ij}$  aus.

$\sum_j c_{ij} \leq 1$  für alle  $i$

Wenn  $< 1$  ist, dann kann der Prozess in diesem Knoten abbrechen.

## 18 Halbring

Ein Halbring ist eine algebraische Struktur  $(H, \otimes, \oplus)$  mit folgenden Eigenschaften:



1.  $\oplus, \otimes$  ist assoziativ:

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c$$

2.  $\oplus$  ist kommutativ:

$$a \oplus b = b \oplus a$$

3. Distributivgesetz:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

Oft ist es nützlich, ein neutrales Element  $e_0$  für  $\oplus$  und  $e_1$  für  $\otimes$  zu haben.

$$a \oplus e_0 = a$$

$$a \otimes e_1 = e_1 \otimes a = a$$

$$a \otimes e_0 = e_0 \otimes a = e_0$$

$e_0$  wird auch absorbierendes Element für  $\otimes$  genannt.

Für das Problem des sichersten Weges sind die neutralen Elemente  $e$  wie folgt:

$$e_1 = 1 \quad e_0 \times a = e_0$$

$$e_0 = 0 \quad e_0 \oplus a = \min(0, a) = a$$

Wir interpretieren  $e_0$  als Summe der leeren Menge.  $e_0$  ist die Lösung, wenn es keinen Weg gibt.

$e_1$  wird als Wert eines leeren Weges ( $s$ ) ohne Kanten interpretiert.

Das klassische kürzeste Wege Problem definiert  $e_0 = \infty$ ,  $e_0 \otimes a = \infty + a = \infty = e_0$  (muss definiert werden!) und  $e_1 = 0$

## 19 Bellman-Ford-Algorithmus

$$d_i^{(k)} = \bigoplus_{\text{Weg von } s \text{ nach } i \text{ mit höchstens } k \text{ Knoten}} c(p)$$

$$d_i^{(k)} = \bigoplus_{\text{Weg von } s \text{ nach } i \text{ mit genau } k \text{ Knoten}}$$

$$d_j^{(k)} = \bigoplus_{i=1}^n (d_i^{(k-1)} \otimes c_{ij}) \oplus \delta_{js}$$

$$\delta_{js} = \begin{cases} e_1, & j = s \\ e_0, & j \neq s \end{cases}$$

Jeder Weg zu  $j$  hat einen Vorgänger  $i$  (außer, wenn es der leere Weg von  $s$  nach  $s$  ist). Wir können die Wege nach  $j$  mit  $\leq k$  Kanten bezüglich  $i$  gruppieren und getrennt aufsummieren. Alle Wege die über  $i$  hereinkommen, haben als letzten Faktor  $c_{ij}$  ausklammern.

$$d_j^k = \bigoplus_{i=1}^n d_i^{k-1} \otimes c_{ij}, d_i^0 = \delta_{is}$$

Das ganze erinnert irgendwie an die Matrizenmultiplikation... und tatsächlich, man kann folgende Vektoren definieren:

$$\begin{aligned} d^{(k)} &= (d_1^{(k)}, d_2^{(k)}, \dots, d_n^{(k)}) \\ d^k &= (d_1^k, d_2^k, \dots, d_n^k) \\ c &= (c_{ij})_{1 \leq i, j \leq n} \end{aligned}$$

Rekursion:

$$\begin{aligned} d^k &= d^{k-1} \otimes C \\ d^{(k)} &= (d^{(k-1)} \otimes C) \oplus (e_1, e_0, \dots, e_0) \end{aligned}$$

Annahme  $s = 1$ !

Matrizenmultiplikation über einem Halbring ist assoziativ (aber niemals kommutativ).

Matrizenaddition ist kommutativ und assoziativ. Es gilt das Distributivgesetz.

Die  $n \times n$  Matrix über einem Halbring bilden wieder einen Halbring, ggf. mit:

$$e_0 = \begin{pmatrix} e_0 & & \\ e_0 & \ddots & \\ & & e_0 \end{pmatrix}, e_1 = \begin{pmatrix} e_1 & & e_0 \\ e_0 & \ddots & \\ e_0 & & e_1 \end{pmatrix}$$

$$d^{(k)} = \bigoplus d^k$$

$$d^{(0)} = d^0 = \{e_1, e_0, \dots, e_0\} = \text{erster Einheitsvektor}$$

$$d^{(k-1)} = d^0 \oplus d^1 \oplus \dots \oplus d^{k-1}$$

$$d^{k-1} \otimes C = \left( \bigoplus_{k=0}^{k-1} d^k \right) \otimes C$$

$$(d^{(k-1)} \otimes C) \oplus d^0 = d^0 \oplus d^1 \oplus \dots \oplus d^k = d^{(k)}$$

## 19.1 Die k-kürzesten Wege (k=2)

$$H = \{(x, y) | x, y \in \mathbb{R} \cup \{\infty\}, x \leq y\}$$

$x$  ist der kürzeste Weg und  $y$  ist der zwei kürzeste Weg.

$$(x, y) \otimes (z, u) = (x + z, \min\{x + u, y + z\})$$

$$(x, y) \oplus (z, u) = \text{die zwei kleinsten Elemente von } (x, y, z, u) \text{ in sortierter Reihenfolge}$$

$$\text{Kantengewichte} = (\text{Kantenlänge}, \infty)$$

$$e_0 = (\infty, \infty), e_1 = (0, \infty)$$

Der zweit kürzeste Weg kann auch Kreise oder Schleifen enthalten. Es gibt höchstens eine Wiederholung eines Knotens  $\rightarrow d^{(n)}$  reicht aus, wenn keine neg. Kreise.

## 20 Fixed Parameter Tractability (Vorlesung 16 am 08.12.)

Sei  $A$  ein NP-vollständiges (wahrscheinlich kein Polynomialzeitalgorithmus), algorithmisches Problem.

### 20.1 Strategien

1. Heuristiken (Pro: gute Ergebnisse, oft schnell; Con: keine Garantie)
2. Approximationalgorithmen (Pro: schnell, Fehler wird abgeschätzt; Con: keine Optimallösung)
3. FPT (Pro: garantiert richtiges Ergebnis; exponentielles Verhalten ist "kontrolliert"; Con: nicht immer anwendbar, immer noch exponentiell)

### 20.2 Definition FPT

Problem mit Parameter  $k$  ist in FPT  $\Leftrightarrow \exists$  Algorithmus mit Laufzeit  $\mathcal{O}(f(k) \cdot n^c)$ . Wobei  $c$  eine Konstante,  $n$  = Eingabegröße und  $f$  eine beliebige Funktion ist.

### 20.3 Beispiel: Unabhängige Knotenmenge auf planaren Graphen

Eingabe:  $G$  ein planarer Graph

Parameter:  $k \in \mathbb{N}$

Frage:  $\exists S \subseteq V \quad |S| \geq k$ ,  $S$  ist unabhängige Knotenmenge.

$\exists$  Algorithmus mit Laufzeit  $6^k \cdot n$

Der naive Algorithmus hat die Laufzeit  $\mathcal{O}(n^{k+1})$  (Teste  $\binom{n}{k} = n^k$  Teilmengen in jeweils  $\mathcal{O}(n)$ )

#### 20.3.1 Suchbaum

Fakt: Jeder planare Graph hat einen Knoten mit Grad  $\leq 5$

Fall 1 : mindestens einer der Knoten  $w_1, \dots, w_5$  ist in  $S$

Fall 2 : keiner der Knoten  $w_1, \dots, w_5$  ist in  $S$

$\Rightarrow$  Es ist mindestens einer der Knoten  $v, w_1, \dots, w_5$  in  $S$ .

#### 20.3.2 Algorithmus

$v$  := Sei Knoten mit Grad  $\leq 5$

$w_1, \dots, w_5$  die Nachbarn

$G_0 = G$  ohne  $v$  und ohne  $N(v)$

$G_1 = G$  ohne  $w_1$  und ohne  $N(w_1)$

...

$G_i = G$  ohne  $w_i$  und ohne  $N(w_i)$

### 20.3.3 Laufzeit

$$\begin{aligned}
 & n \\
 & 6n \\
 & 6^2n \\
 & 6^k n \\
 & \Rightarrow (1 + 6 + 6^2 + \dots + 6^k)n = \left(\frac{6^{k+1} - 1}{6 - 1}\right)n \\
 & = 6 * 6^k * n
 \end{aligned}$$

### 20.3.4 Datenreduktion / Problemkern (Kernelization)

Definition: Sei  $(I, k)$  Eingabe + Parameter  $g(k)$  eine beliebige Funktion.

Ein Algorithmus ist eine Reduktion zu einem Problemkern gdw.  $(I, k) \rightarrow (I', k')$

1. Polynomialzeit

2.  $k' \leq k$

3.  $|I'| \leq g(k)$

1. Beispiel: UK auf planaren Graphen

Wir zeigen  $\exists$  Problemkern  $\leq \mathcal{O}(k)$

Konstruktion  $G = (V, E)$  Eingabe

$\exists$  4-Färbung (geht in polynomieller Zeit)!

Fall 1  $k \leq \frac{n}{4} \Rightarrow$  Problem kann in Polynomialzeit gelöst werden. Problemkern =  $(\emptyset, 0)$

Fall 2  $k \leq \frac{n}{4} \Rightarrow 4k \leq n$ , Problemkern  $(G, k)$  unverändert  $|G| = \mathcal{O}(k)$

2. Beispiel MAX SAT (maximum satisfiability)

Eingabe: Bool'sche Formel  $F$  in KNF,  $m$  Klauseln,  $k \in \mathbb{N}$

Parameter:  $k$

Ausgabe:  $\exists$  Belegung der Variablen mit mindestens  $k$  Klauseln erfüllt?

$F = (x \wedge y \wedge \neg z) \vee (x \wedge \neg y) \vee (x \wedge \neg u \wedge v \wedge z)$

(a)  $k \leq \frac{m}{2} \Rightarrow$  Antwort ist Ja

$\Rightarrow$  Problemkern  $(\emptyset, 0)$

Sei  $B$  irgendeine Belegung und  $\neg B$  die Belegung, die jeder Variablen den jeweils anderen Wahrheitswert zuweist.

Sei  $C$  eine Klausel  $\Rightarrow B$  oder  $\neg B$  erfüllt  $C \Rightarrow$  mindestens die Hälfte der Klauseln werden von  $B$  oder  $\neg B$  erfüllt.

$F = \underbrace{(x \wedge y \wedge \neg z)}_{\text{Länge}=3} \vee \underbrace{(x \wedge \neg y)}_{\text{Länge}=2} \vee \underbrace{(x \wedge \neg u \wedge v \wedge z)}_{\text{Länge}=4}$

Klausel  $C$  ist lang  $\leftrightarrow$  mindestens  $k$  Literale kurz, sonst

$L := \#$  lange Klauseln.

## Beobachtung

wenn  $i$  Klauseln zu erfüllen braucht man nie mehr als  $i$  Variablen zu Belegen (Die restlichen kann man sich später überlegen).

$$F = F_l \wedge F_s$$

$F_l$  = Formel der langen Klauseln

$F_s$  = Formel der kurzen Klauseln

$$\text{Problemkern} = (F_s, k - l)$$

$$\text{Problemkern } \mathcal{O}(k^2)$$

$$\# \text{ Klauseln } m \leq 2k$$

$$\text{Größe der Klausel} \leq k$$

$$\Rightarrow \text{Gesamtgröße} \leq \mathcal{O}(k^2) (\text{Größe des Problemkerns})$$

ACHTUNG: Das Problem wurde vielleicht garnicht verkleinert.

### 20.3.5 Lemma

$\exists FPT \Rightarrow \exists$  Problemkernreduktion

$\Rightarrow$  : Algorithmus  $f(k) * n^c$

Lasse ihn  $n^{c+1}$  viele Schritte laufen.

Fall 1 Problem gelöst.  $\Rightarrow$  Problemkern = triviale Antwort.

Fall 2 Problem nicht gelöst.  $\Rightarrow f(k * n^c) > n^{c+1} \Rightarrow f(k) > n$  Problemkern Identität.

Größe ist  $f(k)$ .

$\Leftarrow$  :  $\exists$  Problemkern der Größe  $g(k)$  mit einem trivialen Algorithmus der in  $h(n)$  das Problem löst.

$$\text{Gesamtlaufzeit: } h(g(k)) + \underbrace{n^c}_{\text{Laufzeit Problemkernreduktion}}$$

$$f := h \circ g$$

$$\leq f(k) + n^c \leq f(k) * n^c$$

$\Rightarrow$  Problem ist in FPT

## 21 Fixed Parameter Tractability (Fortsetzung) (Vorlesung 17 am 08.12.)

### Baumweite

= Maß für die Ähnlichkeit eines Graphen mit einem Baum

Unabhängige Knotenmenge für Bäumen kann man in linearer Zeit berechnen.

Das Spektrum  $\mathcal{O}(2^{\omega} * n)$

Es gibt keinen Polynomialzeitalgorithmus für die unabhängige Knotenmenge in einem Allgemeinen Graphen

### Baumzerlegung der Weite $k$

Definition:  $G = (V, E)$  Graph  $(G, \{X_i | i \in T\}, T)$

Baumzerlegung

$X_i$ -Taschen

$T = (I, E')$  Baum auf den Taschen. (Taschenbaum)

1.  $X_i \subseteq V \quad |X_i| \leq k + 1$
2.  $\bigcup X_i = V$
3.  $T$  Baum
4.  $\forall e = (u, v) \in E \quad \exists i : u, v \in X_i$
5. Konsistenz-Bedingung:  $\forall v \in V : \{i \in I | v \in X_i\}$  (Diese Menge ist zusammenhängend in  $T$ ).

Fakt: Bäume haben Baumweite 1

Was macht man jetzt damit?

Eingabe:  $G = (V, E)$  + Baumzerlegung der Weite  $\omega$  wird eine Zahl  $k \in \mathbb{N}$

Parameter:  $\omega$  die Baumweite

Ausgabe: Existiert  $S$ , UK mit  $|S| \geq k$ ?

Ja, in  $\mathcal{O}(2^{\omega} * w * n)$  Zeit

1.  $A \cap B = X_i$
2.  $v \in A, w \in B \Rightarrow vw \in E$   
 $\Rightarrow v$  oder  $w$  ist in  $X_i$   
 $U \subseteq A, W \subseteq B$  UK in  $G$   
 $U \cap X_i = W \cap X_i$   
Behauptung:  $U \cup W$  ist UK in  $G$   
Beweis per Widerspruch: Sei  $x \in U, y \in W$  und  $(x, y) \in E \Rightarrow$  o.B.d.A.  $x \in X_i$   
 $\Rightarrow x \in B \Rightarrow X_i$  ist  $U \cap X_i = W \cap X_i \Rightarrow x \in B$  (Widerspruch!)

### 21.1 Algorithmus mit DP

Annahme der Taschenbaum  $T$  ist gewurzelt.

### 21.1.1 Teilprobleme

$\forall i$  Index einer Tasche

$\forall S \subseteq X_i$

$f(i, S) :=$  Größe der größten UK  $W$  mit  $X_i \cap W = S$  in den Teiltaschenbaum unter  $i$ .

### 21.1.2 Anker

$i$  sei ein Blatt:  $S \subseteq X_i \forall S$

$$f(i, S) = \begin{cases} |S|, & \text{wenn } S \subseteq K \\ -\infty, & \text{sonst.} \end{cases}$$

### 21.1.3 Rekursion

$i$  hat Kinder  $j$  und weitere  $k$ ,  $S \subseteq X_i$

$$f(i, S) = |S| + \max\{f(j, U) - |U \cap X_i \cap X_j| \mid U \cap X_i \cap X_j = S \cap X_i \cap X_j \text{ und } U \subseteq X_j\}$$

### 21.1.4 Lösungen

$f(i, S) = -\infty$ , wenn  $S$  nicht in UK

- müssen alle Konfigurationen von  $U \subseteq X_j$  anschauen
- $U$  muss kompatibel sein mit  $S$
- bestimmte Knoten nicht doppelt zählen!



## 22 Kürzeste Wege (Fortsetzung) (Vorlesung 18 am 15.12.)

### 22.1 Berechnen aller Paare von kürzesten Wegen

1. kürzeste Wege von einem Startpunkt  $k$ ,  $n$  mal wiederholen - bspw. Bellman-Ford  $n \times \mathcal{O}(mn^2) \leq \mathcal{O}(n^4)$
2. Floyd-Warshall, dynamische Programmierung über Matrizen

Es geht aber auch besser:

#### 22.1.1 Verbesserung

Definiere beliebige Knotengewichte  $u_i, i \in V$

Betrachte modifizierte Kosten:  $\bar{c}_{ij} = c_{ij} + u_i - u_j$

Die kürzesten Wege bezüglich  $\bar{c}_{ij}$  sind die gleichen wie die kürzesten Wege bezüglich  $c_{ij}$ . Der Weg  $i_1, i_2, \dots, i_k$  berechnet sich wie folgt:  $\bar{c}_{i_1 i_2} + \bar{c}_{i_2 i_3} + \dots + \bar{c}_{i_{k-1} i_k} = (c_{i_1 i_2} + u_{i_1} - u_{i_2}) + (c_{i_2 i_3} + u_{i_2} - u_{i_3}) + \dots = (c_{i_1 i_2} + c_{i_2 i_3} + \dots + c_{i_{k-1} i_k}) + u_{i_1} - u_{i_k}$  für alle Wege von  $i_1$  nach  $i_k$  konstant.

Idee: finde  $u_i$ , sodass  $\bar{c}_{ij} \geq 0 \quad \forall i, j \rightarrow$  Algorithmus von Dijkstra  $\Rightarrow \mathcal{O}(n \log n + m)$  anwendbar und damit verbessert sich die Laufzeit anstelle von  $\mathcal{O}(nm)$

Beobachtung:  $d_i^*$  sei die Länge des kürzesten Weges von einem festen Startknotens (z.B.  $s = 1$ ) zu  $i$ . Dann wird mit  $u_i := d_i^*$  die modifizierten Kosten  $\bar{c}_{ij} \geq 0$

Beweis:  $\bar{c}_{ij} = c_{ij} + u_i + u_j = c_{ij} + d_i^* - d_j^* \geq 0 \Leftrightarrow d_j^* \leq d_i^* + c_{ij}$

Jeder Knoten muss von  $s$  erreichbar sein. Man kann einen künstlichen Knoten  $s$  mit Kanten  $s, i \forall i \in V$  und Kosten  $c_{si} = 0$

#### 22.1.2 Algorithmus

1. Füge künstlichen Startknoten  $s$  mit Kanten  $c_{si} = 0$  dazu.
2. Bellman-Ford-Moore-Algorithmus mit  $s$  als Startpunkt.
  - (a) Wenn negativer Kreis auftritt, Abbruch.
  - (b) kürzeste Distanzen  $d_i^*$
3. Für jeden Startknoten  $i$ : Algorithmus von Dijkstra mit Kosten  $\bar{c}_{ij}$ .
4. Ergebnisdistanzen müssen noch umgerechnet werden.
5. kürzester Weg von  $i$  nach  $j \dots -u_i + u_j$  addieren.

#### 22.1.3 Laufzeit

2.  $\mathcal{O}(mn)$
3.  $n * \mathcal{O}(m + n * \log n)$

Summe  $\mathcal{O}(mn + n^2 \log n)$

## 22.2 Floyd-Warshall Algorithmus zur Berechnung aller kürzesten Wege

### 22.2.1 Teilprobleme

$c_{ij}^k$  = alle Wege von  $i$  nach  $j$ , die als Zwischenknoten nur die Knoten  $1, 2, \dots, k$  verwenden dürfen.

### 22.2.2 Lösungen

$c_{ij}^0 = c_{ij}$  ( $i \neq j$ )  $c_{ii}^0 = c_{ii}$  (Wir legen fest: der leere Weg von  $i$  nach  $i$  verwendet als Zwischenknoten den Knoten  $i$  - das ist im Grunde kein Unterschied mehr zu  $c_{ij}^0$ )

### 22.2.3 Rekrusion

$(c_{ij}^{k-1})_{i,j \in V} \rightarrow (c_{ij}^k)_{i,j}$  Ein Weg in  $c_{ij}^k$  ( $i, j \neq k$ ) hat zwei Möglichkeiten:

1. Er verwendet den Knoten  $k$  überhaupt nicht.  $\rightarrow c_{ij}^{k-1}$
2. man kann den Weg aufspalten in
  - (a) Anfangsstück von  $i$  nach  $k \rightarrow c_{ik}^{k-1}$
  - (b) Beliebige  $\leq 0$  Zwischenstüpe von  $k$  nach  $k \rightarrow c_{kk}^{k-1}$
  - (c) Ein Endstück von  $k$  nach  $j \rightarrow c_{kj}^{k-1}$

Jedes dieser Stücke verwendet  $k$  nicht als Zwischenknoten

$$(i, j \neq k) \quad c_{ij}^k = c_{ij}^{k-1} \oplus c_{ik}^{k-1} \otimes [e_1 \oplus c_{kk}^{k-1} \oplus (c_{kk}^{k-1})^2 \oplus (c_{kk}^{k-1})^3 \oplus \dots] \otimes c_{kj}^{k-1}$$

Der Term in den eckigen Klammern ist eine unendliche Summe der folgenden Gestalt:  $e_1 \oplus a \oplus a^2 \oplus a^3 \oplus \dots := a^*$ , wobei bspw.  $a^3$  als  $a \oplus a \oplus a$  zu verstehen ist.

Wir haben angenommen, dass die Rechengesetze (Distrib. etc.) auch für unendliche Summen gelten, sofern diese Existieren.

$$i, j \neq k \quad c_{ij}^k = c_{ij}^{k-1} \oplus c_{ik}^{k-1} \oplus (c_{kk}^{k-1})^* \otimes c_{kj}^{k-1}$$

$$j \neq k \quad c_{kj}^k = (c_{kk}^{k-1})^* \otimes c_{kj}^{k-1}$$

$$i \neq k \quad c_{ik}^k = (c_{kk}^{k-1})^* \otimes c_{ik}^{k-1}$$

$$i \neq k \quad c_{kk}^k = (c_{kk}^{k-1})^*$$

### 22.2.4 Beispiele für $a^*$

$$\oplus = \min$$

$$\otimes = +$$

$$a^* = \min\{0, a, 2a, 3a, \dots\} = \begin{cases} 0, & a \leq 0 \\ \text{undefiniert, sonst} \end{cases}$$

$$\oplus = +$$

$$\otimes = *$$

$$a^* = 1 + a + a^2 + a^3 + \dots = \begin{cases} \frac{1}{1-a}, & |a| < 1 \\ \text{undefiniert, sonst} \end{cases}$$

zweit kürzeste Wege:  $a = (u_1, u_2) \in \mathbb{R}^2, u_1 \leq u_2$

$$a^* = e_1 \oplus a \oplus a^2 \oplus \dots (0, \infty) \oplus (u_1, u_2) \oplus (2u_1, u_1 + u_2) \oplus \dots = \begin{cases} (0, u_1), & u_1 \leq 0 \\ \text{undefiniert, sonst} \end{cases}$$

### 22.2.5 Algorithmus

Initialisiere  $c_{ij}^0 = c_{ij}$  for  $k = 1, \dots, n$

1. Berechne  $c_{kk}^k$
2. Zeile  $k$ :  $c_{kj}^k$  für  $j \neq k$
3. Spalte  $k$ :  $c_{ik}^k$  für  $i \neq k$

4. Berechne die übrigen Elemente  $c_{ij}^k$  für  $i, j \neq k$

Ergebnis:  $c_{ij}$  in Laufzeit:  $\mathcal{O}(n^3)$  mit  $\oplus, \otimes, *$ -Operation

### 22.3 Halbring der formalen Sprachen

$H = \sigma^*$  = Mengen von Wörtern über einem Alphabet  $\sigma$

$\oplus = \cup$

$\otimes = \odot$  (Konkatenation)

Frage: Welche Sprache wird von einem endlichen Automaten akzeptiert.

Einen Automaten kann man als Zustandsgraph interpretieren. In diesem Graphen kann man das algebraische Wegeproblem anwenden.

Algorithmus von Kleene (ca. 1950er), zum Beweis, dass jede Sprache, die von einem NEA akzeptiert wird, durch einen regulären Ausdruck beschrieben werden kann.

## 23 Matrizenmultiplikation und kürzeste Wege (Vorlesung 19 am 19.12.)

Wir definieren uns eine Matrix mit der entsprechenden Matrixmultiplikation:

$$C = \mathbb{R}^{n \times n} C = (C_{ij}) C^2 = C * C (C^2)_{ik} = \sum_{j=1}^n C_{ij} * C_{jk}$$

Also alle Wege mit zwei Kanten von  $i$  nach  $k$ .

Satz: In einem Halbring ist das Element  $(i, j)$  der Matrix  $C^k$  die Summe aller Gewichte, aller Wege von  $i$  nach  $j$  mit  $k$  Kanten.

Beweis durch Induktion mit  $k = 1 : C^1 = C \quad k = 0 C^0 = I$

$k \rightarrow k + 1$

$$C^{k+1} = C^k \otimes C$$

$$(C^{k+1})_{ij} = \bigoplus_{l=1}^n \underbrace{(C^k)_{il} \otimes C_{lj}}_{\substack{\text{Summe aller Wege von } i \text{ nach } l \text{ mit } k \text{ Knoten} \\ \text{alle Wege von } i \text{ nach } j \text{ mit } k+1 \text{ Kanten deren vorletzter Knoten } l \text{ ist}}}$$

Beim Bellman-Ford-Algorithmus:  $d_j^k$  = Weg von Startknoten zu  $j$  mit genau  $k$  Kanten = eine

Zeile von  $C^k = (C^k)_{sj}$

$$d_j^k = (e_0, e_0, \dots, \underbrace{e_1}_s, \dots, e_0, e_0) \otimes C^k$$

$$d_j^k = (d_1^k, \dots, d_n^k) \text{ folgt } d^{k+1} = d^k \otimes C$$

Die Lösung des algebraischen Wegproblems lässt sich auch beschreiben als:

$$I \oplus C \oplus C^2 \oplus C^3 \oplus \dots := C^*$$

Viele Halbringe sind idempotent:  $a \oplus a = a$

z.B.: kürzeste Wege:  $\oplus = \min$

Es reicht bei kürzesten Wegen aus, die Summe  $C^*$  bis  $C^{n-1}$  oder  $C^n$  auszurechnen:

$$\begin{aligned} (I \oplus C)^k &= I^k \oplus \binom{k}{1} C \oplus \binom{k}{2} C^2 \oplus \dots \oplus C^k \\ &= I^k \oplus \binom{k}{1} C \oplus \binom{k}{2} C^2 \oplus \dots = I \oplus C \oplus \dots \oplus C^k \underbrace{C \oplus C \oplus \dots \oplus C}_k = C \end{aligned}$$

### 23.0.1 Algorithmus

(nur für idempotente Halbringe!)  $D = I \oplus C$

wiederhole  $\lceil \log_2 n \rceil$  mal

$$D = D \otimes D \leftarrow D = (I \oplus C)^k = C^{(k)} \text{ für ein } k \leq n$$

falls  $D \oplus D \neq D$  ist, dann hat der Graph negative Kreise, andernfalls ist  $D = C^*$

Halbring:  $(\mathbb{R}, +, *)$

$$\begin{aligned}
C^* &= I + C + C^2 + C^3 + \dots & | \times C \\
C^* * C &= C + C^2 + C^3 + \dots \\
I + C^* C &= I + C + C^2 + C^3 + \dots = C^* \\
I &= C^* (I - C) \\
C^* &= (I - C)^{-1}
\end{aligned}$$

$$\begin{aligned}
a^* &= 1 + a + a^2 + a^3 \dots = \frac{1}{1-a} \text{ falls } |a| < 1 \\
a^* * (1 - a) &= 1 \\
a^* - a^* a &= 1 \\
a^* &= 1 + a^* * a \\
a^* &= e_1 \oplus a^* \otimes a
\end{aligned}$$

Wenn man in Floyd-Warschall Algorithmus die Formel  $a^* = \frac{1}{1-a}$  für alle  $a \neq 1$  blind anwendet und niemals der Wert 1\* berechnet werden musste, dann berechnet der Algorithmus die Matrix  $(I - C)^{-1}$ , die Matrizeninversion.  
Dieser Algorithmus heißt (bis auf kosmetische Änderungen) Gauß-Jordan-Elimination.

## 24 Komplexitätsklassen P und NP (Vorlesung 20 am 05.01.)

Es gibt verschiedene Klassen von Problemlösungen:

1. polynomiell lösbare Probleme (kürzeste Wege bspw.  $\mathcal{O}(n^3)$ )  
P ist die Klasse der Probleme, die sich in polynomieller Zeit lösen\* lassen.  
Genauer: auf einer Turingmaschine oder Registermaschine (mit log. Kostenmaß) lösen lassen.
2. nicht polynomiell lösbare Probleme (Rundreiseproblem mit  $\mathcal{O}(n^2 2^n)$ )

Die Problembeschreibung muss im wesentlichen spezifizieren, wie die Eingabe zu kodieren ist. Das ist wichtig, weil polynomiell = polynomiell in Länge der Eingabe bedeutet. Dazu die Standardannahme: irgendein vernünftiges Eingabeformat; Zahlen werden binär kodiert. (Alternativ unäre Kodierung)

Polynomielle Zeit:  $\exists$  Konstante  $a, b, c$ , sodass die Turingmaschine bei Eingabe der Länge  $n$  in  $an^b + c$  Schritten anhält und die Ausgabe auf das Band schreibt.

Der Speicher (PSPACE) ist automatisch beschränkt, wenn die Zeit polynomiell beschränkt ist. Ein Problem  $A$  (mit Spezifikation der Eingabe) gehört zur Klasse  $\mathcal{P}$ , wenn es sich auf einer Turingmaschine mit polynomieller Laufzeit lösen lässt.

Berechnungsprobleme: (klar) Entscheidungsprobleme: Ausgabe hat nur ein Bit: Ja / Nein. Für die Theorie ist es praktischer ein Entscheidungsproblem zu betrachten.

### 24.1 P

#### 24.1.1 Problem: Hamiltonscher Kreis

Eingabe: ein ungerichteter Graph

Frage: Hat der Graph einen Kreis, der jeden Knoten genau einmal besucht?

Idee/Fakt: Jedes Entscheidungsproblem ist eine formale Sprache  $L \subseteq \sigma^*$

$L = \{x \in \sigma^* \mid x \text{ ist eine gültige Eingabe und die Antwort auf die Frage ist JA}\} = \text{Menge der JA-Eingaben.}$

#### 24.1.2 Problem: Rundreise

Das Beispiel für ein Berechnungsproblem ist das Rundreiseproblem. Berechnungsprobleme lassen sich in der Regel als (ein oder mehrere) Entscheidungsproblem formulieren.

Eingabe: Ein Graph mit Kantengewichten, eine Zahl  $K$ .

Frage: Gibt es eine Tour mit Länge  $K$ ?

Zwischenfrage: ist nicht jedes Berechnungsproblem automatisch ein Entscheidungsproblem?

Antwort: nicht immer, aber allgemein gibt es eine Methode: FRAGE: Ist das  $k$ -te Bit des Ergebnisses 1?

#### 24.1.3 Reduzierbarkeit

Zwei Entscheidungsprobleme  $A, B \subseteq \sigma^*$ .  $A$  heißt polynomiell reduzierbar auf  $B$  ( $A \leq_p B$ ), wenn es eine in polynomieller Zeit berechenbare Funktion  $f : \sigma^* \rightarrow \sigma^*$  gibt, sodass  $\forall x \in \sigma^* : x \in A \Leftrightarrow f(x) \in B$

Umgangssprachlich als:  $A$  wird auf  $B$  zurückgeführt oder  $A$  kann mit Hilfe von  $B$  gelöst werden oder  $A$  ist mindestens so leicht wie  $B$

Satz: Wenn  $A \leq_p B$  und  $B \in P$ , dann  $A \in P$ . Beweis: Annahme:

$f$  lässt sich in  $\leq an^b + c$  Zeit berechnen.  $B$  lässt sich in  $a'n^{b'} + c'$  Zeit entscheiden.

Entscheidungsalgorithmus für  $A$ :

Eingabe  $x$  mit  $|x| = n$ :

1. Berechne  $f(x)$  Laufzeit  $\leq an^b + c$ , sodass wir annehmen können:  $|f(x)| \leq an^b + c$
2. Entscheide  $B$  für Eingabe  $f(x)$ . Korrekt:  $x \in A \Leftrightarrow f(x) \in B$

Laufzeit:

Lemma:  $(u + v)^{b'} \leq 2^{b'}(u^{b'} + v^{b'})$

Beweis: o.B.d.A.  $u \leq v$

$(u + v)^{b'} \leq (v + v)^{b'} = (2v)^{b'} = 2^{b'}v^{b'} \leq 2^{b'}(u^{b'} + v^{b'}) \square$

$$\begin{aligned} a'|f(x)|^{b'} + c' &\leq a'(an^b + c)^{b'} + c' = \underbrace{a'a^{b'}}_{a''} n^{\underbrace{bb'}}_{b^n} + \underbrace{ac^{b'} + c'}_{c''} \\ &= a''n^{b''} + c'' \end{aligned}$$

#### 24.1.4 Beispiel

Hamilton-Kreis  $\leq_p$  Rundreise

Eingabe: Graph.

Gib allen Kanten Gewicht 1 und allen nicht vorhandenen Kanten Gewicht 2. Schranke  $K = n$ .

$\exists$  Rundreise mit Gewicht  $\leq n \Leftrightarrow$  urspr. Graph hat einen Hamiltonkreis.

Sehr einfach, ABER man muss immer 3 Dinge beweisen:

1.  $\Leftarrow$
2.  $\Rightarrow$
3. polynomielle Laufzeit von  $f$

#### 24.2 NP

Die Klasse  $NP$  ist die Klasse der Sprachen  $A \subseteq \sigma^*$ , die von nichtdeterministischen Turingmaschinen in polynomieller Zeit akzeptiert werden.

Nichtdeterministische Turingmaschinen (NTM) ist wie eine deterministische TM, aber es gibt abhängig vom aktuellen Zustand und vom gelesenen Bandsymbol mehrere (beliebig viele) mögliche Aktionen. Berechnung bei konkreter Eingabe ist nicht eindeutig vorgegeben, sondern es gibt mehrere mögliche Berechnungen.

Die von einer NTM in Laufzeit  $f$  akzeptierte Sprache ist  $\{x \in \sigma^* | \text{Es gibt eine Berechnung mit Eingabe } x, \text{ die in Zeit } \leq f(|x|) \text{ in einem akzeptierenden Zustand führt}\}$

##### 24.2.1 Beispiel: Rundreiseproblem $\in NP$

nicht deterministischer Algorithmus: Eingabe:  $n$ , Graph auf  $n$  Knoten mit Kantengewichten, Schranke  $k$ .

1. Schreibe nicht deterministisch eine Permutation der Zahlen  $1, 2, \dots, n$  aufs Band.
2. Berechne die Kosten der entstehenden Tour.
3. Akzeptiere, falls die Kosten  $\leq K$  sind.

NP Probleme, die sich durch erraten lösen lassen. NP ist nicht von vornherein abgeschlossen gegenüber Komplementierung.  $\leftarrow$  ungelöstes Problem  $\rightarrow$  Klasse  $co*NP$

## 25 Alternative NP-Definition (Vorlesung 21 am 09.01.)

Klausurtermin 13.2. oder 16.2.

### 25.1 Polynomiell verifizierbares Zertifikatskriterium

Die bisher gezeigte Definition ist zwar korrekt, aber nicht gut umsetzbar in der Praxis. Wir definieren daher eine alternative die den Nichtdeterminismus etwas anders interpretiert.

Eine Sprache  $A \subseteq \sigma^*$  hat ein p.v.ZK, wenn es zwei Polynome

$$p_1(n) = a_1 n^{b_1} + c_1$$

$$p_2(n) = a_2 n^{b_2} + c_2$$

und eine Funktion

$$f : (\sigma \cup \{\$ \})^* \rightarrow \{\text{JA}, \text{NEIN}\}$$

gibt, mit folgenden Eigenschaften:

1.

$$x \in A \Leftrightarrow \exists y \in \sigma^* \quad |y| \leq p_1(|x|) \text{ und } f(x\$y) = \text{JA}$$

2.  $f$  ist in Laufzeit  $\leq p_2(n)$ , bei Eingabe der Länge  $n$  berechenbar.

**Anmerkungen:**  $y$  heißt das *Zertifikat* (witness);  $f$  ist das *Zertifikationskriterium*

**Satz (alternative Definition):**  $A \in NP \Leftrightarrow A$  hat ein p.v.ZK.

#### 25.1.1 Beispiel: Rundreiseproblem

Zertifikat = Tour

#### 25.1.2 Beweis

$NP \leftrightarrow_{\text{DEF}} nTM$  (nicht deterministische Turingmaschine)

Wir wollen zeigen:  $nTM \Rightarrow pvZK$ :  $nTM \mathcal{M}$  akzeptiert  $A$  in Laufzeit  $t \leq an^b + c$  bei Eingabe der Länge  $n$ . Das Zertifikat ist eine akzeptierende Berechnung für Eingabe  $x$ . Wobei eine Berechnung eine Folge von Konfigurationen der Turingmaschine  $\mathcal{M}$  ist. Die Konfiguration besteht aus Zustand, Bandinhalt und Position des Lesekopfs. Dabei ist die Länge des Zertifikats  $\leq t$  Konfigurationen bzw.  $\leq n + t \leq 0(t(n + t)) = \underbrace{\text{polynomiell in } n, \text{ wenn } t \text{ polynomiell in } n \text{ ist.}}_{A(n)}$

$f(x, y)$  überprüft ob  $y$  eine akzeptierende Berechnung für die Eingabe  $x$  ist.

Rückrichtung:  $pvZK \Rightarrow nTM \mathcal{M}$

$\mathcal{M}$  liest die Eingabe  $x$ , schreibt anschließend  $\$$  und nichtdeterministisch bis zu  $p_1(|x|)$  beliebige Symbole  $y$  auf das Band. Anschließend fährt sie an den Anfang zurück und startet das (deterministische) Programm für  $f$ .

### 25.2 Komplexitätsklassen

$P \subseteq NP$  Vermutung  $P \neq NP$ , aber nicht bewiesen.



### 25.2.1 NP-vollständig

Definition: Ein Problem  $A$  ist NP-vollständig, wenn

1.  $A \in NP$
2.  $\forall B \in NP : B \leq_p A$

Anmerkung zu (b):  $A$  ist mindestens so schwer wie alle anderen Probleme in  $NP$

Wenn nur (b) gilt, dann ist  $A$  NP-schwer (engl. NP-hard). Der Begriff NP-schwer wird für Probleme verwendet, die keine Entscheidungsprobleme sind.

Beobachtungen:

1.  $A$  ist NP-Schwer und  $A \in P \Rightarrow P = NP$
2.  $C$  ist NP-Schwer und  $C \leq_p D \Rightarrow D$  ist NP-schwer.

### 25.3 SAT - satisfiability = Erfüllbarkeit

Eingabe: eine boole'sche Formel in KNF (konjunktiver Normalform).

Frage: Gibt es eine Belegung der Variablen die die Formel wahr macht?

#### 25.3.1 Beispiel KNF

$$\underbrace{(x_1 \vee \neg x_5 \vee x_7 \vee \neg x_8)}_{\text{Literal}} \wedge \underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\text{Klauseln}} \wedge (x_3 \vee x_4) \wedge \dots$$

#### 25.3.2 SATZ: Cook-Levin, 1971

SAT ist NP-vollständig.

1.  $SAT \in NP$  (Zertifikat = Belegung) ✓
2.  $B \in NP$  beliebig: zu zeigen  $B \leq_p SAT$

Es gibt eine nTM  $\mathcal{M}$ , die  $B$  in Laufzeit  $p(n)$  akzeptiert. Wir wollen die Aussage:

$\mathcal{M}$  akzeptiert die Eingabe  $x = x_1, x_2, \dots, x_n$  in  $T$  Schritten.

als Boolesche Formel formulieren. Gegeben Eingabe  $x$  für  $B$ . Ziel:  $x$  in Eingabe  $f(x)$  für SAT transformieren:  $x \in B \Leftrightarrow f(x)$  erfüllbar.

Variablen:

$$b_{i,t,x} = 1, \text{ wenn an Stelle } i \text{ des Bandes zum Zeitpunkt } t \text{ das Symbol } x \text{ steht.}$$

$$\underbrace{(-T \leq i \leq T, 0 \leq t \leq T, x \in P \text{ (Bandalphabet)})}_{\mathcal{O}(T^2) \text{ Boolesche Variablen}}$$

$$p_{i,t} = 1, \text{ wenn Maschine zum Zeitpunkt } t \text{ an Position } i \text{ ist}$$

$$(-T \leq i \leq T, 0 \leq t \leq T)$$

$$z_{t,q} = 1, \text{ wenn } \mathcal{M} \text{ zum Zeitpunkt } t \text{ im Zustand } q \text{ ist.}$$

$$(0 \leq t \leq T, q \in Q(\text{Zustandsmenge}))$$

Klauseln, die gültige Berechnungen charakterisieren:

$$[p_{it} \Rightarrow (b_{j,t,x} \Leftrightarrow b_{j,t+1,x})]$$

$$(0 \leq t \leq T-1, -T \leq i, j \leq +T, i \neq j, x \in T')$$

1. Bandinhalt bleibt unverändert, wenn Kopf nicht an Position  $j$  ist.
2. Kopfposition ändert sich höchstens um 1
3. Ausgangsinhalt des Bandes:  
 $b_{i,0,x_{i+1}} = 1$  für  $0 \leq i \leq n-1$ ,  $b_{b_i,0,B} = 1$  für  $-T \leq i \leq 0$  und  $n \leq i \leq T$
4. Auf jeder Position steht höchstens 1 Symbol:

$$\neg(b_{i,t,x} \wedge b_{i,t,y}) \quad i, t, x \neq y \in T'$$

5. Zu jedem Zeitpunkt gibt es nur einen Zustand
6. Übergangsrelation:  $(z_{t,\underline{q}} \wedge p_{i,t} \wedge b_{i,t,\underline{x}} \Rightarrow (\dots) \vee (\dots) \vee (\dots))$   
 z.B.  $(z_{t+1,q'} \wedge p_{i-1,t+1} \wedge b_{i,t+1,y})$
7. irgendwann erreicht  $\mathcal{M}$  einen akzeptierenden Zustand.

Jeder dieser Ausdrücke enthält  $\leq K$  Variablen ( $k$  hängt von  $\mathcal{M}$  ab). kann die KNF mit  $\leq 2^K$  Klauseln transformiert werden

## 26 Reduktionen (Vorlesung 22 am 12.01.)

$A$  NP-schwer.,  $A \leq_p B \Rightarrow B$  NP-schwer

Wir wissen aus der letzten Vorlesung, dass SAT NP-schwer ist.

### 26.1 Unabhängige Menge (von Knoten)

U.M. ist NP-vollständig.

Eingabe: Ein Graph  $(G = (V, E))$ , Schranke  $k$

Frage: Gibt es eine unabhängige Menge  $S \subseteq V$  mit  $|S| \geq k : \forall uv \in E : u \notin S \vee v \notin S$

#### 26.1.1 Beweis

1. U.M.  $\in$  NP

2.  $\text{SAT} \leq_p \text{U.M.}$

Gegeben: boolesche Formel  $\phi$  in KNF. Wir wollen  $\phi$  in einem Graphen  $f(\phi)$  mit einer Schranke  $k$  überführen, sodass  $\phi$  erfüllbar  $\Leftrightarrow f(\phi)$  hat eine U.M. mit  $k$  Knoten. Anmerkung:  $\phi$  habe  $n$  Variablen  $x_1, \dots, x_n$  und  $m$  Klauseln.

Beispiel Klausel:  $x_1 \vee \neg x_2 \vee x_3$  Allgemein: Für eine Klausel  $u_1 \vee u_2 \vee \dots \vee u_l$  mit  $l$  Literalen. Füge  $l$  neue Knoten hinzu und verbinde sie mit den Knoten  $\neg u_1, \neg u_2, \dots, \neg u_l$ . Verbinde die Knoten untereinander zu einem vollst. Graphen.  $K = m + n$ . zu zeigen:

1.  $\phi$  erfüllbar  $\Leftrightarrow f(\phi)$  hat U.M. mit  $\leq m + n$  Knoten.

$\Rightarrow$  (leicht): Gegeben sei eine erfüllende Belegung:

$S_1 = \{x_i | x_i \text{ ist wahr}\} \cup \{\neg x_i | x_i \text{ ist falsch}\}$  Weil es in jeder Klausel ein wahres Literal gibt, finden wir in der entsprechenden Knotenmenge einen Knoten, der nicht zu einem Knoten aus  $S_1$  benachbart ist.  $T :=$  Menge dieser  $m$  Knoten.  $S = S_1 \cup T$ .

$\Leftarrow$  (schwerer):  $S$  sei eine U.M. mit  $m + n$  Knoten:

Jede Variablenkante und jede Klausel Knotenmenge enthält  $\leq 1$  Knoten aus  $S$ .  
 $|S| = m + n \Rightarrow$  Jede Variablenkante jede Klauselknotenmenge enthält genau 1

als Belegung interpretieren

Knoten aus  $S$ . Die Belegung erfüllt alle Klauseln, dann in jeder Klauselknotenmenge ist ein Knoten in  $S$ . Der entsprechende Nachbau der Variablenkanten  $\notin S \Rightarrow$  Das entsprechende Literal erfüllt die Klausel.

2.  $f$  polynomiell berechenbar  $\leftarrow$  KLAR:  $|V| = 2n +$  Gesamtlänge aller Klauseln.  $|E| = |V|^2$   
✓

#### 26.1.2 Anmerkungen: Wie kommt man drauf?

Manche Probleme sind so klein, dass man die Lösung sofort erkennt und eine Reduktion durchführen kann. Andere Probleme lassen sich nur durch sogenannte Vorrichtungen (engl. Gadgets) so umformen, dass die Reduktion nachvollziehbar und richtig bleibt.

### 26.2 Hamiltonkreis in gerichtetem Graph

ist NP-vollständig. (Spezialfall des Rundreiseproblems)

1.  $\in$  NP ✓

2.  $\text{SAT} \leq_p \text{HAMg}$

### 26.2.1 Variablen Gadgets (Vorrichtungen)

Für jede Variable eine lange Kette, die in zwei Richtungen durchlaufen werden kann.

1. von links nach rechts = wahr
2. von rechts nach links = falsch

### 26.2.2 Klausel Gadgets

Beispiel  $x_1 \vee \neg x_2 \vee x_3$

Jede Klausel ist ein einzelner Knoten: Dieses und mit 2 benachbarten Knoten auf alle enthaltenen Variablenketten verbunden, sodass der Knoten zwischendurch eingefügt werden kann, wenn die Kette in der Reihenfolge durchlaufen wird, die die Klausel wahr macht. Anfangs und Endknoten der Ketten, werden zu einem vollst. Graphen verbunden. Rechts und links von den Nachbarn eines Klauselknoten muss jeweils 1 Nachbar ohne zusätzliche Kanten verbleiben und die Nachbarn von verschiedenen Klauselknoten sind disjunkt.

### 26.2.3 Beweis

$\Rightarrow$  Belegung  $\Rightarrow$  Hamiltonkreis  $\checkmark$

$\Leftarrow$  Hamiltonkreis

Lemma: Wenn ein Hamiltonkreis zu einem Klauselknoten kommt, muss er direkt zur gleichen Kette wieder zurückspringen.

Wir nehmen an: wir haben eine Kette und der Klauselknoten  $w$  ist an irgendeiner Stelle mit der Kette verbunden.  $(u, w), (w, v), (u', u), (v, v') \in E$

Annahme:  $(u, w) \in H$  (Hamiltonkreis),  $(w, v) \notin H$

Fall 1)  $(v, u) \in H \Rightarrow H$  kann  $u'$  nicht besuchen.

Fall 2)  $(u', u) \in H \Rightarrow H$  kann  $v$  nicht besuchen.

Wenn die Annahme gilt, dann muss der Hamiltonkreis aus einer Folgen von Ketten bestehen, in beliebiger Reihenfolge und Orientierung, in die die Klauselknoten eingefügt sind. Orientierung  $\rightarrow$  Belegung ablesen  $\rightarrow$  Weil die Klauselknoten besucht werden müssen.... erfüllt.

HAM-Kreise ist auch NP-vollständig mit ungerichteten planaren Graphen vom Maximalgrad 3.

## 26.3 3-SAT

Spezialfall von SAT, wo jede Klausel genau 3 Literalte enthält. Ist auch NP-vollständig.

1. NP  $\checkmark$
2. Reduktion: Eingabe beliebige Formel  $\phi$  in KNF

zu lange Klausel durch eine Variable aufspalten:

$$(x_1 \vee \neg x_2 \vee x_5 \vee \neg x_7 \vee x_3) \rightarrow (x_1 \vee \neg x_2 \vee x_5 \vee y) \wedge (\neg y \vee \neg x_7 \vee x_3)$$

Was ist mit zu kurzen Klauseln?  $x_1 \vee \neg x_2$

$$(x_1 \vee x_2 \vee y) \quad \underbrace{\wedge(\dots) \wedge(\dots)}_{\text{nur erfüllbar wenn } y \text{ falsch ist}}$$

## 27 Weitere Reduktionen (Vorlesung 23 am 16.01.)

### 27.1 Graphenfärbung

Eingabe: Graph  $G = (V, E)$

Frage: Lässt sich  $G$  mit  $k$  Farben färben?

$\exists g : V \rightarrow \{1, \dots, k\} : \forall uv \in E : g(u) \neq g(v)$ ?

### 27.2 3-Färbbarkeit

wie oben, nur mit  $k = 3$  fest.

Satz: Die 3 Färbbarkeit ist NP-vollständig.

1.  $\in$  NP

2. 3-SAT  $\leq_p$  3-Färbbarkeit

#### 27.2.1 Vorrichtungen (Gadgets)

Wir definieren uns ein (logisches) Gatter:

Lemma:

1. Wenn die beiden Eingänge die gleiche Farbe haben, dann hat der Ausgang ebenfalls diese Farbe.
2. Wenn die Eingänge verschiedene Farben haben, dann kann der Ausgang jede beliebige Farbe haben:

Als nächstes bauen wir uns ein ODER-Gatter mit 3 Eingängen:

#### 27.2.2 Reduktion

Gegeben Boolesche Formel  $\psi$  in 3-SAT Form.

Für jede Klausel, erstelle ein ODER-Gatter mit den entsprechenden Literalen als Eingabeknoten. Der Ausgang des Gatters wird mit dem F-Knoten des Basisdreiecks verbunden. (Beispiel:  $x_1 \vee \neg x_2 \vee x_3$ )

#### 27.2.3 Beweis

Behauptung:  $\psi$  erfüllbar  $\Leftrightarrow G$  ist 3-färbbar.

$\Rightarrow$  erfüllende Bedingung

$x_i, \neg x_i$  entsprechend Färben. Weil die Belegung alle Klauseln erfüllt, sind die Eingaben jedes Klauselgatters nicht (F,F,F), Man kann dann als Klauselgatter immer so färben, dass der Ausgang mit W gefärbt ist, das ist eine gültige Färbung von  $G$ .

$\Leftarrow$  Färbung sei gegeben.

o.B.d.A. Basisdreiecke gefärbt mit W,F,O wie in Abbildung.

$\Rightarrow x_i, \neg x_i$  sind mit W,F gefärbt, entgegengesetzt.

→ Belegung ablesen. Wenn eine Klausel nicht erfüllt ist, sind alle 3 Eingänge des entsprechenden ODER-Gatters  $F, F, F, \dots, F$  ist die einzige mögliche Farbe für den Ausgang. Widerspruch, weil Ausgang mit  $F$  verbunden ist.

$G = f(\psi)$  in polynomieller Zeit berechenbar. ✓

3-Färbbarkeit ist NP-vollständig sogar, wenn man es auf planare Graphen einschränkt.

## 27.3 Reduktion von gewöhnlichen 3-FARB auf 3-FARB in planaren Graphen

Skizze der Reduktion: Beliebige Zeichnung von  $G$ ; Elimination der Kreuzungen.

In jeder gültigen Färbung des Achtecks bekommen  $r$  und  $t$ , sowie  $s$  und  $u$  die gleichen Farben.

## 27.4 Teilmengensumme (TMS)

(Spezialfall des Rucksackproblems)

Eingabe:  $n$  Zahlen  $a_1, a_2, \dots, a_n \in \mathbb{N}$ , Zielsumme  $b$

Frage: gibt es eine  $I \subseteq \{1, \dots, n\}$  mit  $\sum_{i \in I} a_i = b$

Satz: TMS ist NP-vollständig

### 27.4.1 Beweis

1. TMS  $\in$  NP ✓

2. 3-SAT  $\leq_p$  TMS

Gegeben Boolesche Formel  $\psi$  mit  $n$  Variablen und  $m$  Klauseln:

$2n$  Literalzahlen  $a_i, a_j$  (entsprechend  $x_i, x_j$ )

$a_1 = \underbrace{1}_{x_1} \underbrace{0}_{x_2} 0 \dots \underbrace{0}_{x_n} \underbrace{0000}_{c_1} \underbrace{0001}_{c_2} \dots \underbrace{0001}_{c_m}$ , falls  $x_1$  in Klauseln 2 und  $m$  vorkommt.

$b_1 = \underbrace{1}_{x_1} \underbrace{0}_{x_2} 0 \dots \underbrace{0}_{x_n} \underbrace{0001}_{c_1} \underbrace{0000}_{c_2} \dots \underbrace{0}_{c_m}$ , falls  $\neg x_1$  in Klausel 1 vorkommt.

$b = \underbrace{1}_{x_1} \underbrace{0}_{x_2} 0 \dots \underbrace{0}_{x_n} \underbrace{0003}_{c_1} \underbrace{0003}_{c_2} \dots \underbrace{0003}_{c_m}$

Für jede Klausel  $j$  2 Füllerzahlen:

$d_j = e_j = 00000000 \underbrace{0001}_{c_j} 000$

Eingabe für TMS:  $a_1, \dots, a_n, b_1, \dots, b_n, d_1, \dots, d_m, e_1, \dots, e_m$

$b$

Behauptung:  $\psi$  erfüllbar  $\Leftrightarrow$  TMS lösbar

$\Rightarrow$  Belegung gegeben

wähle  $d_i$  oder  $b_i$  aus, je nachdem, ob  $x_i = W$  oder  $x_i = F$

In jeder Klauselspalte  $j$  ist die Summe 1, 2, 3 = (Anzahl der wartenden Literale)

Nimm  $d_j + e_j$  bzw.  $d_j$  bzw. gar nichts zur Lösung dann.

Summe = 111111...133333...3

$\Leftarrow$  Lösung in TMS gegeben. Bei Bildung der Summe gab es keinen Übertrag.

Gesamtsumme in den linken Spalten = 2, in den rechten Spalten = 5  $\Rightarrow$  Summenbedingung muss für jede Dezimalstelle gelten.  $a_i$  oder  $b_i$  ausgewählt, aber nicht beide.  $\rightarrow$  Belegung.

Summe aller  $a_i/b_i$  Variablen ist mindestens 1 in jeder Klauselspalte.  $\rightarrow$  jede Klausel ist erfüllt.

## 27.5 Rucksackproblem

Das Ruckproblem ist NP-vollständig. Laufzeit  $\mathcal{O}(nb)$  (dynam. Prog.)

Einen Algorithmus, der polynomiell in der Größe der Zahlen und nicht unbedingt in der Kodierungslänge der Zahlen (binär) ist, nennt man pseudopolynomiell.

## 28 Komplexitätstheorie (Vorlesung 24 am 19.01.)

### 28.1 Stark-NP-vollständig

Ein Problem heißt stark-NP-vollständig (stark-NP-schwer), wenn es auch dann noch NP-vollst. (-schwer) ist, wenn die Eingabe vollkommenen Zahlen unär kodiert werden: Kodelänge für eine Zahl  $x \in \mathbb{N}$  ist  $x + 1$ . (Bspw.  $9 = 1111111110$ )

Algorithmen mit dieser Eingabekodierung heißen pseudopolynomielle Algorithmen.

### 28.2 Rundreiseproblem ist stark-NP-vollständig

Beweis:

1.  $\in \text{NP}$

2.  $\text{HAMMILTONKREIS} \leq \text{RUNDREISEPROBLEM}$  (unär)

Diese Reduktion ist auch bei unärer Eingabe eine polynomielle Reduktion: die auftretenden Zahlen sind  $1, 2, n$ .

$3\text{-SAT} <_p \text{Teilmenge}$

### 28.3 coNP

$= \{A \subseteq \Sigma^* \mid \Sigma^* - A \in \text{NP}\}$

Probleme in NP, mit vertauschter Antwort JA  $\leftrightarrow$  NEIN

Beispiel: Primzahltest: Eingabe:  $n \in \mathbb{N}$  Frage: Ist  $n$  eine Primzahl? Dieses Problem ist in coNP.

Zertifikat: zwei Faktoren  $p, q > 1$  mit Produkt  $p \cdot q = n$

Das Problem ist sogar in P!

coNP-vollständig, coNP-schwer, analog zu NP.

### 28.4 PSPACE

Probleme die in polynomieller Speicherplatz lösbar.

PSPACE-vollst. - PSPACE-schwer

Landkarte der Komplexitätsklassen:

### 28.5 Turing-Reduzierbarkeit

Ein Berechnungsproblem  $A$  ist NP-schwer, wenn man mit seiner Hilfe (man darf das Problem  $A$  beliebig oft als Unterprogramm aufrufen) jedes Problem in NP in polynomieller Zeit lösen könnte. (bisher)  $\leq_p$  - komplette Reduzierbarkeit: die Eingabe von  $B$  in eine Eingabe von  $A$  umrechnen und darf dann den Algorithmus von  $A$  einmal verwenden und muss die Antwort (JA / NEIN) unverändert übernehmen.

$x \in B \Leftrightarrow f(x) \leftarrow A$

## 29 Approximationsalgorithmen

### 29.1 amortisierte Analyse

Beispiel isotone Regression:  $Q = \emptyset$

wiederhole  $n$ -mal:

$Q.\text{insert}(\dots)$



while bla

...

Q.deletemax()

Ein Schleifendurchlauf kann bis zu  $\mathcal{O}(n)$  Zeit dauern. In Summe also locker in  $\mathcal{O}(n^2)$ . Aber trotzdem ist die Gesamtlaufzeit nur in  $\mathcal{O}(n)$ , weil aus  $Q$  nicht öfters gelöscht, als eingefügt werden. Die amortisierte Laufzeit eines Schleifendurchlaufes ist  $\mathcal{O}(1)$ .

Definition: Datenstrukturen mit Operationen  $A, B, C, \dots$

Wir sagen, die Operationen haben amortisierte Laufzeit  $T_A(n), T_B(n), T_C(n), \dots$  bei Eingabe der Größe  $n$ , wenn jede Folge von  $n_A$  Operationen  $A$ ,  $n_B$  Operationen  $B, \dots$  auf einer leeren Datenstruktur beginnend mit:  $\mathcal{O}(n_A * T_A(n) + n_B * T_B(n) + n_C * T_C(n) + \dots)$  Zeit ausgeführt werden kann.

### 29.1.1 Bsp: Fibonacci-Heap

Fib-Heaps sind Prioritätswarteschlange, die die Operationen:

1. insert
2. findmin
3. merge
4. decreasekey

in  $\mathcal{O}(1)$  amortisierter Zeit und

1. deletemin

in  $\mathcal{O}(\log n)$  amortisierter Zeit ausführen können.

Anwendung: Algorithmus von Dijkstra mit  $n$  Knoten und  $m$  Kanten.

1.  $n$  insert  $\mathcal{O}(1)$
2.  $m$  decreasekey  $\mathcal{O}(1)$
3.  $n$  deletemin  $\mathcal{O}(\log n)$

$\rightarrow \mathcal{O}(n \log n + m)$

## 29.2 Binomialbäume

$B_0$  = Wurzel

Induktive Definition eines Binomialbaumes  $B_i$  vom Rang  $i$  (=Höhe  $i$ ):

$B_i$  erhält man, indem man an die Wurzel von  $B_{i-1}$  eine weitere Kopie von  $B_{i-1}$  als zusätzliches letztes Kind der Wurzel anhängt.

$B_i$  hat  $2^i$  Kinder, und zwar  $\binom{i}{k}$  Kinder auf Tiefe  $k$ .

Eine Binomialhalde ist eine Liste von Binomialbäumen mit verschiedenen Rang, wobei die Knoten in jedem Baum die Haldeneigenschaft haben. Schlüssel eines Knotens  $\geq$  Schlüssel des Elternknotens.

Beispiel: Binomialhalde mit 11 Knoten von  $n = 11 = 8 + 2 + 1 = 1011_2$

Struktur ist durch die Knotenanzahl  $n$  eindeutig vorgegeben. Es gibt eine Binomialhalde  $\leq \log_2 n$  Binomialbäume.

... findmin  $\mathcal{O}(\log n)$  Zeit (Durchsuchen der Wurzeln).

### 29.2.1 merge

Vereinigung von zwei Binomialhalden.

Zwei  $B_i$ s können in Konstanter Zeit zu einem  $B_{i+1}$  kombiniert werden:

Vergleich der Wurzeln: größere Wurzel wird als Kind an die kleinere gehängt.

Algorithmus: verschmelze die beiden Listen von Binomialbäumen zu einer Liste, nach Rang sortiert.

for  $i = 0, 1, 2, \dots \lfloor \log n \rfloor$

while  $\exists$  zwei Bäume  $B_i$ : vereinige sie zu einem Baum  $B_{i+1}$   $B_i = \text{Knoten mit Unterbaum } B_{i-1}$

Laufzeit:  $\mathcal{O}(\log n)$

### 29.2.2 insert

merge mit einer neuen Binomialhalde mit +1 Knoten:  $\mathcal{O}(\log n)$

### 29.2.3 deletemin

finde die Wurzel mit dem kleinsten Schlüssel. lösche sie, vereinige die Knoten (die selbst eine Binomialhalde bilden) mit den übrigen Wurzeln (merge) ...  $\mathcal{O}(\log n)$  Zeit.

## 30 Fibonacci-Halden (Vorlesung 25 am 23.01.)

Ist eine Spezialanwendung von Binomialhalden.

Bei F-Halden kann ein Knoten ein Kind verlieren, ohne dass der Baum dadurch umgebaut werden muss.

Eine F-Halde ist eine Liste von Bäumen, die die Haldeneigenschaft haben. Die Wurzeln sind in einer ungeordneten Liste organisiert.

### 30.1 decreasekey(x)

$x$  bekommt einen kleinen Wert. Die Kante von  $x$  zum Elternknoten ist möglicherweise nicht mehr gültig. Lösche diese Elternkante und mache  $x$  zur Wurzel.

Der Elternknoten von  $x$  wird markiert, zum Zeichen, dass er ein Kind verloren hat. Wenn  $x'$  schon markiert war, löse  $x'$  ebenfalls aus dem Bau und mache  $x'$  zu einer Wurzel.  $\Rightarrow$  Elementknoten  $x''$  von  $x'$  (sofern vorhanden) markieren, bzw. selbst aus dem Baum herausschneiden.

Eine einzelne decreasekey-Operation führt möglicherweise zu einer ganzen Kaskade von *Schnittoperationen*.

Bemerkung: Wurzelknoten sind nie markiert.

### 30.2 merge

Vereinigung von 2 F-Halden. Die beiden Wurzellisten aneinander hängen.

### 30.3 deletemin

1. min unter den Wurzeln finden. Diesen Knoten entfernen und seine Kinder zu Wurzeln machen.
2. Wurzelliste aufräumen, sodass von jedem Grad höchstens ein Baum übrig ist:  
Solange es 2 Wurzeln mit dem gleichen Grad  $i$  gibt, vereinige diese zu einem Baum mit Grad  $i + 1$

#### 30.3.1 mögliche Implementierung

---

```
1 Feld B[0], ..., B[d_{max}] mit BOTTOM initialisiert
2 Eingabeliste L
3 while notempty(L)
4   v = L.pop()
5   i = Grad(v)
6   if B[i] = BOTTOM then B[i] := v
7   else: vereinige v und B[i] zu v'
8       L.insert(v')
9 for i = 0 to d_{max}
10  if B[i] != BOTTOM then L.insert(B[i])
```

---

Laufzeit:  $\mathcal{O}(|L| + d_{max})$

$L_0$  ist die Ausgangsliste, die Whileschleife wird höchstens  $|L_0| + |L_0|$ -mal durchlaufen, denn in jedem Durchlauf wird aus  $L$  ein Element entfernen.

## 30.4 Invariante

Die Kinder eines Knotens  $v$  seien  $v_1, v_2, \dots, v_k$  in der Reihenfolge, in der sie Kinder geworden sind. Dann ist der  $\text{Grad}(v_i) \begin{cases} \geq i - 1 & \text{falls } v_i \text{ unmarkiert ist} \\ \geq i - 2 & \text{falls } v_i \text{ markiert ist} \end{cases}$

### 30.4.1 Beweis

$v$  verliert ein Kind ✓

Ein Kind von  $v$  wird markiert ✓

$v$  bekommt ein zusätzliches Kind  $v'$ .  $v'$  war vorher eine Wurzel und ist daher unmarkiert. ✓

## 30.5 Kleinstmögliche F-Bäume mit Grad $k$

siehe Bäume im Originalskript.

Der Grad der Bäume lässt sich wie folgt berechnen:  $k = k - 1 + k - 2$  und ist  $\mathcal{O}(\log n)$ , ebenfalls gilt:  $|F_k| = |F_{k-1}| + |F_{k-2}| \approx \left(\frac{\sqrt{5}+1}{2}\right)^k$

## 30.6 Amortisierte Analyse

Wir geben der Datenstruktur ein gewisses Budget vor, dass die Datenstruktur für Operationen ausgeben kann oder für einen späteren Zeitpunkt sparen kann.

1 Taler reicht für eine Operation mit Laufzeit  $\mathcal{O}(1)$ .

Jeder markierte Knoten hat 2 Taler. Jeder Wurzelknoten hat 1 Taler.

Der Taler im Wurzelknoten bezahlt für das Aufräumen.  $L^{vor} L^{nach}$  Liste vor bzw. nach dem Aufräumen.  $|L^{nach}| \leq d_{\max}$

### 30.6.1 Laufzeit

$$\begin{aligned} & \mathcal{O}(d_{\max} + |L^{vor}|) \\ &= \mathcal{O}(d_{\max} + (|L^{vor}| - |L^{nach}|) + \underbrace{|L^{nach}|}_{\leq d_{\max}}) \\ &= \mathcal{O}\left(\underbrace{d_{\max}}_{\text{kostet } d_{\max} = \mathcal{O} \log n \text{ Taler}} + \underbrace{|L^{vor}| - |L^{nach}|}_{\text{wird aus gespeicherten Talern bezahlt}}\right) \end{aligned}$$

### 30.6.2 Analyse für decreasekey

Wir stellen 4 Taler zur Verfügung. Elternknoten von  $x'$  unmarkiert:

- 1 Taler für  $x$  als neue Wurzel
- 2 Taler zur Markierung von  $x'$
- 1 Taler bleibt zum Bezahlen der Operationen.

Elternknoten von  $x'$  markiert:

- 1 Taler für  $x$  als neue Wurzel
- 1 Taler zum Bezahlen der Laufzeit.

Bleiben 2 Taler, zusammen mit den 2 die in  $x'$  gespeichert sind  $\rightarrow$  4 Taler.

### 30.6.3 Satz: Amortisierte Laufzeit

Betrachte eine Folge von  $A$  Operationen insert, decreasekey, merge und  $B$  Operationen deletemin auf eine anfangs leere F-Halde, die zu jedem Zeitpunkt höchstens  $n$  Elemente enthält. Dann ist die Laufzeit  $\mathcal{O}(A + B \log n)$

Wir sagen: insert, decreasekey, merge hat eine amortisierte Laufzeit von  $\mathcal{O}(1)$ , deletemin hat amortisierte Laufzeit von  $\mathcal{O}(\log n)$ .

### 30.6.4 Potentialfunktion

Amortisierte Analyse mit einer Potentialfunktion  $\Psi$ .  $\Psi$  ist eine Funktion, die von Zustand der Datenstruktur abhängt. Beispiel F-Halden:  $\Psi = 2\#(\text{markierte Knoten}) + \#(\text{Wurzeln})$

Bei einer Operation mit tatsächlicher Laufzeit  $T$ , definieren wir die amortisierte Laufzeit  $T^a = T - \Psi^{vor} + \Psi^{nach}$

Wenn  $\Psi$  zu Beginn  $= 0$  ist und nie negativ werden kann, dann gilt:

Summe der Tatsächlichen Laufzeiten  $\leq$  Summe der amortisierten Laufzeiten.

(Beweis in anderer Mitschrift)

## 31 Kürzeste Spannbäume, UNION-FIND-Problem (Vorlesung 26 am 26.01.)

### 31.1 Algorithmus von Kruskal für einen kürzesten Spannbaum

1. Sortiere die Kanten nach Gewicht  $w(e_1) \leq w(e_2) \leq \dots \leq w(e_n)$
2.  $T = \emptyset$   
for  $i = 1, \dots, n$   
Kante  $e_i = (u, v)$   
Falls  $u$  und  $v$  in verschiedenen Komponenten\* von  $T$  liegen, setze  $T = T \cup \{e_i\}$

\* ist der Kern des UNION-FIND-Problems.

### 31.2 UNION-FIND-Problem

Grundmenge von  $n$  Elementen (z.B.  $\{1, 2, \dots\}$ ).

Zu Beginn: jedes Element bildet eine eigene Klasse. Partition  $\{\{1\}, \{2\}, \{3\}, \dots, \{n\}\}$

#### 31.2.1 Operationen

1. Gehören 2 Elemente,  $u, v$  zur selben Klasse? (FIND)
2. Vereinige 2 Klassen zu einer größeren Klasse. (UNION)

#### 31.2.2 FIND( $u$ )

Gibt einen Repräsentanten der Klasse zurück, zu der  $u$  gehört.

$\text{FIND}(u) = \text{FIND}(v) \Leftrightarrow u$  und  $v$  gehören zur selben Klasse.

Der Repräsentant kann ein ausgewähltes Element der Klasse, oder z.B. eine Klassennummer sein.

#### 31.2.3 UNION( $a, b$ )

Modifiziert die Klassenzerteilung so, dass die Klassen mit Repräsentanten  $a$  und  $b$  vereinigt werden. ( $a, b$  sind Ergebnisse der FIND-Operation)

#### 31.2.4 Implementierungsmöglichkeiten

- A: Klassen als verkettete Listen speichern. Jedes Element hat einen Zeiger zu seinen Repräsentanten. Die Repräsentanten gehören nicht zur Grundmenge. Jeder Repräsentant hat einen Zeiger auf die Liste der Elemente.

FIND:  $\mathcal{O}(1)$

UNION: Eine der Listen durchlaufen und alle Repräsentantenzeiger umspeichern.  $\mathcal{O}(n)$

A': Jeder Repräsentant speichert die Größe seiner Klasse.

UNION: Umspeichern der kleineren Klasse. Laufzeit  $\mathcal{O}(n)$

ABER: Gesamtlaufzeit aller UNION-Operationen ist  $\mathcal{O}(n \log n)$

Begründung: Die Größe der Klasse, zu der ein Knoten  $u$  gehört, wird mindestens verdoppelt, wenn die Repräsentantenzeiger von  $u$  geändert wird. Das passiert  $\leq \mathcal{O}(\log n)$ -mal.

**B:** Speichern der Menge als Baum. Repräsentant ist die Wurzel eines Baumes.  
 UNION: Einen Baum in den anderen Baum einfügen, indem die Wurzeln verbunden werden.  
 Laufzeit  $\mathcal{O}(1)$ .  
 FIND: Verfolge einen Pfad zur Wurzel. Laufzeit  $\mathcal{O}(n)$

**B':** UNION nach Größe (Anzahl der Knoten). Der kleinere Baum wird als Kind an den größeren gehängt. (Jedesmal, wenn man zu seinem Elternknoten geht und die Größe des Teilbaums mindestens verdoppelt.) Laufzeit für FIND:  $\mathcal{O}(\log n)$

**B'':** UNION nach Rang (Rang=Tiefe). Die Wurzel mit dem kleineren Rang wird als Kind an die andere Wurzel gehängt. Bei Gleichstand wird der Rang der Wurzel um 1 erhöht.  
 Invariante:

- (a) Ein Knoten vom Rang  $r$  enthält mindestens  $2^r$  Knoten in seinem Teilbaum
- (b) Erweiterungen gehen zu einem Knoten mit größerem Rang.

**C:** Speichern als Baum, FIND mit Pfadkompression.

Wenn FIND( $u$ ) eine Folge von Knoten  $u = v_1, v_2, \dots, v_k$  durchläuft, dann setze anschließend Elternzeiger von  $v_1, v_2, \dots, v_{k-2}$  auf  $v_k$ .

Laufzeit von FIND wird verdoppelt, ABER Ersparnis bei zukünftigen FIND-Operationen.

**C':** UNION nach Rang, FIND mit Pfadkompression

Grundmenge mit  $n$  Elementen.  $\leq n - 1$  UNION Operationen.  $m$  FIND-Operationen.

Laufzeit:  $\mathcal{O}(m + n \log n)$  wegen **B''**.

Empirisch  $\mathcal{O}(m + n)$

zunächst  $\mathcal{O}(n + m \log \log n)$

später  $\mathcal{O}(n + m \log^* n)$

$\log^* n = \min\{i \mid \underbrace{\log_2, \log_2, \log_2, \dots, \log_2}_{i\text{-mal}} n \leq 1\}$  endgültig  $\mathcal{O}(n + m \alpha(m, n))$  (R.E. Tarjan)

$\alpha$  ist die inverse Ackermannfunktion, also eine unglaublich langsam wachsende Funktion die in der Praxis nicht mehr von  $(m + n)$

### 31.3 Analyse von C'

Werdegang eines Knoten: Rang steigt in Schritten von 1, solange er eine Wurzel ist. Danach ist er konstant  $= r$ . Der Rang des Elternknoten ist immer  $> r$  und kann steigen.

Seidel und Staur: Wir haben einen Wald  $F$  gegeben: Eine verallgemeinerte Kompressionsfolge für  $F$  ist eine Folge von Pfaden  $P_1, P_2, \dots, P_m$ . Jeder Pfad  $P$  ist:

- (a) ein interner Pfad:  $v_1, v_2, \dots, v_k (k \geq 1), v_{i+1}$  ist Elternknoten von  $v_i$
- (b) ein Geisterpfad: wie oben, nur  $v_{k-1}$  ist Wurzel  $v_k = \text{BOTTOM} (k \geq 2)$

Kompression ersetzt die Elternzeiger von  $v_1, v_2, \dots, v_{k-2}$  durch  $v_k$  (bei einem Geisterpfad werden alle Knoten zu Wurzeln).

Setze  $F_0 = F$

$F_1 = P_i$  angewendet auf  $F_{i-1}$ . Wir müssen annehmen, dass  $P_i$  ein gültigen Pfad für  $F_{i-1}$  ist.

Kosten eines internen Pfades  $= \max 0, k - 2 = \text{Anzahl der Zeigeränderungen}$

Kosten eines Geisterpfades  $= 0$

Kosten der Folge  $C = (P_1, \dots, P_m) = \text{cost}(C) = \text{Summe der Kosten der Pfade}$ . Länge von  $C = l(C) = \#$  der internen Pfade.

## 32 UNION-FIND-Fortsetzung (Vorlesung 27 am 30.01.)

Wiederholung:

Vereinigung nach Rang.

Finde mit Pfadkompression

Hohe Ränge tauchen im Verlauf des Algorithmus weniger häufig auf. Daher formulieren wir folgendes Lemma:

Knoten, die im Laufe des Algorithmus Rang  $r$  haben ist  $\frac{n}{2^r}$

*Beweis.* Wie entsteht ein Knoten vom Rang  $r$ ?

Eine Wurzel wird Kind und ändert durch den Rang nicht mehr.

Induktion:  $r = 0$  ✓

$r-1 \rightarrow r$  ✓

□

Wir haben eine Folge von UNION-FIND Operationen, die wir so anordnen, dass erst alle UNION-Operationen ausgeführt werden und dann die FIND-Operationen ausgeführt werden. Außerdem verallgemeinern wir die Pfadkompression.

Lemma: Gegeben sei eine Folge  $X$  von UNION-FIND-Operationen, ausgehend von der Partition  $\{\{1\}, \{2\}, \dots, \{n\}\}$ , mit  $m$  FIND-Operationen und  $\leq n-1$  UNION-Operationen. Dann existiert ein Wald  $F$  und eine Folge  $\mathcal{C}$  von verallgemeinerten Kompressionen:

(i)  $l(\mathcal{C}) = m$

(ii) Laufzeit von  $X = \mathcal{O}(m + n + \text{cost}(\mathcal{C}))$

*Beweis.* Zuerst alle UNION-Operationen.

Sei  $F$  der entstehende Wald. Anschließend betrachte alle FIND-Operationen:

Es sei  $\text{FIND}(u) = v$ . Füge zu  $\mathcal{C}$  den internen Pfad von  $u$  nach  $v$  dazu.

□

Kosten für FIND =  $\mathcal{O}(1) + \underbrace{\mathcal{O}(\text{Länge des Weges})}_{\leq \text{cost}(P)+2}$

$P$  = Pfad  $v_1, v_2, \dots, v_k$  von  $u$  nach  $v$ .  $\text{cost}(P) = \max\{k-2, 0\}$

Wir ordnen jedem Knoten einen festen Rang zu

= Höhe im Wald  $F$  (erster Wald)

= endgültiger Rang in der Folge  $X$

Dann gibt es  $\leq \frac{n}{2^r}$  Knoten mit Rang  $\geq r$

Wir zerlegen einen Wald  $F$  in einen oberen Wald  $F_+$  bestehend aus dem Knoten mit Rang  $> S$  und einen Wald  $F_-$ , bestehend aus Knoten mit Rang  $\leq S$ . Knoten in  $F_-$ , die einen Elternknoten in  $F_+$  haben, werden zu Wurzeln.

### 32.1 Zerlegungslemma

Gegeben Wald  $F$  mit verallgemeinerte Kompressionsfolge  $\mathcal{C}$ , Zerlegung  $F = F_+ \cup F_-$ . Dann gibt es verallgemeinerte Kompressionsfolgen  $\mathcal{C}_+$  für  $F_+$  und  $\mathcal{C}_-$  für  $F_-$  mit

(i)  $l(\mathcal{C}) = l(\mathcal{C}_+) + l(\mathcal{C}_-)$

(ii)  $\text{cost}(\mathcal{C}) = \underbrace{\text{cost}(\mathcal{C}_+)}_{1.} + \underbrace{\text{cost}(\mathcal{C}_-)}_{2.} + \underbrace{|F_-|}_{3.} + \underbrace{l(\mathcal{C}_+)}_{4.}$



*Beweis.* Wege, die nur in  $F_+$  oder nur in  $F_-$  verlaufen, werden zu  $C_+$  bzw.  $C_-$  zugeteilt.  
 Wege  $P$ , die in  $F_-$  beginnen und in  $F_+$  enden: werden aufgespalten in  $P_- = P \cap F_-$  und  $P_+ = P \cap F_+$ .  $P_+$  wird zu  $C_+$  hinzugefügt.  $P_-$  wird zu einem Geisterpfad  $P_- \cup \{\perp\}$  gemacht und zu  $C_-$  hinzugefügt. (Wird bei  $l(C_-)$  nicht mitgezählt  $\rightarrow$  (i) erfüllt.)  $\square$

## 32.2 Die 4 Ereignisse von $\text{cost}(\mathcal{C})$

1. Knoten aus  $F_+$  bekommt einen neuen Elternknoten in  $F_+$
2. Knoten aus  $F_-$  bekommt einen neuen Elternknoten in  $F_-$
3. Knoten aus  $F_-$ , dessen Elternknoten in  $F_-$  lag, bekommt einen neuen Elternknoten in  $F_+$
4. Knoten aus  $F_-$ , dessen Elternknoten in  $F_+$  lag, bekommt einen neuen Elternknoten in  $F_+$

Anmerkungen: 3. und 4. treten höchstens 1x pro Pfad auf. 4. hat die Zusatzbedingung, dass es nur eintritt, wenn die Pfade einen neuen Pfad in  $P_+ \in C_+$

Stufe 0  $F$  Wald mit  $n$  Knoten,  $\text{Rang} \leq r$

$$\text{cost}(\mathcal{C}) \leq r * n$$

$$\text{cost}(\mathcal{C}_+) \leq |F_+| * r \leq \frac{n}{r} * r = n$$

*Beweis.*  $\text{cost}(\mathcal{C})$  zählt, wie oft ein Knoten einen neuen Elternknoten bekommt. Der Rang des Elternknotens muss um  $\geq 1$  steigen.  $\Rightarrow \leq r$  mal pro Knoten.  $\square$

Stufe 1 Zerlegungslemma für  $s = \lceil \log_2 r \rceil$ ,  $r =$  Schranke für den Rang in  $F$ .

$$\text{cost}(\mathcal{C}) \leq \underbrace{\text{cost}(\mathcal{C}_+)}_{\leq n} + \underbrace{\text{cost}(\mathcal{C}_-)}_{\leq n} + \underbrace{|F_-|}_{\leq n} + \underbrace{l(\mathcal{C}_+)}_{l(\mathcal{C}) - l(\mathcal{C}_-)}$$

$$|F_+| = \text{Knoten mit Rang} \geq s + 1 \leq \frac{n}{2^{s+1}} \frac{n}{r}$$

$$[\text{cost}(\mathcal{C} - l(\mathcal{C}))] \leq 2n + [\text{cost}(\mathcal{C}_-) - l(\mathcal{C}_-)]$$

$$[\text{cost}(\mathcal{C}_- - l(\mathcal{C}_-))] \leq 2n + [\text{cost}(\mathcal{C}_{--}) - l(\mathcal{C}_{--})]$$

$$\text{Gleiches Spiel mit } F_- = \underbrace{F_{--}}_{F_{\leq s'}} \cup F_{> s'}$$

$$s' = \lceil \log 2s \rceil = \lceil \log \log r \rceil$$

$$F \dots \text{Rang} \geq r$$

$$F_- \dots \text{Rang} \geq \log r$$

$$F_{--} \dots \text{Rang} \geq \log \log r$$

...

$$F_0 \text{ Rang} \leq 1$$

$$\rightarrow \log^* \text{ Mal.}$$

$$\Rightarrow \text{cost}(\mathcal{C}) \leq l(\mathcal{C}) + 2n \log^* r$$

$$L_0(r) = \lceil \log_2 r \rceil$$

$L_j(r)$  = wie oft muss ich  $L_{j-1}$  auf  $r$  anwenden, bis das Ergebnis  $\leq 1$  wird.

$$L_1(r) = \log^* r$$

$$L_j(r) = \begin{cases} 0, & \text{falls } r \leq 1 \\ 1 + L_j(L_{j-1}(r)), & r \geq 2 \end{cases}$$