

# Datenstrukturen und effiziente Algorithmen

Skript zur Vorlesung aus den Mitschriften von

Markus Vieth

David Klopp

11. April 2016



# Vorwort

Dieses Skript basiert auf unserer Mitschrift der Vorlesung Datenstrukturen und effiziente Algorithmen (DSeA) im WS 2015/16 an der JGU Mainz (Dozent: Prof. Dr. E. Schömer).

Es handelt sich nicht um eine offizielle Veröffentlichung der Universität.

Wir übernehmen keine Gewähr für die Fehlerfreiheit und Vollständigkeit des Skripts.

Fehler können unter [Github](#) gemeldet werden. Die aktuelle Version dieses Skriptes ist ebenfalls auf [Github](#) zu finden.



# Inhaltsverzeichnis

<b>Vorwort</b>	<b>i</b>
<b>I. Sortieren und Suchen</b>	<b>1</b>
<b>1. Bubblesort</b>	<b>2</b>
1.1. Pseudocode . . . . .	2
1.2. Laufzeitanalyse . . . . .	3
<b>2. Heapsort</b>	<b>4</b>
2.1. Skizze . . . . .	4
2.1.1. Indices innerhalb der Baumstruktur . . . . .	4
2.2. Heap-Eigenschaft . . . . .	4
2.3. Idee . . . . .	5
2.4. Pseudocode . . . . .	5
2.4.1. Phase: Bottom-up Strategie zum Heapaufbau . . . . .	5
2.4.2. Phase: Sortierphase . . . . .	5
2.5. Korrektheitsbetrachtung . . . . .	5
2.6. Laufzeitanalyse . . . . .	6
2.6.1. Zusammenhang von $n$ und $k$ . . . . .	6
2.6.2. Analyse Heapaufbau . . . . .	7
2.6.3. Sortierphase . . . . .	7
2.6.4. Fazit . . . . .	8
<b>3. Landau-Notation</b>	<b>9</b>
3.1. $O(n)$ . . . . .	9
3.2. $\Omega(n)$ . . . . .	9
3.3. $\Theta(n)$ . . . . .	9
3.4. $o(n)$ . . . . .	9
3.4.1. Notation . . . . .	10
<b>4. Mergesort (Divide and Conquer)</b>	<b>11</b>
4.1. Pseudo-Code . . . . .	11
4.2. Laufzeitanalyse . . . . .	11
<b>5. Master-Theorem</b>	<b>13</b>
5.1. Fall 1 : $\left(\frac{a}{b^\alpha}\right) < 1$ . . . . .	13
5.2. Fall 2 : $\left(\frac{a}{b^\alpha}\right) > 1$ . . . . .	14
5.3. Fall 3 : $\left(\frac{a}{b^\alpha}\right) = 1$ . . . . .	14
5.3.1. Beispiel: Mergesort . . . . .	14
<b>6. Schnelle Multiplikation langer Zahlen</b>	<b>15</b>
6.1. Schulmethode zur Multiplikation . . . . .	15
6.2. Karazuba Ofman . . . . .	15

<b>7. Akra-Bazzi Theorem</b>	<b>17</b>
7.1. Beispiel . . . . .	17
7.2. Das Theorem . . . . .	17
7.2.1. Ergänzung für den allgemeinen Fall . . . . .	17
7.3. Beweisidee . . . . .	18
<b>8. Lineare Rekursionsgleichungen</b>	<b>19</b>
8.1. Fibonacci-Zahlen . . . . .	19
8.2. Methode der erzeugenden Funktionen . . . . .	19
8.2.1. Einschub: Beispiel Reihenentwicklung . . . . .	19
8.2.2. Nullstellen des Nennerpolynoms . . . . .	20
8.2.3. Partialbruchzerlegung . . . . .	20
8.2.4. Lösung . . . . .	20
<b>9. Quicksort (Divide and Conquer)</b>	<b>21</b>
9.1. Pseudo-Code . . . . .	21
9.1.1. Zufallspermutation . . . . .	22
9.1.2. Einschub: Stochastik . . . . .	23
9.1.3. Laufzeitanalyse . . . . .	23
<b>10. Median in Linearzeit</b>	<b>25</b>
10.1. Verallgemeinerung . . . . .	25
10.2. Idee . . . . .	25
<b>11. Quickselect</b>	<b>26</b>
11.1. Quicksort . . . . .	26
11.2. Quickselect . . . . .	26
<b>12. Verallgemeinerung von Akra-Bazzi</b>	<b>27</b>
<b>13. Median der Mediane</b>	<b>28</b>
13.1. Ansatz . . . . .	28
13.2. Deterministische Variante für k-Select . . . . .	28
13.3. Laufzeitanalyse für den worst-case . . . . .	28
<b>14. Untere Schranke für vergleichsbasierte Sortierverfahren</b>	<b>29</b>
14.1. Entscheidungsproblem: (Bubblesort) . . . . .	29
14.2. Worst-case Laufzeit . . . . .	30
14.2.1. Lemma: Mittlere Tiefe der Blätter in einem Entscheidungsbaum $> \log_2(n)n$ . . . . .	30
<b>15. Radix-Sort</b>	<b>32</b>
15.1. Beispiel: . . . . .	32
15.2. Pseudo-Code . . . . .	32
<b>16. Binäre Suchbäume</b>	<b>33</b>
16.1. Pseudo-Code . . . . .	34
<b>17. AVL-Bäume von Adelson-Velsky and Landis</b>	<b>35</b>
17.1. Allgemein . . . . .	35
17.2. Laufzeitanalyse . . . . .	35

17.3. Baumtiefe . . . . .	37
17.3.1. AVL-Eigenschaft: . . . . .	37
17.4. Rotationen . . . . .	38
17.5. Pseudo-Code . . . . .	39
<b>18. (a,b)-Suchbäume</b>	<b>40</b>
18.1. Aufspaltung bei Einfügen . . . . .	41
18.2. Verschmelzen von Knoten beim Löschen . . . . .	41
<b>19. Amortisierte Analyse</b>	<b>42</b>
19.1. Bankkonto-Methode . . . . .	42
19.1.1. Kontoführungsschema: für Binärzähler . . . . .	43
19.1.2. Kontoführungsschema für einen 2-5 Baum . . . . .	43
<b>20. Hashing</b>	<b>44</b>
20.1. Laufzeiten der Operationen . . . . .	44
20.1.1. Analyse der Laufzeit zum Einfügen eines neuen Elementes $k$ . . . . .	44
20.1.2. Analyse der Suchzeit für einen Schlüssel $k$ . . . . .	45
20.1.3. Laufzeit beim Löschen von Schlüssel $k$ . . . . .	45
20.2. Universelles Hashing . . . . .	45
20.2.1. Definition . . . . .	46
20.3. Perfektes Hashing . . . . .	47
20.3.1. Definition . . . . .	47
20.3.2. Nachteil . . . . .	48
<b>II. Graphen-Algorithmen</b>	<b>50</b>
<b>21. Einführung</b>	<b>51</b>
21.1. Eulerische Polyederformel . . . . .	51
21.2. Adjazenzmatrix . . . . .	52
21.3. Adjazenzlisten Repräsentation . . . . .	53
<b>22. BFS (Breadth-First Search) Breitensuche</b>	<b>54</b>
22.1. Pseudo-Code . . . . .	54
22.2. Laufzeit . . . . .	54
22.2.1. Definition: Länge kürzesten Weges . . . . .	56
22.2.2. Satz: Richtigkeit des Algorithmus . . . . .	56
22.2.3. Lemma 1: Dreiecksungleichung für kürzeste Wege . . . . .	56
22.2.4. Lemma 2 . . . . .	56
22.2.5. Lemma 3 . . . . .	56
22.2.6. Satz: Richtigkeit des Algorithmus . . . . .	57
<b>23. Kürzeste Wege Algorithmen</b>	<b>59</b>
23.1. Dijkstra-Algorithmus . . . . .	59
23.1.1. Vorläufige Laufzeitanalyse von Dijkstra . . . . .	61
23.2. Bellman-Ford-Algorithmus . . . . .	61
23.2.1. Pseudocode . . . . .	61
23.2.2. Laufzeit: Bellman-Ford . . . . .	61
23.2.3. Korrektheitsbeweis: Bellman-Ford . . . . .	62
23.2.4. Induktionsschritt: $i \rightarrow i + 1$ . . . . .	62

<b>24. All-Pairs-Shortest Path Algorithmen</b>	<b>63</b>
24.1. Allgemein . . . . .	63
24.1.1. Laufzeit zur Berechnung von $D^{(n)}$ . . . . .	64
24.2. Floyd-Warshall-Algorithmus . . . . .	64
24.2.1. Korrektheitsbeweis: . . . . .	64
24.2.2. Beweis der Invariante durch Induktion nach $k$ . . . . .	65
24.3. Naive Lösung . . . . .	65
24.4. Johnson-Algorithmus . . . . .	65
24.4.1. Laufzeit des Johnson-Algorithmus . . . . .	66
<b>25. Minimal aufspannende Bäume MST</b>	<b>67</b>
25.1. Greedy-Algorithmen zur Lösung des MST-Problems: . . . . .	67
25.2. Schnitt-Lemma: . . . . .	67
25.2.1. Beweis für das Schnitt-Lemma . . . . .	68
25.3. Algorithmus von Kruskal . . . . .	68
25.4. Einfache Union-Find-Datenstruktur . . . . .	69
25.4.1. Laufzeit Kruskal . . . . .	69
25.5. Prim-Algorithmus zur Berechnung eines MST . . . . .	70
25.5.1. Korrektheit des Prim-Algorithmus . . . . .	71
25.5.2. Laufzeit des Prim-Algorithmus . . . . .	71
25.5.3. Beispiel des Prim-Algorithmus: . . . . .	71
<b>26. Priority-Queue mittels Fibonacci-Heaps</b>	<b>72</b>
26.1. Operationen eines Binomial-Heaps . . . . .	72
26.1.1. Konsolidierung der Wurzelliste . . . . .	74
26.2. Eigenschaften . . . . .	74
26.2.1. Lemma: Knotenanzahl im Unterbaum . . . . .	74
26.2.2. Beweis . . . . .	74
26.2.3. Satz: Kosten der Operationen . . . . .	76
26.2.4. Beweis . . . . .	76
<b>27. Das Heiratsproblem</b>	<b>77</b>
27.1. Allgemein . . . . .	77
27.1.1. Lemma: (Berge) . . . . .	78
27.1.2. Beweis: . . . . .	78
27.1.3. Pseudo-Code . . . . .	79
27.1.4. Laufzeit . . . . .	79
27.2. Hopcroft-Karp-Algorithmus . . . . .	79
27.2.1. Gesamtzahl der Schleifendurchläufe . . . . .	80
<b>28. Algorithmen zur Berechnung maximaler Flüsse</b>	<b>81</b>
28.1. Allgemein . . . . .	81
28.1.1. Flusserhaltung . . . . .	81
28.1.2. Definition: Fluss . . . . .	81
28.1.3. Definition: Schnitt $(S, T)$ . . . . .	82
28.1.4. Konstruktion des Restnetzwerk $G_f$ . . . . .	82
28.2. Ford-Fulkerson-Algorithmus . . . . .	82
28.3. Restnetzwerk $G_f = (V, E_f)$ . . . . .	83
28.4. Max-Flow-Min-Cut-Theorem . . . . .	83
28.4.1. Beweis: 1. $\Rightarrow$ 2. . . . .	83



28.4.2. Idee . . . . .	83
28.4.3. Zwick . . . . .	85
28.5. Edmonds-Karp Algorithmus . . . . .	87
28.5.1. Lemma: . . . . .	87
28.5.2. Beweis durch Widerspruch . . . . .	87
28.5.3. Lemma . . . . .	88
28.5.4. Beweis . . . . .	88
28.5.5. Laufzeitanalyse von Edmonds-Karp Algorithmus . . . . .	89
28.6. Algorithmus von Dinic . . . . .	89
28.6.1. Sperrfluss (blocking flow) . . . . .	89
28.7. Definition: Sperrfluss . . . . .	89
28.7.1. Pseudo-Code . . . . .	90
28.7.2. Begründung zur Laufzeit . . . . .	90
28.8. Maximum Matching als Flussproblem . . . . .	91
28.8.1. Flussnetzwerke mit Einheitskapazität . . . . .	91
28.8.2. Finden knotendisjunkter Wege . . . . .	93
28.8.3. Ergänzung zum Paper . . . . .	93
<b>29. Dynamische Programmierung</b>	<b>94</b>
29.1. Matrizen Multiplikation . . . . .	94
29.1.1. Bottom-Up Ansatz . . . . .	95
29.1.2. Programm . . . . .	95



Teil I.

# Sortieren und Suchen

# 1. Bubblesort

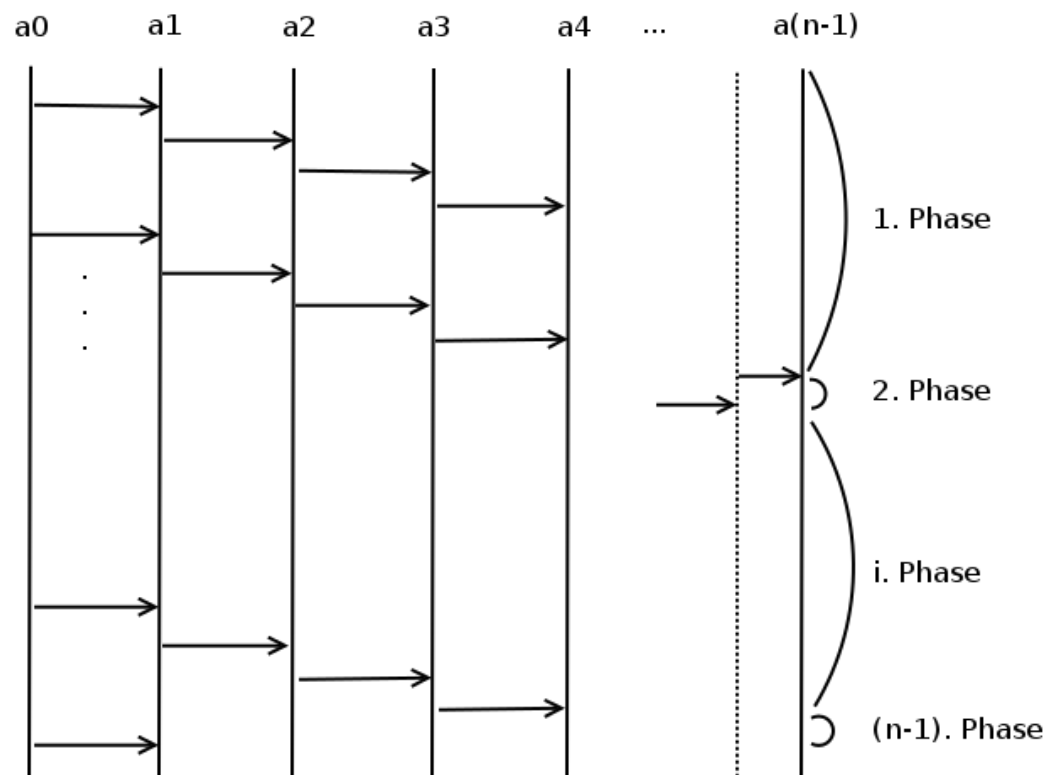


Abbildung 1.1.: Bubblesort

## 1.1. Pseudocode

```
1 void bubblesort (int[] a) {  
2   int n = a.length;  
3   for (int i = 1; i < n; i++) {  
4     for (int j = 0; j < n-i; j++) {  
5       if (a[j] > a[j+1])  
6         swap (a, j, j+1);  
7     }  
8   }  
9 }
```

**Schleifen-Invariante:** Nach dem Ablauf der  $i$ -ten Phase gilt:

Die Feldpositionen  $n-i, \dots, n$  enthalten die korrekt sortierten Feldelemente

**Beweis** durch Induktion nach  $i \stackrel{i=n-1}{\implies}$  Sortierung am Ende korrekt.

## 1.2. Laufzeitanalyse

$T(n)$  = Zahl der durchgeführten Elementvergleiche für eine Eingabemenge von  $n$  Elementen

1.	Phase	$n-1$
2.	Phase	$n-2$
3.	Phase	$n-3$
	$\vdots$	
i.	Phase	$n-i$
	$\vdots$	
(n-1).	Phase	$n-(n-1)=1$
<hr/>		
$1 + 2 + 3 + \dots + (n-1)$		

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

$n$	$T_{real}$
$2^{10}$	8ms
$2^{11}$	11ms
$2^{12}$	26ms
$\vdots$	
$2^{16}$	5,819s
$2^{17}$	23,381s
$\vdots$	
$2^{20}$	16min
$\vdots$	
$2^{26}$	52d

$$T_{real}(n) \approx cn^2 \quad c \approx 10^{-6}$$

$$T_{real}(2n) \approx c \cdot (2n)^2 = 4cn^2 = 4T_{real}(n)$$

$$\frac{T_{real}(2n)}{T_{real}(n)} = 4$$

## 2. Heapsort

z.B. 21 6 4 7 12 5 3 11 14 17 19 8 9 10 42

### 2.1. Skizze



Abbildung 2.1. Heapsort (Ausgangssituation)

#### 2.1.1. Indices innerhalb der Baumstruktur



Abbildung 2.2.: Indices

### 2.2. Heap-Eigenschaft

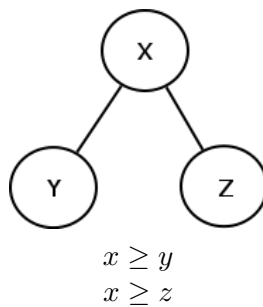


Abbildung 2.3.: Heap-Eigenschaft

## 2.3. Idee

**Phase 1** Stelle die Heap-Eigenschaft überall her  
 $\Rightarrow$  größtes Element steht in der Wurzel

**Phase 2** Tausche Wurzel mit letztem Feldelement  
 (z.B. 42 mit 3)  
 - Entferne letztes Feldelement aus dem Baum  
 - Gehe erneut zu Phase 1

## 2.4. Pseudocode

```

1 void heapify (int[] a, int i, int n) {
2   while (2i + 1 < n) { //linkes Kind von i existiert
3     int j = 2i + 1;
4     if (2i + 2 < n) //rechtes Kind von i existiert
5       if (a[j] < a[j+1])
6         j = j + 1; //j steht für Index des größten Kindes
7     if (a[i] > a[j]) //Vater größer als Kind
8       break; //Abbruch, weil heap bereits erfüllt
9     swap(a,i,j); //Tausch zwischen Vater und Kind
10    i = j;
11  }
12 }
```

### 2.4.1. Phase: Bottom-up Strategie zum Heapaufbau

```

1 for (int i = n/2; i ≥ 0; i--)
2   heapify(a,i,n);
```

### 2.4.2. Phase: Sortierphase

```

1 for (int i = n-1; i ≥ 0; i--) {
2   swap(a,0,i);
3   heapify(a,0,i);
4 }
```

## 2.5. Korrektheitsbetrachtung

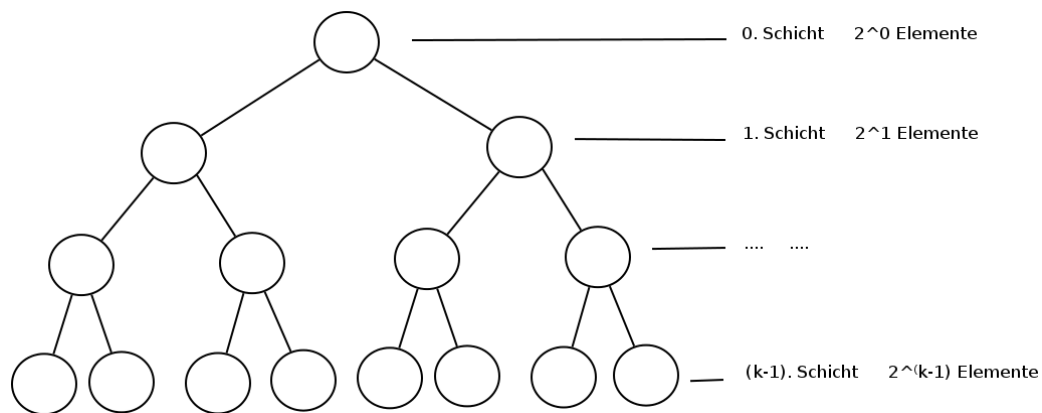
**Invariante beim Heapaufbau:** Beim Durchlauf der for-Schleife wird die Heapeigenschaft vom unteren Baumlevel bis zur Wurzel hergestellt.

**Invariante für Sortierphase:** Nach jedem weiteren Durchlauf der for-Schleife findet ein weiteres Element am Feldende seinen „richtigen Platz“.

## 2. Heapsort

### 2.6. Laufzeitanalyse

$T(n)$  = Zahl der Elementvergleiche.



$n$  := Zahl der Elemente

$k$  := Zahl der Schichten

#### 2.6.1. Zusammenhang von $n$ und $k$

$$n = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

**Merke: Geometrische Reihe**

$$\sum_{i=0}^{k-1} x^i = \frac{1 - x^k}{1 - x} \quad \text{mit } x \neq 1$$



### 2.6.2. Analyse Heapaufbau

$$\sum_{l=0}^{k-1} 2^l (k-1-l)$$

$2^l :=$  Anzahl Knoten auf Level  $l$

$(k-1-l) :=$  Leveldifferenz zwischen  $l$  und der Blattebene

$$\sum_{l=0}^{k-1} (k-1-l) \cdot 2^l = \sum_{l=0}^{k-1} (k-1) \cdot 2^l - \sum_{l=0}^{k-1} l \cdot 2^l = (k-1)(2^k - 1) - 2 \sum_{l=1}^{k-1} l 2^{l-1}$$

#### Nebenrechnung

$$\begin{aligned} \sum_{i=1}^{k-1} i \cdot x^{i-1} &= \frac{d}{dx} \left( \sum_{i=0}^{k-1} x^i \right) = \frac{d}{dx} \left( \frac{x^k - 1}{x - 1} \right) \\ &= \frac{kx^{k-1}(x-1) - (x^k - 1)}{(x-1)^2} \end{aligned}$$

mit  $x = 2$  folgt:  $k \cdot 2^{k-1} - 2^k + 1$

$$\begin{aligned} &= (k-1)(2^k - 1) - k2^k + 2^{k+1} - 2 \\ &= k2^k - 2^k - k + 1 - k2^k + 2^{k+1} - 2 \\ &= -2^k - k - 1 + 2^{k+1} \leq 2^{k+1} \approx 2 \cdot n \end{aligned}$$

$\Rightarrow$  Heapaufbau in lineare Zeit

### 2.6.3. Sortierphase

**1. Versuch**  $n \cdot k$  mit  $n = 2^k - 1 \Leftrightarrow k = \log_2(n+1) \approx n \cdot \log_2(n)$

**2. Versuch (mit Verkleinerung der Liste)**

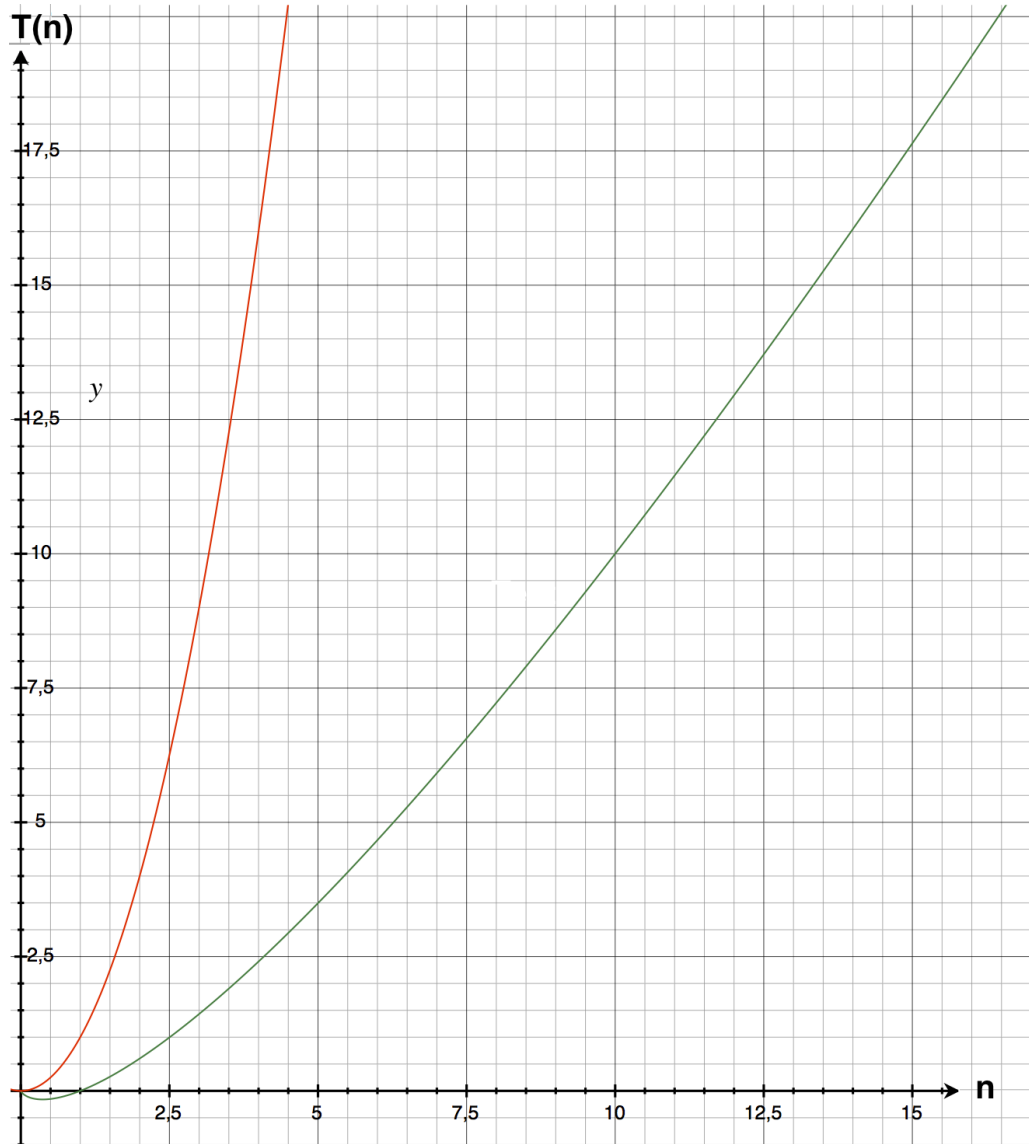
$$\sum_{l=0}^{k-1} 2^l \cdot l = 2 \sum_{l=1}^{k-1} l \cdot 2^{l-1} = 2 \cdot (k \cdot 2^{k-1} - 2^k + 1) \geq k \cdot 2^{k-1} \approx n \cdot \log_2(n)$$

## 2. Heapsort

### 2.6.4. Fazit

**Laufzeit**  $c \cdot n \cdot \log_2(n)$  wobei  $c \in \mathbb{R}$

**Vergleich** Bubblesort  $\leftrightarrow$  Heapsort



### 3. Landau-Notation

$$g, f : \mathbb{N} \rightarrow \mathbb{N}$$

#### 3.1. $O(n)$

$$g(n) \in O(f(n)) \Leftrightarrow c > 0 \wedge n_0 \in \mathbb{N}, \text{ so dass f\"ur alle } n \geq n_0 \text{ gilt: } g(n) \leq c \cdot f(n) \Leftrightarrow \lim_{n \rightarrow \infty} \sup \frac{g(n)}{f(n)} < \infty$$

**Beispiel**

$$\lim_{n \rightarrow \infty} \frac{n \log_2(n)}{n^2} = \lim_{n \rightarrow \infty} \frac{\log_2(n)}{n} = \lim_{n \rightarrow \infty} \frac{\frac{\ln(n)}{\ln(2)}}{n} \stackrel{\text{L' Hopital}}{=} \lim_{n \rightarrow \infty} \frac{1}{\ln(2)} \cdot \frac{1}{n} = \frac{1}{\ln(2)} \lim_{n \rightarrow \infty} \frac{1}{n} = 0$$

#### 3.2. $\Omega(n)$

$$g(n) \in \Omega(f(n)) \Leftrightarrow c > 0 \wedge n_0 \in \mathbb{N}, \text{ so dass f\"ur alle } n \geq n_0 \text{ gilt: } g(n) \geq c \cdot f(n) \Leftrightarrow \lim_{n \rightarrow \infty} \inf \frac{g(n)}{f(n)} > 0$$

**Beispiel**  $g(n) = n^p \quad f(n) = n^q \quad p \geq q$

**Behauptung**  $g(n) \in \Omega(f(n))$

$$\lim_{n \rightarrow \infty} \frac{n^p}{n^q} = \infty > 0$$

#### 3.3. $\Theta(n)$

$$g(n) \in \Theta(f(n)) \Leftrightarrow g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$$

**Beispiel**  $g(n) = n^p + n^{p-1} + c \cdot n^2 \quad f(n) = n^p$

**Behauptung**  $g(n) \in \Theta(f(n))$

.... Rechnung

#### 3.4. $o(n)$

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

**Beispiel**  $g(n) = n \cdot \log_2(n) \quad f(n) = n^2$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0 \quad \text{siehe oben}$$

### 3. Landau-Notation

**Erklärung** "g ist asymptotisch gesehen vernachlässigbar gegenüber f."

#### 3.4.1. Notation

Häufig wird:

$$O(n) = O(n^2) = O(n^2 \cdot \log_2(n))$$

geschrieben, anstelle von:

$$O(n) \subset O(n^2) \subset O(n^2 \cdot \log_2(n))$$

Missbrauch der Notation !!!

## 4. Mergesort (Divide and Conquer)

### 4.1. Pseudo-Code

```
1  int[] a; //Eingabefeld
2  int[] b; //Hilfsfeld

4  void mergesort(int links, int rechts) {
5      if (links ≥ rechts) return;
6      int mitte = (links+rechts)/2;
7      mergesort(links, mitte);
8      mergesort(mitte+1, rechts);
9      merge(links, mitte, rechts);
10 }

12 void merge(int links, int mitte, int rechts) {
13     int i = links;
14     int j = mitte+1;
15     int k = links;
16     while (i ≤ mitte && j ≤ rechts) {
17         if (a[i] < a[j])
18             b[k++] = a[i++];
19         else
20             b[k++] = a[j++];
21     }
22     while (i ≤ mitte)
23         b[k++] = a[i++];
24     while (j ≤ rechts)
25         b[k++] = a[j++];
26     for (k=links; k ≤ rechts; k++)
27         a[k] = b[k];
28 }
```

### 4.2. Laufzeitanalyse

$T(n)$  = Zahl der von Mergesort durchgeführten Elementarvergleiche  $\approx$  Laufzeit

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 \approx 2T\left(\frac{n}{2}\right) + n \quad \text{mit } T(1) = 0$$

Korrekt wäre  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$

Für ungerade Zahlen

#### 4. Mergesort (Divide and Conquer)

$$T(n) = 2 \cdot 2T\left(\frac{n}{2}\right) + n \stackrel{(1)}{=} 2 \left( 2T\left(\frac{n}{4}\right) + \frac{n}{2} \right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$\stackrel{(2)}{=} 4 \cdot \left( 2T\left(\frac{n}{8}\right) + \frac{n}{4} \right) + 2n = 8T\left(\frac{n}{8}\right) + 3n = \dots = 2^i \cdot T\left(\frac{n}{2^i}\right) + in$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2} \quad (1)$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} \quad (2)$$

....

$$T(1) = 0$$

**Rekursionsende**  $\frac{n}{2^i} = 1 \Leftrightarrow 2^i = n \Leftrightarrow i = \log_2(n)$

$$T(n) = 2^{\log_2(n)} T\left(\frac{n}{2^{\log_2(n)}}\right) + n \log_2(n) = nT(1) + \log_2(n) = \log_2(n)$$

#### Abstraktion

$T(n)$  = Laufzeit eines Divide & Conquer Algorithmus der ein Problem dadurch löst, das es in  $a$  Teilprobleme der Größe  $\frac{n}{b}$  zerlegt wird, die rekursiv gelöst werden und anschließend kombiniert werden.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^\alpha \quad \alpha > 0 \quad \text{mit } T(1) = 0$$

$$T(n) = aT\left(\frac{n}{b}\right) + n^\alpha \stackrel{(1)}{=} a^2T\left(\frac{n}{b^2}\right) + a\left(\frac{n}{b}\right)^\alpha + n^\alpha$$

$$(1) \quad T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^\alpha \stackrel{(2)}{=} a^3T\left(\frac{n}{b^3}\right) + a^2\left(\frac{n}{b^2}\right)^\alpha + a^1\left(\frac{n}{b^1}\right)^\alpha + a^0\left(\frac{n}{b^0}\right)^\alpha$$

$$(2) \quad T\left(\frac{n}{b^2}\right) = aT\left(\frac{n}{b^3}\right) + \left(\frac{n}{b^2}\right)^\alpha = a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j \left(\frac{n}{b^j}\right)^\alpha = a^iT\left(\frac{n}{b^i}\right) + n^\alpha \sum_{j=0}^{i-1} \left(\frac{a}{b^\alpha}\right)^j$$

$$\text{mit } i = \log_b(n) \wedge x = \frac{a}{b^\alpha}$$

...

$$T(1) = 0$$

## 5. Master-Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^\alpha$$

$$T(1) = 0$$

$$T(n) = a^i T\left(\frac{n}{b^i}\right) + n^\alpha \sum_{j=0}^{i-1} \left(\frac{a}{b^\alpha}\right)^j$$

o.B.d.A

$$n = b^k \Leftrightarrow k = \log_b(n)$$

### 5.1. Fall 1 : $\left(\frac{a}{b^\alpha}\right) < 1$

$$\left(\frac{a}{b^\alpha}\right) < 1 \Leftrightarrow a < b^\alpha \Leftrightarrow \log_b(a) < \alpha$$

$$\sum_{j=0}^{k-1} x^j = \frac{x^k - 1}{x - 1} \quad \text{für } x \neq 1$$

$$\Rightarrow \sum_{j=0}^{k-1} \left(\frac{a}{b^\alpha}\right)^j \leq \frac{1}{1 - \frac{a}{b^\alpha}} = c'$$

$$\begin{aligned} T(n) &= a^k T(1) + n^\alpha \cdot c' \\ &= c \cdot n^{\log_b(a)} + c' \cdot n^\alpha = \Theta(n^\alpha) \end{aligned}$$

$$\text{Nebenbedingung} \quad a^{\log_b(n)} = \left(b^{\log_b(a)}\right)^{\log_b(n)} = \left(b^{\log_b(n)}\right)^{\log_b(a)} = n^{\log_b(a)}$$

## 5. Master-Theorem

### 5.2. Fall 2 : $\left(\frac{a}{b^\alpha}\right) > 1$

$$\left(\frac{a}{b^\alpha}\right) > 1 \Leftrightarrow \log_b(a) > \alpha$$

$$\sum_{j=0}^{k-1} \left(\frac{a}{b^\alpha}\right)^j = \left(\frac{\left(\frac{a}{b^\alpha}\right)^{\log_b(n)} - 1}{\left(\frac{a}{b^\alpha}\right) - 1}\right) \leq \left(\frac{a}{b^\alpha}\right)^{\log_b(n)} \cdot c'' = \frac{a^{\log_b(n)}}{b^{\alpha \log_b(n)}} = \frac{n^{\log_b(\alpha)}}{n^\alpha}$$

$$T(n) = c \cdot n^{\log_b(a)} + n^\alpha \cdot \frac{n^{\log_b(a)}}{n^\alpha} \cdot c'' = \Theta\left(n^{\log_b(a)}\right)$$

### 5.3. Fall 3 : $\left(\frac{a}{b^\alpha}\right) = 1$

$$\left(\frac{a}{b^\alpha}\right) = 1 \Leftrightarrow a = b^\alpha \Leftrightarrow \log_b(a) = \alpha$$

$$T(n) = c \cdot n^{\log_b(a)} + n^\alpha \cdot \log_b(n) = \Theta(n^\alpha \cdot \log(n))$$

#### 5.3.1. Beispiel: Mergesort

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

$$T(1) = 0$$

**Ermittle**    $a = 2$     $b = 2$     $\alpha = 1$

$$\log_2(2) = 1 = \alpha \Rightarrow 3. \text{ Fall} \Rightarrow \Theta(n \cdot \log(n))$$



## 6. Schnelle Multiplikation langer Zahlen

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline a_{n-1} & \dots & a_i & \dots & a_2 & a_1 & a_0 \\ \hline \end{array} \quad a_i \in \mathbb{B} = \{0, 1\}$$

$$= \sum_{i=0}^{n-1} a_i 2^i$$

$$B = \begin{array}{|c|c|c|c|c|} \hline b_{n-1} & \dots & b_2 & b_1 & b_0 \\ \hline \end{array}$$

$$= \sum_{i=0}^{n-1} b_i 2^i$$

**Frage** Wie schnell können wir zwei n-stellige Binärzahlen addieren/subtrahieren/multiplizieren ?

**Addition**  $\Theta(n)$

### 6.1. Schulmethode zur Multiplikation

1	0	1	1	0	1	·	0	1	0	1	1	1		
		0	0	0	0	0	0							
			1	0	1	1	0	1						
				0	0	0	0	0	0					
					1	0	1	1	0	1				
						1	0	1	1	0	1			
							1	1	0	1	1	0	1	
1	0	0	0	0	0	0	0	1	0	1	1	1		

n-Partialprodukte  
mit höchstens  
2n Ziffern

$n^2$  Aufwand zur Ermittlung der Partialprodukte +  $n \cdot$  Kosten für die Addition von Zahlen der Länge  $2n \Rightarrow \Theta(n^2)$

**Ziel**  $o(n^2)$   $O(n^{1,58})$

### 6.2. Karazuba Ofman

$$A = \begin{array}{|c|c|c|} \hline a_{n-1} & \dots & a_{\frac{n}{2}} \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline a_{\frac{n}{2}-1} & \dots & a_0 \\ \hline \end{array}$$

$$= A_1 \quad = A_0$$

$$A = A_0 + A_1 2^{\frac{n}{2}}$$

$$A \cdot B = (A_0 + A_1 2^{\frac{n}{2}})(B_0 + B_1 2^{\frac{n}{2}})$$

$$= A_0 B_0 + A_0 B_1 2^{\frac{n}{2}} + A_1 B_0 2^{\frac{n}{2}} + A_1 B_1 2^n$$

## 6. Schnelle Multiplikation langer Zahlen

**Legende** ■ markierte Elemente haben die Länge  $\frac{n}{2}$

**Anmerkung** Addition von Zahlen der Länge  $2n$

Sei  $T(n)$  die Laufzeit dieser rekursiven Methode zur Multiplikation zweier  $n$ -stelliger Zahlen:

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + c \cdot n \quad T(1) = c$$

### Mastertheoreme

$$a = 4 \quad b = 2 \quad \alpha = 1 \quad \log_2(4) = 2 > \alpha$$

$$\Rightarrow T(n) = \Theta(n^2)$$

$\Rightarrow$  kein Gewinn bisher!!!

**Ziel** Ermittle Partialprodukte auf anderem Weg

$$1.) (A_0 \text{ ■ } A_1) \text{ ■ } (B_0 \text{ ■ } B_1) = A_0B_0 + A_0B_1 + A_1B_0 + A_1B_1 = P$$

$$2.) A_0 \text{ ■ } B_0$$

$$3.) A_1 \text{ ■ } B_1$$

$$\Rightarrow (A_0B_1 + A_1B_0) \cdot 2^{\frac{n}{2}} = (P \text{ ■ } (A_0B_0) \text{ ■ } (A_1B_1)) \cdot 2^{\frac{n}{2}}$$

Es verbleiben 3 Multiplikationen und ■ Additionen

$$AB = A_0B_0 \text{ ■ } (P - (A_0B_0) - (A_1B_1)) \cdot 2^{\frac{n}{2}} \text{ ■ } A_1B_1 2^n$$

### Mastertheoreme

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + n$$

$$a = 3, \quad b = 2, \quad \alpha = 1$$

$$\log_2(3) > 1 \quad \Rightarrow 2. \text{ Fall}$$

$$\Rightarrow \Theta\left(n^{\log_2(3)}\right) = \Theta\left(n^{1,5849625}\right)$$

## 7. Akra-Bazzi Theorem

### 7.1. Beispiel

$$T(n) = 2T\left(\frac{n}{2}\right) + \log_2(n)$$

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + g(n) & n > n_0 \\ h(n) & 1 \leq n \leq n_0 \end{cases}$$

$$T(n) = \Theta\left(n^\alpha \left(1 + \int_1^n \frac{g(x)}{x^{\alpha+1}} dx\right)\right) \quad \text{mit } \alpha, \text{ so dass gilt: } \frac{a}{b^\alpha} = 1$$

### 7.2. Das Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + g(n)$$

$$T(1) = c$$

$$T(n) = \Theta\left(n^\alpha \left(1 + \int_1^n \frac{g(x)}{x^{1+\alpha}} dx\right)\right) \quad \text{mit } \frac{a}{b^\alpha} = 1 \quad \alpha = \log_b(a)$$

$$\text{z.B. } T(n) = 2T\left(\frac{n}{2}\right) + \log(n)^I$$

#### 7.2.1. Ergänzung für den allgemeinen Fall (siehe Kapitel 12)

$$T(n) = \sum_{i=1}^k \left[ a_i \cdot T\left(\frac{n}{b_i}\right) \right] + g(n)$$

$$T(1) = c$$

$$T(n) = \Theta\left(n^\alpha \left(1 + \int_1^n \frac{g(x)}{x^{1+\alpha}} dx\right)\right) \quad \text{mit } \sum_{i=1}^k \frac{a_i}{b_i^\alpha} = 1$$

---

<sup>I</sup>klausurrelevant

## 7. Akra-Bazzi Theorem

### 7.3. Beweisidee

$$T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + g\left(\frac{n}{b}\right)$$

$$T(n) = a\left(aT\left(\frac{n}{b^2}\right) + g\left(\frac{n}{b}\right)\right) + g(n) = a^2 + \frac{n}{b^2} + a^1 g\left(\frac{n}{b^1}\right) + a^0 g\left(\frac{n}{b^0}\right)$$

$$\Rightarrow a^i T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j g\left(\frac{n}{b^j}\right) \quad \text{Rekursionsende für } i = \log_b(n)$$

$$\Theta(a^{\log_b(n)}) = \Theta(n^\alpha)$$

$$\sum_{j=0}^{\log(n)-1} a^j g\left(\frac{n}{b^j}\right) \approx \int_0^{\log_b(n)} a^j g\left(\frac{n}{b^j}\right) dj$$

#### Substitution

$$x = \frac{n}{b^j} = n \cdot b^{-j} = n \cdot e^{-j \ln(b)}$$

$$\frac{dx}{dj} = n(-\ln(b)) e^{-j \ln(b)} = -n \ln(b) b^j = -\ln(b) x$$

$$\Rightarrow dj = \frac{1}{-\ln(b)x} dx$$

$$a^j = b^{\log_b(a)j} = b^{\alpha j} = (b^j)^\alpha = \left(\frac{n}{x}\right)^\alpha$$

$$= \int_n^1 \left(\frac{n}{x}\right)^\alpha g(x) \left(\frac{1}{-\ln(b)x}\right) dx = \frac{n^\alpha}{\ln(b)} \cdot \int_1^n \frac{g(x)}{x^{1+\alpha}} dx$$

*q.e.d*

## 8. Lineare Rekursionsgleichungen

### 8.1. Fibonacci-Zahlen

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \\ f_0 &= 0 \\ f_1 &= 1 \end{aligned}$$

n	0	1	2	3	4	5	6	7	...
f(n)	0	1	1	2	3	5	8	13	...

Abbildung 8.1.: Fibonacci-Zahlen

### 8.2. Methode der erzeugenden Funktionen

$$\begin{aligned} F(Z) &= \sum_{n=0}^{\infty} f_n Z^n = f_0 \cdot Z^0 + f_1 \cdot Z^1 + \sum_{n=2}^{\infty} (f_{n-1} + f_{n-2}) \cdot Z^n \\ &= Z + \sum_{n=2}^{\infty} f_{n-1} Z^n + \sum_{n=2}^{\infty} f_{n-2} Z^n \\ &= Z + Z \sum_{n=2}^{\infty} f_{n-1} Z^{n-1} + Z^2 \sum_{n=2}^{\infty} f_{n-2} Z^{n-2} \\ &\Leftrightarrow F(Z) = Z + Z \cdot F(Z) + Z^2 \cdot F(Z) \\ &\Leftrightarrow -Z = Z^2 F(Z) + Z F(Z) - F(Z) = F(Z)(Z^2 + Z - 1) \\ &F(Z) = -\frac{Z}{Z^2 + Z - 1} \end{aligned}$$

#### 8.2.1. Einschub: Beispiel Reihenentwicklung

$$\frac{1}{1-Z} = \sum_{n=0}^{\infty} Z^n$$

$$\Rightarrow F(Z) = -\frac{Z}{Z^2 + Z - 1}$$

## 8. Lineare Rekursionsgleichungen

### 8.2.2. Nullstellen des Nennerpolynoms

$Z^2 + Z = 1 \quad   + \left(\frac{1}{2}\right)^2$ $\Leftrightarrow \left(Z + \frac{1}{2}\right)^2 = \frac{5}{4}$ $\Leftrightarrow Z_{1/2} = -\frac{1}{2} \pm \frac{\sqrt{5}}{2}$ $\Rightarrow Z^2 + Z + 1 = (Z + \phi)(Z + \bar{\phi})$	<b>Goldener Schnitt</b> $\phi = \frac{1+\sqrt{5}}{2}$ $\bar{\phi} = \frac{1-\sqrt{5}}{2}$
--	--

### 8.2.3. Partialbruchzerlegung

$$\frac{A}{Z + \phi} + \frac{B}{Z + \bar{\phi}} = \frac{A \cdot (Z + \bar{\phi}) + B(Z + \phi)}{(Z + \phi)(Z + \bar{\phi})}$$

$$\Rightarrow AZ + BZ = -Z \Leftrightarrow A + B = -1 \quad (1)$$

$$A\bar{\phi} + B\phi = 0 \Leftrightarrow B = -\frac{A\bar{\phi}}{\phi} \quad (2)$$

$$(2) \text{ in } (1) \quad A - \frac{A\bar{\phi}}{\phi} = -1 \Leftrightarrow A \left(1 - \frac{\bar{\phi}}{\phi}\right) = -1$$

$$\Leftrightarrow A = -\frac{1}{\sqrt{5}}\phi$$

$$\Rightarrow B = \frac{1}{\sqrt{5}}\bar{\phi}$$

### 8.2.4. Lösung

$$F(Z) = \frac{-Z}{Z^2 + Z + 1} = -\frac{1}{\sqrt{5}} \frac{\phi}{Z + \phi} + \frac{1}{\sqrt{5}} \frac{\bar{\phi}}{Z + \bar{\phi}}$$

$$= \frac{1}{\sqrt{5}} \left( \frac{1}{1 + \frac{Z}{\phi}} - \frac{1}{1 + \frac{Z}{\bar{\phi}}} \right) = \frac{1}{\sqrt{5}} \left( \frac{1}{1 - \phi Z} - \frac{1}{1 - \bar{\phi} Z} \right)$$

$$= \frac{1}{\sqrt{5}} \left( \sum_{n=0}^{\infty} (\phi Z)^n - \sum_{n=0}^{\infty} (\bar{\phi} Z)^n \right) = \sum_{n=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n) \cdot Z^n$$

$$f_n = \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n) \quad \text{mit } \phi = 1,681... \quad \bar{\phi} = -0,681...$$

## 9. Quicksort (Divide and Conquer)



Abbildung 9.1.: Quicksort

⇒ Tausche 15 mit 10  
- Bewege Zeiger erneut  
⇒ Tausche 5 und 24  
- Bewege Zeiger erneut  
⇒ Tausche 16 und 2  
- Bewege Zeiger erneut  
⇒ i wird größer als j ⇒ Abbruch (tausche Pivotelement mit letztem Element in Teilliste 1)  
⇒ es ergeben sich zwei Teillisten

4, 6, 3, 10, 7, 9, 5, 2 |12| 16, 24, 42, 15

**best-case**  $T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log n)$

**worst-case**  $T(n) = T(n-1) + n = \Theta(n^2)$

### 9.1. Pseudo-Code

```
1 void quicksort(int[] a, int links, int rechts) {
2     if (links >= rechts) return;
3     int mitte = partition(a, links, rechts);
4     quicksort(a, links, mitte-1);
5     quicksort(a, mitte+1, rechts);
6 }

8 int partition(int[] a, int links, int rechts) {
9     int r = random(links, rechts);
10    swap(a, r, rechts);
11    int pivot = a[rechts];
12    int i = links;
13    int j = rechts-1;
14    while (i <= j) {
15        if (a[i] > pivot) {
16            swap(a, i, j);
17            j--;
18        } else {
19            i++;
20        }
21    }
```

## 9. Quicksort (Divide and Conquer)

```
22     swap(a, i, rechts);  
23     return i;  
24 }
```



Abbildung 9.2.: Quicksort

### Schleifen-Invariante:

$a[k] > p$  für  $j < k < rechts$

$a[k] \leq p$  für  $links < k < i$

#### 9.1.1. Zufallspermutation

```
1  void randomPermutation(int[] a) {  
2      int n = a.length;  
3      for (int i = n-1; i > 0; i--) {  
4          int r = random(0,i); // gleichverteilte Zufallszahl im Intervall [0,i)  
5          swap(a,r,i);  
6      }  
7  }
```



### 9.1.2. Einschub: Stochastik

#### Fairer Würfel (Erwartungswert):

X sei Zufallsvariable  $\hat{=}$  Anzahl Augen

$$Pr(X = x_i) \quad x_i \in \{1, 2, 3, 4, 5, 6\}$$

$$E(X) = \sum_{i=1}^6 x_i \cdot Pr(X = x_i) = \frac{1}{6} \cdot \sum_{i=1}^6 x_i = \frac{1}{6} \cdot \frac{7 \cdot 6}{2} = 3,5$$

#### Fairer Würfel (Erste Sechs):

X sei Zufallsvariable  $\hat{=}$  Zahl der benötigten Würfe bis zum Auftreten der ersten 6.

$$x_i \in \mathbb{N}$$

$$E(X) = \sum_{i=1}^{\infty} i \cdot Pr(X = i) = \frac{1}{6} \cdot \sum_{i=1}^{\infty} i \cdot \left(\frac{5}{6}\right)^{i-1}$$

Mit der Ableitung der geometrischen Reihe,  $\frac{1}{(1-x)^2}$  folgt:

$$= \frac{1}{6} \cdot \left( \frac{1}{\left(1 - \frac{5}{6}\right)^2} \right) = 6$$

### 9.1.3. Laufzeitanalyse

$T(n)$  = Erwartungswert der Laufzeit von Quicksort bei zufällig gleichverteilter Eingabe-Partition.

$$T(n) = n + \sum_{i=1}^n (T(i-1) + T(n-i)) \cdot \frac{1}{n}$$



Abbildung 9.3.

## 9. Quicksort (Divide and Conquer)

### Lösen durch Einsetzen

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

$$\Leftrightarrow n \cdot T(n) = n^2 + 2 \sum_{i=0}^{n-1} T(i)$$

$$\Leftrightarrow 2(n-1) \cdot T(n-1) = (n-1)^2 + 2 \sum_{i=0}^{n-2} T(i)$$

$$\Leftrightarrow (1) - (2)nT(1) - (n-1)T(n-1) = n^2 - (n-1)^2 + 2T(n-1)$$

$$\Leftrightarrow nT(n) = (n+1)T(n-1) + 2n - 1$$

$$\Leftrightarrow T(n) \leq \frac{n+1}{n} T(n-1) + 2 \leq \frac{n+1}{2} \left( \frac{n}{n+1} \cdot T(n-2) + 2 \right) + 2$$

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

$$\leq \frac{n+1}{n-1} \left( \frac{n-1}{n-2} T(n-3) + 2 \right) + \frac{n+1}{n} \cdot 2 + 2 \cdot 1$$

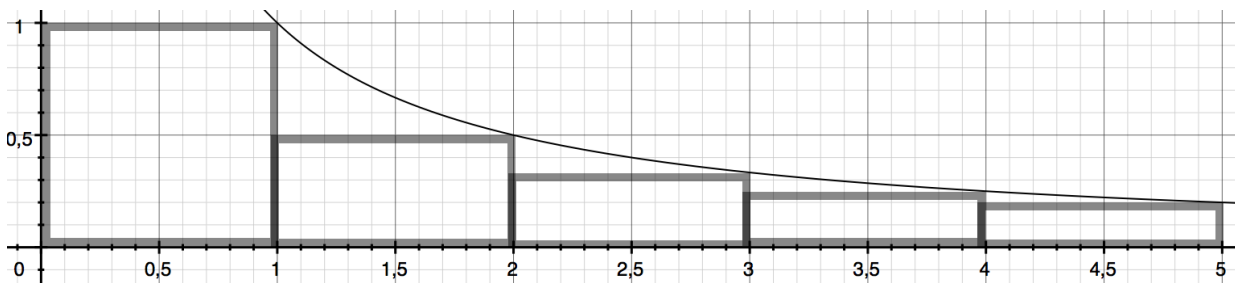
$$= \frac{n+1}{n-2} T(n-3) + 2 \cdot \frac{n+1}{n-1} + 2 \cdot \frac{n+1}{n} + 2 \cdot \frac{n+1}{n+1}$$

$$\Rightarrow T(n) \leq \frac{n+1}{n-(k-1)} T(n-k) + 2(n+1) \sum_{i=1}^{k-2} \frac{1}{n-i} \quad \text{endet für } k = n-1$$

$$T(n) = 2(n+1) \sum_{i=-1}^{n-3} \frac{1}{n-i} = 2(n+1) \sum_{j=3}^{n+1} \frac{1}{j} \leq 2(n+1) H_{n+1} \in O(n \log n) \quad \text{mit } j=n-i$$

### Einschub: Harmonische Reihe

$$H_n = \sum_{i=1}^n \frac{1}{i}$$



$$H_n \leq \int_1^n \frac{1}{x} dx + 1 = 1 + [\ln x]_1^n = \ln(n) + 1$$

$$\ln(n+1) \leq H_n \leq \ln(n) + 1$$

## 10. Median in Linearzeit

Median  $\hat{=}$   $\frac{n}{2}$ -kleinste Element in einer Folge von  $n$  Elementen

### 10.1. Verallgemeinerung

Finde das  $k$ -t kleinste Elemente in der Folge

Naive Strategie:  $O(k \cdot n)$

### 10.2. Idee

```
1 select(int[] a, int k) {}
```

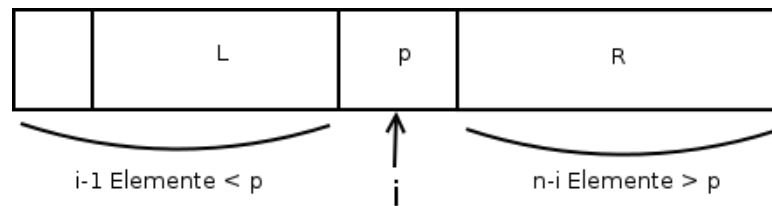


Abbildung 10.1.: Aufruf von Partition

1. **Fall**  $k = i \Rightarrow$  Pivotelement war gesucht
2. **Fall**  $k < i \Rightarrow$  suche rekursiv das  $k$ -t kleinste Element in  $L$
3. **Fall**  $k > i \Rightarrow$  suche rekursiv das  $(k-i)$ -t kleinste Element in  $R$

# 11. Quickselect

## 11.1. Quicksort

$$T(n) = \frac{1}{n} \sum_{i=1}^n T(i-1) + T(n-i) + n \in O(n \log(n))$$

## 11.2. Quickselect

$$T(n) = n + \frac{1}{n} \sum_{i=1}^n \max(T(i-1), T(n-i))$$

**Behauptung**  $\text{Select} \in O(n)$ , also  $T(n) = c \cdot n$

**Beweis** Induktion

$$\begin{aligned} T(n) &= n + \frac{1}{n} \sum_{i=1}^n \max(c(i-1), c(n-i)) \\ &= n + \frac{1}{n} \cdot c \sum_{i=1}^n \max((i-1), (n-i)) \\ &= n + \frac{1}{n} \cdot c \cdot 2 \left( \sum_{i=1}^{n-1} i - \sum_{i=1}^{\frac{n}{c}-1} i \right) \\ &= n + \frac{1}{n} \cdot c \cdot Z \left( \frac{(n-1)n}{Z} - \frac{(\frac{n}{2}-1)\frac{1}{2}}{Z} \right) \\ &= n + \frac{1}{n} c \left( n(n-1) - \frac{n}{2} \left( \frac{n}{2} - 1 \right) \right) = n + \frac{1}{n} \cdot c \left( n^2 - n - \frac{n^2}{4} + \frac{n}{2} \right) \\ &= n + \frac{1}{n} c \left( \frac{3}{4} n^2 - \frac{1}{2} n \right) = n + c \left( \frac{3}{4} n - \frac{1}{2} \right) \\ &\Rightarrow cn = n + c \left( \frac{3}{4} n - \frac{1}{2} \right) = n + \frac{3}{4} cn - \frac{1}{2} c \\ &\Rightarrow cn \geq n + \frac{3}{4} cn \Leftrightarrow c \geq 4 \\ &\quad \text{q.e.d} \end{aligned}$$

## 12. Verallgemeinerung von Akra-Bazzi

$$T_n = \left[ \sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \right] + g(n)$$

**Beispiel**

$$T_n = 1 \cdot T\left(\frac{n}{3}\right) + 1 \cdot T\left(\frac{2n}{3}\right) + n$$

$$T_n = \theta \left( n^\alpha \left( 1 + \int_1^n \frac{g(x)}{x^{1+\alpha}} dx \right) \right)$$

**Klassisch**  $\alpha = \log_b(a)$ ,  $\frac{a}{b^\alpha} = 1$

**Jetzt** Bestimme  $\alpha$  so, dass gilt:

$$\sum_{i=1}^k \frac{a_i}{b_i^\alpha} = 1$$

$$a_1 = a_2 = 1, \quad b_1 = 3, \quad b_2 = \frac{3}{2}, \quad g(n) = n$$

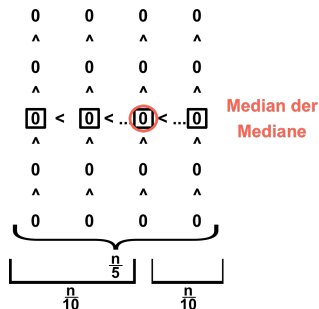
$$\frac{1^\alpha}{3} + \frac{2^\alpha}{3} \stackrel{!}{=} 1 \Rightarrow \alpha = 1$$

$$T(n) = \Theta \left( n \left( 1 + \int_1^n \frac{x}{x^{1+1}} dx \right) \right) = \Theta(n \ln(n))$$

## 13. Median der Mediane

### 13.1. Ansatz

Gruppierung in 5er Päckchen



**Wortlaut** Teile die  $n$  Elemente in 5-er Gruppen. Bestimme innerhalb jeder Gruppe den Median. Bestimme nun den Median der Mediane. Wähle diesen Median als Pivotelement.

$$\exists \frac{3n}{10} \text{ Elemente} \leq p \leq \exists \frac{3n}{10} \text{ Elemente} (\pm 1 \text{ wegen } p)$$

Abbildung 13.1.: Median der Mediane

### 13.2. Deterministische Variante für k-Select

Wähle zu Beginn den Median der Mediane als Pivot Element. Unterteile nun die Folge anhand von  $p$  in zwei Teilfolgen und verfähre von nun an analog zur randomisierten Variante von k-Select.

### 13.3. Laufzeitanalyse für den worst-case

$$T(n) = T\left(\frac{n}{5}\right) + n + T\left(\frac{7n}{10}\right)$$

$$A_1 = \frac{n}{5}, \quad A_2 = n, \quad A_3 = \frac{7n}{10}$$

$A_1$  = Laufzeit zur rekursiven Bestimmung des Medians der Mediane mittels k-Select

$A_2$  = Laufzeit zur Aufteilung in Teilfolgen

$A_3$  = Laufzeit für den Aufruf von k-Select für die größere Teilfolge in jedem Durchlauf, die aber sicher  $\leq n - \frac{3n}{10} = \frac{7n}{10}$  hat.

Wende die verallgemeinerte Form von Akra-Bazzi an:

$$g(n) = n, \quad a_1 = a_2 = 1, \quad b_1 = 5, \quad b_2 = \frac{10}{7}$$

Bestimme

$$\alpha = \left(\frac{1}{5}\right)^\alpha + \left(\frac{7}{10}\right)^\alpha = 1$$

$$\Leftrightarrow \left(\frac{2}{10}\right)^\alpha + \left(\frac{7}{10}\right)^\alpha = 1$$

$$\Rightarrow 0 < \alpha < 1$$

$$n^\alpha \left(1 + \int_1^n \frac{x}{x^{1+\alpha}} dx\right) = n^\alpha \left(1 + \int_1^n x^{-\alpha} dx\right) = n^\alpha \left(1 + \left[\frac{1}{1-\alpha} x^{-\alpha+1}\right]_1^n\right) = n^\alpha \left(1 + \frac{1}{1-\alpha} (n^{-\alpha+1} - 1)\right)$$

# 14. Untere Schranke für vergleichsbasierte Sortiervverfahren

## 14.1. Entscheidungsproblem: (Bubblesort)



Abbildung 14.1.: Entscheidungsbaum am Beispiel Bubblesort

Ein Entscheidungsbaum für einen vergleichsbasierten Sortieralgorithmus besteht aus inneren Knoten, die mit der Vergleichsoperation  $a_i < a_j$  beschriftet sind, wobei sich die Indizes  $i, j$  auf die Position der Elemente in der Eingabefolge beziehen.

Die Blätter des Entscheidungsbaums sind mit den Permutationen beschriftet, die sich nach korrekter Sortierung ergeben.

Jeder korrekte Sortieralgorithmus muss zu einem Entscheidungsbaum mit mindestens  $n!$  Blättern korrespondieren.

**maximale Baumtiefe**  $\hat{=}$  maximale Anzahl durchgeführter Vergleichsoperationen

**mittlere Baumtiefe**  $\hat{=}$  average-case Laufzeit

## 14. Untere Schranke für vergleichsbasierte Sortierverfahren

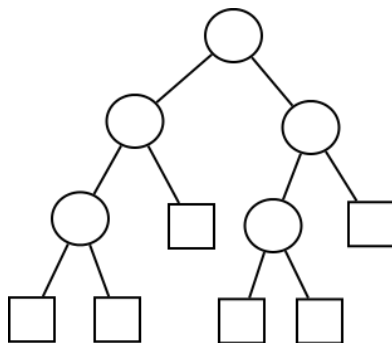
### 14.2. Worst-case Laufzeit

Worst-case Laufzeit eines vergleichsbasierten Sortieralgorithmus

$\hat{=}$  maximale Tiefe des zugehörigen Entscheidungsbaums

$\hat{=}$  mittlere Tiefe der Blätter im zugehörigen Entscheidungsbaums

Sei  $T_{max}$  die maximale Baumtiefe in einem binären Baum. Betrachte nun zunächst den vollständigen binären Baum mit  $\# \text{Blätter} \leq 2$ .



**Untere Schranke**  $t_{max} \geq \log_2(n!) = \Omega(n \log n) = \log_2(n!) \leq t_{mean}$

Abbildung 14.2.: Binärerbaum

#### Herleitung

$$\ln(n!) = \ln(n(n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1) = \ln(n) + \ln(n-1) + \dots + 1$$

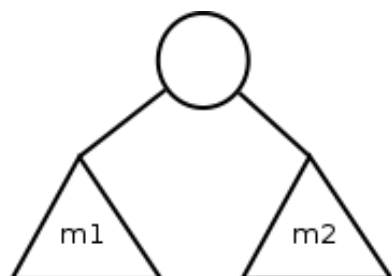
$$= \sum_{i=1}^n \ln(i) \geq \int_1^n \ln(x) dx = [x \ln(x) - x]_1^n = n \ln(n) - n + 1$$

$$\Rightarrow n! \geq e^{n \ln(n) - n + 1} = e \cdot e^{-n} \cdot \left(e^{\ln(n)}\right)^n = e \cdot e^{-n} \cdot n^n = e \left(\frac{n}{e}\right)^n$$

Stirling  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

#### 14.2.1. Lemma: Mittlere Tiefe der Blätter in einem Entscheidungsbaum $> \log_2(n)n$

**Beweis** Induktion nach m (Blattanzahl)



**Untere Schranke**  $m_1, m_2 \hat{=}$  Blattanzahl im linken bzw. rechten Teilbaum der Wurzel

Abbildung 14.3.: Induktions-Ansatz

**Induktions Anfang:**  $m = 1 \quad t_{mean} = \log_2(1) = 0$



**Induktions Behauptung:**  $t_{mean} \geq \log_2(m)$

**Induktions Schritt:** Sei  $m_1 < m, m_2 < m$  (1) und  $m_1 + m_2 = m$  (2)

b  $\hat{=}$  Blatt im Entscheidungsbaum  $T_b$

l  $\hat{=}$  Blatt im linken Teilbaum  $T_l$

r  $\hat{=}$  Blatt im rechten Teilbaum  $T_r$

$$t_{mean}^{links} \geq \log_2(m_1) \quad \text{und} \quad t_{mean}^{rechts} \geq \log_2(m_2)$$

$$\frac{1}{m} \sum_l \cdot t_l = t_{mean}^{links} \geq \log_2(m_1)$$

Verfahre analog für rechts.

$$\sum_b T_b = \sum_l (T_l + 1) + \sum_r (T_r + 1) \geq m_1 + m_2 + m_1 \log_2(m_1) + m_2 \log_2(m_2)$$

Unter der Annahme, dass das Minimum bei  $\frac{m}{2}$  liegt:

$$m_1 \log_2(m_1) + m_2 \log_2(m_2) \geq \frac{m}{2} \log_2\left(\frac{m}{2}\right) \cdot 2 = m \log_2\left(\frac{m}{2}\right) \quad \text{mit (2)}$$

Es folgt somit:

$$t_{mean} = \frac{1}{m} \sum_b T_b \geq \frac{1}{m} \left( m + m \log_2\left(\frac{m}{2}\right) \right) = 1 + \log_2\left(\frac{m}{2}\right) = 1 + \log_2(m) - 1 = \log_2(m)$$

*q.e.d*

## 15. Radix-Sort

### 15.1. Beispiel:

10	1	0	1	0	1	00	001
01	0	1	0	0	1	01	010
00	1	1	1	0	0	01	011
11	1	1	0	1	0	10	100
10	0	0	0	1	1	10	101
01	1	1	1	1	1	11	110
11	0	0	1	1	0	11	111

**Wichtig** Beginne die Sortierung mit dem niedrigsten Bit

### 15.2. Pseudo-Code

```
1 void radixsort(int[] a) { // positives Element
2     int n = a.length;
3     int[] b0 = new int[n];
4     int[] b1 = new int[n];
5     int n0, n1;

7     for (int i=0; i<32; i++) {
8         n0 = n1 = 0;
9         for (int j=0; j<n; j++) {
10            if (a[j] & (1<<i)) { // i-tes Bit von a[j]
11                b1[n1] = a[j];
12                n1 = n1+1;
13            } else {
14                b0[n0] = a[j];
15                n0 = n0+1;
16            }
17        }

19        for (int j=0; j<n0; j++)
20            a[j] = b0[j];
21        for (int j=0; j<n1; j++)
22            a[n0+j] = b1[j];
23    }
24 }
```

## 16. Binäre Suchbäume

**Zahlen** 12, 8, 3, 16, 24, 17, 10, 21, 14, 9

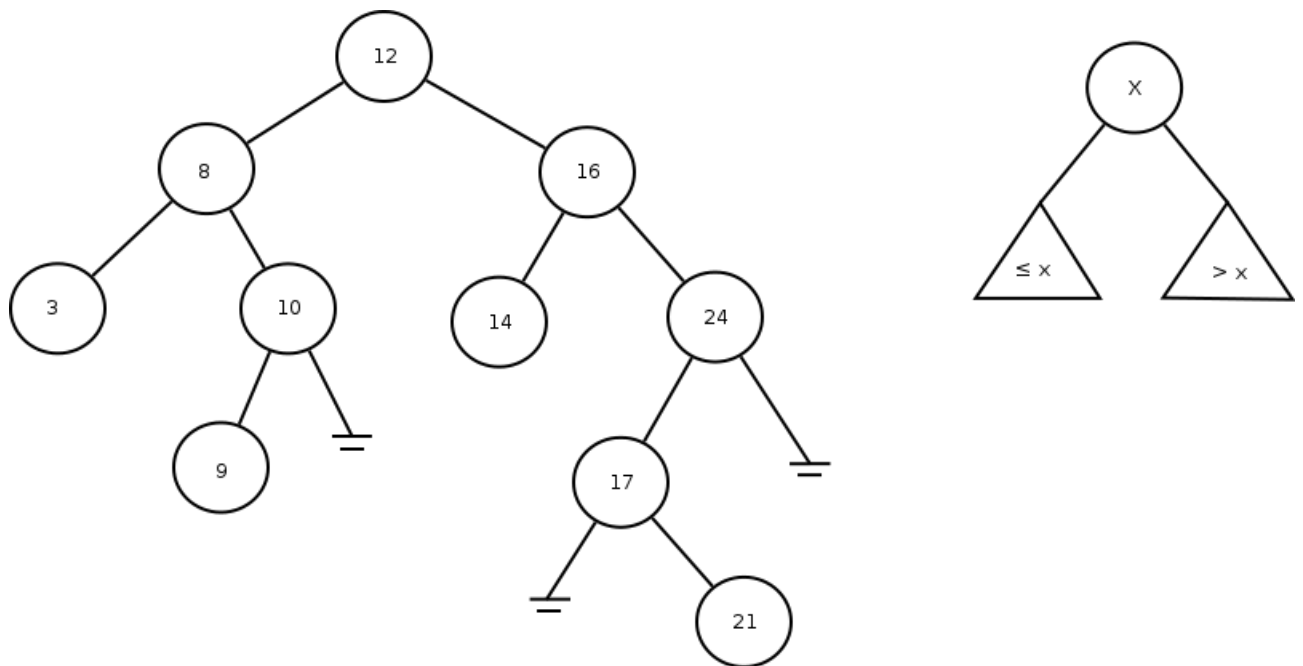


Abbildung 16.1.: Knotenorientierte Speicherung

## 16. Binäre Suchbäume



Abbildung 16.2.: Binärer Suchbaum

### 16.1. Pseudo-Code

```
2  class Node {
3      int key, info;           // info ist optional
4      Node left, right, parent; // parent ist optional
5  }

7  int height(Node node) {
8      if (node == NULL) return 0;
9      return height;
10 }

12 Node insert(Node node, int x) {
13     if (node == NULL)
14         return new Node(x, NULL, NULL);

16     if (node.key > x)
17         node.left = insert(node.left, x);
18     else
19         node.right = insert(node.right, x);
20     return node;
21 }

23 void inorder(Node node) {
24     if (node == NULL) return;
25     inorder(node.left) // linke Hälfte
26     print(node)
27     inorder(node.right) // rechte Hälfte
28 }
```

# 17. AVL-Bäume von Adelson-Velsky and Landis

## 17.1. Allgemein

**Ziel** Binärer Suchbaum mit garantierter Such-, Einfüge- und Löschzeit  $O(\log n)$

**Idee** Definiere eine Balancebedingung, die dafür sorgt, dass die Baumstruktur möglichst nahe an der Idealstruktur eines vollständigen binären Baumes liegt.

Aber gleichzeitig soll es möglich sein "schnell" Strukturänderungen beim Einfügen und Löschen vorzunehmen.

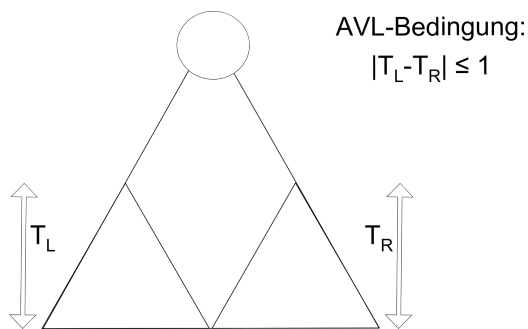
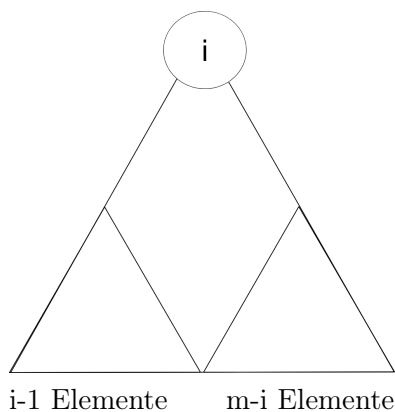


Abbildung 17.1.: AVL-Baum

## 17.2. Laufzeitanalyse

**Ziel** Analyse der erwarteten maximalen Tiefe randomisierter binärer Suchbäume

Sei der Schlüssel der Wurzel das  $i$ -kleinste Element



$T_n \hat{=}$  maximale Tiefe eines randomisierten Suchbaums mit  $\{1, \dots, n\}$  Elementen

Abbildung 17.2.

Für den Fall, dass  $i$  als Wurzelknoten gewählt wird gilt:

$$\begin{aligned} T_n &= \max\{T_{i-1}, T_{n-i}\} + 1 \\ X_n &= 2^{T_n} \text{ exponentielle Tiefe} \\ 2^{T_n} &= 2^{1+\max\{T_{i-1}, T_{n-i}\}} = 2 \cdot 2^{\max\{T_{i-1}, T_{n-i}\}} = 2 \cdot \max\{2^{T_{i-1}}, 2^{T_{n-i}}\} \\ &\Rightarrow X_n = 2 \cdot \max\{X_{i-1}, X_{n-i}\} \end{aligned}$$

Mit der Abschätzung:  $\max\{2^{T_1}, 2^{T_2}\} \leq 2^{T_1} + 2^{T_2}$  folgt:

$$\begin{aligned} E(X_n) &= E\left(\sum_{i=1}^n \frac{1}{n} \cdot 2 \cdot \max\{X_{i-1}, X_{n-i}\}\right) \\ &= \frac{2}{n} \sum_{i=1}^n E(\max\{X_{i-1}, X_{n-i}\}) \leq \frac{2}{n} \sum_{i=1}^n E(X_{i-1} + X_{n-i}) \\ &= \frac{2}{n} \sum_{i=1}^n [E(X_{i-1}) + E(X_{n-i})] \leq \frac{4}{n} \sum_{i=0}^{n-1} E(X_i) \\ n \cdot E(X_n) &= 4 \cdot \sum_{i=0}^{n-1} E(X_i) \quad (1) \\ (n-1) \cdot E(X_{n-1}) &= 4 \cdot \sum_{i=0}^{n-2} E(X_i) \quad (2) \\ nE(X_n) - (n-1)E(X_{n-1}) &= 4E(X_n) \quad (1) - (2) \\ \Leftrightarrow nE(X_n) &= (n+3)E(X_{n-1}) \\ E(X_n) &= \frac{n+3}{n} E(X_{n-1}) = \frac{n+3}{n} \cdot \frac{n+2}{n-1} E(X_{n-2}) = \prod_{i=0}^{n-1} \frac{n+3-i}{n-i} \\ &= \frac{n+3}{n} \cdot \frac{n+2}{n-1} \cdot \frac{n+1}{n-2} \cdot \frac{n}{n-3} \cdot \dots \cdot \frac{6}{3} \cdot \frac{8}{2} \cdot \frac{4}{1} \end{aligned}$$

Mit der "Jensenschen Ungleichung" folgt:

$$\begin{aligned} \sum_i Pr(T = t_i) \cdot f(t_i) &\geq f\left(\sum_i Pr(T = t_i) \cdot t_i\right) = \frac{(n+3)(n+2)(n+1)}{3!} \cdot c \Rightarrow E(X_n) \in O(n^3) \\ X_n &= 2^{T_n}, E(X_n) = E(2^{T_n}) \\ E(f(T)) &\geq f(E(T)) \Leftrightarrow f \text{ konvex} \\ c \cdot n^3 &\geq 2^{E(T_n)}, E(T_n) \leq \log_2(c \cdot n^3) \in O(\log n) \end{aligned}$$

## 17.3. Baumtiefe



Abbildung 17.3.

**Ziel:** Zeige, dass die maximale Tiefe eines AVL-Baums mit  $n$  Knoten ( $\hat{=}$   $n$  gespeicherten Schlüsseln)  $O(\log(n))$  beträgt.

### 17.3.1. AVL-Eigenschaft:

$|h(T_L) - h(T_R)| \leq 1$  muss für jeden Knoten des Baums gelten.  $\Rightarrow$  Suchzeit  $O(\log(n))$  im worst-case.

$n(h)$  = minimale Anzahl von Knoten in AVL-Baum der Tiefe  $h$

$$n(h) \geq 1 + n(h-2) + n(h-1) \text{ mit } n(0) = 0 \text{ und } n(1) = 1$$



Abbildung 17.4.

$$n \geq f(h)^I = \frac{1}{\sqrt{5}} \cdot (\phi^h - \phi^{-h}) \text{ mit}$$

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1,61 \dots$$

$$\Rightarrow n \geq c \cdot \phi^h$$

$$\Leftrightarrow h \leq \log\left(\frac{n}{c}\right)$$

$$\Rightarrow h \in O(\log n)$$

q.e.d.

<sup>I</sup> $f(h)$  meint hierbei die  $h$ -te Fibonacci-Zahl

## 17.4. Rotationen



$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_3)$

$balance(Y) = height(Y.left) - height(Y.right)$

$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T'_2) < Key(Z) < Keys(T_3)$



Abbildung 17.5.: Rotation



## 17.5. Pseudo-Code

```

2  class Node {
3      int key;
4      Node left, right;
5      int height;
6  }

8  int height(Node node) {
9      if (node == null) return 0;
10     return height;
11 }

13 Node rotateRight(Node y) {
14     Node x = y.left;
15     Node T2 = x.right;
16     y.left = T2;
17     x.right = y;
18     y.height = 1+max(height(y.left), height(y.right));
19     x.height = 1+max(height(x.left), height(x.right));
20     return x;
21 }

23 Node rotateLeft(Node y) {
24     //analog
25 }

27 Node insert(Node node, int key) {
28     if (node == null) return new Node(key);
29     if (key < node.key)
30         node.left = insert(node.left, key);
31     else
32         node.right = insert(node.right, key);

34     if (balance(node)>1 && key < node.left.key)
35         return rotateRight(node);
36     if (balance(node)<-1 && key > node.right.key)
37         return rotateLeft(node);
38     if (balance(node)>1 && key > node.left.key) {
39         node.left = rotateLeft(node.left);
40         return rotateRight(node);
41     }
42     if (balance(node)<-1 && key < node.right.key) {
43         node.right = rotateRight(node.right);
44         return rotateLeft(node);
45     }
46     return node;
47 }

```

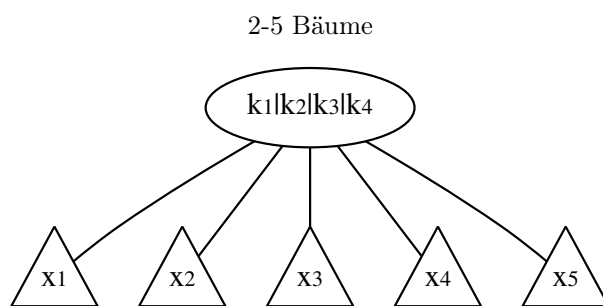
**Anmerkung:** Die Laufzeit des Einfügens bleibt in  $O(\text{Baumtiefe}) = O(\log n)$ . Nur einer der vier Fälle ist notwendig, um die Balance herzustellen.

## 18. (a,b)-Suchbäume

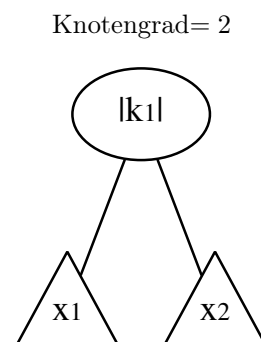
Blattorientierte Speicherung der Elemente

Innere Knoten haben mindestens  $a$  und höchstens  $b$  Kinder und tragen entsprechende Schlüsselwerte, um die Suche zu leiten.

**Beispiel:**



(a)  $x_1 \leq k_1 < x_2 \leq k_2 < x_3 \leq k_3 < x_4 \leq k_4 < x_5$



(b)  $x_1 \leq k_1 < x_2$

$$h \hat{=} \text{Tiefe} \Rightarrow a^h \leq n \leq b^h \Rightarrow \log_b n \leq h \leq \log_a n$$

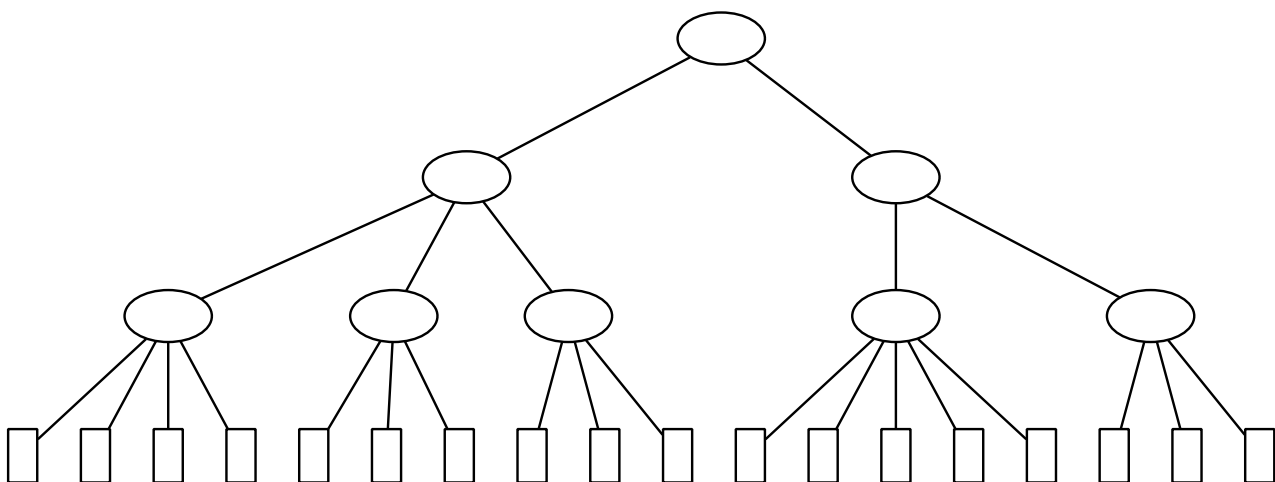
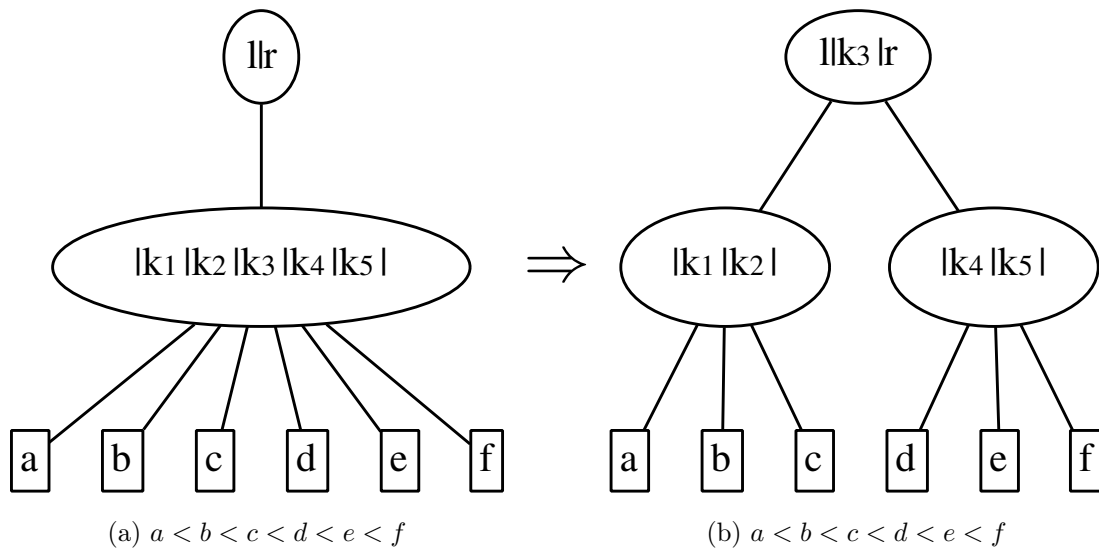
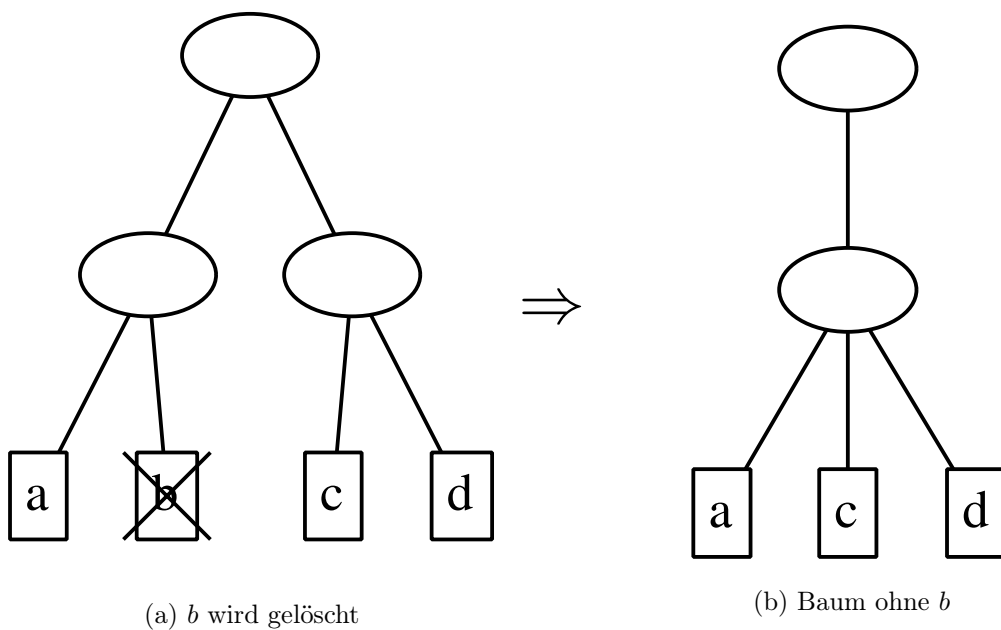


Abbildung 18.2.: Einfügen und Löschen in einem 2-5 Baum

## 18.1. Aufspaltung bei Einfügen



## 18.2. Verschmelzen von Knoten beim Löschen



Aufspalte- und Verschmelze-Operationen können sich von der Blattebene bis zur Wurzel kaskadenartig fortpflanzen. Sie bleiben aber auf den Suchpfad begrenzt.

⇒ Umbaukosten sind beschränkt durch die Baumtiefe =  $O(\log n)$

## 19. Amortisierte Analyse

	000	
	001	Kosten(1) = 1
	010	= 2
	011	= 1
<b>Beispiel: Binärzähler</b>	100	= 3
	101	= 1
	110	= 2
	111	= 1
		<u>11</u>

Kosten der Inkrement-Operation  $\hat{=}$  Zahl der Bit-Flips

Naive Analyse  $2^k = n$

$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \dots + k \cdot \frac{n}{2^k} = \frac{n}{2} \sum_{i=1}^k i \left(\frac{1}{2}\right)^{i-1} = 2^{k+1} - k - 2 = 2n - k - 2$$

Von 0 bis  $n$  im Binärsystem zu zählen kostet  $\leq 2n$  Bit-Flips

**Sprechweise:** amortisierte Kosten einer Inkrement-Operation sind 2  
 Folge von  $n$ -Ops kostet  $2n$

### 19.1. Bankkonto-Methode

$$\text{Konto}(i+1) = \text{Konto}(i) - \text{Kosten}(i) + \text{Einzahlung}(i)$$

$$\sum_{i=1}^n \text{Kosten}(i) = \text{tatsächliche Gesamtkosten} = \sum_{i=1}^n (\text{Einzahlung}(i) + \text{Konto}(i) - \text{Konto}(i+1))$$

$$= \sum_{i=1}^n \text{Einzahlung}(i) + \text{Konto}(1) - \text{Konto}(n+1)$$

000	
001€	Kosten(1) = 1
01€0	= 2
01€1€	= 1
1€00	= 3
1€01€	= 1
1€1€0	= 2
1€1€1€	= 1
	<u>11</u>

### 19.1.1. Kontoführungsschema: für Binärzähler

1€ pro 1 in der Binärdarstellung

Jeder Übergang  $1€ \rightarrow 0$  kann dann mit dem entsprechenden Euro Betrag auf dieser 1 bezahlt werden.

Es gibt pro Inkrement Operation nur einen  $0 \rightarrow 1$  Übergang

2€ Einzahlung für jede Inc-Operation reichen aus um:

1. diesen  $0 \rightarrow 1$  Übergang zu bezahlen
2. die neu entstandene 1€ mit einem Euro zu besparen.

$$GK = 2(2^k - 1) + 0^I - k^{\text{II}} = 2n - k - 2$$

### 19.1.2. Kontoführungsschema für einen 2-5 Baum

**Satz:** Ausgehend von einem leeren 2-5-Baum betrachten wir die Rebalancierungskosten  $C$  (Split- und Fusionsoperationen) für eine Folge von  $m$  Einfüge- oder Löschooperationen. Dann gilt:  $C \in O(m)$

d.h. Amortisierte Kosten der Split- und Fusionsoperationen sind konstant.

! Dies bezieht sich nicht auf die Suchkosten, die in  $O(\log n)$  liegen.

**Beweisidee:**

	1	2	3	4	5	6
<b>Kontoführung:</b>	2€	1€	0€	0€	1€	2€

regelmäßige Einzahlung: 1€

Durch eine Einfüge- oder Löschooperation steigt oder fällt der Knotengrad des direkt betroffenen Knotens um höchstens 1.  $\Rightarrow$  1€ Einzahlung reicht zur Aufrechterhaltung dieses Sparplanes.

Jetzt Beseitigung der temporären 1- und 6-Knoten:

Ein 6-Knoten nutzt 1€ um seinen Split zu bezahlen. Die beiden neu entstehenden 3-Knoten benötigen kein Kapital. Der Vaterknoten des gesplitteten 6-Knotens benötigt ggf. den zweiten verfügbaren €.

Analoge Betrachtung für Fusion eines temp. 1-Knotens.

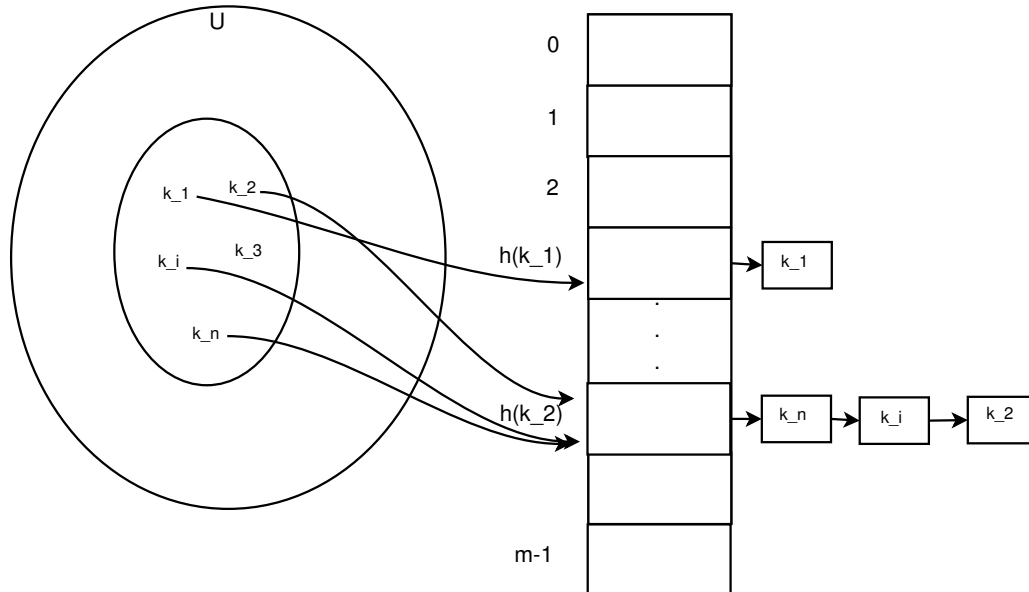
---

<sup>I</sup>Zählerstand(000)

<sup>II</sup>Zählerstand( $\overbrace{111 \dots 1}^k$ )

## 20. Hashing

Abbildung 20.1.: Universum und Hashtabelle der Größe  $m$



$U \subseteq \mathbb{N}$  z.B. 64-Bit-Integer

$n$  = Zahl der zu verwaltenden Schlüssel

$$|U| \gg n$$

Hashfunktion  $h$ :

$$h : U \rightarrow [0, \dots, m-1]$$

$$\text{z.B. } k \mapsto k \bmod m$$

Einfache Annahme: (einfaches uniformes Hashing)

$$\forall k_i, k_j \in U : \Pr(h(k_i) = h(k_j)) = \frac{1}{m}$$

### 20.1. Laufzeiten der Operationen

#### 20.1.1. Analyse der Laufzeit zum Einfügen eines neuen Elementes $k$

- $h(k)$  berechnen  $\rightarrow O(1)$
- Einfügen am Listenanfang in Fach  $h(k)$ .  $\rightarrow O(1)$

### 20.1.2. Analyse der Suchzeit für einen Schlüssel $k$

- $h(k) \rightarrow O(1)$
- Listenlänge zum Fach  $h(k)$  sei  $n_{h(k)}$  also beim Durchlauf der kompletten Liste  $\rightarrow O(n_{h(k)})$

$$E(n_{h(k)}) = \frac{n}{m} = \alpha^I$$

$$\text{Suchzeit(Einfügen)} \in O(1 + \alpha)$$

### 20.1.3. Laufzeit beim Löschen von Schlüssel $k$

- $h(k) \rightarrow O(1)$
- Durchlaufen der Liste  $\rightarrow O(n_{h(k)})$
- Löschen durch „Pointer-Umbiegen“  $\rightarrow O(1)$

## 20.2. Universelles Hashing

**Idee** Arbeite nicht mit einer festen Hashfunktion sondern wähle am Anfang eine zufällige Hashfunktion aus einer Klasse von Hashfunktionen aus.

**z.B.**

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

$p$  sei eine hinreichend große Primzahl  $0 < a < p, 0 \leq b < p$

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) | 0 < a < p, 0 \leq b < p\}$$

$$|\mathcal{H}_{p,m}| = p(p-1)$$

**Definition**  $\mathcal{H}$  heißt universell  $\Leftrightarrow \forall k, l \in U : Pr(h(k) = h(l)) \leq \frac{1}{m}$

**Suchzeit**

$$\mathcal{X}_{k,l} = \begin{cases} 1 & \text{für } h(k) = h(l) \\ 0 & \text{sonst} \end{cases}$$

$$E(n_{h(k)}) = E\left(\sum_{l \in T, l \neq k}\right) = \sum_{l \in T, l \neq k} E(\mathcal{X}_{k,l}) = \sum_{l \in T, l \neq k} Pr(h(k) = h(l)) = \sum_{l \in T, l \neq k} \frac{1}{m} = \frac{n-1}{m} = \alpha$$

Könnte ein boshafter Mitspieler  $n$  Schlüssel bei gegebener fester Hashfunktion wählen, so würde er solche wählen, die auf den gleichen Slot unter gegebener Hashfunktion abgebildet werden.  $\rightsquigarrow$  Durchschnittliche Ablaufzeit von  $O(n)$

**Idee** zufällige Wahl der Hashfunktion aus einer Familie von Funktionen derart, dass die Wahl unabhängig von den zu speichernden Schlüssel ist (universelles Hashing).

---

<sup>1</sup>Belegungsfaktor

## 20. Hashing

### 20.2.1. Definition

Sei  $\mathcal{H}$  eine endliche Menge von Hashfunktionen, welche ein gegebenes Universum  $U$  von Schlüsseln auf  $\{0, \dots, m-1\}$  abbildet. Sie heißt universell, wenn für jedes Paar von Schlüsseln  $k, l \in U$   $l \neq k$  die Anzahl der Hashfunktionen  $h \in \mathcal{H}$  mit  $h(l) = h(k)$  höchstens  $\frac{|\mathcal{H}|}{m}$ . Anders: Für ein zufälliges  $h \in \mathcal{H}$  beträgt die Wahrscheinlichkeit, dass zwei unterschiedliche Schlüssel  $k, l$  kollidieren nicht mehr als  $\frac{1}{m}$  ist.

### Beispiel

$p$  Primzahl, so groß, dass alle möglichen Schlüssel  $k \in U$  im  $0, \dots, p-1$  liegen.  $\mathbb{Z}/p\mathbb{Z}$  bezeichnet den Restklassenring  $\text{mod } p$  (weil  $p$  prim, ist  $\mathbb{Z}/p\mathbb{Z}$  ein Körper).  $\mathbb{Z}/p\mathbb{Z}^*$  ist die Einheitengruppe.

**Annahme:** Die Menge der Schlüssel im Universum  $U$  ist größer als die Anzahl der Slots in der Hashtabelle. Für  $a \in \mathbb{Z}/p\mathbb{Z}^*$  und  $b \in \mathbb{Z}/p\mathbb{Z}$  betrachte:

$$h_{a,b}(k) := (a \cdot k + b \text{ mod } p) \text{ mod } m \quad (*)$$

Damit ergibt sich die Familie

$$\mathbb{Z}/p\mathbb{Z}^* = \{1, \dots, p-1\} \quad \mathbb{Z}/p\mathbb{Z} = \{0, \dots, p-1\} \quad \mathcal{H}_{p,m} = \{h_{a,b} | a \in \mathbb{Z}/p\mathbb{Z}^*, b \in \mathbb{Z}/p\mathbb{Z}^{(*)} \mid |\mathcal{H}| = p(p-1)\}$$

**Satz** Die in  $(*)$  eingeführte Klasse von Hashfunktionen ist universell.

**Beweis** Seien  $k, l$  Schlüssel auf  $\mathbb{Z}/p\mathbb{Z}$  mit  $k \neq l$

Für  $h_{a,b} \in \mathcal{H}_{p,m}$  betrachten wir

$$r = (a \cdot k + b) \text{ mod } p$$

$$s = (a \cdot l + b) \text{ mod } p$$

Es ist  $r \neq s$

Dazu:

$$r - s = a \cdot (k - l) \text{ mod } p \quad (*2)$$

**Angenommen  $r - s = 0$**

$$0 = a \cdot (k - l) \text{ mod } p, \text{ aber } a \in \mathbb{Z}/p\mathbb{Z}^* \Rightarrow a \neq 0 \text{ und } k \neq l \Rightarrow k - l \neq 0$$

Da  $p$  prim ist  $\mathbb{Z}/p\mathbb{Z}$  ein Körper  $\Rightarrow$  kein Nullteiler  $\Rightarrow a \cdot (k - l) \neq 0 \Rightarrow r \neq s$

Daher bilden  $h_{a,b} \in \mathcal{H}_{p,m}$  unterschiedliche Schlüssel  $k, l$  auf unterschiedliche Elemente ab. („Auf dem level  $\text{mod } p$ “ gibt es keine Kollisionen).

Aus  $(*2)$  folgt:

$$(r - s)(k - l)^{-1} = a \text{ mod } p$$

$$r - a \cdot k = b \text{ mod } p \text{ Bijektion zwischen } (k, l) \text{ und } (a, b)$$

Daher ist die Wahrscheinlichkeit, dass zwei Schlüssel  $h \neq l$  kollidieren, gerade die Wahrscheinlichkeit, dass  $r \equiv s \text{ mod } m$ , falls  $r \neq s$  zufällig gewählt (aus  $\mathbb{Z}/p\mathbb{Z}$ ).

Für gegebenes  $r$  gibt es unter den übrigen  $p-1$  Werten für  $s$  höchstens  $\lceil \frac{p-1}{m} \rceil \leq \lceil \frac{p}{m} \rceil - 1$  Möglichkeiten, sodass  $s \neq r \text{ mod } p$  aber  $r = s \text{ mod } m$



**Abschätzung nach oben**

$$\left\lceil \frac{p}{m} \right\rceil - 1 \leq \frac{(p+m-1)}{m} - 1 = \frac{p-1}{m} \text{ Kollisionsmöglichkeiten}$$

Die Wahrscheinlichkeit, dass  $r$  und  $s$  kollidieren  $\bmod m$  Kollisionsmöglichkeiten / Gesamtzahl der Werte

$$= \frac{p-1}{m} \cdot \frac{1}{p-1} = \frac{1}{m}$$

$\Rightarrow$  Für ein Paar von Schlüsseln  $k, l \in \mathbb{Z}/p\mathbb{Z}$  mit  $k \neq l$

$$P[h_{a,b}(k) = h_{a,b}(l)] \leq \frac{1}{m} \Rightarrow \mathcal{H}_{p,m} \text{ universell!}$$

**20.3. Perfektes Hashing**

**Wichtig** Menge der Schlüssel ist im Vorhinein bekannt und ändert sich nicht mehr.

**Beispiele** reserved words bei Programmiersprachen, Dateinamen auf einer CD

**20.3.1. Definition**

Eine Hashmethode heißt perfektes Hashing, falls  $O(1)$  Speicherzugriffe benötigt werden, um die Suche nach einem Element durchzuführen.

**Idee** Zweistufiges Hashing mit universellen Hashfunktionen.

1. Schritt  $n$  Schlüssel,  $m$  Slots durch Verwendung der Hashfunktion  $h$ , welche aus einer Familie universeller Hashfunktionen stammt.
2. Schritt Statt einer Linkedlist im Slot anzulegen, benutzen wir eine kleine zweite Hashtabelle  $S_j$  mit Hashfunktion  $h_j$

**Bild** Schlüssel  $k = \{10, 22, 37, 49, 52, 60, 72, 75\}$

Äußere Hashfunktion  $h(k) = ((a \cdot b) \bmod p) \bmod m$

$$a = 3, \quad b = 42, \quad p = 101, \quad m = 9$$

$$h(10) = \underbrace{(3 \cdot 10 + 42 \bmod 101)}_{=72} \bmod 9 = 0$$

Um zu garantieren, dass keine Kollision auf der zweiten Ebene auftreten, lassen wir die Größe von  $S_j$  gerade  $n_j^2$  sein ( $n_j \neq \#\text{Schlüssel} \rightarrow j\text{Slot}$ ).

Wir verwenden für die Hashfunktion der ersten Ebene eine Funktion aus  $\mathcal{H}_{p,m}$ . Schlüssel die im  $j$ -ten Slot werden in der sekundären Hashtabelle  $S_j$  der Größe  $m_j$  mittels  $h_j$  gehasht.  $h_j \in \mathcal{H}_{p,m}$

**Wir zeigen:** 2 Dinge:

1. Wie versichern wir, dass die zweite Hashfunktion keine Kollision hat.
2. Der erwartete Speicherbedarf ist  $O(n)$

**zu 1.**

## 20. Hashing

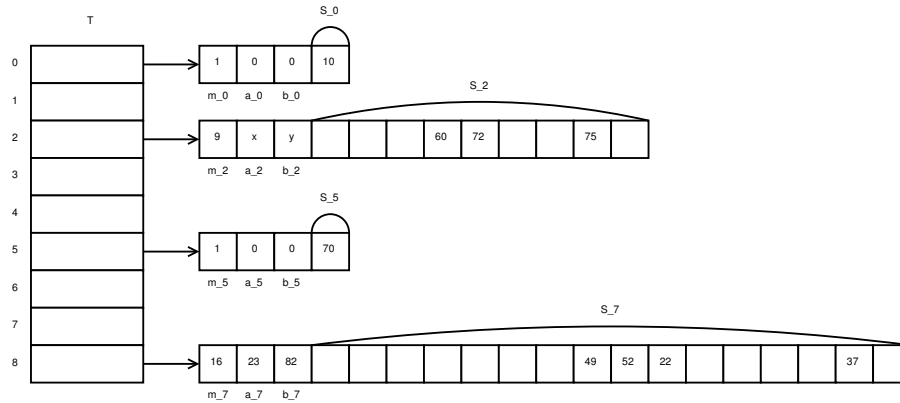


Abbildung 20.2.: Perfekte Hashtabelle

**Satz** Beim Speichern von  $n$  Schlüsseln in einer Hashtabelle der Größe  $m = n^2$  ist die Wahrscheinlichkeit, dass eine Kollision auftritt  $< \frac{1}{2}$

**Beweis:** Es gibt  $\binom{n}{2}$  mögliche Paare, die kollidieren können. Jedes kollidiert mit der Wahrscheinlichkeit  $\leq \frac{1}{m}$ , falls  $h \in \mathcal{H}$  zufällig gewählt wurde.

Sei  $X$  eine zufallsvariable(ZV),  $X$  zählt Kollisionen:

Für  $m = n^2$  ist die erwartete Zahl der Kollisionen:

$$E[X] = \binom{n}{2} \cdot \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n!}{2!(n-2)!n^2} = \frac{(n-1)}{2n} \leq \frac{1}{2}$$

Anwenden der Markow-Ungleichung ( $a=1$ ):

$$P[X \geq 1] \leq \frac{E[X]}{1} = \frac{1}{2} \Rightarrow \text{Wahrscheinlichkeit für irgendeine Kollision ist } < \frac{1}{2}$$

q.e.d

### 20.3.2. Nachteil

Für große  $n$  ist  $m = n^2$  nicht haltbar!

**zu 2.** Wenn die Größe der primären Hashtabelle  $m = n$  ist, dann ist der Platzverbrauch in  $O(n) \curvearrowright$  Betrachte Platzverbrauch der sekundären Hashtabellen.

**Satz** Angenommen wir wollen  $n$  Schlüssel in einer Hashtabelle der Größe  $m = n$  mit Hashfunktion  $h \in \mathcal{H}$ . Dann gilt:

$$E \left[ \sum_{j=0}^{m-1} n_j^2 \right] < 2n$$

**Beweis**

**Betrachte**

$$a^2 = a + 2 \cdot \binom{a}{n} = a + 2 \cdot \frac{a^2 - a}{2} \quad (*3)$$

**Betrachte**

$$E \left[ \sum_{j=0}^{m-1} n_j^2 \right] \stackrel{(*3)}{=} E \left[ \sum_{j=0}^{m-1} \left( n_j + 2 \binom{n_j}{2} \right) \right]$$

$$\stackrel{\text{lini. des EW}}{=} E \left[ \underbrace{\sum_{j=0}^{m-1} n_j}_{=n} \right] + 2 E \left[ \sum_{j=0}^{m-1} \binom{n_j}{2} \right] = n + 2 E \left[ \sum_{j=0}^{m-1} \binom{n_j}{2} \right] \# \text{ der Kollisionen}$$

Da unsere Hashfunktion universell ist, ist die erwartete Zahl dieser Paare:

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}, \text{ da } m = n$$

Somit

$$E \left[ \sum_{j=0}^{m-1} n_j^2 \right] \leq n + 2 \frac{n-1}{2} = 2n - 1 < 2n$$

**Korollar** Speichern wir  $n$  Schlüssel in einer Hashtabelle der Größe  $m = n$  mit einer zufälligen universellen Hashfunktion und setzen die Größe der Hashtabellen der zweiten Ebene auf  $m_j = n_j^2$  für  $j = 0, m = 1$ , so ist der Platzverbrauch des perfekten Hashings weniger als  $2n$ . Die Wahrscheinlichkeit, dass der Platzverbrauch der zweiten Hashtabellen  $\geq 4n$  ist, ist  $\leq \frac{1}{2}$  ohne Beweis. Bei  $n$  Elementen sollte die Hashtabelle  $m = n^2$  groß sein.

Für die universellen Hashfunktionen

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) = (a \cdot k + b) \bmod p \mid 0 < a < p, 0 \leq b < p\}$$

$\binom{n}{1}$  Schlüsselpaare  $(k, l)$  mit  $k \neq l$

$$E(\# \text{Kollisionen}) \leq \binom{n}{2} \cdot \frac{1}{m} \stackrel{\text{II}}{=} \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \leq \frac{1}{2}$$

**Idee** Zweistufiges Verfahren:

- primäre Hashfunktion für Tabelle der Größe  $m = n$

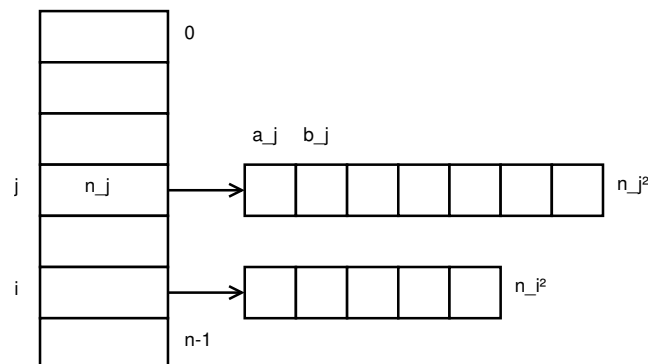


Abbildung 20.3.: Perfektes Hashing

<sup>II</sup>Universalität von  $\mathcal{H}$

Teil II.

# Graphen-Algorithmen

## 21. Einführung

$$G = (V, E) \quad V \text{ vertices, } E \text{ edges} \quad E \subseteq V \times V$$

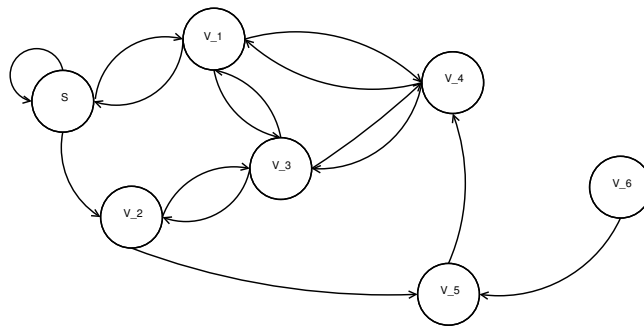


Abbildung 21.1.: Gerichteter Graph

Planare Graphen können ohne Überkreuzung der Kanten in die Ebene eingebettet werden.

### 21.1. Eulerische Polyederformel

$$|V| + |F| = |E| + 2$$

$$8 + 6 = 12 + 2$$

Es gilt:

$$2 \cdot |E| \geq 3 \cdot |F|$$

$$\# \text{gerichtete Kanten} = 2 \cdot |E| = \sum_{i=1}^{|F|} \# \text{Kanten}(f_i)^I \geq 3 \cdot |F|$$

$$|F| \leq \frac{2}{3}|E|, \quad |V| + |F| = |E| + 2 \leq |V| + \frac{2}{3}|E| \Rightarrow \frac{1}{3}|E| + 2 \leq |V|$$

$$\Rightarrow |E| \leq 3 \cdot |V| - 6$$

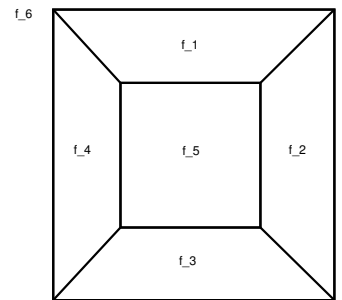


Abbildung 21.2.: Würfel

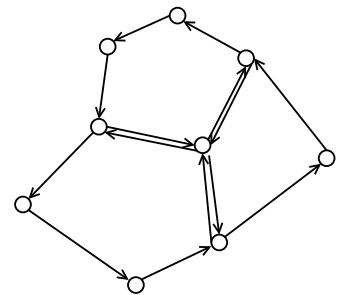


Abbildung 21.3.: Placeholder

<sup>1</sup>Jedes  $f_i$  hat mindestens 3 Kanten

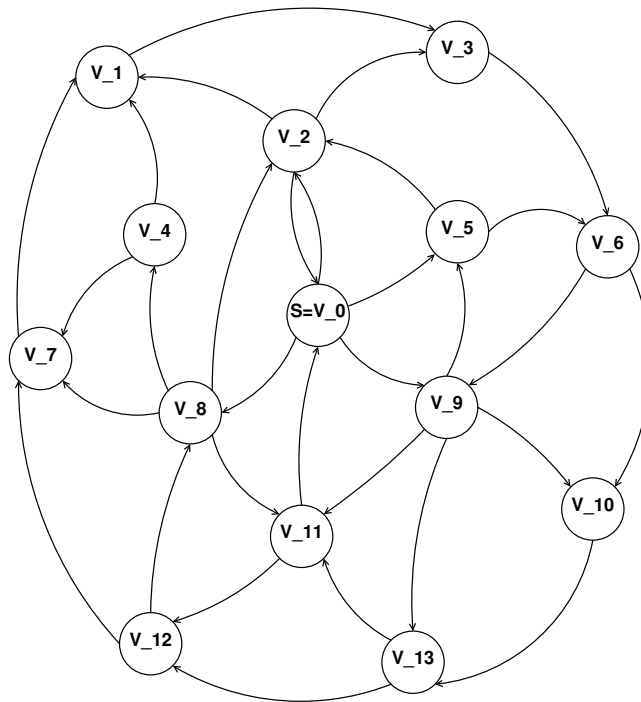


Abbildung 21.4.: Beispiel

## 21.2. Adjazenzmatrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	1	0	0	1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	1	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	1	1	0	0	0
7	0	1	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	1	0	1	0	0	1	1	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0	1	1	1	0	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0	1
11	1	0	0	0	0	0	0	0	0	0	0	1	1	0
12	0	0	0	0	0	0	0	1	1	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	1	1	1

$= A$

$$a \in B^{|V| \times |V|}$$

falls  $G$  ungerichtet  $\Rightarrow A = A^T$

## 21.3. Adjazenzlisten Repräsentation

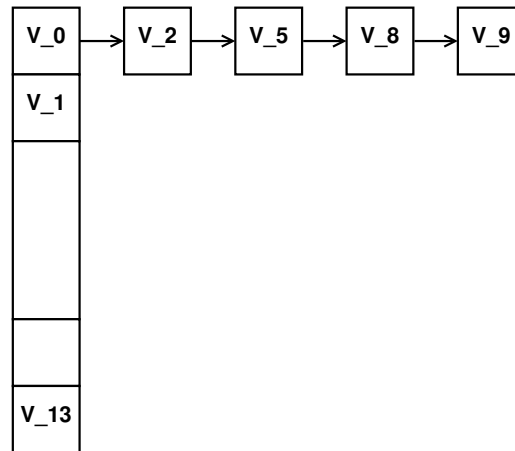


Abbildung 21.5.: Adjazenzliste

Platzbedarf

$$\mathcal{O}(|V| + |E|) = \mathcal{O}\left(|V| + \sum_{i=0}^{|V|-1} \text{outdeg}(v_i)\right)$$

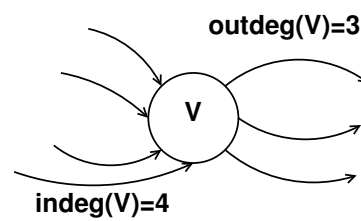


Abbildung 21.6.: indeg und outdeg

## 22. BFS (Breadth-First Search) Breitensuche

### 22.1. Pseudo-Code

```

1 forall (v in V \ {s}) {
2   col[v]=white; // Farbe weiß = unbekannt, grau = bekannt, schwarz = vollkommen bekannt
3   d[v] = infinity; // Distanz
4   pi[v] = NULL; // pi ist Vorgänger
5 }
6 col[s] = grey; // s ist Startknoten
7 d[s] = 0;
8 pi[s] = NULL;

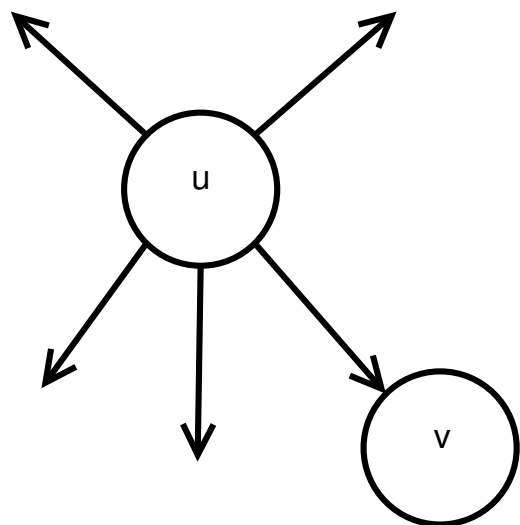
```

Queue	vs	Stack
Schlange		Stapel
empty()		"
push()		"
pop()		
FIFO		FILO
First-In-First-Out		First-In-First-Out

```

1 Queue Q;
2 Q.push(s);
3 while(!Q.empty()) {
4   u = Q.pop();
5   forall ( (u,v) in E ) {
6     if (col[v] == white) {
7       col[v] == grey;
8       d[v] = d[u]+1;
9       pi[v] = u;
10      Q.push(v);
11    }
12  }
13  col[u] = black;
14 }

```



### 22.2. Laufzeit

$$\mathcal{O}(|V| + |E|)$$

Abbildung 22.1.: Grafik zum Beispielcode

**Begründung:** Jeder von  $s$  aus erreichbare Knoten wird nur einmal in die Queue aufgenommen und auch ihr entfernt. Für jeden Knoten muss nur einmal seine Adjazenzliste durchlaufen werden.

$$\Rightarrow \mathcal{O} \left( |V| + \sum_{v \in V} \text{outdeg}(v) \right)$$



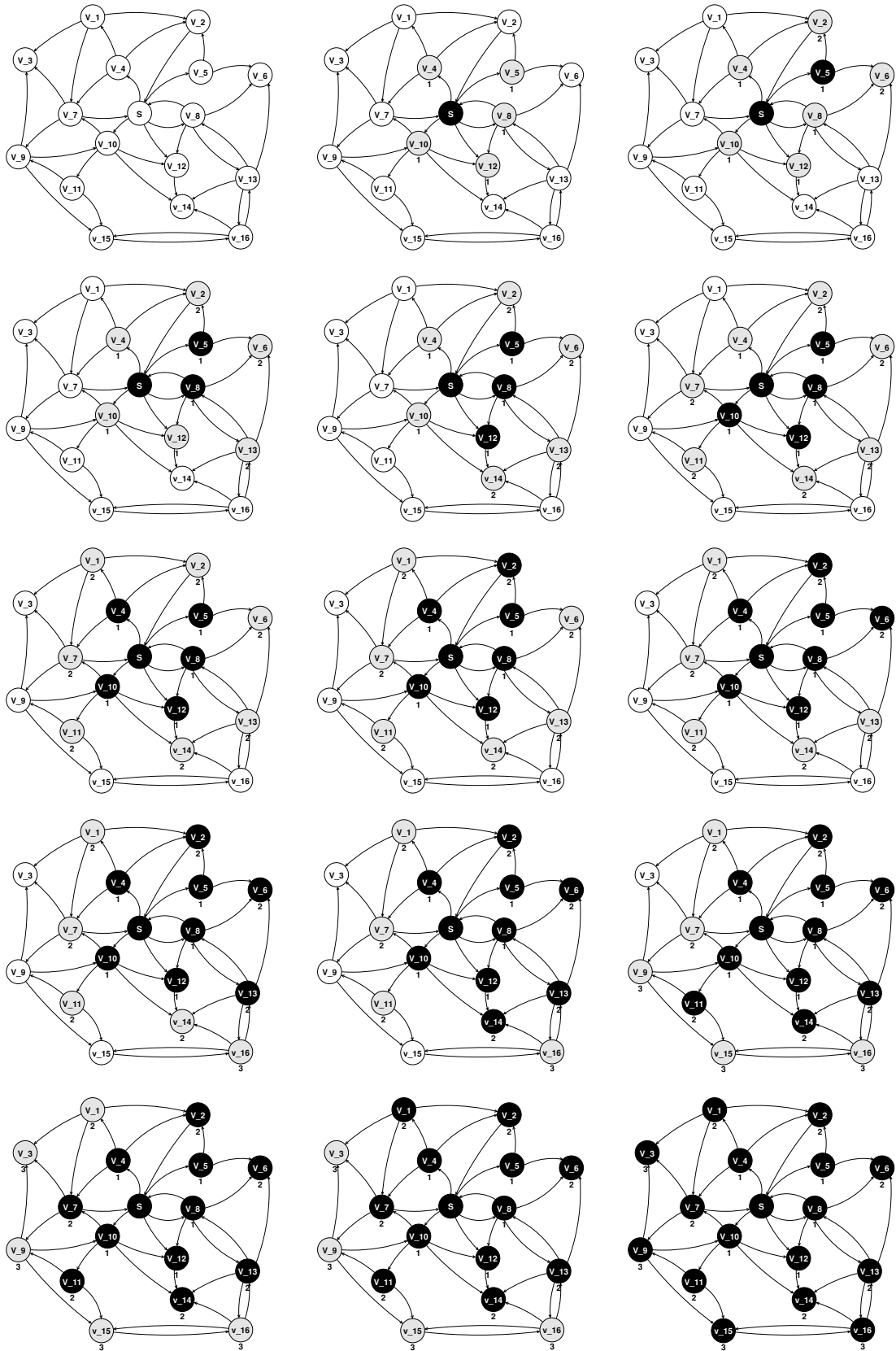


Abbildung 22.2.: Beispiel

## 22. BFS (Breadth-First Search) Breitensuche

### 22.2.1. Definition: Länge kürzesten Weges

$\delta(s, v)$  = Länge eines kürzesten Weges vom Startknoten  $s$  zum Knoten  $v$ .  
Setze  $\delta(s, v) = \infty$ , falls  $v$  nicht erreichbar von  $s$  aus.

### 22.2.2. Satz: Richtigkeit des Algorithmus

Nach Ablauf von BFS<sup>I</sup> gilt

$$\forall v \in V : d[v] = \delta(s, v)$$

### 22.2.3. Lemma 1: Dreiecksungleichung für kürzeste Wege

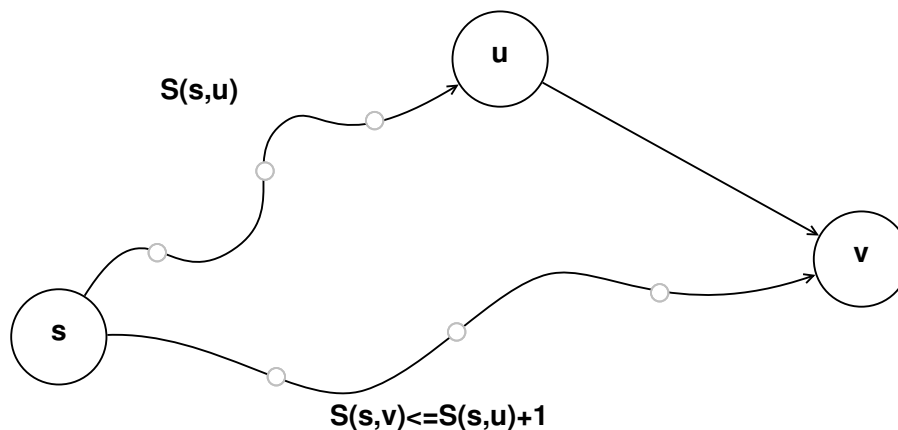


Abbildung 22.3.

### 22.2.4. Lemma 2

Zu jedem Zeitpunkt im Verlauf von BFS gilt:

$$\forall v \in V : d[v] \geq \delta(s, v)$$

**Beweis (induktiv über Zahl der Operationen, die d-Wert verändern)**

**Induktions-Anfang**

$$d[s] = 0$$

**Induktions-Schritt** Knoten  $v$  wird von  $u$  aus neu entdeckt

$$d[u] \geq \delta(s, u)$$

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \stackrel{D.U.}{\geq} \delta(s, v)$$

### 22.2.5. Lemma 3

Sei  $Q = (v_1, v_2, \dots, v_k)$  eine Queue, dann gilt stets:

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[v_1] + 1$$

<sup>I</sup>Breitensuche

**Beweis (induktiv über die Zahl der push- und pop-Operationen)****Induktions-Anfang**

$$d[s] = 0 \checkmark$$

**Induktions-Schritt****pop**

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[v_1] + 1 \stackrel{!}{\leq} d[v_2] + 1$$

**push**

$$d[u] = d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[u] + 1$$

Beachte Kante  $(u, v)$   $v$  ist weiß

$v = v_{k+1}$  wird gepusht

$$d[v_{k+1}] = d[v_1] + 1$$

Zustand von  $Q$  nach push

$$d[v_2] \leq d[v_3] \leq \dots \leq d[v_k] \leq d[v_1] + 1 = d[v_{k+1}] \quad \checkmark$$

**22.2.6. Satz: Richtigkeit des Algorithmus**

Nach Ablauf von  $\text{BFS}^{\text{II}}$  gilt

$$\forall v \in V : d[v] = \delta(s, v)$$

**Beweis durch Widerspruch**

Sei  $v \in V$ , so dass  $d[v] \neq \delta(s, v)$  am Ende des Algorithmus  $\stackrel{\text{Lemma 2}}{\implies} d[v] > \delta(s, v)$

Sei  $v$  so gewählt, dass es der erste Knoten ist mit der Eigenschaft, dass sein d-Wert falsch gesetzt wird.  
d.h. Alle d-Werte bis zu diesem Zeitpunkt sind korrekt.

Sei  $s \mapsto u' \rightarrow v$  ein kürzester Weg  $s$  zu  $v$

Betrachte die Situation bei Bearbeitung von  $u'$ :

**1. Fall**  $v$  ist in diesem Moment schwarz.

$$d[v] > \delta(s, v) = \delta(s, u') + 1 \geq \text{III} d[v] \quad \text{!}$$

**2. Fall**  $v$  ist in diesem Moment weiß.

$$d[v] > \delta(s, u') + 1 = d[u'] + 1 = \text{IV} d[v] \quad \text{!}$$

---

<sup>II</sup>Breitensuche

<sup>III</sup> $v$  vor  $u'$  aus  $Q$  entfernt und Lemma 3.

<sup>IV</sup>wegen Wahl von  $v$ ; d-Wert von  $u'$  muss also korrekt sein

22. BFS (Breadth-First Search) Breitensuche

**3. Fall**  $v$  ist grau.

$$d[v] > \delta(s, u') + 1 = d[u'] + 1 \geq d[u] + 1 = d[v]$$

$d[u] \leq d[u']$ , weil  $u$  vor  $u'$  aus  $Q$  entfernt  $\nmid$

q.e.d.

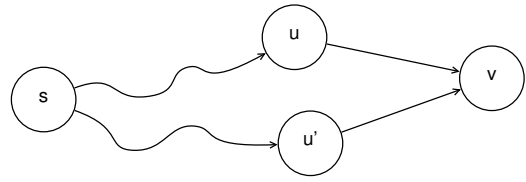


Abbildung 22.4.

## 23. Kürzeste Wege Algorithmen

### 23.1. Dijkstra-Algorithmus

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}_0^+$$

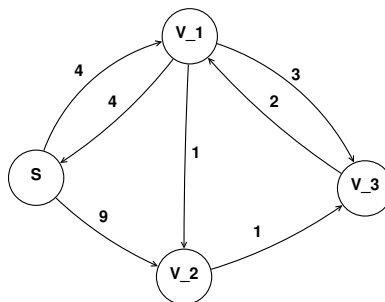


Abbildung 23.1.

Sei  $p = (s = v_0, v_1, v_2, \dots, v_k)$



Abbildung 23.2.

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) = \delta(s, v_k)$$

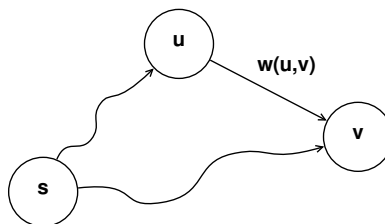


Abbildung 23.3.

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

```

1 relax(u,v,w) {
2   if(d[v] > d[u]+w(u,v) ) {
3     d[v] = d[u] + w(u,v);
4     Π[v] = u;
5   }
6 }

```

### 23. Kürzeste Wege Algorithmen

Betrachte Algorithmen zur kürzesten Wege Berechnung, die Distanzwerte nur mit Hilfe dieser relax-Funktion verändern, dann gilt:

$$d[v] \geq \delta(s, v) \quad \forall v \in V$$

**Beweis**

$$d[v] = d[u] + w(u, v) \stackrel{I.A.}{\geq} \delta(s, u) + w(u, v) \geq \delta(s, v)$$

Induktion über Zahl der reflex-Aufrufe

---

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}^{\geq 0}$$

```

1  forall (v ∈ V) {
2    d[v] = ∞;
3    Π[v] = NULL;
4  }
5  d[s] = 0;
6  S = ∅;
7  PriorityQueue PQ;
8  forall (v ∈ V)
9    PQ.insert((d[v], v));
10 while (!PQ.empty()) {
11   u = PQ.deleteMin();
12   forall ( (u, v) ∈ E ) {
13     if ( d[v] > d[u] + w(u, v) ) {
14       d[v] = d[u] + w(u, v);
15       Π[v] = u;
16       PQ.decreaseKey((d[v], v));
17     }
18   }
19   S = S ∪ {u};
20 }
```

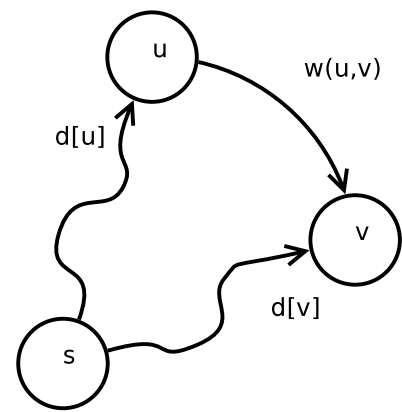


Abbildung 23.4.

**Satz:** Der Dijkstra Algorithmus berechnet alle d-Werte, so dass nach Ablauf des Algorithmus  $\forall v \in V$  gilt:  $d[v] = \delta(s, v)$ .

**Beweis:**

**Annahme:**

$$\exists v \in V : d[v] \neq \delta(s, v)$$

$$\stackrel{LemmaRelax}{\implies} d[v] > \delta(s, v)$$

Sei  $v$  so gewählt, dass  $v$  der erste Knoten mit der Eigenschaft ist, der mit deleteMin der PQ entnommen wird und nach Relaxation aller von ihm ausgehenden Kanten der Menge  $S$  hinzugefügt wird.

Betrachte einen kürzesten Weg  $s \rightsquigarrow v$

$$d[v] > \delta(s, v) \geq \overset{I}{\delta(s, y)} = d[y] = \overset{II}{d[x]} + w(x, y) = d[y] \geq \overset{III}{d[v]} \quad \nexists$$

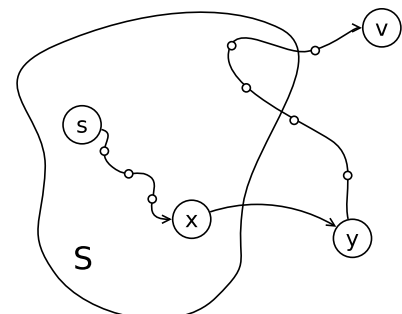


Abbildung 23.5.: Skizze

<sup>I</sup>weil Kantengewichte nicht negativ sein dürfen

<sup>II</sup>x wurde schon zu S hinzugefügt, hat also korrekten d-Wert  $d[x] = \delta(s, x)$

<sup>III</sup>weil v vor y aus der PQ entnommen wird.

### 23.1.1. Vorläufige Laufzeitanalyse von Dijkstra

PQ.insert	x	V
PQ.empty	x	V
PQ.deleteMin	x	V
PQ.decreaseKey	x	E

Mit balanciertem Suchbaum oder mit binärem Heap (siehe 2) können diese Operationen alle in Zeit  $\mathcal{O}(\log |V|)$  realisiert werden.  $\Rightarrow$  Gesamtlaufzeit:  $\mathcal{O}((|V| + |E|) \log |V|)$

Wir werden später zeigen, dass eine Laufzeit von  $\mathcal{O}(|V| \log |V| + |E|)$  möglich ist.

## 23.2. Bellman-Ford-Algorithmus

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}$$

**Voraussetzung**  $G$  enthält keine negativen Zyklen

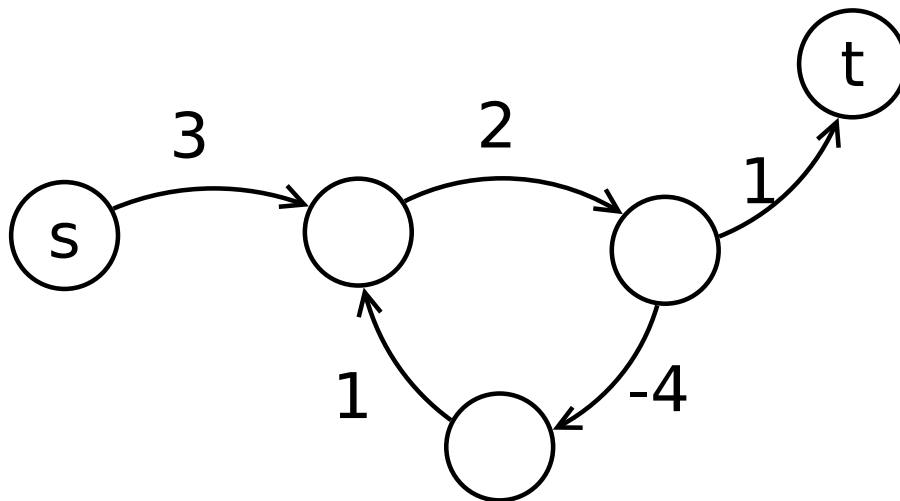


Abbildung 23.6.: Ein verbotener, negativer Zyklus

### 23.2.1. Pseudocode

```

1  forall(v ∈ V) {
2    d[v] = ∞;
3    π[v] = NULL;
4  }
5  d[s] = 0;
6  for(i = 1; i < |V|; i++)
7    forall((u,v) ∈ E)
8      if( d[v] > d[u] + w(u,v) ) {
9        d[v] = d[u] + w(u,v);
10       π[v] = u;
11     }

```

### 23.2.2. Laufzeit: Bellman-Ford

$$\mathcal{O}(|V| \cdot |E|)$$

## 23. Kürzeste Wege Algorithmen

### 23.2.3. Korrektheitsbeweis: Bellman-Ford

**Invariante:** Nach den  $i$ -ten Schleifendurchlauf sind alle Kürzesten Wege korrekt berechnet, die  $\leq i$  Kanten benutzen.

**Beweis: Induktion über  $i$**

#### Induktionsanfang

$i = 0$   $d[s] = 0 = \delta(s, s)$ , da keine negativen Zyklen vorliegen.

### 23.2.4. Induktionsschritt: $i \rightarrow i + 1$

Betrachte kürzesten Weg mit  $i + 1$  Kanten:

$$s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_i \rightarrow v_{i+1}$$

Aufgrund der Induktionsannahme<sup>IV</sup> gilt:  $d[v_i] = \delta(s, v_i)$ , weil  $s = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i$  ein kürzester Weg  $s \rightsquigarrow v_i$  mit  $i$  Kanten ist. Da alle Kanten in der inneren Schleife einmal relaxiert werden, trifft dies insbesondere auf die Kante  $(v_i, v_{i+1})$  zu:

$$d[v_{i+1}] = d[v_i] + w(v_i, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$$

**Frage:** Warum folgt aus der Gültigkeit dieser Invariante die Korrektheit des Algorithmus?

**Antwort** Alle kürzesten Wege benutzen höchstens  $|V| - 1$  Kanten, ansonsten hätten sie einen Zyklus mit Gewicht  $\geq 0$ , den man auch weglassen kann.

```
1 //Erkennung der Existenz negativer Zyklen
2 forall((u,v) ∈ E)
3   if(d[v] > d[u] + w(u,v))
4     negativer Zyklus
```

---

<sup>IV</sup>Die Invariante



## 24. All-Pairs-Shortest Path Algorithmen

### 24.1. Allgemein

Distanzmatrix  $D$  für einen Graphen  $G = (V, E)$   $V = v_1, v_2, \dots, v_n$ ,  $w : E \mapsto \mathbb{R}$

$$d_{ij} = \begin{cases} 0 & \text{für } i = j \\ w(v_i, v_j) & \text{für } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}$$

$$D = (d_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,n}} \in \mathbb{R}^{n \times n}$$

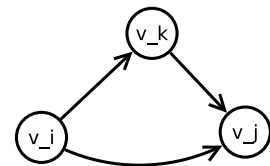


Abbildung 24.1.: Grafik

$$d_{ij}^{(2)} = \min(d_i^{(1)} j, \min_{k=1,\dots,n} (d_{ik}^{(1)} + d_k^{(1)} j))$$

$$D^{(2)} = D^{(1)} \circ D^{(1)} = \min(d_{ik}^{(1)} + d_k^{(1)} j)$$

Vergleich zu Matrixmultiplikation

$$C = A \circ B, \text{ mit } A, B \in \mathbb{R}^{n \times n}$$

$$C_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

im Ring  $(\mathbb{R}, +, \cdot)$

$$C_{ij} = \left( \sum_{k=1,\dots,n} A_{ik} + B_{kj} \right)$$

**Kommutativgesetz**

$$\min(\min(a, b), c) = \min(a, b, c)$$

im „Ring“<sup>I</sup>  $(\mathbb{R}, \min, +)$

**Distributivgesetz**

$$a + \min(b, c) = \min(a + b, a + c)$$

**Assoziativgesetz**

$$A \circ (B \circ C) = (A \circ B) \circ C$$

**Ziel:**  $D^{(n)\text{II}} = D^{(1)} \circ D^{(1)} \circ \dots \circ D^{(1)}$

**Es gilt:**  $D^{(n)} = D^{(n+m)}$  für  $m \geq 1$

<sup>I</sup>der keiner ist

<sup>II</sup>In der Potenz stehen die Anzahl der betrachteten Kanten. n entspricht allen Kanten

## 24. All-Pairs-Shortest Path Algorithmen

### 24.1.1. Laufzeit zur Berechnung von $D^{(n)}$

**Naiv:**  $\mathcal{O}(n^4)$

$$D^{(2)} = D^{(1)} \circ D^{(1)}$$

$$D^{(4)} = D^{(2)} \circ D^{(2)}$$

$$D^{(8)} = D^{(4)} \circ D^{(4)}$$

$\vdots$

$$D^{(2^i)} = D^{(2^{i-1})} \circ D^{(2^{i-1})}$$

Schrittzahl  $i$  so wählen, dass  $2^i \geq n$

**sukzessives Quadrieren:**  $\mathcal{O}(n^3 \log n)$

## 24.2. Floyd-Warshall-Algorithmus

```
1  for (k = 1; k ≤ n; k++)
2    for (i = 1; i ≤ n; i++)
3      for (j = 1; j ≤ n; j++)
4        d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

**Laufzeit**  $\mathcal{O}(n^3)$

### 24.2.1. Korrektheitsbeweis:

**Invariante** Nach dem  $k$ -ten Schleifendurchlauf entspricht  $d_{ij}$  der Weglänge eines kürzesten Weges  $p$  von  $v_i$  nach  $v_j$ , wobei nur Zwischenknoten erlaubt sind, mit Index  $\leq k$

$$p : v_i \rightarrow v_{l_1} \rightarrow v_{l_2} \rightarrow \dots \rightarrow v_{l_m} \rightarrow v_j$$

d.h.  $1 \leq l_1, l_2, \dots, l_m \leq k$

### 24.2.2. Beweis der Invariante durch Induktion nach $k$

$k = 0$ : Nach der Initialisierung von  $D$ , also vor dem 1. Schleifendurchlauf, gilt obige Invariante.

$k - 1 \rightarrow k$ :

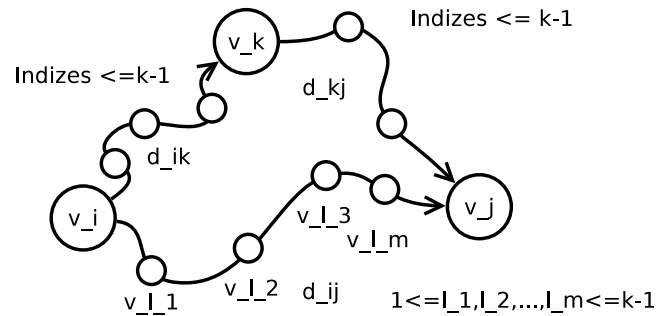


Abbildung 24.2.: Beweis der Invariante

Durch die Operation  $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$  wird die Invariante sichergestellt.

## 24.3. Naive Lösung des All-Pairs Problems durch $|V|$ -malige Anwendung von Bellman-Ford oder Dijkstra-Algorithmus

**Bellman-Ford**  $\mathcal{O}(|V| \cdot |V| \cdot |E|) = \mathcal{O}(|V|^2 \cdot |E|)$

**Dijkstra**  $\mathcal{O}(|V| \cdot (|V| \cdot \log |V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log |V|)$

## 24.4. Johnson-Algorithmus

**Idee:** Neugewichtung der Kanten, so dass keine negativen Kantengewichte mehr vorhanden sind. Anschließend  $|V|$ -mal Dijkstra-Algorithmus ausführen.

**Naiver Ansatz**

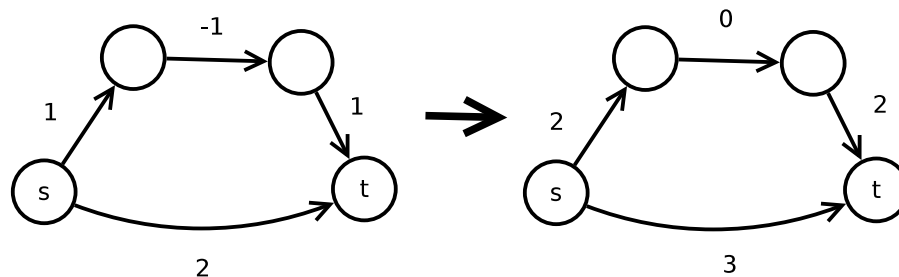


Abbildung 24.3.: Naiver Ansatz, kürzester Weg wird zerstört

## 24. All-Pairs-Shortest Path Algorithmen

### Neuer Ansatz

$$w'(u, v) = \text{pot}^{\text{III}}(u) - \text{pot}(v) + w(u, v) \geq 0$$

Mit dieser Neugewichtung gilt, dass kürzeste Wege bzgl.  $w$  den kürzesten Wegen bzgl.  $w'$  entsprechen.

$$p : s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots v_i \rightarrow v_{i+1} \rightarrow \dots v_k = t$$

$$w'(p) = \sum_{i=0}^{k-1} w'(v_i, v_{i+1}) = \sum_{i=0}^{k-1} [\text{pot}(v_i) - \text{pot}(v_{i+1}) + w(v_i, v_{i+1})]$$

$$\stackrel{\text{Teleskopsumme}}{=} \text{pot}(v_0) - \text{pot}(v_k) + \sum_{i=1}^{k-1} w(v_i, v_{i+1}) = \text{pot}(s) - \text{pot}(t) + w(p)$$

**d.h.** Alle kürzesten Wege  $s \rightsquigarrow t$  unterscheiden sich bzgl.  $w'$  im Vergleich zu  $w$  nur um eine feste additive Konstante  $\text{pot}(s) - \text{pot}(t)$

$$\text{pot}(u) - \text{pot}(v) + w(u, v) \geq 0$$

$$\text{pot}(v) \leq \text{pot}(u) + w(u, v)^{\text{IV}}$$

$$\text{pot}(v) = \delta(z, v)$$

$$G' = (V', E') \quad V' = V \cup z, E' = E \cup (z, v) | v \in V \quad \text{mit } w'(z, v) = 0$$

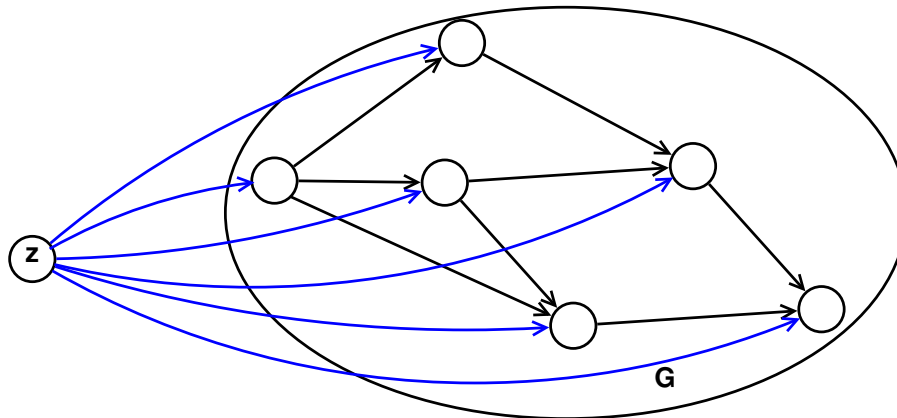


Abbildung 24.4.: Die blau markierten Kanten haben die Länge 0

- Löse single-source-shortest-Path Problem in  $G'$  mit  $z$  als Startknoten
- setze  $\text{pot}(v) = \delta_{G'}(z, v)^{\text{V}}$
- Neugewichtung
- $|V|$ -mal Dijkstra

### 24.4.1. Laufzeit des Johnson-Algorithmus

$$\mathcal{O}(|V| \cdot |E| + |V| \cdot (|V| \cdot \log |V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot |V|)$$

<sup>III</sup>Potentialfunktion

<sup>IV</sup>Dreiecksungleichung

<sup>V</sup>berechnet mit Bellman-Ford

## 25. Minimal aufspannende Bäume MST

### Eingabe

$G = (V, E)$   $E$  ungerichtet  $(u, v) \in E \Rightarrow (v, u) \in E$  mögliche Notation  $\{u, v\}$

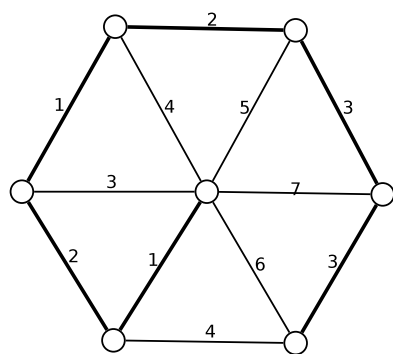
$w : E \rightarrow \mathbb{R}$

### Gesucht

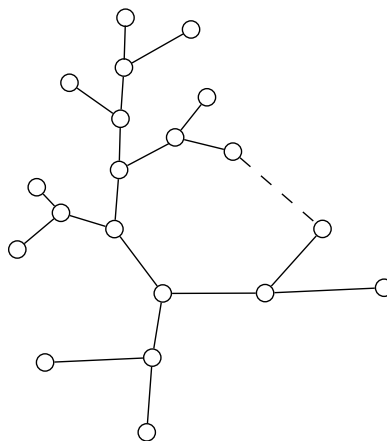
Baum  $T \subseteq E$

$G_T = (V, T)$  zusammenhängend (zykelfrei)

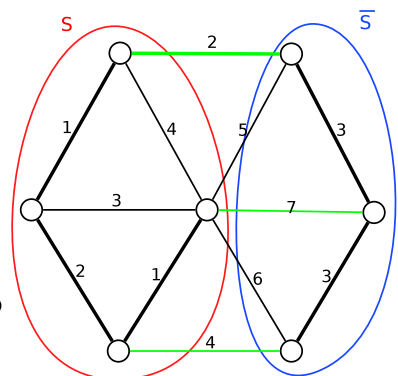
$$w(T) = \sum_{e \in T} w(e) \text{ minimal}$$



(a) Beispiel für einen Spannbaum in einem Graphen



(b) Beispiel für einen Spannbaum



(c) Grün markiert sind alle Kanten, die von  $S$  nach  $\bar{S}$  verlaufen. Wähle die Kante mit dem kleinsten Gewicht, also 2, aus.

Frage  $|T| = ?$

Antwort  $|T| = |V| - 1$

### 25.1. Greedy-Algorithmen zur Lösung des MST-Problems:

Starte mit  $T = \emptyset$ , nehme sukzessive Kanten zu  $T$  hinzu, so dass nach  $|V| - 1$  Schritten der gesuchte MST entstanden ist. Dabei benötigen wir ein Kriterium, das sicherstellt, dass gewählte Kanten zur Gesamtlösung dazugehören.

### 25.2. Schnitt-Lemma:

Betrachte eine Aufteilung (Schnitt) der Knotenmenge  $V$  in  $S$  und  $\bar{S} = V \setminus S$

und Kanten  $(u, v) \in E \cap S \times \bar{S}$

Sei  $e \in E \cap S \times \bar{S}$  mit  $w(e) \leq w(e') \forall e' \in E \cap S \times \bar{S}$  dann gibt es einen MST mit  $e \in \text{MST}$

### 25.2.1. Beweis für das Schnitt-Lemma

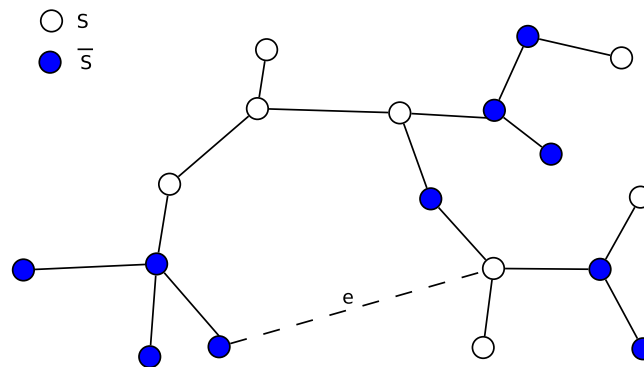


Abbildung 25.2.

Sei  $e$  eine „sichere“ Kante aus dem Schnitt-Lemma.

o.B.d.A.  $u \in S$  und  $v \in \bar{S}$ .

Es gibt eine Zykel in  $T \cup \{e\}$  und darin eine Kante  $e' \in S \times \bar{S}$  mit  $w(e') \geq w(e)$ .

Ersetze  $T' = T \cup \{e\} \setminus \{e'\}$

$w(T') \leq w(T) \Rightarrow w(T') = w(T)$  weil  $T$  ein MST.

q.e.d.

### 25.3. Algorithmus von Kruskal

sortiere Kanten nach ihrem Gewicht aufsteigend

$T = \emptyset$

```

1 forall (u,v) ∈ E in sortierter Reihenfolge {
2   if (find(u) == find(v)) continue;
3   T = T ∪ {(u,v)};
4   union(u,v);
5 }
```

#### Effizienz von Kruskal

**Sortieren:**  $\mathcal{O}(|E| \cdot \log |E|) = \mathcal{O}(|E| \log |V|)$

$2|E|$  viele find-Operationen  $\mathcal{O}(1)$

$|V| - 1$  viele union-Operationen  $\mathcal{O}(|V|)$  naiv

$\mathcal{O}(|E| \log |V| + |E| \cdot 1 + (|V| - 1)|V|) =$

$\mathcal{O}(|E| \log |V| + |V|^2)$

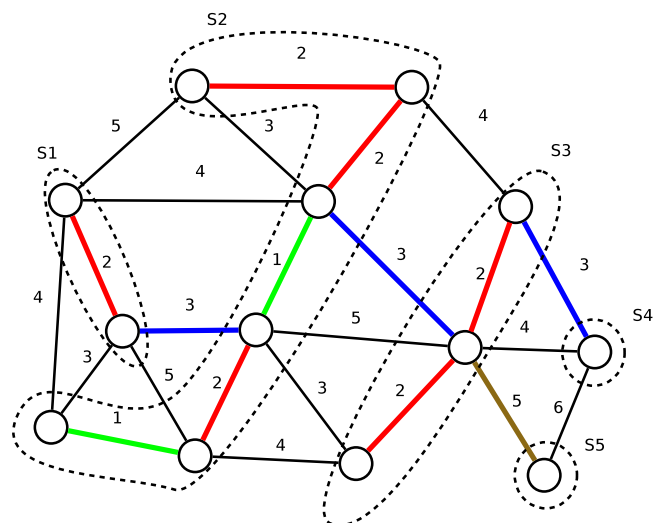


Abbildung 25.3.: Reihenfolge  
grün→rot→blau→braun

#### Idee zum Aufbau einer Union-Find-Datstruktur

Jeder Knoten trägt eine Komponentennummer, die in einem Feld vermerkt ist. Die find-Operation ist durch einen Feld-Zugriff realisierbar. Um die union-Operation zu realisieren, verwalten wir die Knoten einer Komponente in einer einfach verketteten Liste und merken uns die Listenlänge. Wenn zwei Komponenten fusionieren, benennen wir die Komponente mit der kleineren Knotenzahl um, indem wir die zugehörige Liste durchlaufen und die Umbenennung im Feld vornehmen. Und die beiden betroffenen Listen müssen konkateniert werden.

**Beobachtung:** Ein einzelner Knoten erfährt höchstens  $\log |V|$  viele Umbenennungen seiner Komponentennummer, da sich bei jeder Umbenennung die Größe der Komponente zu der er gehört, verdoppelt.

## 25.4. Einfache Union-Find-Datenstruktur

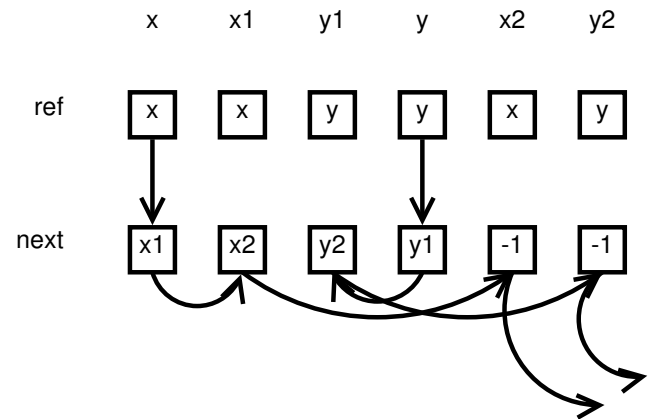
	0	1	2	...	$n-1$
ref	0	1	2	...	$n-1$
size	1	1	1	...	1
next	-1	-1	-1	...	-1

$n = |V|$

```

1  class Partition
2      int[] ref,size,next;
3      Partition(int n) {
4          ref = new int[n];
5          size = new int[n];
6          next = new int [n];
7          for (int i = 0; i < n; i++) {
8              ref[i] = i;
9              size[i] = 1;
10             next[i] = -1;
11         }
12     }
13     int find(int v) {
14         return ref[v];
15     }
16     void union(int u, int v) {
17         int x = ref[u];
18         int y = ref[v];
19
20         if (size[x] > size[y]) {
21             x = ref[v];
22             y = ref[u];
23         }
24         int h = next[y];
25         next[y] = x;
26         int z = y;
27         while( next[z] ≥ 0) {
28             z = next[z];
29             ref[z] = y;
30         }
31         next[z] = h;
32         size[y] = size[y] + size[x];
33     }

```

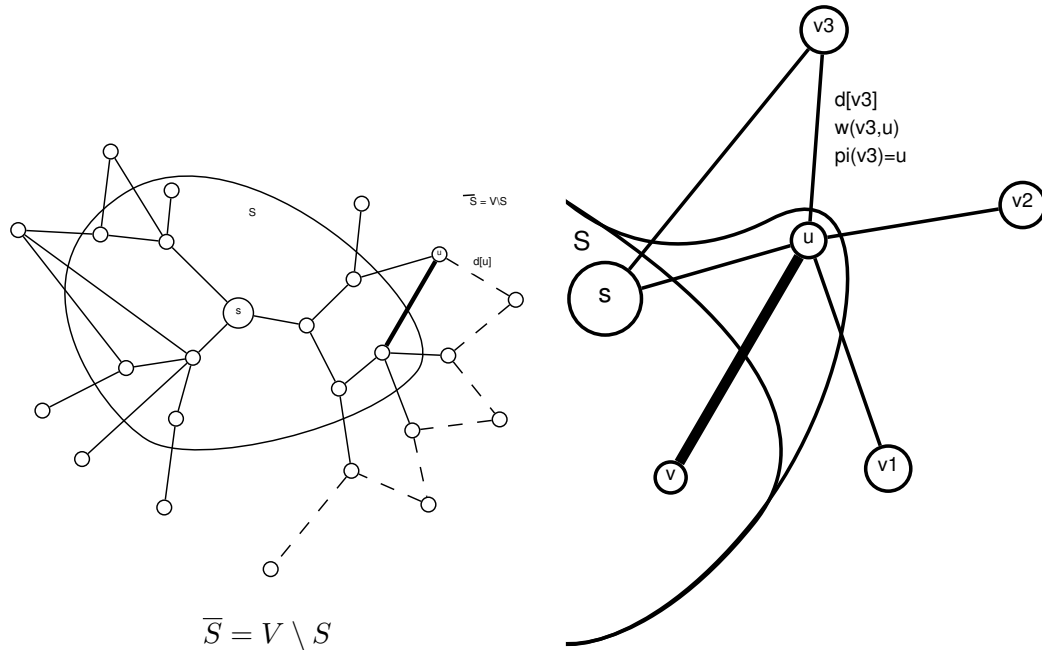


### 25.4.1. Laufzeit Kruskal

$$\mathcal{O}(|E| \cdot \log |V| + |E| + |V| \cdot \log |V|)$$

$$\mathcal{O}(|E| \cdot \log |V|)$$

## 25.5. Prim-Algorithmus zur Berechnung eines MST



$$T = T \cup \{(u, v)\}$$

$$T_{MST} = \{(v, \pi(v)) \mid v \in V \setminus \{s\}\}$$

```

1  PriotityQueue PQ;
2  forall(v ∈ V) {
3      d[v] = ∞;
4      π[v] = NULL;
5      inTree[v] = false;
6      PQ.insert(v, d[v]);
7  }
8  d[s] = 0;
9  PQ.decreaseKey(s, d[s]); //S ≐ PQ
10 T = ∅ //S ≐ {v ∈ V | inTree[v] = true}
11 while(!PQ.empty()) {
12     u = PQ.deleteMin();
13     forall((u, v) ∈ E) {
14         if(inTree[v] == true) continue;
15         if(d[v] > w(u, v)) {
16             d[v] = w(u, v);
17             π[v] = u;
18             PQ.derceaseKey(v, d[v]);
19         }
20     }
21     inTree[u] = true;
22     T = T ∪ (u, v); //T = T ∪ {(u, π[u])} u ≠ s
23 }

```



### 25.5.1. Korrektheit des Prim-Algorithmus

Korrektheit von Prim folgt unmittelbar aus dem Schnitt-Lemma, denn der Algorithmus stellt sicher, dass stets eine Kante gewählt wird, die mit minimalem Gewicht über den Schnitt  $(S, \bar{S})$  führt.

### 25.5.2. Laufzeit des Prim-Algorithmus

$ V  \times \text{PQ.insert}$	$\mathcal{O}(1)$	In Summe $\mathcal{O}( E  +  V  \cdot \log  V )$
$ V  \times \text{PQ.empty}$	$\mathcal{O}(1)$	
$ V  \times \text{PQ.deleteMin}$	$\mathcal{O}(\log  V )$	
$ E  \times \text{PQ.decreaseKey}$	$\mathcal{O}(1)$	

### 25.5.3. Beispiel des Prim-Algorithmus:

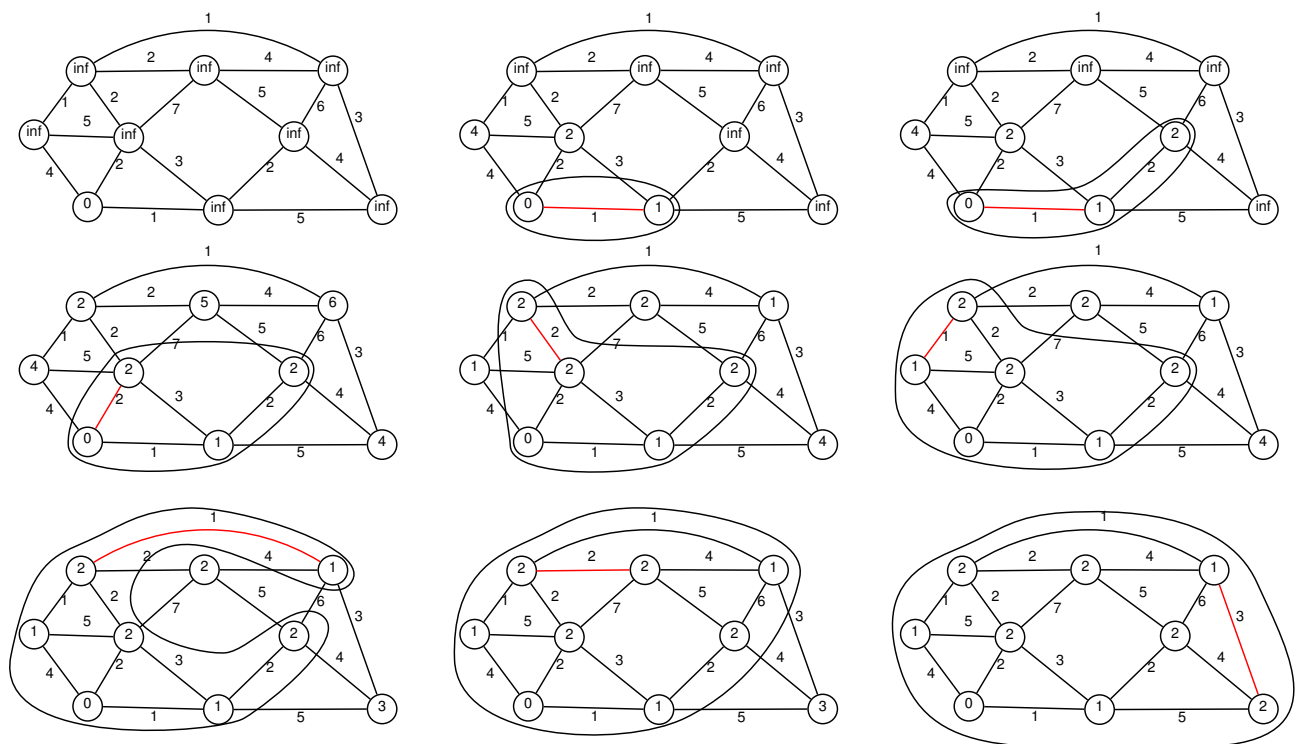


Abbildung 25.5.

## 26. Priority-Queue mittels Fibonacci-Heaps

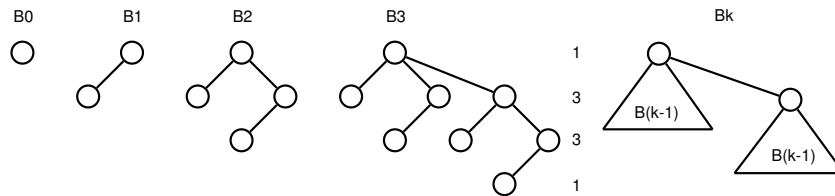


Abbildung 26.1.: Binomial-Bäume

Es gilt:

Zahl der Knoten auf Level  $i$  ist  $\binom{k}{i}$

$$\binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$$

Gesamtzahl aller Knoten =  $2^k$

$$\begin{array}{cccccc} n=0: & & & & & 1 \\ n=1: & & & 1 & & 1 \\ n=2: & & 1 & & 2 & & 1 \\ n=3: & 1 & & 3 & & 3 & & 1 \\ n=4: & 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

## 26.1. Operationen eines Binomial-Heaps

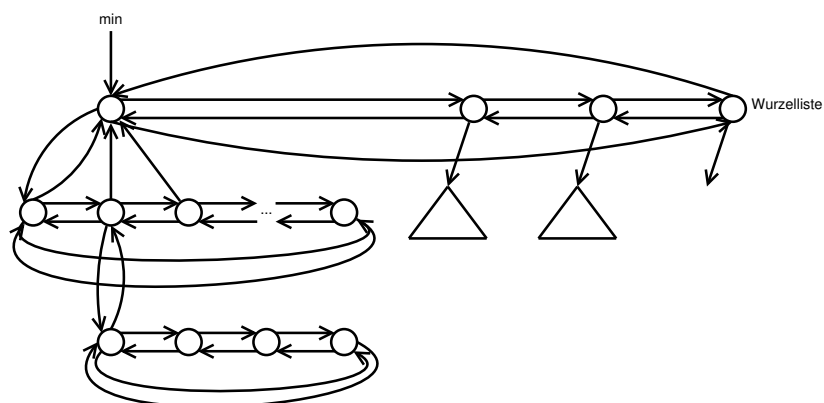


Abbildung 26.2.: Aufbau

- insert
- deleteMin
- decreaseKey

**Idee** Für jeden Knoten wird gelten, dass die Zahl aller Nachfahren  $\geq \Phi^k$   $k$  = Knotengrad

#### Insert

Einzelner Knoten wird einfach in die Wurzelliste gehängt und Minimum wird aktualisiert.

#### DeleteMin

Lösche den Minimumsknoten und übernehme alle seine Kindknoten in die Wurzelliste. Konsolidiere anschließend die Wurzelliste. Nach dem Konsolidieren hat die Wurzelliste nur noch eine „kleine“ Länge und wir bestimmen das neue Minimum durch einen Durchlauf durch diese Liste.

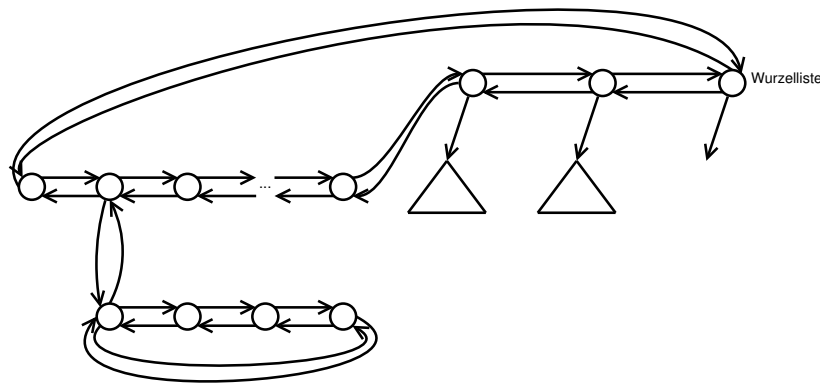


Abbildung 26.3.: DeleteMin-Operation

#### decreaseKey

Wir können davon ausgehen, dass wir den betroffenen Knoten kennen. Falls der erniedrigte Schlüssel des Knotens kleiner als der Schlüssel des Vaterknotens wird, lösen wir den Knoten aus der Kindliste des Vaters und setzen ihn in die Wurzelliste. Wir markieren den Vater, dass er einen Kindknoten verloren hat. Sollte der Vater schon eine Markierung tragen, so wird auch der Vater Knoten abgelöst und in die Wurzelliste gesetzt. Dieser Prozess kann sich kaskadenartig bis zur jeweiligen Wurzel fortsetzen. Bei Aufnahme eines abgelösten Knotens in die Wurzelliste, muss auch das Minimum aktualisiert werden.

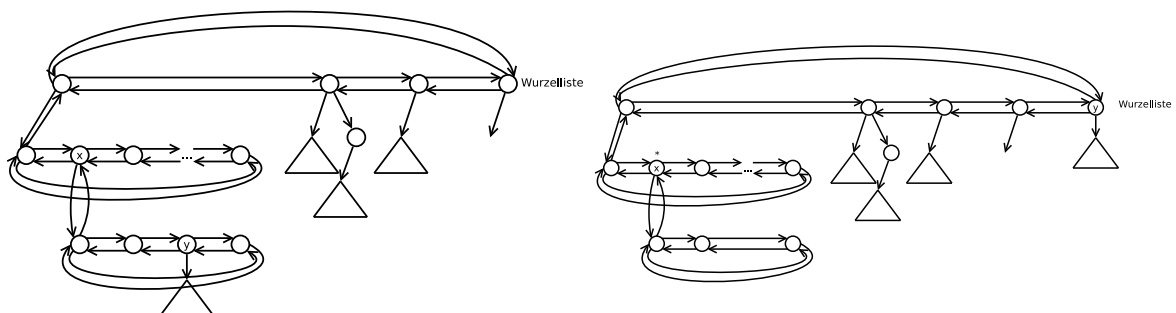


Abbildung 26.4.: decreaseKey-Operation

### 26.1.1. Konsolidierung der Wurzelliste

Nach dem „Lazy Evaluation“-Prinzip wird dieser Vorgang nur nach einem `deleteMin` angestoßen, um die Wurzelliste zu verkürzen. Wir nutzen ein einfaches Feld hinreichender Größe, um temporär Knoten, entsprechend ihren Grades, zu verwalten. Wir durchlaufen die Wurzelliste. Wenn wir einen Knoten vom Grad  $k$  antreffen, schreiben wir ihn in das Feld an Position  $k$  bzw. verschmelzen ihn mit dem Knoten von Grad  $k$ , den wir dort antreffen. Dadurch entsteht gegebenenfalls ein neuer Knoten vom Grad  $k+1$  der an Position  $k+1$  im Feld zu setzen ist. Es kann also zu weiteren Fusionsoperationen kommen. Analogie zu der Übertragungsfortpflanzung beim Binärzähler.

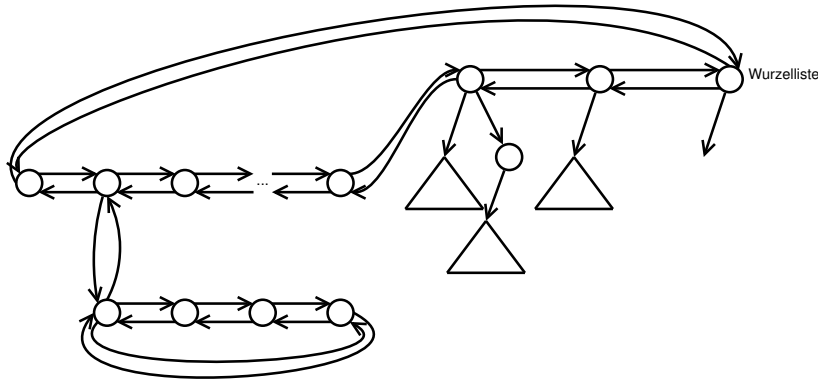


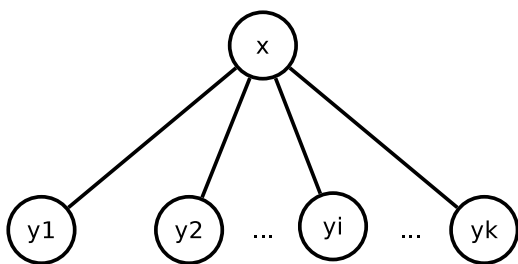
Abbildung 26.5.: Konsolidierungs-Operation

## 26.2. Eigenschaften

### 26.2.1. Lemma: Knotenanzahl im Unterbaum

Für jeden Knoten  $x$  in einem Fibonacci-Heap gilt, dass die Zahl aller Knoten im Unterbaum von  $x$  mindestens  $\Phi^k$  beträgt, wobei  $k = \text{grad}(x)$ .

### 26.2.2. Beweis



Seien  $y_1, y_2, \dots, y_k$  die aktuellen Kindknoten von  $x$ , nummeriert in der Reihenfolge, wie sie (letztmalig) Kindknoten von  $x$  geworden sind. Zum Zeitpunkt, zu dem  $y_i$  Kindknoten von  $x$  geworden ist, existieren bereits die  $i-1$  Kindknoten  $y_1, y_2, \dots, y_{i-1}$ .  $\Rightarrow \text{grad}(x) \geq i-1$

$y_i$  kann nur Kind von  $x$  werden, wenn  $x$  und  $y_i$  gleichen Grad haben.  $\Rightarrow \text{grad}(y_i) \geq i-1$

Da  $y_i$  im Folgenden höchstens einen Kindknoten verlieren kann, gilt:

$$\text{grad}(y_i) \geq i-2$$

Abbildung 26.6.: Schaubild zum Beweis

Sei  $S_k$  die Mindestanzahl von Knoten in einem Unterbaum eines Knoten  $x$  vom Grad  $k$

$$S_k \geq 1 + 1 + \sum_{i=2}^k S_{i-2}$$

Wir zeigen

$$S_k \stackrel{(2)}{\geq} f_{k+2} \stackrel{(1)}{\geq} \Phi^k$$

<sup>1</sup> $k-2$ -te Fibonacci Zahl

(1)

$f_0$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$\dots$
0	1	1	2	3	5	

$$f_{k+2} \geq \Phi^k$$

$$k = 0 : f_2 = 1 \geq \Phi^0 = 1$$

$$k = 1 : f_3 = 2 \geq \Phi^1 = 1,6181\dots$$

$$f_{k+2} = f_{k+1} + f_k \geq \Phi^{k-1} + \Phi^{k-2} = {}^{\text{II}}\Phi^{k-2}(\Phi + 1) = \Phi^{k-2} \cdot \Phi^2 = \Phi^k$$

(2)

$$S_k \geq f_{k+2}?$$

$$S_k \geq 2 + \sum_{i=2}^k S_{i-2}$$

$$k = 0 : S_0 \geq f_2 = 1 \quad \checkmark$$

$$k = 1 : S_1 \geq f_3 = 2 \quad \checkmark$$

$$S_k \geq 2 + \sum_{i=2}^k f_i, \text{ da wegen Induktions-Annahme } S_{i-2} \geq f_{(i-2)+2} = f_i \text{ f\"ur } i < k$$

**Zu zeigen:**

$$2 + \sum_{i=2}^k f_i \geq f_{k+2}$$

**Es gilt:**

$$1 + \sum_{i=1}^k f_i = f_{k+2}$$

$$k = 0 : 1 = f_2 \quad \checkmark$$

$$k = 1 : 1 + f_1 = f_3 = 2 \quad \checkmark$$

$$1 + \sum_{i=1}^{k+1} f_i = (1 + \sum_{i=1}^k f_i) + f_{k+1} = f_{k+2} + f_{k+1} = f_{k+3}$$

q.e.d.

Aus dem Lemma folgt, dass in einem Fibonacci-Heap für  $n^{\text{III}}$  Elemente zu keinem Zeitpunkt ein Knoten vom Grad  $\log_{\phi} n = k$  auftauchen kann. Insbesondere ist die Wurzelliste nach einer Konsolidierung auch nur  $\log_{\phi} n$  lang, weil dort nur Knoten unterschiedlichen Grades auftauchen.

---

${}^{\text{II}}\Phi^2 = \Phi + 1$   
 ${}^{\text{III}}\Phi^k \leq n$

### 26.2.3. Satz: Kosten der Operationen

Mit einem Fibonacci-Heap lassen sich die Operationen `insert`, `deleteMin` und `decreaseKey` mit folgenden amortisierten Kosten realisieren:

<code>insert</code>	$\mathcal{O}(1)$
<code>deletemin</code>	$\mathcal{O}(\log n)$
<code>decreaseKey</code>	$\mathcal{O}(1)$

### 26.2.4. Beweis

Wir verwenden die Bankkonto-Methode zur amortisierten Analyse nach folgendem Schema:

Jeder Knoten in der Wurzelliste wird mit einer  $\text{RE}^{\text{IV}}$  bespart und jeder markierte Knoten der einen Kindknoten verloren hat wird mit 2 RE bespart.

**Bemerkung** Wurzelknoten tragen keine Markierung, obwohl sie eventuell Kindknoten verloren haben.

Wir zeigen nun, dass die oben genannten Kosten für die einzelnen Operationen ausreichen, damit im gesamten Verlauf das Kontoführungsschema, ohne Schulden machen zu müssen, aufrecht erhalten werden kann.

`insert` Einfügen in Wurzelliste +1RE Investition  $\in \mathcal{O}(1)$

`deleteMin` Alle Kindknoten des gelöschten Knotens wandern in die Wurzelliste.

Dafür müssen wir  $\log_{\phi} n$  viele RE investieren. Die Konkatenation der doppelt verketteten Listen kostet nur konstante Zeit. Der ganze Konsolidierungsprozess kann bezahlt werden durch die RE auf den Wurzelknoten. Die anschließende Minimumsuche kostet nur  $\mathcal{O}(\log n)$ , weil die Wurzelliste höchstens  $\log_{\phi} n$  viele Elemente hat.

`decreaseKey`

**Behauptung** Es genügen 4 RE pro Operation

Vorgehensweise:

- 1RE für die Aufnahme eines abgelösten Knotens in die Wurzelliste
- 2RE für die Markierung des Vaterknotens
- 1RE für „sonstige“ konstante Kosten(Pointeraktualisierungen).

Ein Vaterknoten, der schon markiert ist, hat aufgrund der Gültigkeit der Bankkontoführung schon 2RE und bekommt vom abgelösten Kindknoten noch 2 RE. Damit hat er 4RE für seine eigene Ablösung zur Verfügung. Dieses Schema lässt sich also fortsetzen und die Kosten einer Ablösekaskade lassen sich damit decken.

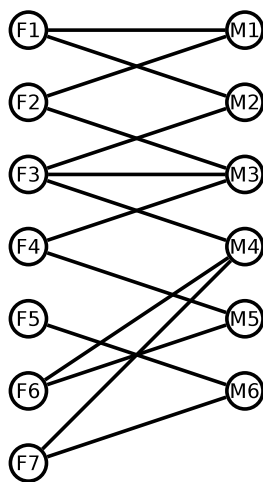
q.e.d.

---

<sup>IV</sup>Rechen Einheit

## 27. Das Heiratsproblem - Maximum cardinality matching in bipartiten Graphen

### 27.1. Allgemein



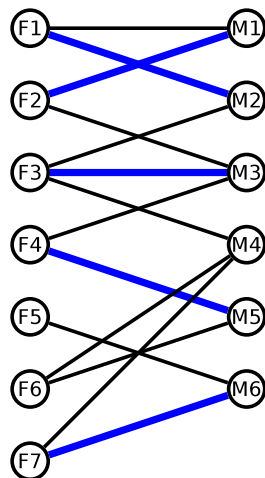
$$G = (V_1 \cup V_2, E)$$

$E' \subseteq E$  heißt Matching, wenn jeder Knoten zu höchstens einer Kante aus  $M$  inzident ist. Freie Knoten sind an keiner Matching-Kante beteiligt.

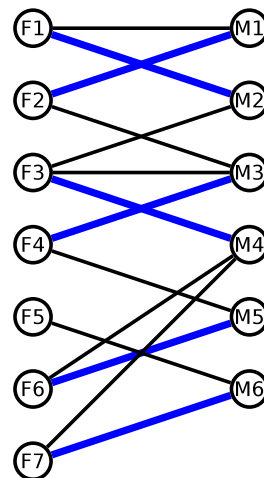
$M$  heißt maximales Matching, wenn  $M$  durch Hinzunahme einer weiteren Kante nicht vergrößert werden kann.

Gesucht ist ein maximum-Matching  $M^*$  mit  $|M^*| \geq |M| \forall M \text{ Matching}$ .

Abbildung 27.1.: Ausgangsproblem



(a) Nicht optimales Matching



(b) optimales Matching

## 27. Das Heiratsproblem

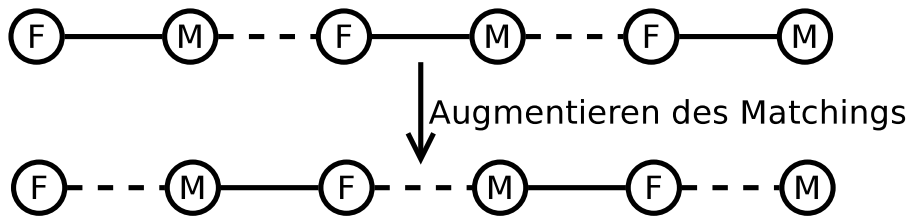


Abbildung 27.3.: Alternierender Pfad

Alternierender Graph, der mit einem Singleknoten startet und endet, nennt man einen augmentierten Pfad.

Zum finden eines augmentierten Pfades verwenden wir folgenden Graphen  $G_M = (V_1 \cup V_2 \cup \{s\}, E')$

$$E' = \{(v_1, v_2) | v_1 \in V_1, v_2 \in V_2, (v_1, v_2) \in E \setminus M\} \cup \{(v_2, v_1) | v_1 \in V_1, v_2 \in V_2, (v_1, v_2) \in M\} \cup \{(s, v_1) | v_1 \in V_1 \text{ frei}\}$$

Mit Hilfe von BFS oder DFS können wir in  $G_M$  augmentierende Pfade leicht finden. Also

$$\text{Zei } \mathcal{O}(|V| + |E|)$$

### 27.1.1. Lemma: (Berge)

Ein Matching  $M$  ist ein maximum-Matching  $\Leftrightarrow$  Es gibt keinen  $M$ -augmentierenden Pfad.

### 27.1.2. Beweis:

“ $A \Rightarrow B$ ”

$\neg B \Rightarrow \neg A$  Es gibt  $M$ -augmentierenden Pfad  $\Rightarrow M$  ist kein maximum Matching

“ $A \Leftarrow B$ ”

$\neg A \Rightarrow \neg B$  Sei  $M$  noch kein maximum Matching.

**z.z.** Es gibt ein  $M$ -augmentierenden Pfad

$M^*$  sei ein maximum Matching, d.h.  $|M^*| > |M|$ .

Betrachte den Graphen  $\tilde{G} = (V_1 \cup V_2, M \oplus M^*)$

Alle Knoten in  $\tilde{G}$  haben höchstens Grad 2, ansonsten wäre ein Knoten inzident zu zwei Kanten aus dem Gleichen Matching  $M$  oder  $M^*$ .

$\tilde{G}$  besteht aus einzelnen Knoten, Pfaden gerader oder ungerader Länge und Zyklen gerader Länge.

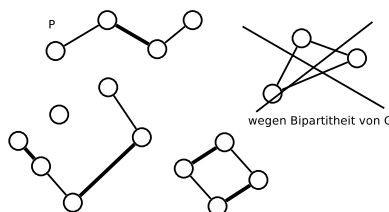


Abbildung 27.4.: Beispiel

**z.z.** Es gibt in  $\tilde{G}$  mindestens einen  $M$ -augmentierenden Pfad  $p$ , der mehr Kanten aus  $M^*$  als aus  $M$  besitzt. Dies gilt, weil ansonsten  $|M^*| \leq |M|$

q.e.d.



## 27.1.3. Pseudo-Code

```

1  M = ∅;
2  do {
3      P = findAugmPath(GM);
4      if (P == NULL) break;
5      M = M ⊕ P;
6  } while(true);

```

Wiederholung: Symmetrische Differenz

$$A \oplus B = A \setminus B \cup B \setminus A$$

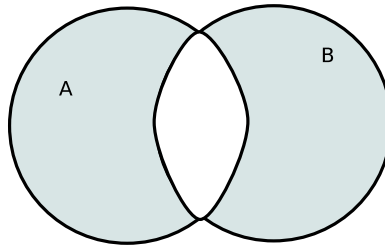


Abbildung 27.5.: Symmetrische Differenz

## 27.1.4. Laufzeit

$$\begin{aligned}
 &\mathcal{O}(\min(|V_1|, |V_2|) \cdot (|V| + |E|)) \\
 &= \mathcal{O}(|V| \cdot |E|)
 \end{aligned}$$

## 27.2. Hopcroft-Karp-Algorithmus

erzielt Laufzeit von  $\mathcal{O}(\sqrt{|V|} \cdot |E|)$

```

1  M = ∅;
2  do {
3      GL = buildLevelGraph(GM); //Knotendisjunkte M-augmentierende Pfade
4      P = findAugmPath(GL); //P = P1 ∪ P2 ∪ ... ∪ Pk
5      M = M ⊕ \mathcal{P};
6  } while(P ≠ ∅);

```

$G_L$  kann mittels BFS in Zeit  $\mathcal{O}(|V| + |E|)$  konstruiert werden. Zum Auffinden einer maximalen Menge von  $M$ -augmentierenden Pfaden in  $G_L$  verwenden wir DFS und entfernen jedes mal den gefunden Pfad  $P_i$  aus  $G_L$ . DFS sorgt dafür, dass  $P_i$  in Zeit  $\mathcal{O}(|P_i|)$  gefunden und gelöscht werden kann.

⇒  $\text{findAugmPaths}(G_L)$  hat nur Laufzeit  $\mathcal{O}(|E|)$

Ohne Beweis (siehe später Edmonds-Krap-Algorithmus) gilt:

Die Zeit der Schichten im Levelnetzgraph des Hopcraft-Karp-Algorithmus steigt mindestens um eins von Durchlauf zu Durchlauf.

$$|M^*| - |M| \leq \frac{|V|}{k} \quad \tilde{G} = (V_1 \cup V_2, M \oplus M^*)$$

$k$  Mindestlänge der  $M$ -augmentierenden Pfade

---

<sup>1</sup>Lemma von Berge

## 27. Das Heiratsproblem

**Ziel** Laufzeit von Hopcraft-Karp =  $\mathcal{O}(\sqrt{|V|} \cdot |E|)$

**1. Phase**  $k$  Schleifendurchläufe  $\Rightarrow$  Danach haben  $M$ -augment. Pfade mindestens die Länge  $k$

**2. Phase** Weitere  $\frac{|V|}{k}$  Schleifendurchläufe genügen, um zum Maximum-Matching  $M^*$  zu gelangen.

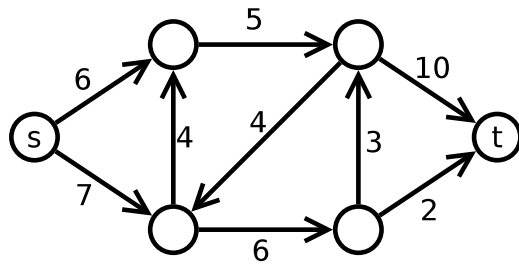
### 27.2.1. Gesamtzahl der Schleifendurchläufe

$$S(k) = k + \frac{|V|}{k} \quad S'(k) = 1 - \frac{|V|}{k^2} = 0 \Rightarrow k = \sqrt{|V|}$$

$\Rightarrow$  Zahl der Schleifendurchläufe  $\leq \sqrt{|V|}$

## 28. Algorithmen zur Berechnung maximaler Flüsse

### 28.1. Allgemein



Gesucht:

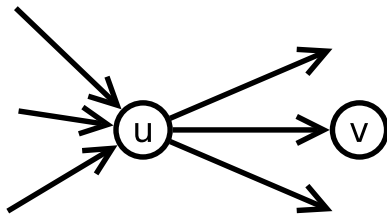
$$c^I : E \rightarrow \mathbb{R}_0^+$$

$$f : E \rightarrow \mathbb{R}_0^+$$

$$\forall (u, v) \in E : \text{Kapazitätsbedingung: } 0 \leq f(u, v) \leq c(u, v)$$

Abbildung 28.1.: ...

#### 28.1.1. Flusserhaltung



$$\forall u \in V \setminus \{s, t\}$$

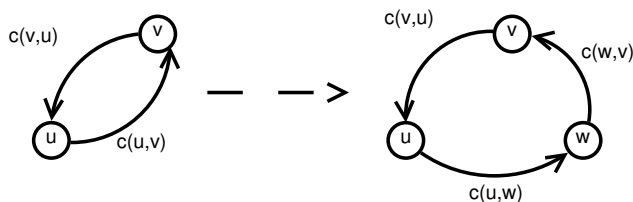
$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

Fluss aus  $u$  heraus      Fluss in  $u$  herein

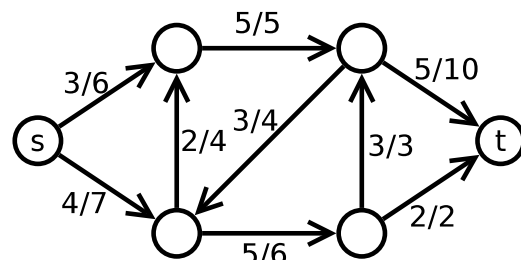
$$(u, v) \notin E \Rightarrow c(u, v) = 0 \Rightarrow f(u, v) = 0$$

$$(u, v) \in E \Rightarrow (v, u) \notin E \quad \text{Konventionen}$$

Abbildung 28.2.: Flusserhalt



(a) Kapazität



(b) Beispiel für einen Fluss

#### 28.1.2. Definition: Fluss

$|f| \hat{=}$  Fluss, der von  $s$  nach  $t$  transportiert wird.

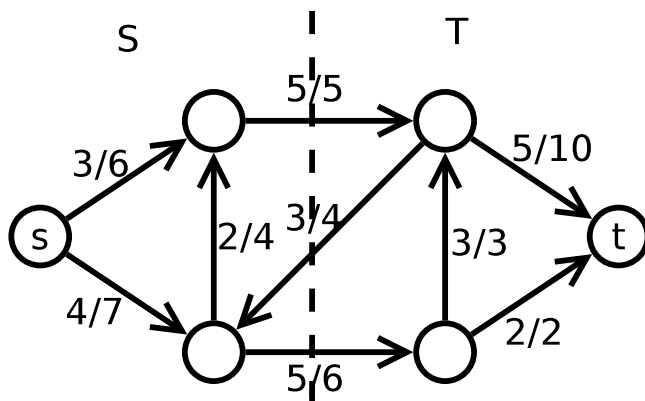
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v)$$

**Gesucht:** maximaler Fluss  $|f|$

<sup>1</sup>Kapazität

## 28. Algorithmen zur Berechnung maximaler Flüsse

### 28.1.3. Definition: Schnitt $(S, T)$



$$S \cup T = V \quad s \in S, t \in T$$

**Schnittkapazität**

$$\sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

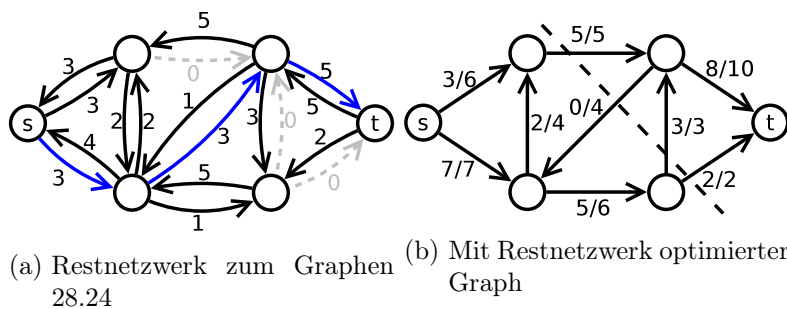
**Fluss über Schnitt**

$$\sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \leq c(S, T)$$

Abbildung 28.4.: Beispiel Schnitt

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \overbrace{\sum_{u \in S \setminus \{s\}} \left( \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right)}^{=0} = \sum_{u \in S} \left( \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) \\ &= \sum_{u \in S} \left( \sum_{v \in S} f(u, v) + \sum_{v \in T} f(u, v) - \sum_{v \in S} f(v, u) - \sum_{v \in T} f(v, u) \right) \\ &= \left( \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \right) + \overbrace{\left( \sum_{u \in S} \sum_{v \in S} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(v, u) \right)}^{=0} = f(S, T) \end{aligned}$$

### 28.1.4. Konstruktion des Restnetzwerk $G_f$



$$|f| = f(S, T) \leq c(S, T)$$

## 28.2. Ford-Fulkerson-Algorithmus

```

1  f = 0;
2  do {
3    p = flussverbessernder Pfad im Restnetzwerk  $G_f$ ;
4     $c_{min}$  = kleinste Restkapazität der Kanten von p;
5    erhöhe den Fluss f entlang von p um  $c_{min}$ 
6  } while (p  $\neq$  NULL);

```

### 28.3. Restnetzwerk $G_f = (V, E_f)$

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{für } (u, v) \in E \\ f(v, u) & \text{für } (v, u) \in E \\ 0 & \text{sonst} \end{cases}$$

### 28.4. Max-Flow-Min-Cut-Theorem

1.  $|f|$  ist maximal  $\Rightarrow$  Es gibt keinen  $f$ -verbessernden Pfad
2. Es gibt einen  $(S, T)$ -Schnitt, so dass  $c(S, T) = |f|$
3.  $c(S, T) = |f| \Rightarrow f$  ist maximal und  $c(S, T)$  minimal

#### 28.4.1. Beweis: 1. $\Rightarrow$ 2.

Gegeben sein ein maximaler Fluss  $f \Rightarrow t$  ist in  $G_f$  nicht von  $s$  erreichbar.

Sei  $S = \{v \in V \mid \exists p: s \rightsquigarrow v \text{ in } G_f\}, T = V \setminus S, t \in T$

#### 28.4.2. Idee

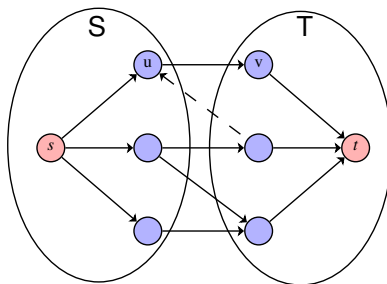


Abbildung 28.6.

**Zeige:** Alle Vorwärtskanten über den Schnitt  $S, T$  sind saturiert, d.h.  $f(u, v) = c(u, v)$  Alle Rückwärtskanten von  $T$  nach  $S$  tragen den Flusswert 0.

**Beweis** Sei  $(u, v) \in S \times T \cap E$

**Anmerkung**  $f(u, v) < c(u, v)$

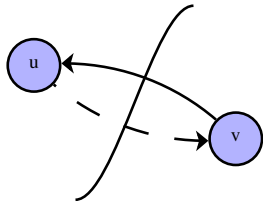
$\Rightarrow (u, v) \in E_f$  mit  $c_f(u, v) = c(u, v) - f(u, v) > 0$

$\Rightarrow v$  ist von  $s$  aus erreichbar:  $s \rightsquigarrow u \rightarrow v$

## 28. Algorithmen zur Berechnung maximaler Flüsse

Sei  $(v, u) \in T \times S$

**Anmerkung:**  $f(v, u) > 0$



$$\Rightarrow (u, v) \in E_f \text{ mit } c_f(u, v) = f(v, u) > 0$$

$\Rightarrow v$  ist auf dem Weg  $s \rightsquigarrow u \rightarrow v$  in  $G_f$  erreichbar.  $\nexists$

Abbildung 28.7.

$$|f| = \sum_{u \in S, v \in T} f(u, v) - \sum_{u \in S, v \in T} f(v, u) = \sum_{u \in S, v \in T} c(u, v) - 0 = c(S, T)$$

1.  $\Rightarrow$  2.  
q.e.d.

$$|f| = c(S, T) \Rightarrow f \text{ maximal und } c(S, T) \text{ minimal.}$$

**Offene Frage** Unter welchen Bedingungen terminiert der Ford-Fulkerson-Algorithmus?

**Anmerkung**  $c : E \rightarrow \mathbb{N}$

$$\Rightarrow |f^*|^{\Pi} \in \mathbb{N}, c_{\min}(p) \in \mathbb{N} \text{ für } p \text{ flussverbessernder Pfad } \geq 1$$

$\Rightarrow$  Es genügen  $|f^*|$  viele Iterationen zur Flussverbesserung

$$\Rightarrow \text{Laufzeit}(|f^*| \cdot |E|)$$

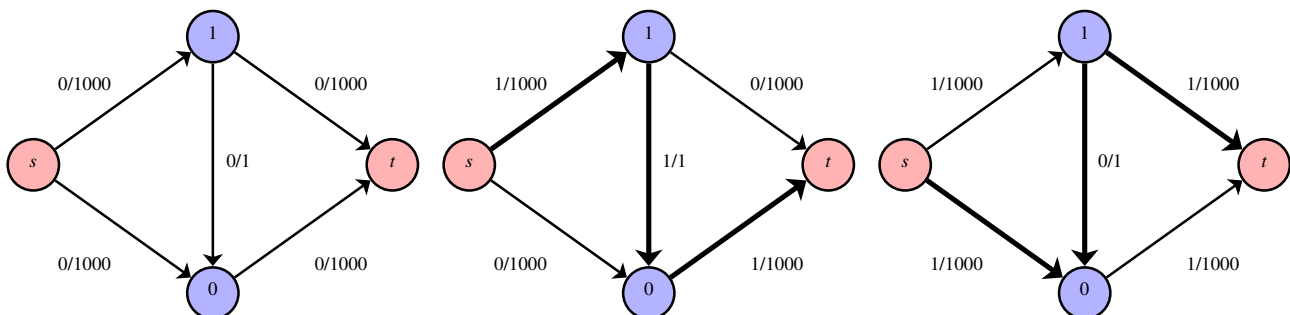
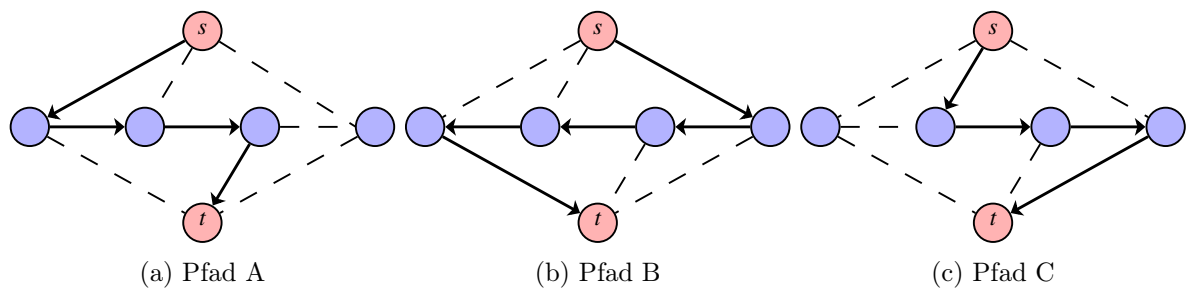
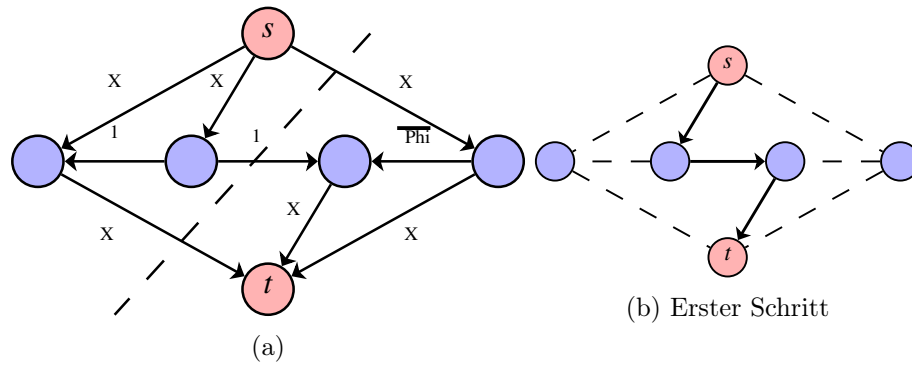


Abbildung 28.8.

<sup>II</sup>maximaler Fluss

### 28.4.3. Zwick

$$\bar{\phi} = \frac{\sqrt{5}-1}{2} \quad X > 1000 \quad |f^*| = 2X + 1$$



Invariante  $\bar{\phi} = 0,613\dots$

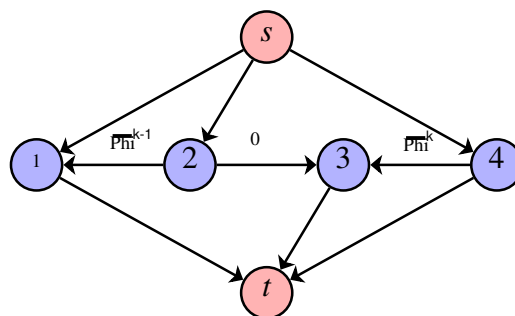


Abbildung 28.11.

$$1 - \bar{\phi} = \bar{\phi}^2$$

$$\bar{\phi}^{k-1} - \bar{\phi}^k = \bar{\phi}^{k+1}$$

## 28. Algorithmen zur Berechnung maximaler Flüsse

**1. Schritt** Schicke  $\bar{\phi}^k$  Flusseinheiten entlang Pfad B

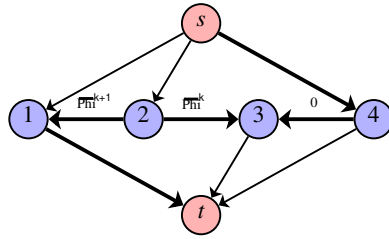


Abbildung 28.12.: Schritt 1

**2. Schritt** Schicke  $\bar{\phi}^k$  entlang C

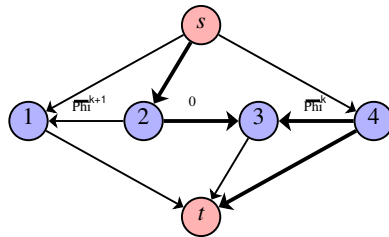


Abbildung 28.13.: Schritt 2

**3. Schritt** Schicke  $\bar{\phi}^{k+1}$  entlang B

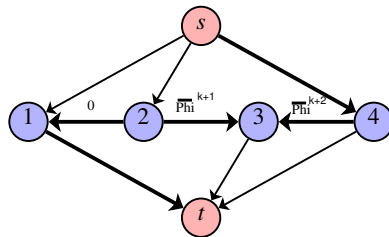


Abbildung 28.14.: Schritt 3

**2. Schritt** Schicke  $\bar{\phi}^{k+1}$  entlang A

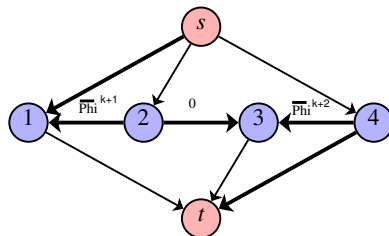


Abbildung 28.15.: Schritt 4

Wir iterieren diese 4 Schritte unendlich oft

$$|f| = 1 + 2 \cdot \sum_{k=1}^{\infty} \bar{\phi}^k + 2 \cdot \sum_{k=1}^{\infty} \bar{\phi}^{k+1} < 7 \leq 2X + 1 \quad \sum_{k=0}^{\infty} \bar{\phi}^k = \frac{1}{1 - \bar{\phi}}$$



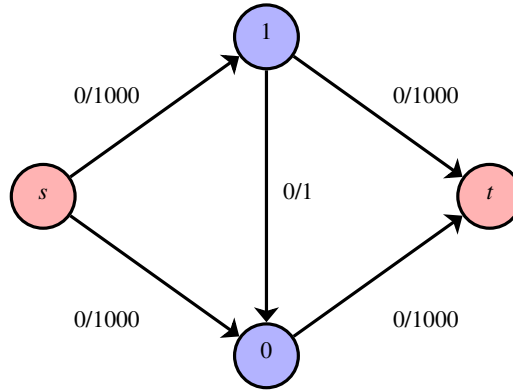


Abbildung 28.16.: Wiederholung

## 28.5. Edmonds-Karp Algorithmus

$$G = (V, E) \quad , c : R \rightarrow \mathbb{R}_0^+, G_F = (V, E_f), G_f^L = (V, E_f^L)$$

```

1  f = 0;
2  while(∃ p ~ t ∈ G_f^L = (V, E_f^L)) {
3    sei c_min(p) die kleinste Restkapazität auf p
4    f(u, v) = { f(u, v) + c_min(p) falls (u, v) ∈ p
                { f(u, v) - c_min(p) falls (v, u) ∈ p
5  }
```

$\delta_f(s, v)$  die kleinste Zahl von Kanten, die in  $G_f^L$  benötigt werden, um von  $s$  nach  $v$  zu gelangen.

### 28.5.1. Lemma:

Im Verlauf des Edmonds-Karp Algorithmus gilt:

$$\delta_{f'}(s, v) \geq \delta_f(s, v)$$

wobei der Fluss  $f'$  durch eine Flussverbesserung aus  $f$  hervorgegangen ist.

### 28.5.2. Beweis durch Widerspruch

**Annamhe**

$$\exists v \in V : \delta_{f'}(s, v) < \delta_f(s, v) \quad (**)$$

sei  $v$  so gewählt, dass  $\delta_{f'}(s, v)$  minimal.

Sei  $s \rightsquigarrow u \rightarrow v$  ein kürzester Weg in  $G_{f'}^L$ .

$$\delta_f(s, u) \leq \delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 \quad (*)$$

**Behauptung**

$$(u, v) \notin E_f^L$$

**Beweis** durch Widerspruch

## 28. Algorithmen zur Berechnung maximaler Flüsse

**Annahme**  $(u, v) \in E_f^L$

$$\delta_f(s, v) \leq \text{III} \delta_f(s, u) + 1 \leq \text{IV} \delta_{f'}(s, v) \not\leq \text{zu } (**)$$

$$\Rightarrow (u, v) \notin E_f^L \text{ aber } (u, v) \in E_{f'}^L$$

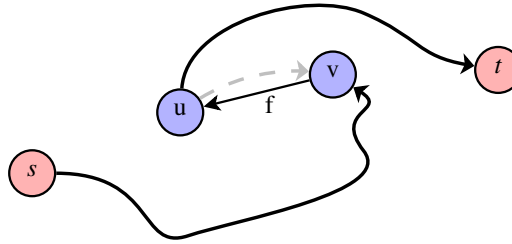


Abbildung 28.17.

d.h. Bei der Flussverbesserung von  $f$  zu  $f'$  wurde die Kante  $(v, u)$  benutzt in  $G_f^L$ .

$$\begin{aligned} \delta_f(s, u) &= \delta_f(s, v) + 1 \\ &\stackrel{(**)}{>} \delta_{f'}(s, v) + 1 \\ &\stackrel{(*)}{\geq} \delta_f(s, u) + 2 \not\leq \end{aligned}$$

q.e.d.

### 28.5.3. Lemma

Eine kante  $(u, v)$  kann un den Level-Rest-Netzwerken höchstens  $\frac{|V|}{2}$  mal saturiert werden und damit temporär aus dem jeweiligen Rest-Netzwerk verschwinden

### 28.5.4. Beweis

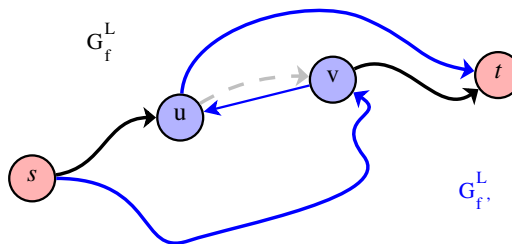


Abbildung 28.18.

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) + 1 \\ \delta_{f'}(s, v) &= \delta_{f'}(s, u) - 1 \end{aligned}$$

<sup>III</sup>Dreiecksungleichung

<sup>IV</sup>wegen (\*)

$$\delta_f(s, u) = \delta_f(s, v) - 1 \leq \delta_{f'}(s, v) - 1 = \delta_{f'}(s, u) - 2$$

$$\Rightarrow \delta_{f'}(s, u) \geq \delta_f(s, u) + 2$$

q.e.d.

### 28.5.5. Laufzeitanalyse von Edmonds-Karp Algorithmus

Bei jeder Flussverbesserung wird mindestens eine Kante saturiert. Jede einzelne Kante kann aber höchstens  $\frac{|V|}{2}$  mal saturiert werden.

$\Rightarrow$  Es gibt höchstens  $\mathcal{O}(|E| \cdot |V|)$  viele Flussverbesserungen, Jede Flussverbesserung kann in  $\mathcal{O}(|E|)$  ausgeführt werden.

$\Rightarrow$  Gesamtlaufzeit:  $\mathcal{O}(|E|^2 \cdot |V|)$

## 28.6. Algorithmus von Dinic

### 28.6.1. Sperrfluss (blocking flow)

$$G_f^L = (V, E_f^L)$$

Wir konstruieren einen Sperrfluss  $g$  für einen Graphen  $H$ , indem wir wiederholt entlang von  $(s, t)$ -Pfaden Fluss von  $s$  nach  $t$  transportieren.

Bevor wir diesen Prozess wiederholen, löschen wir saturierte Kanten aus  $H$ . Läuft man bei der Wegesuche in eine Sackgasse, so muss diese aus  $H$  entfernt werden, damit man zu einem späteren Zeitpunkt nicht wieder in diese Sackgasse gerät.

**Ziel** Algorithmus zur Sperrflussberechnung in Zeit  $\mathcal{O}(|V| \cdot |E|)$

## 28.7. Definition: Sperrfluss

$$G = (V, E) \quad g : E \rightarrow \mathbb{R}^+ \text{ ein Fluss}$$

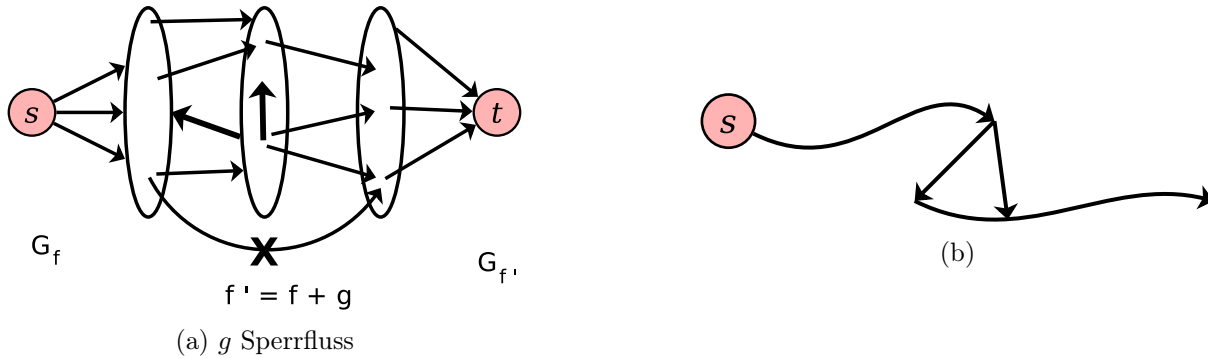
Auf jedem  $s - t$ -Pfad gibt es eine saturierte Kante  $(u, v) \in E$ , d.h.  $g(u, v) = c(u, v)$

**Achtung** Wir betrachten hierbei nicht die Kantenmenge  $E_f$  des Restnetzwerkes.

Es gilt:  $\delta_{f'}(s, v) \geq \delta_f(s, v)$ , wobei  $f'$  aus  $f$  durch eine Flusserhöhung hervorgegangen ist.

Für den Dinic-Algorithmus gilt darüber hinaus, dass  $\delta_{f'}(s, t) \geq \delta_f(s, t) + 1$  wobei  $f'$  aus  $f$  durch eine Flussverbesserung mittels eines Sperrflusses  $g$  hervorgegangen ist.

## 28. Algorithmen zur Berechnung maximaler Flüsse



### Beweisidee

**Konsequenz** Im Dinic-Algorithmus genügt es,  $|V|$  Sperrfluss-Berechnungen durchzuführen.

**Ziel** Sperrfluss in Zeit  $\mathcal{O}(|V| \cdot |E|)$  berechnen.

Gesamtlaufzeit Dinic

$$\mathcal{O}(|V|^2 \cdot |E|)$$

vs. Edmonds-Karp

$$\mathcal{O}(|V| \cdot |E|^2)$$

(Sleator & Tarjan :

$$\mathcal{O}(|V| \cdot |E| \cdot \log |V|))$$

### 28.7.1. Pseudo-Code

```

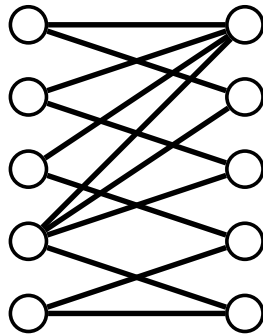
1   $H = G_f^L$ ;
2  stack  $P$ ;
3   $P.push(s)$ ;
4   $g = 0$ ;
5  while(true) {
6       $u = P.top()$ ;
7      if ( $\exists (u, v) \in H$ ) {
8           $P.push(v)$ ;
9          if ( $v \neq t$ ) continue;
10         //  $v = t$ , d.h. flussverbessernder  $s$ - $t$ -Pfad gefunden
11          $c_{min} = \min_{(u,v) \in P} \{c_f(u, v) - g(u, v)\}$ 
12         forall  $(u, v) \in P$  {
13              $g(u, v) = g(u, v) + c_{min}$ ;
14             if ( $g(u, v) == c_f(u, v)$ )
15                 lösche  $(u, v) \in H$ ;
16         }
17          $P.clear()$ ;
18          $P.push(s)$ ;
19         continue;
20     } //end if
21     //Sackgasse
22     lösche alle zu  $u$  inzidenten Kanten aus  $H$ ;
23      $P.pop()$ ;
24     if ( $u == s$ )
25         break;
26 } //end while

```

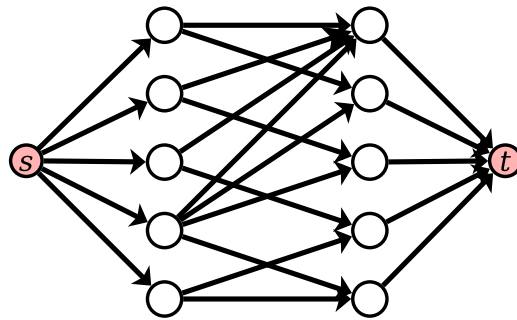
### 28.7.2. Begründung zur Laufzeit

Jeder  $s - t$ -Pfad wird in Zeit  $\mathcal{O}(|V|)$  gefunden. Danach wird mindestens eine Kante aus  $H$  entfernt, weil sie saturiert wird. Jede Kante kann höchstens einmal als Sackgasse betreten werden, weil sie anschließend gelöscht wird.

## 28.8. Maximum Matching als Flussproblem



(a) Matchingproblem



(b) Flussproblem (Kantengewichte sind 1)

$$E \supseteq M^* = \{(u, v) \in E \cap V_1 \times V_2 \mid f(u, v) = 1\}$$

$$\begin{aligned} \text{F.F.-Laufzeit} & \quad \mathcal{O}(|f^*| \cdot |E|) \\ & = \mathcal{O}(|V| \cdot |E|) \end{aligned}$$

$|f|$  ist ganzzahlig = Kardinalität des maximum Matchings  $|M^*|$

### 28.8.1. Flussnetzwerke mit Einheitskapazität (unit capacity network flow)

$$G = (V, E) \quad c : E \rightarrow \{0, 1\}$$

Sperrfluss-Berechnung läuft in  $\mathcal{O}(|E|)$ , weil immer alle Kanten auf den flussverbessernden  $s - t$ -Pfad gelöscht werden können.

**Dinic:**  $\mathcal{O}(|V| \cdot |E|)$

#### Satz

Mit Hilfe des Dinic-Algorithmus lässt sich ein Flussproblem mit Einheitskapazität in Zeit

$$\mathcal{O}\left(\min\left(|E|^{\frac{1}{2}}, |V|^{\frac{2}{3}}\right)\right)$$

lösen.

#### Beweis

**1. Fall** Zeige, dass  $2 \cdot \sqrt{|E|}$  viele Sperrfluss-Berechnungen genügen.

**1. Phase** Zuerst  $\sqrt{|E|}$  Sperrfluss-Berechnungen.  $\Rightarrow$  Fluss  $f$

$$\Rightarrow \delta_f(s, t) \geq \sqrt{|E|}$$

$\Rightarrow$  Schnitt zwischen zwei Levels  $L_i$  und  $L_{i+1}$  mit weniger als  $\sqrt{|E|}$  Kanten.  
 $|f^*| \leq |f| + \sqrt{|E|}$ , weil über diesen Schnitt zusätzlich zu  $f$  noch ein Fluss der Größe  $\sqrt{|E|}$  möglich ist.

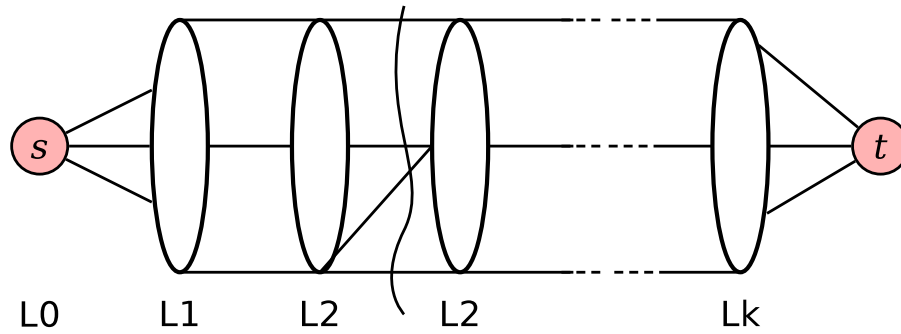


Abbildung 28.21.:  $k \geq \sqrt{|E|}$

$\Rightarrow$  **2. Phase:** Um von  $f$  zu  $f^*$  zu kommen, reichen weitere  $\sqrt{|E|}$  viele Sperrfluss-Berechnungen aus.

## 2. Fall

**1. Phase** Führe  $2 \cdot |V|^{\frac{2}{3}}$  Sperrflussberechnungen durch.  $\Rightarrow \delta_f(s, t) > 2 \cdot |V|^{\frac{2}{3}}$

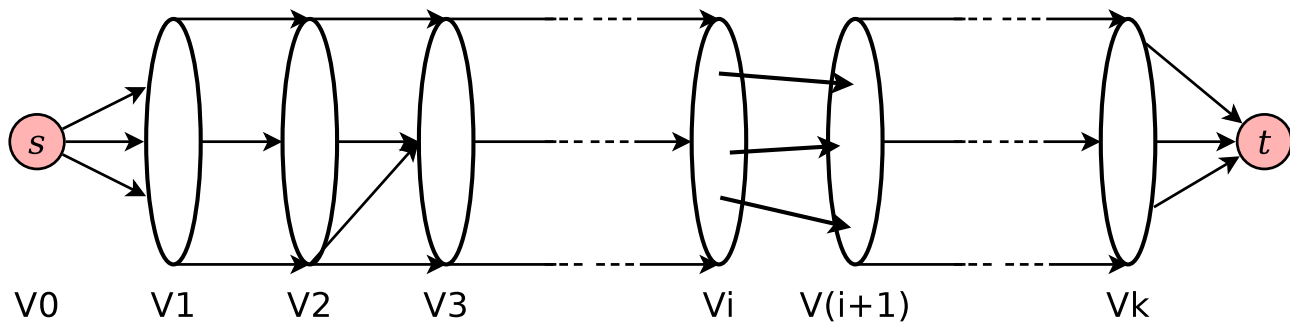


Abbildung 28.22.: Schaubild

## Behauptung

$$k = 2 \cdot |V|^{\frac{2}{3}}$$

$$\exists 0 < i < k : |V_i| \leq |V|^{\frac{1}{3}} \\ \text{und } |V_{i+1}| \leq |V|^{\frac{1}{3}}$$

Falls Behauptung gilt  $\Rightarrow c_f(V_i, V_{i+1}) \leq \# \text{Kanten über diesen Schnitt} \leq |V_i| \cdot |V_{i+1}| \leq |V|^{\frac{2}{3}}$ . Maximaler Fluss  $|f^*|$  ist vom aktuellen Fluss  $f$ , der sich nach der 1. Phase eingestellt hat höchstens noch  $|V|^{\frac{2}{3}}$  entfernt. Deshalb genügen in der 2. Phase noch  $|V|^{\frac{2}{3}}$  Sperrflussberechnungen um  $|f^*|$  zu erreichen.

**Beweis der Behauptung**  $< |V|^{\frac{2}{3}}$  Schichten haben  $> |V|^{\frac{1}{3}}$  Knoten. Andernfalls gäbe es mehr als  $|V|^{\frac{2}{3}} \cdot |V|^{\frac{1}{3}} = |V|$  viele Knoten.

**Beweisidee** Wir färben die Schichten weiß, welche weniger als  $|V|^{\frac{1}{3}}$  Knoten haben und den Rest schwarz, da es mehr weiße als schwarze gibt, müssen 2 weiße aufeinander folgen.:

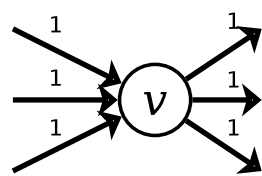
$$\underbrace{w \ s \ w \ s \ w \ s \ \dots \ w \ s}_k$$

$> |V|^{\frac{2}{3}}$  Schichten haben  $\leq |V|^{\frac{1}{3}}$  Knoten  
 $\Rightarrow$  es muss  $i$  geben, mit  $|V_i| \cdot |V_{i+1}| \leq |V|^{\frac{1}{3}}$

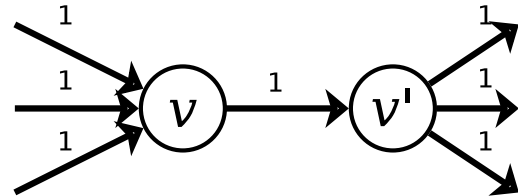
q.e.d.

### 28.8.2. Finden knotendisjunkter Wege

**Idee** Man forme den Graphen wie folgt um:



(a) Original Knoten



(b) Umgeformter Knoten

### 28.8.3. Ergänzung zum Paper<sup>v</sup>

$$\text{pot}_f(v) = \min \left( \sum_{(v,u) \in E} c(v,u) - f(v,u), \sum_{(u,v) \in E} c(u,v) - f(u,v) \right)$$

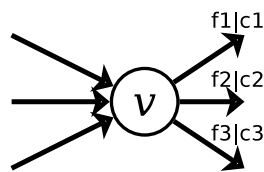


Abbildung 28.24.

<sup>v</sup>Information Processing Letters Volume 7, number 6

# 29. Dynamische Programmierung

## 29.1. Matrizen Multiplikation

$$\mathbb{R}^{n \times m} \ni C = AB \quad A \in \mathbb{R}^{n \times k}, \quad B \in \mathbb{R}^{k \times m}$$

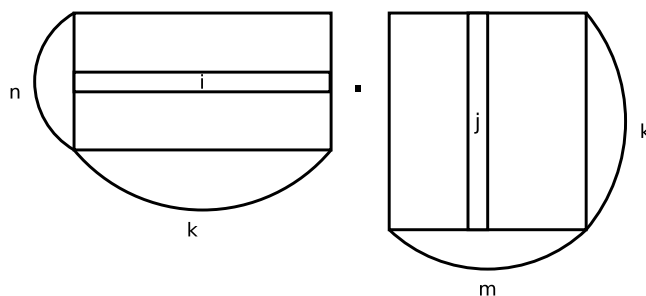


Abbildung 29.1.: Matrix-Multiplikation

$$c_{ij} = \sum_{l=1}^k a_{il} \cdot b_{lj}$$

**Laufzeit**  $\mathcal{O}(n \cdot k \cdot m)$

$$A_1 \cdot A_2 \cdot A_3 \cdot \dots \cdot A_n \quad A_i \in \mathbb{R}^{p_{i-1} \times p_i}$$

**Beispiel**

$$x^T \cdot a \cdot b^T \cdot y, \quad a, b, x, y \in \mathbb{R}^n \quad \text{Spaltenvektoren}$$

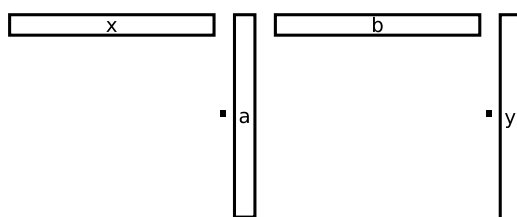


Abbildung 29.2.: Vektor-Multiplikation

$$x^T \cdot \left( (a \cdot b^T) \cdot y \right)$$

$$\mathbb{R}^{n \times n} \ni (a \cdot b^T) \text{ kostet } \mathcal{O}(n^2) \quad (a \cdot b^T) \cdot y \text{ kostet } \mathcal{O}(n^2) \quad x^T \cdot (\dots) \text{ kostet } \mathcal{O}(n)$$

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

$$(x^T \cdot a) \cdot (b^T \cdot y)$$

$$\mathcal{O}(n) + \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$$



**Ansatz**

$$A_1 \cdot A_2 \cdot A_3 \cdot \dots \cdot A_n \quad A_i \in \mathbb{R}^{p_{i-1} \times p_i}$$

Es gibt  $C_{n-1}$  Klammerungen  $C_n = \frac{1}{n+1} \binom{2n}{n}$  exponentiell in  $n \in \Omega(2^n)$

**29.1.1. Bottom-Up Ansatz**

$$A_{ij} = (A_i \cdot A_{i+1} \cdot \dots \cdot A_j) \in \mathbb{R}^{p_{i-1} \times p_j}$$

$M_{ij}$  = optimale Kosten zur Auswertung von  $A_{ij}$

**Gesucht**  $M_{1n}$

$$M_{ij} = \min_{i \leq k < j} \{M_{ik} + M_{(k+1)j} + p_{i-1} \cdot p_k \cdot p_j\}$$

$$M_{ii} = 0$$

**Nebenrechnung**

$$A_{ij} = (A_i \cdot A_{i+1} \cdot \dots \cdot A_k) \cdot (A_{k+2} \cdot \dots \cdot A_j) = A_{ik} \cdot A_{(k+1)j} \quad A_{ik} \in \mathbb{R}^{p_{i-1} \times p_k}, \quad A_{(k+1)j} \in \mathbb{R}^{p_k \times p_j}$$

$$A_{ii} = A_i$$

**29.1.2. Programm**

```

1  for (i = 1; i ≤ n; i++) {
2      M[i][i] = 0;
3  }
4  for (l = 2; l ≤ n; l++) {
5      for (i = 1; i ≤ n - l + 1; i++) {
6          j = i + l - 1;
7          M[i][j] = ∞;
8          for (k = i; k < j; k++) {
9              q = M[i][k] + M[k+1][j] + p[i-1] · p[k] · p[j];
10             if (q < M[i][j]) {
11                 M[i][j] = q;
12                 S[i][j] = k;
13             }
14         }
15     }
16 }
```

$$i \begin{pmatrix} & 1 & 2 & 3 & \dots & j & \dots & n \\ \square & & & & & & & \\ & \square & \square & \square & \square & & & \\ & & \square & & \square & & & \\ & & & \square & \square & & & \\ & & & & \square & & & \\ & & & & & \square & & \\ & & & & & & \square & \\ & & & & & & & \square \end{pmatrix} = M$$

**Laufzeit**  $\mathcal{O}(n^3)$

# Abbildungsverzeichnis

1.1. Bubblesort . . . . .	2
2.1. Heapsort (Ausgangssituation) . . . . .	4
2.2. Indices . . . . .	4
2.3. Heap-Eigenschaft . . . . .	4
8.1. Fibonacci-Zahlen . . . . .	19
9.1. Quicksort . . . . .	21
9.2. Quicksort . . . . .	22
9.3. . . . .	23
10.1. Aufruf von Partition . . . . .	25
13.1. Median der Mediane . . . . .	28
14.1. Entscheidungsbaum am Beispiel Bubblesort . . . . .	29
14.2. Binärerbaum . . . . .	30
14.3. Induktions-Ansatz . . . . .	30
16.1. Knotenorientierte Speicherung . . . . .	33
16.2. Binärer Suchbaum . . . . .	34
17.1. AVL-Baum . . . . .	35
17.2. . . . .	35
17.3. . . . .	37
17.4. . . . .	37
17.5. Rotation . . . . .	38
18.2. Einfügen und Löschen in einem 2-5 Baum . . . . .	40
20.1. Universum und Hashtabelle der Größe $m$ . . . . .	44
20.2. Perfekte Hashtabelle . . . . .	48
20.3. Perfektes Hashing . . . . .	49
21.1. Gerichteter Graph . . . . .	51
21.2. Würfel . . . . .	51
21.3. Placeholder . . . . .	51
21.4. Beispiel . . . . .	52
21.5. Adjazenzliste . . . . .	53
21.6. indeg und outdeg . . . . .	53
22.1. Grafik zum Beispielcode . . . . .	54
22.2. Beispiel . . . . .	55
22.3. . . . .	56
22.4. . . . .	58

23.1. . . . .	59
23.2. . . . .	59
23.3. . . . .	59
23.4. . . . .	60
23.5. Skizze . . . . .	60
23.6. Ein verbotener, negativer Zyklus . . . . .	61
24.1. Grafik . . . . .	63
24.2. Beweis der Invariante . . . . .	65
24.3. Naiver Ansatz, kürzester Weg wird zerstört . . . . .	65
24.4. Die blau markierten Kanten haben die Länge 0 . . . . .	66
25.2. . . . .	68
25.3. Reihenfolge grün→rot→blau→braun . . . . .	68
25.5. . . . .	71
26.1. Binomial-Bäume . . . . .	72
26.2. Aufbau . . . . .	72
26.3. DeleteMin-Operation . . . . .	73
26.4. decreaseKey-Operation . . . . .	73
26.5. Konsolidierungs-Operation . . . . .	74
26.6. Schaubild zum Beweis . . . . .	74
27.1. Ausgangsproblem . . . . .	77
27.3. Alternierender Pfad . . . . .	78
27.4. Beispiel . . . . .	78
27.5. Symmetrische Differenz . . . . .	79
28.1. ... . . . .	81
28.2. Flusserhalt . . . . .	81
28.4. Beispiel Schnitt . . . . .	82
28.6. . . . .	83
28.7. . . . .	84
28.8. . . . .	84
28.11.. . . . .	85
28.12Schritt 1 . . . . .	86
28.13Schritt 2 . . . . .	86
28.14Schritt 3 . . . . .	86
28.15Schritt 4 . . . . .	86
28.16Wiederholung . . . . .	87
28.17.. . . . .	88
28.18.. . . . .	88
28.21 $k \geq \sqrt{ E }$ . . . . .	92
28.22Schaubild . . . . .	92
28.24.. . . . .	93
29.1. Matrix-Multiplikation . . . . .	94
29.2. Vektor-Multiplikation . . . . .	94